HERMES version 2.0
programmed by Gerry Stahl, PhD
Spring 1994

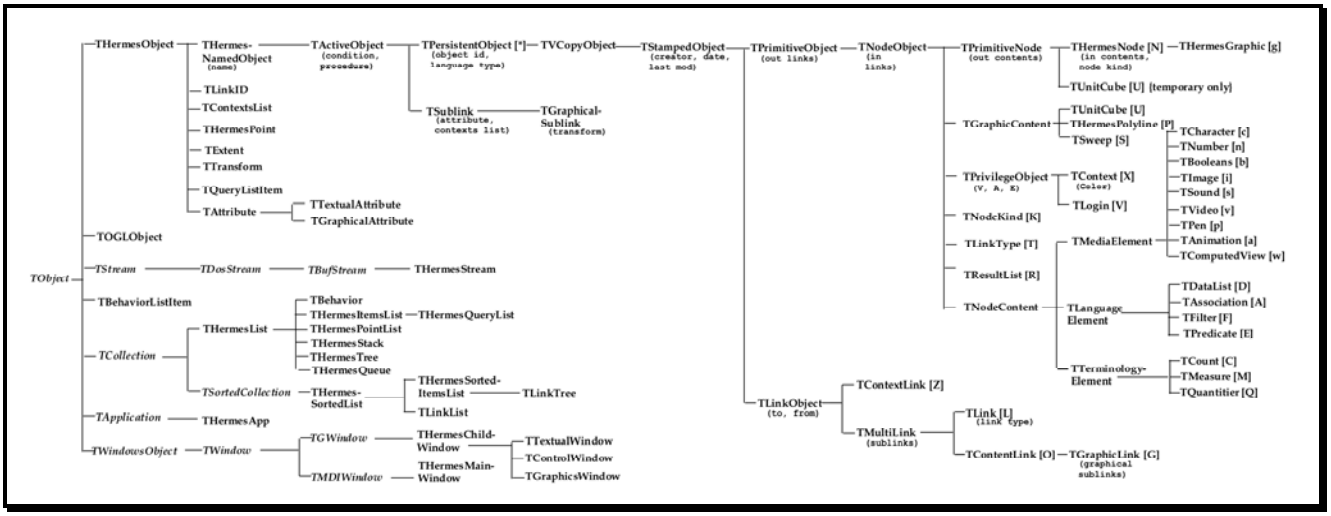
Source Code and Documentation
Table of Contents

1. Hermes.pas main program

```
 {*****************************************
 *
 *      Hermes.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 ******************************************}

{ This is the main program of the Hermes system.
  The Hermes system was proposed in:
      Stahl, Gerry, "Interpretation in Design:
      The Problem of Tacit and Explicit Understanding
      in Computer Support of Cooperative Design"
      Ph.D. Dissertation, Department of Computer Science,
      University of Colorado at Boulder, December 1993.
      The dissertation is copyright (c) 1993 by Gerry Stahl.
      It is available as Tech Report CU-CS-688-93
      or from UMI Dissertation Services. }

program Hermes;

uses HerWorld;

begin
       HermesApp.Init('Hermes Design Environment');
       HermesApp.Run;
       HermesApp.Done;
end.   {Hermes.pas}
```

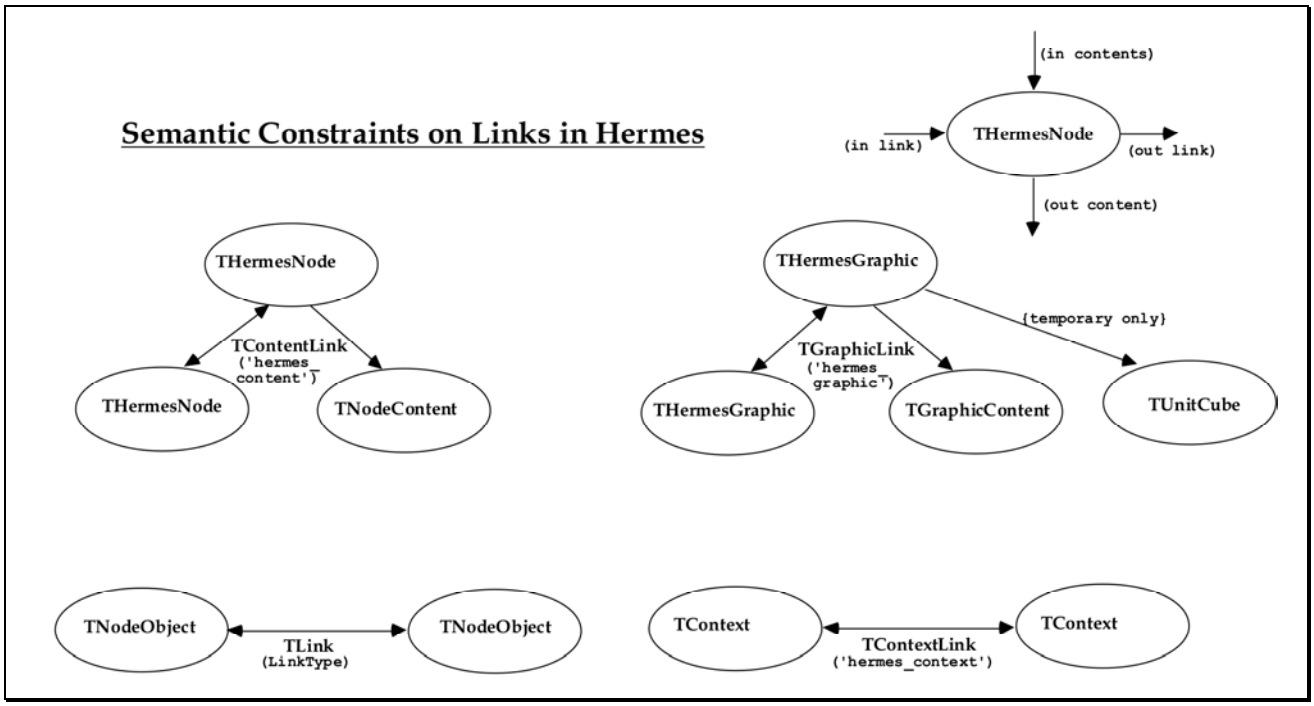
* Hermes object hierarchy

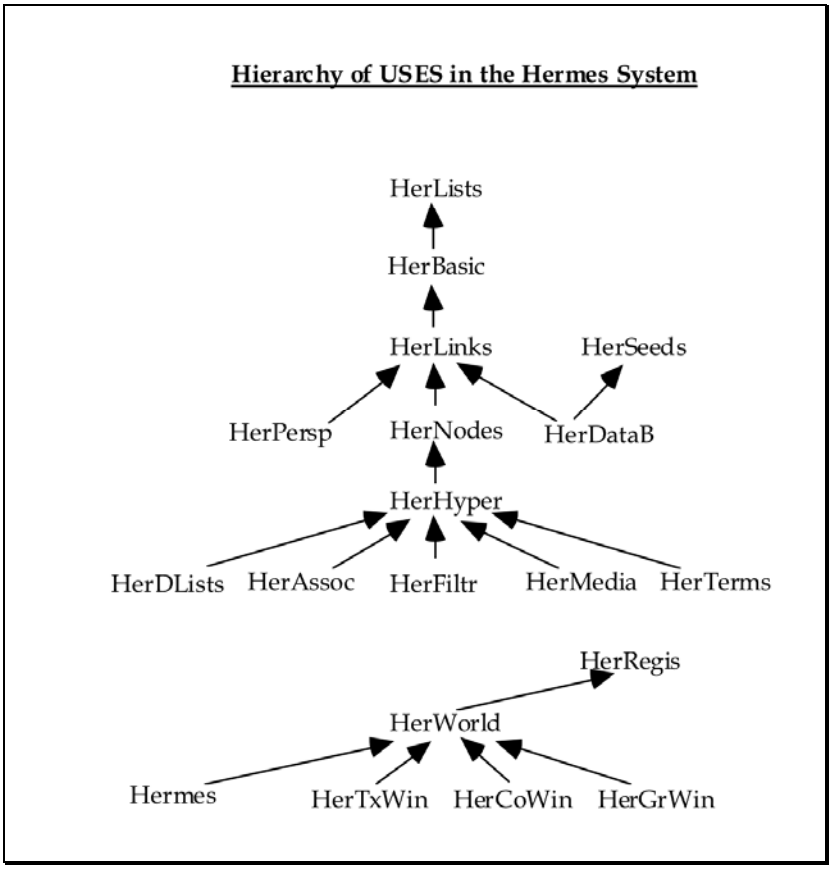
Hermes Substrate Object Hierarchy



Data items for some Hermes objects are in ( ) and language type codes are in [ ]. Objects from Borland
Pascal *are* in *italics*.

The chart above shows the hierarchy of objects defined in the Hermes system. The current design calls for 225 objects: 75 shown here, 12 windows, 76 language, 40 media, 22 terminology elements. This does not include other parts of Phidias: 3-D graphics, additional interface, the OGL system, or the Pd system.






```
To do:

Store Id of NodeKind and LinkType instead of names
in nodes and links to save disk space;
same with "hermes_..." and DBAuthor
and other constant strings stored in common objects like LinkTrees.
```

* Hermes language syntax

Syntax of the **HERMES** Language

All Capitalized Terms are non-terminals. Underlined terms are literal terminals. [Terms in square brackets] are optional. (Words in parentheses) are comments. Other terms describe terminals. The start symbol is DataList.

----------*language elements*----------

DataList ::= display SimpleDataList | ComputedDataList

SimpleDataList ::= a node name | id: an object id | Character | Number | Boolean | NodeKind | LanguageType | items | that (last subject) | this (expression) | those items | contents of ResultList | a DataList name

ComputedDataList ::= DataList Combination DataList | Association of DataList | DataList with their Association | DataList that Filter | Graphic [immediately] in Graphic | DataList in context Context | either DataList or DataList | if Boolean then DataList [, else DataList] | DataList, sorted | DataList, without duplicates

Association ::= SimpleAssociation | InputAssociation | ComputedAssociation | Predicate

SimpleAssociation ::= LinkType | inverse LinkType | name | id | creation date | creator | last modification date | contexts | all associations | [immediate] parts | inverse parts | Dimension | Distance in Units from Graphic [in Graphic] | an Association name

InputAssociation ::= LinkType | InputAssociation with their InputAssociation | InputAssociation and InputAssociation | an input association name

ComputedAssociation ::= Association of Association | Association with their Association | Association that Filter | either Association or Association | if Boolean then Association [, else Association] | Association Combination Association | the Number th Association | Association, sorted | Association, without duplicates

Predicate ::= Association

Filter ::= SimpleFilter | CharacterFilter | NumberFilter | BooleanFilter | ContextFilter | GraphicFilter | ComputedFilter

SimpleFilter ::= equal DataList | named Character | included in DataList | include DataList | of kind NodeKind | of type LanguageType | a Filter name

CharacterFilter ::= include Character

NumberFilter ::= Counter

BooleanFilter ::= true

ContextFilter ::= view [Counter] DataList | inherit from Context | are inherited by Context

GraphicFilter ::= [immediately] contain Graphic | [immediately] contained in Graphic | Measure [Quantifier] Graphic [in Graphic] | have Attribute is Value | have Attribute is Number

ComputedFilter ::= have Counter Association [with those items] | have Quantifier Association that Filter [with those items] | if Boolean then Filter [else Filter] | Filter Connective Filter | are Filter | are not Filter | do not Filter

----------media *elements*----------

Character ::= SimpleCharacter | ComputedCharacter

SimpleCharacter ::= character string | a Character name

ComputedCharacter ::= substring of Character from Number for Number | Character append Character

Number ::= SimpleNumber | ComputedNumber

SimpleNumber ::= real number | a Number name

ComputedNumber ::= count of DataList | minimum DataList | maximum DataList | total of DataList | product of DataList | Number + Number | Number - Number | - Number | Number x Number | Number / Number | Distance in Units between Graphic and Graphic [in Graphic]

Boolean ::= SimpleBoolean | ComputedBoolean

SimpleBoolean ::= true | false | a Boolean name

ComputedBoolean ::= there are Counter DataList | Quantifier DataList Filter | not Boolean | Boolean Connective Boolean | Graphic Measure [Quantifier] DataList [in Graphic]

Graphic ::= SimpleGraphic | ComputedGraphic

SimpleGraphic ::= polyline | a Graphic name

ComputedGraphic ::= DataList (of type graphic)

Image ::= bitmap image | an Image name

Pen ::= pen sketch | a Pen name

Sound ::= sound segment | a Sound name

Video ::= video segment | a Video name

Animation ::= animation segment | an Animation name

ComputedView ::= DataList arranged in a window | a ComputedView name

----------*network elements*----------

NodeKind ::= a NodeKind name

LinkType ::= a LinkType name

Context ::= a Context name
ResultList ::= name of an evaluated DataList

----------*namable terminology elements*----------

Counter ::= (at least one) | more than Number | less than Number | exact Number | not Counter | Counter Connective Counter | a Counter name

Quantifier ::= no | any | all | most | the (only one) | Counter | a Quantifier name

Measure ::= Distance is Counter Units | not Measure | Measure Connective Measure | a Measure name

----------*simple terminology elements*----------

Connective ::= and (logical) | or (logical)

Combination ::= and (unique sorted union) | and also (intersection) | but not (difference) | or (append) | with (and, indented)

Distance ::= central distance | closest distance | x distance | y distance | z distance

Units ::= inches | feet | cm | meters

Dimension ::= length | area | volume | x width | y height | z depth

Attribute ::= font | color | pen width | brush style | brush width | . . . .

Value ::= roman | helvetica | red | blue | striped | plaid | . . . .

LanguageType ::= data lists | associations | filters | characters | numbers | booleans | graphics | images | pens | sounds | videos | animations | computed views | node kinds | link types | result lists | contexts | counters | quantifiers | measures

ObjectTypes

language elements

D	DataList	language expression
A	Association	language expression
F	Filter	language expression
E	Predicate	Association displayed as Predicate

media elements

c	Character	PChar character string
n	Number	real number
b	Boolean	true/false language expression
g	HermesGraphic	vector graphic (composite node hierarchy)
i	Image	bitmap raster image
p	Pen	pen-based sketch
s	Sound	sound segment
v	Video	video segment
a	Animation	animation segment
w	ComputedView	stored composite display (OGL picture)

terminology elements

C	Counter	numeric relation
Q	Quantifier	quantification relation
M	Measure	graphic distance relation

hypermedia elements

K	NodeKind	defined type for nodes
T	LinkType	defined type for links
N	HermesNode	node in the hypermedia database
R	ResultList	stored list of nodes resulting from an evaluation
X	Context	node in the context hierarchy
P	Polyline	primitive polygon in a graphics hierarchy
L	Link	link between nodes
Z	ContextLink	link between context nodes
G	GraphicLink	link in a graphics hierarchy
O	ContentLink	link from node to its content or constituents
V	Login	a user login
U	UnitCube	a primitive unit cube
S	Sweep	a primitive sweep
*	undefined	a PersistentObject with no ObjectType (error)

1	NodeKind synonmy
2	LinkType synonym
3	Predicate synonym
4	Association synonym

DataList

a node name
id: an object id
Character
Number
Boolean
NodeKind
LanguageType
items
that (last subject)
this (expression)
those items
contents of ResultList
a DataList name

DataList Combination DataList
Association of DataList
DataList with their Association
DataList that Filter
Graphic [immediately] in Graphic
DataList in context Context
either DataList or DataList
if Boolean then DataList [, else DataList]
DataList, sorted
DataList, without duplicates

Association

LinkType
inverse LinkType
name
id
creation date
creator
last modification date
contexts
all associations
[immediate] parts
inverse parts
Dimension
Distance in Units from Graphic [in Graphic]
an Association name

LinkType
InputAssociation with their InputAssociation
InputAssociation and InputAssociation
an input association name

Association of Association
Association with their Association

Association that Filter
either Association or Association
if Boolean then Association [, else Association]
Association Combination Association
the Number th Association
Association, sorted
Association, without duplicates

Predicate = Association

Filter

equal DataList
named Character
included in DataList
include DataList
of kind NodeKind
of type LanguageType

include Character

Counter

true

view [Counter] DataList
inherit from Context
are inherited by Context

[immediately] contain Graphic
[immediately] contained in Graphic
Measure [Quantifier] Graphic [in Graphic]
have Attribute is Value
have Attribute is Number

have Counter Association [with those items]
have Quantifier Association that Filter [with those items]
if Boolean then Filter [else Filter]
Filter Connective Filter
are Filter
are not Filter
do not Filter

Character
character string
substring of Character from Number for Number
Character append Character

Number
real number
count of DataList

minimum DataList
maximum DataList
total of DataList
product of DataList
Number + Number
Number - Number
- Number
Number x Number
Number / Number
Distance in Units between Graphic and Graphic [in Graphic]

Boolean

true
false
there are Counter DataList
Quantifier DataList Filter
not Boolean
Boolean Connective Boolean
Graphic Measure [Quantifier] DataList [in Graphic]

Graphic

polyline
DataList (of type graphic)

Counter

(at least one)
more than Number
less than Number
exact Number
not Counter
Counter Connective Counter
a Counter name

Quantifier
no
any
all
most
the (only one)
Counter
a Quantifier name

Measure
Distance is Counter Units
not Measure
Measure Connective Measure

2. HerLists.pas define list objects

```
 {*****************************************
 *
 *      HerLists.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 *****************************************}

unit HerLists;

interface

uses Objects;

{ This unit defines the low-level list structures of the Hermes system,
  as proposed in (Stahl, Ph.D. dissertation, 1993).

  All list, stack, queue and tree data structures should be derived from
  objects defined in this unit.

  Unsorted and sorted data structures are derived as follows:
        TCollection
        THermesList
        THermesStack  THermesQueue   THermesTree  TBehavior  THermesItemsList  THermesPointList
                                                             THermesQueryList

        TSortedCollection
        THermesSortedList
        THermesSortedItemsList   TLinkList
        TLinkTree

        TObject
        TBehaviorListItem

        TBufStream
        THermesStream
}

  const

  {****************************************
  *
  *       LanguageTypes
  *       ot_   object-type
  *
  *       Every object saved on the object stream needs to have a LanguageType.
  *       Therefore, every leaf item on the Hermes Substrate Object Hierarchy needs
  *       to define a valid (and unique) LanguageType. This LanguageType is
  *       used for type checking and for displaying pick lists of given types.
  *       Named objects are indexed on the name index by
  *       their name preceded by their LanguageType for efficient retrieval.
  ****************************************}
       ot_undefined     = '*';   { *      undefined         a PersistentObject with no ObjectType
(error)}
       NoName = ot_undefined;

       ot_ContextLink   = 'Z';   { Z      TContextLink      a link for linking the context hierarchy}
       ot_Link          = 'L';   { L      TLink             a link for normal hypermedia linking }
       ot_ContentLink   = 'O';   { O      TContextLink      a link for linking a node to its content}
       ot_GraphicLink   = 'G';   { G      TGraphicLink      a link for linking a graphic node to its
content}

       ot_HermesNode    = 'N';   { N      THermesNode       a node in the hypermedia }
       ot_HermesGraphic = 'g';   { g      THermesGraphic    a node with graphical content }

      { language elements }

       ot_DataList      = 'D';   { D      TDataList          language expression }
       ot_Association   = 'A';   { A      TAssociation       language expression }
       ot_Filter        = 'F';   { F      TFilter            language expression }
       ot_Predicate     = 'E';   { E      TprEdicate         Association saved as Predicate }
```

```
       { media elements }

       ot_Character     = 'c';   { c      TCharacter         PChar character string }
       ot_Number        = 'n';   { n      TNumber            real number }
       ot_Booleans      = 'b';   { b      TBooleans          true/false language expression }
       ot_Image         = 'i';   { i      TImage             bitmap raster image }
       ot_Pen           = 'p';   { p      TPen               pen-based sketch }
       ot_Sound         = 's';   { s      TSound             sound segment }
       ot_Video         = 'v';   { v      TVideo             video segment }
       ot_Animation     = 'a';   { a      TAnimation         Animation clip }
       ot_ComputedView  = 'w';   { w      TComputedVieW      stored composite display (OGL picture) }

       { terminology elements }

       ot_Counter       = 'C';   { C      TCounter           numeric relation }
       ot_Quantifier    = 'Q';   { Q      TQuantifier        quantification relation }
       ot_Measure       = 'M';   { M      TMeasure           graphic distance relation }

       { hypermedia elements }

       ot_NodeKind      = 'K';   { K      TNodeKind          defined type for nodes }
       ot_LinkType      = 'T';   { T      TLinkType          defined type for links }
       ot_Context       = 'X';   { X      TContext           node in the context hierarchy }
       ot_ResultList    = 'R';   { R      TResultList        stored list of nodes resulting from
DataList }
       ot_LogIn         = 'I';   { I      TLogIn             a user login with V, A, E privileges}

       ot_HermesPolyline = 'P';  { P      THermesPolyline    a node with links and a NodeKind }
       ot_UnitCube      = 'U';   { U      TUnitCube          a standard unit cube graphic }
       ot_Sweep         = 'S';   { S      TSweep             a standard sweep graphic }

       { synonym types }

       ot_NodeKindSynonym     = '1';   { 1      NodeKind synonmy  }
       ot_LinkTypeSynonym     = '2';   { 2      LinkType synonmy  }
       ot_PredicateSynonym    = '3';   { 3      Predicate synonmy  }
       ot_AssociationSynonym  = '4';   { 4      Association synonmy  }

  { The following constants are used for Dynamic virtual Methods. }
       StoreDMT                  = 2;
       EvalDMT                   = 3;
       HermesDisplayDMT          = 4;
       DisplayContentDMT         = 5;
       CopyDMT                   = 6;
       DisplayListDMT            = 7;

type

  {IdType = LongInt;}       {type for ObjectId of PersistentObject}

  LangType = Char;        {type for ObjectType of PersistentObject, defined in HerLinks}

       {****************************************
       *
       *       data structure for THermesObject
       *       This is a TObject specialized for the root of Hermes objects
       ****************************************}

       PHermesObject = ^THermesObject;
       THermesObject = object ( TObject )
      function    HermesDisplay : PChar;          virtual HermesDisplayDMT;    {display textual
definition}
           constructor Load( var S : TStream );
           procedure   Store( var S : TStream );       virtual StoreDMT;
           function    Copy : Pointer;                 virtual CopyDMT;             {most objects
should have a Copy method}
        end;

       {****************************************
       *
       *       data structure for THermesNamedObject
       *       This is a THermesObject with an optional Name
       ****************************************}

       PHermesNamedObject = ^THermesNamedObject;
```

```
      THermesNamedObject = object( THermesObject )
          constructor Init( NewName : PChar );
          constructor Load( var S : TStream );
          destructor  Done;                        virtual;
          procedure   Store( var S : TStream );    virtual StoreDMT;
     function    HermesDisplay : PChar;       virtual HermesDisplayDMT;
          function     Copy : Pointer;              virtual CopyDMT;
          procedure    CopyFrom( Source : PHermesNamedObject );
          function    GetName : PChar;
          procedure   SetName(NewName : PChar);
      private
          Name      : PChar;
end;

      {***************************************
      *
      *    data structure for TLinkId
      *
      ***************************************}

      PLinkId = ^TLinkId;
      TLinkId = object ( THermesObject )
          constructor Init(NewId : LongInt);
          constructor Load( var S : TStream );
     procedure   Store( var S : TStream );    virtual StoreDMT;
     function    HermesDisplay : PChar;       virtual HermesDisplayDMT;
     function    Copy : Pointer;              virtual CopyDMT;
     procedure   CopyFrom(Source : PLinkId);
     procedure   SetId(NewId : LongInt);
          function    GetId : LongInt;
private
          TheId : LongInt;
      end;

      {***************************************
      *
      *    data structure for THermesStream
      *    This is a TStream
      ***************************************}

      PHermesStream = ^THermesStream;
THermesStream = object ( TBufStream )
          procedure Error(Code, Info: Integer);    virtual;
      end;                                {need to register all storable objects}

      {***************************************
      *
      *    data structure for THermesList
      *    This is a TCollection with a Name
      ***************************************}

      PHermesList = ^THermesList;
      THermesList = object( TCollection )
          constructor Init;
          constructor Load(var S : TStream);
          destructor  Done;                        virtual;
          procedure   Store(var S : TStream);      virtual StoreDMT;
          function     Copy : Pointer;              virtual CopyDMT;
          procedure    CopyFrom(Source : PHermesList);
     procedure   DisplayList(Title : PChar);  virtual DisplayListDMT;
     function    HermesDisplay : PChar;       virtual HermesDisplayDMT;
          function    FirstOne : PObject;
          function    LastOne : PObject;
          function     Empty : Boolean;
          function    GetName : PChar;
          procedure   SetName(NewName : PChar);
private
  Name : PChar;
      end;

      {***************************************
      *
      *    data structure for THermesStack
      *    This is a generic Stack
      ***************************************}
```

```
      PHermesStack = ^THermesStack;
      THermesStack = object( THermesList )
          function  Copy : Pointer;                virtual CopyDMT;
          procedure Push(Item: PObject);
          function  Pop : PObject;
          function  TopOne : PObject;
end;

      {***************************************
      *
      *    data structure for THermesQueue
      *    This is a generic Queue
      ****************************************}

      PHermesQueue = ^THermesQueue;
      THermesQueue = object( THermesList )
          function  Copy : Pointer;          virtual CopyDMT;
          procedure Push(Item: PObject);
          function  Pop : PObject;
          function  TopOne : PObject;
      end;

      {***************************************
      *
      *    data structure for THermesTree
      *    This is a generic Tree
      ****************************************}

      PHermesTree = ^THermesTree;
      THermesTree = object( THermesList )
          function Copy : Pointer;            virtual CopyDMT;
       end;

      {*************************************************
      *
      *    data structure for THermesSortedList
      *    This is a sorted list of PChars
      *    (uses binary search -- default no dups)
      *************************************************}

      PHermesSortedList = ^THermesSortedList;
      THermesSortedList = object( TSortedCollection )
    constructor Init;
    constructor Load( var S : TStream );
    destructor  Done; virtual;
          procedure   Store(var S : TStream);         virtual StoreDMT;
    function    HermesDisplay : PChar;          virtual HermesDisplayDMT;
    function    Copy : Pointer;                 virtual CopyDMT;
          procedure   CopyFrom(Source : PHermesSortedList);
    procedure   DisplayList(Title : PChar);     virtual DisplayListDMT;
          function    Compare(Key1, Key2 : Pointer) : Integer; virtual;
          function    GetName : PChar;
          procedure   SetName(NewName : PChar);
private
     Name : PChar;
      end;

      {*********************************************
      *
      *    data structure for THermesItemsList
      *    This is an unsorted list of items with PChar names
      *********************************************}

      PHermesItemsList = ^THermesItemsList;
      THermesItemsList = object( THermesList )
          function  Copy : Pointer;            virtual CopyDMT;
    procedure DisplayList( Title : PChar );  virtual DisplayListDMT;
    function  Union       ( SourceList : PHermesItemsList ) : PHermesItemsList;
    function  Intersection( SourceList : PHermesItemsList ) : PHermesItemsList;
    function  Difference  ( SourceList : PHermesItemsList ) : PHermesItemsList;
    function  Append      ( SourceList : PHermesItemsList ) : PHermesItemsList;
    function  Sort                                          : PHermesItemsList;
    function  SortUnique                                    : PHermesItemsList;
    function  Unique                                        : PHermesItemsList;
       end;
```

```
      {*********************************************
      *
      *    data structure for THermesQueryList
      *    This is an unsorted list of QueryListItems with PChar names
      **********************************************}

      PHermesQueryList = ^THermesQueryList;
      THermesQueryList = object( THermesItemsList )
          function Copy : Pointer;             virtual CopyDMT;
      end;

      {*********************************************
      *
      *    data structure for THermesSortedItemsList
      *    This is a sorted list of items with PChar names
      *    (uses binary search -- default no dups)
      **********************************************}

      PHermesSortedItemsList = ^THermesSortedItemsList;
      THermesSortedItemsList = object( THermesSortedList )
          function  Copy : Pointer;                 virtual CopyDMT;
    procedure DisplayList(Title : PChar);    virtual DisplayListDMT;
    function  Compare(Item1, Item2 : Pointer) : Integer; virtual;
    procedure Append(SourceList : PHermesItemsList);
       end;

      {**************************************
      *
      *    data structure for TLinkList
      *
      **************************************}

PLinkList = ^TLinkList;                    {List of LinkId's, sorted by Id}
      TLinkList = object ( THermesSortedList )         {named by main LinkType}
          function  Copy : Pointer;                virtual CopyDMT;
    procedure CopyFrom(Source : PLinkList);
    procedure DisplayList(Title : PChar);    virtual DisplayListDMT;
    function  Compare(Item1, Item2: Pointer) : Integer; virtual;
      end;

      {**************************************
      *
      *    data structure for TLinkTree
      *
      **************************************}

PLinkTree = ^TLinkTree;                        {List of LinkLists}
      TLinkTree = object ( THermesSortedItemsList )    {sorted by main LinkType, name of LinkLists}
          function    Copy : Pointer;              virtual CopyDMT;
    procedure   CopyFrom(Source : PLinkTree);
    procedure   DisplayList(Title : PChar);  virtual DisplayListDMT;
    function    Compare(Item1, Item2: Pointer) : Integer; virtual;
      end;

      {**************************************
      *
      *       data structure for TBehaviorListItem
      *
      **************************************}

TProcedureDesignator = ( LazyVCopy,
                         UserDefinedProcedure1,
                         UserDefinedProcedure2,
                         UserDefinedProcedure3,
                         UserDefinedProcedure4,
                         UserDefinedProcedure5,
                         UserDefinedProcedure6,
                         UserDefinedProcedure7,
                         NONE);

      PBehaviorListItem = ^TBehaviorListItem;
      TBehaviorListItem = object ( TObject )
          constructor Init;
          constructor Load(var S : TStream);
          destructor  Done;                        virtual;
          procedure   Store(var S : TStream);      virtual;
```

```
      function    HermesDisplay : PChar;       virtual HermesDisplayDMT;
            function    Copy : Pointer;              virtual CopyDMT;
      procedure   CopyFrom(Source : PBehaviorListItem);
            procedure   SetTheProcedure(NewProcedure : TProcedureDesignator);
            procedure   SetParameter1(NewParameter : LongInt);
            procedure   SetParameter2(NewParameter : LongInt);
            procedure   SetParameter3(NewParameter : LongInt);
            procedure   SetParameter4(NewParameter : LongInt);
            function    GetTheProcedure : TProcedureDesignator;
            function    GetParameter1 : LongInt;
            function    GetParameter2 : LongInt;
            function    GetParameter3 : LongInt;
            function    GetParameter4 : LongInt;
  private
             TheProcedure : TProcedureDesignator;    {see enumerated list above}
             Parameter1   : LongInt;
             Parameter2   : LongInt;
             Parameter3   : LongInt;
             Parameter4   : LongInt;
 end;

       {****************************************
       *
       *       data structure for TBehavior
       *
       ****************************************}

       PBehavior = ^TBehavior;
       TBehavior = object( THermesList )        {list of BehaviorListItems}
           function  Copy : Pointer;              virtual CopyDMT;
      procedure AddProcedure(NewProcedure : PBehaviorListItem);
      procedure DeleteProcedure(OldProcedure : PBehaviorListItem);
        end;

  {****************************************
       *
       *    data structure for THermesPointList
       *    this is a collection of Points
       ****************************************}

       PHermesPointList = ^THermesPointList;
       THermesPointList = object (THermesList)
       end;


{=====================================================================}

implementation

uses Strings;

       {****************************************
       *
       *       methods for THermesObject
       *
       ****************************************}
constructor THermesObject.Load( var S : TStream );
 begin
 end;

procedure   THermesObject.Store( var S : TStream );
 begin
 end;

function  THermesObject.HermesDisplay : PChar;
 begin
   HermesDisplay := 'THermesObject';
 end;

function THermesObject.Copy : Pointer;
 begin
   Copy := New(PHermesObject, Init);
 end;

       {****************************************
       *
```

```
        *       methods for THermesNamedObject
        *
        ****************************************}

constructor THermesNamedObject.Init(NewName : PChar);
 begin
   if not inherited Init then Fail;
   if assigned(NewName)
     then Name := StrNew(NewName)
     else Name := StrNew(NoName);
 end;

destructor  THermesNamedObject.Done;
 begin
   if assigned(Name) then
      StrDispose(Name);
   inherited Done;
 end;

constructor THermesNamedObject.Load(var S : TStream);
 begin
   Name := S.StrRead;
 end;

procedure   THermesNamedObject.Store(var S : TStream);
 begin
   S.StrWrite(Name);
 end;

function   THermesNamedObject.HermesDisplay : PChar;
 begin
   HermesDisplay := StrNew(Name);
 end;

function THermesNamedObject.Copy : Pointer;
 var TheNamedObject : PHermesNamedObject;
 begin
   TheNamedObject := New( PHermesNamedObject, Init( NoName ));
   TheNamedObject^.CopyFrom( @Self );
   Copy := TheNamedObject;
 end;

procedure THermesNamedObject.CopyFrom( Source : PHermesNamedObject );
 var TempName : PChar;
 begin
   TempName := Source^.GetName;
   SetName(TempName);
   StrDispose(TempName);
 end;

function THermesNamedObject.GetName : PChar;
 begin
   GetName := StrNew(Name);
 end;

procedure THermesNamedObject.SetName(NewName : PChar);
 begin
   if assigned(Name) then
      StrDispose(Name);
   Name := StrNew(NewName);
 end;

        {****************************************
        *
        *       methods for TLinkId
        *
        ****************************************}

constructor TLinkId.Init(NewId : LongInt);
 begin
   inherited Init;
   TheId := NewId;
 end;

constructor TLinkId.Load(var S : TStream);
 begin
```

```
   S.Read(TheId, SizeOf(LongInt));
 end;

procedure   TLinkId.Store(var S : TStream);
 begin
   S.Write(TheId, SizeOf(LongInt));
 end;

function  TLinkId.HermesDisplay : PChar;
var TempDisplay : array [0..500] of Char;      {create storage for PChar}
    TempString  : String;
    TempPChar   : PChar;
begin
  Str(TheId, TempString);                    {convert LongInt to String}
  StrPCopy(TempDisplay, TempString);         {convert String to PChar}
  HermesDisplay := StrNew(TempDisplay);
end;

function TLinkId.Copy : Pointer;
var TheCopy : PLinkId;
begin
  TheCopy := New(PLinkId, Init(0));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure TLinkId.CopyFrom(Source : PLinkId);
begin
  SetId(Source^.GetId);
end;

procedure   TLinkId.SetId(NewId : LongInt);
begin
  TheId := NewId;
end;

function    TLinkId.GetId : LongInt;
begin
  GetId := TheId;
end;

        {***************************************
        *
        *    methods for THermesStream
        *
        ****************************************}

procedure THermesStream.Error(Code, Info: Integer);
 begin
   Writeln('Stream error: ', Code, ' (', Info, ')');
         Halt(1);
 end;

        {***************************************
        *
        *    methods for THermesList
        *
        ****************************************}

constructor THermesList.Init;
 begin
   inherited Init(1,5);
   Name := StrNew(NoName);
 end;

constructor THermesList.Load(var S : TStream);
 begin
   inherited Load(S);
   Name := S.StrRead;
 end;

destructor  THermesList.Done;
 begin
   if assigned(Name) then
      StrDispose(Name);
   inherited Done;
```

```
 end;

procedure   THermesList.Store(var S : TStream);
 begin
   inherited Store(S);
   S.StrWrite(Name);
 end;

function THermesList.HermesDisplay : PChar;
 begin
   HermesDisplay := StrNew( Name );
 end;

function THermesList.Copy : Pointer;
 var TheObject : PHermesList;
 begin
   New(TheObject, Init);
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
 end;

procedure THermesList.CopyFrom(Source : PHermesList);
     procedure CopyOne(Item : PHermesObject); far;
      begin
        Insert(Item^.Copy);
      end;
 var TempName : PChar;
 begin
   Source^.ForEach(@CopyOne);
   TempName := Source^.GetName;
   SetName( TempName );
   StrDispose( TempName );
 end;

function THermesList.GetName : PChar;
 begin
   GetName := StrNew( Name );
 end;

procedure THermesList.SetName(NewName : PChar);
 begin
   if assigned( Name ) then
      StrDispose(Name);
   if assigned( NewName ) then
      Name := StrNew(NewName)
   else
      Name := nil;
 end;

procedure THermesList.DisplayList(Title : PChar);
 var LineCount : Integer;

 procedure DisplayOne(Item : PChar); far;
     begin  {DisplayOne}
       Writeln(Item);
       Inc(LineCount);
       if (LineCount > 20) then
          LineCount := 0;
     end;   {DisplayOne}
 begin
   LineCount := 0;
   Writeln;
   Writeln(Title);
   Writeln;
   ForEach(@DisplayOne);
   Writeln;
 end;

function THermesList.FirstOne: PObject;
 begin
   if Count > 0 then
      FirstOne := At( 0 )
   else
      FirstOne := nil;
 end;
```

```
function THermesList.LastOne: PObject;
 begin
   if Count > 0 then
      LastOne := At(Count - 1)
   else
      LastOne := nil
 end;

function THermesList.Empty : Boolean;
 begin
   Empty := (Count = 0);
 end;

       {****************************************
       *
       *    methods for THermesStack
       *
       ****************************************}

function THermesStack.Copy : Pointer;
 var TheObject : PHermesStack;
 begin
   New(TheObject, Init);
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
 end;

procedure THermesStack.Push(Item: PObject);
 begin
   Insert( Item );
 end;

function  THermesStack.Pop : PObject;
 var Return : Pointer;
 begin
   Return := LastOne;
   if assigned( Return ) then
      AtDelete( Count - 1 );
   Pop := Return;
 end;

function  THermesStack.TopOne : PObject;
 begin
   TopOne := LastOne;
 end;

       {****************************************
       *
       *    methods for THermesQueue
       *
       ****************************************}

function THermesQueue.Copy : Pointer;
 var TheObject : PHermesQueue;
 begin
   New(TheObject, Init);
   TheObject^.CopyFrom( @Self );
   Copy := TheObject;
 end;

procedure THermesQueue.Push(Item: PObject );
 begin
   Insert(Item);
 end;

function  THermesQueue.Pop : PObject;
 var Return : POINTER;
 begin
   Return := FirstOne;
   if assigned( Return ) then
      AtDelete(0);
   Pop := Return;
 end;

function  THermesQueue.TopOne : PObject;
 begin
```

```
   TopOne := FirstOne;
 end;

        {***************************************
        *
        *    methods for THermesTree
        *
        ***************************************}

function THermesTree.Copy : Pointer;
 var TheObject : PHermesTree;
 begin
   New(TheObject, Init);
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
 end;

  {***************************************
        *
        *    methods for THermesSortedList
        *
        ***************************************}

constructor THermesSortedList.Init;
 begin
   inherited Init(1, 15);
   Name := StrNew(NoName);
 end;

constructor THermesSortedList.Load(var S : TStream);
 begin
   inherited Load(S);
   Name := S.StrRead;
 end;

procedure THermesSortedList.Store(var S : TStream);
 begin
   inherited Store(S);
   S.StrWrite(Name);
 end;

destructor THermesSortedList.Done;
 begin
   if assigned(Name) then
      StrDispose(Name);
   inherited Done;
 end;

procedure THermesSortedList.DisplayList(Title : PChar);
    procedure DisplayOne(Item : PChar); far;
     begin  {DisplayOne}
       Writeln(Item);
     end;   {DisplayOne}
 begin
   Writeln;
   Writeln(Title);
   Writeln;
   ForEach(@DisplayOne);
   Writeln;
 end;

function THermesSortedList.Compare(Key1, Key2: Pointer) : Integer;
 var TheResult : Integer;
 begin
   TheResult := StrComp(PChar(Key1), PChar(Key2));
   if TheResult < 0
     then Compare := -1
     else if TheResult > 0
       then Compare := 1
       else Compare := 0;
 end;

function   THermesSortedList.Copy : Pointer;
 var TheObject : PHermesSortedList;
 begin
   New(TheObject, Init);
```

```
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
 end;

procedure THermesSortedList.CopyFrom(Source : PHermesSortedList);
   procedure CopyOne(Item : PHermesObject); far;
    begin
      Insert(Item^.Copy);
    end;
 var TempName : PChar;
 begin
   Source^.ForEach(@CopyOne);
   TempName := Source^.GetName;
   SetName(TempName);
   StrDispose(TempName);
 end;

function   THermesSortedList.HermesDisplay : PChar;
 begin
   HermesDisplay := StrNew(Name);
 end;

function    THermesSortedList.GetName : PChar;
 begin
   GetName := StrNew(Name);
 end;

procedure   THermesSortedList.SetName(NewName : PChar);
 begin
   if assigned( Name )
     then StrDispose(Name);
   if assigned( NewName )
     then Name := StrNew(NewName)
     else Name := StrNew( NoName );
 end;

       {****************************************
       *
       *    methods for THermesItemsList
       *
       ****************************************}

function THermesItemsList.Copy : Pointer;
 var TheObject : PHermesItemsList;
 begin
   New(TheObject, Init);
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
 end;

procedure THermesItemsList.DisplayList(Title : PChar);
    procedure DisplayOne(Item : PHermesObject); far;
    begin  {DisplayOne}
      Writeln(Item^.HermesDisplay);
    end;   {DisplayOne}
begin
       Writeln;
  Writeln(Title);
  Writeln;
  ForEach(@DisplayOne);
  Writeln;
end;

function THermesItemsList.Union( SourceList : PHermesItemsList ) : PHermesItemsList;
{append copies of items from SourceList to this HermesItemsList
 then remove duplicates and sort}
begin
  Union := Append(SourceList)^.SortUnique;
end;

function THermesItemsList.Intersection( SourceList : PHermesItemsList ) : PHermesItemsList;
{Add any item found that is also in SourceList}
var
  TempList : PHermesItemsList;
    procedure AddOne(Item : PHermesObject); far;
    var
```

```
      Index : Integer;
    begin  {AddOne}
      Index := IndexOf(Item);
      if Index > -1
        then TempList^.Insert(Item);
    end;   {AddOne}
begin
  TempList := New(PHermesItemsList, Init);
  SourceList^.ForEach(@AddOne);
  Intersection := TempList;
end;

function THermesItemsList.Difference( SourceList : PHermesItemsList ) : PHermesItemsList;
{Subtract any item found that is also in SourceList}
    procedure SubtractOne(Item : PHermesObject); far;
    var
      Index : Integer;
    begin  {AddOne}
      Index := IndexOf(Item);
      if Index > -1
        then AtDelete(Index);
    end;   {AddOne}
begin
  SourceList^.ForEach(@SubtractOne);
end;

function THermesItemsList.Append( SourceList : PHermesItemsList ) : PHermesItemsList;
{append copies of items from SourceList to the HermesItemsList}
    procedure AddOne(Item : PHermesObject); far;
    begin  {AddOne}
      Insert(Item^.Copy);
    end;   {AddOne}
begin
  SourceList^.ForEach(@AddOne);
end;

function THermesItemsList.Sort : PHermesItemsList;
var
  TempList1 : PHermesSortedItemsList;
  TempList2 : PHermesItemsList;
    procedure AddOne(Item : PHermesObject); far;
    begin  {AddOne}
      TempList1^.Insert(Item^.Copy);
    end;   {AddOne}
    procedure AddTwo(Item : PHermesObject); far;
    begin  {AddTwo}
      TempList2^.Insert(Item^.Copy);
    end;   {AddTwo}
begin
  TempList1 := New(PHermesSortedItemsList, Init);
  TempList1^.Duplicates := True;
  ForEach(@AddOne);
  TempList2 := New(PHermesItemsList, Init);
  TempList1^.ForEach(@AddTwo);
  Sort := TempList2;
  Dispose(TempList1, Done);
end;

function THermesItemsList.SortUnique : PHermesItemsList;
var
  TempList1 : PHermesSortedItemsList;
  TempList2 : PHermesItemsList;
    procedure AddOne(Item : PHermesObject); far;
    begin  {AddOne}
      TempList1^.Insert(Item^.Copy);
    end;   {AddOne}
    procedure AddTwo(Item : PHermesObject); far;
    begin  {AddTwo}
      TempList2^.Insert(Item^.Copy);
    end;   {AddTwo}
begin
  TempList1 := New(PHermesSortedItemsList, Init);
  TempList1^.Duplicates := False;
  ForEach(@AddOne);
  TempList2 := New(PHermesItemsList, Init);
  TempList1^.ForEach(@AddTwo);
```

```
  SortUnique := TempList2;
  Dispose(TempList1, Done);
end;

function THermesItemsList.Unique : PHermesItemsList;
var
  TempList : PHermesItemsList;
    procedure AddOne(Item : PHermesObject); far;
    var
      Index : Integer;
    begin  {AddOne}
      Index := TempList^.IndexOf(Item);
      if Index = -1                          {if Item not yet in TempList}
        then TempList^.Insert(Item^.Copy);   {then insert copy of it}
    end;   {AddOne}
begin
  TempList := New(PHermesItemsList, Init);
  ForEach(@AddOne);
  Unique := TempList;
end;

        {****************************************
        *
        *    methods for THermesQueryList
        *
        ****************************************}

function THermesQueryList.Copy : Pointer;
var TheObject : PHermesQueryList;
begin
  New(TheObject, Init);
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

        {****************************************
        *
        *    methods for THermesSortedItemsList
        *
        ****************************************}

function THermesSortedItemsList.Copy : Pointer;
var TheObject : PHermesSortedItemsList;
begin
  New(TheObject, Init);
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

procedure THermesSortedItemsList.DisplayList(Title : PChar);
    procedure DisplayOne(Item : PHermesObject); far;
     begin  {DisplayOne}
       Writeln(Item^.HermesDisplay);
     end;   {DisplayOne}
begin
        Writeln;
  Writeln(Title);
  Writeln;
  ForEach(@DisplayOne);
  Writeln;
end;

function THermesSortedItemsList.Compare(Item1, Item2: Pointer) : Integer;
var TheResult : Integer;
begin
  TheResult := StrComp(PHermesNamedObject(Item1)^.GetName, PHermesNamedObject(Item2)^.GetName);
  if TheResult < 0
    then Compare := -1
    else if TheResult > 0
      then Compare := 1
      else Compare := 0;
end;

procedure THermesSortedItemsList.Append(SourceList : PHermesItemsList);
{append copies of items from SourceList to the HermesSortedItemsList}
    procedure AddOne(Item : PHermesObject); far;
```

```
    begin  {AddOne}
      Insert(Item^.Copy);
    end;   {AddOne}
begin
  SourceList^.ForEach(@AddOne);
end;

        {***************************************
        *
        *       methods for TLinkList
        *
        ***************************************}

function   TLinkList.Copy : Pointer;
var TheObject : PLinkList;
begin
        New(TheObject, Init);
        TheObject^.CopyFrom(@Self);
        Copy := TheObject;
end;

procedure   TLinkList.CopyFrom(Source : PLinkList);
    procedure CopyOne(Item : PLinkId); far;
    begin
      Insert(Item^.Copy);
    end;
var
  TempName : PChar;
begin
  Source^.ForEach(@CopyOne);
  TempName := Source^.GetName;
  SetName(TempName);
  StrDispose(TempName);
end;

procedure TLinkList.DisplayList(Title : PChar);
    procedure DisplayOne(Item : PLinkId); far;
    begin  {DisplayOne}
      Writeln(Item^.GetId);
    end;   {DisplayOne}
begin
  Writeln(Title);
  Writeln('Name of List: ', GetName);
  ForEach(@DisplayOne);
  Writeln;
end;

function TLinkList.Compare(Item1, Item2: Pointer) : Integer;
var
  Diff : LongInt;
begin
  if (assigned(Item1) and assigned(Item2))
    then Diff := PLinkId(Item1)^.GetId - PLinkId(Item2)^.GetId
    else Diff := -1;
  if (Diff < 0)
    then Compare := -1
    else
      if (Diff = 0)
        then Compare :=  0
        else Compare :=  1;
end;

        {***************************************
        *
        *       methods for TLinkTree
        *
        ***************************************}

function   TLinkTree.Copy : Pointer;
var
  TheObject : PLinkTree;
begin
        New(TheObject, Init);
        TheObject^.CopyFrom(@Self);
        Copy := TheObject;
end;
```

```
procedure   TLinkTree.CopyFrom(Source : PLinkTree);
    procedure CopyOne(Item : PLinkList); far;
     begin
       Insert(Item^.Copy);
     end;
var
  TempName : PChar;
begin
  Source^.ForEach(@CopyOne);
  TempName := Source^.GetName;
  SetName(TempName);
  StrDispose(TempName);
end;

procedure TLinkTree.DisplayList(Title : PChar);
    procedure DisplayOne(Item : PLinkList); far;
    begin  {DisplayOne}
      Item^.DisplayList(Item^.GetName);
    end;   {DisplayOne}
begin
  Writeln(Title);
  ForEach(@DisplayOne);
  Writeln('...');
end;

function TLinkTree.Compare(Item1, Item2: Pointer) : Integer;
var
  Diff : Integer;
begin
  Diff := StrComp(PLinkList(Item1)^.GetName, PLinkList(Item2)^.GetName);
  if (Diff < 0)
    then Compare := -1
    else
      if (Diff = 0)
        then Compare :=  0
        else Compare :=  1;
end;

        {***************************************
        *
        *       methods for TBehaviorListItem
        *
        ****************************************}

constructor TBehaviorListItem.Init;
 begin
   inherited Init;
   TheProcedure := NONE;
   Parameter1 := 0;
   Parameter2 := 0;
   Parameter3 := 0;
   Parameter4 := 0;
 end;

destructor  TBehaviorListItem.Done;
 begin
   TheProcedure := NONE;
   Parameter1 := 0;
   Parameter2 := 0;
   Parameter3 := 0;
   Parameter4 := 0;
   inherited Done;
 end;

constructor TBehaviorListItem.Load(var S : TStream);
 begin
   S.Read(TheProcedure, SizeOf(TheProcedure));
   S.Read(Parameter1, SizeOf(LongInt));
   S.Read(Parameter2, SizeOf(LongInt));
   S.Read(Parameter3, SizeOf(LongInt));
   S.Read(Parameter4, SizeOf(LongInt));
 end;

procedure   TBehaviorListItem.Store(var S : TStream);
 begin
```

```
   S.Write(TheProcedure, SizeOf(TheProcedure));
   S.Write(Parameter1, SizeOf(LongInt));
   S.Write(Parameter2, SizeOf(LongInt));
   S.Write(Parameter3, SizeOf(LongInt));
   S.Write(Parameter4, SizeOf(LongInt));
 end;

function  TBehaviorListItem.HermesDisplay : PChar;
var
  TempDisplay : array[0..79] of Char;
begin
   CASE TheProcedure of
    LazyVCopy  : StrCopy(TempDisplay, 'VCopyLazy');
    NONE       : StrCopy(TempDisplay, 'none');
    else         StrCopy(TempDisplay, 'undefined')
  end;  {case}
  HermesDisplay := TempDisplay;
 end;

function TBehaviorListItem.Copy : Pointer;
 var TheCopy : PBehaviorListItem;
 begin
   New(TheCopy, Init);
   TheCopy^.CopyFrom(@Self);
   Copy := TheCopy;
 end;

procedure TBehaviorListItem.CopyFrom(Source : PBehaviorListItem);
 begin
   SetTheProcedure(Source^.GetTheProcedure);
   SetParameter1(Source^.GetParameter1);
   SetParameter2(Source^.GetParameter2);
   SetParameter3(Source^.GetParameter3);
   SetParameter4(Source^.GetParameter4);
end;

procedure   TBehaviorListItem.SetTheProcedure(NewProcedure : TProcedureDesignator);
begin
  TheProcedure := NewProcedure;
end;

procedure   TBehaviorListItem.SetParameter1(NewParameter : LongInt);
begin
  Parameter1 := NewParameter;
end;

procedure   TBehaviorListItem.SetParameter2(NewParameter : LongInt);
begin
  Parameter2 := NewParameter;
end;

procedure   TBehaviorListItem.SetParameter3(NewParameter : LongInt);
begin
  Parameter3 := NewParameter;
end;

procedure   TBehaviorListItem.SetParameter4(NewParameter : LongInt);
begin
  Parameter4 := NewParameter;
end;

function    TBehaviorListItem.GetTheProcedure : TProcedureDesignator;
begin
  GetTheProcedure := TheProcedure;
end;

function    TBehaviorListItem.GetParameter1 : LongInt;
begin
  GetParameter1 := Parameter1;
end;

function    TBehaviorListItem.GetParameter2 : LongInt;
begin
  GetParameter2 := Parameter2;
end;
```

```
function    TBehaviorListItem.GetParameter3 : LongInt;
begin
  GetParameter3 := Parameter3;
end;

function    TBehaviorListItem.GetParameter4 : LongInt;
begin
  GetParameter4 := Parameter4;
end;

        {***************************************
        *
        *       methods for TBehavior
        *
        ****************************************}

function TBehavior.Copy : Pointer;
var TheCopy : PBehavior;
begin
  TheCopy := New(PBehavior, Init);
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure TBehavior.AddProcedure(NewProcedure : PBehaviorListItem);
begin
  Insert(NewProcedure);
end;

procedure TBehavior.DeleteProcedure(OldProcedure : PBehaviorListItem);

    function Matches(Item : PBehaviorListItem) : Boolean; far;
    begin  {Matches}
      if (OldProcedure^.GetTheProcedure = Item^.GetTheProcedure)
        then Matches := True
        else Matches := False;
    end;   {Matches}
var
  TheFirst : Pointer;
begin
  TheFirst := FirstThat(@Matches);
  if TheFirst <> nil
    then Delete(TheFirst);
end;

        {***************************************
        *
        *       methods for THermesPointList
        *
        ****************************************}

end.    {HerLists}
```

3. HerBasic.pas define other basic objects

```
 {*****************************************
 *
 *      HerBasic.pas
 *
 *      version 2.0 -- Spring 1994
 *      (c) copyright 1994 by Gerry Stahl
 *      all rights reserved
 *****************************************}

unit HerBasic;

interface

uses Objects,
     HerLists;

{ This unit defines the low-level objects of the Hermes system
  as proposed in (Stahl, Ph.D. dissertation, 1993).

  SPECIFICATION:
               All objects used in the Hermes system should derive from objects in this unit
    or in HerLists or in HerWorld.

       DESIGN:
               Define abstract objects that inherit from the Turbo Pascal
               for Windows library.  Other Hermes units will USE this unit.

               The definitions in this unit customize TPW objects.
               Porting to new versions of Pascal or to other programming systems
               will just need to modify the simple definitions in this unit.
               Improvements to method algorithms made here will be propagated
               to the rest of Hermes.  Definitions in this unit establish
               consistency thoughout Hermes.

               The central Hermes ontology is derived in HerBasic as follows:
                              TObject                 (can be stored on TStream)
                              THermesObject           (can be stored on THermesStream)
                              THermesNamedObject      (data --  Name : PChar)
        TActiveObject           (can have conditionals and procedures)
}

type

       {***************************************
       *
       *    data structure for TActiveObject
       *
       ****************************************}

       PActiveObject = ^TActiveObject;
       TActiveObject = object( THermesNamedObject )
               constructor Init(NewName : PChar);
               destructor  Done;                           virtual;
               constructor Load(var S : TStream);
               procedure   Store(var S : TStream);         virtual StoreDMT;
               function    Copy : Pointer;                 virtual CopyDMT;
    procedure   CopyFrom(Source : PActiveObject);
    function    EvalBooleans : Boolean;
    procedure   SetBehavior(NewBehavior : PBehavior);
    function    GetBehavior : PBehavior;
    procedure   SetCondition(NewCondition : PActiveObject);
    function    GetCondition : PActiveObject;
  private
               TheBehavior   : PBehavior;
               TheCondition  : PActiveObject;        {must be PBoolean -- check in implementation
code}
  end;

       {***************************************
       *
       *    data structure for TContextsList
       *
```

```
      ****************************************}

      PContextsList = ^TContextsList;
      TContextsList = object( THermesObject )        {list of all contexts for}
          constructor Load(var S : TStream);
          constructor Init;                          {an object to be seen in}
          destructor  Done;                        virtual;
          procedure   Store(var S : TStream);     virtual StoreDMT;
     function    HermesDisplay : PChar;      virtual HermesDisplayDMT;
          function    Copy : Pointer;             virtual CopyDMT;
     procedure   CopyFrom(Source : PContextsList);
          procedure   SetOriginalContext(NewContext : LongInt);
          procedure   SetAddedContexts(NewList : PLinkList);
          procedure   SetDeletedContexts(NewList : PLinkList);
          procedure   SetSwitchContext(NewSwitch : LongInt);
     function    GetOriginalContext : LongInt;
          function    GetAddedContexts : PLinkList;
          function    GetDeletedContexts : PLinkList;
          function    GetSwitchContext : LongInt;
private
          OriginalContext : LongInt;           {main context object created in}
          AddedContexts   : PLinkList;         {added contexts for viewing object}
          DeletedContexts : PLinkList;         {deleted contexts for not viewing}
          SwitchContext   : LongInt;           {original to be switched to this one}
      end;

 { The Coordinate Type is used for X, Y, Z coordinates of Points
   for graphical (polyline) objects; To conserve space, change this
   to Real; To increase resolution, change it to Extended}
 Coordinate = Double;

      {****************************************
      *
      *       data structure for THermesPoint
      *       This is a 3-D point
      ****************************************}

      PHermesPoint = ^THermesPoint;
      THermesPoint = object (TObject)
             constructor Init(NewX, NewY, NewZ : Coordinate);
             destructor  Done;                          virtual;
             constructor Load(var S : TStream);
             procedure   Store(var S : TStream);         virtual StoreDMT;
             function              Copy : Pointer;                 virtual CopyDMT;
             procedure             CopyFrom(Source : PHermesPoint);
   procedure   DisplayList(Title : PChar);     virtual DisplayListDMT;
   function    HermesDisplay : PChar;          virtual HermesDisplayDMT;
             function    GetX : Coordinate;
             procedure   SetX(NewX : Coordinate);
             function    GetY : Coordinate;
             procedure   SetY(NewY : Coordinate);
             function    GetZ : Coordinate;
             procedure   SetZ(NewZ : Coordinate);
 private
   Coordinates : array['X' .. 'Z'] of Coordinate;
      end;

      {****************************************
      *
      *    data structure for TExtent
      *
      ****************************************}

      PExtent = ^TExtent;
 TExtent = object (THermesObject)
             constructor Init(NewTop, NewBottom : PHermesPoint);
             destructor  Done;                        virtual;
             constructor Load(var S : TStream);
             procedure   Store(var S : TStream);   virtual StoreDMT;
   function    HermesDisplay : PChar;    virtual HermesDisplayDMT;
             function    Copy : Pointer;           virtual CopyDMT;
   procedure   CopyFrom(Source : PExtent);
             procedure   SetTop(NewTop : PHermesPoint);
             function    GetTop : PHermesPoint;
             procedure   SetBottom(NewBottom : PHermesPoint);
             function    GetBottom : PHermesPoint;
```

```
    private
            Top       : PHermesPoint;
Bottom : PHermesPoint;
    end;

    {****************************************
    *
    *    data structure for TTransform
    *
    ****************************************}

    PTransform = ^TTransform;
    TTransform = object (THermesObject)
            constructor Init(NewFixed, NewDisplace, NewScale, NewRotate : PHermesPoint);
constructor InitDefault;         {all null transforms}
            destructor  Done;                        virtual;
            constructor Load(var S : TStream);
            procedure   Store(var S : TStream);       virtual StoreDMT;
function     HermesDisplay : PChar;        virtual HermesDisplayDMT;
            function    Copy : Pointer;               virtual CopyDMT;
procedure    CopyFrom(Source : PTransform);
            procedure   SetFixed(NewFixed : PHermesPoint);
            function    GetFixed : PHermesPoint;
            procedure   SetDisplace(NewDisplace : PHermesPoint);
            function    GetDisplace : PHermesPoint;
            procedure   SetScale(NewScale : PHermesPoint);
            function    GetScale : PHermesPoint;
            procedure   SetRotate(NewRotate : PHermesPoint);
            function    GetRotate : PHermesPoint;
    private
            Fixed   : PHermesPoint;
            Displace: PHermesPoint;
            Scale   : PHermesPoint;
            Rotate  : PHermesPoint;
    end;

    {****************************************
    *
    *    data structure for TQueryListItem
    *
    ****************************************}

    PQueryListItem = ^TQueryListItem;
    TQueryListItem = object (THermesObject)
            constructor Init;
            constructor Load(var S : TStream);
            procedure   Store(var S : TStream); virtual StoreDMT;
            function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
            function    Copy : Pointer;         virtual CopyDMT;
procedure    CopyFrom(Source : PQueryListItem);
            procedure   SetDataId(NewDataId : LongInt);
            procedure   SetContext(NewContext : LongInt);
            procedure   SetLevel(NewLevel : Integer);
            procedure   SetSequence(NewSequence : Integer);
            procedure   SetHeading(NewHeading : PChar);
            procedure   SetTransformation(NewTransformation     : PTransform);
            function    GetDataId : LongInt;
            function    GetContext : LongInt;
            function    GetLevel : Integer;
            function    GetSequence : Integer;
            function    GetHeading : PChar;
            function    GetTransformation : PTransform;
    private
            DataId         : LongInt;               {the data item's Id}
Context        : LongInt;               {the context}
            Level          : Integer;               {indent level}
Sequence       : Integer;               {physical sequence of item}
            Heading        : PChar;                 {type heading for outline}
            Transformation : PTransform;            {display transform}
    end;

    {****************************************
    *
    *    data structure for TAttribute
    *
    ****************************************}
```

```
        PAttribute = ^TAttribute;
        TAttribute = object (THermesObject)
        end;

        {****************************************
        *
        *    data structure for TTextualAttribute
        *
        ****************************************}

        PTextualAttribute = ^TTextualAttribute;
        TTextualAttribute = object( TAttribute )
  end;

        {****************************************
        *
        *    data structure for TGraphicalAttribute
        *
        ****************************************}

        PGraphicalAttribute = ^TGraphicalAttribute;
        TGraphicalAttribute = object( TAttribute )
  end;

        {****************************************
        *
        *    data structure for TSublink
        *
        ****************************************}

        PSubLink = ^TSublink;
        TSublink = object (TActiveObject)
                constructor Init;
                destructor  Done; virtual;
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PSubLink);
                function    HermesDisplay : PChar; virtual HermesDisplayDMT;
        procedure   SetListofContexts(NewListofContext : PContextsList);
                procedure   SetAttribute(NewAttribute : PAttribute);
                function    GetListofContexts : PContextsList;
                function    GetAttribute : PAttribute;
        private
                ListofContexts    : PContextsList;     {a list of valid contexts}
    TheAttribute      : PAttribute;        {attributes for lines, text, etc}
        end;

        {****************************************
        *
        *    data structure for TGraphicalSublink
        *
        ****************************************}

        PGraphicalSubLink = ^TGraphicalSublink;
        TGraphicalSublink = object (TSublink)
                constructor Init;
                destructor  Done; virtual;
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PGraphicalSubLink);
                function    HermesDisplay : PChar; virtual HermesDisplayDMT;
                procedure   SetTransformation(NewTransformation      : PTransform);
                function    GetTransformation : PTransform;
        private
                TheTransformation      : PTransform;        {a view or modeling transform}
        end;

{*********************************}
{ ErrorMsg Severity Types         }
{*********************************}

        TErrorType = ( et_NOTICE, et_WARNING, et_SERIOUS );
```

```
{*********************************}
{ ErrorMsg procedure              }
{*********************************}

procedure ErrorMsg( Severity: TErrorType; Message: PChar );

{*********************************}
{ DB (debugging) procedure              }
{*********************************}

procedure DB( TheNumber: LongInt );

{*********************************}
{ function StripBlanks            }
{*********************************}

function StripBlanks( ThePChar: PChar ) : PChar;
{normalize a PChar to 40 lower case characters, no leading or trailing blanks}

{*********************************}
{ NotImplemented procedure        }
{*********************************}

procedure NotImplemented;

{*********************************}
{ Pause procedure                 }
{*********************************}

procedure Pause;

{*********************************}
{ Abstract notification procedure }
{*********************************}

procedure Abstract(Caller : PChar);

{=====================================================================}

implementation

uses Strings, OWindows, WinTypes;

        {****************************************
        *
        *       methods for TActiveObject
        *
        ****************************************}

constructor TActiveObject.Init(NewName : PChar);
begin
  inherited Init(NewName);
  TheBehavior := nil;
  TheCondition := nil;
end;

destructor  TActiveObject.Done;
begin
  if assigned (TheBehavior)
    then dispose(TheBehavior, Done);
  if assigned (TheCondition)
    then dispose(TheCondition, Done);
  inherited Done;
end;

constructor TActiveObject.Load(var S : TStream);
begin
  inherited Load (S);
  TheCondition := PActiveObject (S.Get);
  TheBehavior := PBehavior (S.Get);
end;

procedure   TActiveObject.Store(var S : TStream);
begin
  inherited Store(S);
  S.Put(TheCondition);
```

```
  S.Put(TheBehavior);
end;

function    TActiveObject.Copy : Pointer;
var
  TheCopy :PActiveObject;
begin
  New(TheCopy, Init(NoName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure   TActiveObject.CopyFrom(Source : PActiveObject);
begin
   inherited Copy;
   SetCondition(Source^.GetCondition);
   SetBehavior(Source^.GetBehavior);
end;

function TActiveObject.EvalBooleans : Boolean;
begin
  EvalBooleans := false;
  Abstract('TActiveObject.EvalBooleans');
end;

procedure   TActiveObject.SetBehavior(NewBehavior : PBehavior);
begin
  TheBehavior := NewBehavior
end;

function    TActiveObject.GetBehavior : PBehavior;
begin
  GetBehavior := TheBehavior;
end;

procedure   TActiveObject.SetCondition(NewCondition : PActiveObject);
begin
  TheCondition := NewCondition;
end;

function   TActiveObject.GetCondition : PActiveObject;
begin
  GetCondition := TheCondition;
end;

        {****************************************
        *
        *       methods for TContextsList
        *
        ****************************************}

constructor TContextsList.Init;
begin
        inherited Init;
        OriginalContext := 0;
        AddedContexts := New(PLinkList, Init);
        DeletedContexts := New(PLinkList, Init);
        SwitchContext := 0;
end;

destructor  TContextsList.Done;
begin
        OriginalContext := 0;
        if assigned( AddedContexts ) then
           Dispose(AddedContexts, Done);
        if assigned( DeletedContexts ) then
           Dispose(DeletedContexts, Done);
        inherited Done;
end;

constructor TContextsList.Load(var S : TStream);
begin
        S.Read(OriginalContext, SizeOf(LongInt));
        AddedContexts := PLinkList(S.Get);
        DeletedContexts := PLinkList(S.Get);
        S.Read(SwitchContext, SizeOf(LongInt));
```

```
end;

procedure   TContextsList.Store(var S : TStream);
begin
        S.Write(OriginalContext, SizeOf(LongInt));
        if AddedContexts <> nil
    then S.Put(AddedContexts)
    else S.Put(nil);
        if DeletedContexts <> nil
    then S.Put(DeletedContexts)
    else S.Put(nil);
        S.Write(SwitchContext, SizeOf(LongInt));
end;

function  TContextsList.HermesDisplay : PChar;
begin
  Writeln('original context: ', OriginalContext);
  AddedContexts^.DisplayList('Added Contexts');
  DeletedContexts^.DisplayList('Deleted Contexts');
  Writeln('switch from current context to: ', SwitchContext);
  HermesDisplay := ' ';
end;

function TContextsList.Copy : Pointer;
var
  TheCopy : PContextsList;
begin
        New(TheCopy, Init);
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure TContextsList.CopyFrom(Source : PContextsList);
begin
  SetOriginalContext(Source^.GetOriginalContext);
  SetAddedContexts(Source^.GetAddedContexts^.Copy);
  SetDeletedContexts(Source^.GetDeletedContexts^.Copy);
  SetSwitchContext(Source^.GetSwitchContext);
end;

procedure   TContextsList.SetOriginalContext(NewContext : LongInt);
begin
  OriginalContext := NewContext;
end;

procedure   TContextsList.SetAddedContexts(NewList : PLinkList);
begin
  AddedContexts := NewList;
end;

procedure TContextsList.SetDeletedContexts(NewList : PLinkList);
begin
  DeletedContexts := NewList;
end;

procedure   TContextsList.SetSwitchContext(NewSwitch : LongInt);
begin
        SwitchContext := NewSwitch;
end;

function    TContextsList.GetOriginalContext : LongInt;
begin
        GetOriginalContext := OriginalContext;
end;

function    TContextsList.GetAddedContexts : PLinkList;
begin
        GetAddedContexts := AddedContexts;
end;

function    TContextsList.GetDeletedContexts : PLinkList;
begin
        GetDeletedContexts := DeletedContexts;
end;

function    TContextsList.GetSwitchContext : LongInt;
```

```
begin
        GetSwitchContext := SwitchContext;
end;

        {*************************************
        *
        *       methods for THermesPoint
        *
        ****************************************}

constructor THermesPoint.Init(NewX, NewY, NewZ : Coordinate);
begin
  TObject.Init;
  Coordinates['X'] := NewX;
  Coordinates['Y'] := NewY;
  Coordinates['Z'] := NewZ;
end;

destructor  THermesPoint.Done;
begin
  TObject.Done;
end;

constructor THermesPoint.Load(var S : TStream);
begin
  S.Read(Coordinates['X'], SizeOf(Coordinate));
  S.Read(Coordinates['Y'], SizeOf(Coordinate));
  S.Read(Coordinates['Z'], SizeOf(Coordinate));
end;

procedure   THermesPoint.Store(var S : TStream);
begin
  S.Write(Coordinates['X'], SizeOf(Coordinate));
  S.Write(Coordinates['Y'], SizeOf(Coordinate));
  S.Write(Coordinates['Z'], SizeOf(Coordinate));
end;

function              THermesPoint.Copy : Pointer;
var
  TheObject : PHermesPoint;
begin
  New(TheObject, Init(0, 0, 0));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

procedure             THermesPoint.CopyFrom(Source : PHermesPoint);
begin
  SetX(Source^.GetX);
  SetY(Source^.GetY);
  SetZ(Source^.GetZ);
end;

procedure   THermesPoint.DisplayList(Title : PChar);
begin
  Abstract('THermesPoint');
end;

function    THermesPoint.HermesDisplay : PChar;
begin
  Abstract('THermesPoint');
end;

function    THermesPoint.GetX : Coordinate;
begin
  GetX := Coordinates['X'];
end;

procedure   THermesPoint.SetX(NewX : Coordinate);
begin
  Coordinates['X'] := NewX;
end;

function    THermesPoint.GetY : Coordinate;
begin
  GetY := Coordinates['Y'];
```

```
end;

procedure   THermesPoint.SetY(NewY : Coordinate);
begin
  Coordinates['Y'] := NewY
end;

function    THermesPoint.GetZ : Coordinate;
begin
  GetZ := Coordinates['Z'];
end;

procedure   THermesPoint.SetZ(NewZ : Coordinate);
begin
  Coordinates['Z'] := GetZ;
end;

        {*********************************
  *
        *    methods for TExtent
        *
        *********************************}

constructor TExtent.Init(NewTop, NewBottom : PHermesPoint);
begin
  inherited Init;
  Top := NewTop;
  Bottom := NewBottom;
end;

destructor  TExtent.Done;
begin
  if assigned(Top)
    then Dispose(Top, Done);
  if assigned(Bottom)
    then dispose(Bottom, Done);
  inherited Done;
end;

constructor TExtent.Load(var S : TStream);
begin
  Top := PHermesPoint(S.Get);
  Bottom := PHermesPoint(S.Get);
end;

procedure   TExtent.Store(var S : TStream);
begin
  S.Put(Top);
  S.Put(Bottom);
end;

function    TExtent.HermesDisplay : PChar;
begin
  HermesDisplay := 'an extent';
end;

function    TExtent.Copy : Pointer;
var
  TheCopy :PExtent;
begin
  New(TheCopy, Init(Top, Bottom));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure   TExtent.CopyFrom(Source : PExtent);
begin
  SetTop(Source^.GetTop);
  SetBottom(Source^.GetBottom);
end;

procedure   TExtent.SetTop(NewTop : PHermesPoint);
begin
  Top := NewTop;
end;
```

```
function    TExtent.GetTop : PHermesPoint;
begin
  GetTop := Top;
end;

procedure   TExtent.SetBottom(NewBottom : PHermesPoint);
begin
  Bottom := NewBottom;
end;

function    TExtent.GetBottom : PHermesPoint;
begin
  GetBottom := Bottom;
end;

        {*********************************
  *
        *    methods for TTransform
        *
        *********************************}

constructor TTransform.Init(NewFixed, NewDisplace, NewScale, NewRotate : PHermesPoint);
begin
  Fixed := NewFixed;
  Displace := NewDisplace;
  Scale := NewScale;
  Rotate := NewRotate;
end;

constructor TTransform.InitDefault;         {all null transforms}
begin
  Fixed := New(PHermesPoint, Init(0, 0, 0));
  Displace := New(PHermesPoint, Init(0, 0, 0));
  Scale := New(PHermesPoint, Init(0, 0, 0));
  Rotate := New(PHermesPoint, Init(0, 0, 0));
end;

destructor  TTransform.Done;
begin
  if assigned(Fixed) then dispose(Fixed, Done);
  if assigned(Displace) then dispose(Displace, Done);
  if assigned(Scale) then dispose(Scale, Done);
  if assigned(Rotate) then dispose(Rotate, Done);
end;

constructor TTransform.Load(var S : TStream);
begin
  Fixed := PHermesPoint(S.Get);
  Displace := PHermesPoint(S.Get);
  Scale := PHermesPoint(S.Get);
  Rotate := PHermesPoint(S.Get);
end;

procedure   TTransform.Store(var S : TStream);
begin
  S.Put(Fixed);
  S.Put(Displace);
  S.Put(Scale);
  S.Put(Rotate);
end;

function    TTransform.HermesDisplay : PChar;
begin
  HermesDisplay := 'a transform';
end;

function    TTransform.Copy : Pointer;
var
  TheCopy :PTransform;
begin
  New(TheCopy, Init(Fixed, Displace, Scale, Rotate));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure   TTransform.CopyFrom(Source : PTransform);
```

```
begin
  SetFixed(Source^.GetFixed);
  SetDisplace(Source^.GetDisplace);
  SetScale(Source^.GetScale);
  SetRotate(Source^.GetRotate);
end;

procedure   TTransform.SetFixed(NewFixed : PHermesPoint);
begin
  Fixed := NewFixed;
end;

function    TTransform.GetFixed : PHermesPoint;
begin
  GetFixed := Fixed;
end;

procedure   TTransform.SetDisplace(NewDisplace : PHermesPoint);
begin
  Displace := NewDisplace;
end;

function    TTransform.GetDisplace : PHermesPoint;
begin
  GetDisplace := Displace;
end;

procedure   TTransform.SetScale(NewScale : PHermesPoint);
begin
  Scale := NewScale;
end;

function    TTransform.GetScale : PHermesPoint;
begin
  GetScale := Scale;
end;

procedure   TTransform.SetRotate(NewRotate : PHermesPoint);
begin
  Rotate := NewRotate;
end;

function    TTransform.GetRotate : PHermesPoint;
begin
  GetRotate := Rotate;
end;

        {***************************************
        *
        *       methods for TQueryListItem
        *
        ****************************************}

constructor TQueryListItem.Init;
begin
  inherited Init;
  DataId := 0;
(*  Context := HermesApp.GetContext;     *)
  Level := 0;
  Sequence := 0;
  Heading := nil;
  Transformation := nil;
end;

constructor TQueryListItem.Load(var S : TStream);
begin
  S.Read(DataId, SizeOf(LongInt));
  S.Read(Context, SizeOf(LongInt));
  S.Read(Level, SizeOf(Integer));
  S.Read(Sequence, SizeOf(Integer));
  Heading := S.StrRead;
  Transformation := PTransform(S.Get);
end;

procedure   TQueryListItem.Store(var S : TStream);
begin
```

```
  S.Write(DataId, SizeOf(LongInt));
  S.Write(Context, SizeOf(LongInt));
  S.Write(Level, SizeOf(Integer));
  S.Write(Sequence, SizeOf(Integer));
  S.StrWrite(Heading);
  if Transformation <> nil
    then S.Put(Transformation)
    else S.Put(nil);
end;

function  TQueryListItem.HermesDisplay : PChar;
begin
  { HermesDisplay := GetDataId^.HermesDisplay;  }
end;

function TQueryListItem.Copy : Pointer;
var TheCopy : PQueryListItem;
begin
       New(TheCopy, Init);
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure TQueryListItem.CopyFrom(Source : PQueryListItem);
begin
  SetDataId(Source^.GetDataId);
  SetContext(Source^.GetContext);
  SetLevel(Source^.GetLevel);
  SetSequence(Source^.GetSequence);
  SetHeading(Source^.GetHeading);
  SetTransformation(Source^.GetTransformation);
end;

procedure   TQueryListItem.SetDataId(NewDataId : LongInt);
begin
  DataId := NewDataId;
end;

procedure   TQueryListItem.SetContext(NewContext : LongInt);
begin
  Context := NewContext;
end;

procedure   TQueryListItem.SetLevel(NewLevel : Integer);
begin
  Level := NewLevel;
end;

procedure   TQueryListItem.SetSequence(NewSequence : Integer);
begin
  Sequence := NewSequence;
end;

procedure   TQueryListItem.SetHeading(NewHeading : PChar);
begin
  Heading := NewHeading;
end;

procedure   TQueryListItem.SetTransformation(NewTransformation     : PTransform);
begin
  Transformation := NewTransformation;
end;

function    TQueryListItem.GetDataId : LongInt;
begin
  GetDataId := DataId;
end;

function    TQueryListItem.GetContext : LongInt;
begin
  GetContext := Context;
end;

function    TQueryListItem.GetLevel : Integer;
begin
  GetLevel := Level;
```

```
end;

function    TQueryListItem.GetSequence : Integer;
begin
  GetSequence := Sequence;
end;

function    TQueryListItem.GetHeading : PChar;
begin
  GetHeading := Heading;
end;

function    TQueryListItem.GetTransformation : PTransform;
begin
  GetTransformation := Transformation;
end;

        {**********************************
  *
        *    methods for TAttribute
        *
        **********************************}

        {****************************************
        *
        *    methods for TTextualAttribute
        *
        ****************************************}

        {****************************************
        *
        *    methods for TGraphicalAttribute
        *
        ****************************************}

        {****************************************
        *
        *       methods for TSublink
        *
        ****************************************}

constructor TSublink.Init;
 begin
   inherited Init(NoName);
   ListofContexts := New(PContextsList, Init);
   TheAttribute := nil;
 end;

destructor TSublink.Done;
begin
  if assigned( ListofContexts ) then
     Dispose(ListofContexts, Done);
  if assigned( TheAttribute ) then
     Dispose(TheAttribute, Done);
  inherited Done;
end;

constructor TSublink.Load(var S : TStream);
begin
  inherited Load(S);
  ListofContexts := PContextsList(S.Get);
  TheAttribute := nil;
 { TheAttribute := PAttribute(S.Get);  }
end;

procedure   TSublink.Store(var S : TStream);
begin
  inherited Store(S);
  S.Put(ListofContexts);
  { S.Put(TheAttribute); }
end;

function  TSublink.HermesDisplay : PChar;
begin
  HermesDisplay := ListOfContexts^.HermesDisplay;
end;
```

```
function TSublink.Copy : Pointer;
var TheCopy : PSublink;
begin
  New(TheCopy, Init);
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure TSublink.CopyFrom(Source : PSublink);
begin
  inherited CopyFrom(Source);
  SetListofContexts( Source^.GetListofContexts^.Copy );
  SetAttribute(Source^.GetAttribute);
end;

procedure   TSublink.SetListofContexts(NewListofContext : PContextsList);
begin
  ListofContexts := NewListofContext;
end;

procedure   TSublink.SetAttribute(NewAttribute : PAttribute);
begin
  TheAttribute := NewAttribute;
end;

function    TSublink.GetListofContexts : PContextsList;
begin
  GetListofContexts := ListofContexts;
end;

function    TSublink.GetAttribute : PAttribute;
begin
  GetAttribute := TheAttribute;
end;

        {****************************************
        *
        *       methods for TGraphicalSublink
        *
        ****************************************}

constructor TGraphicalSublink.Init;
 begin
   inherited Init;
   ListofContexts := New(PContextsList, Init);
   TheTransformation := nil;
   TheAttribute := nil;
 end;

destructor TGraphicalSublink.Done;
begin
  if assigned( ListofContexts ) then
     Dispose(ListofContexts, Done);
  if assigned( TheTransformation ) then
     Dispose(TheTransformation, Done);
  if assigned( TheAttribute ) then
     Dispose(TheAttribute, Done);
  inherited Done;
end;

constructor TGraphicalSublink.Load(var S : TStream);
begin
  inherited Load(S);
  TheTransformation := PTransform(S.Get);
end;

procedure   TGraphicalSublink.Store(var S : TStream);
begin
  inherited Store(S);
  S.Put( TheTransformation ); { can be NIL }
end;

function  TGraphicalSublink.HermesDisplay : PChar;
begin
  HermesDisplay := ListOfContexts^.HermesDisplay;
```

```
end;

function TGraphicalSublink.Copy : Pointer;
var TheCopy : PGraphicalSublink;
begin
  New(TheCopy, Init);
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure TGraphicalSublink.CopyFrom(Source : PGraphicalSublink);
begin
  inherited CopyFrom(Source);
  SetTransformation(Source^.GetTransformation);
end;

procedure   TGraphicalSublink.SetTransformation(NewTransformation   : PTransform);
begin
  TheTransformation := NewTransformation;
end;

function    TGraphicalSublink.GetTransformation : PTransform;
begin
  GetTransformation := TheTransformation;
end;

        {*********************************}
        { StripBlanks function           }
        {*********************************}

function StripBlanks(ThePChar: PChar) : PChar;
{normalize a PChar to 40 lower case characters, no leading or trailing blanks}
var
  Index    : Word;
  TmpPChar : array[0..120] of Char;
  NewPChar : array[0..120] of Char;
begin  {StripBlanks}
  StripBlanks := nil;
  Index := 0;
  while ((ThePChar[Index] = ' ') and (Index < (StrLen(ThePChar) - 1))) do
    Inc(Index);
  StrLCopy(TmpPChar, ThePChar + Index, 40);    {strip blanks from front}
  Index := StrLen(TmpPChar) - 1;               {limit to 40 characters -- see HerDataB}
  while ((TmpPChar[Index] = ' ') and (Index > 0)) do
    Dec(Index);
  StrLCopy(NewPChar, TmpPChar, Index + 1);     {strip blanks from end}
  StrCopy(TmpPChar, StrLower(NewPChar));        {force lower case}
  StripBlanks := StrNew(TmpPChar);
(*  StrDispose(ThePChar);   *)
end;   {StripBlanks}

        {*********************************}
        { Pause procedure                }
        {*********************************}

procedure Pause;
begin
        Writeln('Please press the ENTER key to continue.');
        Readln;
end;  {Pause}

        {*********************************}
        { ErrorMsg procedure              }
        {*********************************}

procedure ErrorMsg(Severity: TErrorType; Message: PChar);
begin
case Severity of
               et_NOTICE  : MessageBox(Application^.MainWindow^.HWindow, Message, 'NOTICE', mb_Ok or
mb_IconInformation);

               et_WARNING : MessageBox(Application^.MainWindow^.HWindow, Message, 'WARNING', mb_Ok or
mb_IconExclamation);

               et_SERIOUS : MessageBox(Application^.MainWindow^.HWindow, Message, 'SERIOUS ERROR',
mb_Ok or mb_IconStop);
```

```
                  else      MessageBox(Application^.MainWindow^.HWindow, Message, 'ERROR', mb_Ok);
        end;
end;  {ErrorMsg}

procedure DB(TheNumber: LongInt);
var TheMessage  : array[0..40] of Char;
    StrNumber   : String;
    PCharNumber : array[0..40] of Char;
begin
  StrCopy(TheMessage, '***  DEBUGGING MESSAGE # ');
  Str(TheNumber, StrNumber);
  StrPCopy(PCharNumber, StrNumber);
  StrCat(TheMessage, PCharNumber);
  StrCat(TheMessage, '   ***');
  ErrorMsg(et_WARNING, TheMessage);
end;

        {*********************************}
        { NotImplemented procedure        }
        {*********************************}

procedure NotImplemented;
begin
  ErrorMsg(et_NOTICE, 'This feature has not yet been implemented.');
end;

  {*********************************}
        { Abstract notification procedure }
        {*********************************}

procedure Abstract(Caller : PChar);
        {report error because abstract object was referenced}
        {this object or method was only meant to be inherited}
begin
        ErrorMsg(et_Serious, StrCat('ERROR -- abstract procedure called: ', Caller));
        RunError(211);
end;

end.    {HerBasic}
```

4. HerLinks.pas define link objects

```
 {*****************************************
 *
 *      HerLinks.pas
 *
 *      version 2.0 -- Spring 1994
 *      (c) copyright 1994 by Gerry Stahl
 *      all rights reserved
 ******************************************}

unit HerLinks;

interface

uses Objects,
     HerLists, HerBasic, HerOGL2;

{ This unit defines the various kinds of links of the Hermes system
  as proposed in (Stahl, Ph.D. dissertation, 1993).

       DESIGN:

  The object hierarchy for Hermes is defined primarily in HerLists, HerBasic, HerLinks and HerNodes.

  In HerLists, the following part of the Hermes Substrate Object Hierarchy is defined,
  as described in (Stahl, Ph.D. dissertation, 1993):

    TObject                                             (in Turbo Pascal)
       |
    THermesObject                                       (in HerBasic)
       |
    THermesNamedObject
       |
    TActiveObject
       |                                                (in HerLinks)
       ---------------------------------
       |                                |
    PersistentObject                TSublink & TGraphicalSublink
       |
    TVCopyObject
       |
    TStampedObject
       ---------------------------------
       |                                |
    TPrimitiveObject               TLinkObject
       |                                |
    TNodeObject                   ----------------
       |                          |              |
    TPrimitiveObject           TMultiLink     TContextLink
       |                          |
    THermesNode           ----------------
       |                  |               |
    THermesGraphic     TLink          TContentLink
                                           |
                                       TGraphicLink


       A PersistentObject is derived from TActiveObject;
       Most other objects in this unit derive from it.
  An TActiveObject has a Condition, so any object can be conditional
  upon a Booleans language expression.
  It has a Behavior, so navigating to it can cause a procedure to be executed.
  Procedures for Behaviors can be defined in a user-defined DLL.

  Hermes is a hypermedia system consisting of multimedia Nodes, and Links connecting the Nodes.
  All information in Hermes is stored in NodeObjects, which are connected by InLinks and OutLinks.
  A LinkTree is a list of LinkLists, sorted by link Type.
  A LinkList is a list of LinkIds, sorted by ObjectId for fast search.
  NodeObjects include defined expressions in the Hermes language, defined LinkTypes, defined
  NodeKinds, and the contexts (perspectives) hierarchy.
  Regular Nodes have a required TNodeKind and an optional Name, as well as their hypermedia
  InLinks and OutLinks. Their content is in a separate TNodeObject connected by a ContentLink.
  The content TNodeObject may contain media primitives (Media Elements), language expressions
```

```
  (Language Elements or Terminology Elements) or composite vector graphics hierarchies
(GraphicsNode).

  Vector graphics is implemented as a hierarchy of HermesGraphic nodes, which maintain their
Transforms
  and Extents. Rather than having content linked by a ContentLink, these HermesGraphic nodes have
  constituent HermesGraphic nodes or Polylines linked by GraphicLinks.

  A Link connects a FromNode with a ToNode. A Link has a required TLinkType.
  If the Link is of Type "hermes_content" then the ToNode is a media
  primitive (e.g., a PChar, a Real, a Polyline, or the result of a query).

  A Link has a list of Sublinks. The SubLinks implement perspectives.
  The Link is only traversed if the CurrentContext matches the contexts
  in one of the Sublinks of the Link.
  A TSublink contains a list of contexts.
  A list of contexts includes the original context in which the TSublink was
  created, added contexts, deleted contexts, and a switch-from and a switch-to
  context for context switching.
  A TSublink may optionally contain a graphic transformation (fixed point,
  scale, rotation, displace), a set of Attributes (color, thickness, font, etc),
  a BooleansProposition for a conditional link, or a Pascal method behavior.

        By defining a list of SubLinks for a Link, the number of Links that
        need to be read in from memory is minimized.  Also, changes to information
        in the SubLinks can be made without reading/modifying/writing the Nodes
  (which may be large). Combining all links between two nodes is important
  because there may be very bushy trees of sublinks with virtual copying
  and for many functions, one needs to look at them all or at many of them.
  Sublinking also saves duplicating 3 LongInts: TLinkType, ToNode, FromNode.
  ALso, often one only wants to cross one TSublink of a Link (the first one)
  otherwise one would get multiple copies; this is efficiently done with a
  ForEach or ForFirst method on a list of Sublinks.

  Note that Links are used to connect Nodes in the
  Hyperdocument and to connect Nodes and their Data.  In both cases,
  Contexts in the SubLinks determine whether these Links are traversed --
  i.e., whether the Nodes are connected or whether they contain certain
  Data, respectively.  These Nodes and Links are also used for
  the Context tree itself, which determines relationships among the Contexts
  (but their Links all specify the HermesUniversalContext).

  All objects in Hermes -- LanguageElements (like conditionals, predicates and
  queries), MediaElements (CharacterStrings, NumericExpressions,
  BooleansPropositions, etc.) and hypertext elements (Nodes, Links, LinkTypes,
  NodeKinds, DataLists) -- are part of the same hyperdocument.  Therefore,
  any of them can be interconnected, annotated, dated, etc. thru Links.
  They can all be given Names and they are all stored on the same
  object Stream.

        SPECIFICATION:

        HyperMedia in Hermes consists of multimedia Nodes and context-specific Links.
  The structure of the Hypermedia is designed to support the following:
      Efficiency of disk access by separating nodes, links and data;
      Multimedia generality and polymorphism of data types;
      Virtual copying with context numbers;
      Composite nodes with multiple data contents;
      PHIGS graphics, with hierarchical structures;
      Virtual copies of graphic structures;
      Named objects, including virtual copies of them.
      Indexing of nodes, links and data to avoid unnecessary disk reads;
      Inheritance of named objects;
      Inheritance of subnetworks;
      Efficiency, simplicity, generality, elegance of design.
}


type

        {***************************************
        *
        *       data structure for PersistentObject
  *
        *       This is the base for all objects stored
  *       on a HermesStream. Any TObject descendent
```

```
  *       can be stored as part of a PersistentObject,
  *       but only PersistentObjects can be retrieved
  *       from the stream. A PersistentObject has a required
  *       ObjectId for direct random access
  *       and an ObjectType for listing by ObjectType
  *       (each major type of PersistentObject has a
  *       single character ObjectType defined by the Language;
  *       the defined ObjectTypes are listed below)
        ****************************************}

        PPersistentObject = ^TPersistentObject;
        TPersistentObject = object (TActiveObject)
                constructor Init(NewName : PChar);
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    Copy : Pointer; virtual CopyDMT;
                procedure   CopyFrom(Source : PPersistentObject);
    function    DisplayContent : PPicture;  virtual DisplayContentDMT;     {display graphical
contents}
                procedure   SetId(NewObjectId : LongInt);
                function    GetId : LongInt;
                procedure   SetType(NewObjectType : LangType);
                function    GetType : LangType;
        private
                ObjectId   : LongInt;             {indicates location on index to stream}
                ObjectType : LangType;                {specifies 1-character LanguageType from
HerLists}
        end;

  {****************************************
        *
        *    data structure for TVCopyObject
        *
        ****************************************}

        PVCopyObject = ^TVCopyObject;
        TVCopyObject = object (TPersistentObject)
        end;

        {****************************************
        *
        *    data structure for TStampedObject
        *
        ****************************************}

        PStampedObject = ^TStampedObject;
        TStampedObject = object (TVCopyObject)
                constructor Init(NewName : PChar);
                destructor  Done; virtual;
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PStampedObject);
                procedure   SetCreator(NewCreator : PChar);
                procedure   SetDate(NewDate   : LongInt);
                procedure   SetModified(NewModified   : LongInt);
                function    GetCreator : PChar;
                function    GetDate : LongInt;
                function    GetModified : LongInt;
        private
                Creator        : PChar;      {name of person who created this node}
                Date      : LongInt;    {date and time node was created}
                Modified  : LongInt;    {date and time node was last modified}
        end;                      {use PackTime and UnPackTime from WinDOs unit}

        {****************************************
        *
        *    data structure for TLinkObject
        *
        ****************************************}

        PLinkObject = ^TLinkObject;
        TLinkObject = object (TPersistentObject)
                constructor Init;
                destructor  Done; virtual;
                constructor Load(var S : TStream);
```

```
            procedure   Store(var S : TStream); virtual StoreDMT;
            function    Copy : Pointer; virtual CopyDMT;
  procedure   CopyFrom(Source : PLinkObject);
            procedure   SetFromPO(NewFromPO : LongInt);
            procedure   SetToPO(NewToPO   : LongInt);
            function    GetFromPO : LongInt;
            function    GetToPO : LongInt;
private
            FromPO                : LongInt;          {an ObjectId}
            ToPO                  : LongInt;          {an ObjectId}
      end;

      {****************************************
      *
      *    data structure for TContextLink
      *
      ****************************************}

      PContextLink = ^TContextLink;
TContextLink = object ( TLinkObject )
            constructor Init;
      end;

      {****************************************
      *
      *    data structure for TMultiLink
      *
      ****************************************}

      PMultiLink = ^TMultiLink;
      TMultiLink = object (TLinkObject)
            constructor Init;
            destructor  Done; virtual;
            constructor Load(var S : TStream);
            procedure   Store(var S : TStream); virtual StoreDMT;
            function    Copy : Pointer; virtual CopyDMT;
  procedure   CopyFrom(Source : PMultiLink);
            procedure   SetSubLinks(NewSubLinks : PHermesItemsList);
            function    GetSubLinks : PHermesItemsList;
      private
            SubLinks              : PHermesItemsList;      {a list of SubLinks}
      end;

      {****************************************
      *
      *    data structure for TContentLink
      *
      ****************************************}

      PContentLink = ^TContentLink;
       TContentLink = object (TMultiLink)
            constructor Init;
      end;

      {****************************************
      *
      *    data structure for GTraphicLink
      *
      ****************************************}

      PGraphicLink = ^TGraphicLink;
      TGraphicLink = object (TContentLink)
            constructor Init;
      end;

      {****************************************
      *
      *    data structure for TLink
      *
      ****************************************}

      PLink = ^TLink;
      TLink = object (TMultiLink)
            constructor Init;
            destructor  Done; virtual;
            constructor Load(var S : TStream);
```

```
              procedure   Store(var S : TStream); virtual StoreDMT;
              function    Copy : Pointer; virtual CopyDMT;
    procedure CopyFrom(Source : PLink);
              procedure   SetTypeofLink(NewTypeofLink : PChar);
              function    GetTypeofLink : PChar;
       private
              TypeofLink     : PChar;            {a TLinkType name}
       end;

{=====================================================================}

implementation

uses Strings;

       {***************************************
       *
       *       methods for PersistentObject
       *
       ****************************************}

constructor TPersistentObject.Init(NewName : PChar);
 begin
   inherited Init(NewName);
   ObjectId    := 0;
   SetType( ot_undefined );        {abstract ObjectType}
 end;

constructor TPersistentObject.Load(var S : TStream);
 begin
   inherited Load(S);
   S.Read(ObjectId, SizeOf(LongInt));
   S.Read(ObjectType, SizeOf(Char));
 end;

procedure   TPersistentObject.Store(var S : TStream);
 begin
   inherited Store(S);
   S.Write(ObjectId, SizeOf(LongInt));
   S.Write(ObjectType, SizeOf(Char));
 end;

function   TPersistentObject.Copy : Pointer;
 var TheObject : PPersistentObject;
 begin
   New(TheObject, Init(NoName));
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
 end;

procedure   TPersistentObject.CopyFrom(Source : PPersistentObject);
 begin
   inherited CopyFrom(Source);
   SetId(Source^.GetId);
   SetType(Source^.GetType);
 end;

function    TPersistentObject.DisplayContent : PPicture; {display graphical contents}
 begin
   Abstract('PersistentObject:DisplayContent');
 end;

procedure   TPersistentObject.SetId(NewObjectId : LongInt);
 begin
   ObjectId := NewObjectId;
 end;

function    TPersistentObject.GetId : LongInt;
 begin
   GetId := ObjectId;
 end;

procedure   TPersistentObject.SetType(NewObjectType : LangType);
 begin
   ObjectType := NewObjectType;
 end;
```

```
function TPersistentObject.GetType : LangType;
 begin
   GetType := ObjectType;
 end;

        {***************************************
        *
        *       methods for TVCopyObject
        *
        ****************************************}

        {***************************************
        *
        *       methods for TStampedObject
        *
        ****************************************}

constructor TStampedObject.Init(NewName : PChar);
 begin
   inherited Init(NewName);
   Creator := nil;
   Date := 0;
   Modified := 0;
 end;

destructor TStampedObject.Done;
 begin
   if assigned(Creator) then StrDispose(Creator);
   Date := 0;
   Modified := 0;
   inherited Done;
 end;

constructor TStampedObject.Load(var S : TStream);
begin
  inherited Load(S);
  Creator := S.StrRead;
  S.Read(Date, SizeOf(LongInt));
  S.Read(Modified, SizeOf(LongInt));
end;

procedure   TStampedObject.Store(var S : TStream);
begin
  inherited Store(S);
  S.StrWrite(Creator);
  S.Write(Date, SizeOf(LongInt));
  S.Write(Modified, SizeOf(LongInt));
end;

function    TStampedObject.Copy : Pointer;
var
  TheCopy : PStampedObject;
begin
  New(TheCopy, Init(NoName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure   TStampedObject.CopyFrom(Source : PStampedObject);
begin
  inherited CopyFrom(Source);
  SetCreator(Source^.GetCreator);
  SetDate(Source^.GetDate);
  SetModified(Source^.GetModified);
end;

procedure   TStampedObject.SetCreator(NewCreator : PChar);
begin
  Creator := StrNew(NewCreator);
end;

procedure   TStampedObject.SetDate(NewDate   : LongInt);
begin
  Date := NewDate;
end;
```

```
procedure   TStampedObject.SetModified(NewModified   : LongInt);
begin
  Modified := NewModified;
end;

function    TStampedObject.GetCreator : PChar;
begin
  GetCreator := Creator;
end;

function    TStampedObject.GetDate : LongInt;
begin
  GetDate := Date;
end;

function    TStampedObject.GetModified : LongInt;
begin
  GetModified := Modified;
end;

        {****************************************
        *
        *       methods for TLinkObject
        *
        ****************************************}

constructor TLinkObject.Init;
begin
  inherited Init(NoName);
  FromPO := 0;
  ToPO := 0;
end;

destructor  TLinkObject.Done;
begin
  FromPO := 0;
  ToPO := 0;
  inherited Done;
end;

constructor TLinkObject.Load(var S : TStream);
begin
  inherited Load(S);
  S.Read(FromPO, SizeOf(LongInt));
  S.Read(ToPO, SizeOf(LongInt));
end;

procedure   TLinkObject.Store(var S : TStream);
begin
  inherited Store(S);
  S.Write(FromPO, SizeOf(LongInt));
  S.Write(ToPO, SizeOf(LongInt));
end;

function    TLinkObject.Copy : Pointer;
var
  TheCopy : PLinkObject;
begin
  New(TheCopy, Init);
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure   TLinkObject.CopyFrom(Source : PLinkObject);
begin
  inherited CopyFrom(Source);
  SetFromPO(Source^.GetFromPO);
  SetToPO(Source^.GetToPO);
end;

procedure   TLinkObject.SetFromPO(NewFromPO : LongInt);
begin
  FromPO := NewFromPO;
end;
```

```
procedure   TLinkObject.SetToPO(NewToPO      : LongInt);
begin
  ToPO := NewToPO;
end;

function    TLinkObject.GetFromPO : LongInt;
begin
  GetFromPO := FromPO;
end;

function    TLinkObject.GetToPO : LongInt;
begin
  GetToPO := ToPO;
end;


        {****************************************
        *
        *       methods for TContextLink
        *
        ****************************************}

constructor TContextLink.Init;
begin
  inherited Init;
  SetType( ot_ContextLink );
end;


        {****************************************
        *
        *       methods for TMultiLink
        *
        ****************************************}

constructor TMultiLink.Init;
begin
  inherited Init;
  SubLinks := New(PHermesItemsList, Init);
end;

destructor  TMultiLink.Done;
begin
  if assigned(SubLinks) then dispose(SubLinks, Done);
  inherited Done;
end;

constructor TMultiLink.Load(var S : TStream);
begin
  inherited Load(S);
  SubLinks := PHermesItemsList(S.Get);
end;

procedure   TMultiLink.Store(var S : TStream);
begin
  inherited Store(S);
  S.Put(SubLinks);
end;

function    TMultiLink.Copy : Pointer;
var
  TheCopy : PMultiLink;
begin
  New(TheCopy, Init);
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure   TMultiLink.CopyFrom(Source : PMultiLink);
begin
  inherited CopyFrom(Source);
  SetSubLinks(Source^.GetSubLinks);
end;

procedure   TMultiLink.SetSubLinks(NewSubLinks : PHermesItemsList);
begin
```

```
  SubLinks := NewSubLinks;
end;

function    TMultiLink.GetSubLinks : PHermesItemsList;
begin
  GetSubLinks := SubLinks;
end;

  {****************************************
        *
        *       methods for ContentLink
        *
        ****************************************}

constructor TContentLink.Init;
begin
  inherited Init;
  SetType(  ot_ContentLink );
end;


        {****************************************
        *
        *       methods for GraphicLink
        *
        ****************************************}

constructor TGraphicLink.Init;
begin
  inherited Init;
  SetType( ot_GraphicLink);
end;

  {****************************************
        *
        *       methods for Link
        *
        ****************************************}

constructor TLink.Init;
begin
  inherited Init;
  SetType( ot_Link );
  TypeofLink := NoName;
end;

destructor  TLink.Done;
begin
  StrDispose(TypeOfLink);
  inherited Done;
end;

constructor TLink.Load(var S : TStream);
begin
  inherited Load(S);
  TypeofLink := S.StrRead;
end;

procedure   TLink.Store(var S : TStream);
begin
  inherited Store(S);
  S.StrWrite(TypeofLink);
end;

function TLink.Copy : Pointer;
var TheCopy : PLink;
begin
  New(TheCopy, Init);
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure TLink.CopyFrom(Source : PLink);
begin
  inherited CopyFrom(Source);
  SetTypeofLink(Source^.GetTypeofLink);
```

```
end;

procedure   TLink.SetTypeofLink(NewTypeofLink : PChar);
begin
  TypeofLink := StrNew(NewTypeofLink);
end;

function    TLink.GetTypeofLink : PChar;
begin
  GetTypeofLink := TypeofLink;
end;

end.      {**** unit HerLinks  ****}
```

5. HerNodes.pas define node objects

```
 {*****************************************
 *
 *               HerNodes.pas
 *
 *      version 2.0 -- Spring 1994
 *      (c) copyright 1994 by Gerry Stahl
 *      all rights reserved
 ******************************************}

unit HerNodes;

interface

uses Objects,
     HerLists, HerBasic, HerLinks, HerOGL2;

{ This unit defines  components of the Hermes version of hypermedia
  as proposed in (Stahl, Ph.D. dissertation, 1993).

  It completes the Hermes Substrate Object Hierarchy outlined and begun
  in HerLinks. Successive objects add hypermedia links out and in to nodes,
  and content links out and in to nodes.

  The following objects are defined:

  TPrimitiveObject
  TNodeObject
  TPrimitiveNode
  THermesNode
  THermesGraphic

  TGraphicContent : TUnitCube, THermesPolyline, TSweep
  TContext
  TNodeKind
  TLinkTYpe
  TLogin
  TResultList
  (TNodeContent is defined in HerHyper)
}

type

        {***************************************
        *
        *    data structure for TPrimitiveObject
        *    has hypermedia out-links
        ****************************************}

        PPrimitiveObject = ^TPrimitiveObject;
        TPrimitiveObject = object (TStampedObject)
                constructor Init(NewName : PChar);
                destructor  Done; virtual;
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PPrimitiveObject);
                procedure   SetOutLinks(NewOutLinks   : PLinkTree);
                function    GetOutLinks : PLinkTree;
    function    GetOutLinksOfType(TLinkType: PChar) : PLinkList;
    procedure   AddOutLink(TLinkType: PChar; LinkName: LongInt);
        private
                OutLinks : PLinkTree;      {list of lists of outgoing links}
        end;

        {***************************************
        *
        *    data structure for TNodeObject
        *    has hypermedia in-links
        ****************************************}

        PNodeObject = ^TNodeObject;
        TNodeObject = object (TPrimitiveObject)
```

```
               constructor Init(NewName : PChar);
               destructor  Done; virtual;
               constructor Load(var S : TStream);
               procedure   Store(var S : TStream); virtual StoreDMT;
               function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PNodeObject);
    function    DisplayContent : PPicture;  virtual DisplayContentDMT;     {display graphical
contents}
               procedure   SetInLinks(NewInLinks : PLinkTree);
               function    GetInLinks : PLinkTree;
    function    GetInLinksOfType(LinkType: PChar) : PLinkList;
    procedure   AddInLink(LinkType: PChar; LinkName: LongInt);
       private
               InLinks : PLinkTree;      {list of lists of incoming links}
       end;

       {****************************************
       *
       *    data structure for TPrimitiveNode
       *
       ****************************************}

       PPrimitiveNode = ^TPrimitiveNode;
       TPrimitiveNode = object (TNodeObject)
               constructor Init(NewName : PChar);
               destructor  Done; virtual;
               constructor Load(var S : TStream);
               procedure   Store(var S : TStream); virtual StoreDMT;
               function    HermesDisplay : PChar; virtual HermesDisplayDMT;
               function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PPrimitiveNode);
               procedure   SetContentOutLinks(NewContentLinks      : PLinkList);
               function    GetContentOutLinks : PLinkList;
    procedure   AddContentOutLink(LinkName: LongInt);
    procedure   AddContentNode(TheNode : PNodeObject; TheTransform : PTransform);
       private
               ContentOutLinks  : PLinkList;      {list of content out-links}
       end;

       {****************************************
       *
       *    data structure for THermesNode
       *    has content-in link and NodeKind
       ****************************************}

       PHermesNode = ^THermesNode;
       THermesNode = object (TPrimitiveNode)
               constructor Init(NewName : PChar);
               destructor  Done; virtual;
               constructor Load(var S : TStream);
               procedure   Store(var S : TStream); virtual StoreDMT;
    function    Eval(InputList : PHermesQueryList) : PHermesQueryList; virtual EvalDMT;
               function    HermesDisplay : PChar; virtual HermesDisplayDMT;
    function    DisplayContent : PPicture;  virtual DisplayContentDMT;     {display graphical
contents}
               function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PHermesNode);
               procedure   SetKind(NewKind : PChar);
               procedure   SetContentInLinks(NewContentLinks       : PLinkList);
               function    GetKind : PChar;
               function    GetContentInLinks : PLinkList;
    procedure   AddContentInLink(LinkName: LongInt);
       private
               Kind                        : PChar;          {Name of THermesNodeKind}
               ContentInLinks  : PLinkList;      {list of content links}
       end;

       {****************************************
       *
       *    data structure for THermesGraphic
       *
       ****************************************}

       PHermesGraphic = ^THermesGraphic;
       THermesGraphic = object (THermesNode)
               constructor Init(NewName : PChar);
```

```
  end;

        {****************************************
        *
        *    data structure for TPrivilegeObject
        *    stores View, Annotate, Edit privilege values
  *    and password
        ****************************************}

  Privilege = 1 .. 99;     {range of valid privilege levels}

const

  DefaultPrivilege = 20;    {all privileges for Logins and Contexts}
                            {are initialized to this default level}
type

        PPrivilegeObject = ^TPrivilegeObject;
        TPrivilegeObject = object (TNodeObject)
                constructor Init(NewName : PChar);
                destructor  Done; virtual;
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
                function    Copy : Pointer;         virtual CopyDMT;
    procedure   CopyFrom(Source : PPrivilegeObject);
                procedure   SetView(NewView : Privilege);
                procedure   SetAnnotate(NewAnnotate   : Privilege);
                procedure   SetEdit(NewEdit   : Privilege);
                procedure   SetPassword(NewPassword : PChar);
                function    GetView     : Privilege;
                function    GetAnnotate : Privilege;
                function    GetEdit     : Privilege;
                function    GetPassword : PChar;
  private
    View     : Privilege;
    Annotate : Privilege;
    Edit     : Privilege;
    Password : PChar;
  end;

        {****************************************
        *
        *    data structure for TLogin
        *
        ****************************************}

        PLogin = ^TLogin;
        TLogin = object (TPrivilegeObject)
                constructor Init(NewName : PChar);
  end;

        {****************************************
        *
        *    data structure for Context
        *    this is a node for the contexts hierarchy
  *    it has no ContentLinks or TNodeKind
        ****************************************}

        PContext = ^TContext;
  TContext = object (TPrivilegeObject)
                constructor Init(NewName : PChar);
                destructor  Done; virtual;
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
                function    Copy : Pointer;         virtual CopyDMT;
    procedure   CopyFrom(Source : PContext);
                procedure   SetColor(NewColor : PChar);
                function    GetColor : PChar;
                procedure   SetVEdit(NewVEdit : Boolean);
                function    GetVEdit : Boolean;
  private
    Color     : PChar;
    VEdit     : Boolean;
        end;
```

```
      {****************************************
      *
      *    data structure for TNodeKind
      *
      ****************************************}

      PNodeKind = ^TNodeKind;
  TNodeKind = object (TNodeObject)
    constructor Init(NewName : PChar);
                destructor  Done; virtual;
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PNodeKind);
    procedure   SetInstanceCount(NewInstanceCount : Integer);
    function    GetInstanceCount : Integer;
    procedure   DecInstanceCount;
    procedure   IncInstanceCount;
   private
    InstanceCount   : Integer;           {how many Nodes of this Kind exist}
      end;

      {****************************************
      *
      *    data structure for TLinkType
      *
      ****************************************}

      PLinkType = ^TLinkType;
      TLinkType = object (TNodeObject)        {a main link type}
                constructor Init(NewName : PChar);           {may have synonyms}
                destructor  Done; virtual;
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PLinkType);
    procedure   SetInstanceCount(NewInstanceCount : Integer);
    function    GetInstanceCount : Integer;
    procedure   DecInstanceCount;
    procedure   IncInstanceCount;
         private
    InstanceCount   : Integer;      {how many Links of this Type exist}
      end;

      {****************************************
      *
      *    data structure for ResultList
      *
      ****************************************}

      PResultList = ^TResultList;
      TResultList = object (TNodeObject)
                constructor Init(NewName : PChar);
                destructor  Done; virtual;
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PResultList);
    procedure   SetListOfData(NewListOfData : PHermesItemsList);
    function    GetListOfData : PHermesItemsList;
   private
    ListOfData : PHermesItemsList;
      end;

      {****************************************
      *
      *    data structure for TGraphicContent
      *
      ****************************************}

      PGraphicContent = ^TGraphicContent;
      TGraphicContent = object ( TNodeObject )
      end;

      {****************************************
```

```
        *
        *    data structure for TUnitCube
        *
        ****************************************}

        PUnitCube = ^TUnitCube;
        TUnitCube = object ( TGraphicContent )
                constructor Init(NewName : PChar);
        end;

        {****************************************
        *
        *    data structure for THermesPolyline
        *
        ****************************************}

        PHermesPolyline = ^THermesPolyline;
        THermesPolyline = object ( TGraphicContent )
                constructor Init(NewName : PChar);
                destructor  Done; virtual;
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
                function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PHermesPolyline);
                procedure   SetListofPoints( NewListofPoints : PHermesPointList );
                function    GetListofPoints : PHermesPointList;
        private
                ListofPoints : PHermesPointList;
        end;

  {****************************************
        *
        *    data structure for TSweep
        *
        ****************************************}

        PSweep = ^TSweep;
        TSweep = object ( TGraphicContent )
                constructor Init(NewName : PChar);
        end;

{=====================================================================}

implementation

uses Strings,
     HerDataB, HerWorld, HerPersp, HerOGL3;

        {****************************************
        *
        *       methods for TPrimitiveObject
        *
        ****************************************}

constructor TPrimitiveObject.Init(NewName : PChar);
begin
  inherited Init(NewName);
  OutLinks := New(PLinkTree, Init);
end;

destructor  TPrimitiveObject.Done;
begin
  if assigned(OutLinks) then dispose(OutLinks, Done);
  inherited Done;
end;

constructor TPrimitiveObject.Load(var S : TStream);
begin
  inherited Load(S);
  OutLinks := PLinkTree(S.Get);
end;

procedure   TPrimitiveObject.Store(var S : TStream);
begin
  inherited Store(S);
  S.Put(OutLinks);
```

```
end;

function    TPrimitiveObject.Copy : Pointer;
var
  TheCopy : PPrimitiveObject;
begin
  New(TheCopy, Init(NoName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure   TPrimitiveObject.CopyFrom(Source : PPrimitiveObject);
begin
  inherited CopyFrom(Source);
  SetOutLinks(Source^.GetOutLinks^.Copy);
end;

procedure   TPrimitiveObject.SetOutLinks(NewOutLinks : PLinkTree);
begin
  OutLinks := NewOutLinks;
end;

function    TPrimitiveObject.GetOutLinks : PLinkTree;
begin
  GetOutLinks := OutLinks;
end;

function    TPrimitiveObject.GetOutLinksOfType(TLinkType: PChar) : PLinkList;
var
  TheList  : PLinkList;
  TheIndex : Integer;
  MainType : PChar;
begin
  TheList := New(PLinkList, Init);
  TheIndex := DBGetSynonymId(TLinkType, MainType, 'T');     {LinkLists named by main synonym}
  if (TheIndex = -1)
    then StrCopy(MainType, TLinkType);
  TheList^.SetName(MainType);                               {create empty List of TLinkType}
  TheIndex := -1;
  if OutLinks^.Search(TheList, TheIndex)                    {check if it exists}
    then  TheList := OutLinks^.At(TheIndex);
  GetOutLinksOfType := TheList;
end;

procedure   TPrimitiveObject.AddOutLink(TLinkType: PChar; LinkName : LongInt);
var
  TheList  : PLinkList;
  TheIndex : Integer;
  MainType : PChar;
begin
  TheList := New(PLinkList, Init);
TheIndex := DBGetSynonymId(TLinkType, MainType, 'T');
  if (TheIndex = -1)
    then StrCopy(MainType, TLinkType);
  TheList^.SetName(MainType);                   {create test List of TLinkType}
  TheIndex := -1;
  if OutLinks^.Search(TheList, TheIndex)        {check if it already exists}
    then
      begin
        Dispose(TheList, Done);
        TheList := OutLinks^.At(TheIndex);        {then ptr to it}
        TheList^.Insert(New(PLinkId, Init(LinkName)));     {add link to sublist}
      end
    else
      begin
        TheList^.Insert(New(PLinkId, Init(LinkName)));     {add link to sublist}
        OutLinks^.Insert(TheList);
      end;
end;

        {***************************************
        *
        *       methods for TNodeObject
        *
        ****************************************}
```

```
constructor TNodeObject.Init(NewName : PChar);
begin
  inherited Init(NewName);
  InLinks := New(PLinkTree, Init);
end;

destructor  TNodeObject.Done;
begin
  if assigned(InLinks) then dispose(InLinks, Done);
  inherited Done;
end;

constructor TNodeObject.Load(var S : TStream);
begin
  inherited Load(S);
  InLinks := PLinkTree(S.Get);
end;

procedure   TNodeObject.Store(var S : TStream);
begin
  inherited Store(S);
  S.Put(InLinks);
end;

function    TNodeObject.Copy : Pointer;
var
  TheCopy : PNodeObject;
begin
  New(TheCopy, Init(NoName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure   TNodeObject.CopyFrom(Source : PNodeObject);
begin
  inherited CopyFrom(Source);
  SetInLinks(Source^.GetInLinks);
end;

function   TNodeObject.DisplayContent : PPicture;
{□efine a default graphical content display to display the object's name}
var
  TheLabel    : PLabel;
  TheRichText : PRichText;
begin
  TheLabel := New(PLabel, InitDefault);
  TheLabel^.SetText(GetName);
  TheRichText := New(PRichText, InitDefault);
  TheRichText^.Add(TheLabel);
  DisplayContent := TheRichText;
end;

procedure   TNodeObject.SetInLinks(NewInLinks : PLinkTree);
begin
  InLinks := NewInLinks;
end;

function    TNodeObject.GetInLinks : PLinkTree;
begin
  GetInLinks := InLinks;
end;

function    TNodeObject.GetInLinksOfType(LinkType: PChar) : PLinkList;
var
  TheList  : PLinkList;
  TheIndex : Integer;
  MainType : PChar;
begin
  TheList := New(PLinkList, Init);
  TheIndex := DBGetSynonymId(LinkType, MainType, 'T');     {LinkLists named by main synonym}
  if (TheIndex = -1)
    then StrCopy(MainType, LinkType);
  TheList^.SetName(MainType);                              {create empty List of TLinkType}
  TheIndex := -1;
  if InLinks^.Search(TheList, TheIndex)                   {check if it exists}
    then  TheList := OutLinks^.At(TheIndex);
```

```
  GetInLinksOfType := TheList;
end;

procedure   TNodeObject.AddInLink(LinkType: PChar; LinkName : LongInt);
var
  TheList  : PLinkList;
  TheIndex : Integer;
  MainType : PChar;
begin
  TheList := New(PLinkList, Init);
  MainType := NoName;
  TheIndex := DBGetSynonymId(LinkType, MainType, ot_LinkType);
  if (TheIndex = -1)
    then StrCopy(MainType, LinkType);
  TheList^.SetName(MainType);                   {create test List of TLinkType}
  TheIndex := -1;
  if InLinks^.Search(TheList, TheIndex)        {check if it already exists}
    then
      begin
        Dispose(TheList, Done);                  {if sublist exists}
        TheList := InLinks^.At(TheIndex);        {then ptr to it}
      end
    else
      begin
        InLinks^.Insert(TheList);                 {else create it}
      end;
  TheList^.Insert(New(PLinkId, Init(LinkName)));    {add link to sublist}
end;

       {****************************************
       *
       *       methods for TPrimitiveNode
       *
       ****************************************}

constructor TPrimitiveNode.Init(NewName : PChar);
 begin
   inherited Init(NewName);
   ContentOutLinks := New(PLinkList, Init);
   ContentOutLinks^.SetName('hermes_content_out');
 end;

destructor TPrimitiveNode.Done;
begin
  if assigned( ContentOutLinks )  then
    Dispose(ContentOutLinks, Done);
  inherited Done;
end;

constructor TPrimitiveNode.Load(var S : TStream);
begin
       inherited Load(S);
  ContentOutLinks := PLinkList(S.Get);
end;

procedure TPrimitiveNode.Store(var S : TStream);
begin
  inherited Store(S);
  S.Put(ContentOutLinks);
end;

function  TPrimitiveNode.HermesDisplay : PChar;
{to HermesDisplay a TPrimitiveNode, display its definition and its Contents}
var DisplayArray : array[0..2000] of Char;
   procedure DisplayIt(ANode : PPersistentObject); far;
    begin  {DisplayIt}
      StrCat(DisplayArray, ANode^.HermesDisplay);
      StrCat(DisplayArray, '   --   ');
    end;   {DisplayIt}
begin
  StrCopy(DisplayArray, 'contents = ');
  GetContentOutLinks^.ForEach(@DisplayIt);   {display each Content}
  HermesDisplay := StrNew(DisplayArray);
end;

function TPrimitiveNode.Copy : Pointer;
```

```
var TheCopy : PPrimitiveNode;
begin
  New(TheCopy, Init(NoName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure TPrimitiveNode.CopyFrom(Source : PPrimitiveNode);
begin
  inherited CopyFrom(Source);
  SetContentOutLinks( Source^.GetContentOutLinks^.Copy );
end;

procedure   TPrimitiveNode.SetContentOutLinks(NewContentLinks      : PLinkList);
begin
  ContentOutLinks := NewContentLinks;
end;

function    TPrimitiveNode.GetContentOutLinks : PLinkList;
begin
  GetContentOutLinks := ContentOutLinks;
end;

procedure   TPrimitiveNode.AddContentOutLink(LinkName: LongInt);
begin
  ContentOutLinks^.Insert(New(PLinkId, Init(LinkName)));     {add link to list}
end;

procedure   TPrimitiveNode.AddContentNode(TheNode : PNodeObject; TheTransform : PTransform);
{If already a content link to this TPrimitiveNode, add sublink;
   else add new link.}
begin
  Abstract('TPrimitiveNode.AddContentNode');    {temporary}
end;

        {****************************************
        *
        *       methods for THermesNode
        *
        ****************************************}

constructor THermesNode.Init(NewName : PChar);
 begin
   inherited Init(NewName);
   SetType( ot_HermesNode );
   Kind := StrNew(NoName);
   ContentInLinks := New(PLinkList, Init);
   ContentInLinks^.SetName('hermes_content_in');
 end;

destructor THermesNode.Done;
begin
  StrDispose(Kind);
  if assigned( ContentInLinks )  then
    Dispose(ContentInLinks, Done);
  inherited Done;
end;

constructor THermesNode.Load(var S : TStream);
begin
        inherited Load(S);
  Kind := S.StrRead;
  ContentInLinks := PLinkList(S.Get);
end;

procedure THermesNode.Store(var S : TStream);
begin
  inherited Store(S);
  S.StrWrite(Kind);
  S.Put(ContentInLinks);
end;

function THermesNode.Eval(InputList : PHermesQueryList) : PHermesQueryList;
var
  TheList : PHermesQueryList;
  TheItem : PQueryListItem;
```

```
  TheCondition : PActiveObject;
begin
  TheCondition := GetCondition;
  if ((not assigned(TheCondition)) or TheCondition^.EvalBooleans)
    then
      begin
              TheItem := New(PQueryListItem, Init);
              TheItem^.SetDataId(GetId);       {store a copy of this THermesNode}
              TheList := New(PHermesQueryList, Init);
              TheList^.Insert(TheItem);
      end
    else     {The Condition not passed, return empty list}
      begin
        TheList := New(PHermesQueryList, Init);
      end;
  Eval := TheList;
end;

function  THermesNode.HermesDisplay : PChar;
{to HermesDisplay a THermesNode, display its definition and its Contents}
var
  DisplayArray : array[0..2000] of Char;
  TheCondition : PActiveObject;
   procedure DisplayIt(ANode : PPersistentObject); far;
    begin  {DisplayIt}
      StrCat(DisplayArray, ANode^.HermesDisplay);
      StrCat(DisplayArray, '   --   ');
    end;   {DisplayIt}
begin
  StrCopy(DisplayArray, 'Node of kind = ');
  StrCat(DisplayArray, Kind);
  StrCat(DisplayArray, '   contents = ');
  GetContentOutLinks^.ForEach(@DisplayIt);   {display each Content}
  TheCondition := GetCondition;
  if ((TheCondition = nil) or TheCondition^.EvalBooleans)
    then
      begin
        StrCat(DisplayArray, ', if ');
        StrCat(DisplayArray, TheCondition^.HermesDisplay);
      end;
  HermesDisplay := StrNew(DisplayArray);
end;


function  THermesNode.DisplayContent: PPicture;
{to Display a THermesNode, display a TPicture of the content of all its Contents}
 var
  ThePicture     : PPicture;
  NextLineOrigin : PGPoint;
      procedure AddOne(ALink: PLinkId); far;
      var
        TheContentId : LongInt;
        TheLink      : PLink;
        TheContext   : LongInt;
        TheContent   : PPersistentObject;
        TheDisplay   : PGraphic;
      begin   {AddOne}
        TheLink := PLink(DBGet(ALink^.GetId));
        TheContext := HermesApp.GetContext;
        if TraversesLink(TheContext, TheLink)
          then
            begin
              TheContentId := TheLink^.GetToPO;
              TheContent := DBGet(TheContentId);
              TheDisplay := PGraphic(TheContent^.DisplayContent);
              TheDisplay^.SetGraphicId(TheContentId);         {set the Display's GraphicId to the
Content's Id}
              TheDisplay^.PositionAt(NextLineOrigin);
              if (TypeOf(TheDisplay^) = TypeOf(TRichText))
                then
                  if PRichText(TheDisplay)^.Count > 0
                    then PRichText(TheDisplay)^.GetNextLinePos(NextLineOrigin^)
                else
                  if (TypeOf(TheDisplay^) = TypeOf(TLabel))
                    then
                      if (PLabel(TheDisplay)^.TextLength > 0)
```

```
                        then PLabel(TheDisplay)^.GetNextLinePos(NextLineOrigin^)
                else
                  if (TypeOf(TheDisplay^) = TypeOf(TPicture))   {if content is a Picture, then just
use THermesNode name}
                    then
                      begin
                        TheDisplay := New(PLabel, InitDefault);
                        if ((GetName <> nil) and (StrComp(GetName, NoName) <> 0))
                          then PLabel(TheDisplay)^.SetText(GetName);
                        PLabel(TheDisplay)^.GetNextLinePos(NextLineOrigin^)
                      end
                else NextLineOrigin^.Build(TheDisplay^.Left, TheDisplay^.Bottom + 15);
              ThePicture^.FastAdd(PGraphic(TheDisplay));
            end;
      end;    {AddOne}
begin  {DisplayContent}
  ThePicture := New(PPicture, InitDefault);
  NextLineOrigin := New(PGPoint, Init(0,0));
  GetContentOutLinks^.ForEach(@AddOne);
  ThePicture^.FindBounds;
  ThePicture^.SetGraphicId(GetId);           {set ThePicture's GraphicId to the THermesNode's Id}
  DisplayContent := ThePicture;
end;   {DisplayContent}


function THermesNode.Copy : Pointer;
var TheCopy : PHermesNode;
begin
  TheCopy := New(PHermesNode, Init(NoName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure THermesNode.CopyFrom(Source : PHermesNode);
begin
  inherited CopyFrom(Source);
  SetKind(Source^.GetKind);
  SetContentInLinks( Source^.GetContentInLinks^.Copy );
end;

procedure   THermesNode.SetKind(NewKind : PChar);
begin
  StrDispose(Kind);
  Kind := StrNew(NewKind);
end;

procedure   THermesNode.SetContentInLinks(NewContentLinks   : PLinkList);
begin
  ContentInLinks := NewContentLinks;
end;

function    THermesNode.GetKind : PChar;
begin
  GetKind := StrNew(Kind);
end;

function    THermesNode.GetContentInLinks : PLinkList;
begin
  GetContentInLinks := ContentInLinks;
end;

procedure   THermesNode.AddContentInLink(LinkName: LongInt);
var
 Dest : array [0..79] of Char;
 Strng : string[79];
begin
  if assigned(ContentInLinks)
    then ContentInLinks^.Insert(New(PLinkId, Init(LinkName)))
    else
      begin
        ErrorMsg(et_NOTICE, 'ContentInLinks not assigned');     {add link to list}
        Str(LinkName ,Strng);
        StrPCopy(Dest, Strng);
        ErrorMsg(et_NOTICE,Dest) ;
      end;
end;
```

```
        {***************************************
        *
        *       methods for THermesGraphic
        *
        ***************************************}

constructor THermesGraphic.Init(NewName : PChar);
 begin
   inherited Init(NewName);
   SetType( ot_HermesGraphic );
 end;

        {***************************************
        *
        *       methods for TPrivilegeObject
        *
        ***************************************}
 constructor TPrivilegeObject.Init(NewName : PChar);
 begin
   inherited Init(NewName);
   SetView(DefaultPrivilege);        {initialize all Logins and Contexts}
   SetAnnotate(DefaultPrivilege);    {to their default (public) values}
   SetEdit(DefaultPrivilege);
   SetPassword(NoName);
 end;

 destructor  TPrivilegeObject.Done;
 begin
   StrDispose(Password);
   inherited Done;
 end;

 constructor TPrivilegeObject.Load(var S : TStream);
 begin
   inherited Load(S);
   Password := S.StrRead;
   S.Read(View, SizeOf(Privilege));
   S.Read(Annotate, SizeOf(Privilege));
   S.Read(Edit, SizeOf(Privilege));
 end;

 procedure   TPrivilegeObject.Store(var S : TStream);
 begin
   inherited Store(S);
   S.StrWrite(Password);
   S.Write(View, SizeOf(Privilege));
   S.Write(Annotate, SizeOf(Privilege));
   S.Write(Edit, SizeOf(Privilege));
 end;

 function    TPrivilegeObject.HermesDisplay : PChar;
 begin
   HermesDisplay := 'TPrivilegeObject';
 end;

 function    TPrivilegeObject.Copy : Pointer;
var
  TheCopy : PPrivilegeObject;
begin
  New(TheCopy, Init(NoName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
 end;

 procedure   TPrivilegeObject.CopyFrom(Source : PPrivilegeObject);
 begin
  inherited CopyFrom(Source);
  SetPassword(Source^.GetPassword);
  SetView(Source^.GetView);
  SetAnnotate(Source^.GetAnnotate);
  SetEdit(Source^.GetEdit);
 end;

 procedure   TPrivilegeObject.SetView(NewView : Privilege);
 begin
```

```
   View := NewView;
 end;

 procedure   TPrivilegeObject.SetAnnotate(NewAnnotate       : Privilege);
 begin
   Annotate := NewAnnotate;
 end;

 procedure   TPrivilegeObject.SetEdit(NewEdit       : Privilege);
 begin
   Edit := NewEdit;
 end;

 procedure   TPrivilegeObject.SetPassword(NewPassword       : PChar);
 begin
  Password := StrNew(NewPassword);
 end;

 function    TPrivilegeObject.GetView     : Privilege;
 begin
   GetView := View;
 end;

 function    TPrivilegeObject.GetAnnotate : Privilege;
 begin
   GetAnnotate := Annotate;
 end;

 function    TPrivilegeObject.GetEdit     : Privilege;
 begin
   GetEdit := Edit;
 end;

 function    TPrivilegeObject.GetPassword     : PChar;
 begin
   GetPassword := StrNew(Password);
end;

       {****************************************
       *
       *       methods for TLogin
       *
       ****************************************}

constructor TLogin. Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType( ot_Login );
end;

       {****************************************
       *
       *       methods for TContext
       *
       ****************************************}

constructor TContext.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType( ot_Context );
  VEdit := True;
  Color := StrNew('black');
end;

destructor  TContext.Done;
begin
  if assigned(Color) then StrDispose(Color);
  inherited Done;
end;

constructor TContext.Load(var S : TStream);
begin
  inherited Load(S);
  Color := S.StrRead;
  S.Read(VEdit, SizeOf(Boolean));
end;
```

```
procedure   TContext.Store(var S : TStream);
begin
  inherited Store(S);
  S.StrWrite(Color);
  S.Write(VEdit, SizeOf(Boolean));
end;

function    TContext.HermesDisplay : PChar;
begin
  HermesDisplay := 'TContext';
end;

function    TContext.Copy : Pointer;
var
  TheCopy : PContext;
begin
  New(TheCopy, Init(NoName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure   TContext.CopyFrom(Source : PContext);
begin
  inherited CopyFrom(Source);
  SetColor(Source^.GetColor);
  SetVEdit(Source^.GetVEdit);
end;

procedure   TContext.SetColor(NewColor       : PChar);
begin
  Color := StrNew(NewColor);
end;

function    TContext.GetColor : PChar;
begin
  GetColor := StrNew(Color);
end;

procedure   TContext.SetVEdit(NewVEdit       : Boolean);
begin
  VEdit := NewVEdit;
end;

function    TContext.GetVEdit : Boolean;
begin
  GetVEdit := VEdit;
end;

        {****************************************
        *
        *       methods for TNodeKind
        *
        ****************************************}

constructor TNodeKind.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType(ot_NodeKind);
  InstanceCount := 0;
end;

destructor  TNodeKind.Done;
begin
  InstanceCount := 0;
  inherited Done;
end;

constructor TNodeKind.Load(var S : TStream);
begin
  inherited Load(S);
  S.Read(InstanceCount, SizeOf(Integer));
end;

procedure   TNodeKind.Store(var S : TStream);
begin
```

```
  inherited Store(S);
  S.Write(InstanceCount, SizeOf(Integer));
end;

function TNodeKind.Copy : Pointer;
var TheCopy : PNodeKind;
begin
  New(TheCopy, Init(GetName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure TNodeKind.CopyFrom(Source : PNodeKind);
begin
  inherited CopyFrom(Source);
  SetInstanceCount(Source^.GetInstanceCount);
end;

procedure   TNodeKind.SetInstanceCount(NewInstanceCount : Integer);
begin
  InstanceCount := NewInstanceCount;
end;

function    TNodeKind.GetInstanceCount : Integer;
begin
  GetInstanceCount := InstanceCount;
end;

procedure   TNodeKind.DecInstanceCount;
begin
  Dec(InstanceCount);
end;

procedure   TNodeKind.IncInstanceCount;
begin
  Inc(InstanceCount);
end;

        {****************************************
        *
        *       methods for TLinkType
        *
        ****************************************}

constructor TLinkType.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType( ot_LinkType );
  InstanceCount := 0;
end;

destructor  TLinkType.Done;
begin
  InstanceCount := 0;
  inherited Done;
end;

constructor TLinkType.Load(var S : TStream);
begin
  inherited Load(S);
  S.Read(InstanceCount, SizeOf(Integer));
end;

procedure   TLinkType.Store(var S : TStream);
begin
  inherited Store(S);
  S.Write(InstanceCount, SizeOf(Integer));
end;

function TLinkType.Copy : Pointer;
var TheCopy : PLinkType;
begin
  New(TheCopy, Init(GetName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;
```

```
procedure TLinkType.CopyFrom(Source : PLinkType);
begin
  inherited CopyFrom(Source);
  SetInstanceCount(Source^.GetInstanceCount);
end;

procedure   TLinkType.SetInstanceCount(NewInstanceCount : Integer);
begin
  InstanceCount := NewInstanceCount;
end;

function    TLinkType.GetInstanceCount : Integer;
begin
  GetInstanceCount := InstanceCount;
end;

procedure   TLinkType.DecInstanceCount;
begin
  Dec(InstanceCount);
end;

procedure   TLinkType.IncInstanceCount;
begin
  Inc(InstanceCount);
end;

        {***************************************
        *
        *       methods for TResultList
        *
        ***************************************}

constructor TResultList.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType(ot_ResultList);
  ListOfData := nil;
end;

destructor  TResultList.Done;
begin
  if assigned( ListOfData ) then
     Dispose(ListOfData, Done);
  inherited Done;
end;

constructor TResultList.Load(var S : TStream);
begin
  inherited Load(S);
  ListOfData := PHermesItemsList(S.Get);
end;

procedure   TResultList.Store(var S : TStream);
begin
  inherited Store(S);
  S.Put(ListOfData);
end;

function TResultList.Copy : Pointer;
var TheObject : PResultList;
begin
  New(TheObject, Init(NoName));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

procedure TResultList.CopyFrom(Source : PResultList);
begin
   inherited CopyFrom(Source);
  SetListOfData(Source^.GetListOfData^.Copy);
end;

procedure   TResultList.SetListOfData(NewListOfData : PHermesItemsList);
begin
  ListOfData := NewListOfData;
```

```
end;

function    TResultList.GetListOfData : PHermesItemsList;
begin
  GetListOfData := ListOfData;
end;

       {**********************************
  *
       *    methods for TGraphicContent
       *
       **********************************}

       {**********************************
  *
       *    methods for TUnitCube
       *
       **********************************}

constructor TUnitCube.Init(NewName : PChar);
 begin
   inherited Init(NewName);
   SetType( ot_UnitCube );
 end;

       {**********************************
  *
       *    methods for THermesPolyline
       *
       **********************************}

constructor THermesPolyline.Init(NewName : PChar);
 begin
   inherited Init(NewName);
   ListOfPoints := New(PHermesPointList, Init);
   SetType( ot_HermesPolyline );
 end;

destructor THermesPolyline.Done;
 begin
   if assigned(ListOfPoints) then
      dispose(ListOfPoints, Done);
   inherited Done;
 end;

constructor THermesPolyline.Load(var S : TStream);
 begin
   inherited Load(S);
   ListOfPoints := PHermesPointList(S.Get);
 end;

procedure THermesPolyline.Store(var S : TStream);
 begin
   inherited Store(S);
   S.Put(ListOfPoints);
 end;

function THermesPolyline.Copy : Pointer;
 var TheObject : PHermesPolyline;
 begin
   New(TheObject, Init(NoName));
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
 end;

procedure THermesPolyline.CopyFrom(Source : PHermesPolyline );
 begin
   inherited CopyFrom(Source);
   SetListofPoints(Source^.GetListofPoints);
 end;

procedure THermesPolyline.SetListofPoints( NewListofPoints : PHermesPointList);
 begin
 ListOfPoints := NewListofPoints;
 end;
```

```
function THermesPolyline.GetListofPoints : PHermesPointList;
 begin
   GetListofPoints := ListofPoints;
 end;

        {*********************************
  *
        *    methods for TSweep
        *
        *********************************}

constructor TSweep.Init(NewName : PChar);
 begin
   inherited Init(NewName);
   SetType( ot_Sweep );
 end;

end.      {**** unit HerNodes  ****}
```

6. HerHyper.pas define hypermedia objects

```
 {*****************************************
 *
 *      HerHyper.pas
 *
 *      version 2.0 -- Spring 1994
 *      (c) copyright 1994 by Gerry Stahl
 *      all rights reserved
 *****************************************}

unit HerHyper;

interface

uses Objects,
     HerLists, HerNodes, HerOGL2;

{ This unit defines the media and language components of the Hermes system
  as proposed in (Stahl, Ph.D. dissertation, 1993).

       DESIGN:
    TNodeContent is the ancestor for
    TMediaElement, TLanguageElement, TTerminologyElement,
    from which the media and language options derive.
}

type

        {***************************************
        *
        *      simple terminology elements for the Hermes language
        *
        ****************************************}

  Terminology   = (AND_LOGICAL, OR_LOGICAL, {add Connectives here}

                   AND_UNION, AND_ALSO, BUT_NOT, OR_APPEND, {add Combinations here}

                   CENTRAL_DISTANCE, CLOSEST_DISTANCE, X_DISTANCE, Y_DISTANCE, Z_DISTANCE, {add
Distances here}

                   FEET, INCHES, METERS, CENTIMETERS, SQ_FEET, SQ_METERS, CU_FEET, CU_METERS, {add
Units here}

                   LENGTH, AREA, VOLUME, X_WIDTH, Y_HEIGHT, Z_DEPTH, {add Dimensions here}

                   FONT, COLOR, PEN_WIDTH, BRUSH_STYLE, BRUSH_WIDTH,
                   LINE_COLOR, LINE_STYLE, LINE_WIDTH, LINE_MIXING,
                   FILL_COLOR, FILL_STYLE, FILL_MIXING, {add Attributes here}

                   TIMES_ROMAN, HELETICA, RED, BLUE, STRIPED, PLAID, {add Values here}

                   DATALISTS, ASSOCIATIONS, FILTERS, CHARACTERS, NUMBERS, BOOLS,
                   GRAPHICS, IMAGES, PENS, SOUNDS, VIDEOS, ANIMATIONS, COMPUTEDVIEWS,
                   COUNTS, QUANTIFIERS, MEASURES, NODEKINDS, LINKTYPES,    {add LanguageTypes here}

                   unassigned);

  Connective   = AND_lOGICAL .. OR_LOGICAL;

  Combination  = AND_UNION .. OR_APPEND;

  Distance     = CENTRAL_DISTANCE .. Z_DISTANCE;

  Units        = FEET .. CU_METERS;

  Dimension    = LENGTH .. Z_DEPTH;

  Attribute    = FONT .. FILL_MIXING;

  Value        = TIMES_ROMAN .. PLAID;

  LanguageType = DATALISTS .. LINKTYPES;
```

```
      {****************************************
      *
      *    data structure for TNodeContent
      *    all media and language expressions derive from here
      ****************************************}

      PNodeContent = ^TNodeContent;
TNodeContent = object (TNodeObject)
             constructor Init(NewName : PChar);
             destructor  Done; virtual;
             constructor Load(var S : TStream);
             procedure   Store(var S : TStream);         virtual StoreDMT;
             function    Copy : Pointer;                 virtual CopyDMT;
  procedure   CopyFrom(Source : PNodeContent);
  procedure   SetParam1(NewParam : PNodeContent);
  function    GetParam1 : PNodeContent;
  procedure   SetParam2(NewParam : PNodeContent);
  function    GetParam2 : PNodeContent;
  procedure   SetParam3(NewParam : PNodeContent);
  function    GetParam3 : PNodeContent;
  procedure   SetParam4(NewParam : PNodeContent);
  function    GetParam4 : PNodeContent;
  procedure   SetParam5(NewParam : PNodeContent);
  function    GetParam5 : PNodeContent;
  procedure   SetTerm1(NewTerm : Terminology);
  function    GetTerm1 : Terminology;
  procedure   SetTerm2(NewTerm : Terminology);
  function    GetTerm2 : Terminology;
  procedure   SetText(NewText : PChar);
  function    GetText : PChar;
  procedure   SetReal(NewReal : Real);
  function    GetReal : Real;
  procedure   SetContentId(NewId : LongInt);
  function    GetContentId : LongInt;
private
  Param1  : PNodeContent;      {the syntax options for ContentNodes}
  Param2  : PNodeContent;      {can have several parameters}
  Param3  : PNodeContent;      {that have to be saved}
  Param4  : PNodeContent;
  Param5  : PNodeContent;
  Term1   : Terminology;
  Term2   : Terminology;
  TheText : PChar;
  TheReal : Real;
  TheId   : LongInt;
      end;

      {****************************************
      *
      *    data structure for TMediaElements
*
      ****************************************}

PMediaElement = ^TMediaElement;
      TMediaElement = object (TNodeContent)
  {function    DisplaySelf( Display_Mode : TDisplayModes;
    Destination_Space : POINTER ) : PPersistentObject; virtual DisplayContentDMT;}
      end;


      PCharacter = ^TCharacter;
      TCharacter = object( TMediaElement )
   constructor Init(NewName : PChar);
        function    EvalCharacter : PChar; virtual;
   function    DisplayContent : PPicture; virtual DisplayContentDMT;
end;


      PNumber = ^TNumber;
      TNumber = object (TMediaElement)
   constructor Init(NewName : PChar);
        function    EvalNumber : Real; virtual;
   function    DisplayContent : PPicture; virtual DisplayContentDMT;
      end;
```

```
      PBooleans = ^TBooleans;
      TBooleans = object (TMediaElement)
    constructor Init(NewName : PChar);
         function    EvalBooleans : Boolean; virtual;
    function    DisplayContent : PPicture; virtual DisplayContentDMT;
      end;

      PImage = ^TImage;
      TImage = object (TMediaElement)
    constructor Init(NewName : PChar);
         function    EvalImage : Pointer; virtual;
    function    Copy: Pointer; virtual CopyDMT;
      end;

      PPen = ^TPen;
      TPen = object (TMediaElement)
    constructor Init(NewName : PChar);
         function    EvalPen : Pointer; virtual;
    function    Copy: Pointer; virtual CopyDMT;
      end;

      PSound = ^TSound;
      TSound = object (TMediaElement)
    constructor Init(NewName : PChar);
         function    EvalSound : Pointer; virtual;
    function    Copy: Pointer; virtual CopyDMT;
      end;

      PVideo = ^TVideo;
      TVideo = object (TMediaElement)
    constructor Init(NewName : PChar);
         function    EvalVideo : Pointer; virtual;
    function    Copy: Pointer; virtual CopyDMT;
      end;

      PAnimation = ^TAnimation;
      TAnimation = object (TMediaElement)
    constructor Init(NewName : PChar);
         function    EvalAnimation : Pointer; virtual;
    function    Copy: Pointer; virtual CopyDMT;
      end;

(*
      PComputedView = ^TComputedView;
      TComputedView = object (TMediaElement)
        function    EvalView : Pointer; virtual;
             constructor Init(NewName : PChar);
             destructor  Done; virtual;
             constructor Load(var S : TStream);
             procedure   Store(var S : TStream); virtual StoreDMT;
    function    HermesDisplay : PChar; virtual HermesDisplayDMT;
    function    DisplayContent : PObject; virtual DisplayContentDMT;
             function    Copy : Pointer; virtual CopyDMT;
    procedure   CopyFrom(Source : PComputedView);
    procedure   SetPicture(NewPicture: PPicture);
    function    GetPicture: PPicture;
    procedure   SetSpace(NewSpace: PGraphSpace);
    function    GetSpace: PGraphSpace;
    procedure   SetIcon(NewIcon: PIcon);
    function    GetIcon: PIcon;
      private
             ThePicture     : PPicture;
             TheSpace       : PGraphSpace;
             TheIcon        : PIcon;
      end;
*)

      {****************************************
      *
      *    data structure for TLanguageElements
      *
      ****************************************}

      PLanguageElement = ^TLanguageElement;
      TLanguageElement = object (TNodeContent)
    function Eval(InputList : PHermesQueryList): PHermesQueryList; virtual EvalDMT;
```

```
        end;

        PDataList = ^TDataList;
        TDataList = object (TLanguageElement)
    constructor Init(NewName : PChar);
        end;

        PAssociation = ^TAssociation;
        TAssociation = object (TLanguageElement)
    constructor Init(NewName : PChar);
        end;

        PPredicate = ^TPredicate;
        TPredicate = object (TLanguageElement)
    constructor Init(NewName : PChar);
        end;

        PFilter = ^TFilter;
        TFilter = object (TLanguageElement)
    constructor Init(NewName : PChar);
        end;

        {****************************************
        *
        *    data structure for TTerminologyElements
        *
        ****************************************}

        PTerminologyElement = ^TTerminologyElement;
        TTerminologyElement = object (TNodeContent)
        end;

        PCounter = ^TCounter;
        TCounter = object (TTerminologyElement)
    constructor Init(NewName : PChar);
    function    EvalCounter(TheCounter : Real) : Boolean; virtual;
        end;

        PQuantifier = ^TQuantifier;
        TQuantifier = object (TTerminologyElement)
    constructor Init(NewName : PChar);
    function    EvalQuantifier(TheCounter, TheTotal : Real) : Boolean; virtual;
        end;

        PMeasure = ^TMeasure;
        TMeasure = object (TTerminologyElement)
    constructor Init(NewName : PChar);
    function    EvalMeasure(FromGraphic, ToGraphic, InGraphic : PHermesGraphic) : Boolean; virtual;
        end;

{=====================================================================}

implementation

uses Strings,
     HerBasic;

        {****************************************
        *
        *       methods for TNodeContent
        *
        ****************************************}

constructor TNodeContent.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType( ot_undefined );
  Param1 := nil;   {initialize to nil}
  Param2 := nil;   {then users check if assigned}
  Param3 := nil;
  Param4 := nil;
  Param5 := nil;
  Term1 := unassigned;
  Term2 := unassigned;
  TheText := StrNew(NoName);
  TheReal := 0;
```

```
  TheId := 0;
end;

destructor  TNodeContent.Done;
begin
  if assigned(Param1) then
    Dispose(Param1, Done);
  if assigned(Param2 ) then
    Dispose(Param2, Done);
  if assigned(Param3 ) then
    Dispose(Param3, Done);
  if assigned(Param4 ) then
    Dispose(Param4, Done);
  if assigned(Param5 ) then
    Dispose(Param5, Done);
  if assigned(TheText) then
    StrDispose(TheText);
  inherited Done;
end;

constructor TNodeContent.Load(var S : TStream);
begin
  inherited Load(S);
  Param1 := PNodeContent(S.Get);
  Param2 := PNodeContent(S.Get);
  Param3 := PNodeContent(S.Get);
  Param4 := PNodeContent(S.Get);
  Param5 := PNodeContent(S.Get);
  S.Read(Term1, SizeOf(Terminology));
  S.Read(Term2, SizeOf(Terminology));
  TheText := S.StrRead;
  S.Read(TheReal, SizeOf(Real));
  S.Read(TheId, SizeOf(LongInt));
end;

procedure   TNodeContent.Store(var S : TStream);
begin
  inherited Store(S);
  S.Put(Param1);
  S.Put(Param2);
  S.Put(Param3);
  S.Put(Param4);
  S.Put(Param5);
  S.Write(Term1, SizeOf(Terminology));
  S.Write(Term2, SizeOf(Terminology));
  S.StrWrite(TheText);
  S.Write(TheReal, SizeOf(Real));
  S.Write(TheId, SizeOf(LongInt));
end;

function TNodeContent.Copy : Pointer;
var
  TheObject : PNodeContent;
begin
       New(TheObject, Init(NoName));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

procedure TNodeContent.CopyFrom(Source : PNodeContent);
begin
  inherited CopyFrom(Source);
  if (Source^.Param1 = nil)
    then Param1 := nil
    else Param1 := Source^.Param1^.Copy;
  if (Source^.Param2 = nil)
    then Param2 := nil
    else Param2 := Source^.Param2^.Copy;
  if (Source^.Param3 = nil)
    then Param3 := nil
    else Param3 := Source^.Param3^.Copy;
  if (Source^.Param4 = nil)
    then Param4 := nil
    else Param4 := Source^.Param4^.Copy;
  if (Source^.Param5 = nil)
    then Param5 := nil
```

```
    else Param5 := Source^.Param4^.Copy;
  SetTerm1(Source^.GetTerm1);
  SetTerm2(Source^.GetTerm2);
  SetText(Source^.GetText);
  SetReal(Source^.GetReal);
  SetContentId(Source^.GetContentId);
end;

procedure   TNodeContent.SetParam1(NewParam : PNodeContent);
begin
  Param1 := NewParam;
end;

function    TNodeContent.GetParam1 : PNodeContent;
begin
  GetParam1 := Param1;
end;

procedure   TNodeContent.SetParam2(NewParam : PNodeContent);
begin
  Param2 := NewParam;
end;

function    TNodeContent.GetParam2 : PNodeContent;
begin
  GetParam2 := Param2;
end;

procedure   TNodeContent.SetParam3(NewParam : PNodeContent);
begin
  Param3 := NewParam;
end;

function    TNodeContent.GetParam3 : PNodeContent;
begin
  GetParam3 := Param3;
end;

procedure   TNodeContent.SetParam4(NewParam : PNodeContent);
begin
  Param4 := NewParam;
end;

function    TNodeContent.GetParam4 : PNodeContent;
begin
  GetParam4 := Param4;
end;

procedure   TNodeContent.SetParam5(NewParam : PNodeContent);
begin
  Param5 := NewParam;
end;

function    TNodeContent.GetParam5 : PNodeContent;
begin
  GetParam5 := Param5;
end;

procedure   TNodeContent.SetTerm1(NewTerm : Terminology);
begin
  Term1 := NewTerm;
end;

function    TNodeContent.GetTerm1 : Terminology;
begin
  GetTerm1 := Term1;
end;

procedure   TNodeContent.SetTerm2(NewTerm : Terminology);
begin
  Term2 := NewTerm;
end;

function    TNodeContent.GetTerm2 : Terminology;
begin
  GetTerm2 := Term2;
```

```
end;

procedure   TNodeContent.SetText(NewText : PChar);
begin
  TheText := StrNew(NewText);
end;

function    TNodeContent.GetText : PChar;
begin
  GetText := StrNew(TheText);
end;

procedure   TNodeContent.SetReal(NewReal : Real);
begin
  TheReal := NewReal;
end;

function    TNodeContent.GetReal : Real;
begin
  GetReal := TheReal;
end;

procedure   TNodeContent.SetContentId(NewId : LongInt);
begin
  TheId := NewId;
end;

function    TNodeContent.GetContentId : LongInt;
begin
  GetContentId := TheId;
end;

        {****************************************
        *
        *       methods for TMediaElement
        *
        ****************************************}

        {****************************************
        *
        *       methods for TCharacter
        *
        ****************************************}

constructor TCharacter.Init(NewName : PChar);
begin
   inherited Init(NewName);
   SetType( ot_Character );
end;

function TCharacter.EvalCharacter: PChar;
begin
  Abstract('TCharacter.EvalCharacter');
end;

function TCharacter.DisplayContent : PPicture;
var
  TheLabel : PLabel;
  TheRichText : PRichText;
begin
  TheLabel := New(PLabel, InitDefault);
  TheLabel^.SetText(EvalCharacter);
  TheRichText := New(PRichText, InitDefault);
  TheRichText^.Add(TheLabel);
  DisplayContent := TheRichText;
end;

(*
function              TCharacter.DisplaySelf
         (     Display_Mode            : TDisplayModes;
               Destination_Space       : POINTER                ) : PPersistentObject;
var TheRichText              : PRichText;
    TextColor                : PColor;
    BackColor                : PColor;
    TheTextPen               : PTextPen;
begin
```

```
  CASE Display_Mode OF
    _Content, _Content_Representation :
      begin
        TheRichText := NEW ( PRichText, Init ( tc_Tools ) );
        TheRichText^.SetType ( '^' );
        TheRichText^.SetID ( GetID );
        AsRichText ( TheRichText^ );

        TextColor   := NEW ( PColor, Init ( ps_Black ) );
        BackColor   := NEW ( PColor, Init ( ps_White ) );
        TheTextPen  := NEW ( PTextPen, Init ( TextColor, BackColor ) );
        TheTextPen^.Height := 15;                 {set font attributes}
        TheTextPen^.Style := [ ts_Bold ];
        TheTextPen^.Font := fs_PropSerif;
        TheRichText^.SetTextPen ( TheTextPen );

        DISPOSE ( TextColor, Done );
        DISPOSE ( BackColor, Done );
        DISPOSE ( TheTextPen, Done );

        DisplaySelf := TheRichText;
      end;
    _Representation : DisplaySelf := MakeIconRep ( @Self, 'B_TEXT', Destination_Space );
  end;
end;
*)

       {****************************************
       *
       *       methods for TNumber
       *
       ****************************************}

constructor TNumber.Init(NewName : PChar);
begin
   inherited Init(NewName);
   SetType( ot_Number );
end;

function TNumber.EvalNumber: Real;
begin
  Abstract('TNumber.EvalNumber');
end;

function TNumber.DisplayContent : PPicture;
var
  TheLabel    : PLabel;
  TheRichText : PRichText;
  TheNumber   : real;
  TheString   : string;
  TheTChar    : array[0..79] of Char;
  ThePChar    : array[0..79] of Char;
begin
  TheLabel := New(PLabel, InitDefault);
  TheNumber := EvalNumber;
  if Frac(TheNumber) = 0
    then Str(TheNumber:8:0, TheString)
    else Str(EvalNumber:12:4, TheString);
  StrPCopy(TheTChar, TheString);
  StrCopy(ThePChar, ' n =');
  StrCat(ThePChar, TheTChar);
  TheLabel^.SetText(StrNew(ThePChar));
  TheRichText := New(PRichText, InitDefault);
  TheRichText^.Add(TheLabel);
  DisplayContent := TheRichText;
end;

       {****************************************
       *
       *       methods for TBooleans
       *
       ****************************************}

constructor TBooleans.Init(NewName : PChar);
begin
   inherited Init(NewName);
```

```
   SetType( ot_Booleans );
end;

function TBooleans.EvalBooleans: Boolean;
begin
  Abstract('TBooleans.EvalBooleans');
end;

function TBooleans.DisplayContent : PPicture;
var
  TheLabel : PLabel;
  TheRichText : PRichText;
begin
  TheLabel := New(PLabel, InitDefault);
  if EvalBooleans
    then  TheLabel^.SetText(StrNew('True'))
    else  TheLabel^.SetText(StrNew('False'));
  TheRichText := New(PRichText, InitDefault);
  TheRichText^.Add(TheLabel);
  DisplayContent := TheRichText;
end;

        {****************************************
        *
        *       methods for TImage
        *
        ****************************************}

constructor TImage.Init(NewName : PChar);
begin
   inherited Init(NewName);
   SetType( ot_Image );
end;

function TImage.EvalImage: Pointer;
begin
  Abstract('TImage.EvalImage');
end;

function TImage.Copy: Pointer;
 var TheObject : PImage;
 begin
   New(TheObject, Init(NoName));
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
end;

        {****************************************
        *
        *       methods for TPen
        *
        ****************************************}

constructor TPen.Init(NewName : PChar);
begin
   inherited Init(NewName);
   SetType( ot_Pen );
end;

function TPen.EvalPen: Pointer;
begin
  Abstract('TPen.EvalPen');
end;

function TPen.Copy: Pointer;
 var TheObject : PPen;
 begin
   New(TheObject, Init(NoName));
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
end;

        {****************************************
        *
        *       methods for TSound
        *
```

```
        ****************************************}

constructor TSound.Init(NewName : PChar);
begin
   inherited Init(NewName);
   SetType( ot_Sound );
end;

function TSound.EvalSound: Pointer;
begin
  Abstract('TSound.EvalSound');
end;

function TSound.Copy: Pointer;
 var TheObject : PSound;
 begin
   New(TheObject, Init(NoName));
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
end;

        {****************************************
        *
        *       methods for TVideo
        *
        ****************************************}

constructor TVideo.Init(NewName : PChar);
begin
   inherited Init(NewName);
   SetType( ot_Video );
end;

function TVideo.EvalVideo: Pointer;
begin
  Abstract('TVideo.EvalVideo');
end;

function TVideo.Copy: Pointer;
 var TheObject : PVideo;
 begin
   New(TheObject, Init(NoName));
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
end;

        {****************************************
        *
        *       methods for TAnimation
        *
        ****************************************}

constructor TAnimation.Init(NewName : PChar);
begin
   inherited Init(NewName);
   SetType( ot_Animation );
end;

function TAnimation.EvalAnimation: Pointer;
begin
  Abstract('TAnimation.EvalAnimation');
end;

function TAnimation.Copy: Pointer;
 var TheObject : PAnimation;
 begin
   New(TheObject, Init(NoName));
   TheObject^.CopyFrom(@Self);
   Copy := TheObject;
end;

(*
  {****************************************
  *
  *       methods for ComputedView
  *
```

```
  ****************************************}

function    TComputedView.EvalView : Pointer;
begin

end;

constructor TComputedView.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType( ot_ComputedView );
  SetName(NewName);
  ThePicture := New(PPicture, InitDefault);
  TheSpace := New(PGraphSpace, InitDefault);
  TheIcon := nil;
end;

destructor  TComputedView.Done;
begin
{  if (ThePicture <> nil)
    then Dispose(ThePicture, Done);   }{don't allow this to be disposed; it might be saved to disk}{
  if (TheSpace <> nil)
    then Dispose(TheSpace, Done);
  if (TheIcon <> nil)
    then Dispose(TheIcon, Done);   }
end;

constructor TComputedView.Load(var S : TStream);
begin
  Inherited Load(S);
  ThePicture := PPicture(S.Get);
  TheSpace   := PGraphSpace(S.Get);
  TheIcon    := PIcon(S.Get);
end;

procedure   TComputedView.Store(var S : TStream);
begin
  Inherited Store(S);
  S.Put(ThePicture);
  S.Put(TheSpace);
  S.Put(TheIcon);
end;

function    TComputedView.HermesDisplay : PChar;
begin
  HermesDisplay := nil;
  if ((GetName <> nil) and (StrComp(GetName, NoName) <> 0))
    then HermesDisplay := GetName;
end;

function    TComputedView.DisplayContent : PObject;
begin
  DisplayContent := ThePicture;
end;

function TComputedView.Copy : Pointer;
var
  TheCopy : PComputedView;
begin
        New(TheCopy, Init(GetName));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;

procedure TComputedView.CopyFrom(Source : PComputedView);
begin
  SetPicture(Source^.GetPicture);
  SetSpace(Source^.GetSpace);
  SetIcon(Source^.GetIcon);
end;

procedure   TComputedView.SetPicture(NewPicture: PPicture);
begin
  ThePicture := NewPicture;
end;
```

```
function    TComputedView.GetPicture: PPicture;
begin
  GetPicture := ThePicture;
end;

procedure   TComputedView.SetSpace(NewSpace: PGraphSpace);
begin
  TheSpace := NewSpace;
end;

function    TComputedView.GetSpace: PGraphSpace;
begin
  GetSpace := TheSpace;
end;

procedure   TComputedView.SetIcon(NewIcon: PIcon);
begin
  TheIcon := NewIcon;
end;

function    TComputedView.GetIcon: PIcon;
begin
  GetIcon := TheIcon;
end;

*)

        {****************************************
        *
        *       methods for LanguageElement
        *
        ****************************************}

function TLanguageElement.Eval(InputList : PHermesQueryList): PHermesQueryList;
begin
  if assigned( InputList ) then
     Eval := InputList
  else
    begin
      Eval := New(PHermesQueryList, Init);
      Abstract('LanguageElement.Eval');
    end; {else}
end;

  {****************************************
  *
  *       methods for TDataList
  *
  ****************************************}

constructor TDataList.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType( ot_DataList );
end;

  {****************************************
  *
  *       methods for TAssociation
  *
  ****************************************}

constructor TAssociation.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType( ot_Association );
end;

  {****************************************
  *
  *       methods for TFilter
  *
  ****************************************}

constructor TFilter.Init(NewName : PChar);
begin
```

```
  inherited Init(NewName);
  SetType( ot_Filter);
end;

  {****************************************
  *
  *       methods for TPredicate
  *
  ****************************************}

constructor TPredicate.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType( ot_Predicate);
end;

        {****************************************
        *
        *       methods for TerminologyElement
        *
        ****************************************}

  {****************************************
  *
  *       methods for Counter
  *
  ****************************************}

constructor TCounter.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType( ot_Counter );
end;

function    TCounter.EvalCounter(TheCounter : Real) : Boolean;
begin
  Abstract('TCounter.EvalCounter');
end;

  {****************************************
  *
  *       methods for Quantifier
  *
  ****************************************}

constructor TQuantifier.Init(NewName : PChar);
begin
  inherited Init(NewName);
  SetType( ot_Quantifier );
end;

function    TQuantifier.EvalQuantifier(TheCounter, TheTotal : Real) : Boolean;
begin
  Abstract('TQuantifier.EvalQuantifier');
end;

  {****************************************
  *
  *       methods for Measure
  *
  ****************************************}

constructor TMeasure.Init(NewName : PChar);
begin
   inherited Init(NewName);
   SetType( ot_Measure );
end;

function    TMeasure.EvalMeasure(FromGraphic, ToGraphic, InGraphic : PHermesGraphic) : Boolean;
begin
   Abstract('TMeasure.EvalMeasure');
end;


end.      {**** unit HerHyper  ****}
```

7. HerMedia.pas define media objects

```
 {*****************************************
 *
 *      HerMedia.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 ******************************************}

unit HerMedia;

 { This unit defines the options for Characters, Numbers, and Booleans in the Hermes language.
   These options were proposed in (Stahl, Ph.D. dissertation, 1993).  }

interface

uses Objects,
     HerLists, HerHyper;

type

  {***************************************
        *
        *    data structure for TCharacters
        *
        ****************************************}

  { SimpleCharacter = character string }
        PCharacterSimple = ^TCharacterSimple;
        TCharacterSimple = object( TCharacter )
           constructor Init(NewName : PChar; NewText : PChar);
           function    EvalCharacter : PChar; virtual;
     function    HermesDisplay : PChar; virtual HermesDisplayDMT;
     function    Copy : Pointer;        virtual CopyDMT;
  end;

  { ComputedCharacter = substring of Character from Number for Number }
        PCharacterSubstring = ^TCharacterSubstring;
        TCharacterSubstring = object( TCharacter )
           constructor Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent;
NewParam3 : PNodeContent);
           function    EvalCharacter : PChar; virtual;
     function    HermesDisplay : PChar; virtual HermesDisplayDMT;
     function    Copy : Pointer;        virtual CopyDMT;
  end;

  { ComputedCharacter = Character append Character }
        PCharacterAppend = ^TCharacterAppend;
        TCharacterAppend = object( TCharacter )
           constructor Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent);
           function    EvalCharacter : PChar; virtual;
     function    HermesDisplay : PChar; virtual HermesDisplayDMT;
     function    Copy : Pointer;        virtual CopyDMT;
  end;

        {***************************************
        *
        *    data structure for TNumbers
  *
        *    These options operate on TNumbers only;
  *    if none are found, they return 0
        ****************************************}

  { SimpleNumber = real }
        PNumberSimple = ^TNumberSimple;
        TNumberSimple = object( TNumber )
           constructor Init(NewName : PChar; NewReal : Real );
           function    EvalNumber : Real; virtual;
     function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
     function    Copy : Pointer;         virtual CopyDMT;
  end;
```

```
{ ComputedNumber = count of DataList }
      PNumberCount = ^TNumberCount;
      TNumberCount = object( TNumber )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent );
         function    EvalNumber : Real; virtual;
   function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
   function    Copy : Pointer;         virtual CopyDMT;
end;

{ ComputedNumber = minimum of DataList }
      PNumberMinimum = ^TNumberMinimum;
      TNumberMinimum = object( TNumber )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent );
         function    EvalNumber : Real; virtual;
   function    HermesDisplay : PChar;   virtual HermesDisplayDMT;
   function    Copy : Pointer;          virtual CopyDMT;
end;

{ ComputedNumber = maximum of DataList }
      PNumberMaximum = ^TNumberMaximum;
      TNumberMaximum = object( TNumber )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent );
         function    EvalNumber : Real; virtual;
   function    HermesDisplay : PChar;    virtual HermesDisplayDMT;
   function    Copy : Pointer;           virtual CopyDMT;
end;

{ ComputedNumber = total of DataList }
      PNumberTotal = ^TNumberTotal;
      TNumberTotal = object( TNumber )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent );
         function    EvalNumber : Real; virtual;
   function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
   function    Copy : Pointer;         virtual CopyDMT;
end;

{ ComputedNumber = product of DataList }
      PNumberProduct = ^TNumberProduct;
      TNumberProduct = object( TNumber )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent );
         function    EvalNumber : Real; virtual;
   function    HermesDisplay : PChar;   virtual HermesDisplayDMT;
   function    Copy : Pointer;          virtual CopyDMT;
end;

{ ComputedNumber = Number + Number }
      PNumberSum = ^TNumberSum;
      TNumberSum = object( TNumber )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent);
         function    EvalNumber : Real; virtual;
   function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
   function    Copy : Pointer;         virtual CopyDMT;
end;

{ ComputedNumber = Number - Number }
      PNumberDifference = ^TNumberDifference;
      TNumberDifference = object( TNumber )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent);
         function    EvalNumber : Real; virtual;
   function    HermesDisplay : PChar;   virtual HermesDisplayDMT;
   function    Copy : Pointer;          virtual CopyDMT;
end;

{ ComputedNumber = - Number }
      PNumberNegative = ^TNumberNegative;
      TNumberNegative = object( TNumber )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent );
         function    EvalNumber : Real; virtual;
   function    HermesDisplay : PChar;   virtual HermesDisplayDMT;
   function    Copy : Pointer;          virtual CopyDMT;
end;

{ ComputedNumber = Number x Number }
      PNumberTimes = ^TNumberTimes;
      TNumberTimes = object( TNumber )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent );
```

```
         function    EvalNumber : Real; virtual;
   function    HermesDisplay : PChar;   virtual HermesDisplayDMT;
   function    Copy : Pointer;          virtual CopyDMT;
end;

{ ComputedNumber = Number / Number }
      PNumberQuotient = ^TNumberQuotient;
      TNumberQuotient = object( TNumber )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent );
         function    EvalNumber : Real;        virtual;
   function    HermesDisplay : PChar;    virtual HermesDisplayDMT;
   function    Copy : Pointer;           virtual CopyDMT;
end;

{ ComputedNumber = Distance in Units between Graphic and Graphic [in Graphic] }
      PNumberDistance = ^TNumberDistance;
      TNumberDistance = object( TNumber )
         constructor Init( NewName : PChar;
                     NewTerm1 : Terminology; NewTerm2 : Terminology;
                     NewParam1 : PNodeContent; NewParam2 : PNodeContent;
                     NewParam3 : PNodeContent );
         function    EvalNumber : Real;   virtual;
   function    HermesDisplay : PChar;   virtual HermesDisplayDMT;
   function    Copy : Pointer;          virtual CopyDMT;
end;

      {***************************************
      *
      *    data structure for TBooleans
      *
      ****************************************}

{ SimpleBooleans = True }
PBooleanTrue = ^TBooleanTrue;
      TBooleanTrue = object( TBooleans )
         constructor Init(NewName : PChar);
         function    EvalBooleans : Boolean; virtual;
   function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
   function    Copy : Pointer;         virtual CopyDMT;
end;

{ SimpleBooleans = False }
PBooleanFalse = ^TBooleanFalse;
      TBooleanFalse = object( TBooleans )
         constructor Init(NewName : PChar);
         function    EvalBooleans : Boolean; virtual;
   function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
   function    Copy : Pointer;         virtual CopyDMT;
end;

{ ComputedBooleans = there are Counter DataList }
PBooleanCounter = ^TBooleanCounter;
      TBooleanCounter = object( TBooleans )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent );
         function    EvalBooleans : Boolean; virtual;
   function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
   function    Copy : Pointer;         virtual CopyDMT;
end;

{ ComputedBooleans = Quantifier DataList Filter }
PBooleanQuantifier = ^TBooleanQuantifier;
      TBooleanQuantifier = object( TBooleans )
         constructor Init( NewName : PChar;
                     NewParam1 : PNodeContent; NewParam2 : PNodeContent;
                     NewParam3 : PNodeContent );
         function    EvalBooleans : Boolean; virtual;
   function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
   function    Copy : Pointer;         virtual CopyDMT;
end;

{ ComputedBooleans = not Boolean }
PBooleanNot = ^TBooleanNot;
      TBooleanNot = object( TBooleans )
         constructor Init(NewName : PChar; NewParam1 : PNodeContent );
         function    EvalBooleans : Boolean; virtual;
   function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
```

```
     function    Copy : Pointer;         virtual CopyDMT;
  end;

  { ComputedBooleans = Boolean Connective Boolean }
  PBooleanConnective = ^TBooleanConnective;
       TBooleanConnective = object( TBooleans )
          constructor Init(NewName : PChar; NewParam1 : PNodeContent; NewTerm1 : Terminology;
                       NewParam2 : PNodeContent );
          function    EvalBooleans : Boolean; virtual;
     function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
     function    Copy : Pointer;         virtual CopyDMT;
  end;

  { ComputedBooleans = Graphic Measure Graphic [in Graphic] }
  PBooleanMeasure = ^TBooleanMeasure;
       TBooleanMeasure = object( TBooleans )
    constructor Init(NewName : PChar; NewParam1, NewParam2, NewParam3,
                     NewParam4, NewParam5 : PNodeContent );
         function    EvalBooleans : Boolean; virtual;
    function    HermesDisplay : PChar;  virtual HermesDisplayDMT;
    function    Copy : Pointer;         virtual CopyDMT;
  end;

       {=============  IMPLEMENTATION  ======================================}

implementation

uses Strings,
     HerBasic, HerLinks, HerNodes, HerGraph;

       {****************************************
       *
       *    function ListOfNumbers
       *    returns a list of all items that are TNumbers in DataList
  *    used internally to this Implementation
       ****************************************}

function ListOfNumbers(TheDataList : PNodeContent) : PHermesList;
var
  FullList :PHermesQueryList;
  NumberList : PHermesList;
    procedure IfNumeric(AnItem: PQueryListItem); far;
    var
      TheData : PPersistentObject;
    begin {IfNumeric}
(*
      TheData := DBGetId(AnItem^.GetDataId);
*)
      if (TheData^.GetType = ot_Number)
        then NumberList^.Insert(TheData);
    end; {IfNumeric}
begin
  NumberList := New(PHermesList, Init);             {create empty list}
  FullList := PDataList(TheDataList)^.Eval(nil);    {evaluate the DataList}
  FullList^.ForEach(@IfNumeric);                    {store numeric items in NumberList}
  ListOfNumbers := NumberList;
end;

       {****************************************
       *
       *    function ListOfGraphics
       *    returns a list of all items that are THermesGraphics in DataList
  *    used internally to this Implementation
       ****************************************}

function ListOfGraphics(TheDataList : PNodeContent) : PHermesList;
var
  FullList :PHermesQueryList;
  GraphicsList : PHermesList;
    procedure IfGraphic(AnItem: PQueryListItem); far;
    var
      TheData : PPersistentObject;
    begin {IfGraphic}
(*
      TheData := DBGetId(AnItem^.GetDataId);
*)
```

```
      if (TheData^.GetType = ot_HermesGraphic)
        then GraphicsList^.Insert(TheData);
    end; {IfGraphic}
begin
  GraphicsList := New(PHermesList, Init);             {create empty list}
  FullList := PDataList(TheDataList)^.Eval(nil);    {evaluate the DataList}
  FullList^.ForEach(@IfGraphic);                    {store numeric items in NumberList}
  ListOfGraphics := GraphicsList;
end;

        {***************************************
        *
        *    methods for TCharacters
        *
        ****************************************}

{ SimpleCharacter = character string }
constructor TCharacterSimple.Init(NewName, NewText: PChar);
begin
  inherited Init(NewName);
  SetText(NewText);
end;

function TCharacterSimple.EvalCharacter : PChar;
begin
  EvalCharacter := GetText;
end;

function TCharacterSimple.HermesDisplay : PChar;
var
  ThePChar  : array [0..2000] of Char;
begin
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, GetText);
  HermesDisplay := ThePChar;
end;

function TCharacterSimple.Copy : Pointer;
var
  TheObject : PCharacterSimple;
begin
  New(TheObject, Init(NoName, GetText));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;


  { ComputedCharacter = substring of Character from Number for Number }
constructor TCharacterSubstring.Init(NewName : PChar; NewParam1 : PNodeContent;
                                     NewParam2 : PNodeContent; NewParam3 : PNodeContent);
begin
  inherited Init(NewName);
  SetParam1(NewParam1);
  SetParam2(NewParam2);
  SetParam3(NewParam3);
end;

function TCharacterSubstring.EvalCharacter : PChar;
var
  TheString : String;
  TheStart  : Integer;
  TheLen    : Integer;
  TheCopy   : String;
  ThePChar  : array[0..79] of Char;
begin
  TheString := StrPas(PCharacter(GetParam1)^.EvalCharacter);
  TheStart  := Round(PNumber(GetParam2)^.EvalNumber);
  TheLen    := Round(PNumber(GetParam3)^.EvalNumber);
  TheCopy   := System.Copy(TheString, TheStart, TheLen);
  StrPCopy(ThePChar, TheCopy);
  EvalCharacter := ThePChar;
end;

function TCharacterSubstring.HermesDisplay : PChar;
var
  ThePChar  : array [0..2000] of Char;
```

```
begin
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, 'substring of');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  StrCat(ThePChar, ' from');
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  StrCat(ThePChar, ' for');
  StrCat(ThePChar, GetParam3^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function TCharacterSubstring.Copy : Pointer;
var
  TheObject : PCharacterSubstring;
begin
  New(TheObject, Init(NoName, GetParam1, GetParam2, GetParam3));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;


  { ComputedCharacter = Character append Character }
constructor TCharacterAppend.Init(NewName : PChar; NewParam1 : PNodeContent;
                                  NewParam2 : PNodeContent );
begin
  inherited Init(NewName);
  SetParam1(NewParam1);
  SetParam2(NewParam2);
end;

function TCharacterAppend.EvalCharacter : PChar;
var
  ThePChar : array [0..2000] of Char;
begin
  StrCopy(ThePChar, PCharacter(GetParam1)^.EvalCharacter);
  StrCat(ThePChar, PCharacter(GetParam2)^.EvalCharacter);
  EvalCharacter := ThePChar;
end;

function TCharacterAppend.HermesDisplay : PChar;
var
  ThePChar  : array [0..2000] of Char;
begin
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  StrCat(ThePChar, ' append');
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function TCharacterAppend.Copy : Pointer;
var
  TheObject : PCharacterAppend;
begin
  New(TheObject, Init(NoName, GetParam1, GetParam2));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

        {***************************************
        *
        *    methods for TNumbers
        *
        ***************************************}

{ NumberSimple = real }
constructor TNumberSimple.Init(NewName : PChar; NewReal : Real );
begin
  Inherited Init(NewName);
  SetReal(NewReal);
end;

function    TNumberSimple.EvalNumber : Real;
begin
  EvalNumber := GetReal;
end;
```

```
function    TNumberSimple.HermesDisplay : PChar;
var
  TheString  : String;
  TheTemp   : array[0..79] of Char;
  ThePChar   : array[0..79] of Char;
begin
  Str(EvalNumber:8:0, TheString);
  StrPCopy(TheTemp, TheString);
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, TheTemp);
  HermesDisplay := ThePChar;
end;

function   TNumberSimple.Copy : Pointer;
var
  TheObject : PNumberSimple;
begin
  New(TheObject, Init(NoName, GetReal));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{ NumberCount = count of DataList }
constructor TNumberCount.Init(NewName : PChar; NewParam1 : PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
end;

function    TNumberCount.EvalNumber : Real;
begin
  EvalNumber := (ListOfNumbers(GetParam1))^.Count;
end;

function    TNumberCount.HermesDisplay : PChar;
var
  ThePChar   : array[0..2000] of Char;
begin
  StrCopy(ThePChar, ' count of');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function   TNumberCount.Copy : Pointer;
var
  TheObject : PNumberCount;
begin
  New(TheObject, Init(NoName, GetParam1));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{NumberMinimum = minimum of DataList }
constructor TNumberMinimum.Init(NewName : PChar; NewParam1 : PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
end;

function    TNumberMinimum.EvalNumber : Real;
var
  NumberList : PHermesList;
  Min        : Real;
  Test       : Real;
    procedure ProcessOne(AnItem : PNumber); far;
    begin {ProcessOne}
      Test := AnItem^.EvalNumber;
      if (Test < Min)
        then Min := Test;
    end; {ProcessOne}
begin
  NumberList := ListOfNumbers(GetParam1);
  if (NumberList^.Count > 0)
    then
      begin
```

```
        Min := PNumber(NumberList^.FirstOne)^.EvalNumber;
        NumberList^.ForEach(@ProcessOne);
      end
    else Min := 0;
  EvalNumber := Min;
end;

function    TNumberMinimum.HermesDisplay : PChar;
var
  ThePChar   : array[0..79] of Char;
begin
  StrCopy(ThePChar, ' minimum of');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function   TNumberMinimum.Copy : Pointer;
var
  TheObject : PNumberMinimum;
begin
  New(TheObject, Init(NoName, GetParam1));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{ NumberMaximum = maximum of DataList }
constructor TNumberMaximum.Init(NewName : PChar; NewParam1 : PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
end;

function    TNumberMaximum.EvalNumber : Real;
var
  NumberList : PHermesList;
  Max        : Real;
  Test       : Real;
    procedure ProcessOne(AnItem : PNumber); far;
    begin {ProcessOne}
      Test := AnItem^.EvalNumber;
      if (Test > Max)
        then Max := Test;
    end; {ProcessOne}
begin
  NumberList := ListOfNumbers(GetParam1);
  if (NumberList^.Count > 0)
    then
      begin
        Max := PNumber(NumberList^.FirstOne)^.EvalNumber;
        NumberList^.ForEach(@ProcessOne);
      end
    else Max := 0;
  EvalNumber := Max;
end;

function    TNumberMaximum.HermesDisplay : PChar;
var
  ThePChar   : array[0..79] of Char;
begin
  StrCopy(ThePChar, ' maximum of');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function   TNumberMaximum.Copy : Pointer;
var
  TheObject : PNumberMaximum;
begin
  New(TheObject, Init(NoName, GetParam1));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{ NumberTotal = total of DataList }
constructor TNumberTotal.Init(NewName : PChar; NewParam1 : PNodeContent );
begin
```

```
  Inherited Init(NewName);
  SetParam1(NewParam1);
end;

function    TNumberTotal.EvalNumber : Real;
var
  NumberList : PHermesList;
  Total      : Real;
  Test       : Real;
    procedure ProcessOne(AnItem : PNumber); far;
    begin {ProcessOne}
      Test := AnItem^.EvalNumber;
      Total := Total + Test;
    end; {ProcessOne}
begin
  NumberList := ListOfNumbers(GetParam1);
  Total := 0;
  NumberList^.ForEach(@ProcessOne);
  EvalNumber := Total;
end;

function    TNumberTotal.HermesDisplay : PChar;
var
  ThePChar   : array[0..79] of Char;
begin
  StrCopy(ThePChar, ' total of');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function   TNumberTotal.Copy : Pointer;
var
  TheObject : PNumberTotal;
begin
  New(TheObject, Init(NoName, GetParam1));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{ NumberProduct = product of DataList }
constructor TNumberProduct.Init(NewName : PChar; NewParam1 : PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
end;

function    TNumberProduct.EvalNumber : Real;
var
  NumberList : PHermesList;
  Total      : Real;
  Test       : Real;
    procedure ProcessOne(AnItem : PNumber); far;
    begin {ProcessOne}
      Test := AnItem^.EvalNumber;
      Total := Total * Test;
    end; {ProcessOne}
begin
  NumberList := ListOfNumbers(GetParam1);
  Total := 0;
  NumberList^.ForEach(@ProcessOne);
  EvalNumber := Total;
end;

function    TNumberProduct.HermesDisplay : PChar;
var
  ThePChar   : array[0..79] of Char;
begin
  StrCopy(ThePChar, ' product of');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function   TNumberProduct.Copy : Pointer;
var
  TheObject : PNumberProduct;
begin
```

```
  New(TheObject, Init(NoName, GetParam1));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{ NumberSum = Number 1 + Number2 }
constructor TNumberSum.Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
  SetParam2(NewParam2);
end;

function    TNumberSum.EvalNumber : Real;
begin
  EvalNumber := PNumber(GetParam1)^.EvalNumber + PNumber(GetParam2)^.EvalNumber;
end;

function    TNumberSum.HermesDisplay : PChar;
var
  ThePChar   : array[0..79] of Char;
begin
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  StrCat(ThePChar, ' +');
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function   TNumberSum.Copy : Pointer;
var
  TheObject : PNumberSum;
begin
  New(TheObject, Init(NoName, GetParam1, GetParam2));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{ NumberDifference = Number1 - Number2 }
constructor TNumberDifference.Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 :
PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
  SetParam2(NewParam2);
end;

function    TNumberDifference.EvalNumber : Real;
begin
  EvalNumber := PNumber(GetParam1)^.EvalNumber - PNumber(GetParam2)^.EvalNumber;
end;

function    TNumberDifference.HermesDisplay : PChar;
var
  ThePChar   : array[0..79] of Char;
begin
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  StrCat(ThePChar, ' -');
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function   TNumberDifference.Copy : Pointer;
var
  TheObject : PNumberDifference;
begin
  New(TheObject, Init(NoName, GetParam1, GetParam2));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{ NumberNegative = - Number1 }
constructor TNumberNegative.Init(NewName : PChar; NewParam1 : PNodeContent );
begin
  Inherited Init(NewName);
```

```
  SetParam1(NewParam1);
end;

function    TNumberNegative.EvalNumber : Real;
begin
  EvalNumber := - PNumber(GetParam1)^.EvalNumber;
end;

function    TNumberNegative.HermesDisplay : PChar;
var
  ThePChar   : array[0..79] of Char;
begin
  StrCopy(ThePChar, ' -');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function   TNumberNegative.Copy : Pointer;
var
  TheObject : PNumberNegative;
begin
  New(TheObject, Init(NoName, GetParam1));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{ NumberTimes = Number1 x Number2 }
constructor TNumberTimes.Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
  SetParam2(NewParam2);
end;

function    TNumberTimes.EvalNumber : Real;
begin
  EvalNumber := PNumber(GetParam1)^.EvalNumber * PNumber(GetParam2)^.EvalNumber;
end;

function    TNumberTimes.HermesDisplay : PChar;
var
  ThePChar   : array[0..79] of Char;
begin
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  StrCat(ThePChar, ' x');
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function   TNumberTimes.Copy : Pointer;
var
  TheObject : PNumberTimes;
begin
  New(TheObject, Init(NoName, GetParam1, GetParam2));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{ NumberQuotient = Number1 / Number2 }
constructor TNumberQuotient.Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent
);
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
  SetParam2(NewParam2);
end;

function    TNumberQuotient.EvalNumber : Real;
begin
  EvalNumber := PNumber(GetParam1)^.EvalNumber / PNumber(GetParam2)^.EvalNumber;
end;

function    TNumberQuotient.HermesDisplay : PChar;
var
  ThePChar   : array[0..79] of Char;
```

```
begin
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  StrCat(ThePChar, ' /');
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function   TNumberQuotient.Copy : Pointer;
var
  TheObject : PNumberQuotient;
begin
  New(TheObject, Init(NoName, GetParam1, GetParam2));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

{ NumberDistance = Distance in Units between Graphic and Graphic [in Graphic] }
constructor TNumberDistance.Init(NewName : PChar;
                                 NewTerm1 : Terminology; NewTerm2 : Terminology;
                                 NewParam1 : PNodeContent; NewParam2 : PNodeContent;
                                 NewParam3 : PNodeContent );
begin
  Inherited Init(NewName);
  SetTerm1(NewTerm1);
  SetTerm2(NewTerm2);
  SetParam1(NewParam1);
  SetParam2(NewParam2);
  SetParam3(NewParam3);
end;

function    TNumberDistance.EvalNumber : Real;
begin
  EvalNumber := DistanceBetween(GetTerm1, GetTerm2, PHermesGraphic(GetParam1),
                PHermesGraphic(GetParam2), PHermesGraphic(GetParam3));
end;

function    TNumberDistance.HermesDisplay : PChar;
var
  ThePChar   : array[0..79] of Char;
begin
  StrCopy(ThePChar, ' list of');
  case GetTerm1 of
    CENTRAL_DISTANCE  : StrCat(ThePChar, ' central distances');
    CLOSEST_DISTANCE  : StrCat(ThePChar, ' closest distances');
    X_DISTANCE        : StrCat(ThePChar, ' x distances');
    Y_DISTANCE        : StrCat(ThePChar, ' y distances');
    Z_DISTANCE        : StrCat(ThePChar, ' z distances');
  end; {case}
  StrCat(ThePChar, ' in');
  case GetTerm2 of
    INCHES      : StrCat(ThePChar, ' inches');
    FEET        : StrCat(ThePChar, ' feet');
    CENTIMETERS : StrCat(ThePChar, ' cm.');
    METERS      : StrCat(ThePChar, ' meters');
  end; {case}
  StrCat(ThePChar, ' among');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  StrCat(ThePChar, ',');
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  if (GetParam3 <> nil)
  then
    begin
      StrCat(ThePChar, ' in');
      StrCat(ThePChar, GetParam3^.HermesDisplay);
    end; {if}
  HermesDisplay := ThePChar;
end;

function   TNumberDistance.Copy : Pointer;
var
  TheObject : PNumberDistance;
begin
  New(TheObject, Init(NoName, GetTerm1, GetTerm2, GetParam1, GetParam2, GetParam3));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
```

```
end;

        {*************************************
        *
        *    methods for TBooleans
        *
        *************************************}

  { BooleanTrue = True }
constructor TBooleanTrue.Init(NewName : PChar);
begin
  Inherited Init(NewName);
end;

function    TBooleanTrue.EvalBooleans : Boolean;
begin
  EvalBooleans := True;
end;

function    TBooleanTrue.HermesDisplay : PChar;
begin
  HermesDisplay := StrNew(' True');
end;

function    TBooleanTrue.Copy : Pointer;
var
  TheObject : PBooleanTrue;
begin
  New(TheObject, Init(NoName));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

  { BooleanFalse = False }
constructor TBooleanFalse.Init(NewName : PChar);
begin
  Inherited Init(NewName);
end;

function    TBooleanFalse.EvalBooleans : Boolean;
begin
  EvalBooleans := False;
end;

function    TBooleanFalse.HermesDisplay : PChar;
begin
  HermesDisplay := StrNew(' False');
end;

function    TBooleanFalse.Copy : Pointer;
var
  TheObject : PBooleanFalse;
begin
  New(TheObject, Init(NoName));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

  { BooleanCounter = there are Counter DataList }
constructor TBooleanCounter.Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 : PNodeContent
);
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
  SetParam2(NewParam2);
end;

function    TBooleanCounter.EvalBooleans : Boolean;
var
  TheCount : Integer;
begin
  TheCount := (PDataList(GetParam2)^.Eval(nil))^.Count;
  EvalBooleans := ( PCounter(GetParam1)^.EvalCounter(TheCount) );
end;

function    TBooleanCounter.HermesDisplay : PChar;
```

```
var
  ThePChar  : array [0..2000] of Char;
begin
  StrCopy(ThePChar, ' there are');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function    TBooleanCounter.Copy : Pointer;
var
  TheObject : PBooleanCounter;
begin
  New(TheObject, Init(NoName, GetParam1, GetParam2));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

  { BooleanQuantifier = Quantifier DataList Filter }
constructor TBooleanQuantifier.Init(NewName : PChar; NewParam1 : PNodeContent; NewParam2 :
PNodeContent;
                      NewParam3 : PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
  SetParam2(NewParam2);
  SetParam3(NewParam3);
end;

function    TBooleanQuantifier.EvalBooleans : Boolean;
var
  TheTotalList  : PHermesQueryList;
  TheTotal      : Real;
  TheFilterList : PHermesQueryList;
  TheCount      : Real;
begin
  TheTotalList  := PDataList(GetParam2)^.Eval(nil);
  TheTotal      := TheTotalList^.Count;
  TheFilterList := PFilter(GetParam3)^.Eval(TheTotalList);
  TheCount      := TheFilterList^.Count;
  EvalBooleans  := PQuantifier(GetParam1)^.EvalQuantifier(TheCount, TheTotal);
end;

function    TBooleanQuantifier.HermesDisplay : PChar;
var
  ThePChar  : array [0..2000] of Char;
begin
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  StrCat(ThePChar, GetParam3^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function    TBooleanQuantifier.Copy : Pointer;
var
  TheObject : PBooleanQuantifier;
begin
  New(TheObject, Init(NoName, GetParam1, GetParam2, GetParam3));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

  { BooleanNot = not Boolean }
constructor TBooleanNot.Init(NewName : PChar; NewParam1 : PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
end;

function    TBooleanNot.EvalBooleans : Boolean;
begin
  EvalBooleans := not PBooleans(GetParam1)^.EvalBooleans;;
end;

function    TBooleanNot.HermesDisplay : PChar;
```

```
var
  ThePChar  : array [0..2000] of Char;
begin
  StrCopy(ThePChar, ' not');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function    TBooleanNot.Copy : Pointer;
var
  TheObject : PBooleanNot;
begin
  New(TheObject, Init(NoName, GetParam1));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

  { BooleanConnective = Boolean Connective Boolean }
constructor TBooleanConnective.Init(NewName : PChar; NewParam1 : PNodeContent;
                                          NewTerm1 : Terminology;
                                          NewParam2 : PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
  SetTerm1(NewTerm1);
  SetParam2(NewParam2);
end;

function    TBooleanConnective.EvalBooleans : Boolean;
begin
  case GetTerm1 of
    AND_LOGICAL :   begin
                      EvalBooleans := ( PBooleans(GetParam1)^.EvalBooleans and
                                        PBooleans(GetParam2)^.EvalBooleans );
                    end;
    OR_LOGICAL  :   begin
                      EvalBooleans := ( PBooleans(GetParam1)^.EvalBooleans or
                                        PBooleans(GetParam2)^.EvalBooleans );
                    end;
  end; {case}
end;

function    TBooleanConnective.HermesDisplay : PChar;
var
  ThePChar  : array [0..2000] of Char;
begin
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  case GetTerm1 of
    AND_LOGICAL : StrCat(ThePChar, ' and (logical)');
    OR_LOGICAL  : StrCat(ThePChar, ' or (logical)');
  end; {Case}
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  HermesDisplay := ThePChar;
end;

function    TBooleanConnective.Copy : Pointer;
var
  TheObject : PBooleanConnective;
begin
  New(TheObject, Init(NoName, GetParam1, GetTerm1, GetParam2));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

  { BooleanMeasure = Graphic Measure [Quantifier] Graphic [in Graphic] }
constructor TBooleanMeasure.Init(NewName : PChar; NewParam1, NewParam2, NewParam3,
                                 NewParam4, NewParam5 : PNodeContent );
begin
  Inherited Init(NewName);
  SetParam1(NewParam1);
  SetParam2(NewParam2);
  SetParam3(NewParam3);
  SetParam4(NewParam4);
  SetParam5(NewParam5);
end;
```

```
function    TBooleanMeasure.EvalBooleans : Boolean;
var
  TheCounter   : Integer;
  TheTotal     : Integer;
  GraphicsList : PHermesList;
    procedure ProcessOne(AnItem : PHermesGraphic); far;
    begin {ProcessOne}
      if ( PMeasure(GetParam2)^.EvalMeasure(PHermesGraphic(GetParam1), AnItem,
PHermesGraphic(GetParam5)) )
        then Inc(TheCounter);                 {count how many satisfy Measure}
    end; {ProcessOne}
begin
  GraphicsList := ListOfGraphics(GetParam4);   {get items from DataList that are THermesGraphics}
  TheCounter := 0;
  TheTotal := GraphicsList^.Count;
  GraphicsList^.ForEach(@ProcessOne);
  if ( PQuantifier(GetParam3) <> nil )                        {if optional Quantifier is defined}
    then EvalBooleans := PQuantifier(GetParam3)^.EvalQuantifier(TheCounter, TheTotal)
    else EvalBooleans := (TheCounter > 0);
end;

function    TBooleanMeasure.HermesDisplay : PChar;
var
  ThePChar  : array [0..2000] of Char;
begin
  StrCopy(ThePChar, ' ');
  StrCat(ThePChar, GetParam1^.HermesDisplay);
  StrCat(ThePChar, GetParam2^.HermesDisplay);
  if (GetParam3 <> nil)
    then StrCat(ThePChar, GetParam3^.HermesDisplay);
  if (GetParam4 <> nil)
    then StrCat(ThePChar, GetParam4^.HermesDisplay);
  if (GetParam5 <> nil)
    then
      begin
        StrCat(ThePChar, ' in');
        StrCat(ThePChar, GetParam5^.HermesDisplay);
      end;
  HermesDisplay := ThePChar;
end;

function    TBooleanMeasure.Copy : Pointer;
var
  TheObject : PBooleanMeasure;
begin
  New(TheObject, Init(NoName, GetParam1, GetParam2, GetParam3, GetParam4, GetParam5));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;


end.    {HerMedia}
```

```
 {*****************************************
 *
 *      HerLangu.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 ******************************************}

unit HerLangu;

interface

uses HerLists, HerHyper;

 { This unit defines the options for DataLists, Associations, and Filters in the Hermes language.
   These options were proposed in (Stahl, Ph.D. dissertation, 1993).

  DESIGN:
    When navigating through hypertext it is possible to encounter cycles.  A query that does this
will not
  terminate gracefully unless a check is made.  The function Cycles does this check.  It is called
by:
        TAssociationType.Eval     TAssociationInverseType.Eval
        TAssociationPart.Eval     TAssociationInversePart.Eval

  It is possible to define a Relationship either directly or indirectly in terms of itself.  This is
  useful for recursive definitions.  However, such recursive definitions can be ill-formed.
  A check is therefore made to ensure that the number of recusive calls is not unbounded.
  A limit to the number of recursive calls is set as a user preference.  This is checked in
  the same procedures.

        MODIFICATIONS:
        To add a new option to the syntax of the language:
        Define the Data Structure and the Methods in the appropriate unit.
        Add the object two places in HerRegis, assigning a unique number.
        Add the object to the interface.  Test the new option.
 }

type

        {***************************************
        *
        *    data structure for DataLists
        *
        ****************************************}

  {*** TDataListNodeKind = NodeKind ***}
   PDataListNodeKind      = ^TDataListNodeKind;
   TDataListNodeKind = object (TDataList)
      constructor Init(NewName : PChar; ANodeKind : PChar);
      function Copy : Pointer; virtual CopyDMT;
      function Eval(InputList: PHermesQueryList) : PHermesQueryList; virtual EvalDMT;
      function HermesDisplay: PChar; virtual HermesDisplayDMT;
       end;

  {*** TDataListOf = Association of DataList ***}
   PDataListOf      = ^TDataListOf;
   TDataListOf = object (TDataList)
      constructor Init(NewName : PChar; AnAssociation : PNodeContent; ADataList : PNodeContent);
      function Copy : Pointer; virtual CopyDMT;
      function Eval(InputList: PHermesQueryList) : PHermesQueryList; virtual EvalDMT;
      function HermesDisplay: PChar; virtual HermesDisplayDMT;
       end;

  {*** TDataListId = id: object id ***}
   PDataListId      = ^TDataListId;
   TDataListId = object (TDataList)
      constructor Init(NewName : PChar; AnId : LongInt);
      function Copy : Pointer; virtual CopyDMT;
      function Eval(InputList: PHermesQueryList) : PHermesQueryList; virtual EvalDMT;
```

```
      function HermesDisplay: PChar; virtual HermesDisplayDMT;
       end;

       {****************************************
       *
       *    data structure for Associations
       *
       ****************************************}

  {*** TAssociationType = LinkType ***}
   PAssociationType      = ^TAssociationType;
   TAssociationType = object (TAssociation)
      constructor Init(NewName : PChar; AType : PChar);
      function Copy : Pointer; virtual CopyDMT;
      function Eval(InputList: PHermesQueryList) : PHermesQueryList; virtual EvalDMT;
      function HermesDisplay: PChar; virtual HermesDisplayDMT;
       end;

  {*** TAssociationAll = all associations ***}
   PAssociationAll      = ^TAssociationAll;
   TAssociationAll = object (TAssociation)
      constructor Init(NewName : PChar);
      function Copy : Pointer; virtual CopyDMT;
      function Eval(InputList: PHermesQueryList) : PHermesQueryList; virtual EvalDMT;
      function HermesDisplay: PChar; virtual HermesDisplayDMT;
       end;

  {*** TAssociationOf = Association of Association ***}
   PAssociationOf      = ^TAssociationOf;
   TAssociationOf = object (TAssociation)
      constructor Init(NewName : PChar; Assoc1, Assoc2 : PNodeContent);
      function Copy : Pointer; virtual CopyDMT;
      function Eval(InputList: PHermesQueryList) : PHermesQueryList; virtual EvalDMT;
      function HermesDisplay: PChar; virtual HermesDisplayDMT;
       end;

  {*** TAssociationWith = Association with their Association ***}
   PAssociationWith      = ^TAssociationWith;
   TAssociationWith = object (TAssociation)
      constructor Init(NewName : PChar; Assoc1, Assoc2 : PNodeContent);
      function Copy : Pointer; virtual CopyDMT;
      function Eval(InputList: PHermesQueryList) : PHermesQueryList; virtual EvalDMT;
      function HermesDisplay: PChar; virtual HermesDisplayDMT;
       end;

  {*** TAssociationComb = Association Combination Association ***}
   PAssociationComb      = ^TAssociationComb;
   TAssociationComb = object (TAssociation)
      constructor Init(NewName : PChar; Assoc1 : PNodeContent; Comb : Terminology; Assoc2 :
PNodeContent);
      function Copy : Pointer; virtual CopyDMT;
      function Eval(InputList: PHermesQueryList) : PHermesQueryList; virtual EvalDMT;
      function HermesDisplay: PChar; virtual HermesDisplayDMT;
       end;

       {****************************************
       *
       *    data structure for Filters
       *
       ****************************************}

       {=============  IMPLEMENTATION  =====================================}

implementation

uses Strings,
     HerBasic, HerNodes, HerLinks, HerPersp, HerDataB, HerWorld;

const
  TheStack : Integer = 0;  {this typed constant is initialized to 0
                            at start of program.  Then it is treated as
                            a stack counter by several Eval methods.
                            It is incremented and decremented as Eval is
                            entered and exited.}

{ All macro and predicate processing must come thru certain Eval methods.
```

```
  So, all infinite looping can be trapped there.  Three methods are used:
  1.)  The variable Stack is incremented each time thru there, to count
       how many times it is called recursively by macro expansion.
       A user-preference determines a maximum count, to prevent run-time error.
  2.)  Function Cycles is called to make sure a node is not added to the
       output list twice at the same level, which would mean a loop in
       the hypertext network.
  3.)  The InputList is checked for being empty.  If it is empty, macro processing
       stops to avoid infinite looping at the ends of recursive search trees.
}

function Cycles(TheList: PHermesQueryList; TheItem: PQueryListItem): Boolean;
 {return True if adding TheItem to TheList would create a cycle
  This function is equivalent to maintaining and checking a stack,
  which when an TheItem was being added to the TheList at indent level N
  would contain N items: the last one at each of levels N-1, N-2, ..., 0.
  So to check if any one of these N items has the same Id and Context as TheItem
  and therefore would represent a cycle which would create an
  infinite list, we start at the end of TheList and find the first
  item with level= N-1 and compare its Id and Context to TheItem's Id and
  Context.  If the Ids and the Contexts are
  the same, TheItem is already on TheList and we don't want to add it again.
  Otherwise we continue back up TheList to the first at N-2 ....}
var
  TheId      : LongInt;
  TheLevel   : Integer;
  TheContext : Integer;
  TheIndex   : Integer;
    function RecurseCycles(AnIndex : Integer; ALevel : Integer) : Boolean;
    begin
      RecurseCycles := False;                                    {set default return value}
      while ((AnIndex >= 0) and not (PQueryListItem(TheList^.At(AnIndex))^.GetLevel = ALevel-1))
        do Dec(AnIndex);
      if (AnIndex >= 0) then                                     {if not finished and found
ALevel-1}
        if ((PQueryListItem(TheList^.At(AnIndex))^.GetDataId = TheId)
        and (PQueryListItem(TheList^.At(AnIndex))^.GetContext = TheContext))
          then
            RecurseCycles := True                                {found cycle; return}
          else                                                   {else, recurse}
            RecurseCycles := RecurseCycles(AnIndex - 1, ALevel - 1);
    end;  {RecurseCycles}
begin
  TheId      := TheItem^.GetDataId;
  TheLevel   := TheItem^.GetLevel;                  {TheItem is at indent level = TheLevel}
  TheContext := TheItem^.GetContext;
  TheIndex   := TheList^.Count - 2;                 {TheItem is at Index = Count-1 in TheList}
  Cycles     := RecurseCycles(TheIndex, TheLevel);
end;

procedure StackOverflow;
var
  TheResponse : Integer;
begin  {StackOverflow}
  TheStack := 0;        {reset StackCount}
  ErrorMsg( et_WARNING, 'StackLimit exceeded -- probably a circular reference');
  Halt(1);           {could add a dialog for user to choose whether to halt}
end;    {StackOverflow}

procedure IncStack;
begin
  Inc(TheStack);
  if (TheStack > HermesApp.GetStackLimit)
    then
      StackOverflow;
end;

        {****************************************
        *
        *    methods for DataLists
        *
        ****************************************}

  {*** TDataListNodeKind = NodeKind ***}
constructor TDataListNodeKind.Init(NewName : PChar; ANodeKind : PChar);
begin
```

```
  inherited Init(NewName);
  if assigned(ANodeKind)
    then SetText(ANodeKind);
end;

function TDataListNodeKind.Copy : Pointer;
var
  TheObject : PDataListNodeKind;
begin
       New(TheObject, Init(NoName, nil));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

function TDataListNodeKind.Eval(InputList: PHermesQueryList) : PHermesQueryList;
begin
                          (* ??? *)
end;

function TDataListNodeKind.HermesDisplay: PChar;
begin
  HermesDisplay := GetText;
end;

  {*** TDataListOf = Association of DataList ***}
constructor TDataListOf.Init(NewName : PChar; AnAssociation : PNodeContent; ADataList :
PNodeContent);
begin
  inherited Init(NewName);
  if assigned(AnAssociation)
    then SetParam1(AnAssociation);
  if assigned(ADataList)
    then SetParam2(ADataList);
end;

function TDataListOf.Copy : Pointer;
var
  TheObject : PDataListOf;
begin
       New(TheObject, Init(NoName, nil, nil));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

function TDataListOf.Eval(InputList: PHermesQueryList) : PHermesQueryList;
var
  TheList : PHermesQueryList;
  TheAssoc : PAssociation;
  TheDataList : PDataList;
begin
  TheAssoc := PAssociation(GetParam1);
  TheDataList := PDataList(GetParam2);
  if assigned(TheDataList)
    then
      TheList := TheDataList^.Eval(InputList)
    else
      if assigned(InputList)
        then
          TheList := InputList
        else
            TheList := New(PHermesQueryList, Init);
  if assigned(TheAssoc)
    then
      Eval := TheAssoc^.Eval(TheList)
    else
      Eval := TheList;
end;

function TDataListOf.HermesDisplay: PChar;
var
  DisplayArray : Array[0..2000] of Char;
  TheAssoc : PAssociation;
  TheDataList : PDataList;
begin
  TheAssoc := PAssociation(GetParam1);
  TheDataList := PDataList(GetParam2);
```

```
  if assigned(TheAssoc)
    then StrCopy(DisplayArray, TheAssoc^.HermesDisplay)
    else StrCopy(DisplayArray, ' ');
  StrCat(DisplayArray, ' of');
  if assigned(TheDataList)
    then StrCopy(DisplayArray, TheDataList^.HermesDisplay)
    else StrCopy(DisplayArray, ' ');
  HermesDisplay := StrNew(DisplayArray);
end;


  {*** TDataListId = id: object id ***}
constructor TDataListId.Init(NewName : PChar; AnId : LongInt);
begin
  inherited Init(NewName);
  SetContentId(AnId);
end;

function TDataListId.Copy : Pointer;
var
  TheObject : PDataListId;
begin
        New(TheObject, Init(NoName, 0));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

function TDataListId.Eval(InputList: PHermesQueryList) : PHermesQueryList;
var
  NodeList : PHermesQueryList;
  ListItem : PQueryListItem;
begin
  NodeList := New(PHermesQueryList, Init);
  ListItem := New(PQueryListItem, Init);
  ListItem^.SetContext(HermesApp.GetContext);      {save the current context}
  ListItem^.SetDataId(GetContentId);                       {store the Id}
  ListItem^.SetLevel(0);
  ListItem^.SetHeading(NoName);
  ListItem^.SetSequence(1);
  NodeList^.Insert(ListItem);                      {store  node for output}
  Eval := NodeList;
end;

function TDataListId.HermesDisplay: PChar;
var
  TheString : String[79];
  ThePChar  : array [0..79] of Char;
begin
  Str(GetContentId, TheString);
  StrPCopy(ThePChar, TheString);
  HermesDisplay := ThePChar;
end;


        {****************************************
        *
        *    methods for Associations
        *
        ****************************************}

  {*** TAssociationType = LinkType ***}
constructor TAssociationType.Init(NewName : PChar; AType : PChar);
begin
  inherited Init(NewName);
  if assigned(AType)
    then SetText(AType);
end;

function TAssociationType.Copy : Pointer;
var
  TheObject : PAssociationType;
begin
        New(TheObject, Init(NoName, NoName));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;
```

```
function TAssociationType.Eval(InputList: PHermesQueryList) : PHermesQueryList;
{Check for possible looping with function Cycles and StackCount}
var
  TheTypeId  : LongInt;
  MainType   : PChar;
  TheType        : PChar;
  LOut           : PHermesQueryList;
  TheKey         : LongInt;
  SwitchKey      : LongInt;
  TheMsg         : array[0..80] of Char;
  TheAssociation : PAssociation;
  ThePredicate   : PPredicate;

    procedure EvalOneNode(Item : PQueryListItem); far;
    var
      TheLevel    : Integer;
      TheSequence : Integer;
      TheTempNode : PPrimitiveObject;
      LinksFrom   : PLinkTree;
      ListOfType  : PLinkList;
      Index       : Integer;
      TestLinkList: PLinkList;

        procedure DoALink(ALinkId : PLinkId); far;
        var
          TheLink  : PLink;
          TheToPO  : LongInt;
          TheNode  : PHermesNode;
          NodeList : PHermesQueryList;
          ListItem : PQueryListItem;

            procedure DoSubLink(ASubLink : PSubLink); far;
            begin  {DoSubLink}
              SwitchKey := TheKey;                        {initialize to node's context}
              if TraversesSubLink(SwitchKey, ASubLink)    {note - SwitchKey is a var parameter}
                then                                      {if this sublink is valid for SwitchKey}
                  begin
                    NodeList := New(PHermesQueryList, Init);
                    ListItem := New(PQueryListItem, Init);
                    ListItem^.SetContext(SwitchKey);      {save the current context}
                    ListItem^.SetDataId(TheNode^.GetId);       {store the ToPO node}
                    ListItem^.SetLevel(TheLevel);         {saved from InputList}
                    ListItem^.SetHeading(TheType);        {Type called for}
                    ListItem^.SetSequence(TheSequence);   {position in ListOfType}
                    if not Cycles(LOut, ListItem)         {check for cycles in output list}
                      then NodeList^.Insert(ListItem);    {store another node for output}
                 (*   NodeList := TheNode^.Eval(NodeList);  {eval Cond or Virtual node}   *)
                    LOut^.Append(NodeList);               {add the node to output list}
                    Dispose(NodeList, Done);              {Append makes copies of the items}
                  end;
            end;   {DoSubLink}

        begin  {DoALink}                        {process a link of type T from node}
          Inc(TheSequence);                     {increment TheSequence count}
          TheLink := PLink(DBGet(ALinkId^.GetId));
          SwitchKey := TheKey;                  {initialize to node's context}
          if TraversesLink(SwitchKey, TheLink)
            then                                {if there is a valid sublink in the link}
              begin
                TheToPO := TheLink^.GetToPO;                {get id of ToNode}
                TheNode := PHermesNode(DBGet(TheToPO));
                TheLink^.GetSubLinks^.ForEach(@DoSubLink);   {process each SubLink}
              end;
        end;   {DoALink}

    begin  {EvalOneNode}                        {evaluate Type of one node on InputList}
      TheLevel := Item^.GetLevel;               {save the Level of the node}
      TheKey := Item^.GetContext;               {save the Context of the node}
      TheTempNode := PPrimitiveObject(DBGet(Item^.GetDataId));   {get the Node}
      LinksFrom := TheTempNode^.GetOutLinks;    {get its LinkTree}
      if assigned(LinksFrom)
      then
        begin
          TestLinkList := New(PLinkList, Init);
          TestLinkList^.SetName(MainType);
```

```
          if LinksFrom^.Search(TestLinkList, Index)  {search it for list of MainType}
            then
              begin
                TheSequence := 0;                    {initialize TheSequence count}
                ListOfType := LinksFrom^.At(Index);  {get list of LinkIds of MainType}
                ListOfType^.ForEach(@DoALink);       {process each Link of this Type}
              end;
          dispose(TestLinkList, Done);
        end;
    end;   {EvalOneNode}

begin  {Eval}
  IncStack;                                  {increment and check stack counter on recursive entry}
  TheType := GetText;
  LOut := New(PHermesQueryList, Init);       {initialize output list}
  TheAssociation := PAssociation(DBGetNamedObject(TheType, ot_Association, HermesApp.GetContext));
  if assigned(TheAssociation)                         {first process as named Association}
    then LOut := TheAssociation^.Eval(InputList)
    else
      begin
        ThePredicate := PPredicate(DBGetNamedObject(TheType, ot_Predicate, HermesApp.GetContext));
        if assigned(ThePredicate)                       {second process as named Predicate}
          then LOut := ThePredicate^.Eval(InputList)
          else
            begin
              TheTypeId := DBGetSynonymId(TheType, MainType, ot_LinkType);
              if (TheTypeId > 0)                        {third process as LinkType}
                then InputList^.ForEach(@EvalOneNode)
                else
                  begin
                    StrCopy(TheMsg, 'EType: Type not found -- ');
                    StrCat(TheMsg, TheType);
                    ErrorMsg(et_NOTICE, TheMsg);         {fourth process as error}
                  end;
            end;
      end;
  Eval := LOut;
  Dec(TheStack);                             {decrement stack counter on recursive exit}
end;

function TAssociationType.HermesDisplay: PChar;
begin
  HermesDisplay := GetText;
end;



  {*** TAssociationAll = all associations ***}
constructor TAssociationAll.Init(NewName : PChar);
begin
  inherited Init(NewName);
end;

function TAssociationAll.Copy : Pointer;
var
  TheObject : PAssociationAll;
begin
       New(TheObject, Init(NoName));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

function TAssociationAll.Eval(InputList: PHermesQueryList) : PHermesQueryList;
{Check for possible looping with function Cycles and StackCount}
var
  TheTypeId  : LongInt;
  MainType   : PChar;
  TheType        : PChar;
  LOut           : PHermesQueryList;
  TheKey         : LongInt;
  SwitchKey      : LongInt;
  TheMsg         : array[0..80] of Char;
  TheAssociation : PAssociation;
  ThePredicate   : PPredicate;

    procedure EvalOneNode(Item : PQueryListItem); far;
```

```
    var
      TheLevel    : Integer;
      TheSequence : Integer;
      TheTempNode : PHermesNode;
      LinksFrom   : PLinkTree;
      ListOfType  : PLinkList;
      Index       : Integer;
      TestLinkList: PLinkList;

       procedure DoAType(AList : PLinkList); far;

        procedure DoALink(ALinkId : PLinkId); far;
        var
          TheLink  : PLink;
          TheToPO  : LongInt;
          TheNode  : PHermesNode;
          NodeList : PHermesQueryList;
          ListItem : PQueryListItem;

            procedure DoSubLink(ASubLink : PSubLink); far;
            begin  {DoSubLink}
              SwitchKey := TheKey;                        {initialize to node's context}
              if TraversesSubLink(SwitchKey, ASubLink)    {note - SwitchKey is a var parameter}
                then                                      {if this sublink is valid for SwitchKey}
                  begin
                    NodeList := New(PHermesQueryList, Init);
                    ListItem := New(PQueryListItem, Init);
                    ListItem^.SetContext(SwitchKey);      {save the current context}
                    ListItem^.SetDataId(TheNode^.GetId);       {store the ToPO node}
                    ListItem^.SetLevel(TheLevel);         {saved from InputList}
                    ListItem^.SetHeading(TheType);        {Type called for}
                    ListItem^.SetSequence(TheSequence);   {position in ListOfType}
                    if not Cycles(LOut, ListItem)         {check for cycles in output list}
                      then NodeList^.Insert(ListItem);    {store another node for output}
                 (*   NodeList := TheNode^.Eval(NodeList);  {eval Cond or Virtual node}   *)
                    LOut^.Append(NodeList);               {add the node to output list}
                    Dispose(NodeList, Done);              {Append makes copies of the items}
                  end;
            end;   {DoSubLink}

        begin  {DoALink}                          {process a link of type T from node}
          Inc(TheSequence);                       {increment TheSequence count}
          TheLink := PLink(DBGet(ALinkId^.GetId));
          SwitchKey := TheKey;                    {initialize to node's context}
          if TraversesLink(SwitchKey, TheLink)
            then                                  {if there is a valid sublink in the link}
              begin
                TheToPO := TheLink^.GetToPO;                 {get id of ToNode}
                TheNode := PHermesNode(DBGet(TheToPO));
                TheLink^.GetSubLinks^.ForEach(@DoSubLink);   {process each SubLink}
              end;
        end;   {DoALink}

       begin  {DoAType}                          {process a Type from node}
         TheSequence := 0;                       {initialize TheSequence count}
         AList^.ForEach(@DoAType);               {process each Link of this Type}
       end;   {DoAType}

    begin  {EvalOneNode}                          {evaluate Type of one node on InputList}
      TheLevel := Item^.GetLevel;                 {save the Level of the node}
      TheKey := Item^.GetContext;                 {save the Context of the node}
      TheTempNode := PHermesNode(Item^.GetDataId);   {get the Node}
      LinksFrom := TheTempNode^.GetOutLinks;      {get its LinkTree}
      TheSequence := 0;                           {initialize TheSequence count}
      LinksFrom^.ForEach(@DoAType);               {process LinkList of each Type}
    end;   {EvalOneNode}

begin  {Eval}
  IncStack;                                  {increment and check stack counter on recursive entry}
  LOut := New(PHermesQueryList, Init);       {initialize output list}
  InputList^.ForEach(@EvalOneNode);
  Eval := LOut;
  Dec(TheStack);                            {decrement stack counter on recursive exit}
end;

function TAssociationAll.HermesDisplay: PChar;
```

```
var
  Temp : Array [0..2000] of Char;
begin
  StrCopy(Temp, ' everything ');
  HermesDisplay := Temp;
end;

  {*** TAssociationOf = Association of Association ***}
constructor TAssociationOf.Init(NewName : PChar; Assoc1, Assoc2 : PNodeContent);
begin
  inherited Init(NewName);
  SetParam1(Assoc1);
  SetParam2(Assoc2);
end;

function TAssociationOf.Copy : Pointer;
var
  TheObject : PAssociationOf;
begin
       New(TheObject, Init(NoName, nil, nil));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

function TAssociationOf.Eval(InputList: PHermesQueryList) : PHermesQueryList;
var
  OutputList : PHermesQueryList;
  FirstList  : PHermesQueryList;
  Assoc1 : PAssociation;
  Assoc2 : PAssociation;
begin  {Eval}
  OutputList := New(PHermesQueryList, Init);
  Assoc1 := PAssociation(GetParam1);
  Assoc2 := PAssociation(GetParam2);
  if assigned(Assoc2)
    then
      begin
        FirstList := Assoc2^.Eval(InputList);
        if assigned(Assoc1)
          then
            OutputList := Assoc1^.Eval(FirstList)
          else
            OutputList := FirstList;
      end
    else
      if assigned(Assoc1)
        then OutputList := Assoc1^.Eval(InputList);
  Eval := OutputList;
end;

function TAssociationOf.HermesDisplay: PChar;
var
  Temp : Array [0..2000] of Char;
  Assoc1 : PAssociation;
  Assoc2 : PAssociation;
begin
  Assoc1 := PAssociation(GetParam1);
  Assoc2 := PAssociation(GetParam2);
  StrCopy(Temp, ' ');
  if assigned(Assoc1)
    then
      begin
        StrCat(Temp, Assoc1^.HermesDisplay);
        if assigned(Assoc2)
          then
            begin
              StrCat(Temp, ' of');
              StrCat(Temp, Assoc2^.HermesDisplay);
            end;
      end
    else
      if assigned(Assoc2)
        then StrCat(Temp, Assoc2^.HermesDisplay);
  HermesDisplay := Temp;
end;
```

```
  {*** TAssociationWith = Association with their Association ***}
constructor TAssociationWith.Init(NewName : PChar; Assoc1, Assoc2 : PNodeContent);
begin
  inherited Init(NewName);
  SetParam1(Assoc1);
  SetParam2(Assoc2);
end;

function TAssociationWith.Copy : Pointer;
var
  TheObject : PAssociationWith;
begin
        New(TheObject, Init(NoName, nil, nil));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

function TAssociationWith.Eval(InputList: PHermesQueryList) : PHermesQueryList;
var
  OutputList : PHermesQueryList;
  FirstList  : PHermesQueryList;
  Assoc1 : PAssociation;
  Assoc2 : PAssociation;

    procedure DoOne(TheItem : PQueryListItem); far;
    var
      Sublist    : PHermesQueryList;
      SecondList : PHermesQueryList;

        procedure IndentOne(AnItem : PQueryListItem); far;
        begin  {IndentOne}
          AnItem^.SetLevel(AnItem^.GetLevel + TheItem^.GetLevel + 1);     {indent}
          OutputList^.Insert(AnItem);                                  {then insert it in output}
        end;   {IndentOne}

    begin  {DoOne}
      OutputList^.Insert(TheItem^.Copy);            {insert a copy of the item in the output list}
      Sublist := New(PHermesQueryList, Init);
      Sublist^.Insert(TheItem^.Copy);               {make a new list with just the item}
      SecondList := Assoc2^.Eval(Sublist);          {eval Assoc2 on this new list}
      SecondList^.ForEach(@IndentOne);              {indent each result and insert it in output}
      Dispose(Sublist, Done);
    end;   {DoOne}

begin  {Eval}
  OutputList := New(PHermesQueryList, Init);
  Assoc1 := PAssociation(GetParam1);
  Assoc2 := PAssociation(GetParam2);
  if assigned(Assoc1)
    then
      begin
        FirstList := Assoc1^.Eval(InputList);
        if assigned(Assoc2)
          then
            FirstList^.ForEach(@DoOne)
          else
            OutputList := FirstList;
      end;
  Eval := OutputList;
end;

function TAssociationWith.HermesDisplay: PChar;
var
  Temp : Array [0..2000] of Char;
  Assoc1 : PAssociation;
  Assoc2 : PAssociation;
begin
  Assoc1 := PAssociation(GetParam1);
  Assoc2 := PAssociation(GetParam2);
  StrCopy(Temp, ' ');
  if assigned(Assoc1)
    then
      begin
        StrCat(Temp, Assoc1^.HermesDisplay);
        if assigned(Assoc2)
          then
```

```
            begin
              StrCat(Temp, ' with their ');
              StrCat(Temp, Assoc2^.HermesDisplay);
            end;
      end
    else
      if assigned(Assoc2)
        then StrCat(Temp, Assoc2^.HermesDisplay);
  HermesDisplay := Temp;
end;

  {*** TAssociationComb = Association Combination Association ***}
constructor TAssociationComb.Init(NewName : PChar; Assoc1 : PNodeContent; Comb : Terminology; Assoc2
: PNodeContent);
begin
  inherited Init(NewName);
  SetParam1(Assoc1);
  SetTerm1(Comb);
  SetParam2(Assoc2);
end;

function TAssociationComb.Copy : Pointer;
var
  TheObject : PAssociationComb;
begin
       New(TheObject, Init(NoName, nil, unassigned, nil));
  TheObject^.CopyFrom(@Self);
  Copy := TheObject;
end;

function TAssociationComb.Eval(InputList: PHermesQueryList) : PHermesQueryList;
var
  Assoc1 : PAssociation;
  Comb   : Terminology;
  Assoc2 : PAssociation;
  HQList : PHermesItemsList;
  OutList : PHermesItemsList;
begin
  Assoc1 := PAssociation(GetParam1);
  Comb   := GetTerm1;
  Assoc2 := PAssociation(GetParam2);
  HQList := New(PHermesQueryList, Init);
  if assigned(Assoc1)
    then
      begin
        HQList := Assoc1^.Eval(InputList);
        if assigned(Assoc2)
          then
             case Comb of
                AND_UNION     :  OutList := HQList^.Union(Assoc2^.Eval(InputList));          {unique
sorted union}
                AND_ALSO      :  OutList := HQList^.Intersection(Assoc2^.Eval(InputList));
{intersection}
                BUT_NOT       :  OutList := HQList^.Difference(Assoc2^.Eval(InputList));
{difference}
                OR_APPEND     :  OutList := HQList^.Append(Assoc2^.Eval(InputList));         {append}
             end;  {case}
      end
    else
      if assigned(Assoc2)
        then
          Eval := Assoc2^.Eval(InputList)
        else
          Eval := New(PHermesQueryList, Init);
end;

function TAssociationComb.HermesDisplay: PChar;
var
  Temp : Array [0..2000] of Char;
  Assoc1 : PAssociation;
  Assoc2 : PAssociation;
  Comb   : Terminology;
begin
  Assoc1 := PAssociation(GetParam1);
  Assoc2 := PAssociation(GetParam2);
  Comb   := GetTerm1;
```

```
  StrCopy(Temp, ' ');
  if assigned(Assoc1)
    then
      begin
        StrCat(Temp, Assoc1^.HermesDisplay);
        if assigned(Assoc2)
          then
            begin
              case Comb of
                AND_UNION     :  StrCat(Temp, ' and');        {unique sorted union}
                AND_ALSO      :  StrCat(Temp, ' and also');   {intersection}
                BUT_NOT       :  StrCat(Text, ' but not');    {difference}
                OR_APPEND     :  StrCat(Temp, ' or');         {append}
              end;  {case}
              StrCat(Temp, Assoc2^.HermesDisplay);
            end;
      end
    else
      if assigned(Assoc2)
        then StrCat(Temp, Assoc2^.HermesDisplay);
  HermesDisplay := Temp;
end;

       {*************************************
       *
       *    methods for Filters
       *
       **************************************}


end.    {HerLangu}
```

```
 {*****************************************
 *
 *      HerTerms.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 *****************************************}

unit HerTerms;


 { This unit defines the Namable Terminology Element options of the Hermes Language. }


interface

uses
       HerLists, HerBasic, HerHyper;


type

       {****************************************
       *
       *    data structure for Counter
       *
       ****************************************}

       PCountOne = ^TCountOne;
       TCountOne = object( TCounter )
          constructor Init(NewName : PChar; NewParam1 : PNodeContent;
                     NewParam2 : PNodeContent; NewParam3 : PNodeContent);
          function    EvalCharacter : PChar; virtual;
     function    HermesDisplay : PChar; virtual HermesDisplayDMT;
     function    Copy : Pointer;        virtual CopyDMT;
  end;


       {****************************************
       *
       *    data structure for Quantifier
       *
       ****************************************}


       {****************************************
       *
       *    data structure for Measure
       *
       ****************************************}



       {=====================================================================}
       {=============  IMPLEMENTATION  =====================================}
       {=====================================================================}

implementation

uses Strings;


       {****************************************
       *
       *    methods for Counts
       *
       ****************************************}

constructor TCountOne.Init(NewName : PChar; NewParam1 : PNodeContent;
                           NewParam2 : PNodeContent; NewParam3 : PNodeContent);
begin
```

```
end;


function    TCountOne.EvalCharacter : PChar;
begin
end;


function    TCountOne.HermesDisplay : PChar;
begin
end;


function    TCountOne.Copy : Pointer;
begin
end;


        {*************************************
        *
        *    methods for Quantifiers
        *
        ****************************************}


        {*************************************
        *
        *    methods for Measures
        *
        ****************************************}


end.    {HerTerms}
```

10. HerDataB.pas define database procedures

```
 {*****************************************
 *
 *      HerDataB.pas
 *
 *      version 2.0 -- Spring 1994
 *      (c) copyright 1994 by Gerry Stahl
 *      all rights reserved
 *****************************************}

unit HerDataB;

{ SPECIFICATION
  This unit defines an interface to the persistent store of Hermes
  as proposed in (Stahl, Ph.D. dissertation, 1993).

  This unit can be re-written to interface to other database
  systems like Mikroplis or an OODBMS.


  The following types of objects are maintained as persistent objects:
  Nodes, Links, Types, Query, etc. (and their descendents).


  DESIGN
  All database related functionality is encapsulated in this unit.
  This should make conversion to other DBMSs straight-forward.
  An alternative object-oriented design would be to make a unit for
  NamedObjects which defines operations on the names files as methods
  and a unit for PersistentObjects which defines operations on the
  object stream and its index as methods. (See HerBasic for these objects).

  The current implementation stores all persistent objects on an object stream.
  An index is maintained to the stream for random-access by ObjectId.
  Record N of the index contains the byte location of ObjectId N.
  Thus any access by ObjectId requires exactly 2 data accesses, no
  matter how large the stream.  The stream may be packed, but not the index.
  In addition, there is a file to store the names of Nodes, Links, Queries,
  etc.  This file is indexed on name by the efficient ToolBox
  routines.  This file contains the ObjectIds of named objects.
  When an object is referenced only by name, this file translates the
  name to its ObjectId for access from the stream. For files of up to a
  million names, this adds about two more disk accesses.


            HerStream                           HerNames
      .DBF            .IDX                  .NAM           .NDX
  Location: <<   Record = Id:  <<       Type-Name <<   Type-Name
    Object    <<   Location      <<       Record    <<      <<<
                <<   <<<           <<       <<<       <<<<<<
                  <<<                <<<<<<<


  The name file includes some types not saved on the Stream;
  these are synonyms of "main types", which are saved on the stream.
  Synonyms have the same Id as their main type.

  Case is ignored in spelling of names on name files.

  The Stream and the Name File and their Index files must all be
  opened and closed by the main Application.Init, for the sake of speed.

  Any objects referenced during a session are also maintained on 1 or 2
  collections in memory, to avoid fetches from disk.  This is especially
  necessary for recursive references to predicate names in queries
  and for traversing the context tree when checking for context matches.
  If memory runs low, the Collections can be emtied and started over --
  for instance whenever a new query is processed.
}

interface

uses OWindows, Objects, ODialogs, WinDos, Strings,
```

```
     HerLists, HerBasic, HerLinks;

const
  NameKeyLength = 40;     {names restricted to 40 characters}
                          {must agree with Hermes.Typ}
type
  ObjectTypes = Char;                          {each type of object is identified by a Char}

  NameKey = String[NameKeyLength];             {type for names stored in Pascal Strings}

  StrName = array[0..NameKeyLength] of Char;   {type for names stored in PChar Strings}

procedure DBOpenDataBase;
  {open object stream, its index, name file, its index, memory collections}

procedure DBCloseDataBase;
  {close object stream, its index, name file, its index, memory collections}

procedure DBMerge(Dest, Source : PChar);
  {merge data in two sets of object stream, index, name file, index}

function  DBIdOf(Name: PChar; ObjectType: ObjectTypes): LongInt;
  {get Id from database with Name; if not found, create an Id.}

function DBNameOf(Id: LongInt): PChar;
  {get Name from database with Id; if not found, return NoName.}

function  DBGetNamedObject(Name: PChar; ObjectType: ObjectTypes; Context: LongInt):
PPersistentObject;
  {get a NamedObject from database with Name; return nil if not successful.
   Note -- NamedObject like Queries, Predicates, BooleanPropositions, etc. can be named and saved
   to disk (the object stream.  In such a case, A Node with the Name and ObjectType is first saved.
   Then the NamedObject itself is saved and linked to the named node.  This facilitates having
   different definitions of the same named Clause in different contexts.  It also facilitates
   linking Clauses -- e.g., with "is_a" inheritance links.  The functions DBGetNamedObject
   and DBPutNamedObject maintain the linking of named Node to unnamed Clause.}

function  DBPutNamedObject(TheObject: PPersistentObject; Context: LongInt): LongInt;
  {put Object^.GetName in name file; return Id of named Node.
   This also calls DBPut to Put the Content on the Stream. }

function  DBGetName(Name: PChar; ObjectType: ObjectTypes): PPersistentObject;
  {get object from database with Name; return nil if not successful.
   Note -- this function automatically calls DBGet to access the stream.}

function  DBPutName(TheObject : PPersistentObject): Boolean;
  {put ObjectId and Name in name file; return true if successful.
   This should be called whenever a persistent object is named or
   when its name is changed. }

function  DBDeleteName(Name: PChar; ObjectType: ObjectTypes): Boolean;
  {delete object from name file by Name; return true if successful.}

function  DBListNames(ObjectType: ObjectTypes): PHermesList;
  {return list of all names on this file}

function  DBGet(ObjectId: LongInt): PPersistentObject;
  {get object from database by ObjectId; return nil if not successful}

function  DBPut(TheObject: PPersistentObject): Boolean;
  {put object in database by ObjectId; return true if successful}
  {This should be called whenever a persistent object is Initialized}

function  DBDelete(ObjectId: LongInt): Boolean;
  {delete object from database by ObjectId; return true if successful}

function DBReplace(OldId: LongInt; NewObject: PPersistentObject): Boolean;

function  DBGetLink(FromNodeId: LongInt; ToNodeId: LongInt; LinkType: PChar) : LongInt;
  {if there is a Link between From and To of MainType, return its Id, else return 0
   if LinkType = 'hermes_content', 'hermes_context' or 'hermes_graphic' get
   corresponding type of link}

function  DBAssignId(TheObject : PPersistentObject): Boolean;
  {assign an id, but do not put on stream or name file}
```

```
function  DBGetNodeType(Id: LongInt; var TheType: Char;
                                     var TheLoc: LongInt): Boolean;
  {return NodeType from StreamIndexRecord; return false if past Eof}

function  DBGetObjectId(Name: PChar; TheType: Char) : LongInt;
  {return Id of object; return 0 if not found}

function  DBListObjects : PHermesList;
  {return list of all object names in object stream}

function  DBGetSynonymId(OldName: PChar; var MainName: PChar;
                         ObjectType: ObjectTypes): LongInt;
  {return Id and MainName for a synonym}

function  DBAddSynonym(NewName: PChar; MainName: PChar;
                       ObjectType: ObjectTypes): Boolean;
  {define a new type as a synonym for an existing MainName}

function  DBListSynonyms(MainName: PChar; ObjectType: ObjectTypes): PHermesList;
  {return a list of synonyms for a MainName}

function DBInputNameKey : PChar;
  {check input of a Name}

function DBGetCurrent : LongInt;
  {return current date and time for timestamping nodes}

function  DBTest: Boolean;
  {test suite for this unit; return true if tests passed}

procedure View(Message: PChar; Destination: PChar; LineFeed: Boolean);
  {output Message to screen or other Destination}
  {Destination = ' '}

{=====================================================================}

implementation

uses WinProcs, WinTypes,
     TAccess, TAHigh,
     HerNodes, HerPersp, HerWorld, HerSeeds;

type        {these must agree with Hermes.Typ for TAccess}

  NameFileRec = record        {record structure for name files}
    RecStatus : LongInt;      {active/deleted}
    Name      : NameKey;      {object name}
    ObjId     : LongInt;      {object id}
    ObjType   : Char;         {object type}
  end;

  StreamIndexRec = record     {record structure for index to stream}
    Loc       : LongInt;      {deleted = 0}
    Size      : Integer;      {number of bytes on stream for this object}
    ObjType   : Char;         {object type}
  end;


 {****************************************
 *
 *       data structure of NamesListObject
 *       NameFileRecs sorted by ObjType + Name
 ****************************************}
  PNamesListObject = ^TNamesListObject;
  TNamesListObject = object(THermesNamedObject)
    constructor Init(TheType : Char; TheName : PChar; TheId : LongInt);
                constructor Load(var S : TStream);
                procedure   Store(var S : TStream); virtual StoreDMT;
    function     HermesDisplay : PChar;          virtual HermesDisplayDMT;
                function    Copy : Pointer;           virtual CopyDMT;
    procedure    CopyFrom(Source : PNamesListObject);
                procedure   SetObjId(NewObjId : LongInt);  virtual;
                procedure   SetObjType(NewObjType : Char); virtual;
                function    GetObjId   : LongInt;  virtual;
                function    GetObjType : Char;    virtual;
  private
```

```
    ObjId   : LongInt;
    ObjType : Char;
  end;


 {****************************************
 *
 *       data structure of NamesList
 *       NameFileRecs sorted by ObjType + Name
 ****************************************}
  PNamesList = ^TNamesList;
  TNamesList = object(THermesSortedList)
    constructor Init;
    procedure   DisplayList(Title : PChar); virtual DisplayListDMT;
    function    Compare(Key1, Key2 : Pointer) : Integer; virtual;
  end;



 {****************************************
 *
 *       data structure of ObjectsList
 *       objects sorted by Id
 ****************************************}
  PObjectsList = ^TObjectsList;
  TObjectsList = object(THermesSortedList)
    constructor Init;
    procedure   DisplayList(Title : PChar); virtual DisplayListDMT;
    function    Compare(Key1, Key2 : Pointer) : Integer; virtual;
  end;


 {****************************************
 *
 *       data structure of ConvertList
 *       Id pairs sorted by OldId
 ****************************************}
  PConvertList = ^TConvertList;
  TConvertList = object(THermesSortedList)
    function    Compare(Key1, Key2 : Pointer) : Integer; virtual;
  end;


 {****************************************
 *
 *       data structure of ConvertPair
 *       Id pairs
 ****************************************}
  PConvertPair = ^TConvertPair;
  TConvertPair = object(THermesObject)
    constructor Init(AnOldId, ANewId : LongInt);
    destructor  Done; virtual;
    function    GetOldId : LongInt; virtual;
    function    GetNewId : LongInt; virtual;
  private
    OldId : LongInt;
    NewId : LongInt;
  end;

var
  HerStream       : PHermesStream;     {stream for all persistent objects}

  HerStreamBackup : PHermesStream;     {backup copy of stream}

  HerIndex        : File of StreamIndexRec;  {index file to HerStream}

  HerIndexBackup  : File of StreamIndexRec;  {index file to HerStreamBackup}

  HerNames        : DataSet;           {file and index of object names}

  HerNamesList    : PNamesList;        {in memory list of names}
                                       {indexed on ObjType+Name}
  HerObjectsList  : PObjectsList;      {in memory list of objects}
                                       {indexed on ObjId}
  ExitSave        : Pointer;           {pointer to old ExitProc}
                        {** exit procedures can be dangerous in DLLs **}
```

```
const
  StreamBufferSize = 1024;      {buffer size for buffered read/write}

 {****************************************
 *
 *       procedure DBCopyFile
 *
 ****************************************}

procedure DBCopyFile(FromFile, ToFile : PChar);
  {make a copy of a file}
var
  FromF, ToF : file;
  NumRead, NumWritten : Word;
  Buf : array[1..4096] of Char;
begin
  Assign(FromF, FromFile);
  Reset(FromF, 1);
  Assign(ToF, ToFile);
  Rewrite(ToF, 1);
  repeat
    BlockRead(FromF, Buf, SizeOf(Buf), NumRead);
    BlockWrite(ToF, Buf, NumRead, NumWritten);
  until (NumRead = 0) or (NumWritten <> NumRead);
  Close(FromF);
  Close(ToF);
end;


 {****************************************
 *
 *       function DBOpen
 *
 ****************************************}

function DBOpen : Boolean;
  {open object stream and its index for read/write}
  {stream has name DBName; index ends in .IDX}
  {make backups of the files}
var
  Obj      : PHermesNode;
  DirInfo  : TSearchRec;    {declared in Dos unit}
  DBFName  : array[0..fsPathName] of Char;
  IDXName  : array[0..fsPathName] of Char;
  DBKName  : array[0..fsPathName] of Char;
  IBKName  : array[0..fsPathName] of Char;
  Dir       : array[0..fsDirectory] of Char;
  Name     : array[0..fsFileName] of Char;
  Ext      : array[0..fsExtension] of Char;
begin
  FileSplit(HermesApp.GetDBName, Dir, Name, Ext);
  if Name[0] = #0 then StrCopy(Name, 'Hermes');
  if Ext[0] = #0 then StrCopy(Ext, '.DBF');
  StrECopy(StrECopy(StrECopy(DBFName, Dir), Name), Ext);
  StrECopy(StrECopy(StrECopy(IDXName, Dir), Name), '.IDX');
  StrECopy(StrECopy(StrECopy(DBKName, Dir), Name), '.DBK');
  StrECopy(StrECopy(StrECopy(IBKName, Dir), Name), '.IBK');

  FindFirst(DBFName, faArchive, DirInfo);           {try to find object stream}
  if (DosError <> 0)                        {if it does not exist}
    then                                    {then create stream}
      begin
        HerStream := New(PHermesStream, Init(DBFName, stCreate, StreamBufferSize));
        Dispose(HerStream, Done);
      end;

  Assign(HerIndex, IDXName);
  FindFirst(IDXName, faArchive, DirInfo);    {try to find index to stream}
  if (DosError <> 0)                        {if it does not exist}
    then                                    {then create index}
      begin
        Rewrite(HerIndex);
        Close(HerIndex);
      end;

  HerStream := New(PHermesStream,Init(DBFName, stOpen, StreamBufferSize));
```

```
  Reset(HerIndex);                          {open stream and index}
  DBOpen := true;

  if (FileSize(HerIndex) = 0) then          {if creating new database, create header record 0 and
seed data}
    begin
      ErrorMsg(et_NOTICE, '      Creating Seed Data     ');
      Obj := New(PHermesNode, Init('HEADER'));      {create object with id = 0}
      DBPut(Obj);
      CreateSeed;                       {Create some standard Hermes objects -- in HerSeeds}
      ErrorMsg(et_NOTICE, '      Created Seed Data     ');
      HermesApp.SetContext(HermesUniversalContextId);
    end;
end;  {DBOpen}


 {****************************************
 *
 *       function DBOpenDataSet
 *
 ****************************************}
function DBOpenDataSet: Boolean;
  {open dataset for read/write}
var
  DBFName  : array[0..fsPathName] of Char;
  IDXName  : array[0..fsPathName] of Char;
  MBKName  : array[0..fsPathName] of Char;
  NBKName  : array[0..fsPathName] of Char;
  Path     : array[0..fsDirectory] of Char;
  Dir       : array[0..fsDirectory] of Char;
  Name     : array[0..fsFileName] of Char;
  Ext      : array[0..fsExtension] of Char;
begin
  StrCopy(Path, HermesApp.GetDBName);
  FileSplit(Path, Dir, Name, Ext);
  StrECopy(StrECopy(StrECopy(DBFName, Dir), Name), '.NAM');
  StrECopy(StrECopy(StrECopy(IDXName, Dir), Name), '.NDX');
  StrECopy(StrECopy(StrECopy(MBKName, Dir), Name), '.MBK');
  StrECopy(StrECopy(StrECopy(NBKName, Dir), Name), '.NBK');

  TAOpen(HerNames, DBFName, SizeOf(NameFileRec), IDXName, SizeOf(NameKey) - 1);
  if not Ok
    then
      begin
        TACreate(HerNames, DBFName, SizeOf(NameFileRec), IDXName, SizeOf(NameKey) - 1);
        if not Ok
          then ErrorMsg(et_SERIOUS, 'Could not create data set.');
      end;

  DBOpenDataSet := Ok;
end;  {DBOpenDataSet}


 {****************************************
 *
 *       procedure DBOpenDatabase
 *
 ****************************************}
procedure DBOpenDatabase;
  {open database files for read/write}
begin
  ExitSave := ExitProc;                  {store previous ExitProc}
  ExitProc := @DBCloseDatabase;             {close all files on run-time error}
  HerNamesList := New(PNamesList, Init);
  HerObjectsList := New(PObjectsList, Init);
  DBOpenDataSet;
  DBOpen;
end;  {DBOpenDatabase}


 {****************************************
 *
 *       procedure DBPackStream
 *
 ****************************************}
function DBPackStream: Boolean;
```

```
var
  TheRecord  : StreamIndexRec;
  TheObject  : PPersistentObject;
  DBFName    : PChar;
  DBFBack    : PChar;
  TempFile   : File;
begin
  DBFName := HermesApp.GetDBName;
  DBFBack := 'Temp.Tmp';
  HerStreamBackup := New(PHermesStream, Init(DBFBack, stCreate, StreamBufferSize));
  HerStream := New(PHermesStream, Init(DBFName, stOpen, StreamBufferSize));
  HerStream^.Seek(0);
  HerStreamBackup^.CopyFrom(HerStream^, HerStream^.GetSize);
  HerStream^.Flush;
  HerStreamBackup^.Flush;
  Dispose(HerStreamBackup, Done);
  Dispose(HerStream, Done);                    {backup stream to Temp.Tmp}

  HerStreamBackup := New(PHermesStream, Init(DBFBack, stOpen, StreamBufferSize));
  HerStream := New(PHermesStream, Init(DBFName, stCreate, StreamBufferSize));
  HerStreamBackup^.Seek(0);
  Reset(HerIndex);                             {reopen streams and index}
     Read(HerIndex,TheRecord);                 {first copy record 0}
     HerStream^.CopyFrom(HerStreamBackup^, TheRecord.Size);
  While not Eof(HerIndex) do                   {then copy the rest}
    begin
      Read(HerIndex, TheRecord);
      if (TheRecord.Loc <> 0) then
        begin
          HerStreamBackup^.Seek(TheRecord.Loc);
          TheRecord.Loc := HerStream^.GetPos;    {revise location of object}
          HerStream^.CopyFrom(HerStreamBackup^, TheRecord.Size);
          Seek(HerIndex, FilePos(HerIndex)-1);   {go back and update record}
          Write(HerIndex, TheRecord);
        end;
    end;  {while}
  HerStream^.Flush;
  HerStreamBackup^.Flush;
  Dispose(HerStreamBackup, Done);
  Dispose(HerStream, Done);
  Close(HerIndex);
  Assign(TempFile, 'Temp.Tmp');
  Erase(TempFile);
  DBPackStream := true;
end;  {DBPackStream}


 {****************************************
 *
 *       function DBBackupFiles
 *
 ****************************************}
function DBBackupFiles : Boolean;
  {make backups of the files}
var
  Obj      : PPersistentObject;
  DirInfo  : TSearchRec;    {declared in Dos unit}
  DBFName  : array[0..fsPathName] of Char;
  IDXName  : array[0..fsPathName] of Char;
  NAMName  : array[0..fsPathName] of Char;
  NDXName  : array[0..fsPathName] of Char;
  DBKName  : array[0..fsPathName] of Char;
  IBKName  : array[0..fsPathName] of Char;
  MBKName  : array[0..fsPathName] of Char;
  NBKName  : array[0..fsPathName] of Char;
  Dir       : array[0..fsDirectory] of Char;
  Name     : array[0..fsFileName] of Char;
  Ext      : array[0..fsExtension] of Char;
begin
  FileSplit(HermesApp.GetDBName, Dir, Name, Ext);
  if Name[0] = #0 then StrCopy(Name, 'Hermes');
  if Ext[0] = #0 then StrCopy(Ext, '.DBF');
  StrECopy(StrECopy(StrECopy(DBFName, Dir), Name), Ext);
  StrECopy(StrECopy(StrECopy(IDXName, Dir), Name), '.IDX');
  StrECopy(StrECopy(StrECopy(NAMName, Dir), Name), '.NAM');
  StrECopy(StrECopy(StrECopy(NDXName, Dir), Name), '.NDX');
```

```
  StrECopy(StrECopy(StrECopy(DBKName, Dir), Name), '.DBK');
  StrECopy(StrECopy(StrECopy(IBKName, Dir), Name), '.IBK');
  StrECopy(StrECopy(StrECopy(MBKName, Dir), Name), '.MBK');
  StrECopy(StrECopy(StrECopy(NBKName, Dir), Name), '.NBK');

  DBCopyFile(DBFName, DBKName);         {backup Stream}
  DBCopyFile(IDXName, IBKName);         {backup Stream Index}
  DBCopyFile(NAMName, MBKName);         {backup Names}
  DBCopyFile(NDXName, NBKName);         {backup Names Index}
end;    {DBBackupFiles}


 {****************************************
 *
 *       procedure DBCloseDatabase
 *
 ****************************************}

procedure DBCloseDatabase;
  {close the stream and index}
begin
  ExitProc := ExitSave;              {restore ExitProc -- see DBOpenDataBase}
  Dispose(HerStream, Done);          {close disk Stream}
  Close(HerIndex);                   {close index to Stream}
  TAClose(HerNames);                 {close dataset of names file}
  DBBackupFiles;                     {make copies of 4 files}
  DBPackStream;                      {pack Stream}
  Dispose(HerNamesList, Done);       {close collection of names}
  Dispose(HerObjectsList, Done);     {close collection of objects}
end;  {DBCloseDatabase}


 {****************************************
 *
 *       function DBMerge
 *
 ****************************************}
procedure DBMerge(Dest, Source : PChar);
  {Merge information from Source data files into Dest files.
   a. Merge named objects
   b. Merge unnamed objects
   c. Convert all object Id references from Source to new Ids
   Dest and Source are full path names;
   Dest may be HermesApp.GetDBSource if merging extra data,
   or Dest may be a new filename and Source may be HermesApp.GetDBSource
   if Packing current Source files into new Dest files.}

var
  DirInfo      : TSearchRec;    {declared in Dos unit}
  DBFSource    : array[0..fsPathName] of Char;
  IDXSource    : array[0..fsPathName] of Char;
  NAMSource    : array[0..fsPathName] of Char;
  NDXSource    : array[0..fsPathName] of Char;
  DBFDest      : array[0..fsPathName] of Char;
  IDXDest      : array[0..fsPathName] of Char;
  NAMDest      : array[0..fsPathName] of Char;
  NDXDest      : array[0..fsPathName] of Char;
  Dir           : array[0..fsDirectory] of Char;
  Name         : array[0..fsFileName] of Char;
  Ext          : array[0..fsExtension] of Char;
  SourceStream : PHermesStream;
  SourceIndex  : File of StreamIndexRec;
  SourceNames  : DataSet;
  DestStream   : PHermesStream;
  DestIndex    : File of StreamIndexRec;
  DestNames    : DataSet;
  SourceRecord : StreamIndexRec;
  DestRecord   : StreamIndexRec;
  TheRecord    : NameFileRec;
  TheKey       : NameKey;
  Obj          : PPersistentObject;
  FirstNewId   : LongInt;
  OldId        : LongInt;
  NewId        : LongInt;
  Conversions  : PConvertList;
```

```
    function ConvertId(TheOldId : LongInt) : LongInt;
    var
      TheIndex : Integer;
      TestPair : PConvertPair;
    begin  {ConvertId}
      TestPair := New(PConvertPair, Init(TheOldId, 0));
      if Conversions^.Search(TestPair, TheIndex)
        then ConvertId := PConvertPair(Conversions^.At(TheIndex))^.GetNewId
        else ConvertId := 0;
    end;   {ConvertId}


    procedure ConvertNode(TheNode : PHermesNode);

       procedure ConvertLink(ALinkId : PLinkId); far;
       begin  {ConvertLink}
         ALinkId^.SetId(ConvertId(ALinkId^.GetId));
       end;   {ConvertLink}

       procedure ConvertLinkList(ALinkList : PLinkList); far;

           procedure ConvertLinkId(ALinkId : PLinkId); far;
           begin  {ConvertLinkId}
             ALinkId^.SetId(ConvertId(ALinkId^.GetId));
           end;   {ConvertLinkId}

       begin  {ConvertLinkList}
         ALinkList^.ForEach(@ConvertLinkId);
       end;   {ConvertLinkList}

    begin  {ConvertNode}
      TheNode^.GetInLinks^.ForEach(@ConvertLinkList);
      TheNode^.GetOutLinks^.ForEach(@ConvertLinkList);
      TheNode^.GetContentOutLinks^.ForEach(@ConvertLink);
    end;   {ConvertNode}


    procedure ConvertLink(var TheLink : PLink);

        procedure ConvertSubLink(ASubLink: PSubLink); far;

            procedure ConvertLink(ALinkId : PLinkId); far;
            begin  {ConvertLink}
              ALinkId^.SetId(ConvertId(ALinkId^.GetId));
            end;   {ConvertLink}

        var
          TheContexts : PContextsList;
        begin  {ConvertSubLink}
          TheContexts := ASubLink^.GetListofContexts;
          TheContexts^.SetOriginalContext(ConvertId(TheContexts^.GetOriginalContext));
          TheContexts^.GetAddedContexts^.ForEach(@ConvertLink);
          TheContexts^.GetDeletedContexts^.ForEach(@ConvertLink);
          TheContexts^.SetSwitchContext(ConvertId(TheContexts^.GetSwitchContext));
        end;   {ConvertSubLink}

    begin  {ConvertLink}
      TheLink^.SetFromPO(ConvertId(TheLink^.GetFromPO));
      TheLink^.SetToPO(ConvertId(TheLink^.GetToPO));
      TheLink^.GetSubLinks^.ForEach(@ConvertSubLink);
    end;   {ConvertLink}


begin
  DBCloseDatabase;               {must have all Hermes files closed}

{open files for Source}
  FileSplit(Source, Dir, Name, Ext);
  if Name[0] = #0 then StrCopy(Name, 'Source');
  if Ext[0] = #0 then StrCopy(Ext, '.DBF');
  StrECopy(StrECopy(StrECopy(DBFSource, Dir), Name), Ext);
  StrECopy(StrECopy(StrECopy(IDXSource, Dir), Name), '.IDX');
  StrECopy(StrECopy(StrECopy(NAMSource, Dir), Name), '.NAM');
  StrECopy(StrECopy(StrECopy(NDXSource, Dir), Name), '.NDX');
```

```
  FindFirst(DBFSource, faArchive, DirInfo);         {try to find object stream}
  if (DosError <> 0)                               {if it does not exist}
    then                                           {then create stream}
      begin
        Writeln('Creating new Source Object Stream . . . .');
        SourceStream := New(PHermesStream, Init(DBFSource, stCreate, StreamBufferSize));
        Dispose(SourceStream, Done);
      end;

  Assign(SourceIndex, IDXSource);
  FindFirst(IDXSource, faArchive, DirInfo);         {try to find index to stream}
  if (DosError <> 0)                             {if it does not exist}
    then                                         {then create index}
      begin
        Rewrite(SourceIndex);
        Close(SourceIndex);
      end;

  SourceStream := New(PHermesStream,Init(DBFSource, stOpen, StreamBufferSize));
  Reset(SourceIndex);                       {open stream and index}

  TAOpen(SourceNames, NAMSource, SizeOf(NameFileRec), NDXSource, SizeOf(NameKey) - 1);
  if not Ok
    then
      begin
        Writeln('Creating new dataset for Source. . . .');
        TACreate(SourceNames, NAMSource, SizeOf(NameFileRec), NDXSource, SizeOf(NameKey) - 1);
        if not Ok
          then Writeln('Could not create data set.');
      end;

{open files for Dest}
  FileSplit(Dest, Dir, Name, Ext);
  if Name[0] = #0 then StrCopy(Name, 'Dest');
  if Ext[0] = #0 then StrCopy(Ext, '.DBF');
  StrECopy(StrECopy(StrECopy(DBFDest, Dir), Name), Ext);
  StrECopy(StrECopy(StrECopy(IDXDest, Dir), Name), '.IDX');
  StrECopy(StrECopy(StrECopy(NAMDest, Dir), Name), '.NAM');
  StrECopy(StrECopy(StrECopy(NDXDest, Dir), Name), '.NDX');

  FindFirst(DBFDest, faArchive, DirInfo);           {try to find object stream}
  if (DosError <> 0)                               {if it does not exist}
    then                                           {then create stream}
      begin
        Writeln('Creating new Dest Object Stream . . . .');
        DestStream := New(PHermesStream, Init(DBFDest, stCreate, StreamBufferSize));
        Dispose(DestStream, Done);
      end;

  Assign(DestIndex, IDXDest);
  FindFirst(IDXDest, faArchive, DirInfo);    {try to find index to stream}
  if (DosError <> 0)                             {if it does not exist}
    then                                         {then create index}
      begin
        Rewrite(DestIndex);
        Close(DestIndex);
      end;

  DestStream := New(PHermesStream,Init(DBFDest, stOpen, StreamBufferSize));
  Reset(DestIndex);                         {open stream and index}

  if (FileSize(DestIndex) = 0) then         {create header record 0}
    begin
      Obj := New(PPersistentObject, Init('HEADER'));
      Obj^.SetId(0);
      DestStream^.Seek(DestStream^.GetSize);
      DestRecord.Loc       := DestStream^.GetPos;
      DestStream^.Put(Obj);
      DestRecord.Size      := DestStream^.GetPos - DestRecord.Loc;
      DestRecord.ObjType   := obj^.GetType;
      Write(DestIndex, DestRecord);
    end;

  TAOpen(DestNames, NAMDest, SizeOf(NameFileRec), NDXDest, SizeOf(NameKey) - 1);
  if not Ok
    then
```

```
      begin
        Writeln('Creating new dataset for Dest. . . .');
        TACreate(DestNames, NAMDest, SizeOf(NameFileRec), NDXDest, SizeOf(NameKey) - 1);
        if not Ok
          then Writeln('Could not create data set.');
      end;

{a. Merge stream objects}
  FirstNewId := FileSize(DestIndex);       {save first new Id}
  Conversions := New(PConvertList, Init);  {create collection for conversions}
  Reset(SourceIndex);                      {start at beginning of SourceIndex}
  while not eof(SourceIndex) do
    begin
      Read(SourceIndex, SourceRecord);
      if ((SourceRecord.Loc > 0) and (SourceRecord.Loc < SourceStream^.GetSize))
        then
          begin
            SourceStream^.Seek(SourceRecord.Loc);
            Obj := PPersistentObject(SourceStream^.Get);
            OldId := Obj^.GetId;
            NewId := FileSize(DestIndex);
            Obj^.SetId(NewId);
            DestStream^.Seek(DestStream^.GetSize);
            DestRecord.Loc       := DestStream^.GetPos;
            DestStream^.Put(Obj);
            DestRecord.Size      := DestStream^.GetPos - DestRecord.Loc;
            DestRecord.ObjType   := obj^.GetType;
            Seek(DestIndex, NewId);
            Write(DestIndex, DestRecord);
            Conversions^.Insert(New(PConvertPair, Init(OldId, NewId)));
          end;
    end;  {while}

{b. Merge names}
  TAReset(SourceNames);
  repeat
    TANext(SourceNames, TheRecord, TheKey);
    if (Ok and (TheRecord.RecStatus = 0))
      then
        begin
          TheRecord.ObjId := ConvertId(TheRecord.ObjId);  {convert OldId to NewId}
          TAInsert(DestNames, TheRecord, TheKey);
          if not Ok                                {in case of name conflicts}
            then TAUpdate(DestNames, TheRecord, TheKey);  {use name from Source}
        end;
  until not Ok;

{c. Convert all object Id references from Source to new Ids}
  Seek(DestIndex, FirstNewId);
  while not eof(DestIndex) do
    begin
      Read(DestIndex, DestRecord);
      if ((DestRecord.Loc > 0) and (DestRecord.Loc < DestStream^.GetSize))
        then
          begin
            DestStream^.Seek(DestRecord.Loc);
            Obj := PPersistentObject(DestStream^.Get);
            if (Obj^.GetType = 'Z')
              then
                begin
                  ConvertNode(PHermesNode(Obj));
                  DestStream^.Seek(DestRecord.Loc);
                  DestStream^.Put(Obj);
                end;
            if (Obj^.GetType = 'L')
              then
                begin
                  ConvertLink(PLink(Obj));
                  DestStream^.Seek(DestRecord.Loc);
                  DestStream^.Put(Obj);
                end;
          end;
    end;  {while}

{close files and clean up}
  Dispose(SourceStream, Done);
```

```
  Dispose(DestStream, Done);
  Close(SourceIndex);
  Close(DestIndex);
  TAClose(SourceNames);
  TAClose(Destnames);
  Dispose(Conversions, Done);
  DBOpenDatabase;               {open all Hermes files again}
end;  {DBMerge}


 {****************************************
 *
 *       function DBFlush
 *
 ****************************************}

function DBFlush: Boolean;
  {flush buffer of object stream}
begin
  HerStream^.Flush;
  DBFlush := True;
end;  {DBFlush}


 {****************************************
 *
 *       function DBNamesFlush
 *
 ****************************************}

function DBNamesFlush: Boolean;
  {flush buffer of names dataset}
begin
{ TAFlush(HerNames); }     {problem in TAccess.CloseFileHandle call to MsDos}
  DBNamesFlush := True;
end;  {DBNamesFlush}


 {****************************************
 *
 *       function DBIdOf
 *
 ****************************************}

function DBIdOf(Name: PChar; ObjectType: ObjectTypes): LongInt;
var
  TheRecord : NameFileRec;
  TheName   : String;
  KeyName   : NameKey;
  TheIndex  : Integer;
  TheId     : LongInt;
  TheItem   : PNamesListObject;
  TheRec    : StreamindexRec;
  TheObject : PPersistentObject;
begin
  Name :=  StrLower(Name);
                                    {first try to get it from HerNamesList}
  TheItem := New(PNamesListObject, Init(ObjectType, Name, 0));
  if HerNamesList^.Search(TheItem, TheIndex)
    then
      begin
        Dispose(TheItem, Done);
        TheItem := HerNamesList^.At(TheIndex);
        DBIdOf := TheItem^.GetObjId;
      end
    else
      begin                         {else try to get it from Name FIle}
        Dispose(TheItem, Done);
        TheName := StrPas(Name);
        KeyName := ObjectType + TheName;
        TARead(HerNames, TheRecord, KeyName, ExactMatch);   {search index for Name}
        if Ok
          then
            begin
              DBIdOf := TheRecord.ObjId;
            end
```

```
          else
            begin                       {else assign a new Id}
              TheId := FileSize(HerIndex);
              TheRec.Loc := 0;
              TheRec.Size := 0;
              Seek(HerIndex, TheId);
              Write(HerIndex, TheRec);              {store on stream Index}

              TheRecord.RecStatus := 0;
              TheRecord.Name := TheName;
              TheRecord.ObjId := TheId;
              TheRecord.ObjType := ObjectType;
              TAWrite(HerNames, TheRecord, KeyName);
                                                     {store on HerNamesList}
              TheItem := New(PNamesListObject, Init(ObjectType, Name, TheId));
              HerNamesList^.Insert(TheItem);
                                                    {put dummy object on HerObjectsList}
              TheObject := New(PPersistentObject, Init(NoName));
              TheObject^.SetId(TheId);
              TheObject^.SetType('?');              {flag as dummy}
              HerObjectsList^.Insert(TheObject);

              DBIdOf := TheId;
            end;
      end;
end;  {DBIdOf}



 {****************************************
 *
 *       function DBNameOf
 *
 ****************************************}

function DBNameOf(Id: LongInt): PChar;
var
  TheIndex  : Integer;
  TheObject : PPersistentObject;
begin
  TheObject := New(PPersistentObject, Init(NoName));
  TheObject^.SetId(Id);
  if HerObjectsList^.Search(TheObject, TheIndex)
    then DBNameOf := PPersistentObject(HerObjectsList^.At(TheIndex))^.GetName
    else
      begin
        TheObject := DBGet(Id);
        if (TheObject <> nil)
          then DBNameOf := StrNew(TheObject^.GetName)
          else DBNameOf := NoName;
      end;
end;  {DBNameOf}


 {****************************************
 *
 *       function DBGetNamedObject
 *
 ****************************************}

function  DBGetNamedObject(Name: PChar; ObjectType: ObjectTypes; Context: LongInt):
PPersistentObject;
  {get a NamedObject from database with Name; return nil if not successful.
   Note -- NamedObjects like DataLists, Characters, Booleans, etc. can be named and saved
   to disk (the object stream).  In such a case, A THermesNode with the Name and ObjectType is first
saved.
   Then the TNodeContent itself is saved and linked to the named node.  This facilitates having
   different definitions of the same named Content in different contexts.  It also facilitates
   linking Contents -- e.g., with "is_a" inheritance links.  The functions DBGetNamedObject
   and DBPutNamedObject maintain the linking of named Node to unnamed content.}
var
  TheNode     : PHermesNode;
  ThePO       : PPersistentObject;
  TheLinkId   : PLinkId;
  TheLink     : PLink;
  TheLinkList : PLinkList;
```

```
      function MatchesContext(ALinkId : PLinkId): Boolean; far;
      begin   {MatchesContext}
        TheLink := PLink(DBGet(ALinkId^.GetId));
        if assigned(TheLink)
          then MatchesContext := TraversesLink(Context, TheLink)
          else MatchesContext := false;
      end;    {MatchesContext}
begin
  DBGetNamedObject := nil;
  TheLinkId := nil;
  TheNode := PHermesNode(DBGetName(Name, ObjectType));
  if assigned(TheNode)
    then
      begin
        TheLinkList := TheNode^.GetContentOutLinks;
        if assigned(TheLinkList)
          then TheLinkId := TheLinkList^.FirstThat(@MatchesContext)
          else ErrorMsg(et_WARNING, 'DBGetNamedObject: TheLinkList not assigned');
        if assigned(TheLinkId)
          then
            begin
              TheLink := PLink(DBGet(TheLinkId^.GetId));
              DBGetNamedObject := DBGet(TheLink^.GetToPO);
            end
          else ErrorMsg(et_WARNING, 'DBGetNamedObject: TheLinkId not assigned');
      end;
end;


 {****************************************
 *
 *       function DBPutNamedObject
 *
 ****************************************}

function  DBPutNamedObject(TheObject: PPersistentObject; Context: LongInt): LongInt;
  {put TheObject^.GetName in name file; return true if successful.
   This creates a node with the name and links it to TheObject as its content in the context}
var
  TheNode     : PHermesNode;
  TheName     : PChar;
  TheType     : Char;
begin
  TheName := TheObject^.GetName;
  TheType := TheObject^.GetType;
  TheNode := nil;
  if ((StrComp(TheName, NoName) <> 0)  and  (StrComp(TheName, '') <> 0)
      and (StrComp(TheName, ' ') <> 0)  and  (StrComp(TheName, '*') <> 0))
    then TheNode := PHermesNode(DBGetName(TheName, TheType))
    else TheName := NoName;
  if ((TheNode <> nil) and (StrComp(TheName, NoName) <> 0))
    then                            {if there is a name and the named node already exists}
      begin
        DBPut(TheObject);
        ModifyNode(TheNode^.GetId, 0, TheObject^.GetId, Context, 0, nil, nil)
      end
    else                            {if there is no name or the named node does not already exist}
      begin
        TheNode := New(PHermesNode, Init(TheName));
        TheNode^.SetType(TheType);
        if (StrComp(TheName, NoName) = 0)
          then DBPut(TheNode)
          else DBPutName(TheNode);
        DBPut(TheObject);
        CreateHypertextSwitch(0, TheNode^.GetId, TheObject^.GetId, HermesContentLinkType, Context, 0,
nil);
      end;
  DBPutNamedObject := TheNode^.GetId;
end;


 {****************************************
 *
 *       function DBGetName
 *
 ****************************************}
```

```
function  DBGetName(Name: PChar; ObjectType: ObjectTypes): PPersistentObject;
  {get object with Name; return Nil if not successful}
var
  TheRecord : NameFileRec;
  TheName   : NameKey;
  TheIndex  : Integer;
  TheId     : LongInt;
  TheItem   : PNamesListObject;
begin
  Ok := False;
  DBGetName := Nil;                             {in case nothing found}
  Name :=  StrLower(Name);
                                      {first try to get it from HerNamesList}
  TheItem := New(PNamesListObject, Init(ObjectType, StrNew(Name), 0));
  if HerNamesList^.Search(TheItem, TheIndex)
    then
      begin
        Dispose(TheItem, Done);
        TheItem := HerNamesList^.At(TheIndex);
        TheId := TheItem^.GetObjId;
        Ok := True;
      end
    else
      begin
        Dispose(TheItem, Done);
        TheName := StrPas(Name);
        TheName := ObjectType + TheName;
        TARead(HerNames, TheRecord, TheName, ExactMatch);   {search index for Name}
        if Ok
          then
            begin
              TheId := TheRecord.ObjId;
              TheItem := New(PNamesListObject, Init(ObjectType, Name, TheId));
              HerNamesList^.Insert(TheItem);        {save on HerNamesList}
            end;
      end;
  if Ok
    then DBGetName := DBGet(TheId);          {get object from stream by id}
end;  {DBGetName}



 {****************************************
 *
 *       function DBPutName
 *
 ****************************************}
function DBPutName(TheObject : PPersistentObject): Boolean;
  {put Name and ObjectId in TheSet; return true if successful}
var
  TheItem   : PNamesListObject;
  TheIndex  : Integer;
  TheRecord : NameFileRec;
  TheName   : PChar;
  TempName  : NameKey;
begin
  TheName := StrLower(TheObject^.GetName);
  if ((StrComp(TheName, NoName) = 0)  or  (StrComp(TheName, '') = 0)
      or (StrComp(TheName, ' ') = 0)  or  (StrComp(TheName, '*') = 0))
    then TheName := NoName;
  if (StrComp(TheName, NoName) <> 0)
    then                              {if there is a name}
      begin
        TheItem := New(PNamesListObject, Init(TheObject^.GetType,
                                              TheObject^.GetName,
                                              TheObject^.GetId));
        if HerNamesList^.Search(TheItem, TheIndex)
          then
            begin
              Dispose(TheItem, Done);
              TheItem := HerNamesList^.At(TheIndex);
              TheObject^.SetId(TheItem^.GetObjId);    {save the Id from HerNamesList}
            end
          else
            begin
              TempName := StrPas(TheName);
```

```
              TempName := TheObject^.GetType + TempName;
              TARead(HerNames, TheRecord, TempName, True);    {read if exact match}
              if Ok
                then
                  begin
                    TheObject^.SetId(TheRecord.ObjId);         {save the Id from Name File}
                  end
                else
                  begin
                    TheObject^.SetId(FileSize(HerIndex));    {calculate new Id}
                    TheRecord.RecStatus := 0;
                    TheRecord.Name := StrPas(TheName);
                    TheRecord.ObjId := TheObject^.GetId;
                    TheRecord.ObjType := TheObject^.GetType;
                    TAWrite(HerNames, TheRecord, TempName);  {save on Name File}
                  end;
              TheItem^.SetObjId(TheObject^.GetId);
              HerNamesList^.Insert(TheItem);
            end;
      end;
  DBPut(TheObject);
  DBNamesFlush;
  DBPutName := True;
end;  {DBPutName}



 {****************************************
 *
 *       function DBDeleteName
 *
 ****************************************}
function DBDeleteName(Name: PChar; ObjectType: ObjectTypes): Boolean;
  {delete name and object by Name; return true if successful}
var
  TheItem   : PNamesListObject;
  TheIndex  : Integer;
  TheKey    : String[NameKeyLength + 1];
  TheRecord : NameFileRec;
begin
  Name := StrLower(Name);
  TheItem := New(PNamesListObject, Init(ObjectType, Name, 0));
  if HerNamesList^.Search(TheItem, TheIndex)
    then HerNamesList^.AtDelete(TheIndex);        {then delete from list}
  Dispose(TheItem, Done);

  DBDeleteName := False;                            {in case not successful}
  TheKey := ObjectType + StrPas(Name);
  TARead(HerNames, TheRecord, TheKey, ExactMatch);    {find record on index file}
  if Ok then
    begin
      TADelete(HerNames, TheKey);                   {delete name from index}
      DBNamesFlush;
      if (ObjectType <> 'T')            {don't delete Types because of synonyms}
        then DBDelete(TheRecord.ObjId);  {delete ObjectId from stream}
      DBDeleteName := true;
    end;
end;  {DBDeleteName}



 {****************************************
 *
 *       function DBGet
 *
 ****************************************}
function  DBGet(ObjectId: LongInt): PPersistentObject;
  {get object by ObjectId; return Nil if not successful or Id=0}
var
  TheRecord : StreamIndexRec;
  TheItem   : PPersistentObject;
  TheIndex  : Integer;
begin
  if ObjectId <= 0
    then
      begin
```

```
        TheItem := nil;
        ErrorMsg(et_NOTICE, 'DBGet: ObjectId <= 0');
        Exit;
      end;

  TheItem := New(PPersistentObject, Init(NoName));
  TheItem^.SetId(ObjectId);
  if HerObjectsList^.Search(TheItem, TheIndex)
    then                                          {if on list}
      begin                                       {then get from list}
        Dispose(TheItem, Done);
        TheItem := HerObjectsList^.At(TheIndex);
        if (TheItem^.GetType = '?')
          then TheItem := nil;                {if dummy object from DBIdOf}
      end
    else                                      {else get from stream}
      begin
        Dispose(TheItem, Done);
        TheItem := nil;
        if ((ObjectId > 0) and (ObjectId < FileSize(HerIndex)))
          then
            begin
              Seek(HerIndex, ObjectId);       {find index record N=ObjectId}
              Read(HerIndex, TheRecord);      {read the index record}
              if (TheRecord.Loc > 0)
                then
                  begin
                    HerStream^.Seek(TheRecord.Loc); {find location on stream}
                    TheItem := PPersistentObject(HerStream^.Get);      {read the object from stream}
                    if (TheItem <> nil)
                      then HerObjectsList^.Insert(TheItem);      {save on HerObjectsList}
                  end;
            end
          else
            begin
             ErrorMsg(et_NOTICE, 'DBGet: error in ObjectId');
             TheItem := nil;
           end;
      end;
  DBGet := TheItem;
end;  {DBGet}



 {****************************************
 *
 *       function DBPut
 *
 ****************************************}
function DBPut(TheObject: PPersistentObject): Boolean;
        {put object in database by ObjectId; return true if successful}
var
        TheRec                : StreamIndexRec;
  NameString   : PChar;
        Length                : Integer;
  TheIndex    : Integer;
  TheId       : LongInt;
begin   {DBPut}
  if (TheObject^.GetId = 0)
    then TheObject^.SetId(FileSize(HerIndex));

        HerStream^.Seek(HerStream^.GetSize);         {go to end of stream}
        TheRec.Loc := HerStream^.GetPos;             {get location for object on stream}
  DBFlush;
        HerStream^.Put(TheObject);                   {store object on HerStream}
  Length := HerStream^.GetPos - TheRec.Loc;    {calculate size on stream}
  DBFlush;
  TheRec.Size := Length;                       {store size of object on stream}
  TheRec.ObjType := TheObject^.GetType;        {store Type of object on stream}
  Seek(HerIndex, TheObject^.GetId);            {seek record=ObjectId on index}
        Write(HerIndex, TheRec);                     {store location of stream in HerIndex}
  if HerObjectsList^.Search(TheObject, TheIndex)
    then HerObjectsList^.AtDelete(TheIndex);
  HerObjectsList^.Insert(TheObject);           {put on object list}

  DBFlush;
```

```
  DBPut := true;
end;   {DBPut}


 {****************************************
 *
 *       function DBDelete
 *
 ****************************************}
function DBDelete(ObjectId: LongInt): Boolean;
  {delete object in database by ObjectId; return true if successful}
var
  TheRec : StreamIndexRec;
  TheItem   : PPersistentObject;
  TheIndex  : Integer;
begin
  TheItem := New(PPersistentObject, Init(NoName));
  TheItem^.SetId(ObjectId);
  if HerObjectsList^.Search(TheItem, TheIndex)
    then HerObjectsList^.AtDelete(TheIndex);      {delete from list}
  Dispose(TheItem, Done);

  Seek(HerIndex, ObjectId);
  Read(HerIndex, TheRec);
  TheRec.Loc := 0;                     {indicate object deleted}
  Seek(HerIndex, ObjectId);            {go back and update}
  Write(HerIndex, TheRec);             {store 0 in HerIndex}
  DBDelete := true;
end;  {DBDelete}


 {****************************************
 *
 *       function DBReplace
 *
 ****************************************}
function DBReplace(OldId: LongInt; NewObject: PPersistentObject): Boolean;
  {Replace object in database at ObjectId; return true if successful}
begin
  DBDelete(OldId);
  DBPut(NewObject);
end;  {DBReplace}


 {****************************************
 *
 *       function DBGetLink
 *
 ****************************************}
function  DBGetLink(FromNodeId: LongInt; ToNodeId: LongInt; LinkType: PChar) : LongInt;
  {if there is a Link between From and To of MainType, return its Id, else return 0
   if LinkType = 'hermes_content', 'hermes_context' or 'hermes_graphic' get
   corresponding type of link}

      function GoesToNode(ALinkId : PLinkId): Boolean; far;
      begin   {GoesToNode}
        GoesToNode := (DBGet(PLink(DBGet(ALinkId^.GetId))^.GetToPO)^.GetId = ToNodeId);
      end;    {GoesToNode}

var
  TheFromHermesNode : PHermesNode;
  TheFromNodeObject : PNodeObject;
  TheLinkList       : PLinkList;
  TheLinkId         : PLinkId;
begin
  DBGetLink := 0;
  if (LinkType = HermesContentLinkType) then    {TContentLink}
      begin
        TheFromHermesNode := PHermesNode(DBGet(FromNodeId));
        TheLinkList := TheFromHermesNode^.GetContentOutLinks;
      end
  else if (LinkType = HermesGraphicLinkType) then    {TGraphicLink}
      begin
        TheFromHermesNode := PHermesNode(DBGet(FromNodeId));
        TheLinkList := TheFromHermesNode^.GetContentOutLinks;
      end
```

```
  else if (LinkType = HermesContextLinkType) then    {TContextLink}
      begin
        TheFromNodeObject := PNodeObject(DBGet(FromNodeId));
        TheLinkList := TheFromNodeObject^.GetOutLinksOfType(LinkType);
      end
  else                     {external TLink of type LinkType}
      begin
        TheFromNodeObject := PNodeObject(DBGet(FromNodeId));
        TheLinkList := TheFromNodeObject^.GetOutLinksOfType(LinkType);
      end;
  TheLinkId := TheLinkList^.FirstThat(@GoesToNode);
  if (TheLinkId <> nil)
    then DBGetLink := TheLinkId^.GetId;
end;


 {****************************************
 *
 *       function DBAssignId
 *
 ****************************************}
function  DBAssignId(TheObject : PPersistentObject): Boolean;
  {assign an id, but do not put on stream or name file}
var
  TheRec : StreamindexRec;
begin
  TheObject^.SetId(FileSize(HerIndex));
  TheRec.Loc := 0;
  TheRec.Size := 0;

  Seek(HerIndex, TheObject^.GetId);
  Write(HerIndex, TheRec);
  DBAssignId := True;
end;  {DBAssignId}



 {****************************************
 *
 *       function DBGetNodeType
 *
 ****************************************}
function  DBGetNodeType(Id: LongInt; var TheType: Char;
                                     var TheLoc: LongInt): Boolean;
  {return Node Type from StreamIndexRecord; return false if past Eof}
var
  TheRecord : StreamIndexRec;
begin
  if (Id > (FileSize(HerIndex) - 1))
    then
      begin
        TheType := ot_Context;
        TheLoc := 0;
        DBGetNodeType := False;
      end
    else
      begin
        Seek(HerIndex, Id);
        Read(HerIndex, TheRecord);
        TheLoc  := TheRecord.Loc;
        TheType := TheRecord.ObjType;
        DBGetNodeType := True;
      end;
end;  {DBGetNodeType}



 {****************************************
 *
 *       function DBGetObjectId
 *
 ****************************************}
function  DBGetObjectId(Name: PChar; TheType: Char) : LongInt;
  {return Id of object; return 0 if not found}
var
  TheSet : DataSet;
```

```
  TheRecord : NameFileRec;
  TheString : String;
begin
  DBGetObjectId := 0;                            {in case nothing found}
  Name := StrLower(Name);
  TheString := TheType + StrPas(Name);
        TARead(HerNames, TheRecord, TheString, ExactMatch);   {search index for Name}
        if Ok then
          begin
            DBGetObjectId := TheRecord.ObjId;      {return id}
          end
end;  {DBGetObjectId}


 {****************************************
 *
 *       function DBListNames
 *
 ****************************************}
function DBListNames(ObjectType: ObjectTypes): PHermesList;
  {return list of all names in this file}
var
  TheRecord : NameFileRec;
  TheList   : PHermesList;
  TheKey    : String;
  ThePChar  : StrName;
begin
        TheList := New(PHermesList, Init);          {initialize empty list}
  TheKey := ObjectType;
  TAReset(HerNames);                {set to start of file}
  TARead(HerNames, TheRecord, TheKey, PartialMatch);
  if Ok                             {if a record found for this ObjType}
    then
      begin
        while (Ok and (TheKey[1] = ObjectType)) do
          begin
                       if (TheRecord.ObjId <> 0)
              then
                begin
                  StrPCopy(ThePChar, TheRecord.Name);
                  TheList^.Insert(StrNew(ThePChar));  {add Name to list}
                end;
                      TANext(HerNames, TheRecord, TheKey);
          end;  {end while}
      end;
        DBListNames := TheList;
end;  {DBList}



 {****************************************
 *
 *       function DBListObjects
 *
 ****************************************}
function  DBListObjects:PHermesList;
  {return list of all object names in object stream}
var
  TheList   : PHermesList;
  TheRecord : StreamIndexRec;
  TheObject : PPersistentObject;
  TheName   : PChar;
  TheId     : LongInt;
  SavePlace : LongInt;
begin
  TheList := New(PHermesList, Init);       {initialize empty list}
  Writeln;
  Writeln('List of Objects on Stream');
  Reset(HerIndex);                        {start at beginning of index}
  while not Eof(HerIndex) do              {go through index file}
    begin
      Read(HerIndex, TheRecord);
      if ((TheRecord.Loc <> 0) and (TheRecord.Loc < HerStream^.GetSize))
      then                                 {skip records with Loc = 0}
        begin
          SavePlace := FilePos(HerIndex);  {save Index file position}
```

```
          HerStream^.Seek(TheRecord.Loc);
          Write(TheRecord.Loc:6, '  ');
          TheObject := PPersistentObject(HerStream^.Get);
          TheName := TheObject^.GetName;
          Write(TheName:20, '   ');      {print object's name}
        {  TheObject^.HermesDisplay;    }
          Writeln;
          TheList^.Insert(TheObject);      {add object to list}
          Seek(HerIndex, SavePlace);       {restore Index file position}
        end;
    end;
end;  {DBListObjects}



 {****************************************
 *
 *       function DBGetSynonymId
 *
 ****************************************}
function DBGetSynonymId(OldName: PChar; var MainName: PChar;
                        ObjectType: ObjectTypes): LongInt;
  {return Id and MainName for a synonym Type}
var
  MainType : PPersistentObject;
begin
  MainName := NoName;
  DBGetSynonymId := -1;                     {return -1 if MainType not found}
  OldName := StrLower(OldName);
  MainType := DBGetName(OldName, ObjectType);     {get Main object by synonym id}
  if (assigned(MainType) )
    then
      begin
        MainName := MainType^.GetName;            {return MainName}
        if (StrLen(MainName) > 0)
          then DBGetSynonymId := MainType^.GetId;        {return Id}
      end;
end;  {DBGetSynonymId}

 {****************************************
 *
 *       function DBAddSynonym
 *
 ****************************************}
function DBAddSynonym(NewName: PChar; MainName: PChar;
                      ObjectType: ObjectTypes): Boolean;
  {define a new type as a synonym for an existing MainName}
var
  TheString : String;
  TheRecord : NameFileRec;
  TheItem   : PNamesListObject;
  THeIndex  : Integer;
begin
  DBAddSynonym := False;
  TheString := ObjectType + StrPas(StrLower(MainName));
  TARead(HerNames, TheRecord, TheString, ExactMatch);   {search index for MainName}
  if Ok
    then
      begin
        TheItem := New(PNamesListObject, Init(ObjectType, NewName, TheRecord.ObjId));
        if HerNamesList^.Search(TheItem, TheIndex)
          then
            begin
              Dispose(TheItem);
              ErrorMsg(et_NOTICE, 'This synonym is already defined');
            end
          else
            begin
              TheString := StrPas(NewName);
              TheRecord.Name := TheString;
              TheString := ObjectType + TheString;
              TAWrite(HerNames, TheRecord, TheString);   {save on Names file}
              HerNamesList^.Insert(TheItem);             {save on HerNamesList}
              DBAddSynonym := True;
            end;
      end
```

```
    else ErrorMsg(et_NOTICE, 'DBAddSynonym: The main name was not found.');
end;  {DBAddSynonym}



 {****************************************
 *
 *       function DBListSynonyms
 *
 ****************************************}
function DBListSynonyms(MainName: PChar; ObjectType: ObjectTypes): PHermesList;
  {return a list of synonyms for a MainName}
var
  TheName   : PChar;
  MainId    : LongInt;
  TheString : String;
  TheRecord : NameFileRec;
  TheList   : PHermesList;
begin
  TheName   := StrLower(MainName);
  TheString := StrPas(TheName);
  MainId    := DBGetSynonymId(TheName, MainName, ObjectType);
  TheList   := New(PHermesList, Init);    {initialize empty list}
  TAReset(HerNames);                      {set to start of file}
  repeat                                  {loop thru all records}
    TANext(HerNames, TheRecord, TheString);
    if (Ok and (TheRecord.ObjId = MainId)) then
      begin
        StrPCopy(TheName, TheString);
        TheList^.Insert(StrNew(TheName + 1));  {add Name to list if right Id}
      end;                                     {first strip off first character for ObjectType}
  until not Ok;                                {Ok = False at EOF}
  DBListSynonyms := TheList;
end;  {DBListSynonyms}


       {****************************************
       *
       *       methods for TNamesListObject
       *
       ****************************************}

constructor TNamesListObject.Init(TheType: Char; TheName: PChar; TheId: LongInt);
begin
       INHERITED Init(StrLower(TheName));
  ObjId := TheId;
  ObjType := TheType;
end;

constructor TNamesListObject.Load(var S : TStream);
begin
  INHERITED Load(S);
  S.Read(ObjId, SizeOf(LongInt));
  S.Read(ObjType, SizeOf(Char));
end;

procedure   TNamesListObject.Store(var S : TStream);
begin
  INHERITED Store(S);
       S.Write(ObjId, SizeOf(LongInt));
       S.Write(ObjType, SizeOf(Char));
end;

function  TNamesListObject.HermesDisplay : PChar;
begin
  HermesDisplay := GetName;
end;

function TNamesListObject.Copy : Pointer;
var TheCopy : PNamesListObject;
begin
       TheCopy := New(PNamesListObject, Init(ot_undefined, nil, 0));
  TheCopy^.CopyFrom(@Self);
  Copy := TheCopy;
end;
```

```
procedure TNamesListObject.CopyFrom(Source : PNamesListObject);
begin
  SetName(Source^.GetName);
  SetObjId(Source^.GetObjId);
  SetObjType(Source^.GetObjType);
end;

procedure TNamesListObject.SetObjId(NewObjId : LongInt);
begin
  ObjId := NewObjId;
end;

procedure TNamesListObject.SetObjType(NewObjType : Char);
begin
  ObjType := NewObjType;
end;

function TNamesListObject.GetObjId : LongInt;
begin
  GetObjId := ObjId;
end;

function TNamesListObject.GetObjType : Char;
begin
  GetObjType := ObjType;
end;



 {****************************************
 *
 *       methods for NamesList
 *
 ****************************************}
constructor TNamesList.Init;
begin
  INHERITED Init;
  Duplicates := False;          {replace old objects with new ones of same Key}
  SetLimit(100);
end;


procedure TNamesList.DisplayList(Title : PChar);
    procedure DisplayOne(Item : PNamesListObject); far;
    begin  {DisplayOne}
      Writeln(Item^.GetObjType, '   ', Item^.GetName, '   ', Item^.GetObjId);
    end;   {DisplayOne}
begin  {DisplayList}
       Writeln;
  Writeln(Title);
  Writeln;
  ForEach(@DisplayOne);
  Writeln;
end;   {DisplayList}


function TNamesList.Compare(Key1, Key2: Pointer) : Integer;
var
  Diff : Integer;
  TheKey1  : array[0..NameKeyLength + 1] of Char;
  TheKey2  : array[0..NameKeyLength + 1] of Char;
  KeyChar1 : PChar;
  KeyChar2 : PChar;
  TheNLO  : PNamesListObject;
  TheType : Char;
  TheString1 : String;
  TheString2 : String;
begin
  TheNLO := PNamesListObject(Key1);
  TheType := TheNLO^.GetObjType;
  TheString1 := TheType;
  TheString2 := StrPas(TheNLO^.GetName);
  TheString1 := TheString1 + TheString2;
  StrPCopy(TheKey1, TheString1);
  KeyChar1 := TheKey1;
```

```
  TheNLO := PNamesListObject(Key2);
  TheType := TheNLO^.GetObjType;
  TheString1 := TheType;
  TheString2 := StrPas(TheNLO^.GetName);
  TheString1 := TheString1 + TheString2;
  StrPCopy(TheKey2, TheString1);
  KeyChar2 := TheKey2;

  Diff := StrComp(KeyChar1, KeyChar2);
  if (Diff < 0)
    then Compare := -1
    else
      if (Diff = 0)
        then Compare :=  0
        else Compare :=  1;
end;


 {****************************************
 *
 *       methods for ObjectsList
 *
 ****************************************}
constructor TObjectsList.Init;
begin
  INHERITED Init;
  Duplicates := False;
  SetLimit(100);
end;


procedure TObjectsList.DisplayList(Title : PChar);
    procedure DisplayOne(Item : PHermesNamedObject); far;
    begin  {DisplayOne}
      Writeln(Item^.GetName);
    end;   {DisplayOne}
begin  {DisplayList}
       Writeln;
  Writeln(Title);
  Writeln;
  ForEach(@DisplayOne);
  Writeln;
end;   {DisplayList}


function TObjectsList.Compare(Key1, Key2: Pointer) : Integer;
var
  Diff : Integer;
begin
  Diff := PPersistentObject(Key1)^.GetId - PPersistentObject(Key2)^.GetId;
  if Diff < 0
    then Compare := -1
    else
      if Diff = 0
        then Compare :=  0
        else Compare :=  1;
end;


 {****************************************
 *
 *       methods for ConvertList
 *
 ****************************************}

function TConvertList.Compare(Key1, Key2 : Pointer) : Integer;
var
  Diff : LongInt;
begin
  Diff := PConvertPair(Key1)^.GetOldId - PConvertPair(Key2)^.GetOldId;
  if Diff < 0
    then Compare := -1
    else
      if Diff = 0
        then Compare :=  0
        else Compare :=  1;
```

```
end;


 {****************************************
 *
 *       methods for ConvertPair
 *       Id pairs
 ****************************************}
constructor TConvertPair.Init(AnOldId, ANewId : LongInt);
begin
  INHERITED Init;
  OldId := AnOldId;
  NewId := ANewId;
end;

destructor  TConvertPair.Done;
begin
  OldId := 0;
  NewId := 0;
  INHERITED Done;
end;

function    TConvertPair.GetOldId : LongInt;
begin
  GetOldId := OldId;
end;

function    TConvertPair.GetNewId : LongInt;
begin
  GetNewId := NewId;
end;


 {****************************************
 *
 *       function DBInputNameKey
 *
 ****************************************}
function DBInputNameKey : PChar;
var
  TheName  : String;
  ThePCHar : PChar;
begin
  Write(' Enter: ');
  Readln(TheName);
  if (TheName = '')                          {allow '' for 'n'}
    then TheName := 'n';                     {default: Return = 'n'}
  TheName := Copy(TheName, 1, NameKeyLength - 1);  {NameKey = String[40]}
  StrPCopy(ThePChar, TheName);
  DBInputNameKey := StrLower(ThePChar);
end;   {DBInputNameKey}



 {****************************************
 *
 *       function DBGetCurrent
 *
 ****************************************}
function DBGetCurrent : LongInt;
  {return current date and time for timestamping nodes}
var
  Time : LongInt;
  DT : TDateTime;
  DayOfWeek, Sec100 : Word;
begin
  GetDate(DT.Year, DT.Month, DT.Day, DayOfWeek);
  GetTime(DT.Hour, DT.Min, DT.Sec, Sec100);
  PackTime(Dt, Time);
  DBGetCurrent := Time;
end;   {DBGetCurrent}



 {****************************************
 *
```

```
 *       function DBTest
 *
 ****************************************}
function DBTest: Boolean;
  {test suite for this unit; return true if tests passed}
var
  Ok   : Boolean;
  Obj  : PPersistentObject;
  BN   : PHermesNode;
  BL   : PLink;
  BT   : PLinkType;
  Name : array[0..NameKeyLength] of Char;
  Id   : LongInt;
  TheList  : PHermesList;
begin
  Writeln('*** test suite for HerDataB ***');

  StrCopy(Name, 'Issues              ');
  BN := New(PHermesNode, Init(Name));
  BN^.SetName(Name);
  Ok := DBPutName(BN);

  Obj := New(PPersistentObject, Init(NoName));
  Ok := DBPut(Obj);

  StrCopy(Name, 'D                   ');
  BN := New(PHermesNode, Init(NoName));
  BN^.SetName(Name);
  Ok := DBPutName(BN);

  StrCopy(Name, 'E                   ');
  BL := New(PLink, Init);
  BL^.SetName(Name);
  Ok := DBPutName(BL);

  StrCopy(Name, 'A                   ');
  BN := New(PHermesNode, Init(NoName));
  BN^.SetName(Name);
  Ok := DBPutName(BN);

  StrCopy(Name, 'Answers             ');
  BT := New(PLinkType, Init(Name));
  Ok := DBPutName(BT);

  Ok := DBDeleteName('Issues              ', 'Z');
  if Ok
    then Writeln('deleted it')
    else Writeln('deletion failed');

  StrCopy(Name, 'Arguments           ');
  BT := New(PLinkType, Init(Name));
  Ok := DBPutName(BT);

  Obj := New(PPersistentObject, Init(NoName));
  Ok := DBPut(Obj);

  Obj := New(PPersistentObject, Init(NoName));
  Ok := DBPut(Obj);

  HerNamesList^.DisplayList('listing of HerNamesList:');

  HerObjectsList^.DisplayList('listing of HerObjectsList:');

  TheList := DBListNames(ot_HermesNode);
  TheList := DBListNames(ot_Link);
  TheList := DBListNames(ot_LinkTYpe);
  TheList := DBListNames(ot_NodeKind);
  TheList := DBListObjects;

  Writeln('*** test suite completed ***');
end;  {DBTest}



 {****************************************
 *
```

```
 *       procedure View
 *
 ****************************************}
procedure View(Message: PChar; Destination: PChar; LineFeed: Boolean);
begin
  Write(Message);         {for TurboVision interface, change to}
  if LineFeed             {insert in collection for lister}
    then Writeln;
end;  {Views}



end.    {HerDataB}
```

11. HerGraph.pas define graphics procedures

```
 {*****************************************
 *
 *      HerGraph.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 *****************************************}

unit HerGraph;

 { This unit defines useful graphics objects and functions.  }

interface

uses HerNodes, HerHyper;

type

  PTemp = ^TTemp;
  TTemp = object(THermesGraphic)
  end;

  {***************************************
        *
        *    function DistanceBetween
        *
        ***************************************}

  function DistanceBetween(Distance: Terminology; Units: Terminology; FromGraphic: PHermesGraphic;
                                     ToGraphic: PHermesGraphic; InGraphic: PHermesGraphic) : Real;

implementation


  {***************************************
        *
        *    function DistanceBetween
        *
        ***************************************}

  function DistanceBetween(Distance: Terminology; Units: Terminology; FromGraphic: PHermesGraphic;
                                     ToGraphic: PHermesGraphic; InGraphic: PHermesGraphic) : Real;
  begin
    case Distance of
      CENTRAL_DISTANCE  : begin end;
      CLOSEST_DISTANCE  : begin end;
      X_DISTANCE        : begin end;
      Y_DISTANCE        : begin end;
      Z_DISTANCE        :
    end; {case}
    case Units of
      INCHES      : begin end;
      FEET        : begin end;
      CENTIMETERS : begin end;
      METERS      : begin end;
    end; {case}
  end;


end.       {HerGraph}
```

```
 {*****************************************
 *
 *               HerPersp.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 *****************************************}

unit HerPersp;

interface

uses Objects, Strings,
     HerLists, HerBasic, HerLinks;

{ This unit defines the Hermes perspectives mechanism
  as proposed in (Stahl, Ph.D. dissertation, 1993).

  It also creates Test Data and runs tests:

    1.  create a tree of Contexts and display it.

    2.  create a tree of nodes and links with Contexts
        and traverse from a StartNode in a Context.

    3.  provide procedures to implement the various methods of
        virtual copying.


       SPECIFICATION:

    Method 1:  Virtual copy of one context into another.

    Method 2:  Re-use of a subnetwork by switching contexts.

    Method 3:  Modification within a context
               a.  Delete a node in a context.
               b.  Modify a node by delete list and replace.
               c.  Delete a link in a context.
               d.  Modify a link in a context.

    Method 4:  Virtual copy an individual node
               a.  Virtual copy one node.
               b.  Virtual copy a node and its subtree of linked nodes.
               c.  Virtual copy one node and do others in future traversals.

    Method 5:  Physical copy an individual node.


       DESIGN:

    Method 1:  procedure CopyContext(OldContext, NewContext: LongInt);
               (if NewContext does not exist then create it;
                add a link to OldContext)


    Method 2:  procedure CreateHypertextSwitch(LinkName, FromNode, ToNode: LongInt;
                                   LinkType: PChar; Context, SwitchContext : LongInt);
               (if LinkName does not exist then create it;
                add new SubList with Switch to LinkName.)

    Method 3:
               a.  procedure DeleteNode(OldNode, Context : LongInt);
                   (for all InLinks and all OutLinks from OldNode,
                    for all SubLinks that fit add Context to DeleteLists.
                    Could actually delete OldNode if it has just 1 SubLink
                    and OriginalContext = Context and Switch = nil and
                    Adds = nil; must also delete all InLinks and all
                    OutLinks, including references to them in any nodes)

               b.  procedure ModifyNode(NodeName, OldContent, NewContent, Context : LongInt);
```

```
                    (add Context to DeleteList of all SubLinks of
                     link from OldNode to OldContent;
                     add link from OldNode to NewContent in Context)

                c.  procedure DeleteLink(OldLink, Context : LongInt);
                    (for each SubLink that fits add Context to delete list)

                d.  procedure ModifyLink(OldLink, Context : LongInt);
                    (delete as above; add new Sublink with Context)

    Method 4:
                a.  procedure VCopyNode(OldNode, OldContext, NewContext : LongInt);
                    (for all InLinks and all OutLinks from OldNode,
                     for all SubLinks that fit OldContext
                     add NewContext to AddLists.)

                b.  procedure VCopySubtree(OldNode, OldContext, NewContext : LongInt);
                    (do VCopyNode of original OldNode; then recursively do
                     VCopySubtree of its children nodes in OldContext.)

                c.  procedure VCopyLazySubtree(OldNode, OldContext, NewContext : LongInt);
                    (do 4a; then set Behavior to 4a in links out of OldNode)


    Method 5:  procedure PCopyNode(OldNode, OldContext, NewContext : LongInt);
               (get contents of OldNode in OldContext; for each content make
                a copy and link it to OldNode in NewContext.)
}

const
      HermesContentLinkType  : PChar = 'hermes_content';
      HermesContentNodeKind  : PChar = 'hermes_content';
      HermesContextLinkType  : PChar = 'hermes_context';
      HermesContextNodeKind  : PChar = 'hermes_context';
      HermesGraphicLinkType  : PChar = 'hermes_graphic';
      HermesGraphicNodeKind  : PChar = 'hermes_graphic';
      HermesUniversalContext : PChar = 'hermes_universal_context';
      HermesUniversalContextId : LongInt = 0;

        {****************************************
        *
        *    procedure headers
        *
        ****************************************}

  function TraversesLink(var TheKey : LongInt; TheLink : PLink) : Boolean;

  function TraversesSublink(var TheKey : LongInt; TheSubLink : PSubLink) : Boolean;

  procedure AddRationale(FromNode, NewNode, LinkType, Context, Text : PChar);

  procedure ModifyRationale(FromNode, NewNode, LinkType, Context, Text : PChar);

  procedure CopyContextNames(OldContext, NewContext: PChar);

  procedure CreateType(Name: PChar);

  procedure CreateKind(Name: PChar);

  procedure CreateContextNode(Name: PChar);

  procedure CreateBooleanNode(Name: PChar; Context: PChar);

  procedure CreateNumericNode(Name: PChar; Context: PChar);

  procedure CreateHyperTextNode(Name: PChar; Text: PChar; Context: PChar);

  procedure CreateHypertextLink(LinkName, FromNode, ToNode, LinkType, Context : PChar);

  procedure CreateHypertextSwitch( LinkId, FromNode, ToNode: LongInt; LinkType: PChar;
                                   Context, SwitchContext : LongInt; NewSublink : PSubLink );

  procedure CreateTextContent(NodeNameId: LongInt; Text: PChar; Context: LongInt);

  procedure DeleteNode(OldNode, Context : LongInt);
(*
```

```
  procedure ModifyNode(NodeId, ContentId, ContextId : LongInt);
*)
  procedure TestModifyNode(OldNode, Context : LongInt);

  procedure ModifyNode( NodeId, OldContentId, NewContentId, ContextId: LongInt;
                     NewIndex: Integer; NewTransform: PTransform; NewAttribute: PAttribute );

  procedure DeleteLink(OldLink, Context : LongInt);

  procedure ModifyLink(OldLink, Context : LongInt);

  procedure VCopyNode(OldNode, OldContext, NewContext : LongInt);

  procedure VCopySubtree(OldNode, OldContext, NewContext : LongInt);

  procedure VCopyLazySubtree(OldNode, OldContext, NewContext : LongInt);

  procedure PCopyNode(OldNode, OldContext, NewContext : LongInt);

  function DisplayContextTree: PHermesList;

  function DisplayInheritedContexts(OriginalContext: PChar) : PHermesList;

  function DisplayHypertextTree(CurrentNode: PChar; CurrentContext: PChar) : PHermesList;

{=====================================================================}

implementation

uses WinProcs, WinTypes, OWindows, ODialogs,
     HerNodes, HerMedia, HerDataB, HerWorld, HerLangu, HerPrivs;

const

 TheNextId   : LongInt = 100;        {counter for consequtive new ObjectIds}
 TempContext : LongInt = 100;         {storage for Context in ModifyNode and DeleteNode}

function GetContextParents(Key : LongInt; var ParentList : PLinkList) : Boolean;
 {*   return list of parents of a Context in ContextTree }
 var TheIndex : Integer;
     TheNode  : PHermesNode;
     TheList  : PLinkList;
     TheOutLinks : PLinkTree;
 begin
   ParentList := nil;
   TheNode := PHermesNode(DBGet(Key));
   if assigned( TheNode ) then
    begin
     TheOutLinks := TheNode^.GetOutLinks;
     TheList := New(PLinkList, Init);
     TheList^.SetName(HermesContextLinkType);
     if TheOutLinks^.Search(TheList, TheIndex) then
        ParentList := TheOutLinks^.At(TheIndex);
     Dispose(TheList, Done);
   end;
   GetContextParents := assigned( ParentList );
 end;


function GetContextChildren(Key : LongInt; var ChildList : PLinkList) : Boolean;
 {*   return list of children of a Context in ContextTree }
 var TheIndex : Integer;
     TheNode  : PHermesNode;
     TheInLinks : PLinkTree;
     TheList  : PLinkList;
 begin
   ChildList := nil;
   TheNode := PHermesNode(DBGet(Key));
   if assigned( TheNode ) then
    begin
      TheInLinks := TheNode^.GetInLinks;
      TheList := New(PLinkList, Init);
      TheList^.SetName(HermesContextLinkType);
      if TheInLinks^.Search(TheList, TheIndex)
      then ChildList := TheInLinks^.At(TheIndex);
      Dispose(TheList, Done);
```

```
    end;
   GetContextChildren := (ChildList <> nil);
 end;


function TraversesSublink( var TheKey : LongInt; TheSubLink : PSubLink ) : Boolean;
 {*   called in Node traversal and by TraversesLink to check if
  *   the current key Context Traverses the SubLink }
 var {TheCondition  : PCondition;}
     TheBehavior   : PBehavior;
     TheContexts   : PContextsList;
     TheOriginal   : LongInt;
     TheAdded      : PLinkList;
     TheDeleted    : PLinkList;
     TheSwitch     : LongInt;
     ReturnValue   : Boolean;

 function Fits(Key, Lock : LongInt) : Boolean;
  { called by TraversesLink to recursively check if Key Context
    fits the Lock Contexts defined for a given Link}

  function Ancestor(Key, Lock : LongInt) : Boolean;

   function OpensLock(ALinkName : PLinkId) : Boolean; far;
    var TheAncestorKey : LongInt;
        TheLink        : PLink;
    begin  {OpensLock}
      TheLink := PLink(DBGet(ALinkName^.GetId));
      if assigned( TheLink ) then
       begin
         TheAncestorKey := TheLink^.GetToPO;
         OpensLock := (TheAncestorKey = Lock)
                      or (Ancestor(TheAncestorKey, Lock));
       end
      else ErrorMsg(et_NOTICE, 'OpensLock: Link not found');
    end;   {OpensLock}

   var ParentList : PLinkList;
   begin  {Ancestor}
     if GetContextParents(Key, ParentList) then
        Ancestor := (ParentList^.FirstThat(@OpensLock) <> nil)
     else Ancestor := False;
   end;   {Ancestor}

  begin  {Fits}
    Fits := (Key = HermesUniversalContextId)    {Fits if key is universal}
         or (Lock = HermesUniversalContextId)   {Fits if lock is universal}
         or (Key = Lock)                        {Fits if key = lock}
         or (Ancestor(Key, Lock));              {Fits if ancestor of key = lock}
  end;   {Fits}

 function KeyFits(TheContextItem : PLinkId) : Boolean; far;
  begin   {KeyFits}
    KeyFits := Fits(TheKey, TheContextItem^.GetId);
  end;    {KeyFits}

 procedure DoProcedure(Item : PBehaviorListItem); far;
  var YesNo  : Integer;
      Param1 : LongInt;
      Param2 : LongInt;
      Param3 : LongInt;
      Param4 : LongInt;
      TheName   : PChar;
      TheString : StrName;
  begin  {DoProcedure}
    WITH Item^ do
     begin
       case GetTheProcedure of
         LazyVCopy : begin
                       Param1 := LongInt(GetParameter1);
                       Param2 := LongInt(GetParameter2);
                       Param3 := LongInt(GetParameter3);
                       TheBehavior^.DeleteProcedure(Item);
                       TheName := DBNameOf(Param1);
                       if StrLen(TheName) > 1 then
                          YesNo := MessageBox(Application^.MainWindow^.HWindow, DBNameOf(Param1),
```

```
                                                    'VCopy this node?', mb_YesNo or mb_IconQuestion)
                        else
                         begin
                           Str(Param1, TheString);
                           YesNo := MessageBox(Application^.MainWindow^.HWindow, TheString,
                                                 'VCopy this node?', mb_YesNo or mb_IconQuestion);
                         end;
                        if (YesNo = id_Yes) then
                           VCopyLazySubtree(Param1, Param2, Param3);
                      end;      {set new Procedures and delete the old one}

         NONE      : begin
                      end;
        else
           Errormsg(et_NOTICE, 'procedure not legal');
        end;  {case}
     end;  {with}
  end;   {DoProcedure}

 begin   {TraversesSublink}
   if (HermesUniversalContextId = 0) then
      HermesUniversalContextId := DBIdOf(HermesUniversalContext, ot_Context);

   {TheCondition  := TheSubLink^.GetCondition;}
   TheBehavior   := TheSubLink^.GetBehavior;
   TheContexts   := TheSubLink^.GetListofContexts;
   TheOriginal   := TheContexts^.GetOriginalContext;
   TheAdded      := TheContexts^.GetAddedContexts;
   TheDeleted    := TheContexts^.GetDeletedContexts;
   TheSwitch     := TheContexts^.GetSwitchContext;

   if (TheDeleted^.FirstThat(@KeyFits) = nil ) then {if TheKey not on Delete list}
    begin
      if Fits(TheKey, TheOriginal) then             {then check if it fits TheOriginal}
         ReturnValue := True                        {or if it fits one of TheAdded}
      else ReturnValue := (TheAdded^.FirstThat(@KeyFits) <> nil)
    end
   else
      ReturnValue := False;

 { if ReturnValue
     then if ((TheCondition <> 0) and not TheCondition^.EvalBool)
       then ReturnValue := False;      }   {Check Condition}

   TraversesSublink  := ReturnValue;

   if ReturnValue then                {if found a fit}
    begin
      if assigned ( TheBehavior ) then                {execute the Behaviors}
         TheBehavior^.ForEach(@DoProcedure);

      if (TheSwitch <> 0) then           {and it requires a context switch}
         TheKey := TheSwitch;    {then return new context as TheKey}
    end;
 end;   {TraversesSublink}


       {****************************************
       *
       *   function TraversesLink;
       *
  *   called in Node traversal to check if
  *   the current key Context Traverses at
  *   least one SubLink of the given Link
       ****************************************}

function TraversesLink(var TheKey : LongInt; TheLink : PLink) : Boolean;

    function OneTraverses(TheSubLink : PSubLink) : Boolean; far;
    begin  {OneTraverses}
      OneTraverses := TraversesSublink(TheKey, TheSubLink);
    end;   {OneTraverses}

begin  {TraversesLink}
  if (HermesUniversalContextId = 0)
    then HermesUniversalContextId := DBIdOf(HermesUniversalContext, ot_Context);
```

```
  if (TheKey = HermesUniversalContextId)
    then TraversesLink := True
    else TraversesLink := (TheLink^.GetSubLinks^.FirstThat(@OneTraverses) <> nil);
end;   {TraversesLink}


        {****************************************
        *
        *   function GetNodeChildren
        *
  *   return list of children Nodes of a Node in a Context
        ****************************************}

function GetNodeChildren(OldNode, OldContext : LongInt) : PLinkList;
var
  ChildrenList : PLinkList;
  TheNode  : PHermesNode;

    procedure ListChildren(AList : PLinkList); far;

        procedure ListAChild(ALink : PLinkId); far;
        var
          TheLink :PLink;
        begin  {ListAChild}
          TheLink := PLink(DBGet(ALink^.GetId));
          if (TheLink <> nil)
            then ChildrenList^.Insert(New(PLinkId, Init(TheLink^.GetToPO)))
            else ErrorMsg(et_NOTICE, 'ListAChild: Link not found');
        end;   {ListAChild}

    begin  {ListChildren}
      AList^.ForEach(@ListAChild);
    end;   {ListChildren}

begin  {GetNodeChildren}
  ChildrenList := New(PLinkList, Init);
  TheNode := PHermesNode(DBGet(OldNode));
  if TheNode <> nil                 {if Node found}
    then TheNode^.GetOutLinks^.ForEach(@ListChildren)
    else ErrorMsg(et_NOTICE, 'GetNodeChildren: OldNode not found');
  GetNodeChildren := ChildrenList;
end;  {GetNodeChildren}


        {****************************************
        *
        *   function GetNodeContents
        *
  *   return list of content Nodes of a Node in a Context
        ****************************************}

function GetNodeContents(OldNode, OldContext : LongInt) : PLinkList;
var
  ContentList : PLinkList;
  TheNode  : PHermesNode;

        procedure ListAContent(ALink : PLinkId); far;
        var
          TheLink : PLink;
        begin  {ListAContent}
          TheLink := PLink(DBGet(ALink^.GetId));
                                                  {check for OldContext}
          if (  (TheLink <> nil) and TraversesLink(OldContext, TheLink)  )
            then ContentList^.Insert(New(PLinkId, Init(TheLink^.GetToPO)))
            else ErrorMsg(et_NOTICE, 'ListAContent: Link not found');
        end;   {ListAContent}

begin  {GetNodeContents}
  ContentList := New(PLinkList, Init);
  TheNode := PHermesNode(DBGet(OldNode));
  if TheNode <> nil
    then
      TheNode^.GetContentOutLinks^.ForEach(@ListAContent)
      else ErrorMsg(et_NOTICE, 'GetNodeContents: OldNode not found');
  GetNodeContents := ContentList;
```

```
end;  {GetNodeContents}


        {****************************************
        *
        *       procedure AddRationale
  *
        ****************************************}

procedure AddRationale(FromNode, NewNode, LinkType, Context, Text : PChar);
var
  TheNode    : PCharacterSimple;
  TheNamerId : LongInt;
begin
  TheNode := New(PCharacterSimple, Init(NewNode, Text));
  TheNamerId := DBPutNamedObject(TheNode, DBIdOf(Context, ot_Context));
  CreateHypertextSwitch(0, DBIdOf(FromNode, ot_Character), TheNamerId, LinkType, DBIdOf(Context,
ot_Context), 0, nil);
end;


        {****************************************
        *
        *       procedure ModifyRationale
  *
        ****************************************}

procedure ModifyRationale(FromNode, NewNode, LinkType, Context, Text : PChar);
var
  TheNode    : PCharacterSimple;
  TheNamerId : LongInt;
begin
  TheNode := New(PCharacterSimple, Init(NewNode, Text));
  TheNamerId := DBPutNamedObject(TheNode, DBIdOf(Context, ot_Context));
  ModifyNode(TheNamerId, 0, TheNode^.GetId, DBIdOf(Context, ot_Context), 0, nil, nil);
end;


        {****************************************
        *
        *   procedure CreateType;
        *
  *   create a link Type
        ****************************************}

procedure CreateType(Name: PChar);
begin
  DBPutName(New(PLinkType, Init(Name)));
end;


        {****************************************
        *
        *   procedure CreateKind;
        *
  *   create a node Kind
        ****************************************}

procedure CreateKind(Name: PChar);
begin
  DBPutName(New(PNodeKind, Init(Name)));
end;


        {****************************************
        *
        *   procedure CreateContextNode;
        *
  *   create a Context node for the Context Tree
        ****************************************}

procedure CreateContextNode(Name: PChar);
var
  TheNode  : PContext;
begin
  TheNode := New(PContext, Init(Name));
```

```
  DBPutName(TheNode);
end;


        {*************************************
        *
        *   procedure CreateContextLink;
        *
  *   link up two Nodes in the ContextTree
        ****************************************}

procedure CreateContextLink(FromNode, ToNode : LongInt);
var
  TheLink     : PContextLink;
  TheToNode   : PContext;
  TheFromNode : PContext;
  TestList    : PLinkList;
  TestTree    : PLinkTree;
  TheIndex    : Integer;

    function Matches(ALinkId : PLinkId): Boolean; far;
    var
      TestLink : PContextLink;
      Result   : Boolean;
    begin  {Matches}
      TestLink := PContextLink(DBGet(ALinkId^.GetId));
      if TestLink <> nil
        then Result := ((TestLink^.GetToPO = ToNode) and (TestLink^.GetFromPO = FromNode))
        else Result := False;
      Matches := Result;
    end;   {Matches}

begin   {CreateContextLink}
  TheFromNode := PContext(DBGet(FromNode));
  if assigned(TheFromNode)                          {if found FromNode}
    then
      begin
        TheToNode := PContext(DBGet(ToNode));
        if assigned(TheToNode)                      {if found ToNode}
          then
            begin
              TestList := New(PLinkList, Init);
              TestList^.SetName(HermesContextLinkType);
              TestTree := TheToNode^.GetInLinks;
              if TestTree^.Search(TestList, TheIndex)
                then
                  begin
                    Dispose(TestList, Done);
                    TestList := TestTree^.At(TheIndex);
                  end;
              if TestList^.FirstThat(@Matches) = nil
                then                       {Link does not already exist}
                  begin
                    TheLink := New(PContextLink, Init);
                    TheLink^.SetFromPO(FromNode);
                    TheLink^.SetToPO(ToNode);
                    DBPut(TheLink);
                    TheToNode^.AddInLink(HermesContextLinkType, TheLink^.GetId);
                    TheFromNode^.AddOutLink(HermesContextLinkType, TheLink^.GetId);
                    DBPutName(TheToNode);
                    DBPutName(TheFromNode);
                  end;
            end
          else ErrorMsg(et_NOTICE, 'CreateContextLink: ToNode not found'); {update links in Nodes}
      end
    else ErrorMsg(et_NOTICE, 'CreateContextLink: FromNode not found');
end;    {CreateContextLink}


        {*************************************
        *
        *   procedure CreateHypertextLink;
        *
        ****************************************}

procedure CreateHypertextLink(LinkName, FromNode, ToNode, LinkType, Context : PChar);
```

```
begin
  if (StrLen(LinkName) = 0)
    then CreateHypertextSwitch(0, DBIdOf(FromNode, ot_Character),
                               DBIdOf(ToNode, ot_Character), LinkType, DBIdOf(Context, ot_Context),
0, nil)
    else CreateHypertextSwitch(DBIdOf(LinkName, ot_Link), DBIdOf(FromNode, ot_Character),
                               DBIdOf(ToNode, ot_Character), LinkType, DBIdOf(Context, ot_Context),
0, nil);
end;


       {****************************************
       *
       *   function CreateTextContent
       *
       ****************************************}

procedure CreateTextContent(NodeNameId: LongInt; Text: PChar; Context: LongInt);
 var TheContentId     : LongInt;
     TheContentNode   : PCharacterSimple;
 begin
   TheContentNode := New(PCharacterSimple, Init(NoName, Text));
   DBPut(TheContentNode);
   TheContentId := TheContentNode^.GetId;     {Id set by DBPut}
   CreateHypertextSwitch(0, NodeNameId, TheContentId, HermesContentLinkType, Context, 0, nil);
 end;



       {****************************************
       *
       *   procedure CreateHyperTextNode;
       *
       ****************************************}

procedure CreateHyperTextNode(Name: PChar; Text: PChar; Context: PChar);
 var TheNode : PCharacterSimple;
 begin
   TheNode := New(PCharacterSimple, Init(Name, Text));
   DBPutNamedObject(TheNode, DBIdOf(Context, ot_Context));
 end;


       {****************************************
       *
       *   procedure CreateNumericNode;
       *
       ****************************************}

procedure CreateNumericNode(Name: PChar; Context: PChar);
 var
   TheNode : PNumberSimple;
 begin
   TheNode := New(PNumberSimple, Init(Name, 3.1416));
   DBPutNamedObject(TheNode, DBIdOf(Context, ot_Context));
 end;


       {****************************************
       *
       *   procedure CreateBooleanNode;
       *
       ****************************************}

procedure CreateBooleanNode(Name: PChar; Context: PChar);
 var
   TheNode : PBooleanTrue;
 begin
   TheNode := New(PBooleanTrue, Init(Name));
   DBPutNamedObject(TheNode, DBIdOf(Context, ot_Context));
 end;


       {****************************************
       *
       *   procedure CopyContextNames
```

```
        *
        ****************************************}

procedure CopyContextNames(OldContext, NewContext: PChar);
 var TheContext : PContext;
 begin
   TheContext := PContext(DBGetName(NewContext, ot_Context));
   if not assigned( TheContext ) then
      CreateContextNode(NewContext);

   TheContext := PContext(DBGetName(OldContext, ot_Context));
   if assigned( TheContext ) then
      CreateContextLink(DBIdOf(NewContext, ot_Context), DBIdOf(OldContext, ot_Context))
   else
      ErrorMsg(et_NOTICE, 'CopyContext: OldContext not found');
 end;



        {****************************************
        *
        *         CreateHypertextSwitch;
        *
        ****************************************}

procedure CreateHypertextSwitch( LinkId, FromNode, ToNode: LongInt; LinkType: PChar;
                                 Context, SwitchContext : LongInt; NewSublink : PSubLink );

    function ContextTraversesSubLink(ASubLink : PSubLink) : Boolean; far;
    begin  {ContextTraversesSubLink}
      ContextTraversesSubLink := TraversesSubLink(Context, ASubLink);
    end;   {ContextTraversesSubLink}

var
  TheSublink  : PSublink;
  TheLink     : PLink;
  TheContexts : PContextsList;
  TheToNode   : PHermesNode;
  TheFromNode : PHermesNode;
  MainType    : PChar;
  TypeId      : LongInt;
  TheDeleteList : PLinkList;
  NewDeleteList : PLinkList;
  TheSubLinkList: PHermesItemsList;
begin
 if PrivsCreateLink(Context)    {do privilege checking in HerPrivs}
 then
 begin
  TheFromNode := PHermesNode(DBGet(FromNode));
  if assigned(TheFromNode) then                          {if found FromNode}
   begin
     TheToNode := PHermesNode(DBGet(ToNode));
     if assigned(TheToNode) then                         {if found ToNode}
      begin
        NewDeleteList := nil;
        TheLink := nil;
        if (LinkId = 0)                                        {if link not specified}
          then LinkId := DBGetLink(FromNode, ToNode, LinkType);       {then see if one exists}
        if (LinkId > 0)
          then TheLink := PLink(DBGet(LinkId));
        if not assigned( TheLink ) then
          begin
            TheLink := New(PLink, Init);           {if cant get link, create one}
            TheLink^.SetFromPO(FromNode);
            TheLink^.SetToPO(ToNode);
            TheLink^.SetTypeofLink(LinkType);
          end;

        if assigned( NewSublink )
          then TheSubLink := NewSubLink
          else New(TheSubLink, Init);            {create new SubLink}

        New(TheContexts, Init);           {create new ListofContexts for it}
        TheContexts^.SetOriginalContext(Context);        {insert Context}
        if (SwitchContext <> 0)
          then TheContexts^.SetSwitchContext(SwitchContext);   {insert SwitchContext}
```

```
        TheSubLink^.SetListofContexts(TheContexts);
        TheLink^.GetSubLinks^.Insert(TheSubLink);         {insert new SubList}

        DBPut(TheLink);
        LinkId := TheLink^.GetId;         {in case Id just assigned}

        if (StrComp(LinkType, HermesContentLinkType) = 0)
          then
            begin                                         {case HermesContentLinkType}
              TheFromNode^.AddContentOutLink(LinkId);
              if (TheToNode^.GetType = ot_HermesNode)
                then TheToNode^.AddContentInLink(LinkId);
            end
          else                                             {case HermesContextLinkType}
            if (StrComp(LinkType, HermesContextLinkType) = 0)
              then
                begin
                  TheFromNode^.AddOutLink(LinkType, LinkId);
                  TheToNode^.AddInLink(LinkType, LinkId);
                end
              else                                             {case HermesGraphicLinkType}
                if (StrComp(LinkType, HermesGraphicLinkType) = 0)
                  then
                    begin
                      TheFromNode^.AddContentOutLink(LinkId);
                      if (TheToNode^.GetType = ot_HermesGraphic)
                        then TheToNode^.AddContentInLink(LinkId);
                    end
                  else                                             {case other LinkType}
                    begin
                      TheFromNode^.AddOutLink(LinkType, LinkId);
                      TheToNode^.AddInLink(LinkType, LinkId);
                    end;

        DBPut(TheFromNode);
        DBPut(TheToNode);
      end    {if assigned(TheToNode)}
        else ErrorMsg(et_NOTICE, 'CreateHypertextSwitch: ToNode not found');
   end    {if assigned(TheFromNode)}
     else ErrorMsg(et_NOTICE, 'CreateHypertextSwitch: FromNode not found');
   end    {if PrivsCreateLink}
  else PrivsErrorMsg;
end;


        {***************************************
        *
        *   procedure DeleteNode
        *
  *   for all InLinks, OutLinks, ContentIn, ContentOut from OldNode,
  *   for all SubLinks add Context to DeleteLists.
  *   (Could actually delete OldNode if it has just 1 SubLink
  *   and OriginalContext = Context and Switch = nil and
  *   Adds = nil; must also delete all InLinks and all
  *   OutLinks, including references to them in any nodes.)
        ****************************************}

procedure DeleteNode(OldNode, Context : LongInt);
var
  TheNode     : PHermesNode;
  TheLinks    : PLinkTree;
  TheContents : PLinkList;

     procedure AddToDeleteList(ASubLink : PSubLink); far;
     var
        ContextsList    : PContextsList;
     begin  {AddToDeleteList}
        ContextsList := ASubLink^.GetListOfContexts;      {check edit privileges in HerPrivs}
        if PrivsEditnode(ContextsList^.GetOriginalContext, TempContext)
          then ContextsList^.GetDeletedContexts^.Insert(New(PLinkId, Init(TempContext)))
          else PrivsErrorMsg;
     end;   {AddToDeleteList}

    procedure DoTheLink(ALink : PLinkId); far;
    var
      TheLink  : PLink;
```

```
    begin   {DoTheLink}
       TheLink := PLink(DBGet(ALink^.GetId));
       if assigned(TheLink) then
        begin
          TheLink^.GetSubLinks^.ForEach(@AddToDeleteList);
          DBPut(TheLink);
        end
       else
          ErrorMsg(et_NOTICE, 'DeleteNode: TheLink not found');
    end;    {DoTheLink}

    procedure DoTheLinkList(AList : PLinkList); far;
    begin   {DoTheLinkList}
      if assigned(AList)
        then AList^.ForEach(@DoTheLink);
    end;    {DoTheLinkList}

 begin
   TempContext := Context;     {save current context in constant for this unit}
   TheNode := PHermesNode(DBGet(OldNode));
   if assigned(TheNode) then
    begin
      TheLinks := TheNode^.GetInLinks;
      if assigned( TheLinks )
        then TheLinks^.ForEach(@DoTheLinkList);
      TheLinks := TheNode^.GetOutLinks;
      if assigned(TheLinks )
        then TheLinks^.ForEach(@DoTheLinkList);
      TheContents := TheNode^.GetContentInLinks;
      if assigned( TheContents )
        then TheContents^.ForEach(@DoTheLink);
      TheContents := TheNode^.GetContentOutLinks;
      if assigned(TheContents )
        then TheContents^.ForEach(@DoTheLink);
    end
   else
      ErrorMsg(et_NOTICE, 'DeleteNode: OldNode not found');
 end;


       {****************************************
       *
       *   procedure ModifyNode
       *   revised for graphics transforms and attributes
       ****************************************}

procedure ModifyNode( NodeId, OldContentId, NewContentId, ContextId: LongInt;
                      NewIndex: Integer; NewTransform: PTransform; NewAttribute: PAttribute );
{ If NewContentId = 0 then no change to content.
  If NewTransform = nil then no change to Transform.
  If NewAttribute = nil then no change to Attribute.
  OldContentId must be <> 0 for multiple content and virtual copies.
  NewIndex = 0 is default for no virtual copies.
  1. If changing Content, then delete ContextId in OldLink and
     create NewLink to NewContentId.
  2. If changing Transform, then
     If ContextId = OriginalContext then modify Transform in Sublink
     else delete ContextId in Sublink and create NewSublink with ContextId.
  3. If changing Attribute, then proceed as in (2).   }
 var
   TheNode           : PHermesNode;
   OldLink           : PLink;
   TheLinkId         : PLinkId;
   OldSubLink        : PSubLink;
   NewLink           : PLink;
   NewSubLink        : PSubLink;

     function ContextTraversesLink(ALink : PLinkId) : Boolean; far;
       var
         TestLink : PLink;
         TheVarContext : LongInt;   {must be passed as a VAR parameter}
       begin  {ContextTraversesLink}
         TestLink := PLink(DBGet(ALink^.GetId));
         TheVarContext := TempContext;
         if (OldContentId > 0)
           then ContextTraversesLink := assigned(TestLink)
```

```
                                        and TraversesLink(TheVarContext, TestLink)
                                        and (TestLink^.GetToPO = OldContentId)
          else ContextTraversesLink := assigned(TestLink)
                                        and TraversesLink(TheVarContext, TestLink);
      end;   {ContextTraversesLink}

    function ContextTraversesSubLink(ASubLink : PSubLink) : Boolean; far;
      var
        TheVarContext : LongInt;   {must be passed as a VAR parameter}
      begin  {ContextTraversesSubLink}
        TheVarContext := TempContext;
        ContextTraversesSubLink := TraversesSubLink(TheVarContext, ASubLink);
      end;   {ContextTraversesSubLink}

    function GetTheSublink : PSublink;
      begin  {GetTheSublink}
        GetTheSublink := nil;
        OldLink := nil;
        TheNode := PHermesNode(DBGet(NodeId));
        if assigned(TheNode) then
          begin
            TheLinkId := TheNode^.GetContentOutlinks^.FirstThat(@ContextTraversesLink);
            if assigned(TheLinkId) then
              begin
                OldLink := PLink(DBGet(TheLinkId^.GetId));
                if assigned(OldLink) then
                  begin
                    if (NewIndex > 0)
                      then OldSublink := OldLink^.GetSublinks^.At(NewIndex)
                      else OldSublink := OldLink^.GetSublinks^.FirstThat(@ContextTraversesSublink);
                    if assigned(OldSublink)
                      then GetTheSublink := OldSublink
                      else ErrorMsg(et_NOTICE, 'ModifyNode: OldSubLink not assigned');
                  end
                else ErrorMsg(et_NOTICE, 'ModifyNode: OldLink not assigned');
              end
            else ErrorMsg(et_NOTICE, 'ModifyNode: TheLinkId not assigned');
          end
        else ErrorMsg(et_NOTICE, 'ModifyNode: TheNode not assigned');
      end;   {GetTheSublink}

 begin
   TempContext := ContextId;          {save current context in constant for this unit}
   if (NewContentId > 0) then        { 1. changing Content }
     begin
       OldSublink := GetTheSublink;
       if assigned(OldSubLink) then
         if PrivsEditNode(OldSubLink^.GetListOfContexts^.GetOriginalContext, ContextId)
           then
             begin
                     if assigned(OldSubLink) then
                OldSubLink^.GetListOfContexts^.GetDeletedContexts^.Insert(New(PLinkId,
Init(ContextId)));
               if assigned(OldLink)
                 then DBPut(OldLink);

               NewLink := New(PLink, Init);
               NewSublink := New(PSubLink, Init);
               NewSubLink^.GetListOfContexts^.SetOriginalContext(ContextId);
               NewLink^.GetSubLinks^.Insert(NewSubLink);
               NewLink^.SetTypeOfLink(HermesContentLinkType);
               NewLink^.SetFromPO(NodeId);
               NewLink^.SetToPO(NewContentId);
               DBPut(NewLink);

               TheNode^.AddContentOutLink(NewLink^.GetId);
               DBPut(TheNode);
             end
           else PrivsErrorMsg;
     end;   {(NewContentId > 0)}

   if assigned(NewTransform) then        { 2. changing Transform }
     begin
       OldSublink := GetTheSublink;
       if assigned(OldSublink) then
         begin
```

```
          if (ContextId <> OldSublink^.GetListOfContexts^.GetOriginalContext) then
            begin
              OldSubLink^.GetListOfContexts^.GetDeletedContexts^.Insert(New(PLinkId,
Init(ContextId)));

              NewSublink := New(PSublink, Init);
              NewSublink^.GetListOfContexts^.SetOriginalContext(ContextId);
              PGraphicalSublink(NewSublink)^.SetTransformation(NewTransform);
              OldLink^.GetSubLinks^.Insert(NewSublink);
            end
          else PGraphicalSublink(OldSublink)^.SetTransformation(NewTransform);
          DBPut(OldLink);
        end
      else ErrorMsg(et_NOTICE, 'ModifyNode (2): OldSubLink not assigned');
    end;   { assigned(NewTransform) }

   if assigned(NewAttribute) then        { 3. changing Attribute }
     begin
       OldSublink := GetTheSublink;
       if assigned(OldSublink) then
         begin
           if (ContextId <> OldSublink^.GetListOfContexts^.GetOriginalContext) then
             begin
               OldSubLink^.GetListOfContexts^.GetDeletedContexts^.Insert(New(PLinkId,
Init(ContextId)));

               NewSublink := New(PSublink, Init);
               NewSublink^.GetListOfContexts^.SetOriginalContext(ContextId);
               NewSublink^.SetAttribute(NewAttribute);
               OldLink^.GetSubLinks^.Insert(NewSublink);
             end
           else OldSublink^.SetAttribute(NewAttribute);
           DBPut(OldLink);
         end
       else ErrorMsg(et_NOTICE, 'ModifyNode (3): OldSubLink not assigned');
     end;     {assigned(NewAttribute)}
 end;


       {****************************************
       *
       *   procedure TestModifyNode
       *
       ****************************************}

procedure TestModifyNode(OldNode, Context : LongInt);
 var TheOldNode    : PHermesNode;
     OldLink       : LongInt;
     TheNewContent : PCharacterSimple;
     OldContent    : LongInt;
     NewContent    : LongInt;
 begin
   TheOldNode := PHermesNode(DBGet(OldNode));
   if assigned(TheOldNode) then
     begin
       OldLink := PLinkId(TheOldNode^.GetContentOutLinks^.At(0))^.GetId;
       OldContent := PPersistentObject(DBGetNamedObject(TheOldNode^.GetName, TheOldNode^.GetType,
Context))^.GetId;
       TheNewContent := New(PCharacterSimple, Init(NoName, 'alternative content'));
       DBPut(TheNewContent);
       NewContent := TheNewContent^.GetId;
       ModifyNode(OldNode, OldContent, NewContent, Context, 0, nil, nil);
     end
   else
    ErrorMsg(et_NOTICE, 'TestModifyNode: TheOldNode not found');
 end;


       {****************************************
       *
       *   procedure DeleteLink
       *
       ****************************************}

procedure DeleteLink(OldLink, Context : LongInt);
```

```
 procedure DeleteSubLink(ASubLink : PSubLink); far;
  begin  {DeleteSubLink}
    ASubLink^.GetListOfContexts^.GetDeletedContexts^.Insert(New(PLinkId, Init(Context)));
  end;   {DeleteSubLink}

 var TheLink  : PLink;
 begin  {DeleteLink}
   TheLink := PLink(DBGet(OldLink));
   if assigned( TheLink ) then
    begin
      TheLink^.GetSubLinks^.ForEach(@DeleteSubLink);
      DBPut(TheLink);
    end
   else
      ErrorMsg(et_NOTICE, 'DeleteLink: The link not found');
 end;   {DeleteLink}


       {****************************************
       *
       *   procedure ModifyLink
       *
       ****************************************}

procedure ModifyLink(OldLink, Context : LongInt);
 var TheLink     : PLink;
     TheContexts : PContextsList;
     TheSubLink  : PSubLink;
 begin
   DeleteLink(OldLink, Context);
   TheContexts := New(PContextsList, Init);
   TheContexts^.SetOriginalContext(Context);
   TheSubLink := New(PSubLink, Init);
   TheSubLink^.SetListOfContexts(TheContexts);

   TheLink := PLink(DBGet(OldLink));
   if assigned( TheLink ) then
    begin
      TheLink^.GetSubLinks^.Insert(TheSubLink);
      DBPut(TheLink);
    end
   else
      ErrorMsg(et_NOTICE, 'ModifyLink: The link not found');
 end;


       {****************************************
       *
       *   procedure VCopyNode
       *
  *   for all InLinks and all OutLinks from OldNode,
  *   for all SubLinks that fit OldContext
  *   add NewContext to AddLists.
       ****************************************}

procedure VCopyNode(OldNode, OldContext, NewContext : LongInt);

 procedure DoTheLinkList(AList : PLinkList); far;

  procedure DoTheLink(ALink : PLinkID); far;

   procedure AddToAddList(ASubLink : PSubLink); far;
    begin  {AddToAddList}
      ASubLink^.GetListOfContexts^.GetAddedContexts^.Insert(New(PLinkId, Init(NewContext)));
    end;   {AddToAddList}

   var TheLink  : PLink;
   begin   {DoTheLink}
     TheLink := PLink(DBGet(ALink^.GetId));
     if assigned( TheLink ) then
      begin
        if TraversesLink(OldContext, TheLink) then
         begin
           TheLink^.GetSubLinks^.ForEach(@AddToAddList);
           DBPut(TheLink);
         end;
```

```
      end
     else ErrorMsg(et_NOTICE, 'VCopyNode: TheLink not found');
   end;    {DoTheLink}

  begin   {DoTheLinkList}
    AList^.ForEach(@DoTheLink);
  end;    {DoTheLinkList}

 var TheNode     : PHermesNode;
     TheLinks    : PLinkTree;
     TheLinkList : PLinkList;
     TheIndex    : Integer;
 begin  {VCopyNode}
   TheNode := PHermesNode(DBGet(OldNode));
   if assigned ( TheNode ) then
    begin
      TheLinks := TheNode^.GetInLinks;
      if assigned( TheLinks ) then
         TheLinks^.ForEach(@DoTheLinkList);
      TheLinks := TheNode^.GetOutLinks;
      if assigned( TheLinks ) then
         TheLinks^.ForEach(@DoTheLinkList);
      TheLinkList := TheNode^.GetContentOutLinks;
      if assigned( TheLinkList ) then
         DoTheLinkList(TheLinkList);
    end
   else
      ErrorMsg(et_NOTICE, 'VCopyNode: OldNode not found');
 end;  {VCopyNode}



       {****************************************
       *
       *   procedure VCopySubtree
       *
  *   (do VCopyNode of original OldNode;
  *    then recursively do VCopySubtree of its
  *    children nodes in OldContext.)
  *   (Could be more efficient by not using
  *    calls to other routines,
  *    but Nodes and Links are already
  *    in memory anyway.)
       ****************************************}

procedure VCopySubtree(OldNode, OldContext, NewContext : LongInt);

 procedure Recurse(ChildNode : PLinkId); far;
  begin  {Recurse}
    VCopySubtree(ChildNode^.GetId, OldContext, NewContext);
  end;   {Recurse}

 begin {VCopySubtree}
   VCopyNode(OldNode, OldContext, NewContext);
   GetNodeChildren(OldNode, OldContext)^.ForEach(@Recurse);
 end;  {VCopySubtree}


       {****************************************
       *
       *   procedure VCopyLazySubtree
       *
       ****************************************}

procedure VCopyLazySubtree(OldNode, OldContext, NewContext : LongInt);

 procedure DoList(AList : PLinkList); far;

  procedure DoLink(LinkId : PLinkId); far;
   var NewProcedure : PBehaviorListItem;
       ALink        : PLink;
       TheIndex     : Integer;

   procedure DoSubLink(ASubLink : PSubLink); far;
    var NewBehavior : PBehavior;
    begin  {DoSubLink}
```

```
      if ((OldContext = HermesUniversalContextId) or TraversesSublink(OldContext, ASubLink)) then
       begin
         NewProcedure := New(PBehaviorListItem, Init);
         NewProcedure^.SetTheProcedure(LazyVCopy);
         NewProcedure^.SetParameter1(ALink^.GetToPO);
         NewProcedure^.SetParameter2(OldContext);
         NewProcedure^.SetParameter3(NewContext);
         NewProcedure^.SetParameter4(0);

         NewBehavior := ASubLink^.GetBehavior;
         if NOT assigned( NewBehavior ) then
            NewBehavior := New(PBehavior, Init);
         NewBehavior^.AddProcedure(NewProcedure);
         ASubLink^.SetBehavior(NewBehavior);
       end;
    end;   {DoSubLink}

   begin  {DoLink}
     ALink := PLink(DBGet(LinkId^.GetId));
     if assigned( ALink ) then
      begin
        ALink^.GetSubLinks^.ForEach(@DoSubLink);
        DBPut(ALink);
      end
     else
        ErrorMsg(et_NOTICE, 'VCopyLazySubtree: Link not found');
   end;   {DoLink}

  begin  {DoList}
    AList^.ForEach(@DoLink);
  end;   {DoList}

 var TheNode   : PHermesNode;
     TheIndex  : Integer;
 begin   {VCopyLazySubtree}
   VCopyNode(OldNode, OldContext, NewContext);      {VCopy OldNode}
   TheNode := PHermesNode(DBGet(OldNode));
   if assigned( TheNode ) then
      TheNode^.GetOutLinks^.ForEach(@DoList)
   else
      ErrorMsg(et_NOTICE, 'VCopyLazySubtree: Node not found');
 end;   {VCopyLazySubtree}


       {****************************************
       *
       *   procedure PCopyNode
       *
  *   get contents of OldNode in OldContext;
  *   for each content make a copy and
  *   link it to OldNode in NewContext
       ****************************************}

procedure PCopyNode(OldNode, OldContext, NewContext : LongInt);

 procedure CopyAnObject(AnObject : PLinkId); far;
  var TheObject  : PPersistentObject;
      TheCopy    : PPersistentObject;
      TheLink    : PLink;
      ObjectId   : LongInt;
  begin  {CopyAnObject}
    TheObject := DBGet(AnObject^.GetId);
    if assigned( TheObject ) then
     begin
       TheCopy := PPersistentObject( TheObject^.Copy );
       TheCopy^.SetName(NoName);
       TheCopy^.SetId(0);
       DBPut(TheCopy);
       CreateHypertextSwitch(0, OldNode, TheCopy^.GetId,
                             HermesContentLinkType, NewContext, 0, nil);
     end
    else ErrorMsg(et_NOTICE, 'PCopyNode: TheObject not found');
  end;   {CopyAnObject}

 begin  {PCopyNode}
   GetNodeContents(OldNode, OldContext)^.ForEach(@CopyAnObject);
```

```
 end;   {PCopyNode}


       {*************************************
       *
       *   procedure DisplayHypertextTree;
       *
       **************************************}

function DisplayHypertextTree(CurrentNode: PChar; CurrentContext: PChar) : PHermesList;
 var ResultList : PHermesList;
     TheIndex : Integer;

 procedure TraverseHypertextTree(StartNode : LongInt; Level : Integer; TheContext : LongInt);

  function Indent(TheText : PChar; Level : Integer) : PChar;
   var S : array[0..200] of Char;
       I : Integer;
   begin  {Indent}
     StrCopy(S, ' ');
     for I := 1 to Level do
       StrCat(S, '    ');      {indent 4 x Indent spaces}
     StrCat(S, TheText);
     Indent := S;
   end;   {Indent}

  procedure DisplayOne(ALinkItem : PLinkId);
   var TheLink        : PLink;
       TheContent     : LongInt;
       TheResult      : PChar;
       TheContentNode : PPersistentObject;
   begin  {DisplayOne}
     TheLink := PLink(DBGet(ALinkItem^.GetId));
     if assigned( TheLink ) then
      begin
        if TraversesLink(TheContext, TheLink) then      {check Context Fits}
         begin
           TheContent := TheLink^.GetToPO;
           TheContentNode := DBGet(TheContent);
           if assigned( TheContentNode ) then
            begin
              TheResult := StrNew(Indent(TheContentNode^.HermesDisplay, Level));
              ResultList^.Insert(TheResult);
            end
           else
              ErrorMsg(et_NOTICE, 'Node does not exist: ')
         end;
      end
     else
        ErrorMsg(et_NOTICE, 'Link does not exist: ');
   end;   {DisplayOne}

  procedure Recurse(ALinkList : PLinkList); far;

   procedure TraverseOne(ALinkId : PLinkId);
    var TheLink       : PLink;
        SwitchContext : LongInt;
    begin  {TraverseOne}
      TheLink := PLink(DBGet(ALinkId^.GetId));
      if assigned( TheLink ) AND
         TraversesLink(TheContext, TheLink) then      {check Context Fits}
         TraverseHypertextTree(TheLink^.GetToPO, Level + 1, TheContext);
                                                        {TheContext is a var parameter}
    end;   {TraverseOne}                                {it may have changed value}
                                                          {to switch contexts}
   begin  {Recurse}
     ALinkList^.ForEach(@TraverseOne);
   end;   {Recurse}

  var TheNode : PHermesNode;
  begin  {TraverseHypertextTree}
    TheNode := PHermesNode(DBGet(StartNode));
    if assigned( TheNode ) then
     begin
       TheNode^.GetContentOutLinks^.ForEach(@DisplayOne);
       TheNode^.GetOutLinks^.ForEach(@Recurse);   {then go thru its out-links}
```

```
     end
    else
       ErrorMsg(et_NOTICE, 'TraverseHypertextTree: Node does not exist');
  end;   {TraverseHypertextTree}

 begin  {DisplayHypertextTree}
   ResultList := New(PHermesList, Init);
   TraverseHypertextTree(DBIdOf(CurrentNode, 'O'), 0, DBIdOf(CurrentContext, ot_Context));
{indent = 0}
   DisplayHypertextTree := ResultList;
 end;   {DisplayHypertextTree}


       {****************************************
       *
       *   procedure DisplayContextTree;
       *
       ****************************************}

function DisplayContextTree: PHermesList;
 var ResultList : PHermesList;

 procedure TraverseContextTree(StartContext : LongInt; Level : Integer);
  var TheContext  : PContext;
      TheInLinks  : PLinkTree;
      TheLinkList : PLinkList;
      TheName     : LongInt;
      TheResult   : PChar;

  function Indent(Name : PChar; Level : Integer) : PChar;
   var S : array[0..200] of Char;
       I : Integer;
   begin  {Indent}
     StrCopy(S, ' ');
     for I := 1 to Level do
        StrCat(S, '    ');      {indent 4 x Indent spaces}
     StrCat(S, Name);          {concatenate Name}
     Indent := S;
   end;   {Indent}

  procedure TraverseOne(ALinkId : PLinkId);
   var TheLink  : PLink;
       TheIndex : Integer;
   begin  {TraverseOne}
     TheLink := PLink(DBGet(ALinkId^.GetId));
     if assigned( TheLink ) then
        TraverseContextTree(TheLink^.GetFromPO, Level + 1);
   end;   {TraverseOne}

  begin  {TraverseContextTree}
    TheResult := StrNew(Indent(DBNameOf(StartContext), Level));
    ResultList^.Insert(TheResult);                    {add Name to ResultList}

    if GetContextChildren(StartContext, TheLinkList) then
       TheLinkList^.ForEach(@TraverseOne);       {recurse}
  end;   {TraverseContextTree}

 begin  {DisplayContextTree}
   ResultList := New(PHermesList, Init);
   if (HermesUniversalContextId = 0) then
      HermesUniversalContextId := DBIdOf(HermesUniversalContext, ot_Context);
   TraverseContextTree(HermesUniversalContextId, 0);   {start at HermesUniversalContext}
   DisplayContextTree := ResultList;
 end;   {DisplayContextTree}


       {****************************************
       *
       *   procedure DisplayInheritedContexts;
       *
       ****************************************}

function DisplayInheritedContexts(OriginalContext: PChar) : PHermesList;
 var ResultList : PHermesList;
     OriginalId : LongInt;
```

```
 procedure TraverseContextTree(StartContext : LongInt; Level : Integer);
  var TheContext  : PContext;
      TheInLinks  : PLinkTree;
      TheLinkList : PLinkList;
      TheName     : LongInt;
      TheResult   : PChar;

  function Indent(Name : PChar; Level : Integer) : PChar;
   var S : array[0..80] of Char;
       I : Integer;
   begin  {Indent}
     StrCopy(S, ' ');
     for I := 1 to Level do
        StrCat(S, '    ');      {indent 4 x Indent spaces}
     StrCat(S, Name);          {concatenate Name}
     Indent := S;
   end;   {Indent}

  procedure TraverseOne(ALinkId : PLinkId);
   var TheLink  : PLink;
       TheIndex : Integer;
   begin  {TraverseOne}
     TheLink := PLink(DBGet(ALinkId^.GetId));
     if assigned( TheLink ) then
        TraverseContextTree(TheLink^.GetToPO, Level + 1);
   end;   {TraverseOne}

  begin  {TraverseContextTree}
    TheResult := StrNew(Indent(DBNameOf(StartContext), Level));
    ResultList^.Insert(TheResult);                    {add Name to ResultList}

    if GetContextParents(StartContext, TheLinkList) then
       TheLinkList^.ForEach(@TraverseOne);       {recurse}
  end;   {TraverseContextTree}

 begin  {DisplayInheritedContexts}
   ResultList := New(PHermesList, Init);
   OriginalId := DBIdOf(OriginalContext, ot_Context);
   TraverseContextTree(OriginalId, 0);   {start at OriginalContext}
   DisplayInheritedContexts := ResultList;
 end;   {DisplayInheritedContexts}


end.      {**** unit HerVCopy  ****}
```

13. HerWorld.pas define the main application

```
 {*****************************************
 *
 *      HerWorld.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 ******************************************}

unit HerWorld;

{   This unit defines HermesAPP, the Application, as THermesAPP.
    The Application's main window is defined as a Hermes MainWindow.
    This is an MDI main window which includes an MDI client area.
    The  THermesMainWindow controls the application's menu.
    MDI Child windows are defined as THermesChildWindow.

    This unit defines the main application and windows of the Hermes system
    as proposed in (Stahl, Ph.D. dissertation, 1993).

               Several constants for the Hermes Application are defined,
               including user preferences and constants for Stack Checking.
    The user can set the StackLimit at which evaluation is terminated to avoid a
    stack overflow from infinite loops in poorly designed Predicate definitions.
}

interface

{$define TADebug}               {turn on debugging mode in TAccess database system}

{$R Hermes}                     {use Hermes.RES for menu and other resources}

uses OWindows, Objects, ODialogs, WinDos, WinProcs, WinTypes, Strings, OStdDlgs,
     HerOGL2;

{$I HerMenu.INC}                {include  THermesMainWindow menu constants}

type

        {*************************************
        *                                    *
        *    data structure for THermesApp   *
  *                                    *
  **************************************}

        THermesApp = object (TApplication)
    procedure   InitMainWindow; virtual;
    procedure   InitInstance; virtual;
    destructor  Done; virtual;
    procedure   SetDBName(NewName: PChar); virtual;
    function    GetDBName: PChar; virtual;
    procedure   SetDBAuthor(NewAuthor: PChar); virtual;
    function    GetDBAuthor: PChar; virtual;
    procedure   SetTimeStamp(NewTimeStamp: Boolean); virtual;
               function    GetTimeStamp: Boolean; virtual;
    procedure   SetStackLimit(NewLimit: Integer); virtual;
    function    GetStackLimit: Integer; virtual;
    procedure   SetIndent(NewIndent: Integer); virtual;
    function    GetIndent: Integer; virtual;
    procedure   SetContextFast(NewContext: LongInt); virtual;
    procedure   SetContext(NewContext: LongInt); virtual;
    function    GetContext: LongInt; virtual;
    procedure    InitPreferences; virtual;
    procedure    SavePreferences; virtual;
  private
    DBName     : PChar;           {name of the object stream}
    DBAuthor   : PChar;           {user's login name}
    TimeStamp  : Boolean;         {if time stamp links should be maintained}
    StackLimit : Integer;         {limit for number of macro calls}
    Indent     : Integer;         {number of spaces to indent query results}
    Context    : LongInt;         {current context or perspective}
  end;
```

```
  {************************************
  *
        *    data structure for  THermesMainWindow
  *
  ************************************}

        PHermesMainWindow = ^THermesMainWindow;
        THermesMainWindow = object (TMDIWindow)
               constructor Init(ATitle: PChar; AMenu : HMenu);
               destructor  Done; virtual;

               procedure   SetUpWindow; virtual;
    function    CreateChild: PWindowsObject; virtual;
    procedure   UpdateWindowTitle; virtual;
    function    CanClose : Boolean; virtual;
    function    GetClassName: PChar; virtual;
    procedure   GetWindowClass(var AWndClass: TWndClass); virtual;

    function    NumberOfChildWindows     : Integer; virtual;
    function    NumberOfTextualWindows   : Integer; virtual;
    function    NumberOfGraphicalWindows : Integer; virtual;
    function    NumberOfControlWindows   : Integer; virtual;

    {*** File menu ***}

               procedure   CMFileNew                (var Msg : TMessage); virtual cm_First +
cm_FileNew;
               procedure   CMFileOpen               (var Msg : TMessage); virtual cm_First +
cm_FileOpen;
               procedure   CMFileSaveAs             (var Msg : TMessage); virtual cm_First +
cm_FileSaveAs;
               procedure   CMFileCommit             (var Msg : TMessage); virtual cm_First +
cm_FileCommit;
               procedure   CMFilePrint              (var Msg : TMessage); virtual cm_First +
cm_FilePrint;
               procedure   CMFilePrinterSetup       (var Msg : TMessage); virtual cm_First +
cm_FilePrinterSetup;
               procedure   CMFilePrinterChoose      (var Msg : TMessage); virtual cm_First +
cm_FilePrinterChoose;

    {*** Edit menu ***}

               {  procedure   CMFileExit
       procedure   CMEditUndo            (These methods are defined by Windows)
                  procedure   CMEditCut
                  procedure   CMEditCopy
                  procedure   CMEditPaste
                  procedure   CMEditClear
                  procedure   CMEditDelete  }
               procedure   CMEditName               (var Msg : TMessage); virtual cm_First +
cm_EditName;


    {*** Controls menu ***}

               procedure   CMControlsSelectContext  (var Msg : TMessage); virtual cm_First +
cm_ControlsSelectContext;
               procedure   CMControlsPerspective    (var Msg : TMessage); virtual cm_First +
cm_ControlsPerspective;
               procedure   CMControlsNavigator      (var Msg : TMessage); virtual cm_First +
cm_ControlsNavigator;
               procedure   CMControlsObjects        (var Msg : TMessage); virtual cm_First +
cm_ControlsObjects;
               procedure   CMControlsQueries        (var Msg : TMessage); virtual cm_First +
cm_ControlsQueries;
               procedure   CMControlsEntryForm      (var Msg : TMessage); virtual cm_First +
cm_ControlsEntryForm;
               procedure   CMControlsQueryResults   (var Msg : TMessage); virtual cm_First +
cm_ControlsQueryResults;
               procedure   CMControlsVCopy          (var Msg : TMessage); virtual cm_First +
cm_ControlsVCopy;
               procedure   CMControlsAdd            (var Msg : TMessage); virtual cm_First +
cm_ControlsAdd;
               procedure   CMControlsModify         (var Msg : TMessage); virtual cm_First +
cm_ControlsModify;
```

```
              procedure   CMControlsLink           (var Msg : TMessage); virtual cm_First +
cm_ControlsLink;

    {*** Contexts menu ***}

              procedure   CMContextsNew             (var Msg : TMessage); virtual cm_First +
cm_ContextsNew;
              procedure   CMContextsModify          (var Msg : TMessage); virtual cm_First +
cm_ContextsModify;
              procedure   CMContextsSelect          (var Msg : TMessage); virtual cm_First +
cm_ContextsSelect;
              procedure   CMContextsDisplay         (var Msg : TMessage); virtual cm_First +
cm_ContextsDisplay;


    {*** Logins menu ***}

              procedure   CMLoginsNew               (var Msg : TMessage); virtual cm_First +
cm_LoginsNew;
              procedure   CMLoginsModify            (var Msg : TMessage); virtual cm_First +
cm_LoginsModify;
              procedure   CMLoginsDelete            (var Msg : TMessage); virtual cm_First +
cm_LoginsDelete;
              procedure   CMLoginsDisplay           (var Msg : TMessage); virtual cm_First +
cm_LoginsDisplay;


    {*** Nodes menu ***}

              procedure   CMNodesView               (var Msg : TMessage); virtual cm_First +
cm_NodesView;
              procedure   CMNodesAnnotate           (var Msg : TMessage); virtual cm_First +
cm_NodesAnnotate;
              procedure   CMNodesCreate             (var Msg : TMessage); virtual cm_First +
cm_NodesCreate;
              procedure   CMNodesModify             (var Msg : TMessage); virtual cm_First +
cm_NodesModify;
              procedure   CMNodesDelete             (var Msg : TMessage); virtual cm_First +
cm_NodesDelete;
              procedure   CMNodesPromote            (var Msg : TMessage); virtual cm_First +
cm_NodesPromote;


    {*** Window menu ***}                {These methods are inherited from TMDIWindow}

    {*** Help menu ***}

              procedure   CMHelp                  (var Msg : TMessage); virtual cm_First + cm_Help;
              procedure   CMAbout                 (var Msg : TMessage); virtual cm_First +
cm_About;
       end;


  ChildType = (TEXTUAL, GRAPHICAL, CONTROL, TOOLBAR, CONTROLBAR, PALETTEBAR);


  {************************************
  *
       *    data structure for THermesChildWindow
  *
  *************************************}

       PHermesChildWindow = ^THermesChildWindow;
       THermesChildWindow = object (TGWindow)
              constructor Init(AParent : PWindowsObject; ATitle : PChar);
    function    CanClose : Boolean; virtual;
    procedure   SetChildType(NewChildType : ChildType);
    function    GetChildType : CHildType;
  private
    TheChildType : ChildType;
       end;

var
  HermesApp : THermesApp;

{=================================================================}
```

```
implementation

uses HerBasic, HerRegis, HerDataB, HerMedia, HerPersp,HerNodes, HerLinks,
     HerLists, HerHyper, HerOGL1, HerTxwin, HerCoWin, HerPrivs;

  {************************************
  *
        *    methods for THermesApp
  *
  ************************************}

procedure THermesApp.InitMainWindow;
begin
  RegisterWObjects;                                    {register all ObjectWindows objects -- in unit
WObjects}
        HerRegister;                                        {register all Hermes objects        -- in
unit HerRegis}
  RegisterOGL;                                         {register all ObjectGraphicsobjects -- in
StrmReg.Inc}
        MainWindow := New (PHermesMainWindow, Init(Application^.Name, LoadMenu(HInstance, 'HERMES')));
  PrivsSwitchOff;                                      {turn off privilege checking}
        InitPreferences;                                    {get database name, etc from
C:\Hermes\Hermes.Sta}
  DBOpenDatabase;                                      {open database Stream and its Index -- in unit
HerDataB}
  if (DBGetName(HermesUniversalContext, ot_Context) <> nil)
    then HermesUniversalContextId := DBIdOf(HermesUniversalContext, ot_Context)
    else HermesUniversalContextId := 0;

  PrivsSwitchOn;                                        {turn on privilege checking}
  if not PrivsCheckLogin then
      ErrorMsg(et_NOTICE, 'Invalid Login -- you will have guest viewing permissions.');
end;


procedure THermesApp.InitInstance;
begin
  inherited InitInstance;
  HAccTable := LoadAccelerators(HInstance, 'HERMES');
end;


destructor THermesApp.Done;
begin
  SavePreferences;           {save database name and preferences}
  StrDispose(DBName);
        DBAuthor  := ' ';
  TimeStamp := False;
  StackLimit := 0;
  Indent    := 0;
  inherited Done;
end;


procedure   THermesApp.SetDBName(NewName: PChar);
begin
  StrDispose(DBName);
  DBName := StrNew(NewName);            {set data of THermesApp}
end;


function    THermesApp.GetDBName: PChar;
begin
        GetDBName := StrNew(DBName);
end;


procedure   THermesApp.SetDBAuthor(NewAuthor: PChar);
begin
  DBAuthor := StrNew(NewAuthor);
end;


function    THermesApp.GetDBAuthor: PChar;
begin
```

```
  GetDBAuthor := StrNew(DBAuthor);
end;


procedure   THermesApp.SetTimeStamp(NewTimeStamp: Boolean);
begin
  TimeStamp := NewTimeStamp;
end;


function    THermesApp.GetTimeStamp: Boolean;
begin
  GetTimeStamp := TimeStamp;
end;


procedure   THermesApp.SetStackLimit(NewLimit: Integer);
begin
  StackLimit := NewLimit;
end;


function    THermesApp.GetStackLimit: Integer;
begin
  GetStackLimit := StackLimit;
end;


procedure   THermesApp.SetIndent(NewIndent: Integer);
begin
  Indent := NewIndent;
end;


function    THermesApp.GetIndent: Integer;
begin
  GetIndent := Indent;
end;


procedure   THermesApp.SetContextFast(NewContext: LongInt);
begin
  Context := NewContext;
end;


procedure   THermesApp.SetContext(NewContext: LongInt);
var
  TheTitle       : array[0..120] of Char;
begin
  Context := NewContext;
  StrCopy(TheTitle, Application^.Name);
  StrCat(TheTitle, '  [');
  StrCat(TheTitle, HermesApp.GetDBName);
  StrCat(TheTitle, ']  ');
  StrCat(TheTitle, DBNameOf(Context));
  SetWindowText(MainWindow^.HWindow, TheTitle);
end;


function    THermesApp.GetContext: LongInt;
begin
  GetContext := Context;
end;


procedure   THermesApp.InitPreferences;
var
       State_File : Text;
  STAName    : PChar;
  DirInfo    : TSearchRec;    {declared in Dos unit}
       OldName    : array[0..fsPathName] of Char;
  AString    : String;
  ANumber    : Longint;
  Code       : Integer;
begin
```

```
  STAName := 'C:\HERMES\Hermes.STA';          {state file for saving stream name}
  FindFirst(STAName, faArchive, DirInfo);           {try to find object stream}
  if (DosError = 0)                              {if it does exist}
    then                                         {then set name of stream}
                      begin
                              Assign(State_File, STAName);
                      Reset(State_File);
                      Readln(State_File, OldName);
                              SetDBName(OldName);                  {set DBName}
                      Readln(State_File, OldName);
        SetDBAuthor(OldName);                 {save DBAuthor}
        Readln(State_File, AString);
        Val(AString, ANumber, Code);
        SetStackLimit(ANumber);               {set StackLimit}
        Readln(State_File, AString);
        Val(AString, ANumber, Code);
        SetIndent(ANumber);                   {set Indent}
        Readln(State_File, AString);
        Val(AString, ANumber, Code);
        SetContextFast(ANumber);              {set Context}
        Readln(State_File, AString);
        if (AString = 'TRUE')
          then SetTimeStamp(true)
          else SetTimeStamp(false);           {set TimeStamp}
                      Close(State_File);
              end
              else
      begin
        StrCopy(OldName, 'Hermes.DBF');
                              SetDBName(OldName);                  {default stream name}
             SetDBAuthor('gerry');
        SetStackLimit(100);
        SetIndent(4);
        SetContextFast(0);                        {hermes_universal_context}
        SetTimeStamp(False);
                      end;
end;


procedure   THermesApp.SavePreferences;
var
       State_File : Text;
  STAName    : PChar;
  DirInfo    : TSearchRec;    {declared in Dos unit}
       POldName   : array[0..fsPathName] of Char;
  Number     : String;
begin
  STAName := 'C:\HERMES\Hermes.STA';    {state file for saving stream name}
       Assign(State_File, STAName);
       Rewrite(State_File);
       Writeln(State_File, GetDBName);       {save database stream name}
       Writeln(State_File, GetDBAuthor);     {save DBAuthor}
  Str(GetStackLimit, Number);
       Writeln(State_File, Number);          {save StackLimit}
  Str(GetIndent, Number);
       Writeln(State_File, Number);          {save Indent}
  Str(GetContext, Number);
       Writeln(State_File, Number);          {save Context}
  if GetTimeStamp
    then Writeln(State_File, 'TRUE')    {save TimeStamp}
    else Writeln(State_File, 'FALSE');
  Close(State_File);
end;


  {************************************
  *
       *    methods for  THermesMainWindow
  *
       ************************************}

constructor THermesMainWindow.Init(ATitle: PChar; AMenu: HMenu);
begin
  TMDIWindow.Init(ATitle, AMenu);
  ChildMenuPos := 9;                {position of Window menu}
  {PaletteBar := nil; }
```

```
  Attr.X := 0;
  Attr.Y := 0;
  Attr.W := 1024;     {max = 1024}
  Attr.H := 768;      {max = 768}
end;


destructor  THermesMainWindow.Done;
begin
       DBCloseDataBase;           {close Stream -- in HerDataB}
  TMDIWindow.Done;
end;


procedure   THermesMainWindow.SetUpWindow;
var
  NoticeWnd : PNotifyWin;
begin
  TMDIWindow.SetUpWindow;
  UpdateWindowTitle;
  Application^.MakeWindow(New(PSelectWin, Init(@Self)));
end;


function    THermesMainWindow.CreateChild;
begin
  TMDIWIndow.CreateChild;
end;


procedure   THermesMainWindow.UpdateWindowTitle;
var
  TheTitle       : array[0..120] of Char;
begin
  StrCopy(TheTitle, Application^.Name);
  StrCat(TheTitle, '    [');
  StrCat(TheTitle, HermesApp.GetDBName);
  StrCat(TheTitle, ']    ');
  StrCat(TheTitle, DBNameOf(HermesApp.GetContext));
  SetWindowText(HWindow, TheTitle);
end;


function    THermesMainWindow.CanClose : Boolean;
begin
  CanClose := (MessageBox(HWindow, 'Do you really want to exit Hermes?  Files will be packed and
backed up.',
               'Exit Hermes', mb_YesNo or mb_IconQuestion) = id_Yes);
end;


function   THermesMainWindow.GetClassName: PChar;
begin
  GetClassName := ' THermesMainWindow';
end;


procedure   THermesMainWindow.GetWindowClass(var AWndClass: TWndClass);
begin
  TMDIWindow.GetWindowClass(AWndClass);
  AWndClass.hIcon := LoadIcon(HInstance, 'HERMES');
end;


function    THermesMainWindow.NumberOfChildWindows     : Integer;
begin
  NumberOfChildWindows := 0;
end;


function    THermesMainWindow.NumberOfTextualWindows   : Integer;
begin
  NumberOfTextualWindows := 0;
end;
```

```
function    THermesMainWindow.NumberOfGraphicalWindows : Integer;
 begin
  NumberOfGraphicalWindows := 0;
end;


function    THermesMainWindow.NumberOfControlWindows   : Integer;
begin
  NumberOfControlWindows := 0;
end;


    {*** File menu ***}

procedure   THermesMainWindow.CMFileNew(var Msg : TMessage);
var
  TheFile: array[0..fsPathName] of Char;
begin
  StrCopy(TheFile, '*.DBF');
  if Application^.ExecDialog(New(PFileDialog, Init(@Self, PChar('FILE_NEW'), TheFile))) = id_OK
    then
      begin
        DBCloseDatabase;
        HermesApp.SetDBName(TheFile);
        DBOpenDatabase;
        UpdateWindowTitle;
      end;
end;


procedure    THermesMainWindow.CMFileOpen(var Msg : TMessage);
var
  TheFile: array[0..fsPathName] of Char;
begin
  StrCopy(TheFile, '*.DBF');
  if Application^.ExecDialog(New(PFileDialog, Init(@Self, PChar('FILE_OPEN'), TheFile))) = id_OK
    then
      begin
        DBCloseDatabase;
        HermesApp.SetDBName(TheFile);
        DBOpenDatabase;
        UpdateWindowTitle;
      end;
end;

procedure    THermesMainWindow.CMFilePrint(var Msg : TMessage);
var
  Page     : Integer;
  CopiesTxt: array [0..3] of Char;
  Copies   : Integer;
  NumCode  : Integer;
  Continue : Boolean;
begin
(*
  Str(1, CopiesTxt);
  if Application^.ExecDialog(New(PInputDialog, Init(@Self, 'Print',
                                 'Number of Copies?', CopiesTxt,
                               SizeOf(CopiesTxt))))
   = id_Ok
  then
  begin
    Val(CopiesTxt, Copies, NumCode);
    if NumCode <> 0 then Copies := 1;

    if Printer^.Start(FileName) > 0 then
    begin
      Page     := 1;
      Continue := True;
      while (Page <= Copies) and Continue do
      begin
        DrawWindow^.Print(Printer);
        if Page < Copies then        { Printer^.Finish does last one }
          Continue := (Printer^.NextFrame > 0);
        Inc(Page);
      end;
    end;
```

```
    Printer^.Finish;
  end;
should be in graphicsWindow menu *)
end;


procedure    THermesMainWindow.CMFilePrinterSetup(var Msg : TMessage);
begin
  {Printer^.Configure; }
end;


procedure    THermesMainWindow.CMFilePrinterChoose(var Msg : TMessage);
begin
  {Printer^.Choose; }
end;


procedure    THermesMainWindow.CMFileSaveAs (var Msg : TMessage);
begin

end;


procedure    THermesMainWindow.CMFileCommit (var Msg : TMessage);
begin
  DBCloseDatabase;     {Create new backup files and dispose memory collections}
  DBOpenDatabase;
end;


    {*** Edit menu ***}

procedure    THermesMainWindow.CMEditName(var Msg : TMessage);
begin

end;


    {*** Controls menu ***}

procedure    THermesMainWindow.CMControlsSelectContext(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PSelectWin, Init(@Self)));
end;


procedure    THermesMainWindow.CMControlsPerspective(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PContextsWin, Init(@Self)));
end;


procedure    THermesMainWindow.CMControlsNavigator(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PNavigatorWin, Init(@Self, DBIdOf('one', 'C'))));
end;


procedure    THermesMainWindow.CMControlsObjects(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PObjectsWin, Init(@Self)));
end;


procedure    THermesMainWindow.CMControlsQueries(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PQueriesWin, Init(@Self)));
end;


procedure    THermesMainWindow.CMControlsEntryForm(var Msg : TMessage);
begin
  Application^.ExecDialog(New(PEntryFormWin, Init(@Self)));
end;
```

```
procedure    THermesMainWindow.CMControlsQueryResults(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PResultWin, Init(@Self, nil)));
end;


procedure    THermesMainWindow.CMControlsVCopy(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PVCopyWin, Init(@Self)));
end;


procedure    THermesMainWindow.CMControlsAdd(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PTextWin, Init(@Self, 0)));
end;


procedure    THermesMainWindow.CMControlsModify(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PModifyWin,  Init(@Self, DBIdOf('two', ot_Character))));
end;

procedure    THermesMainWindow.CMControlsLink(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PLinksWin, Init(@Self)));
end;


    {*** Contexts menu ***}

procedure   THermesMainWindow.CMContextsNew(var Msg : TMessage);
{create a new Context}
var
  EditText         : array[0..41] of Char;
  TheContext       : PContext;
  TheParentContext : PContext;
begin
  StrCopy(EditText, 'new context name');
  if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                'Define New Context', 'Enter the new context name: ',
                EditText, SizeOf(EditText)))) = id_OK) then
    begin
      StrCopy(EditText, StripBlanks(EditText));
      TheContext := PContext(DBGetName(EditText, ot_Context));
      if assigned(TheContext)
        then ErrorMsg(et_WARNING, 'A context with this name already exists.')
        else
          begin
            TheContext := New(PContext, Init(EditText));
            StrCopy(EditText, '                                        ');
            if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                'Define New Context', 'Enter the context password: ',
                EditText, SizeOf(EditText)))) = id_OK) then
              begin
                StrCopy(EditText, StripBlanks(EditText));
                TheContext^.SetPassword(EditText);
                if (MessageBox(0, 'Allow virtual editing of contents of this context?',
                'Define New Context', mb_YesNoCancel)= id_No)
                  then TheContext^.SetVEdit(False);

                if (MessageBox(0, 'Really create this context?', 'Define New Context',
                               mb_YesNoCancel) = id_Yes)
                  then DBPutName(TheContext);

                StrCopy(EditText, '                                        ');
                while (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                                 'Define New Context', 'Enter name of a parent context, or cancel: ',
                                 EditText, SizeOf(EditText)))) = id_OK) do
                  begin
                    StrCopy(EditText, StripBlanks(EditText));
                    TheParentContext := PContext(DBGetName(EditText, ot_Context));
                    if assigned(TheParentContext)then
                      if PrivsCreateChildContext(TheParentContext^.GetId)
                        then CopyContextNames(TheParentContext^.GetName, TheContext^.GetName)
```

```
                        else ErrorMsg(et_WARNING, 'You do not have permission to inherit this parent
context.');
                  end;
              end;
          end;
    end;
end;

procedure   THermesMainWindow.CMContextsModify(var Msg : TMessage);
{modify a Context}
var
  EditText         : array[0..41] of Char;
  TheContext  : PContext;
  TheParentContext : PContext;
begin
  StrCopy(EditText, 'old context name');
  if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                'Modify a Context', 'Enter the old context name: ',
                EditText, SizeOf(EditText)))) = id_OK) then
    begin
      StrCopy(EditText, StripBlanks(EditText));
      TheContext := PContext(DBGetName(EditText, ot_Context));
      if not assigned(TheContext)
        then ErrorMsg(et_WARNING, 'No context with this name exists.')
        else
          begin
            StrCopy(EditText, '                                        ');
            if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                'Modify a Context', 'Enter the old context password: ',
                EditText, SizeOf(EditText)))) = id_OK) then
              begin
                StrCopy(EditText, StripBlanks(EditText));
                if (StrComp(EditText, TheContext^.GetPassword) = 0) then
                  begin
                    StrCopy(EditText, '                                        ');
                    if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                           'Modify a Context', 'Enter a new context password, or cancel: ',
                           EditText, SizeOf(EditText)))) = id_OK)
                      then
                        begin
                          StrCopy(EditText, StripBlanks(EditText));
                          TheContext^.SetPassword(EditText);
                        end;
                    if (MessageBox(0, 'Allow virtual editing of contents of this context?',
                           'Modify a Context', mb_YesNoCancel)= id_No)
                      then TheContext^.SetVEdit(False);

                    if (MessageBox(0, 'Really modify this context?', 'Modify a Context',
                                   mb_YesNoCancel) = id_Yes)
                      then DBPutName(TheContext);

                    StrCopy(EditText, '                                        ');
                    while (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                                     'Modify a Context', 'Enter name of a new parent context, or
cancel: ',
                                     EditText, SizeOf(EditText)))) = id_OK) do
                      begin
                        StrCopy(EditText, StripBlanks(EditText));
                        TheParentContext := PContext(DBGetName(EditText, ot_Context));
                        if assigned(TheParentContext)then
                          if PrivsCreateChildContext(TheParentContext^.GetId)
                            then CopyContextNames(TheParentContext^.GetName, TheContext^.GetName)
                            else ErrorMsg(et_WARNING, 'You do not have permission to inherit this
parent context.');
                      end;
                  end;
              end;
          end;
    end;
end;

procedure   THermesMainWindow.CMContextsSelect(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PSelectWin, Init(@Self)));
end;
```

```
procedure   THermesMainWindow.CMContextsDisplay(var Msg : TMessage);
begin
  Application^.MakeWindow(New(PContextsWin, Init(@Self)));
end;



    {*** Logins menu ***}

procedure   THermesMainWindow.CMLoginsNew(var Msg : TMessage);
{create a new Login}
var
  EditText      : array[0..41] of Char;
  TheLogin : PLogin;
begin
  StrCopy(EditText, 'new login name');
  if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
               'Define New Login', 'Enter the new user name: ',
               EditText, SizeOf(EditText)))) = id_OK) then
    begin
      StrCopy(EditText, StripBlanks(EditText));
      TheLogin := PLogin(DBGetName(EditText, ot_Login));
      if assigned(TheLogin)
        then ErrorMsg(et_WARNING, 'A login with this name already exists.')
        else
          begin
            TheLogin := New(PLogin, Init(EditText));
            StrCopy(EditText, '                                        ');
            if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                'Define New Login', 'Enter the password: ',
                EditText, SizeOf(EditText)))) = id_OK) then
              begin
                StrCopy(EditText, StripBlanks(EditText));
                TheLogin^.SetPassword(EditText);
                if (MessageBox(0, 'Really create this Login?', 'Define New Login', mb_YesNoCancel)
                    = id_Yes)
                  then DBPutName(TheLogin);
              end;
          end;
     end;
end;

procedure   THermesMainWindow.CMLoginsModify(var Msg : TMessage);
{allow a Login password to be changed -- if the old password is entered}
var
  EditText      : array[0..41] of Char;
  TheLogin : PLogin;
begin
  StrCopy(EditText, 'old login name');
  if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
               'Modify a Login', 'Enter the old user name: ',
               EditText, SizeOf(EditText)))) = id_OK) then
    begin
      StrCopy(EditText, StripBlanks(EditText));
      TheLogin := PLogin(DBGetName(EditText, ot_Login));
      if not assigned(TheLogin)
        then ErrorMsg(et_WARNING, 'No login with this name exists.')
        else
          begin
            StrCopy(EditText, 'old login password');
            if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                  'Modify a Login', 'Enter the old password: ',
                  EditText, SizeOf(EditText)))) = id_OK) then
              begin
                StrCopy(EditText, StripBlanks(EditText));
                if (StrComp(EditText, TheLogin^.GetPassword) = 0) then
                  begin
                    StrCopy(EditText, 'new login password');
                    if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                           'Modify a Login', 'Enter the new password: ',
                           EditText, SizeOf(EditText)))) = id_OK) then
                      begin
                        StrCopy(EditText, StripBlanks(EditText));
                        TheLogin^.SetPassword(EditText);
                        if (MessageBox(0, 'Really modify this Login?', 'Modify a Login',
mb_YesNoCancel)
```

```
                          = id_Yes)
                        then DBPutName(TheLogin);
                    end;
                end;
              end;
          end;
    end;
end;

procedure   THermesMainWindow.CMLoginsDelete(var Msg : TMessage);
{allow a Login password to be deleted -- if the old password is entered}
var
  EditText      : array[0..41] of Char;
  TheLogin : PLogin;
begin
  StrCopy(EditText, 'the login name');
  if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
               'Delete a Login', 'Enter the old user name: ',
               EditText, SizeOf(EditText)))) = id_OK) then
    begin
      TheLogin^.SetPassword(EditText);
      TheLogin := PLogin(DBGetName(EditText, ot_Login));
      if not assigned(TheLogin)
        then ErrorMsg(et_WARNING, 'No login with this name exists.')
        else
          begin
            StrCopy(EditText, 'the login password');
            if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                  'Delete a Login', 'Enter the password: ',
                  EditText, SizeOf(EditText)))) = id_OK) then
              begin
                if (StrComp(EditText, TheLogin^.GetPassword) = 0) then
                  if(MessageBox(0, 'Really delete this Login?', 'Delete a Login', mb_YesNoCancel)
                            = id_Yes)
                    then DBDeleteName(TheLogin^.GetName, ot_Login);
              end;
          end;
    end;
end;

procedure   THermesMainWindow.CMLoginsDisplay(var Msg : TMessage);
begin
  ErrorMsg(et_NOTICE, 'You do not have permissions for this display.');
end;


    {*** Nodes menu ***}

procedure   THermesMainWindow.CMNodesView(var Msg : TMessage);
var
  TheIndex : Integer;
  TheNode  : PPersistentObject;
  TheId    : LongInt;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_Character), TheIndex, TheSelection, True);
  if (TheIndex > -1) then
    begin
      TheId := DBIdOf(TheSelection, ot_Character);
      if TheId > 0
        then
          begin
            TheNode := DBGet(TheId);
            if assigned(TheNode)
              then Application^.MakeWindow(New(PNavigatorWin, Init(@Self, TheId)))
              else ErrorMsg(et_NOTICE, 'HerWorld.CMNodesView: The item was not found.');
          end;
    end
  else ErrorMsg(et_NOTICE, 'No item was chosen to view.');
end;

procedure   THermesMainWindow.CMNodesAnnotate(var Msg : TMessage);
var
  TheIndex : Integer;
  TheNode  : PPersistentObject;
```

```
  TheId    : LongInt;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_Character), TheIndex, TheSelection, True);
  if (TheIndex > -1) then
    begin
      TheId := DBIdOf(TheSelection, ot_Character);
      if TheId > 0
        then
          begin
            TheNode := DBGet(TheId);
            if assigned(TheNode)
              then Application^.MakeWindow(New(PNavigatorWin, Init(@Self, TheId)))
              else ErrorMsg(et_NOTICE, 'HerWorld.CMNodesAnnotate: The item was not found.');
          end;
    end
  else ErrorMsg(et_NOTICE, 'No item was chosen to view.');
end;

procedure   THermesMainWindow.CMNodesCreate(var Msg : TMessage);
var
  EditText : array[0..41] of Char;
  TheNode  : PCharacterSimple;
  TheName  : array[0..41] of Char;
begin
  StrCopy(EditText, 'new node name');
  if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                'Define New Character Node', 'Enter the new node name: ',
                EditText, SizeOf(EditText)))) = id_OK) then
    begin
      StrCopy(TheName, StripBlanks(EditText));
      TheNode := PCharacterSimple(DBGetName(TheName, ot_Character));
      if assigned(TheNode)
        then ErrorMsg(et_WARNING, 'A character node with this name already exists.')
        else
          begin
            StrCopy(EditText, '                                        ');
            if (Application^.ExecDialog(New(PInputDialog, Init(@Self,
                'Define New Character Node', 'Enter the text: ',
                EditText, SizeOf(EditText)))) = id_OK) then
              begin
                StrCopy(EditText, StripBlanks(EditText));
                TheNode := New(PCharacterSimple, Init(TheName, EditText));
                if (MessageBox(0, 'Really create this Node?', 'Define New Character Node',
mb_YesNoCancel)
                    = id_Yes)
                  then DBPutNamedObject(TheNode, HermesApp.GetContext);
              end;
          end;
     end;
end;

procedure   THermesMainWindow.CMNodesModify(var Msg : TMessage);
var
  TheIndex : Integer;
  TheNode  : PPersistentObject;
  TheId    : LongInt;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_Character), TheIndex, TheSelection, True);
  if (TheIndex > -1) then
    begin
      TheId := DBIdOf(TheSelection, ot_Character);
      if TheId > 0
        then
          begin
            TheNode := DBGet(TheId);
            if assigned(TheNode)
              then Application^.MakeWindow(New(PModifyWin,  Init(@Self, TheId)))
              else ErrorMsg(et_NOTICE, 'HerWorld.CMNodesModify: The item was not found.');
          end;
    end
  else ErrorMsg(et_NOTICE, 'No item was chosen to view.');
end;
```

```
procedure   THermesMainWindow.CMNodesDelete(var Msg : TMessage);
var
  TheIndex : Integer;
  TheNode  : PPersistentObject;
  TheId    : LongInt;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_Character), TheIndex, TheSelection, True);
  if (TheIndex > -1) then
    begin
      TheId := DBIdOf(TheSelection, ot_Character);
      if TheId > 0
        then
          begin
            TheNode := DBGet(TheId);
            if assigned(TheNode)
              then
                if(MessageBox(0, 'Really delete this Node?', 'Delete a Node',
                                  mb_YesNoCancel) = id_Yes)
                  then DeleteNode(TheId, HermesApp.GetContext);
          end
        else ErrorMsg(et_NOTICE, 'HerWorld.CMNodesDelete: The item was not found.');
    end
  else ErrorMsg(et_NOTICE, 'HerWorld.CMNodesDelete: No item was chosen to view.');
end;

procedure   THermesMainWindow.CMNodesPromote(var Msg : TMessage);
begin
  { This is Task II
    it uses PrivsEditContext
    Rule #3 }
end;



    {*** Window menu ***}


    {*** Help menu ***}

procedure    THermesMainWindow.CMHelp(var Msg : TMessage);
begin

end;


procedure    THermesMainWindow.CMAbout(var Msg : TMessage);
{var
  About: PAboutBox;    }
begin
{  New(About, Init(@Self));
  Application^.MakeWindow(About);
  About^.Show(sw_Normal);
  SetFocus(About^.HWindow);     }
end;


  {************************************
  *
        *    methods for THermesChildWindow
  *
        ************************************}

constructor THermesChildWindow.Init(AParent : PWindowsObject; ATitle : PChar);
begin
  TGWindow.Init(AParent, ATitle);
  TheChildType := CONTROL;     {default}
  Attr.Style := ws_Visible or ws_Overlapped or ws_Caption or ws_SysMenu or ws_ThickFrame
                or ws_MinimizeBox or ws_MaximizeBox or ws_Child;
end;

function    THermesChildWindow.CanClose : Boolean;
begin
  CanClose := true;
```

```
end;

procedure    THermesChildWindow.SetChildType(NewChildType : ChildType);
begin
  TheChildType := NewChildType;
end;

function    THermesChildWindow.GetChildType : ChildType;
begin
  GetChildType := TheChildType;
end;


end. {HerWorld}
```

14. HerCoWin.pas define control windows

```
 {*****************************************
 *
 *      HerCoWin.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 ******************************************}

unit HerCoWin;

interface

uses OWindows, Objects, ODialogs, WinTypes,
     HerLists, HerBasic, HerWorld, HerHyper, HerMedia, HerNodes,
     HerLangu, HerDataB, HerOGL1, HerOGL2, HerOGL3,
     Strings, OStdDlgs, WinProcs, HerPersp;


{ This unit defines ControlWindows. They are HermesChildWindows that are defined as windows with
controls.

       SPECIFICATION:
  ControlWindows stack in the left third of the screen.
  They are windows which can be minimized but not maximized.
  There is a window defined for each Control menu item.  There can be multiple copies.
  They act like non-modal dialogs.  They call modal dialogs to select information from lists, etc.
  They open TextualWindows to display query results, etc. or GraphicalWindows to display graphic
objects.

       DESIGN:
  ControlWindows derive from a ControlWindow, which is a HermesChildWindow (defined in HerWorld).
  They are windows with controls -- not DlgWindows with resources.
}

const

  wm_QueryMsg = wm_User + 1;

  id_LB1L = 1001;    {control for ListerWin dialog}

  id_BN1S = 1150;    {controls for SelectWin}
  id_BN2S = 1155;
  id_ST1S = 1160;
(*
  id_BN1G = 1170;    {controls for GraphicsButtonWin}
  id_BN2G = 1171;
  id_BN3G = 1172;
  id_BN4G = 1173;
  id_ST1G = 1174;
*)
  id_BN1C = 1100;    {controls for ContextsWin}
  id_BN2C = 1101;
  id_ST1C = 1102;
  id_EB1C = 1103;
  id_BN3C = 1104;
  id_BN4C = 1105;
  id_BN5C = 1106;
  id_LB1C = 1107;
  id_BN6C = 1108;
  id_BN7C = 1109;
  id_BN8C = 1110;
  id_BN9C = 1111;

  id_GB1N = 1200;    {controls for NavigatorWin}
  id_RB1N = 1201;
  id_RB2N = 1202;
  id_RB3N = 1203;
  id_RB4N = 1204;
  id_RB5N = 1205;
  id_ST1N = 1210;
  id_LB1N = 1211;
```

```
id_ST2N = 1212;
id_LB2N = 1213;
id_ST0N = 1214;

id_BN1A = 1300;    {controls for AuthorWin}

id_GB1O = 1400;    {controls for ObjectsWin}
id_BN1O = 1410;
id_BN2O = 1411;
id_BN3O = 1412;
id_BN4O = 1413;
id_RB1O = 1420;
id_RB2O = 1421;
id_RB3O = 1422;
id_RB4O = 1423;
id_RB5O = 1424;
id_RB6O = 1425;
id_RB7O = 1426;
id_RB8O = 1427;
id_RB9O = 1428;
id_RB10O = 1429;
id_RB11O = 1430;
id_RB12O = 1431;
id_RB13O = 1432;
id_RB14O = 1433;
id_RB15O = 1434;
id_RB16O = 1435;
id_RB17O = 1436;
id_RB18O = 1437;
id_RB19O = 1438;
id_RB20O = 1439;
id_RB21O = 1440;
id_RB22O = 1441;
id_RB23O = 1442;
id_RB24O = 1443;
id_RB25O = 1444;
id_RB26O = 1445;
id_RB27O = 1446;
id_RB28O = 1447;
id_RB29O = 1448;
id_RB30O = 1449;
id_RB31O = 1450;
id_RB32O = 1451;
id_RB33O = 1452;
id_RB34O = 1453;
id_RB35O = 1454;
id_ST1O  = 1455;
id_ST2O  = 1456;
id_ST3O  = 1457;
id_ST4O  = 1458;
id_BN5O  = 1460;
id_BN6O  = 1461;

id_ST0E = 1517;    {controls for EntryFormWin}
id_BN0E = 1518;
id_ST1E = 1500;
id_EB1E = 1501;
id_ST2E = 1502;
id_BN1E = 1503;
id_GB1E = 1504;
id_RB1E = 1505;
id_BN2E = 1506;
id_RB2E = 1507;
id_BN3E = 1508;
id_ST3E = 1509;
id_BN4E = 1511;
id_RB3E = 1512;
id_BN5E = 1513;
id_ST4E = 1514;
id_BN6E = 1515;
id_ST5E = 1516;
id_BN7E = 1590;
id_BN8E = 1591;

id_BN1Q = 1601;    {controls for QueriesWin}
id_ST1Q = 1631;
```

```
  id_GB1Q = 1606;
  id_RB1Q = 1607;
  id_BN3Q = 1608;
  id_BN4Q = 1610;
  id_RB2Q = 1611;
  id_BN5Q = 1613;
  id_BN6Q = 1614;
  id_BN7Q = 1615;
  id_BN8Q = 1616;
  id_BN9Q = 1617;
  id_BN10Q = 1618;
  id_BN11Q = 1620;
  id_BN12Q = 1622;
  id_BN13Q = 1624;
  id_BN14Q = 1626;
  id_ST3Q  = 1630;
  id_BN16Q = 1646;
  id_ST4Q  = 1647;
  id_BN17Q = 1645;
  id_BN18Q = 1644;

  id_BN1L = 1700;    {controls for LinksWin}
  id_BN2L = 1701;
  id_BN4L = 1703;
  id_BN5L = 1704;
  id_BN6L = 1705;
  id_BN7L = 1706;
  id_ST1L = 1707;
  id_ST2L = 1708;
  id_ST3L = 1709;
  id_ST4L = 1710;
  id_ST5L = 1711;
  id_EB1L = 1712;

  id_BN1M = 1800;    {controls for ModifyWin}
  id_ST1M = 1801;
  id_ST2M = 1802;
  id_EB1M = 1803;
  id_BN2M = 1804;
  id_ST3M = 1805;
  id_BN3M = 1806;
  id_ST4M = 1807;
  id_ST5M = 1808;
  id_EB2M = 1809;
  id_BN4M = 1810;
  id_BN5M = 1811;
  id_EB3M = 1812;

  id_BN1T = 1900;    {controls for TextWin}
  id_ST1T = 1901;
  id_ST2T = 1902;
  id_EB1T = 1903;
  id_BN2T = 1904;
  id_ST3T = 1905;
  id_BN3T = 1906;
  id_ST4T = 1907;
  id_ST5T = 1908;
  id_EB2T = 1909;
  id_BN4T = 1910;
  id_BN5T = 1911;


type

  LBTransfer = record       {transfer record for LISTER dialog}
    Strings: PCollection;
    Selection: Integer;
  end;


  ControlType = (CONTEXTS, NAVIGATOR, AUTHOR, OBJECTTYPE, ENTRYFORM, QUERIES, LINKS, MODIFY,
TEXTNODE, UNDEFINED);


        {*************************************
        *
```

```
        *    data structure for ControlWindow
        *
        ****************************************}

  PControlWindow = ^ControlWindow;

        ControlWindow = object (THermesChildWindow)
    TheControlType : ControlType;

    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    procedure   SetUpWindow; virtual;
        end;


        {****************************************
        *
        *    data structure for SelectWin
        *
        ****************************************}

  PSelectWin = ^SelectWin;

        SelectWin = object (ControlWindow)
    constructor Init(AParent: PWindowsObject);
    procedure   SetUpWindow; virtual;
    procedure   WMSize(var Msg: TMessage); virtual wm_First + wm_Size;
    procedure   IDBN1S(var Msg: TMessage); virtual id_First + id_BN1S;
    procedure   IDBN2S(var Msg: TMessage); virtual id_First + id_BN2S;
 private
    BN1S : PButton;
    BN2S : PButton;
    ST1S : PStatic;
        end;

(*
        {****************************************
        *
        *    data structure for GraphicsButtonWin
        *
        ****************************************}

  PGraphicsButtonWin = ^GraphicsButtonWin;

        GraphicsButtonWin = object (TGWindow)
    constructor Init(AParent: PWindowsObject);
    procedure   SetUpWindow; virtual;
    procedure   AdjustSpace; virtual;
    procedure   IDBN1G(var Msg: TMessage); virtual id_First + id_BN1G;
    procedure   IDBN2G(var Msg: TMessage); virtual id_First + id_BN2G;
    procedure   IDBN3G(var Msg: TMessage); virtual id_First + id_BN3G;
    procedure   IDBN4G(var Msg: TMessage); virtual id_First + id_BN4G;
 private
    BN1G : PButton;
    BN2G : PButton;
    BN3G : PButton;
    BN4G : PButton;
    ST1G : PStatic;
        end;
*)

        {****************************************
        *
        *    data structure for ContextsWin
        *
        ****************************************}

  PContextsWin = ^ContextsWin;

        ContextsWin = object (ControlWindow)
    constructor Init(AParent: PWindowsObject);
    procedure   SetUpWindow; virtual;
    procedure   ListInherited;
    procedure   IDBN1C(var Msg: TMessage); virtual id_First + id_BN1C;
    procedure   IDBN2C(var Msg: TMessage); virtual id_First + id_BN2C;
    procedure   IDBN3C(var Msg: TMessage); virtual id_First + id_BN3C;
    procedure   IDBN4C(var Msg: TMessage); virtual id_First + id_BN4C;
```

```
  procedure   IDBN5C(var Msg: TMessage); virtual id_First + id_BN5C;
  procedure   IDBN6C(var Msg: TMessage); virtual id_First + id_BN6C;
  procedure   IDBN7C(var Msg: TMessage); virtual id_First + id_BN7C;
  procedure   IDBN8C(var Msg: TMessage); virtual id_First + id_BN8C;
  procedure   IDBN9C(var Msg: TMessage); virtual id_First + id_BN9C;
private
  OldContext : StrName;
  NewContext : StrName;
  BN1C : PButton;
  BN2C : PButton;
  ST1C : PStatic;
  EB1C : PEdit;
  BN3C : PButton;
  BN4C : PButton;
  BN5C : PButton;
  LB1C : PListBox;
  BN6C : PButton;
  BN7C : PButton;
  BN8C : PButton;
  BN9C : PButton;
     end;


     {****************************************
     *
     *    data structure for NavigatorWin
     *
     ****************************************}

PNavigatorWin = ^NavigatorWin;

     NavigatorWin = object (ControlWindow)
  constructor Init(AParent: PWindowsObject; ANodeId: LongInt);
  procedure   FillOutLists; virtual;
  procedure   FillInLists; virtual;
  procedure   FillPredicatesList; virtual;
  procedure   SetUpWindow; virtual;
  procedure   IDGB1N(var Msg: TMessage); virtual id_First + id_GB1N;
  procedure   IDLB1N(var Msg: TMessage); virtual id_First + id_LB1N;
  procedure   IDLB2N(var Msg: TMessage); virtual id_First + id_LB2N;
private
  TheNode : PHermesNode;
  ST0N : PGWindow;
  GB1N : PGroupBox;
  RB1N : PRadioButton;
  RB2N : PRadioButton;
  RB3N : PRadioButton;
  RB4N : PRadioButton;
  RB5N : PRadioButton;
  ST1N : PStatic;
  LB1N : PListBox;
  ST2N : PStatic;
  LB2N : PListBox;
     end;


     {****************************************
     *
     *    data structure for AuthorWin
     *
     ****************************************}

PAuthorWin = ^AuthorWin;

     AuthorWin = object (ControlWindow)
  constructor Init(AParent: PWindowsObject);
     end;


     {****************************************
     *
     *    data structure for ObjectsWin
     *
     ****************************************}

PObjectsWin = ^ObjectsWin;
```

```
      ObjectsWin = object (ControlWindow)
    constructor Init(AParent: PWindowsObject);
    procedure   SetUpWindow; virtual;
    function    GetClass: Char; virtual;
    procedure   IDBN1O(var Msg: TMessage); virtual id_First + id_BN1O;
    procedure   IDBN2O(var Msg: TMessage); virtual id_First + id_BN2O;
    procedure   IDBN3O(var Msg: TMessage); virtual id_First + id_BN3O;
    procedure   IDBN4O(var Msg: TMessage); virtual id_First + id_BN4O;
    procedure   IDBN5O(var Msg: TMessage); virtual id_First + id_BN5O;
    procedure   IDBN6O(var Msg: TMessage); virtual id_First + id_BN6O;
    procedure   CreateType; virtual;
    procedure   CreateKind; virtual;
    procedure   CreatePlural(NewName: PChar; ObjectType: ObjectTypes); virtual;
  private
    GB1O : PGroupBox;
    BN1O : PButton;
    BN2O : PButton;
    BN3O : PButton;
    BN4O : PButton;
    BN5O : PButton;
    BN6O : PButton;
    ST1O : PStatic;
    ST2O : PStatic;
    ST3O : PStatic;
    ST4O : PStatic;
    RB1O : PRadioButton;
    RB2O : PRadioButton;
    RB3O : PRadioButton;
    RB4O : PRadioButton;
    RB5O : PRadioButton;
    RB6O : PRadioButton;
    RB7O : PRadioButton;
    RB8O : PRadioButton;
    RB9O : PRadioButton;
    RB10O : PRadioButton;
    RB11O : PRadioButton;
    RB12O : PRadioButton;
    RB13O : PRadioButton;
    RB14O : PRadioButton;
    RB15O : PRadioButton;
    RB16O : PRadioButton;
    RB17O : PRadioButton;
    RB18O : PRadioButton;
    RB19O : PRadioButton;
    RB20O : PRadioButton;
    RB21O : PRadioButton;
    RB22O : PRadioButton;
    RB23O : PRadioButton;
    RB24O : PRadioButton;
    RB25O : PRadioButton;
    RB26O : PRadioButton;
    RB27O : PRadioButton;
    RB28O : PRadioButton;
    RB29O : PRadioButton;
    RB30O : PRadioButton;
    RB31O : PRadioButton;
    RB32O : PRadioButton;
    RB33O : PRadioButton;
    RB34O : PRadioButton;
    RB35O : PRadioButton;
      end;


      {****************************************
      *
      *    data structure for EntryFormWin
      *
      ****************************************}

  PEntryFormWin = ^EntryFormWin;

      EntryFormWin = object (TDlgWindow)
    constructor Init(AParent: PWindowsObject);
    procedure   SetUpWindow; virtual;
    procedure   WMQueryMsg(var Msg: TMessage); virtual wm_First + wm_QueryMsg;
```

```
  procedure   IDBN0E(var Msg: TMessage); virtual id_First + id_BN0E;
  procedure   IDBN1E(var Msg: TMessage); virtual id_First + id_BN1E;
  procedure   IDBN2E(var Msg: TMessage); virtual id_First + id_BN2E;
  procedure   IDBN3E(var Msg: TMessage); virtual id_First + id_BN3E;
  procedure   IDBN4E(var Msg: TMessage); virtual id_First + id_BN4E;
  procedure   IDBN5E(var Msg: TMessage); virtual id_First + id_BN5E;
  procedure   IDBN6E(var Msg: TMessage); virtual id_First + id_BN6E;
  procedure   IDBN7E(var Msg: TMessage); virtual id_First + id_BN7E;
  procedure   IDBN8E(var Msg: TMessage); virtual id_First + id_BN8E;
private
  TheType : PChar;
  TheEF1  : PNodeContent;
  TheEF2  : PNodeContent;
  TheEF3  : PNodeContent;
  TheEF4  : PNodeContent;
  NewEF   : PNodeContent;
  LastBN  : Integer;
  ST0E : PStatic;
  BN0E : PButton;
  ST1E : PStatic;
  EB1E : PEdit;
  ST2E : PStatic;
  BN1E : PButton;
  GB1E : PGroupBox;
  RB1E : PRadioButton;
  BN2E : PButton;
  RB2E : PRadioButton;
  BN3E : PButton;
  ST3E : PStatic;
  BN4E : PButton;
  RB3E : PRadioButton;
  BN5E : PButton;
  ST4E : PStatic;
  BN6E : PButton;
  ST5E : PStatic;
  BN7E : PButton;
  BN8E : PButton;
  BN9E : PButton;
     end;


     {****************************************
     *
     *    data structure for QueriesWin
     *
     ****************************************}

PQueriesWin = ^QueriesWin;

     QueriesWin = object (TDlgWindow)
  constructor Init(AParent: PWindowsObject);
  procedure   SetUpWindow; virtual;
  procedure   WMQueryMsg(var Msg: TMessage); virtual wm_First + wm_QueryMsg;
  procedure   IDBN1Q(var Msg: TMessage); virtual id_First + id_BN1Q;
  procedure   IDBN3Q(var Msg: TMessage); virtual id_First + id_BN3Q;
  procedure   IDBN4Q(var Msg: TMessage); virtual id_First + id_BN4Q;
  procedure   IDBN5Q(var Msg: TMessage); virtual id_First + id_BN5Q;
  procedure   IDBN6Q(var Msg: TMessage); virtual id_First + id_BN6Q;
  procedure   IDBN7Q(var Msg: TMessage); virtual id_First + id_BN7Q;
  procedure   IDBN8Q(var Msg: TMessage); virtual id_First + id_BN8Q;
  procedure   IDBN9Q(var Msg: TMessage); virtual id_First + id_BN9Q;
  procedure   IDBN10Q(var Msg: TMessage); virtual id_First + id_BN10Q;
  procedure   IDBN11Q(var Msg: TMessage); virtual id_First + id_BN11Q;
  procedure   IDBN12Q(var Msg: TMessage); virtual id_First + id_BN12Q;
  procedure   IDBN13Q(var Msg: TMessage); virtual id_First + id_BN13Q;
  procedure   IDBN14Q(var Msg: TMessage); virtual id_First + id_BN14Q;
  procedure   IDBN16Q(var Msg: TMessage); virtual id_First + id_BN16Q;
  procedure   IDBN17Q(var Msg: TMessage); virtual id_First + id_BN17Q;
  procedure   IDBN18Q(var Msg: TMessage); virtual id_First + id_BN18Q;
 private
  TheEntryForm      : PNodeContent;
  TheNode           : LongInt;
  TheRelationship1  : PNodeContent;
  TheRelationship2  : PNodeContent;
  TheSubject        : PNodeContent;
  TheFilter         : PNodeContent;
```

```
  TheHermeneutic    : PNodeContent;
  TheBoolean        : PBooleans;
  TheQuery          : PNodeContent;
  NewQ      : PDataList;
  LastBN    : Integer;
  BN1Q : PButton;
  ST1Q : PEdit;
  GB1Q : PGroupBox;
  RB1Q : PRadioButton;
  BN3Q : PButton;
  BN4Q : PButton;
  RB2Q : PRadioButton;
  BN5Q : PButton;
  BN6Q : PButton;
  BN7Q : PButton;
  BN8Q : PButton;
  BN9Q : PButton;
  BN10Q : PButton;
  BN11Q : PButton;
  BN12Q : PButton;
  BN13Q : PButton;
  BN14Q : PButton;
  ST3Q  : PStatic;
  BN16Q : PButton;
  ST4Q  : PStatic;
  BN17Q : PButton;
  BN18Q : PButton;
      end;


      {****************************************
      *
      *    data structure for LinksWin
      *
      ****************************************}

PLinksWin = ^LinksWin;

      LinksWin = object (ControlWindow)
  constructor Init(AParent: PWindowsObject);
  procedure   SetUpWindow; virtual;
  procedure   IDBN1L(var Msg: TMessage); virtual id_First + id_BN1L;
  procedure   IDBN2L(var Msg: TMessage); virtual id_First + id_BN2L;
  procedure   IDBN4L(var Msg: TMessage); virtual id_First + id_BN4L;
  procedure   IDBN5L(var Msg: TMessage); virtual id_First + id_BN5L;
  procedure   IDBN6L(var Msg: TMessage); virtual id_First + id_BN6L;
  procedure   IDBN7L(var Msg: TMessage); virtual id_First + id_BN7L;
private
BN1L : PButton;    {controls for LinksWin}
BN2L : PButton;
BN4L : PButton;
BN5L : PButton;
BN6L : PButton;
BN7L : PButton;
ST1L : PStatic;
ST2L : PStatic;
ST3L : PStatic;
ST4L : PStatic;
ST5L : PStatic;
EB1L : PEdit;
      end;


      {****************************************
      *
      *    data structure for ModifyWin
      *
      ****************************************}

PModifyWin = ^ModifyWin;

      ModifyWin = object (ControlWindow)
  constructor Init(AParent: PWindowsObject; ANodeId: LongInt);
  destructor  Done; virtual;
  procedure   SetUpWindow; virtual;
  procedure   SetUpContent; virtual;
```

```
  procedure   IDBN4M(var Msg: TMessage); virtual id_First + id_BN4M;
  procedure   IDBN5M(var Msg: TMessage); virtual id_First + id_BN5M;
private
  TheNode    : PHermesNode;
  TheContent : PCharacter;
  TheList    : PHermesList;
  TheIndex   : Integer;
  EB2M : PEdit;
  EB3M : PEdit;
  BN4M : PButton;
  BN5M : PButton;
     end;


     {****************************************
     *
     *    data structure for TextWin
     *
     ****************************************}

PTextWin = ^TextWin;

     TextWin = object (ControlWindow)
  constructor Init(AParent: PWindowsObject; ALinkedNodeId: LongInt);
  procedure   SetUpWindow; virtual;
  procedure   IDBN1T(var Msg: TMessage); virtual id_First + id_BN1T;
  procedure   IDBN2T(var Msg: TMessage); virtual id_First + id_BN2T;
  procedure   IDBN3T(var Msg: TMessage); virtual id_First + id_BN3T;
  procedure   IDBN4T(var Msg: TMessage); virtual id_First + id_BN4T;
  procedure   IDBN5T(var Msg: TMessage); virtual id_First + id_BN5T;
private
  TheLinkedNode : PHermesNode;
  BN1T : PButton;
  ST1T : PStatic;
  ST2T : PStatic;
  EB1T : PEdit;
  BN2T : PButton;
  ST3T : PStatic;
  BN3T : PButton;
  ST4T : PStatic;
  ST5T : PStatic;
  EB2T : PEdit;
  BN4T : PButton;
  BN5T : PButton;
     end;


     {****************************************
     *
     *    data structure for NotifyWin
     *
     ****************************************}

PNotifyWin = ^NotifyWin;

     NotifyWin = object (ControlWindow)
  constructor Init(AParent: PWindowsObject; ATitle: PChar; AMessage: PChar);
     end;


     {****************************************
     *
     *    data structure for ListerWin
     *
     ****************************************}

PListerWin = ^ListerWin;

     ListerWin = object (TDlgWindow)
  constructor Init(AParent: PWindowsObject; Sorted: Boolean);
  procedure   IDLB1L(var Msg: TMessage); virtual id_First + id_LB1L;
private
  LB1L : PListBox;
     end;
```

```
{interface procedures:}

procedure FillLister(AParent: PWindowsObject; AList: PHermesList; var AnIndex: Integer;
                     var ASelection: PChar; Sorted: Boolean);
           {transfer AList and call LISTER Dialog; return ASelection and AnIndex}



{=====================================================================}
implementation

uses HerLinks, HerTxWin, HerPrivs;

procedure FillLister(AParent: PWindowsObject; AList: PHermesList; var AnIndex: Integer;
                                                 var ASelection: PChar; Sorted: Boolean);
{transfer AList and call LISTER Dialog; return ASelection and AnIndex}
var
  TheDlg    : PListerWin;
  PCharBuf  : array[0..120] of Char;
  LBBuffer  : LBTransfer;
      procedure TransferOne(Item: PChar); far;
      begin  {TransferOne}
        LBBuffer.Strings^.Insert(StrNew(Item));
      end;   {TransferOne}
begin  {FillLister}
  ASelection := nil;
  TheDlg := New(PListerWin, Init(AParent, Sorted));
  LBBuffer.Strings := New(PCollection, Init(5, 5));
  AList^.ForEach(@TransferOne);
  LBBuffer.Selection := -1;
  TheDlg^.TransferBuffer := @LBBuffer;

  if (Application^.ExecDialog(TheDlg) <> id_Cancel) and (LBBuffer.Selection > -1)
    then StrCopy(PCharBuf, LBBuffer.Strings^.At(LBBuffer.Selection))
    else StrCopy(PCharBuf, '');
  ASelection := StrNew(PCharBuf);
  AnIndex := LBBuffer.Selection;
end;   {FillLister}


       {****************************************
       *
       *       methods for ControlWindow
       *
       ****************************************}

constructor ControlWindow.Init(AParent: PWindowsObject; ATitle: PChar);
var
  ARect  : TRect;
  AppIcon : TIcon;
  AThird : Integer;
  TheChildType : ChildType;
begin
  THermesChildWindow.Init(AParent, ATitle);
  TheChildType := CONTROL;
  TheControlType := UNDEFINED;
  Attr.Style := ws_Visible or ws_Overlapped or ws_Caption or ws_SysMenu or ws_ThickFrame
                or ws_MinimizeBox or ws_Child and (not ws_MaximizeBox);
  GetClientRect(Application^.MainWindow^.HWindow, ARect);
  AThird :=  (ARect.right - ARect.left) div 3;
  Attr.X := ARect.Left + 1;
  Attr.Y := ARect.top + 1;
  Attr.H := ARect.Bottom - ARect.Top - 76;
  Attr.W := AThird - 60;
  AppIcon.Init(0,0, 'CONTROL_WINDOW');
  SetIcon(@AppIcon);
  AppIcon.Done;

                                     {get count of ControlWindows}
                                     {add offset for stack of ControlWindows}
end;

procedure ControlWindow.SetUpWindow;
begin
  THermesChildWindow.SetUpWindow;
```

```
  Attr.Style := ws_Visible or ws_Overlapped or ws_Caption or ws_SysMenu or ws_ThickFrame
                or ws_MinimizeBox or ws_Child and (not ws_MaximizeBox);
end;


       {***************************************
       *
       *       methods for SelectWin
       *
       ***************************************}

constructor SelectWin.Init(AParent: PWindowsObject);
var
  ARect  : TRect;
  AppIcon: TIcon;
begin
  ControlWindow.Init(AParent, 'Select Context');
  TheControlType := CONTEXTS;
  GetClientRect(Application^.MainWindow^.HWindow, ARect);
  AppIcon.Init(0,0, 'SELECT_CONTEXT');
  SetIcon(@AppIcon);
  AppIcon.Done;
  with Attr do
    begin
      X := ARect.Right - 230;
      Y := ARect.Top + 1;
      H := 70;
      W := 229;
      BN1S := New(PButton, Init(@Self, id_BN1S, 'Select Context', 10, 4, 110, 18, True));
      { BN2S := New(PButton, Init(@Self, id_BN2S, 'Critique', 130, 4, 80, 18, True)); }
      ST1S := New(PStatic, Init(@Self, id_ST1S, '', 5, 25 , 200, 20, 40));
    end;
end;

procedure SelectWin.SetUpWindow;
begin  {SetUpWindow}
  ControlWindow.SetUpWindow;
  if HermesApp.GetContext > 0
    then ST1S^.SetText(DBNameOf(HermesApp.GetContext))       {insert name of current perspective}
    else ST1S^.SetText('hermes_universal_context');
end;   {SetUpWindow}

procedure SelectWin.WMSize(var Msg: TMessage);
begin  {SetUpWindow}
  ControlWindow.WMSize(Msg);
  if HermesApp.GetContext > 0
    then ST1S^.SetText(DBNameOf(HermesApp.GetContext))       {revise name of current perspective}
    else ST1S^.SetText('hermes_universal_context');
end;   {SetUpWindow}

procedure SelectWin.IDBN1S(var Msg: TMessage);         {Select context}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DisplayContextTree, TheIndex, TheSelection, False);
  if (TheIndex > -1)
    then
      begin
        TheSelection := StripBlanks(TheSelection);
        ST1S^.SetText(TheSelection);
        HermesApp.SetContext(DBIdOf(TheSelection, ot_Context));
      end;
end;

procedure SelectWin.IDBN2S(var Msg: TMessage);         {Critique}
begin
end;
(*
       {***************************************
       *
       *       methods for GraphicsButtonWin
       *
       ***************************************}
```

```
constructor GraphicsButtonWin.Init(AParent: PWindowsObject);
begin
  TGWindow.Init(AParent, 'Graphics Buttons');
  Attr.Style := Attr.Style or ws_Child or ws_Border;

  BN1G := New(PButton, Init(@Self, id_BN1G, 'Save Drawing', 5, 3, 110, 18, True));
  BN2G := New(PButton, Init(@Self, id_BN2G, 'Rationale:', 125, 3, 110, 18, False));
  BN3G := New(PButton, Init(@Self, id_BN3G, 'Critique', 245, 3, 110, 18, True));
  BN4G := New(PButton, Init(@Self, id_BN4G, 'Context:', 5, 25, 110, 18, False));
  ST1G := New(PStatic, Init(@Self, id_ST1G, '', 125, 25, 400, 18, 40));
end;

procedure GraphicsButtonWin.SetUpWindow;
begin  {SetUpWindow}
  TGWindow.SetUpWindow;
  if HermesApp.GetContext > 0
    then ST1G^.SetText(DBNameOf(HermesApp.GetContext))       {insert name of current perspective}
    else ST1G^.SetText('hermes_universal_context');
end;   {SetUpWindow}

procedure GraphicsButtonWin.AdjustSpace;
var
  CRect: TMathRect;
begin
  CRect.InitDefault;
  GetDisplayRect(CRect);
  Space^.SetAspectLock(gs_Unlocked);
  Space^.Recalculate;
  Space^.MappingRect^.SetTop(Space^.WorldRect^.Top);
  Space^.MappingRect^.SetBottom(Space^.WorldRect^.Bottom);
  Space^.DisplayRect^.SetTop(CRect.Top);
  Space^.DisplayRect^.SetBottom(CRect.Bottom);
  CRect.Done;
end;

procedure GraphicsButtonWin.IDBN1G(var Msg: TMessage);         {Save Drawing button}
begin
  PGraphicsWindow(Parent)^.CMSaveDrawing(Msg);
end;

procedure GraphicsButtonWin.IDBN2G(var Msg: TMessage);         {Rationale button}
var
  TheId     : LongInt;
  TheParent : PGraphicsWindow;
begin
  TheParent := PGraphicsWindow(Parent);
  if (StrLen(TheParent^.DrawingName) = 0)
    then TheParent^.CMSaveDrawing(Msg);

  if (StrLen(TheParent^.DrawingName) <> 0)
    then
      begin
        TheId := DBIdOf(TheParent^.DrawingName, 'W');
        Application^.MakeWindow(New(PNavigatorWin, Init(Application^.MainWindow, TheId)));
      end;
end;

procedure GraphicsButtonWin.IDBN3G(var Msg: TMessage);         {Critique button}
var
  TheId     : LongInt;
  TheParent : PGraphicsWindow;
begin
  TheParent := PGraphicsWindow(Parent);
  if (StrLen(TheParent^.DrawingName) = 0)
    then TheParent^.CMSaveDrawing(Msg);

  if (StrLen(TheParent^.DrawingName) <> 0)
    then
      begin
        TheId := DBIdOf(TheParent^.DrawingName, 'W');
        Application^.MakeWindow(New(PCritiqueWin, Init(Application^.MainWindow, TheId)));
      end;
end;

procedure GraphicsButtonWin.IDBN4G(var Msg: TMessage);         {Context button}
var
```

```
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DisplayContextTree, TheIndex, TheSelection, False);
  if (TheIndex > -1)
    then
      begin
        TheSelection := StripBlanks(TheSelection);
        ST1G^.SetText(TheSelection);
        HermesApp.SetContext(DBIdOf(TheSelection, ot_Context));
      end;
end;
*)

        {****************************************
        *
        *       methods for ContextsWin
        *
        ****************************************}

constructor ContextsWin.Init(AParent: PWindowsObject);
begin
  ControlWindow.Init(AParent, 'Contexts Control Panel');
  TheControlType := CONTEXTS;
  with Attr do
    begin
      BN1C := New(PButton, Init(@Self, id_BN1C, 'Select from contexts list', 20, 10, W-50, 20,
False));
      BN2C := New(PButton, Init(@Self, id_BN2C, 'Select from contexts tree', 20, 35, W-50, 20,
False));
      ST1C := New(PStatic, Init(@Self, id_ST1C, 'New or selected context:', 5, 65 , W-20, 20, 0));
      EB1C := New(PEdit, Init(@Self, id_EB1C, '', 5, 85, W-20, 25, 40, False));
      BN3C := New(PButton, Init(@Self, id_BN3C, 'Save as new context', 20, 120, W-50, 20, False));
      BN4C := New(PButton, Init(@Self, id_BN4C, 'Use as current context', 20, 150, W-50, 20, False));
      BN5C := New(PButton, Init(@Self, id_BN5C, 'List inherited contexts', 20, 180, W-50, 20,
False));
      LB1C := New(PListBox, Init(@Self, id_LB1C, 5, 205, W-20, 230));
      LB1C^.Attr.Style := ws_Child or lbs_Notify or ws_Border or ws_Visible or ws_VScroll or
ws_HScroll and (not lbs_Sort);
      BN6C := New(PButton, Init(@Self, id_BN6C, 'Inherit another context', 20, 440, W-50, 20,
False));
      BN7C := New(PButton, Init(@Self, id_BN7C, 'Uninherit a context', 20, 465, W-50, 20, False));
      BN8C := New(PButton, Init(@Self, id_BN8C, 'Accept', 20, 495, (W div 2)-50, 20, False));
      BN9C := New(PButton, Init(@Self, id_BN9C, 'Cancel', (W div 2)+20, 495, (W div 2)-50, 20,
False));
    end;
end;

procedure ContextsWin.SetUpWindow;
var
  ContextId : LongInt;
begin  {SetUpWindow}
  ControlWindow.SetUpWindow;
  ContextId := HermesApp.GetContext;
  if ContextId > 0
    then StrCopy(OldContext, DBNameOf(ContextId))       {insert name of current perspective in EB1C}
    else StrCopy(OldContext, 'hermes_universal_context');
  StrCopy(NewContext, OldContext);
  EB1C^.SetText(NewContext);
end;   {SetUpWindow}

procedure ContextsWin.IDBN1C(var Msg: TMessage);         {Select from contexts list}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_Context), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then
      begin
        StrCopy(NewContext, TheSelection);
        EB1C^.SetText(NewContext);
      end;
end;
```

```
procedure ContextsWin.IDBN2C(var Msg: TMessage);         {Select from contexts tree}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DisplayContextTree, TheIndex, TheSelection, False);
  if (TheIndex > -1)
    then
      begin
        TheSelection := StripBlanks(TheSelection);
        StrCopy(NewContext, TheSelection);
        EB1C^.SetText(NewContext);
      end;
end;   {IDBN2C}

procedure ContextsWin.IDBN3C(var Msg: TMessage);         {Save as new context}
begin
  EB1C^.GetText(NewContext, 40);                         {read entry in EditBox}
  if (DBGet(DBIdOf(NewContext, ot_Context)) = nil)             {if it is not already a context}
    then CreateContextNode(NewContext)                   {then create new context}
    else ErrorMsg( et_NOTICE, 'The new or selected context is already defined as a context.');
end;

procedure ContextsWin.IDBN4C(var Msg: TMessage);         {Use as current context}
begin
  HermesApp.SetContext(DBIdOf(NewContext, ot_Context));
end;

procedure ContextsWin.ListInherited;                    {List inherited contexts}
    procedure DisplayOne(Item: PChar); far;
    begin  {DisplayOne}
      LB1C^.AddString(Item);
    end;   {DisplayOne}
 begin  {ListInherited}
   LB1C^.ClearList;
   DisplayInheritedContexts(NewContext)^.ForEach(@DisplayOne);       {from HerVCopy}
 end;   {ListInherited}

procedure ContextsWin.IDBN5C(var Msg: TMessage);         {List inherited contexts}
begin
  ListInherited;
end;

procedure ContextsWin.IDBN6C(var Msg: TMessage);         {Inherit another context}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  ListInherited;
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_Context), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then CopyContextNames(TheSelection, NewContext);
  ListInherited;
end;

procedure ContextsWin.IDBN7C(var Msg: TMessage);         {Uninherit a context}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_Context), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then
      begin
        NotImplemented;
      end;
  ListInherited;
end;

procedure ContextsWin.IDBN8C(var Msg: TMessage);         {Accept}
begin
  Done;
```

```
end;

procedure ContextsWin.IDBN9C(var Msg: TMessage);         {Cancel}
begin
  HermesApp.SetContext(DBIdOf(OldContext, ot_Context));
  Done;
end;


       {****************************************
       *
       *       methods for NavigatorWin
       *
       ****************************************}

constructor NavigatorWin.Init(AParent: PWindowsObject; ANodeId: LongInt);
begin
  ControlWindow.Init(Application^.MainWindow, 'Navigating the HyperText');
  TheControlType := NAVIGATOR;

  TheNode := PHermesNode(DBGet(ANodeId));
  if not assigned(TheNode)
    then ErrorMsg(et_WARNING, 'NavigatorWin.Init: The node not found.');
  with Attr do
    begin
      ST0N := New(PGWindow,Init(@Self, ' '));
      ST0N^.Attr.Style := ws_Child or ws_Visible or lbs_Notify or ws_VScroll or ws_HScroll or
ws_Border;
      ST0N^.Attr.X := 5;
      ST0N^.Attr.Y := 5;
      ST0N^.Attr.W := W-20;
      ST0N^.Attr.H := 60;
      ST0N^.SetGCursor(cs_Finger);
      GB1N := New(PGroupBox, Init(@Self, id_GB1N, 'operation', 5, 75, W-20, 130));
      RB1N := New(PRadioButton, Init(@Self, id_RB1N, 'Navigate out-going links', 20, 100, W-40, 20,
GB1N));
      RB2N := New(PRadioButton, Init(@Self, id_RB2N, 'Navigate in-coming links', 20, 120, W-40, 20,
GB1N));
      RB3N := New(PRadioButton, Init(@Self, id_RB3N, 'Edit the text', 20, 140, W-40, 20, GB1N));
      RB4N := New(PRadioButton, Init(@Self, id_RB4N, 'Author or Annotate', 20, 160, W-40, 20, GB1N));
      RB5N := New(PRadioButton, Init(@Self, id_RB5N, 'Cancel', 20, 180, W-40, 20, GB1N));
      ST1N := New(PStatic, Init(@Self, id_ST1N, 'Out-going Links:', 15, 215 , W-20, 20, 0));
      LB1N := New(PListBox, Init(@Self, id_LB1N, 5, 237, W-20, 150));
      LB1N^.Attr.Style := LB1N^.Attr.Style or ws_HScroll;
      ST2N := New(PStatic, Init(@Self, id_ST2N, 'Predicates:', 15, 395 , W-20, 20, 0));
      LB2N := New(PListBox, Init(@Self, id_LB2N, 5, 417, W-20, H-445));
      LB2N^.Attr.Style := LB2N^.Attr.Style or ws_HScroll;
    end;
end;

procedure NavigatorWin.SetUpWindow;
var
  ContextId : LongInt;
  AList     : PHermesList;
begin  {SetUpWindow}
  ControlWindow.SetUpWindow;
  ST0N^.Picture^.Add(PPicture(TheNode^.DisplayContent));
  RB1N^.SetCheck(bf_Checked);
  if assigned(TheNode)
    then
      begin
        FillOutLists;
        FillPredicatesList;
      end;
end;   {SetUpWindow}

procedure NavigatorWin.FillOutLists;
      procedure ListOne(Item: PLinkList); far;
      begin   {ListOne}
        LB1N^.AddString(Item^.GetName);
      end;    {ListOne}
begin   {FillOutLists}
  LB1N^.ClearList;
  TheNode^.GetOutLinks^.ForEach(@ListOne);    {set out-going links and relevant predicates}
  LB1N^.SetSelIndex(-1);
  LB2N^.ClearList;
```

```
end;    {FillOutLists}

procedure NavigatorWin.FillInLists;
      procedure ListOne(Item: PLinkList); far;
      begin   {ListOne}
        LB1N^.AddString(Item^.GetName);
      end;    {ListOne}
begin
  LB1N^.ClearList;
  TheNode^.GetInLinks^.ForEach(@ListOne);    {set in-coming links}
  LB1N^.SetSelIndex(-1);
  LB2N^.ClearList;
end;


procedure NavigatorWin.FillPredicatesList;
      function IncludeE(AnEntryFormName: PChar): Boolean;

          function IncludesOne(ALinkList : PLinkList):Boolean;
          var
            ATypeId  : LongInt;
            MainName : PChar;

              function IsIncluded(AType: PChar; AnEntryForm: PNodeContent): Boolean;
              var
                NewEntryForm : PAssociation;
                Ok           : Boolean;
              begin  {IsIncluded}
                Ok := False;
                IsIncluded := False;
                if ((AnEntryForm <> nil) and (ATypeId > 0))            {if EF & Type both found in
context}
                  then
                    begin
                      if (TypeOf(AnEntryForm^) = TypeOf(TAssociation))
                        then
                          begin                                        {check if Types match}
                            Ok := (DBGetSynonymId(AnEntryForm^.GetText, MainName, ot_LinkType) =
ATypeId);
                            if not Ok
                              then
                                begin                                   {check if TheType is really
an EntryForm}
                                  NewEntryForm := PAssociation(DBGetNamedObject
                                                  (AnEntryForm^.GetText, ot_Association,
HermesApp.GetContext));
                                  if (NewEntryForm <> nil)              {if TheType is another
EntryForm, not just a Type}
                                    then Ok := IsIncluded(AType, NewEntryForm);
                                end;
                            IsIncluded := Ok;
                          end

                      else if (TypeOf(AnEntryForm^) = TypeOf(TAssociationComb))
                        then IsIncluded := (IsIncluded(AType, AnEntryForm^.GetParam1)
                                            or IsIncluded(AType, AnEntryForm^.GetParam2))

                      else if (TypeOf(AnEntryForm^) = TypeOf(TAssociationWith))
                        then IsIncluded := IsIncluded(AType, AnEntryForm^.GetParam1)

                      else IsIncluded := False;
                    end;
              end;   {IsIncluded}

          var
            AType : PChar;
          begin  {IncludesOne}
            AType := ALinkList^.GetName;
            ATypeId := DBGetSynonymId(AType, MainName, ot_LinkType);
            IncludesOne := IsIncluded(AType,
                     PAssociation(DBGetNamedObject(AnEntryFormName, ot_Association,
HermesApp.GetContext)));
          end;   {IncludesOne}

      var
        FirstItem : PChar;
```

```
      begin  {IncludeE}
        FirstItem := TheNode^.GetOutLinks^.FirstThat(@IncludesOne);
        IncludeE := (FirstItem <> nil);    {if this EntryForm is relevant to some link Type of
TheNode}
      end;   {IncludeE}

      function IncludeP(AnItem: PChar): Boolean;
      begin  {IncludeP}
        IncludeP := False;           {*** change this when Predicates are defined ***}
      end;   {IncludeP}

      procedure ListOneE(AnEntryForm: PChar); far;
      begin   {ListOneE}
        if IncludeE(AnEntryForm)           {if EntryForm is relevant, list it}
          then LB2N^.AddString(AnEntryForm);
      end;    {ListOneE}

      procedure ListOneP(Item: PChar); far;
      begin   {ListOneP}
        if IncludeP(Item)                  {if Predicate is relevant, list it}
          then LB2N^.AddString(Item);
      end;    {ListOneP}
begin
  LB2N^.ClearList;
  DBListNames(ot_Association)^.ForEach(@ListOneE);    {set Association links}
  DBListNames(ot_Predicate)^.ForEach(@ListOneP);      {set Predicate links}
  LB2N^.SetSelIndex(-1);
end;

procedure   NavigatorWin.IDGB1N(var Msg: TMessage);        {radio button pressed}
var
  TheDlg : PControlWindow;
begin
  if      (RB1N^.GetCheck = bf_Checked) then                                    {navigate out-going
links}
    begin
      FillOutLists;
      FillPredicatesList;
      ST1N^.SetText('Out-going Links');
      ST2N^.SetText('Predicates');
    end

  else if (RB2N^.GetCheck = bf_Checked) then                                    {navigate in-coming
links}
    begin
      FillInLists;
      ST1N^.SetText('In-coming Links');
      ST2N^.SetText(' ');
    end

  else if (RB3N^.GetCheck = bf_Checked)
    then Application^.MakeWindow(New(PModifyWin, Init(Application^.MainWindow, TheNode^.GetId)))
{Edit the text node}

  else if (RB4N^.GetCheck = bf_Checked)
    then Application^.MakeWindow(New(PTextWin, Init(Application^.MainWindow, TheNode^.GetId)))
{Add a new text node}

  else if (RB5N^.GetCheck = bf_Checked)
    then if CanClose                                       {Close}
      then Destroy;
end;

procedure   NavigatorWin.IDLB1N(var Msg: TMessage);        {Links}
var
  TheSelection : StrName;
  TheDataList  : PDataListOf;
  TheName      : array[0..91] of Char;
begin
  if (Msg.lParamHi = lbn_SelChange) or (Msg.lParamHi = lbn_DblClk)
    then
      begin
        LB2N^.SetSelIndex(-1);
        LB1N^.GetSelString(TheSelection, 40);
        StrCopy(TheName, TheSelection);
        StrCat(TheName, ' of ');
```

```
        StrCat(TheName, TheNode^.GetName);
        TheDataList := New(PDataListOf, Init(TheName, New(PAssociationType, Init(NoName,
TheSelection)),
                                          New(PDataListId, Init(NoName, TheNode^.GetId))));
        Application^.MakeWindow(New(PResultWin, Init(Application^.MainWindow, TheDataList)));
      end;                                {open window with Query: TheSelection of TheNode}
end;

procedure   NavigatorWin.IDLB2N(var Msg: TMessage);         {Predicates}
var
  TheSelection : StrName;
  TheDataList  : PDataListOf;
  TheEntryForm : PAssociation;
begin
  if (Msg.lParamHi = lbn_SelChange) or (Msg.lParamHi = lbn_DblClk)
    then
      begin
        LB1N^.SetSelIndex(-1);
        LB2N^.GetSelString(TheSelection, 40);
        TheEntryForm := PAssociation(DBGetNamedObject(TheSelection, ot_Association,
HermesApp.GetContext));
        TheDataList := New(PDataListOf, Init(NoName, TheEntryForm,
                                          New(PDataListId, Init(NoName, TheNode^.GetId))));
        Application^.MakeWindow(New(PResultWin, Init(Application^.MainWindow, TheDataList)));
      end;                                                {open window with Query: TheSelection of
TheNode}
end;



        {****************************************
        *
        *       methods for AuthorWin
        *
        ****************************************}

constructor AuthorWin.Init(AParent: PWindowsObject);
begin
  ControlWindow.Init(Application^.MainWindow, 'Control Box for Adding Rationale Entries');
  TheControlType := AUTHOR;
end;


        {****************************************
        *
        *       methods for ObjectsWin
        *
        ****************************************}

constructor ObjectsWin.Init(AParent: PWindowsObject);
begin
  ControlWindow.Init(Application^.MainWindow, 'Control Box for Editing Objects');
  TheControlType := OBJECTTYPE;
  with Attr do
    begin
      GB1O  := New(PGroupBox, Init(@Self, id_GB1O, 'object class', 5, 5, W-20, 450));
      RB1O  := New(PRadioButton, Init(@Self, id_RB1O,  'Character', 20, 30, 100, 20, GB1O));
      RB2O  := New(PRadioButton, Init(@Self, id_RB2O,  'Number', W div 2, 30, 100, 20, GB1O));
      RB3O  := New(PRadioButton, Init(@Self, id_RB3O,  'Booleans', 20, 50, 100, 20, GB1O));
      RB4O  := New(PRadioButton, Init(@Self, id_RB4O,  'Image', W div 2, 50, 100, 20, GB1O));
      RB5O  := New(PRadioButton, Init(@Self, id_RB5O,  'Pen', 20, 70, 100, 20, GB1O));
      RB6O  := New(PRadioButton, Init(@Self, id_RB6O,  'Sound', W div 2, 70, 100, 20, GB1O));
      RB7O  := New(PRadioButton, Init(@Self, id_RB7O,  'Video', 20, 90, 100, 20, GB1O));
      RB8O  := New(PRadioButton, Init(@Self, id_RB8O,  'Animation', W div 2, 90, 100, 20, GB1O));
      RB9O  := New(PRadioButton, Init(@Self, id_RB9O,  'View', 20, 110, 100, 20, GB1O));
      RB10O := New(PRadioButton, Init(@Self, id_RB10O, '3-D Graphic', W div 2, 110, 100, 20, GB1O));
      RB11O := New(PRadioButton, Init(@Self, id_RB11O, 'Polyline', 20, 130, 100, 20, GB1O));
      RB12O := New(PRadioButton, Init(@Self, id_RB12O, 'Cube', W div 2, 130, 100, 20, GB1O));
      RB13O := New(PRadioButton, Init(@Self, id_RB13O, 'Sweep', 20, 150, 100, 20, GB1O));
      RB14O := New(PRadioButton, Init(@Self, id_RB14O, 'Node', W div 2, 150, 100, 20, GB1O));
      RB15O := New(PRadioButton, Init(@Self, id_RB15O, 'Result', 20, 170, 100, 20, GB1O));
      RB16O := New(PRadioButton, Init(@Self, id_RB16O, 'DataList', W div 2, 170, 100, 20, GB1O));
      RB17O := New(PRadioButton, Init(@Self, id_RB17O, 'Association', 20, 190, 100, 20, GB1O));
      RB18O := New(PRadioButton, Init(@Self, id_RB18O, 'Assoc Syn', W div 2, 190, 100, 20, GB1O));
      RB19O := New(PRadioButton, Init(@Self, id_RB19O, 'Predicate', 20, 210, 100, 20, GB1O));
      RB20O := New(PRadioButton, Init(@Self, id_RB20O, 'Pred Syn', W div 2, 210, 100, 20, GB1O));
```

```
      RB21O := New(PRadioButton, Init(@Self, id_RB21O, 'Filter', 20, 230, 100, 20, GB1O));
      RB22O := New(PRadioButton, Init(@Self, id_RB22O, 'Counter', W div 2, 230, 100, 20, GB1O));
      RB23O := New(PRadioButton, Init(@Self, id_RB23O, 'Quantifier', 20, 250, 100, 20, GB1O));
      RB24O := New(PRadioButton, Init(@Self, id_RB24O, 'Measure', W div 2, 250, 100, 20, GB1O));
      RB25O := New(PRadioButton, Init(@Self, id_RB25O, 'Node Kind', 20, 270, 100, 20, GB1O));
      RB26O := New(PRadioButton, Init(@Self, id_RB26O, 'Kind Syn', W div 2, 270, 100, 20, GB1O));
      RB27O := New(PRadioButton, Init(@Self, id_RB27O, 'Link Type', 20, 290, 100, 20, GB1O));
      RB28O := New(PRadioButton, Init(@Self, id_RB28O, 'Type Syn', W div 2, 290, 100, 20, GB1O));
      RB29O := New(PRadioButton, Init(@Self, id_RB29O, 'Context', 20, 310, 100, 20, GB1O));
      RB30O := New(PRadioButton, Init(@Self, id_RB30O, 'Login', W div 2, 310, 100, 20, GB1O));
      RB31O := New(PRadioButton, Init(@Self, id_RB31O, 'Link', 20, 330, 100, 20, GB1O));
      RB32O := New(PRadioButton, Init(@Self, id_RB32O, 'Content Link', W div 2, 330, 100, 20, GB1O));
      RB33O := New(PRadioButton, Init(@Self, id_RB33O, 'Graphic Link', 20, 350, 100, 20, GB1O));
      RB34O := New(PRadioButton, Init(@Self, id_RB34O, 'Context Link', W div 2, 350, 100, 20, GB1O));
      RB35O := New(PRadioButton, Init(@Self, id_RB35O, 'undefined', 20, 370, 100, 20, GB1O));
      ST1O  := New(PStatic, Init(@Self, id_ST1O, 'context:', 10, H - 130, 60, 20, 40));
      ST2O  := New(PStatic, Init(@Self, id_ST2O, '', 70, H - 130, 300, 20, 40));
      ST3O  := New(PStatic, Init(@Self, id_ST3O, 'name:', 10, H - 105, 60, 20, 40));
      ST4O  := New(PStatic, Init(@Self, id_ST4O, '', 70, H - 105, 300, 20, 40));
      BN1O  := New(PButton, Init(@Self, id_BN1O, 'List', 95, H - 80, 70, 20, False));
      BN2O  := New(PButton, Init(@Self, id_BN2O, 'Create', 180, H - 80, 70, 20, False));
      BN3O  := New(PButton, Init(@Self, id_BN3O, 'Edit', 95, H - 55, 70, 20, False));
      BN4O  := New(PButton, Init(@Self, id_BN4O, 'Delete', 180, H - 55, 70, 20, False));
      BN5O  := New(PButton, Init(@Self, id_BN5O, 'Annotate', 10, H - 55, 70, 20, False));
      BN6O  := New(PButton, Init(@Self, id_BN6O, 'Context', 10, H - 80, 70, 20, False));
    end;
end;

procedure ObjectsWin.SetUpWindow;
var
  ContextId : LongInt;
begin  {SetUpWindow}
  ControlWindow.SetUpWindow;
  RB1O^.SetCheck(bf_Checked);
  ContextId := HermesApp.GetContext;
  if ContextId > 0
    then ST2O^.SetText(DBNameOf(ContextId))       {insert name of current perspective}
    else ST2O^.SetText('hermes_universal_context');
end;   {SetUpWindow}

function ObjectsWin.GetClass: Char;
begin
  if      (RB1O^.GetCheck = bf_Checked) then GetClass := ot_Character
  else if (RB2O^.GetCheck = bf_Checked) then GetClass := ot_Number
  else if (RB3O^.GetCheck = bf_Checked) then GetClass := ot_Booleans
  else if (RB4O^.GetCheck = bf_Checked) then GetClass := ot_Image
  else if (RB5O^.GetCheck = bf_Checked) then GetClass := ot_Pen
  else if (RB6O^.GetCheck = bf_Checked) then GetClass := ot_Sound
  else if (RB7O^.GetCheck = bf_Checked) then GetClass := ot_Video
  else if (RB8O^.GetCheck = bf_Checked) then GetClass := ot_Animation
  else if (RB9O^.GetCheck = bf_Checked) then GetClass := ot_ComputedView
  else if (RB10O^.GetCheck = bf_Checked) then GetClass := ot_HermesGraphic
  else if (RB11O^.GetCheck = bf_Checked) then GetClass := ot_HermesPolyline
  else if (RB12O^.GetCheck = bf_Checked) then GetClass := ot_UnitCube
  else if (RB13O^.GetCheck = bf_Checked) then GetClass := ot_Sweep
  else if (RB14O^.GetCheck = bf_Checked) then GetClass := ot_HermesNode
  else if (RB15O^.GetCheck = bf_Checked) then GetClass := ot_ResultList
  else if (RB16O^.GetCheck = bf_Checked) then GetClass := ot_DataList
  else if (RB17O^.GetCheck = bf_Checked) then GetClass := ot_Association
  else if (RB18O^.GetCheck = bf_Checked) then GetClass := ot_AssociationSynonym
  else if (RB19O^.GetCheck = bf_Checked) then GetClass := ot_Predicate
  else if (RB20O^.GetCheck = bf_Checked) then GetClass := ot_PredicateSynonym
  else if (RB21O^.GetCheck = bf_Checked) then GetClass := ot_Filter
  else if (RB22O^.GetCheck = bf_Checked) then GetClass := ot_Counter
  else if (RB23O^.GetCheck = bf_Checked) then GetClass := ot_Quantifier
  else if (RB24O^.GetCheck = bf_Checked) then GetClass := ot_Measure
  else if (RB25O^.GetCheck = bf_Checked) then GetClass := ot_NodeKind
  else if (RB26O^.GetCheck = bf_Checked) then GetClass := ot_NodeKindSynonym
  else if (RB27O^.GetCheck = bf_Checked) then GetClass := ot_LinkType
  else if (RB28O^.GetCheck = bf_Checked) then GetClass := ot_LinkTypeSynonym
  else if (RB29O^.GetCheck = bf_Checked) then GetClass := ot_Context
  else if (RB30O^.GetCheck = bf_Checked) then GetClass := ot_Login
  else if (RB31O^.GetCheck = bf_Checked) then GetClass := ot_Link
  else if (RB32O^.GetCheck = bf_Checked) then GetClass := ot_ContentLink
  else if (RB33O^.GetCheck = bf_Checked) then GetClass := ot_GraphicLink
  else if (RB34O^.GetCheck = bf_Checked) then GetClass := ot_ContextLink
```

```
  else if (RB35O^.GetCheck = bf_Checked) then GetClass := ot_undefined
end;

procedure ObjectsWin.IDBN1O(var Msg: TMessage);                          {List Button}
var
  TheIndex     : Integer;
  TheSelection : PChar;
  MainName     : PChar;
      procedure ListSynonyms(ObjectType: ObjectTypes);
      begin  {ListSynonyms}
        TheIndex := -1;
        FillLister(@Self, DBListNames(ObjectType), TheIndex, MainName, True);
        if (TheIndex > -1)
          then FillLister(@Self, DBListSynonyms(MainName, ObjectType), TheIndex, TheSelection, True);
      end;   {ListSynonyms}
begin
  TheIndex := -1;
  case GetClass of
    ot_NodeKindSynonym : ListSynonyms(ot_NodeKind);
    ot_LinkTypeSynonym : ListSynonyms(ot_LinkType);
    ot_PredicateSynonym : ListSynonyms(ot_Predicate);
    ot_AssociationSynonym : ListSynonyms(ot_Association);
  else
    FillLister(@Self, DBListNames(GetClass), TheIndex, TheSelection, True);
  end;   {case}
  if (TheIndex > -1)
    then ST4O^.SetText(TheSelection);
end;

procedure ObjectsWin.IDBN2O(var Msg: TMessage);                          {Create Button}
      procedure CreateSynonym(ObjectType: ObjectTypes);
      var
        MainName : PChar;
        NewName  : StrName;
        TheIndex : Integer;
      begin  {CreateSynonym}
        StrCopy(MainName, '');
        StrCopy(NewName, '');
        TheIndex := -1;
        FillLister(@Self, DBListNames(ObjectType), TheIndex, MainName, True);
        if (TheIndex > -1)
          then
            if (HermesApp.ExecDialog(New(PInputDialog, Init(@Self, 'List Synonyms',
                       'Enter name for new synonym: ', NewName, NameKeyLength))) = id_OK)
                 and (StrLen(NewName) > 0)
              then DBAddSynonym(NewName, MainName, ObjectType);
      end;   {CreateSynonym}
begin
  case GetClass of
    ot_Context :  Application^.ExecDialog(New(PContextsWin, Init(@Self)));
    ot_NodeKind :  CreateKind;
    ot_LinkType :  CreateType;
    ot_HermesNode :  Application^.ExecDialog(New(PTextWin, Init(@Self, 0)));
    ot_Link :  Application^.ExecDialog(New(PLinksWin, Init(@Self)));
    ot_ResultList :  {Application^.ExecDialog(New(PResultWin, Init(@Self, nil)))};
    ot_DataList :  Application^.ExecDialog(New(PQueriesWin, Init(@Self)));
    ot_Association :  NotImplemented;
    ot_Predicate :  NotImplemented;
    ot_Filter :  NotImplemented;

    ot_NodeKindSynonym :  CreateSynonym(ot_NodeKind);
    ot_LinkTypeSynonym :  CreateSynonym(ot_LinkType);
    ot_PredicateSynonym :  CreateSynonym(ot_Predicate);
    ot_AssociationSynonym :  CreateSynonym(ot_Association);
  else
    NotImplemented;
  end;  {case}
end;

procedure ObjectsWin.IDBN3O(var Msg: TMessage); {Edit Button}
begin
  case GetClass of
    ot_Context :  NotImplemented;
    ot_NodeKind :  NotImplemented;
  else     NotImplemented;
  end;  {case}
```

```
end;

procedure ObjectsWin.IDBN4O(var Msg: TMessage); {Delete Button}
begin
  case GetClass of
    ot_Context :  NotImplemented;
    ot_NodeKind :  NotImplemented;
  else     NotImplemented;
  end;  {case}
end;

procedure ObjectsWin.IDBN5O(var Msg: TMessage);  {Annotate Button}
var
  TheObject : StrName;
  TheId     : LongInt;
begin
  IDBN1O(Msg);          {make sure object is from GetClass}
  if (ST4O^.GetText(TheObject, 40) > 0)
    then
      begin
        TheId :=  DBIdOf(TheObject, GetClass);
        if (TheId > 0)
          then Application^.MakeWindow(New(PNavigatorWin, Init(@Self, TheId)))
          else ErrorMsg(et_NOTICE, 'Could not find the object.');
      end
    else ErrorMsg(et_NOTICE, 'No object was chosen.');
end;

procedure ObjectsWin.IDBN6O(var Msg: TMessage);  {Context Button}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DisplayContextTree, TheIndex, TheSelection, False);
  if (TheIndex > -1)
    then
      begin
        TheSelection := StripBlanks(TheSelection);
        ST2O^.SetText(TheSelection);
        HermesApp.SetContext(DBIdOf(TheSelection, ot_Context));
      end;
end;

procedure ObjectsWin.CreateType;
var
       NewName         : Array [0..NameKeyLength] of Char;
begin
  StrCopy(NewName, ' ');
  if (HermesApp.ExecDialog(New(PInputDialog, Init(@Self, 'Create Link Type',
                            'Enter name of new Link Type', NewName, NameKeyLength))) = id_OK)
                      and (StrLen(NewName) > 0)
    then
      if (DBGetObjectId(NewName, ot_LinkType) > 0)
        then ErrorMsg(et_NOTICE, 'This Link Type name already exists.')
        else
                              if DBPutName(New(PLinkType, Init(NewName)))
            then CreatePlural(NewName, ot_LinkType);
end;


procedure ObjectsWin.CreateKind;
var
       NewName         : Array [0..NameKeyLength] of Char;
begin
  StrCopy(NewName, ' ');
  if (HermesApp.ExecDialog(New(PInputDialog, Init(@Self, 'Create Node Kind',
                            'Enter name of new Node Kind', NewName, NameKeyLength))) = id_OK)
                      and (StrLen(NewName) > 0)
    then
      if (DBGetObjectId(NewName, ot_NodeKind) > 0)
        then ErrorMsg(et_NOTICE, 'This Node Kind name already exists.')
        else
          if DBPutName(New(PNodeKind, Init(NewName)))
            then CreatePlural(NewName, ot_NodeKind);
end;
```

```
procedure ObjectsWin.CreatePlural(NewName: PChar; ObjectType: ObjectTypes);
var
       PluralName : Array [0..NameKeyLength] of Char;
begin
       if (NewName[StrLen(NewName) - 1] = 's')       {create its plural also}
              then
                     StrLCopy(PluralName, NewName,  StrLen(NewName)-1)
              else
      if (NewName[StrLen(NewName) - 1] = 'y')
                                then
         begin
                                              StrLCopy(PluralName, NewName, StrLen(NewName)-1);
           StrCat(PluralName, 'ies');
         end
                                else
         begin
           StrCopy(PluralName, NewName);
                                              StrCat(PluralName, 's');
         end;

       if (HermesApp.ExecDialog(New(PInputDialog, Init(@Self, 'Save plural?',
                     'Edit plural name:', PluralName, NameKeyLength))) = id_OK)
    then
      if (DBGetObjectId(PluralName, ObjectType) = 0)
                                then DBAddSynonym(PluralName, NewName, ObjectType)
        else MessageBox(HWindow, PluralName, 'This plural already exists', mb_OK);
end;


       {****************************************
       *
       *       methods for EntryFormWin
       *
       ****************************************}

constructor EntryFormWin.Init(AParent: PWindowsObject);
begin
  TDlgWindow.Init(AParent, 'ENTRYFORMS');
  ST0E := New(PStatic,      InitResource(@Self, id_ST0E,80));
  BN0E := New(PButton,      InitResource(@Self, id_BN0E));
  ST1E := New(PStatic,      InitResource(@Self, id_ST1E,80));
  EB1E := New(PEdit,        InitResource(@Self, id_EB1E, 40));
  BN1E := New(PButton,      InitResource(@Self, id_BN1E));
  ST2E := New(PStatic,      InitResource(@Self, id_ST2E, 40));
  GB1E := New(PGroupBox,    InitResource(@Self, id_GB1E));
  RB1E := New(PRadioButton, InitResource(@Self, id_RB1E));
  BN2E := New(PButton,      InitResource(@Self, id_BN2E));
  RB2E := New(PRadioButton, InitResource(@Self, id_RB2E));
  BN3E := New(PButton,      InitResource(@Self, id_BN3E));
  ST3E := New(PStatic,      InitResource(@Self, id_ST3E, 20));
  BN4E := New(PButton,      InitResource(@Self, id_BN4E));
  RB3E := New(PRadioButton, InitResource(@Self, id_RB3E));
  BN5E := New(PButton,      InitResource(@Self, id_BN5E));
  ST4E := New(PStatic,      InitResource(@Self, id_ST4E, 5));
  BN6E := New(PButton,      InitResource(@Self, id_BN6E));
  ST5E := New(PStatic,      InitResource(@Self, id_ST5E, 400));
  ST5E^.Attr.Style :=  ST5E^.Attr.Style or ss_BlackFrame;
  BN7E := New(PButton,      InitResource(@Self, id_BN7E));
  BN8E := New(PButton,      InitResource(@Self, id_BN8E));
  BN9E := New(PButton,      InitResource(@Self, id_Cancel));
end;

procedure EntryFormWin.SetUpWindow;
begin
  TDlgWindow.SetUpWindow;
  RB1E^.SetCheck(bf_Checked);    {default: E::= T}
  EB1E^.SetText('');
  ST2E^.SetText(PContext(DBGet(HermesApp.GetContext))^.GetName);
  ST5E^.SetText('');
  TheType := nil;
  TheEF1  := nil;
  TheEF2  := nil;
  TheEF3  := nil;
  TheEF4  := nil;
```

```
  NewEF   := nil;
  LastBN  := -1;
  Show(sw_Show);
end;

procedure EntryFormWin.WMQueryMsg(var Msg: TMessage);
begin
  case LastBN of
    1 : begin
          TheEF1 := PAssociation(Msg.lParam);
          NewEF  := New(PAssociationWith, Init(NoName, TheEF1, TheEF2));
        end;

    2 : begin
          TheEF2 := PAssociation(Msg.lParam);
          NewEF  := New(PAssociationWith, Init(NoName, TheEF1, TheEF2));
        end;

    3 : begin
          TheEF3 := PAssociation(Msg.lParam);
          NewEF  := New(PAssociationComb, Init(NoName, TheEF3, AND_UNION, TheEF4));
        end;

    4 : begin
          TheEF4 := PAssociation(Msg.lParam);
          NewEF  := New(PAssociationComb, Init(NoName, TheEF3, AND_UNION, TheEF4));
        end;
  end;  {case}
  ST5E^.SetText(NewEF^.HermesDisplay);
end;

procedure EntryFormWin.IDBN0E(var Msg: TMessage);         {old EF}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  FillLister(@Self, DBListNames(ot_Association), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then
      begin
        NewEF := PAssociation(DBGetNamedObject(TheSelection, ot_Association, HermesApp.GetContext));
        if (NewEF <> nil)
          then
            begin
              ST5E^.SetText(NewEF^.HermesDisplay);
              EB1E^.SetText(NewEF^.GetName);
              if (TypeOf(NewEF^) = TypeOf(TAssociationType))
                then
                  begin
                    TheType := NewEF^.GetText;
                    RB1E^.Check;
                    RB2E^.UnCheck;
                    RB3E^.UnCheck;
                  end
              else if (TypeOf(NewEF^) = TypeOf(TAssociationWith))
                then
                  begin
                    TheEF1 := NewEF^.GetParam1;
                    TheEF2 := NewEF^.GetParam2;
                    RB1E^.UnCheck;
                    RB2E^.Check;
                    RB3E^.UnCheck;
                  end
              else if (TypeOf(NewEF^) = TypeOf(TAssociationComb))
                then
                  begin
                    TheEF3 := NewEF^.GetParam1;
                    TheEF4 := NewEF^.GetParam2;
                    RB1E^.UnCheck;
                    RB2E^.UnCheck;
                    RB3E^.Check;
                  end;
            end;
      end;
end;
```

```
procedure EntryFormWin.IDBN1E(var Msg: TMessage);         {context}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  FillLister(@Self, DisplayContextTree, TheIndex, TheSelection, False);
  if (TheIndex > -1)
    then
      begin
        TheSelection := StripBlanks(TheSelection);
        ST2E^.SetText(TheSelection);
        HermesApp.SetContext(DBIdOf(TheSelection, ot_Context));
        ST5E^.SetText('');
        NewEF   := nil;
      end;
end;

procedure EntryFormWin.IDBN2E(var Msg: TMessage);         {T}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  if (RB1E^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 0;
        FillLister(@Self, DBListNames(ot_LinkType), TheIndex, TheSelection, True);
        if (TheIndex > -1)
          then
            begin
              TheType := StrNew(TheSelection);
              NewEF := New(PAssociationType, Init(NoName, TheType));
              ST5E^.SetText(NewEF^.HermesDisplay);
            end;
      end;
end;

procedure EntryFormWin.IDBN3E(var Msg: TMessage);         {EF1}
var
  TheDlg : PEntryFormWin;
begin
  if (RB2E^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 1;
        TheDlg := New(PEntryFormWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure EntryFormWin.IDBN4E(var Msg: TMessage);         {EF2}
var
  TheDlg : PEntryFormWin;
begin
  if (RB2E^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 2;
        TheDlg := New(PEntryFormWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure EntryFormWin.IDBN5E(var Msg: TMessage);         {EF3}
var
  TheDlg : PEntryFormWin;
begin
  if (RB3E^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 3;
        TheDlg := New(PEntryFormWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;
```

```
procedure EntryFormWin.IDBN6E(var Msg: TMessage);         {EF4}
var
  TheDlg : PEntryFormWin;
begin
  if (RB3E^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 4;
        TheDlg := New(PEntryFormWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure EntryFormWin.IDBN7E(var Msg: TMessage);         {Save EF}
var
  TheName : StrName;
begin
  if (NewEF <> nil)
    then
      begin
        EB1E^.GetText(TheName, 40);
        if (StrLen(TheName) > 0)
          then                            {save new EntryForm}
            begin
              NewEF^.SetName(TheName);
              NewEF := NewEF^.Copy;
              DBPutNamedObject(NewEF, HermesApp.GetContext);
            end
          else ErrorMsg(et_NOTICE, 'No name given, cannot save the EntryForm.');
        SendMessage(Parent^.HWindow, wm_QueryMsg, 0, LongInt(NewEF));
      end;
  if CanClose
    then EndDlg(id_Ok);
end;

procedure EntryFormWin.IDBN8E(var Msg: TMessage);         {OK}
var
  TheName : StrName;
begin
  if (NewEF <> nil)
    then SendMessage(Parent^.HWindow, wm_QueryMsg, 0, LongInt(NewEF));
  if CanClose
    then EndDlg(id_Ok);
end;


        {****************************************
        *
        *       methods for QueriesWin
        *
        ****************************************}

constructor QueriesWin.Init(AParent: PWindowsObject);
begin
  TDlgWindow.Init(AParent, 'QUERIES');
  BN1Q := New(PButton,      InitResource(@Self, id_BN1Q));
  ST1Q := New(PEdit,        InitResource(@Self, id_ST1Q, 40));
  GB1Q := New(PGroupBox,    InitResource(@Self, id_GB1Q));
  RB1Q := New(PRadioButton, InitResource(@Self, id_RB1Q));
  BN3Q := New(PButton,      InitResource(@Self, id_BN3Q));
  BN4Q := New(PButton,      InitResource(@Self, id_BN4Q));
  RB2Q := New(PRadioButton, InitResource(@Self, id_RB2Q));
  BN5Q := New(PButton,      InitResource(@Self, id_BN5Q));
  BN6Q := New(PButton,      InitResource(@Self, id_BN6Q));
  BN7Q := New(PButton,      InitResource(@Self, id_BN7Q));
  BN8Q := New(PButton,      InitResource(@Self, id_BN8Q));
  BN9Q := New(PButton,      InitResource(@Self, id_BN9Q));
  BN10Q := New(PButton,      InitResource(@Self, id_BN10Q));
  BN11Q := New(PButton,      InitResource(@Self, id_BN11Q));
  BN12Q := New(PButton,      InitResource(@Self, id_BN12Q));
  BN13Q := New(PButton,      InitResource(@Self, id_BN13Q));
  BN14Q := New(PButton,      InitResource(@Self, id_BN14Q));
  ST3Q  := New(PStatic,      InitResource(@Self, id_ST3Q, 40));
  BN16Q := New(PButton,      InitResource(@Self, id_BN16Q));
  ST4Q  := New(PStatic,      InitResource(@Self, id_ST4Q, 40));
  BN17Q := New(PButton,      InitResource(@Self, id_BN17Q));
```

```
  BN18Q := New(PButton,      InitResource(@Self, id_BN18Q));
end;

procedure QueriesWin.SetUpWindow;
begin
  TDlgWindow.SetUpWindow;
  RB1Q^.SetCheck(bf_Checked);    {default: Q::= E of O}
  ST1Q^.SetText('');
  ST3Q^.SetText('');
  ST4Q^.SetText(PContext(DBGet(HermesApp.GetContext))^.GetName);

  TheEntryForm      := nil;
  TheNode           := 0;
  TheRelationship1  := nil;
  TheRelationship2  := nil;
  TheSubject        := nil;
  TheFilter         := nil;
  TheHermeneutic    := nil;
  TheBoolean        := nil;
  TheQuery          := nil;
  NewQ              := nil;
  LastBN            := -1;
  Show(sw_Show);
end;

procedure QueriesWin.WMQueryMsg(var Msg: TMessage);
begin
  case LastBN of
    1 : begin    {EntryForm}
          TheEntryForm := PAssociation(Msg.lParam);
          if (NewQ <> nil)
            then Dispose(NewQ, Done);
          if (TheNode <> 0)
            then NewQ  := New(PDataListOf, Init(NoName, TheEntryForm, New(PDataListId, Init(NoName,
TheNode))))
            else NewQ := nil;
        end;

    2 : begin    {Node}
          {does not come thru here}
        end;

    3 : begin    {Art1}
        end;

    4 : begin    {Relationship1}
        end;

    5 : begin    {Prp}
         end;

    6 : begin    {Art2}
         end;

    7 : begin    {Subject}
         end;

    8 : begin    {Filter}
         end;

    9 : begin    {Hermeneutic}
         end;

    10 : begin   {Relationship2}
         end;

    11 : begin   {Boolean}
         end;

    12 : begin   {Query}
         end;

  end;  {case}

  if (NewQ <> nil)
    then ST3Q^.SetText(NewQ^.HermesDisplay);
```

```
end;

procedure QueriesWin.IDBN1Q(var Msg: TMessage);         {Select Query button}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  FillLister(@Self, DBListNames('Q'), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then
      begin
        NewQ := PDataList(DBGetNamedObject(TheSelection, 'Q', HermesApp.GetContext));
        ST3Q^.SetText(NewQ^.HermesDisplay);
        ST1Q^.SetText(NewQ^.GetName);
        if (TypeOf(NewQ^) = TypeOf(TDataListOf))
          then
            begin
              TheEntryForm := NewQ^.GetParam1;
              TheNode      := NewQ^.GetId;
              RB1Q^.Check;
              RB2Q^.UnCheck;
            end;
      end;
end;

procedure QueriesWin.IDBN3Q(var Msg: TMessage);         {EF}
var
  TheDlg : PEntryFormWin;
begin
  if (RB1Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 1;
        TheDlg := New(PEntryFormWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure QueriesWin.IDBN4Q(var Msg: TMessage);         {nOde}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  if (RB1Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 2;
        TheIndex := -1;
        FillLister(@Self, DBListNames('C'), TheIndex, TheSelection, True);
        if (TheIndex > -1)
          then
            begin
              TheNode := DBIdOf(TheSelection, 'C');         {TheNode : LongInt}
              NewQ  := New(PDataListOf, Init(NoName, TheEntryForm, New(PDataListId, Init(NoName,
TheNode))));
              if (NewQ <> nil)
                then ST3Q^.SetText(NewQ^.HermesDisplay);
            end;
      end;
end;

procedure QueriesWin.IDBN5Q(var Msg: TMessage);         {Art1}
var
  TheDlg : PQueriesWin;                      {change to: PArtWin}
begin
  if (RB2Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 3;
        TheDlg := New(PQueriesWin, Init(@Self));     {change to: PArtWin}
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure QueriesWin.IDBN6Q(var Msg: TMessage);         {R1}
var
```

```
  TheDlg : PQueriesWin;
begin
  if (RB2Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 4;
        TheDlg := New(PQueriesWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure QueriesWin.IDBN7Q(var Msg: TMessage);         {Prp }
var
  TheDlg : PQueriesWin;
begin
  if (RB2Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 5;
        TheDlg := New(PQueriesWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure QueriesWin.IDBN8Q(var Msg: TMessage);         {Art2}
var
  TheDlg : PQueriesWin;
begin
  if (RB2Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 6;
        TheDlg := New(PQueriesWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure QueriesWin.IDBN9Q(var Msg: TMessage);         {Subject}
var
  TheDlg : PQueriesWin;
begin
  if (RB2Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 7;
        TheDlg := New(PQueriesWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure QueriesWin.IDBN10Q(var Msg: TMessage);         {Filter}
var
  TheDlg : PQueriesWin;
begin
  if (RB2Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 8;
        TheDlg := New(PQueriesWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure QueriesWin.IDBN11Q(var Msg: TMessage);         {Hermeneutic}
var
  TheDlg : PQueriesWin;
begin
  if (RB2Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 9;
        TheDlg := New(PQueriesWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;
```

```
procedure QueriesWin.IDBN12Q(var Msg: TMessage);         {Relationship2}
var
  TheDlg : PQueriesWin;
begin
  if (RB2Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 10;
        TheDlg := New(PQueriesWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure QueriesWin.IDBN13Q(var Msg: TMessage);         {Boolean}
var
  TheDlg : PQueriesWin;
begin
  if (RB2Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 11;
        TheDlg := New(PQueriesWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure QueriesWin.IDBN14Q(var Msg: TMessage);         {Query}
var
  TheDlg : PQueriesWin;
begin
  if (RB2Q^.GetCheck = bf_Checked)
    then
      begin
        LastBN := 12;
        TheDlg := New(PQueriesWin, Init(@Self));
        Application^.ExecDialog(TheDlg);
      end;
end;

procedure QueriesWin.IDBN16Q(var Msg: TMessage);         {Select Context}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DisplayContextTree, TheIndex, TheSelection, False);
  if (TheIndex > -1)
    then
      begin
        TheSelection := StripBlanks(TheSelection);
        ST4Q^.SetText(TheSelection);
        HermesApp.SetContext(DBIdOf(TheSelection, ot_Context));
      end;
end;

procedure QueriesWin.IDBN17Q(var Msg: TMessage);         {Save Query}
var
  TheName : StrName;
begin
  if (NewQ <> nil)
    then
      begin
        SendMessage(Parent^.HWindow, wm_QueryMsg, 0, LongInt(NewQ));
        ST1Q^.GetText(TheName, 40);
        if (HermesApp.ExecDialog(New(PInputDialog, Init(@Self, 'Save Query',
                       'Enter name for new query: ', TheName, NameKeyLength))) = id_OK)
                 and (StrLen(TheName) > 0)
          then
            begin
              NewQ^.SetName(TheName);
              DBPutNamedObject(NewQ, HermesApp.GetContext);
              ErrorMsg(et_NOTICE, 'The new query has been saved.');
              if CanClose
                then Ok(Msg);
            end;
      end;
```

```
end;

procedure QueriesWin.IDBN18Q(var Msg: TMessage);         {OK}
begin
  if (NewQ <> nil)
    then SendMessage(Parent^.HWindow, wm_QueryMsg, 0, LongInt(NewQ));
  if CanClose
    then Ok(Msg);
end;


        {*************************************
        *
        *       methods for LinksWin
        *
        *************************************}

constructor LinksWin.Init(AParent: PWindowsObject);
begin
  ControlWindow.Init(AParent, 'Control for Links');
  TheControlType := LINKS;
  with Attr do
    begin
      BN1L := New(PButton, Init(@Self, id_BN1L, 'Select From Node', 5, 10, 200, 20, False));
      ST1L := New(PStatic, Init(@Self, id_ST1L, '', 5, 35 , 200, 20, 0));
      BN2L := New(PButton, Init(@Self, id_BN2L, 'Select To Node', 5, 65, 200, 20, False));
      ST2L := New(PStatic, Init(@Self, id_ST2L, '', 5, 90, 200, 20, 0));
      ST3L := New(PStatic, Init(@Self, id_ST3L, 'Name', 5, 120 , 200, 20, 0));
      EB1L := New(PEdit,   Init(@Self, id_EB1L, '', 5, 145, 200, 20, 40, False));
      BN4L := New(PButton, Init(@Self, id_BN4L, 'Select Link Type', 5, 175, 200, 20, False));
      ST4L := New(PStatic, Init(@Self, id_ST4L, '', 5, 200 , 200, 20, 0));
      BN5L := New(PButton, Init(@Self, id_BN5L, 'Select Context', 5, 230, 200, 20, False));
      ST5L := New(PStatic, Init(@Self, id_ST5L, '', 5, 255, 200, 20, 0));
      BN6L := New(PButton, Init(@Self, id_BN6L, 'Accept', 20, H-50, 100, 20, False));
      BN7L := New(PButton, Init(@Self, id_BN7L, 'Cancel', (W div 2)+20, H-50, 100, 20, True));
    end;
end;

procedure LinksWin.SetUpWindow;
begin
  ControlWindow.SetUpWindow;
  ST5L^.SetText(DBGet(HermesApp.GetContext)^.GetName);
end;

procedure LinksWin.IDBN1L(var Msg: TMessage);        {Select From Node}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_Character), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then ST1L^.SetText(TheSelection);
end;

procedure LinksWin.IDBN2L(var Msg: TMessage);        {Select To Node}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_Character), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then ST2L^.SetText(TheSelection);
end;

procedure LinksWin.IDBN4L(var Msg: TMessage);        {Select Link Type}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_LinkType), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then ST4L^.SetText(TheSelection);
end;
```

```
procedure LinksWin.IDBN5L(var Msg: TMessage);        {Select Context}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DisplayContextTree, TheIndex, TheSelection, False);
  if (TheIndex > -1)
    then
      begin
        TheSelection := StripBlanks(TheSelection);
        ST5L^.SetText(TheSelection);
      end;
end;

procedure LinksWin.IDBN6L(var Msg: TMessage);        {Accept}
var
  TheFromNode : StrName;
  TheToNode   : StrName;
  TheName     : StrName;
  TheType     : StrName;
  TheContext  : LongInt;
begin
  EB1L^.GetText(TheName, 40);
  if (StrLen(TheName) > 0)
    then StrCopy(TheName, NoName);
  ST1L^.GetText(TheFromNode, 40);
  ST2L^.GetText(TheToNode, 40);
  ST4L^.GetText(TheType, 40);
  TheContext := HermesApp.GetContext;
  if (StrLen(TheFromNode) = 0)
    then ErrorMsg(et_NOTICE, 'A From Node must be selected.')
  else if (StrLen(TheToNode) = 0)
    then ErrorMsg(et_NOTICE, 'A To Node must be selected.')
  else if (StrLen(TheType) = 0)
    then ErrorMsg(et_NOTICE, 'A Link Type must be selected.')
  else
    begin
      CreateHypertextSwitch(0, DBIdOf(TheFromNode, 'C'), DBIdOf(TheToNode, 'C'), TheType, TheContext,
0, nil);
    end;
  ErrorMsg(et_NOTICE, 'The link is formed.');
  Done;
end;

procedure LinksWin.IDBN7L(var Msg: TMessage);        {Cancel}
begin
  Done;
end;


        {****************************************
        *
        *       methods for ModifyWin
        *
  *       modify existing Node: TheNode <> nil
        ****************************************}

constructor ModifyWin.Init(AParent: PWindowsObject; ANodeId: LongInt);

    function ContextTraversesSubLink(ASubLink : PSubLink) : Boolean; far;
    var
      TheVarContext : LongInt;   {must be passed as a VAR parameter}
    begin  {ContextTraversesSubLink}
      TheVarContext := HermesApp.GetContext;
      ContextTraversesSubLink := TraversesSubLink(TheVarContext, ASubLink);
    end;   {ContextTraversesSubLink}

    procedure BuildList(ALinkId : PLinkId); far;
    var
      TheLink    : PLink;
      TheSublink : PSublink;
      ThePO      : PPersistentObject;
      TheContext : LongInt;
    begin  {BuildList}
```

```
      TheLink := PLink(DBGet(ALinkId^.GetId));
      TheContext := HermesApp.GetContext;
      if assigned(TheLink)then
        if TraversesLink(TheContext, TheLink)then
          begin
            TheSublink := TheLink^.GetSublinks^.FirstThat(@ContextTraversesSublink);
            if assigned(TheSublink) then
              if PrivsEditNode(TheSublink^.GetListOfContexts^.GetOriginalContext, TheContext)
                then
                  begin
                    ThePO := DBGet(TheLink^.GetToPO);
                    if (ThePO^.GetType = ot_Character)
                      then TheList^.Insert(ThePO);
                  end;
          end;
    end;   {BuildList}

var
  ARect           : TRect;
  AThird          : Integer;
begin
  ControlWindow.Init(AParent, 'Modify a Node');
  TheControlType := MODIFY;
  TheContent := nil;
  with Attr do
    begin
      GetClientRect(Application^.MainWindow^.HWindow, ARect);
      AThird :=  (ARect.right - ARect.left) div 3;
      X :=  ARect.left + (2 * AThird) - 140;             {same as for top of GraphicalWindow}
      Y :=  ARect.top + 43 + ((ARect.Bottom - ARect.Top - 74) div 2);
      W :=  AThird + 139;
      H :=  (ARect.Bottom - ARect.Top - 74) div 2;
      EB2M := New(PEdit,   Init(@Self, id_EB2M, '', 5, 5, W-15, H-90, 4000, True));
      EB3M := New(PEdit,   Init(@Self, id_EB3M, '', 25, H-80, W-15, 25, 80, False));
      BN4M := New(PButton, Init(@Self, id_BN4M, 'Accept', (W div 2)-120, H-50, 100, 20, True));
      BN5M := New(PButton, Init(@Self, id_BN5M, 'Cancel', (W div 2)+20,  H-50, 100, 20, False));
    end;

  if (ANodeId <> 0)
    then
      begin
        TheNode := PHermesNode(DBGet(ANodeId));
        TheIndex := 0;
        TheList := New(PHermesList, Init);
        if assigned(TheNode)
          then TheNode^.GetContentOutLinks^.ForEach(@BuildList)
          else ErrorMsg(et_WARNING, 'Valid node id not found');
      end
    else ErrorMsg(et_WARNING, 'Valid node id not specified');
end;

destructor ModifyWin.Done;
begin
{  if (TheNode <> nil)
    then Dispose(TheNode, Done);
  if (TheList <> nil)
    then Dispose(TheList, Done);  }  {can't dispose -- might be saved to collection}
  ControlWindow.Done;
end;

procedure ModifyWin.SetUpWindow;
var
  NumLines: Integer;
begin
  ControlWindow.SetUpWindow;
  SetUpContent;
end;

procedure ModifyWin.SetUpContent;
var
  NumLines: Integer;
  TheMessage : array[0..80] of Char;
  IndexStr   : array[0..80] of Char;
  CountStr   : array[0..80] of Char;
begin
  if (TheList^.Count > TheIndex)
```

```
    then
      begin
        TheContent := PCharacterSimple(TheList^.At(TheIndex));
        EB2M^.SetText(TheContent^.EvalCharacter);
        Inc(TheIndex);
        Str(TheIndex, IndexStr);
        Str(TheList^.Count, CountStr);
        StrCopy(TheMessage, 'This is content #');
        StrCat(TheMessage, IndexStr);
        StrCat(TheMessage, '  of  ');
        StrCat(TheMessage, CountStr);
        StrCat(TheMessage, ' contents in this context.');
        EB3M^.SetText(TheMessage);
      end
    else
      begin
        ErrorMsg(et_NOTICE, 'There are no statements to edit');
        {Done; }
      end;
end;

procedure ModifyWin.IDBN4M(var Msg: TMessage);        {Accept}
var
  TheLine    : array[0..400] of Char;
  TheText    : array[0..4000] of Char;
  NumLines   : Integer;
  Index      : Integer;
begin
  NumLines := EB2M^.GetNumLines;        {concatenate text from edit box}
  Index := 0;
  StrCopy(TheText, '');
  While (Index < NumLines) do
    begin
      if EB2M^.GetLine(TheLine, EB2M^.GetLineLength(Index)+3, Index)
        then StrCat(TheText, TheLine);
      Inc(Index);
      if (Index < NumLines)
        then StrCat(TheText, #13#10#0);
    end;
  TheContent^.SetText(TheText);          {save content node with new text}
  DBPut(TheContent);
  ErrorMsg(et_NOTICE, 'The revised text has been saved.');
  Done;
end;

procedure ModifyWin.IDBN5M(var Msg: TMessage);        {Cancel}
begin
  Done;
end;


        {****************************************
        *
        *       methods for TextWin
        *
  *       add an independent node if TheLinkedNode = nil
  *       add a new node linked to TheLinkNode if TheLinkedNode <> nil
        ****************************************}

constructor TextWin.Init(AParent: PWindowsObject; ALinkedNodeId: LongInt);
begin
  ControlWindow.Init(AParent, 'Add a Node');
  TheControlType := TEXTNODE;

  if (ALinkedNodeId = 0)
    then TheLinkedNode := nil
    else TheLinkedNode := PHermesNode(DBGet(ALinkedNodeId));

  with Attr do
    begin
      if (TheLinkedNode <> nil)
        then
          begin
            BN1T := New(PButton, Init(@Self, id_BN1T, 'Select Link Type', 40, 10, W-90, 20, False));
            ST1T := New(PStatic, Init(@Self, id_ST1T, '', 5, 35 , W-20, 20, 0));
          end;
```

```
      ST2T := New(PStatic, Init(@Self, id_ST2T, 'Name:', 20, 65 , W-20, 20, 0));
      EB1T := New(PEdit,   Init(@Self, id_EB1T, '', 5, 90, W-20, 25, 40, False));
      BN2T := New(PButton, Init(@Self, id_BN2T, 'Select Node Kind', 40, 120, W-90, 20, False));
      ST3T := New(PStatic, Init(@Self, id_ST3T, '', 5, 145, W-20, 20, 0));
      BN3T := New(PButton, Init(@Self, id_BN3T, 'Select context', 40, 165, W-90, 20, False));
      ST4T := New(PStatic, Init(@Self, id_ST4T, '', 5, 190 , W-20, 20, 0));
      ST5T := New(PStatic, Init(@Self, id_ST5T, 'Text:', 20, 220 , W-20, 20, 0));
      EB2T := New(PEdit,   Init(@Self, id_EB2T, '', 5, 245, W-20, 230, 4000, True));
      BN4T := New(PButton, Init(@Self, id_BN4T, 'Accept', 20, 490, (W div 2)-40, 20, False));
      BN5T := New(PButton, Init(@Self, id_BN5T, 'Cancel', (W div 2)+20, 490, (W div 2)-40, 20,
False));
    end;
end;

procedure TextWin.SetUpWindow;
begin
  ControlWindow.SetUpWindow;
  ST4T^.SetText(DBNameOf(HermesApp.GetContext));
end;

procedure TextWin.IDBN1T(var Msg: TMessage);        {Select Link Type}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_LinkType), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then ST1T^.SetText(TheSelection);
end;

procedure TextWin.IDBN2T(var Msg: TMessage);        {Select Node Kind}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_NodeKind), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then ST3T^.SetText(TheSelection);
end;

procedure TextWin.IDBN3T(var Msg: TMessage);        {Select Context}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DisplayContextTree, TheIndex, TheSelection, False);
  if (TheIndex > -1)
    then
      begin
        TheSelection := StripBlanks(TheSelection);
        ST4T^.SetText(TheSelection);
        HermesApp.SetContext(DBIdOf(TheSelection, ot_Context));
      end;
end;

procedure TextWin.IDBN4T(var Msg: TMessage);        {Accept}
var
  TheType    : StrName;
  TheName    : StrName;
  TheKind    : StrName;
  TheContext : StrName;
  TheLine    : array[0..400] of Char;
  TheText    : array[0..4000] of Char;
  TheNodeId  : LongInt;
  TheNamedNode   : PHermesNode;
  TheContentNode : PCharacterSimple;
  NumLines       : Integer;
  Index          : Integer;
begin
  EB1T^.GetText(TheName, 40);
  if (StrComp(TheName, '') = 0)
    then StrCopy(TheName, NoName);
  ST3T^.GetText(TheKind, 40);
```

```
  ST4T^.GetText(TheContext, 40);
  if (StrComp(TheContext, '') = 0)
    then StrCopy(TheContext, HermesUniversalContext);
  NumLines := EB2T^.GetNumLines;        {concatenate text from edit box}
  Index := 0;
  StrCopy(TheText, '');
  While (Index < NumLines) do
    begin
      if EB2T^.GetLine(TheLine, EB2T^.GetLineLength(Index)+3, Index)
        then StrCat(TheText, TheLine);
      Inc(Index);
      if (Index < NumLines)
        then StrCat(TheText, #13#10#0);
    end;

  TheContentNode := New(PCharacterSimple, Init(TheName, TheText));                    {save new node,
unlinked}
  TheNodeId := DBPutNamedObject(TheContentNode, DBIdOf(TheContext, ot_Context));
  if (StrComp(TheKind, '') <> 0)
    then
      begin
        TheNamedNode := PHermesNode(DBGet(TheNodeId));
        TheNamedNode^.SetKind(TheKind);
        DBPutName(TheNamedNode);
      end;

  if (TheLinkedNode = nil)
    then
        Done
    else                                                  {then link them up}
      begin
        ST1T^.GetText(TheType, 40);
        if (StrLen(TheType) = 0)
          then
              ErrorMsg(et_NOTICE, 'A Link Type must be specified before this can be accepted.')
          else
            begin
              CreateHypertextSwitch(0, TheLinkedNode^.GetId, TheNodeId, TheType, DBIdOf(TheContext,
ot_Context), 0, nil);
              ErrorMsg(et_NOTICE, 'The new node is saved.');
              Done;
            end;
      end;
end;

procedure TextWin.IDBN5T(var Msg: TMessage);        {Cancel}
begin
  Done;
end;


        {****************************************
        *
        *       methods for NotifyWin
        *
        ****************************************}

constructor NotifyWin.Init(AParent: PWindowsObject; ATitle: PChar; AMessage: PChar);
var
  TempStatic : PStatic;
begin
  ControlWindow.Init(AParent, ATitle);
  TheControlType := UNDEFINED;

  Attr.X := 100;
  Attr.Y := 10;
  Attr.H := 100;
  Attr.W := 100;
  TempStatic := New(PStatic, Init(@Self, 101, AMessage, 50, 30, 90, 30, StrLen(AMessage)));
end;


        {****************************************
        *
        *       methods for ListerWin
        *
```

```
        ****************************************}

constructor ListerWin.Init(AParent: PWindowsObject; Sorted: Boolean);
begin
  TDlgWindow.Init(AParent, 'LISTER');
  LB1L := New(PListBox, InitResource(@Self, id_LB1L));
  if Sorted
    then LB1L^.Attr.Style := LB1L^.Attr.Style or lbs_Sort
    else LB1L^.Attr.Style := LB1L^.Attr.Style and (not lbs_Sort);
end;

procedure   ListerWin.IDLB1L(var Msg: TMessage);
begin
  case Msg.lParamHi of
    lbn_SelChange :
      begin
      end;
    lbn_DblClk:
      begin
        TransferData(tf_GetData);
        if CanClose
          then EndDlg(id_Ok);
      end;
  end;   {case}
end;



end.      {**** unit HerCoWin  ****}
```

15. HerTxWin.pas define text windows

```
 {*****************************************
 *
 *      HerTxWin.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 ******************************************}

unit HerTxWin;

interface

uses OWindows, Objects, ODialogs, WinTypes,
     HerBasic, HerDataB, HerWorld, HerLangu,
     HerHyper, HerOGL1, HerOGL2, HerOGL3,
     Strings, OStdDlgs, WinProcs, HerPersp, HerMedia, HerLists;

{ This unit defines Text Display HermesChildWindows. They generally contain query results.
       SPECIFICATION:
}

const

  wm_QueryMsg = wm_User + 1;     {user message; must agree with HerCoWin const}

  id_LB1C = 5301;    {controls for CritiqueWin}

  id_BN1R = 5101;    {controls for ResultWin}
  id_BN2R = 5102;
  id_BN3R = 5103;
  id_BN4R = 5104;
  id_BN5R = 5105;
  id_ST1R = 5106;
  id_ST2R = 5107;
  id_ST3R = 5108;
  id_ST4R = 5109;
  id_ST5R = 5110;
  id_ST6R = 5111;
  id_LB1R = 5112;

  id_BN1V = 5201;    {controls for VCopyWin}
  id_BN2V = 5202;
  id_ST1V = 5205;
  id_ST2V = 5206;
  id_ST3V = 5207;
  id_LB1V = 5209;

type

       {***************************************
       *
       *    data structure for QueryWindow
       *
       ***************************************}

  PQueryWindow = ^QueryWindow;

       QueryWindow = object (TGWindow)
    constructor Init(AParent: PGWindow; ATitle: PChar);
    procedure AdjustScroller; virtual;
    procedure AdjustMappingRect; virtual;
    procedure WMSize         (var Msg: TMessage); virtual wm_First + wm_Size;
    procedure LeftButtonDown (var Msg: TMessage); virtual;
    procedure WMLButtonUp    (var Msg: TMessage); virtual wm_First + wm_LButtonUp;
    procedure WMMouseMove    (var Msg: TMessage); virtual wm_First + wm_MouseMove;
       end;


       {***************************************
       *
       *    data structure for TextualWindow
```

```
      *
      ****************************************}

PTextualWindow = ^TextualWindow;

      TextualWindow = object (THermesChildWindow)
  constructor Init(AParent: PWindowsObject; ATitle: PChar);
  procedure WMSize(var Msg: TMessage); virtual wm_First + wm_Size;
  procedure SizeChildren; virtual;
      end;


      {****************************************
      *
      *    data structure for CritiqueWin
      *
      ****************************************}

PCritiqueWin = ^CritiqueWin;

      CritiqueWin = object (TextualWindow)
  constructor Init(AParent: PWindowsObject; ADrawingId: LongInt);
  procedure SetUpWindow; virtual;
  procedure SizeChildren; virtual;
  procedure FillResultListBox; virtual;
private
  LB1C : PQueryWindow;
end;


      {****************************************
      *
      *    data structure for ResultWin
      *
      ****************************************}

PResultWin = ^ResultWin;

      ResultWin = object (TextualWindow)
  constructor Init(AParent: PWindowsObject; AQuery: PDataList);
  procedure SetUpWindow; virtual;
  procedure FillResultListBox; virtual;
  procedure IDBN1(var Msg: TMessage); virtual id_First + id_BN1R;
  procedure IDBN2(var Msg: TMessage); virtual id_First + id_BN2R;
  procedure IDBN3(var Msg: TMessage); virtual id_First + id_BN3R;
  procedure IDBN4(var Msg: TMessage); virtual id_First + id_BN4R;
  procedure IDBN5(var Msg: TMessage); virtual id_First + id_BN5R;
  procedure WMQueryMsg(var Msg: TMessage); virtual wm_First + wm_QueryMsg;
private
  TheQuery   : PDataList;
  TheContext : StrName;
  TheResult  : PHermesQueryList;
  BN1R : PButton;
  BN2R : PButton;
  BN3R : PButton;
  BN4R : PButton;
  BN5R : PButton;
  ST1R : PStatic;
  ST2R : PStatic;
  ST3R : PStatic;
  ST4R : PStatic;
  ST5R : PStatic;
  ST6R : PStatic;
  LB1R : PQueryWindow;
      end;


      {****************************************
      *
      *    data structure for VCopyWin
      *
      ****************************************}

PVCopyWin = ^VCopyWin;

      VCopyWin = object (TextualWindow)
```

```
    constructor Init(AParent: PWindowsObject);
    procedure SetUpWindow; virtual;
    procedure FillResultListBox; virtual;
    procedure IDBN1(var Msg: TMessage); virtual id_First + id_BN1V;
    procedure IDBN2(var Msg: TMessage); virtual id_First + id_BN2V;
  private
    TheStartNode : LongInt;
    TheResult    : PHermesQueryList;
    TheContext   : StrName;
    BN1V : PButton;
    BN2V : PButton;
    ST1V : PStatic;
    ST2V : PStatic;
    ST3V : PStatic;
    LB1V : PQueryWindow;
       end;

procedure DisplayQuery(AResultList: PHermesQueryList; AWindow: PQueryWindow);

{=====================================================================}

implementation

uses HerLinks, HerNodes, HerCoWin;

        {****************************************
        *
        *       Procedure WordWrap
        *
        ****************************************}

function WordWrap(OldPicture: PPicture; TheWidth: Integer; NextLinePt: PGPoint): PPicture;
var
  NewPicture : PPicture;
  TheLabel   : PLabel;
  Label2     : PLabel;
  TheRichText: PRichText;
  RichText2  : PRichText;
  TheText    : PChar;
  SplitPt    : Integer;
  TheLength  : Word;

      procedure WrapOne(TheGraphic: PGraphic); far;
      begin   {WrapOne}
        if (TypeOf(TheGraphic^) = TypeOf(TLabel))
          then
            begin
              TheLabel := PLabel(TheGraphic);
              while TheLabel^.TextLength > TheWidth do
                begin
                  TheLabel^.PositionAt(NextLinePt);
                  Label2 := New(PLabel, InitDefault);
                  TheLabel^.Split(TheWidth, Label2^);
                  TheLabel^.GetNextLinePos(NextLinePt^);
                  Label2^.PositionAt(NextLinePt);
                  NewPicture^.FastAdd(TheLabel);
                  TheLabel := Label2;
                end;
              NewPicture^.FastAdd(TheLabel);
            end
        else if (TypeOf(TheGraphic^) = TypeOf(TRichText))
          then
            begin
              TheRichText := PRichText(TheGraphic);
              while (TheRichText^.TextLength > TheWidth + 2) do
                begin
                  TheRichText^.PositionAt(NextLinePt);
                  RichText2 := New(PRichText, InitDefault);
                  TheLength := TheRichText^.TextLength;
                  SplitPt := TheWidth - 1;
                  TheRichText^.SplitWords(SplitPt, RichText2^);  {break between words}
                  TheRichText^.GetNextLinePos(NextLinePt^);
                  NewPicture^.FastAdd(TheRichText);
                  RichText2^.PositionAt(NextLinePt);
                  TheRichText := RichText2;
                end;
```

```
              TheRichText^.GetNextLinePos(NextLinePt^);
              NewPicture^.FastAdd(TheRichText);
            end
        else if (TypeOf(TheGraphic^) = TypeOf(TPicture))
          then PPicture(TheGraphic)^.ForEach(@WrapOne)
        else NewPicture^.FastAdd(TheGraphic);
      end;    {WrapOne}

begin   {WordWrap}
  NewPicture := New(PPicture, Init(tc_Tools));
  OldPicture^.ForEach(@WrapOne);
  NewPicture^.FindBounds;
  WordWrap := NewPicture;
end;     {WordWrap}


        {****************************************
        *
        *       Procedure DisplayQuery
        *
        ****************************************}

procedure DisplayQuery(AResultList: PHermesQueryList; AWindow: PQueryWindow);
var
  NextLineOrigin : PGPoint;
  DummyPict : PPicture;

      procedure InsertOne(Item: PQueryListItem); far;
      var
        TheDisplay : PPicture;
        TheNode    : PPersistentObject;
        OldContext : LongInt;
      begin  {InsertOne}
        TheNode := DBGet(Item^.GetDataId);
        if assigned(TheNode) then
          begin
            OldContext := HermesApp.GetContext;
            HermesApp.SetContextFast(Item^.GetContext);           {set context for this node}
            TheDisplay := TheNode^.DisplayContent;
            HermesApp.SetContextFast(OldContext);                 {restore context}
            if assigned(TheDisplay) then
              begin
                TheDisplay^.SetGraphicId(TheNode^.GetId);             {set the Display's GraphicId to
TheNode's Id}
                NextLineOrigin^.Build(5 + (20 * Item^.GetLevel), NextLineOrigin^.Y);     {indent}
                TheDisplay^.PositionAt(NextLineOrigin);               {position under last display}
                TheDisplay := WordWrap(TheDisplay, (AWindow^.Attr.W  *  2  div  TextPen_StdHeight),
NextLineOrigin);
                                                                      {wrap text to fit in window}
                TheDisplay^.SetGraphicId(TheNode^.GetId);             {set the Display's GraphicId to
TheNode's Id}
                AWindow^.Picture^.FastAdd(TheDisplay);
                if (TheDisplay^.Count > 0)
                  then NextLineOrigin^.Build(TheDisplay^.Left, TheDisplay^.Bottom + 20);
              end;
          end;
      end;   {InsertOne}

begin  {DisplayQuery}
  DummyPict := New(PPicture, InitDefault);
  DummyPict^.PositionAt(New(PGPoint, Init(25, 122)));
  AWindow^.Picture^.FastAdd(DummyPict);
  AWindow^.Picture^.FindBounds;                              {hack to work around bug in PositionAt}

  NextLineOrigin := New(PGPoint, Init(5,0));
  AResultList^.ForEach(@InsertOne);                           {insert AResultList items in AWindow}
  AWindow^.Picture^.FindBounds;
  AWindow^.Invalidate;
  AWindow^.AdjustMappingRect;
end;   {DisplayQuery}


        {****************************************
        *
        *       methods for QueryWindow
        *
```

```
        ****************************************}

constructor QueryWindow.Init(AParent: PGWindow; ATitle: PChar);
begin
  TGWindow.Init(AParent, '');
  Attr.Style := Attr.Style or ws_VScroll or ws_HScroll;
  SetGCursor(cs_finger);
  Scroller := New(PGScroller, Init(@Self, 8, 15, 200, 100));
  Scroller^.TrackMode := False;
  Scroller^.AutoMode := False;
end;

procedure QueryWindow.AdjustScroller;
var
  ARect : TMathRect;
begin
  ARect.InitDefault;
  Picture^.GetBoundsRect(ARect);
  Space^.SetWorldRect(@ARect);
  Scroller^.SetRange(ARect.Left + ARect.Width, ARect.Top + ARect.Height);
  Scroller^.ScrollTo(0, 0);
  ARect.Done;
end;

procedure QueryWindow.WMSize(var Msg: TMessage);
begin
  TGWindow.WMSize(Msg);
  AdjustMappingRect;
end;

procedure QueryWindow.AdjustMappingRect;
var
  DisplayRect : TMathRect;
begin
  DisplayRect.InitDefault;
  GetDisplayRect(DisplayRect);
  Space^.SetMappingRect(@DisplayRect);
  Space^.FitToRectangle(@DisplayRect);
  AdjustScroller;
end;

procedure QueryWindow.LeftButtonDown(var Msg: TMessage);
var
  SelectedGraphic : PGraphic;
  MousePt         : PGPoint;
begin
  if not IsLButtonDown
    then
      begin
        IsLButtonDown := True;
        SetCapture(HWindow);
        SelectedGraphic := nil;
        MousePt := New(PGPoint, Init(Msg.LParamLo, Msg.LParamHi));
        SelectedGraphic := Picture^.ThatContains(MousePt);
        Dispose(MousePt, Done);
        Port^.Associate(@Self);
        if (SelectedGraphic <> nil)
          then
            begin
              SelectedGraphic^.Invert(Port);
              Application^.MakeWindow(New(PNavigatorWin, Init(@Self,
SelectedGraphic^.GetGraphicId)));
              SelectedGraphic^.Invert(Port);
            end;
        Port^.Dissociate;
      end;
end;

procedure QueryWindow.WMLButtonUp(var Msg: TMessage);
begin
  if IsLButtonDown
    then
      begin
        IsLButtonDown := False;
        ReleaseCapture;
      end;
```

```
end;

procedure QueryWindow.WMMouseMove(var Msg: TMessage);
begin
  {do nothing}
end;


        {****************************************
        *
        *       methods for TextualWindow
        *
        ****************************************}

constructor TextualWindow.Init(AParent: PWindowsObject; ATitle: PChar);
var
  AppIcon : TIcon;
  ARect  : TRect;
  AThird : Integer;
begin
  THermesChildWindow.Init(AParent, ATitle);
  SetChildType(TEXTUAL);
  SizeChildren;
  SetGCursor(cs_finger);
  AppIcon.Init(0,0, 'TEXT_WINDOW');
  SetIcon(@AppIcon);
  AppIcon.Done;
  GetClientRect(Application^.MainWindow^.HWindow, ARect);
  AThird :=  (ARect.right - ARect.left) div 3;
  Attr.X :=  ARect.left + AThird - 60;
  Attr.Y :=  ARect.top + 1;
  Attr.W :=  AThird + 20;
  Attr.H :=  ARect.bottom - ARect.top - 2;
end;

procedure TextualWindow.WMSize(var Msg: TMessage);
begin
  THermesChildWindow.WMSize(Msg);
  SizeChildren;
end;

procedure TextualWindow.SizeChildren;
var
  ARect  : TRect;
  AThird : Integer;
begin
  GetClientRect(Application^.MainWindow^.HWindow, ARect);
  AThird :=  (ARect.right - ARect.left) div 3;
  Attr.X :=  ARect.left + AThird - 60;
  Attr.Y :=  ARect.top + 1;
  Attr.W :=  AThird + 20;
  Attr.H :=  ARect.bottom - ARect.top - 2;
end;


        {****************************************
        *
        *       methods for CritiqueWin
        *
        ****************************************}

constructor CritiqueWin.Init(AParent: PWindowsObject; ADrawingId: LongInt);
var
  ARect  : TRect;
begin
  TextualWindow.Init(AParent, 'Critique');
  GetClientRect(Application^.MainWindow^.HWindow, ARect);
  Attr.Y :=  (ARect.top + ARect.bottom) div 2;
  Attr.H :=  ARect.bottom - Attr.Y;
  LB1C := New(PQueryWindow,Init(@Self, ''));
  LB1C^.Attr.Style := ws_Child or ws_Visible or lbs_Notify or ws_VScroll or ws_HScroll or ws_Border;
  LB1C^.Attr.X := 3;
  LB1C^.Attr.Y := 3;
  LB1C^.Attr.W := Attr.W - 15;
  LB1C^.Attr.H := Attr.H - 30;
  LB1C^.SetGCursor(cs_Finger);
```

```
end;

procedure CritiqueWin.SizeChildren;
var
  ARect  : TRect;
begin
  GetClientRect(Application^.MainWindow^.HWindow, ARect);
  Attr.Y :=  (ARect.top + ARect.bottom) div 2;
  Attr.H :=  ARect.bottom - Attr.Y;
  LB1C := New(PQueryWindow,Init(@Self, ''));
  LB1C^.Attr.Style := ws_Child or ws_Visible or lbs_Notify or ws_VScroll or ws_HScroll or ws_Border;
  LB1C^.Attr.X := 3;
  LB1C^.Attr.Y := 3;
  LB1C^.Attr.W := Attr.W - 15;
  LB1C^.Attr.H := Attr.H - 30;
  LB1C^.SetGCursor(cs_Finger);
end;

procedure CritiqueWin.SetUpWindow;
begin  {SetUpWindow}
  TextualWindow.SetUpWindow;
  FillResultListBox;
end;   {SetUpWindow}

procedure CritiqueWin.FillResultListBox;
var
  TheResult : PHermesQueryList;
  TheQuery  : PDataList;
begin  {FillResultListBox}
  LB1C^.Picture^.Init(tc_Tools);
  TheQuery := PDataList(DBGetNamedObject('hermes_critique', ot_DataList, HermesUniversalContextId));
  TheResult := TheQuery^.Eval(nil);
  if (TheResult^.Count = 0)
    then TheResult := PDataList(DBGetNamedObject('hermes_null_critique', ot_DataList,
HermesUniversalContextId))^.Eval(nil);
  DisplayQuery(TheResult, LB1C);
end;   {FillResultListBox}


       {****************************************
       *
       *       methods for ResultWin
       *
       ****************************************}

constructor ResultWin.Init(AParent: PWindowsObject; AQuery: PDataList);
begin
  TextualWindow.Init(AParent, 'Design Rationale');
  BN1R := New(PButton, Init(@Self, id_BN1R, 'Select Results:', 10, 10, 100, 20, False));
  ST1R := New(PStatic, Init(@Self, id_ST1R, ' ', 120, 10 , Attr.W-125, 20, 0));
  BN2R := New(PButton, Init(@Self, id_BN2R, 'Select Query:', 10, 35, 100, 20, False));
  ST2R := New(PStatic, Init(@Self, id_ST2R, ' ', 120, 35 , Attr.W-125, 20, 0));
  BN3R := New(PButton, Init(@Self, id_BN3R, 'Create Query:', 10, 60, 100, 20, False));
  ST3R := New(PStatic, Init(@Self, id_ST3R, ' ', 120, 60 , Attr.W-125, 20, 0));
  ST4R := New(PStatic, Init(@Self, id_ST4R, ' ', 5, 85, Attr.W - 15, 40, 0));
  BN4R := New(PButton, Init(@Self, id_BN4R, 'Select Context:', 10, 130, 100, 20, False));
  ST5R := New(PStatic, Init(@Self, id_ST5R, ' ', 120, 130, Attr.W-135, 20, 0));
  LB1R := New(PQueryWindow,Init(@Self, ''));
  LB1R^.Attr.Style := ws_Child or ws_Visible or lbs_Notify or ws_VScroll or ws_HScroll or ws_Border;
  LB1R^.Attr.X := 5;
  LB1R^.Attr.Y := 155;
  LB1R^.Attr.W := Attr.W - 15;
  LB1R^.Attr.H := Attr.H - 215;
  LB1R^.SetGCursor(cs_Finger);
  ST6R := New(PStatic, Init(@Self, id_ST6R, ' ', 10, Attr.H-50, Attr.W-160, 20, 0));
  BN5R := New(PButton, Init(@Self, id_BN5R, 'Save Results', Attr.W-140, Attr.H-50, 120, 20, False));
  TheQuery := AQuery;
end;

procedure ResultWin.SetUpWindow;
begin  {SetUpWindow}
  TextualWindow.SetUpWindow;
  if HermesApp.GetContext > 0
    then StrCopy(TheContext, DBNameOf(HermesApp.GetContext))       {insert name of current
perspective in ST2}
    else StrCopy(TheContext, 'Hermes_Universal_Context');
```

```
  ST5R^.SetText(TheContext);
  TheResult := nil;
  if assigned(TheQuery)
    then
      begin
        ST2R^.SetText(TheQuery^.GetName);
        ST4R^.SetText(TheQuery^.HermesDisplay);
        FillResultListBox;
      end;
end;   {SetUpWindow}

procedure ResultWin.FillResultListBox;
var
  TheCount   : StrName;
begin  {FillResultListBox}
  if (TheQuery <> nil)
    then
      begin
        LB1R^.Picture^.Init(tc_Tools);
        TheResult := TheQuery^.Eval(nil);
        DisplayQuery(TheResult, LB1R);
        if (TheResult^.Count > 0)
          then
            begin
              Str(TheResult^.Count, TheCount);
              StrCat(TheCount, ' results');
            end
          else StrCopy(TheCount, 'There are no results');
        ST6R^.SetText(TheCount);
      end;
end;   {FillResultListBox}

procedure ResultWin.IDBN1(var Msg: TMessage);     {Select Results button}
var
  TheIndex     : Integer;
  TheSelection : PChar;
  TheResultList  : PResultList;
  TheCount     : StrName;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_ResultList), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then
      begin
        LB1R^.Picture^.Init(tc_Tools);
        TheResultList := PResultList(DBGetName(TheSelection, ot_ResultList));
        ST1R^.SetText(TheResultList^.GetName);
        TheResult := PHermesQueryList(TheResultList^.GetListOfData);
        DisplayQuery(TheResult, LB1R);
        if (TheResult^.Count > 0)
          then
            begin
              Str(TheResult^.Count, TheCount);
              StrCat(TheCount, ' results');
            end
          else StrCopy(TheCount, 'There are no results');
        ST6R^.SetText(TheCount);
      end;
end;

procedure ResultWin.IDBN2(var Msg: TMessage);     {Select Query button pressed}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;                                     {use FillLister from HerCoWin}
  FillLister(@Self, DBListNames(ot_DataList), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then
      begin
        TheQuery := PDataList(DBGetNamedObject(TheSelection, ot_ResultList, HermesApp.GetContext));
        ST2R^.SetText(TheQuery^.GetName);
        ST4R^.SetText(TheQuery^.HermesDisplay);
        FillResultListBox;
      end;
end;
```

```
procedure ResultWin.IDBN3(var Msg: TMessage);     {Create Query button pressed}
var
  TheDlg : PQueriesWin;
begin
  TheDlg := New(PQueriesWin, Init(@Self));
  if (Application^.ExecDialog(TheDlg) = id_Ok)       {sends a WMQueryMsg}
    then FillResultListBox;
end;

procedure ResultWin.WMQueryMsg(var Msg: TMessage);
begin
  TheQuery := PDataList(Msg.lParam);
  ST3R^.SetText(TheQuery^.GetName);
  ST4R^.SetText(TheQuery^.HermesDisplay);
end;

procedure ResultWin.IDBN4(var Msg: TMessage);     {Select Context button pressed}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;                                     {use FillLister from HerCoWin}
  FillLister(@Self, DisplayContextTree, TheIndex, TheSelection, False);
  if (TheIndex > -1)
    then
      begin
        TheSelection := StripBlanks(TheSelection);
        StrCopy(TheContext, TheSelection);
        ST5R^.SetText(TheContext);
        HermesApp.SetContext(DBIdOf(TheSelection, ot_Context));
        FillResultListBox;
      end;
end;

procedure ResultWin.IDBN5(var Msg: TMessage);     {Save Results button pressed}
var
  TheResultList : PResultList;
  TheName     : StrName;
begin
  TheResultList := New(PResultList, Init(NoName));
  TheResultList^.SetListOfData (TheResult);
  StrCopy(TheName, ' ');
  if (Application^.ExecDialog(New(PInputDialog, Init(@Self, 'Name for Results List',
                                        'Enter name: ', TheName, SizeOf(TheName)))) = id_OK)
    then
      begin
        TheResultList^.SetName(TheName);
        DBPutName(TheResultList);
      end;
end;


        {****************************************
        *
        *       methods for VCopyWin
        *
        ****************************************}

constructor VCopyWin.Init(AParent: PWindowsObject);
begin
  TextualWindow.Init(AParent, 'Virtual Copying Demo');
  BN1V := New(PButton, Init(@Self, id_BN1V, 'Start Node:', 10, 10, 100, 20, False));
  ST1V := New(PStatic, Init(@Self, id_ST1V, ' ', 120, 10 , Attr.W-125, 20, 0));
  BN2V := New(PButton, Init(@Self, id_BN2V, 'Context:', 10, 35, 80, 20, False));
  ST2V := New(PStatic, Init(@Self, id_ST2V, ' ', 100, 35, Attr.W-105, 20, 0));
  LB1V := New(PQueryWindow,Init(@Self, ''));
  LB1V^.Attr.Style := ws_Child or ws_Visible or lbs_Notify or ws_VScroll or ws_HScroll or ws_Border;
  LB1V^.Attr.X := 5;
  LB1V^.Attr.Y := 60;
  LB1V^.Attr.W := Attr.W - 17;
  LB1V^.Attr.H := Attr.H - 105;
  LB1V^.SetGCursor(cs_Finger);
  ST3V := New(PStatic, Init(@Self, id_ST3V, ' ', 10, Attr.H-45, Attr.W-20, 20, 0));
end;
```

```
procedure VCopyWin.SetUpWindow;
begin  {SetUpWindow}
  TextualWindow.SetUpWindow;
  if HermesApp.GetContext > 0
    then StrCopy(TheContext, DBNameOf(HermesApp.GetContext))       {insert name of current
perspective in ST2}
    else StrCopy(TheContext, 'Hermes_Universal_Context');
  ST2V^.SetText(TheContext);
  TheResult := nil;
  TheStartNode := 0;
end;   {SetUpWindow}

procedure VCopyWin.FillResultListBox;
var
  TheCount       : StrName;
  TheAssociation : PAssociation;
  TheDataList    : PDataListOf;
begin  {FillResultListBox}
  if (TheStartNode <> 0)
    then
      begin
        LB1V^.Picture^.Init(tc_Tools);                      {clear the Picture to background color}
        TheAssociation := PAssociation(DBGetNamedObject('hermes_link_tree', ot_Association,
HermesUniversalContextId));
        TheDataList := New(PDataListOf, Init(NoName, TheAssociation, New(PDataListId, Init(NoName,
TheStartNode))));
        TheResult := TheDataList^.Eval(nil);
        DisplayQuery(TheResult, LB1V);
        if (TheResult^.Count > 0)
          then
            begin
              Str(TheResult^.Count, TheCount);
              StrCat(TheCount, ' results');
            end
          else StrCopy(TheCount, 'There are no results');
        ST3V^.SetText(TheCount);
      end;
end;   {FillResultListBox}

procedure VCopyWin.IDBN1(var Msg: TMessage);     {Start Node button}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;
  FillLister(@Self, DBListNames(ot_Character), TheIndex, TheSelection, True);
  if (TheIndex > -1)
    then
      begin
        TheStartNode := DBIdOf(TheSelection, ot_Character);
        ST1V^.SetText(TheSelection);
        FillResultListBox;
      end;
end;

procedure VCopyWin.IDBN2(var Msg: TMessage);     {Context button pressed}
var
  TheIndex     : Integer;
  TheSelection : PChar;
begin
  TheIndex := -1;                                     {use FillLister from HerCoWin}
  FillLister(@Self, DisplayContextTree, TheIndex, TheSelection, False);
  if (TheIndex > -1)
    then
      begin
        TheSelection := StripBlanks(TheSelection);
        StrCopy(TheContext, TheSelection);
        ST2V^.SetText(TheContext);
        HermesApp.SetContext(DBIdOf(TheSelection, ot_Context));
        FillResultListBox;
      end;
end;


end.      {**** unit HerTxWin  ****}
```

16. HerRegis.pas register all objects for stream

```
 {*****************************************
 *
 *      HerRegis.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright 1993 by Gerry Stahl
 *      all rights reserved
 *****************************************}

unit HerRegis;

{ This unit registers objects for storage in the Hermes database object stream.
  Abstract objects do not have to be registered -- only objects and components
  that are instantiated and stored on the stream must be registered.}

interface

procedure HerRegister;

{=====================================================}

implementation

uses
  Objects,
  HerLists, HerBasic, HerLinks, HerNodes, HerHyper, HerMedia, HerLangu, HerTerms;

const
                  {*** unit HerLists objects ***}

RHermesObject: TStreamRec = (
  ObjType : 1000;                              {THermesObject 1000}
  VmtLink: Ofs(TypeOf(THermesObject)^);
  Load: @THermesObject.Load;
  Store: @THermesObject.Store
  );

RHermesNamedObject: TStreamRec = (
  ObjType : 1005;                              {THermesNamedObject 1005}
  VmtLink: Ofs(TypeOf(THermesNamedObject)^);
  Load: @THermesNamedObject.Load;
  Store: @THermesNamedObject.Store
  );

RLinkId: TStreamRec = (
  ObjType : 1010;                              {TLinkId 1010}
  VmtLink: Ofs(TypeOf(TLinkId)^);
  Load: @TLinkId.Load;
  Store: @TLinkId.Store
  );

RHermesList: TStreamRec = (
  ObjType : 1020;                              {THermesList 1020}
  VmtLink: Ofs(TypeOf(THermesList)^);
  Load: @THermesList.Load;
  Store: @THermesList.Store
  );

RHermesStack: TStreamRec = (
  ObjType : 1030;                              {THermesStack 1030}
  VmtLink: Ofs(TypeOf(THermesStack)^);
  Load: @THermesStack.Load;
  Store: @THermesStack.Store
  );

RHermesQueue: TStreamRec = (
  ObjType : 1040;                              {THermesQueue 1040}
  VmtLink: Ofs(TypeOf(THermesQueue)^);
  Load: @THermesQueue.Load;
  Store: @THermesQueue.Store
  );
```

```
RHermesTree: TStreamRec = (
  ObjType : 1050;                              {THermesTree 1050}
  VmtLink: Ofs(TypeOf(THermesTree)^);
  Load: @THermesTree.Load;
  Store: @THermesTree.Store
  );

RHermesSortedList: TStreamRec = (
  ObjType : 1060;                              {THermesSortedList 1060}
  VmtLink: Ofs(TypeOf(THermesSortedList)^);
  Load: @THermesSortedList.Load;
  Store: @THermesSortedList.Store
  );

RHermesItemsList: TStreamRec = (
  ObjType : 1070;                              {THermesItemsList 1070}
  VmtLink: Ofs(TypeOf(THermesItemsList)^);
  Load: @THermesItemsList.Load;
  Store: @THermesItemsList.Store
  );

RHermesQueryList: TStreamRec = (
  ObjType : 1080;                              {THermesQueryList 1080}
  VmtLink: Ofs(TypeOf(THermesQueryList)^);
  Load: @THermesQueryList.Load;
  Store: @THermesQueryList.Store
  );

RHermesSortedItemsList: TStreamRec = (
  ObjType : 1090;                              {THermesSortedItemsList 1090}
  VmtLink: Ofs(TypeOf(THermesSortedItemsList)^);
  Load: @THermesSortedItemsList.Load;
  Store: @THermesSortedItemsList.Store
  );

RLinkList: TStreamRec = (
  ObjType : 1100;                              {TLinkList 1110}
  VmtLink: Ofs(TypeOf(TLinkList)^);
  Load: @TLinkList.Load;
  Store: @TLinkList.Store
  );

RLinkTree: TStreamRec = (
  ObjType : 1120;                              {TLinkTree 1120}
  VmtLink: Ofs(TypeOf(TLinkTree)^);
  Load: @TLinkTree.Load;
  Store: @TLinkTree.Store
  );

RBehaviorListItem: TStreamRec = (
  ObjType : 1130;                              {TBehaviorListItem 1130}
  VmtLink: Ofs(TypeOf(TBehaviorListItem)^);
  Load: @TBehaviorListItem.Load;
  Store: @TBehaviorListItem.Store
  );

RBehavior: TStreamRec = (
  ObjType : 1140;                              {TBehavior 1140}
  VmtLink: Ofs(TypeOf(TBehavior)^);
  Load: @TBehavior.Load;
  Store: @TBehavior.Store
  );

RHermesPointList: TStreamRec = (
  ObjType : 1150;                              {THermesPointList 1150}
  VmtLink: Ofs(TypeOf(THermesPointList)^);
  Load: @THermesPointList.Load;
  Store: @THermesPointList.Store
  );


                  {*** unit HerBasic objects ***}

RActiveObject: TStreamRec = (
  ObjType : 2000;                              {TActiveObject 2000}
  VmtLink: Ofs(TypeOf(TActiveObject)^);
```

```
  Load: @TActiveObject.Load;
  Store: @TActiveObject.Store
  );

RContextsList: TStreamRec = (
  ObjType : 2010;                              {TContextsList 2010}
  VmtLink: Ofs(TypeOf(TContextsList)^);
  Load: @TContextsList.Load;
  Store: @TContextsList.Store
  );

RHermesPoint: TStreamRec = (
  ObjType : 2020;                              {THermesPoint 2020}
  VmtLink: Ofs(TypeOf(THermesPoint)^);
  Load: @THermesPoint.Load;
  Store: @THermesPoint.Store
  );

 RExtent: TStreamRec = (
  ObjType : 2030;                              {TExtent 2030}
  VmtLink: Ofs(TypeOf(TExtent)^);
  Load: @TExtent.Load;
  Store: @TExtent.Store
  );

RTransform: TStreamRec = (
  ObjType : 2040;                              {TTransform 2040}
  VmtLink: Ofs(TypeOf(TTransform)^);
  Load: @TTransform.Load;
  Store: @TTransform.Store
  );

RQueryListItem: TStreamRec = (
  ObjType : 2050;                              {TQueryListItem 2050}
  VmtLink: Ofs(TypeOf(TQueryListItem)^);
  Load: @TQueryListItem.Load;
  Store: @TQueryListItem.Store
  );

RAttribute: TStreamRec = (
  ObjType : 2060;                              {TAttribute 2060}
  VmtLink: Ofs(TypeOf(TAttribute)^);
  Load: @TAttribute.Load;
  Store: @TAttribute.Store
  );

RTextualAttribute: TStreamRec = (
  ObjType : 2065;                              {TTextualAttribute 2065}
  VmtLink: Ofs(TypeOf(TTextualAttribute)^);
  Load: @TTextualAttribute.Load;
  Store: @TTextualAttribute.Store
  );

RGraphicalAttribute: TStreamRec = (
  ObjType : 2070;                              {TGraphicalAttribute 2070}
  VmtLink: Ofs(TypeOf(TGraphicalAttribute)^);
  Load: @TGraphicalAttribute.Load;
  Store: @TGraphicalAttribute.Store
  );

RSublink: TStreamRec = (
  ObjType : 2080;                              {TSublink 2080}
  VmtLink: Ofs(TypeOf(TSublink)^);
  Load: @TSublink.Load;
  Store: @TSublink.Store
  );

RGraphicalSublink: TStreamRec = (
  ObjType : 2090;                              {TGraphicalSublink 2090}
  VmtLink: Ofs(TypeOf(TGraphicalSublink)^);
  Load: @TGraphicalSublink.Load;
  Store: @TGraphicalSublink.Store
  );


                  {*** unit HerLinks objects ***}
```

```
RContextLink: TStreamRec = (
  ObjType : 3010;                              {TContextLink 3010}
  VmtLink: Ofs(TypeOf(TContextLink)^);
  Load: @TContextLink.Load;
  Store: @TContextLink.Store
  );

RContentLink: TStreamRec = (
  ObjType : 3020;                              {TContentLink 3020}
  VmtLink: Ofs(TypeOf(TContentLink)^);
  Load: @TContentLink.Load;
  Store: @TContentLink.Store
  );

RGraphicLink: TStreamRec = (
  ObjType : 3030;                              {TGraphicLink 3030}
  VmtLink: Ofs(TypeOf(TGraphicLink)^);
  Load: @TGraphicLink.Load;
  Store: @TGraphicLink.Store
  );

RLink: TStreamRec = (
  ObjType : 3040;                              {TLink 3040}
  VmtLink: Ofs(TypeOf(TLink)^);
  Load: @TLink.Load;
  Store: @TLink.Store
  );

                  {*** unit HerNodes objects ***}

RHermesNode: TStreamRec = (
  ObjType : 3110;                              {THermesNode 3110}
  VmtLink: Ofs(TypeOf(THermesNode)^);
  Load: @THermesNode.Load;
  Store: @THermesNode.Store
  );

RHermesGraphic: TStreamRec = (
  ObjType : 3120;                              {THermesGraphic 3120}
  VmtLink: Ofs(TypeOf(THermesGraphic)^);
  Load: @THermesGraphic.Load;
  Store: @THermesGraphic.Store
  );

RPrivilegeObject: TStreamRec = (
  ObjType : 3130;                              {TPrivilegeObject 3130}
  VmtLink: Ofs(TypeOf(TPrivilegeObject)^);
  Load: @TPrivilegeObject.Load;
  Store: @TPrivilegeObject.Store
  );

RLogin: TStreamRec = (
  ObjType : 3140;                              {TLogin 3140}
  VmtLink: Ofs(TypeOf(TLogin)^);
  Load: @TLogin.Load;
  Store: @TLogin.Store
  );

RContext: TStreamRec = (
  ObjType : 3150;                              {TContext 3150}
  VmtLink: Ofs(TypeOf(TContext)^);
  Load: @TContext.Load;
  Store: @TContext.Store
  );

RNodeKind: TStreamRec = (
  ObjType : 3160;                              {TNodeKind 3160}
  VmtLink: Ofs(TypeOf(TNodeKind)^);
  Load: @TNodeKind.Load;
  Store: @TNodeKind.Store
  );

RLinkType: TStreamRec = (
  ObjType : 3170;                              {TLinkType 3170}
  VmtLink: Ofs(TypeOf(TLinkType)^);
```

```
  Load: @TLinkType.Load;
  Store: @TLinkType.Store
  );

RResultList: TStreamRec = (
  ObjType : 3180;                              {TResultList 3180}
  VmtLink: Ofs(TypeOf(TResultList)^);
  Load: @TResultList.Load;
  Store: @TResultList.Store
  );

RUnitCube: TStreamRec = (
  ObjType : 3190;                              {TUnitCube 3190}
  VmtLink: Ofs(TypeOf(TUnitCube)^);
  Load: @TUnitCube.Load;
  Store: @TUnitCube.Store
  );

RHermesPolyline: TStreamRec = (
  ObjType : 3200;                              {THermesPolyline 3200}
  VmtLink: Ofs(TypeOf(THermesPolyline)^);
  Load: @THermesPolyline.Load;
  Store: @THermesPolyline.Store
  );

RSweep: TStreamRec = (
  ObjType : 3210;                              {TSweep 3210}
  VmtLink: Ofs(TypeOf(TSweep)^);
  Load: @TSweep.Load;
  Store: @TSweep.Store
  );

                  {*** unit HerHyper objects ***}

RNodeContent: TStreamRec = (
  ObjType : 4000;                              {TNodeContent 4000}
  VmtLink: Ofs(TypeOf(TNodeContent)^);
  Load: @TNodeContent.Load;
  Store: @TNodeContent.Store
  );

RImage: TStreamRec = (
  ObjType : 4010;                              {TImage 4010}
  VmtLink: Ofs(TypeOf(TImage)^);
  Load: @TImage.Load;
  Store: @TImage.Store
  );

RPen: TStreamRec = (
  ObjType : 4020;                              {TPen 4020}
  VmtLink: Ofs(TypeOf(TPen)^);
  Load: @TPen.Load;
  Store: @TPen.Store
  );

RSound: TStreamRec = (
  ObjType : 4030;                              {TSound 4030}
  VmtLink: Ofs(TypeOf(TSound)^);
  Load: @TSound.Load;
  Store: @TSound.Store
  );

RVideo: TStreamRec = (
  ObjType : 4040;                              {TVideo 4040}
  VmtLink: Ofs(TypeOf(TVideo)^);
  Load: @TVideo.Load;
  Store: @TVideo.Store
  );

RAnimation: TStreamRec = (
  ObjType : 4050;                              {TAnimation 4050}
  VmtLink: Ofs(TypeOf(TAnimation)^);
  Load: @TAnimation.Load;
  Store: @TAnimation.Store
  );
```

```
(*
RComputedView: TStreamRec = (
  ObjType : 4060;                              {TComputedView 4060}
  VmtLink: Ofs(TypeOf(TComputedView)^);
  Load: @TComputedView.Load;
  Store: @TComputedView.Store
  );
*)

RDataList: TStreamRec = (
  ObjType : 4100;                              {TDataList 4100}
  VmtLink: Ofs(TypeOf(TDataList)^);
  Load: @TDataList.Load;
  Store: @TDataList.Store
  );

RAssociation: TStreamRec = (
  ObjType : 4110;                              {TAssociation 4110}
  VmtLink: Ofs(TypeOf(TAnimation)^);
  Load: @TAssociation.Load;
  Store: @TAssociation.Store
  );

RFilter: TStreamRec = (
  ObjType : 4120;                              {TFilter 4120}
  VmtLink: Ofs(TypeOf(TFilter)^);
  Load: @TFilter.Load;
  Store: @TFilter.Store
  );

RPredicate: TStreamRec = (
  ObjType : 4130;                              {TPredicate 4130}
  VmtLink: Ofs(TypeOf(TPredicate)^);
  Load: @TPredicate.Load;
  Store: @TPredicate.Store
  );

RCounter: TStreamRec = (
  ObjType : 4140;                              {TCounter 4140}
  VmtLink: Ofs(TypeOf(TCounter)^);
  Load: @TCounter.Load;
  Store: @TCounter.Store
  );

RMeasure: TStreamRec = (
  ObjType : 4150;                              {TMeasure 4150}
  VmtLink: Ofs(TypeOf(TMeasure)^);
  Load: @TMeasure.Load;
  Store: @TMeasure.Store
  );

RQuantifier: TStreamRec = (
  ObjType : 4160;                              {TQuantifier 4160}
  VmtLink: Ofs(TypeOf(TQuantifier)^);
  Load: @TQuantifier.Load;
  Store: @TQuantifier.Store
  );

                  {*** unit HerMedia objects ***}

RCharacterSimple: TStreamRec = (
  ObjType : 5010;                              {TCharacterSimple 5010}
  VmtLink: Ofs(TypeOf(TCharacterSimple)^);
  Load: @TCharacterSimple.Load;
  Store: @TCharacterSimple.Store
  );

RCharacterSubstring: TStreamRec = (
  ObjType : 5020;                              {TCharacterSubstring 5020}
  VmtLink: Ofs(TypeOf(TCharacterSubstring)^);
  Load: @TCharacterSubstring.Load;
  Store: @TCharacterSubstring.Store
  );

RCharacterAppend: TStreamRec = (
  ObjType : 5030;                              {TCharacterAppend 5030}
```

```
  VmtLink: Ofs(TypeOf(TCharacterAppend)^);
  Load: @TCharacterAppend.Load;
  Store: @TCharacterAppend.Store
  );

RNumberSimple: TStreamRec = (
  ObjType : 5040;                              {TNumberSimple 5040}
  VmtLink: Ofs(TypeOf(TNumberSimple)^);
  Load: @TNumberSimple.Load;
  Store: @TNumberSimple.Store
  );

RNumberCount: TStreamRec = (
  ObjType : 5050;                              {TNumberCount 5050}
  VmtLink: Ofs(TypeOf(TNumberCount)^);
  Load: @TNumberCount.Load;
  Store: @TNumberCount.Store
  );

RNumberMinimum: TStreamRec = (
  ObjType : 5060;                              {TNumberMinimum 5060}
  VmtLink: Ofs(TypeOf(TNumberMinimum)^);
  Load: @TNumberMinimum.Load;
  Store: @TNumberMinimum.Store
  );

RNumberMaximum: TStreamRec = (
  ObjType : 5070;                              {TNumberMaximum 5070}
  VmtLink: Ofs(TypeOf(TNumberMaximum)^);
  Load: @TNumberMaximum.Load;
  Store: @TNumberMaximum.Store
  );

RNumberTotal: TStreamRec = (
  ObjType : 5080;                              {TNumberTotal 5080}
  VmtLink: Ofs(TypeOf(TNumberTotal)^);
  Load: @TNumberTotal.Load;
  Store: @TNumberTotal.Store
  );

RNumberProduct: TStreamRec = (
  ObjType : 5090;                              {TNumberProduct 5090}
  VmtLink: Ofs(TypeOf(TNumberProduct)^);
  Load: @TNumberProduct.Load;
  Store: @TNumberProduct.Store
  );

RNumberSum: TStreamRec = (
  ObjType : 5100;                              {TNumberSum 5100}
  VmtLink: Ofs(TypeOf(TNumberSum)^);
  Load: @TNumberSum.Load;
  Store: @TNumberSum.Store
  );

RNumberDifference: TStreamRec = (
  ObjType : 5110;                              {TNumberDifference 5110}
  VmtLink: Ofs(TypeOf(TNumberDifference)^);
  Load: @TNumberDifference.Load;
  Store: @TNumberDifference.Store
  );

RNumberNegative: TStreamRec = (
  ObjType : 5120;                              {TNumberNegative 5120}
  VmtLink: Ofs(TypeOf(TNumberNegative)^);
  Load: @TNumberNegative.Load;
  Store: @TNumberNegative.Store
  );

RNumberTimes: TStreamRec = (
  ObjType : 5130;                              {TNumberTimes 5130}
  VmtLink: Ofs(TypeOf(TNumberTimes)^);
  Load: @TNumberTimes.Load;
  Store: @TNumberTimes.Store
  );

RNumberQuotient: TStreamRec = (
```

```
  ObjType : 5140;                              {TNumberQuotient 5140}
  VmtLink: Ofs(TypeOf(TNumberQuotient)^);
  Load: @TNumberQuotient.Load;
  Store: @TNumberQuotient.Store
  );

RNumberDistance: TStreamRec = (
  ObjType : 5150;                              {TNumberDistance 5150}
  VmtLink: Ofs(TypeOf(TNumberDistance)^);
  Load: @TNumberDistance.Load;
  Store: @TNumberDistance.Store
  );

RBooleanTrue: TStreamRec = (
  ObjType : 5160;                              {TBooleanTrue 5160}
  VmtLink: Ofs(TypeOf(TBooleanTrue)^);
  Load: @TBooleanTrue.Load;
  Store: @TBooleanTrue.Store
  );

RBooleanFalse: TStreamRec = (
  ObjType : 5170;                              {TBooleanFalse 5170}
  VmtLink: Ofs(TypeOf(TBooleanFalse)^);
  Load: @TBooleanFalse.Load;
  Store: @TBooleanFalse.Store
  );

RBooleanCounter: TStreamRec = (
  ObjType : 5180;                              {TBooleanCounter 5180}
  VmtLink: Ofs(TypeOf(TBooleanCounter)^);
  Load: @TBooleanCounter.Load;
  Store: @TBooleanCounter.Store
  );

RBooleanQuantifier: TStreamRec = (
  ObjType : 5190;                              {TBooleanQuantifier 5190}
  VmtLink: Ofs(TypeOf(TBooleanQuantifier)^);
  Load: @TBooleanQuantifier.Load;
  Store: @TBooleanQuantifier.Store
  );

RBooleanNot: TStreamRec = (
  ObjType : 5200;                              {TBooleanNot 5200}
  VmtLink: Ofs(TypeOf(TBooleanNot)^);
  Load: @TBooleanNot.Load;
  Store: @TBooleanNot.Store
  );

RBooleanConnective: TStreamRec = (
  ObjType : 5210;                              {TBooleanConnective 5210}
  VmtLink: Ofs(TypeOf(TBooleanConnective)^);
  Load: @TBooleanConnective.Load;
  Store: @TBooleanConnective.Store
  );

RBooleanMeasure: TStreamRec = (
  ObjType : 5220;                              {TBooleanMeasure 5220}
  VmtLink: Ofs(TypeOf(TBooleanMeasure)^);
  Load: @TBooleanMeasure.Load;
  Store: @TBooleanMeasure.Store
  );


                  {*** unit HerLangu objects ***}

RDataListNodeKind: TStreamRec = (
  ObjType : 6010;                              {TDataListNodeKind 6010}
  VmtLink: Ofs(TypeOf(TDataListNodeKind)^);
  Load: @TDataListNodeKind.Load;
  Store: @TDataListNodeKind.Store
  );

RDataListOf: TStreamRec = (
  ObjType : 6020;                              {TDataListOf 6020}
  VmtLink: Ofs(TypeOf(TDataListOf)^);
  Load: @TDataListOf.Load;
```

```
  Store: @TDataListOf.Store
  );

RDataListId: TStreamRec = (
  ObjType : 6030;                              {TDataListId 6030}
  VmtLink: Ofs(TypeOf(TDataListId)^);
  Load: @TDataListId.Load;
  Store: @TDataListId.Store
  );

RAssociationType: TStreamRec = (
  ObjType : 6040;                              {TAssociationType 6040}
  VmtLink: Ofs(TypeOf(TAssociationType)^);
  Load: @TAssociationType.Load;
  Store: @TAssociationType.Store
  );

RAssociationAll: TStreamRec = (
  ObjType : 6050;                              {TAssociationAll 6050}
  VmtLink: Ofs(TypeOf(TAssociationAll)^);
  Load: @TAssociationAll.Load;
  Store: @TAssociationAll.Store
  );

RAssociationOf: TStreamRec = (
  ObjType : 6060;                              {TAssociationOf 6060}
  VmtLink: Ofs(TypeOf(TAssociationOf)^);
  Load: @TAssociationOf.Load;
  Store: @TAssociationOf.Store
  );

RAssociationWith: TStreamRec = (
  ObjType : 6070;                              {TAssociationWith 6070}
  VmtLink: Ofs(TypeOf(TAssociationWith)^);
  Load: @TAssociationWith.Load;
  Store: @TAssociationWith.Store
  );

RAssociationComb: TStreamRec = (
  ObjType : 6080;                              {TAssociationComb 6080}
  VmtLink: Ofs(TypeOf(TAssociationComb)^);
  Load: @TAssociationComb.Load;
  Store: @TAssociationComb.Store
  );

(*
                  {*** unit HerTerms objects ***}

RCountOne: TStreamRec = (
  ObjType : 6080;                              {TCountOne 7010}
  VmtLink: Ofs(TypeOf(TCountOne)^);
  Load: @TCountOne.Load;
  Store: @TCountOne.Store
  );
*)

{REGISTER ALL OBJECTS}

procedure HerRegister;
begin
                                    {*** unit HerLists objects ***}
  RegisterType(RHermesObject);
  RegisterType(RHermesNamedObject);
  RegisterType(RLinkId);
  RegisterType(RHermesList);
  { RegisterType(RHermesOGLObject);}
  RegisterType(RHermesStack);
  RegisterType(RHermesQueue);
  RegisterType(RHermesTree);
  RegisterType(RHermesSortedList);
  RegisterType(RHermesItemsList);
  RegisterType(RHermesQueryList);
  RegisterType(RHermesSortedItemsList);
  RegisterType(RLinkList);
  RegisterType(RLinkTree);
  RegisterType(RBehaviorListItem);
```

```
  RegisterType(RBehavior);
  RegisterType(RHermesPointList);
                                    {*** unit HerBasic objects ***}
  RegisterType(RActiveObject);
  RegisterType(RContextsList);
  RegisterType(RHermesPoint);
  RegisterType(RExtent);
  RegisterType(RTransform);
  RegisterType(RQueryListItem);
  RegisterType(RAttribute);
  RegisterType(RTextualAttribute);
  RegisterType(RGraphicalAttribute);
  RegisterType(RSublink);
  RegisterType(RGraphicalSublink);

                                    {*** unit HerLinks objects ***}
{  RegisterType(RPersistentObject);
  RegisterType(RVCopyObject);
  RegisterType(RStampedObject);
  RegisterType(RLinkObject);   }
  RegisterType(RContextLink);
{  RegisterType(RMultiLink);   }
  RegisterType(RContentLink);
  RegisterType(RGraphicLink);
  RegisterType(RLink);
                                      {*** unit HerNodes objects ***}
{  RegisterType(RPrimitiveObject);
  RegisterType(RNodeObject);
  RegisterType(RPrimitiveNode);      }
  RegisterType(RHermesNode);
  RegisterType(RHermesGraphic);
  RegisterType(RPrivilegeObject);
  RegisterType(RLogin);
  RegisterType(RContext);
  RegisterType(RNodeKind);
  RegisterType(RLinkType);
  RegisterType(RResultList);
  {RegisterType(RGraphicContent);}
  RegisterType(RUnitCube);
  RegisterType(RHermesPolyline);
  RegisterType(RSweep);
                                      {*** unit HerHyper objects ***}
  {RegisterType(RTerminology);}
  RegisterType(RNodeContent);
  {RegisterType(RMediaElement);}
  {RegisterType(RLanguageElement);}
  {RegisterType(RTerminologyElement);}
  {RegisterType(RCharacter);}
  {RegisterType(RNumber);}
  {RegisterType(RBooleans);}
  RegisterType(RImage);
  RegisterType(RPen);
  RegisterType(RSound);
  RegisterType(RVideo);
  RegisterType(RAnimation);
  {RegisterType(RComputedView);}
  RegisterType(RDataList);
  RegisterType(RAssociation);
  RegisterType(RFilter);
  RegisterType(RPredicate);
  RegisterType(RCounter);
  RegisterType(RQuantifier);
  RegisterType(RMeasure);
                                    {*** unit HerMedia objects ***}
  RegisterType(RCharacterSimple);
  RegisterType(RCharacterSubstring);
  RegisterType(RCharacterAppend);
  RegisterType(RNumberSimple);
  RegisterType(RNumberCount);
  RegisterType(RNumberMinimum);
  RegisterType(RNumberMaximum);
  RegisterType(RNumberTotal);
  RegisterType(RNumberProduct);
  RegisterType(RNumberSum);
  RegisterType(RNumberDifference);
  RegisterType(RNumberNegative);
```

```
  RegisterType(RNumberTimes);
  RegisterType(RNumberQuotient);
  RegisterType(RNumberDistance);
  RegisterType(RBooleanTrue);
  RegisterType(RBooleanFalse);
  RegisterType(RBooleanCounter);
  RegisterType(RBooleanQuantifier);
  RegisterType(RBooleanNot);
  RegisterType(RBooleanConnective);
  RegisterType(RBooleanMeasure);
                                  {*** unit HerLangu objects ***}
  RegisterType(RDataListNodeKind);
  RegisterType(RDataListOf);
  RegisterType(RDataListId);
  RegisterType(RAssociationType);
  RegisterType(RAssociationAll);
  RegisterType(RAssociationOf);
  RegisterType(RAssociationWith);
  RegisterType(RAssociationComb);
end;  {HerRegister}


end.    {unit HerRegis}
```

```
 {*****************************************
 *
 *      HerSeeds.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 *****************************************}

 unit HerSeeds;

{  HerSeeds creates seed data for Hermes. }

interface

procedure  CreateSeed;

{=========================================================================}

implementation

uses HerLists, HerBasic, HerNodes, HerPersp, HerDataB, HerHyper, HerLangu;

        {****************************************
        *
        *   procedure DefineRationale;
        *
        ****************************************}

procedure DefineRationale;
begin
  CreateHypertextNode('issue for Lunar habitat design', 'What Should be the design of this lunar
habitat?',
                      'lunar habitat design task');

  AddRationale('issue for Lunar habitat design', 'issue 1', 'subissue', 'lunar habitat design task',
    'What should be considered in the design of the lunar habitat?');
  AddRationale('issue for Lunar habitat design', 'issue 2', 'subissue', 'lunar habitat design task',
    'Should the module be buried?');
  AddRationale('issue for Lunar habitat design', 'issue 3', 'subissue', 'lunar habitat design task',
    'What should be the design of the safe haven?');
  AddRationale('issue for Lunar habitat design', 'issue 4', 'subissue', 'lunar habitat design task',
    'Should space station nodes be reused or should lunar habitat nodes be specially designed?');
  AddRationale('issue for Lunar habitat design', 'issue 5', 'subissue', 'lunar habitat design task',
    'What should be the design of the cross section of the lunar habitat module?');
  AddRationale('issue for Lunar habitat design', 'issue 6', 'subissue', 'lunar habitat design task',
    'What should be the layout of the interior of the lunar habitat module?');

  AddRationale('issue 6', 'issue 6.1', 'subissue', 'lunar habitat design task',
    'What activities will take place in the interior of the lunar habitat module?');
  AddRationale('issue 6', 'issue 6.2', 'subissue', 'lunar habitat design task',
    'What functional areas are needed to support these activities?');
  AddRationale('issue 6', 'issue 6.3', 'subissue', 'lunar habitat design task',
    'What should be the design of the cross section of the lunar habitat module?');
  AddRationale('issue 6', 'issue 6.4', 'subissue', 'lunar habitat design task',
    'Should there be a corridor down the center of the lunar habitat module?');
  AddRationale('issue 6', 'issue 6.5', 'subissue', 'lunar habitat design task',
    'What should be the size of the equipment-module grid for the lunar habitat module?');
  AddRationale('issue 6', 'issue 6.6', 'subissue', 'lunar habitat design task',
    'What should be the arrangement and design of the functional areas and components of the lunar
habitat?');

  AddRationale('issue 6.6', 'issue 6.6.1', 'subissue', 'lunar habitat design task',
    'What should be the design for stowage?');
  AddRationale('issue 6.6', 'issue 6.6.2', 'subissue', 'lunar habitat design task',
    'What should be the design for sleeping?');
  AddRationale('issue 6.6', 'issue 6.6.3', 'subissue', 'lunar habitat design task',
    'What should be the design for crew stations?');
  AddRationale('issue 6.6', 'issue 6.6.4', 'subissue', 'lunar habitat design task',
    'What should be the design for the toilet and shower?');
  AddRationale('issue 6.6', 'issue 6.6.5', 'subissue', 'lunar habitat design task',
    'What should be the design for galley-wardroom?');
```

```
  AddRationale('issue 6.6.2', 'issue 6.6.2.1', 'subissue', 'lunar habitat design task',
    'Should there be a dedicated sleep area?');
  AddRationale('issue 6.6.2', 'issue 6.6.2.2', 'subissue', 'lunar habitat design task',
    'Where should the sleep area be located?');
  AddRationale('issue 6.6.2', 'issue 6.6.2.3', 'subissue', 'lunar habitat design task',
    'How should the crew stations and bunks be aligned?');
  AddRationale('issue 6.6.2', 'issue 6.6.2.4', 'subissue', 'lunar habitat design task',
    'What should be the design and layout of the bunks?');

  AddRationale('issue 6.6.2.4', 'issue 6.6.2.4.1', 'subissue', 'lunar habitat design task',
    'What should be the size of the bunks?');
  AddRationale('issue 6.6.2.4', 'issue 6.6.2.4.2', 'subissue', 'lunar habitat design task',
    'What should be the access to the bunks?');
  AddRationale('issue 6.6.2.4', 'issue 6.6.2.4.3', 'subissue', 'lunar habitat design task',
    'What should be the arrangement of the bunks?');

{define alternative perspectives on rationale}

  AddRationale('issue 6.6.2.4.3', 'answer 1', 'answer', 'clayton lewis context',
    'The bunks should be lined along the outer walls.');
  AddRationale('answer 1', 'argument 1.1', 'argument', 'clayton lewis context',
    'This arrangement provides easy access from the central corridor.');
  AddRationale('answer 1', 'argument 1.2', 'argument', 'clayton lewis context',
    'This arrangement keeps the central corridor open.');
  AddRationale('answer 1', 'argument 1.3', 'argument', 'clayton lewis context',
    'This arrangement allows bunks and crew stations to be aligned vertically.');
  AddRationale('issue 6.6.2.4.3', 'answer 2', 'answer', 'clayton lewis context',
    'The bunks should be oriented one way on one wall; the other on the other.');
  AddRationale('answer 2', 'argument 2.1', 'argument', 'clayton lewis context',
    'This arrangement provides privacy by placing sleeping heads apart.');

  ModifyRationale('issue 6.6.2.4.3', 'answer 1', 'answer', 'mike eisenberg context',
    'The bunks should be arranged in a pinwheel confiuration.');
  ModifyRationale('answer 1', 'argument 1.1', 'argument', 'mike eisenberg context',
    'This arrangement saves space and does not protrude over so much of the habitat.');
  ModifyRationale('answer 1', 'argument 1.2', 'argument', 'mike eisenberg context',
    'This arrangement allows for fuller separation of public and private spaces.');
  ModifyRationale('answer 1', 'argument 1.3', 'argument', 'mike eisenberg context',
    'The pinwheel does not have to extend over the galley-wardroom area.');
  ModifyRationale('issue 6.6.2.4.3', 'answer 2', 'answer', 'mike eisenberg context',
    'The bunks should be accessed through the center of the pinwheel.');
  ModifyRationale('answer 2', 'argument 2.1', 'argument', 'mike eisenberg context',
    'This arrangement minimizes the space reserved for access to the bunks.');

  ModifyRationale('issue 6.6.2.4.3', 'answer 1', 'answer', 'gerhard fischer context',
    'The bunks should be lined up across the habitat.');
  ModifyRationale('answer 1', 'argument 1.1', 'argument', 'gerhard fischer context',
    'This arrangement provides for easy expansion or contraction for more or fewer bunks.');
  ModifyRationale('answer 1', 'argument 1.2', 'argument', 'gerhard fischer context',
    'This arrangement creates more uniform spaces.');
  ModifyRationale('answer 1', 'argument 1.3', 'argument', 'gerhard fischer context',
    'Uniformity keeps things simpler for manufacture, redesign, maintenance.');
  ModifyRationale('issue 6.6.2.4.3', 'answer 2', 'answer', 'gerhard fischer context',
    'The bunks should be made narrower than the standard module.');
  ModifyRationale('answer 2', 'argument 2.1', 'argument', 'gerhard fischer context',
    'This saves wasted space and allows this arrangement to be as compact as the pinwheel.');


{MSIS IV}

  CreateHypertextNode('issues from MSIS',
    'What are the design considerations and requirements for development of manned space systems?',
'MSIS');

  AddRationale('issues from MSIS', 'issues from MSIS IV', 'subissue', 'MSIS',
    'What are the design considerations and requirements for Space Station Freedom?');
  AddRationale('issues from MSIS IV', 'MSIS IV 7.2.4', 'subissue', 'MSIS',
    'What are the design considerations and requirements for sleep?');
  AddRationale('MSIS IV 7.2.4', 'MSIS IV 7.2.4.2', 'subissue', 'MSIS',
    'What are the sleep design considerations?');
  AddRationale('MSIS IV 7.2.4', 'MSIS IV 7.2.4.3', 'subissue', 'MSIS',
    'What are the sleep design requirements?');
  AddRationale('MSIS IV 7.2.4.3', 'MSIS IV 7.2.4.3.a', 'answer', 'MSIS',
    'Facilities -- Adequate sleep facilities shall be provided.  See MSIS IV 10.4');
  AddRationale('MSIS IV 7.2.4.3', 'MSIS IV 7.2.4.3.b', 'answer', 'MSIS',
```

```
    'Duration -- Scheduling should allow a minimum sleep period of 8 hours per day with a' +
    ' minimum of 6 hours uninterrupted sleep.');
  AddRationale('MSIS IV 7.2.4.3', 'MSIS IV 7.2.4.3.c', 'answer', 'MSIS',
    'Pharmaceuticals -- Appropriate sleep aid medication shall be made available to' +
    ' crewmembers via a controlled access system.');

  AddRationale('issues from MSIS IV', 'MSIS IV 10.4', 'subissue', 'MSIS',
    'What are the design considerations and requirements for crew quarters?');
  AddRationale('MSIS IV 10.4', 'MSIS IV 10.4.3', 'subissue', 'MSIS',
    'What are the individual crew quarters design requirements?');
  AddRationale('MSIS IV 10.4.3', 'MSIS IV 10.4.3.1', 'answer', 'MSIS',
    'A dedicated, private crew quarter shall be provided for each crewmember.');
  AddRationale('MSIS IV 10.4.3', 'MSIS IV 10.4.3.2', 'subissue', 'MSIS',
    'What are the design requirements for one-person individual crew quarters?');
  AddRationale('MSIS IV 10.4.3.2', 'MSIS IV 10.4.3.2.a', 'answer', 'MSIS',
    'Communication -- Two-way audio/visual/data communications shall be provided between the crew
quarters, ' +
    'other station areas, and the ground.  The system shall have the capability of alerting the ' +
    'crew quaters occupant in an emergency.');
  AddRationale('MSIS IV 10.4.3.2', 'MSIS IV 10.4.3.2.b', 'answer', 'MSIS',
    'Environmental Controls -- Independent lighting, ventilation (direction and velocity), and' +
    ' temperature control shall be provided in the crew quarters and shall be adjustable from a sleep
restraint.');
  AddRationale('MSIS IV 10.4.3.2', 'MSIS IV 10.4.3.2.c', 'answer', 'MSIS',
    'Noise -- The noise levels in the crew quarters shall be as defined in MSIS IV 5.4.3.2.3.1.');
  AddRationale('MSIS IV 10.4.3.2', 'MSIS IV 10.4.3.2.d', 'answer', 'MSIS',
    'Movement -- The vibration and acceleration of the crew quarters shall be minimized to the
maximum extent possible.');
  AddRationale('MSIS IV 10.4.3.2', 'MSIS IV 10.4.3.2.e', 'answer', 'MSIS',
    'Stowage -- Facilities shall be provided in the crew quarters for stowing: Bedding, Clothing, and
Personal Items.');
  AddRationale('MSIS IV 10.4.3.2', 'MSIS IV 10.4.3.2.f', 'answer', 'MSIS',
    'Volume -- Sufficient volume shall be provided to meet functional and performance
requirements.');
  AddRationale('MSIS IV 10.4.3.2.f', 'MSIS IV 10.4.3.2.f.1', 'subanswer', 'MSIS',
    '1.50 cubic meters (53 cubic feet) for sleeping.');
  AddRationale('MSIS IV 10.4.3.2.f', 'MSIS IV 10.4.3.2.f.2', 'subanswer', 'MSIS',
    '0.63 cubic meters (22 cubic feet) for temporary and permanent stowage of operational and
personal equipment.');
  AddRationale('MSIS IV 10.4.3.2.f', 'MSIS IV 10.4.3.2.f.3', 'subanswer', 'MSIS',
    '1.19 cubic meters (42 cubic feet) for donning and doffing clothing.');
  AddRationale('MSIS IV 10.4.3.2.f', 'MSIS IV 10.4.3.2.f.4', 'subanswer', 'MSIS',
    'Additional free volume, as necessary, for using a desk, computer/communication syste, trash
stowage, personal grooming,' +
    ' convalescence, off-duty activities, and access to stowage or equipment without interference to
or from permanently' +
    ' mounted or temporarily stowed hardware.');
  AddRationale('MSIS IV 10.4.3.2.f', 'MSIS IV 10.4.3.2.f.5', 'subanswer', 'MSIS',
    'The internal dimensions of the crew quarters shall be sufficient to accomodate the largest
crewmember body size' +
    ' under consideration.');
  AddRationale('MSIS IV 10.4.3.2', 'MSIS IV 10.4.3.2.g', 'answer', 'MSIS',
    'Exit and Entry -- The opening shall be sufficiently large to allow contingency entry by an EVA
suited crewmember.' +
    ' Doors shall meet the requirements of MSIS IV 8.10.3.');
  AddRationale('MSIS IV 10.4.3.2', 'MSIS IV 10.4.3.2.h', 'answer', 'MSIS',
    'Privacy -- The individual cew quarters shall provide visual privacy to and from the occupant and
acoustic privacy' +
    ' as defined in MSIS IV 5.4.3.2.3.1.');
  AddRationale('MSIS IV 10.4.3.2', 'MSIS IV 10.4.3.2.i', 'answer', 'MSIS',
    'Restraints -- Restraints shall be provided as necessary for activities such as sleeping,
dressing, recreation,' +
    ' and cleaning.');
  AddRationale('MSIS IV 10.4.3.2', 'MSIS IV 10.4.3.2.j', 'answer', 'MSIS',
    'Locks -- Individual crew quarters doors shall be equiped with locks allowing the occupant(s) to
lock and' +
    ' unlock the door from inside.  Provisions shall be made for emergency override.');

  AddRationale('issues from MSIS IV', 'MSIS IV 8.10', 'subissue', 'MSIS',
    'What are the design considerations and requirements for hatches and doors?');
  AddRationale('MSIS IV 8.10', 'MSIS IV 8.10.3', 'subissue', 'MSIS',
    'What are the hatch and door design requirements?');
  AddRationale('MSIS IV 8.10.3', 'MSIS IV 8.10.3.1', 'subissue', 'MSIS',
    'What are the location design requirements?');
  AddRationale('MSIS IV 8.10.3.1', 'MSIS IV 8.10.3.1.a', 'subissue', 'MSIS',
```

```
    'Internal Door Placement -- Enclosed crew stations shall have entrances or exits to permit
unrestricted flow'  +
    ' for all anticipated traffic.  They shall be located so personnel who are entering or leaving
will not' +
    ' interfere with operations or traffic flow.');
  AddRationale('MSIS IV 8.10.3.1', 'MSIS IV 8.10.3.1.b', 'subissue', 'MSIS',
    'Away From Hazards -- In compartments with a single ingress/egress, the opening shall not be
located near flammable,' +
    ' explosive, or other hazards such that the energy content, if released, will result in damage
which prevents access' +
    ' through the entrance.');
  AddRationale('MSIS IV 8.10.3.1', 'MSIS IV 8.10.3.1.c', 'subissue', 'MSIS',
    'Emergency Passage -- Capability shall be provided to allow emergency exit and rescue entry into
a compartment.');
  AddRationale('MSIS IV 8.10.3.1', 'MSIS IV 8.10.3.1.d', 'subissue', 'MSIS',
    'External Pressure Hatches -- Hatches opening directly into space vacuum shall be self-sealing
(e.g., inward opening).');
  AddRationale('MSIS IV 8.10.3', 'MSIS IV 8.10.3.2', 'subissue', 'MSIS',
    'What are the pressure hatch indicator and visual display design requirements?');
  AddRationale('MSIS IV 8.10.3', 'MSIS IV 8.10.3.3', 'subissue', 'MSIS',
    'What are the opening and closing mechanism design requirements?');
  AddRationale('MSIS IV 8.10.3', 'MSIS IV 8.10.3.4', 'subissue', 'MSIS',
    'What are the operating forces design requirements?');
  AddRationale('MSIS IV 8.10.3', 'MSIS IV 8.10.3.5', 'subissue', 'MSIS',
    'What are the minimum size design requirements?');
  AddRationale('MSIS IV 8.10.3', 'MSIS IV 8.10.3.6', 'subissue', 'MSIS',
    'What are the operations interface design requirements?');
  AddRationale('MSIS IV 8.10.3', 'MSIS IV 8.10.3.7', 'subissue', 'MSIS',
    'What are the shape design requirements?');

  AddRationale('issues from MSIS IV', 'MSIS IV 5.4.3.2.3.1', 'subissue', 'MSIS',
    'What are the design requirements for wide-band, long-term annoyance noise exposure?');
  AddRationale('MSIS IV 5.4.3.2.3.1', 'MSIS IV 5.4.3.2.3.1.a', 'answer', 'MSIS',
    'Maximum Continuous Noise -- The maximum allowable broad band sound pressure levels produced by
the summation of all the' +
    ' individual sound pressure levels from all operating systems and subsystems considered at a
given time shall not exceed' +
    ' the Noise Criteria (NC) 50 contour for work periods and the NC 40 contour for sleep
compartments.');
  AddRationale('MSIS IV 5.4.3.2.3.1', 'MSIS IV 5.4.3.2.3.1.b', 'answer', 'MSIS',
    'Sleep Compartment Noise Level -- ' +
    ' 1. In sleep areas, the contiuous broadband noise level shall not be less than the NC 25
contour.' +
    ' 2. Hearing protection devices shall be available in sleep areas to provide aural isolation as
needed.');
  AddRationale('MSIS IV 5.4.3.2.3.1', 'MSIS IV 5.4.3.2.3.1.c', 'answer', 'MSIS',
   'The A-weighted sound pressure level for any given NC curve shall not exceed the level shown in
the chart.');

  CreateHypertextLink('', 'MSIS IV 7.2.4.3.a',  'MSIS IV 10.4',        'reference', 'MSIS');
  CreateHypertextLink('', 'MSIS IV 10.4.3.2.c', 'MSIS IV 5.4.3.2.3.1', 'reference', 'MSIS');
  CreateHypertextLink('', 'MSIS IV 10.4.3.2.g', 'MSIS IV 8.10.3',      'reference', 'MSIS');
  CreateHypertextLink('', 'MSIS IV 10.4.3.2.h', 'MSIS IV 5.4.3.2.3.1', 'reference', 'MSIS');
end;

        {***************************************
        *
        *   procedure DefineContextTree;
        *
        ***************************************}

procedure DefineContextTree;
var
  TheContext : PContext;
  TheLogin   : PLogin;
begin
  CopyContextNames(HermesUniversalContext, 'virtual copying sample data');
  CopyContextNames('virtual copying sample data', 'a');
  CopyContextNames('a', 'b');
  CopyContextNames('b', 'c');
  CopyContextNames('b', 'd');
  CopyContextNames('d', 'h');
  CopyContextNames('virtual copying sample data', 'e');
  CopyContextNames('e', 'f');
  CopyContextNames('e', 'g');
  CopyContextNames('f', 'd');
```

```
CopyContextNames('c', 'i');
CopyContextNames('d', 'i');

CopyContextNames(HermesUniversalContext, 'MSIS');
CopyContextNames('MSIS', 'lunar habitat design task');
CopyContextNames('lunar habitat design task', 'computer science hackers');
CopyContextNames('computer science hackers', 'Clayton Lewis context');
CopyContextNames('Clayton Lewis context', 'Mike Eisenberg context');
CopyContextNames('Mike Eisenberg context', 'Gerhard Fischer context');
CopyContextNames('Gerhard Fischer context', 'Gerry Stahl context');
CopyContextNames('lunar habitat design task', 'environmental designers');
CopyContextNames('environmental designers', 'Mark Gross context');
CopyContextNames('environmental designers', 'Ray McCall context');
CopyContextNames('Ray McCall context', 'Gerry Stahl context');

CopyContextNames(HermesUniversalContext, 'context for misc explorations');

{create 'guest' Login and Context}
TheLogin := New(PLogin, Init('guest'));
TheLogin^.SetPassword('guest');
DBPutName(TheLogin);
TheContext := New(PContext, Init('guest'));
TheContext^.SetPassword('guest');
TheCOntext^.SetVEdit(True);
DBPutName(TheContext);
CopyContextNames(HermesUniversalContext, 'guest');

TheLogin := New(PLogin, Init('nasa'));
TheLogin^.SetPassword('nasa');
DBPutName(TheLogin);
TheContext := New(PContext, Init('nasa'));
TheContext^.SetPassword('nasa');
TheCOntext^.SetVEdit(False);
DBPutName(TheContext);
CopyContextNames(HermesUniversalContext, 'nasa');

TheLogin := New(PLogin, Init('gerry'));
TheLogin^.SetPassword('gerry');
DBPutName(TheLogin);
TheContext := New(PContext, Init('gerry'));
TheContext^.SetPassword('gerry');
TheCOntext^.SetVEdit(False);
DBPutName(TheContext);
CopyContextNames(HermesUniversalContext, 'gerry');

TheLogin := New(PLogin, Init('ray'));
TheLogin^.SetPassword('ray');
DBPutName(TheLogin);
TheContext := New(PContext, Init('ray'));
TheContext^.SetPassword('ray');
TheCOntext^.SetVEdit(True);
DBPutName(TheContext);
CopyContextNames('gerry', 'ray');

TheLogin := New(PLogin, Init('john'));
TheLogin^.SetPassword('john');
DBPutName(TheLogin);
TheContext := New(PContext, Init('john'));
TheContext^.SetPassword('john');
TheCOntext^.SetVEdit(True);
DBPutName(TheContext);
CopyContextNames('gerry', 'john');

TheLogin := New(PLogin, Init('pat'));
TheLogin^.SetPassword('pat');
DBPutName(TheLogin);
TheContext := New(PContext, Init('pat'));
TheContext^.SetPassword('pat');
TheCOntext^.SetVEdit(True);
DBPutName(TheContext);
CopyContextNames('ray', 'pat');

TheLogin := New(PLogin, Init('erik'));
TheLogin^.SetPassword('erik');
DBPutName(TheLogin);
TheContext := New(PContext, Init('erik'));
```

```
  TheContext^.SetPassword('erik');
  TheCOntext^.SetVEdit(True);
  DBPutName(TheContext);
  CopyContextNames('ray', 'erik');

  TheLogin := New(PLogin, Init('dale'));
  TheLogin^.SetPassword('dale');
  DBPutName(TheLogin);
  TheContext := New(PContext, Init('dale'));
  TheContext^.SetPassword('dale');
  TheCOntext^.SetVEdit(True);
  DBPutName(TheContext);
  CopyContextNames('john', 'dale');

  TheLogin := New(PLogin, Init('nancy'));
  TheLogin^.SetPassword('nancy');
  DBPutName(TheLogin);
  TheContext := New(PContext, Init('nancy'));
  TheContext^.SetPassword('nancy');
  TheCOntext^.SetVEdit(True);
  DBPutName(TheContext);
  CopyContextNames('john', 'nancy');

end;

        {****************************************
        *
        *   procedure DefineHypertextTree;
        *
        ****************************************}

procedure DefineHypertextTree;
begin
  CreateHypertextNode('one', 'Node one in context a.', 'a');
  CreateHypertextNode('two', 'Node two in context a.', 'a');
  CreateHypertextNode('three', 'Node three in context a.', 'a');
  CreateHypertextNode('four', 'Node four in context a.', 'a');
  CreateHypertextNode('five', 'Node five in context b.', 'b');
  CreateHypertextNode('six', 'Node six in context b.', 'b');
  CreateHypertextNode('seven', 'Node seven in context a.', 'a');
  CreateHypertextNode('eight', 'Node eight in context a.', 'a');
  CreateHypertextNode('nine', 'Node nine in context a.', 'a');
  CreateHypertextNode('ten', 'Node ten in context g.', 'g');
  CreateHypertextNode('eleven', 'Node eleven in context a.', 'a');
  CreateHypertextNode('twelve', 'Node twelve in context c.', 'c');
  CreateHypertextNode('thirteen', 'Node thirteen in context c.', 'c');
  CreateHypertextNode('fourteen', 'Node fourteen in context c.', 'c');
  CreateHypertextNode('fifteen',   'Node fifteen in context d.', 'd');
  CreateHypertextNode('sixteen',   'Node sixteen in context d.', 'd');
  CreateHypertextNode('twentyone', 'Node twentyone in context g.', 'g');
  CreateHypertextNode('seventeen', 'Node seventeen in context e.', 'e');
  CreateBooleanNode  ('eighteen',  'e');
  CreateHypertextNode('nineteen',  'Node nineteen in context f.', 'f');
  CreateNumericNode  ('twenty',    'e');
  CreateHypertextLink('link_one', 'one', 'two', 'hermes_link', 'a');
  CreateHypertextLink('link_two', 'one', 'three', 'hermes_link', 'a');
  CreateHypertextLink('link_three', 'two', 'four', 'hermes_link', 'a');   {link up the Hypertexts}
  CreateHypertextLink('link_four', 'two', 'five', 'hermes_link', 'b');
  CreateHypertextLink('link_five', 'three', 'six', 'hermes_link', 'b');
  CreateHypertextLink('link_six', 'three', 'seven', 'hermes_link', 'a');
  CreateHypertextLink('link_seven', 'three', 'eight', 'hermes_link', 'a');
  CreateHypertextLink('link_eight', 'four', 'nine', 'hermes_link', 'c');
  CreateHypertextLink('link_nine', 'four', 'ten', 'hermes_link', 'b');
  CreateHypertextLink('link_ten', 'four', 'eleven', 'hermes_link', 'a');
  CreateHypertextLink('link_eleven', 'ten', 'twelve', 'hermes_link', 'c');
  CreateHypertextLink('link_twelve', 'one', 'thirteen', 'hermes_link', 'a');
  CreateHypertextLink('link_thirteen', 'three', 'fourteen', 'hermes_link', 'c');
  CreateHypertextLink('link_fourteen', 'two', 'fifteen', 'hermes_link', 'd');
  CreateHypertextLink('link_fifteen', 'five', 'sixteen', 'hermes_link', 'd');
  CreateHypertextSwitch(0, DBIdOf('seventeen', ot_Character),
                  DBIdOf('eighteen', ot_Booleans), 'hermes_link', DBIdOf('e', ot_Context), 0, nil);
  CreateHypertextLink('link_sixteen', 'seventeen', 'nineteen', 'hermes_link', 'e');
  CreateHypertextSwitch(0, DBIdOf('eighteen', ot_Booleans),
                  DBIdOf('twenty', ot_Number), 'hermes_link', DBIdOf('e', ot_Context), 0, nil);
  CreateHypertextSwitch(0, DBIdOf('twenty', ot_Number),
```

```
                    DBIdOf('twentyone', ot_Character), 'hermes_link', DBIdOf('g', ot_Context), 0,
nil);
  CreateHypertextSwitch(DBIdOf('switch_one', ot_Link), DBIdOf('ten', ot_Character),
DBIdOf('seventeen', ot_Character),
                    'hermes_link', DBIdOf('d', ot_Context), DBIdOf('f', ot_Context), nil);
  CreateHypertextSwitch(DBIdOf('switch_two', ot_Link), DBIdOf('fifteen', ot_Character),
DBIdOf('seventeen', ot_Character),
                    'hermes_link', DBIdOf('d', ot_Context), DBIdOf('g', ot_Context), nil);

{COMMENT OUT here to eliminate some tests}

  TestModifyNode(DBIdOf('four', ot_Character), DBIdOf('b', ot_Context));
  DeleteNode(DBIdOf('three', ot_Character), DBIdOf('b', ot_Context));
(*  VCopyNode(DBIdOf('thirteen', ot_Character), DBIdOf('c', ot_Context), DBIdOf('d', ot_Context));
  VCopySubtree(DBIdOf('one', ot_Character), DBIdOf('d', ot_Context), DBIdOf('g', ot_Context));
  PCopyNode(DBIdOf('nineteen', ot_Character), DBIdOf('f', ot_Context), DBIdOf('g', ot_Context));
  DeleteLink(DBIdOf('link_four', ot_Link), DBIdOf('d', ot_Context));
  ModifyLink(DBIdOf('link_three', ot_Link), DBIdOf('d', ot_Context));

  VCopyLazySubtree(DBIdOf('one', ot_Character), DBIdOf('c', ot_Context), DBIdOf('d', ot_Context));
*)
  CreateTextContent(DBIdOf('two', ot_Character), 'more content', DBIdOf('c', ot_Context));
  CreateTextContent(DBIdOf('two', ot_Character), 'yet more content', DBIdOf('c', ot_Context));
  CreateTextContent(DBIdOf('two', ot_Character), 'even more content', DBIdOf('c', ot_Context));

end;

       {****************************************
       *
       *   procedure DefineTypes;
       *
       ****************************************}

procedure DefineTypes;
var
  TheAssociation : PAssociation;
  TheDataList    : PDataList;
begin
  CreateContextNode(HermesUniversalContext);
  if (HermesUniversalContextId = 0)
    then HermesUniversalContextId := DBIdOf(HermesUniversalContext, ot_Context);

  CreateType(HermesContentLinkType);     {hermes_content}
  CreateType(HermesContextLinkType);     {hermes_context}
  CreateType(HermesGraphicLinkType);     {hermes_graphic}
  CreateType('hermes_link');
  CreateType('hermes_is_a');
  CreateType('hermes_critic');
  CreateType('hermes_adjacent_to');
  CreateType('hermes_is_part_of');
  CreateType('hermes_is_also_a');
  CreateType('hermes_consists_of');
  CreateType('hermes_scalable');
  CreateType('hermes_auto_critic');
  CreateType('issue');                              {PHI Types}
  CreateType('answer');
  CreateType('argument');
  CreateType('chair');
  CreateType('table');
  CreateType('habitat');
  DBAddSynonym('issues', 'issue', ot_LinkType);
  DBAddSynonym('subissue', 'issue', ot_LinkType);
  DBAddSynonym('subissues', 'issue', ot_LinkType);
  DBAddSynonym('answers', 'answer', ot_LinkType);
  DBAddSynonym('subanswer', 'answer', ot_LinkType);
  DBAddSynonym('subanswers', 'answer', ot_LinkType);
  DBAddSynonym('arguments', 'argument', ot_LinkType);
  DBAddSynonym('subargument', 'argument', ot_LinkType);
  DBAddSynonym('subarguments', 'argument', ot_LinkType);
  DBAddSynonym('habitats', 'habitat', ot_LinkType);
  DBAddSynonym('chairs', 'chair', ot_LinkType);
  DBAddSynonym('tables', 'table', ot_LinkType);
  CreateType('annotation');
  CreateType('author');
  CreateType('objection');
  CreateType('critique');
```

```
CreateType('generalization');
CreateType('specification');
CreateType('reference');
CreateType('replacement');
CreateType('summary');
CreateType('keyword');
CreateType('status');
CreateType('consequence');
CreateType('precondition');
CreateType('argument for');
CreateType('argument against');
DBAddSynonym('question', 'issue', ot_LinkType);            {IBIS Types}
DBAddSynonym('questions', 'issue', ot_LinkType);
DBAddSynonym('position', 'answer', ot_LinkType);
DBAddSynonym('positions', 'answer', ot_LinkType);
DBAddSynonym('decision', 'answer', ot_LinkType);
DBAddSynonym('decisions', 'answer', ot_LinkType);
DBAddSynonym('response', 'answer', ot_LinkType);
DBAddSynonym('responses', 'answer', ot_LinkType);
DBAddSynonym('support', 'argument', ot_LinkType);
DBAddSynonym('supports', 'argument', ot_LinkType);
DBAddSynonym('comment', 'annotation', ot_LinkType);
DBAddSynonym('comments', 'annotation', ot_LinkType);
DBAddSynonym('suggestion', 'annotation', ot_LinkType);
DBAddSynonym('annotations', 'annotation', ot_LinkType);
DBAddSynonym('suggestions', 'annotation', ot_LinkType);
DBAddSynonym('authors', 'author', ot_LinkType);
DBAddSynonym('objections', 'objection', ot_LinkType);
DBAddSynonym('critiques', 'critique', ot_LinkType);
DBAddSynonym('generalizations', 'generalization', ot_LinkType);
DBAddSynonym('specifications', 'specification', ot_LinkType);
DBAddSynonym('references', 'reference', ot_LinkType);
DBAddSynonym('replacements', 'replacement', ot_LinkType);
DBAddSynonym('summaries', 'summary', ot_LinkType);
DBAddSynonym('keywords', 'keyword', ot_LinkType);
DBAddSynonym('statuses', 'status', ot_LinkType);
DBAddSynonym('consequences', 'consequence', ot_LinkType);
DBAddSynonym('preconditions', 'precondition', ot_LinkType);
DBAddSynonym('arguments for', 'argument for', ot_LinkType);
DBAddSynonym('arguments against', 'argument against', ot_LinkType);
DBAddSynonym('pro', 'argument for', ot_LinkType);
DBAddSynonym('con', 'argument against', ot_LinkType);

CreateKind(HermesContextNodeKind);      {hermes_context}
CreateKind(HermesContentNodeKind);      {hermes_content}
CreateKind(HermesGraphicNodeKind);      {hermes_graphic}
CreateKind('hermes_node');
CreateKind('issue');
CreateKind('answer');
CreateKind('argument');
DBAddSynonym('issues', 'issue', ot_NodeKind);
DBAddSynonym('subissue', 'issue', ot_NodeKind);
DBAddSynonym('subissues', 'issue', ot_NodeKind);
DBAddSynonym('answers', 'answer', ot_NodeKind);
DBAddSynonym('subanswer', 'answer', ot_NodeKind);
DBAddSynonym('subanswers', 'answer', ot_NodeKind);
DBAddSynonym('arguments', 'argument', ot_NodeKind);
DBAddSynonym('subargument', 'argument', ot_NodeKind);
DBAddSynonym('subarguments', 'argument', ot_NodeKind);
CreateKind('annotation');
CreateKind('author');
CreateKind('objection');
CreateKind('critique');
CreateKind('generalization');
CreateKind('specification');
CreateKind('reference');
CreateKind('replacement');
CreateKind('summary');
CreateKind('keyword');
CreateKind('status');
CreateKind('consequence');
CreateKind('precondition');
CreateKind('argument for');
CreateKind('argument against');
CreateKind('habitat');
CreateKind('chair');
```

```
  CreateKind('table');
  CreateKind('seat');
  CreateKind('leg');
  CreateKind('top');
  DBAddSynonym('question', 'issue', ot_NodeKind);            {IBIS Types}
  DBAddSynonym('questions', 'issue', ot_NodeKind);
  DBAddSynonym('position', 'answer', ot_NodeKind);
  DBAddSynonym('positions', 'answer', ot_NodeKind);
  DBAddSynonym('decision', 'answer', ot_NodeKind);
  DBAddSynonym('decisions', 'answer', ot_NodeKind);
  DBAddSynonym('response', 'answer', ot_NodeKind);
  DBAddSynonym('responses', 'answer', ot_NodeKind);
  DBAddSynonym('support', 'argument', ot_NodeKind);
  DBAddSynonym('supports', 'argument', ot_NodeKind);
  DBAddSynonym('comment', 'annotation', ot_NodeKind);
  DBAddSynonym('comments', 'annotation', ot_NodeKind);
  DBAddSynonym('suggestion', 'annotation', ot_NodeKind);
  DBAddSynonym('annotations', 'annotation', ot_NodeKind);
  DBAddSynonym('suggestions', 'annotation', ot_NodeKind);
  DBAddSynonym('authors', 'author', ot_NodeKind);
  DBAddSynonym('objections', 'objection', ot_NodeKind);
  DBAddSynonym('critiques', 'critique', ot_NodeKind);
  DBAddSynonym('generalizations', 'generalization', ot_NodeKind);
  DBAddSynonym('specifications', 'specification', ot_NodeKind);
  DBAddSynonym('references', 'reference', ot_NodeKind);
  DBAddSynonym('replacements', 'replacement', ot_NodeKind);
  DBAddSynonym('summaries', 'summary', ot_NodeKind);
  DBAddSynonym('keywords', 'keyword', ot_NodeKind);
  DBAddSynonym('statuses', 'status', ot_NodeKind);
  DBAddSynonym('consequences', 'consequence', ot_NodeKind);
  DBAddSynonym('preconditions', 'precondition', ot_NodeKind);
  DBAddSynonym('arguments for', 'argument for', ot_NodeKind);
  DBAddSynonym('arguments against', 'argument against', ot_NodeKind);
  DBAddSynonym('pro', 'argument for', ot_NodeKind);
  DBAddSynonym('con', 'argument against', ot_NodeKind);
  DBAddSynonym('habitats', 'habitat', ot_NodeKind);
  DBAddSynonym('chairs', 'chair', ot_NodeKind);
  DBAddSynonym('tables', 'table', ot_NodeKind);

  TheAssociation := New(PAssociationWith, Init(('subissue_tree'), New(PAssociationType, Init(NoName,
'subissues')),
                                                New(PAssociationType, Init(NoName,
'subissue_tree'))));
  DBPutNamedObject(TheAssociation, HermesUniversalContextId);

  TheAssociation := New(PAssociationWith, Init('issue_tree', New(PAssociationType, Init(NoName,
'issues')),
                                                New(PAssociationType, Init(NoName,
'subissue_tree'))));
  DBPutNamedObject(TheAssociation, HermesUniversalContextId);

  TheAssociation := New(PAssociationWith, Init('subargument_tree', New(PAssociationType, Init(NoName,
'subarguments')),
                                                New(PAssociationType, Init(NoName,
'subargument_tree'))));
  DBPutNamedObject(TheAssociation, HermesUniversalContextId);

  TheAssociation := New(PAssociationWith, Init('argument_tree', New(PAssociationType, Init(NoName,
'arguments')),
                                                New(PAssociationType, Init(NoName,
'subargument_tree'))));
  DBPutNamedObject(TheAssociation, HermesUniversalContextId);

  TheAssociation := New(PAssociationWith, Init('subanswer_tree', New(PAssociationType, Init(NoName,
'subanswers')),
                                                New(PAssociationType, Init(NoName,
'subanswer_tree'))));
  DBPutNamedObject(TheAssociation, HermesUniversalContextId);

  TheAssociation := New(PAssociationWith, Init('answer_tree', New(PAssociationType, Init(NoName,
'answers')),
                                                New(PAssociationType, Init(NoName,
'subanswer_tree'))));
  DBPutNamedObject(TheAssociation, HermesUniversalContextId);
```

```
  TheAssociation := New(PAssociationWith, Init('deliberation', New(PAssociationType, Init(NoName,
'answer_tree')),
                                                New(PAssociationType, Init(NoName,
'argument_tree'))));
  DBPutNamedObject(TheAssociation, HermesUniversalContextId);

  TheAssociation := New(PAssociationWith, Init('discussion', New(PAssociationType, Init(NoName,
'issue_tree')),
                                                New(PAssociationType, Init(NoName, 'deliberation'))));
  DBPutNamedObject(TheAssociation, HermesUniversalContextId);

  TheAssociation := New(PAssociationWith, Init('hermes_link_tree', New(PAssociationType, Init(NoName,
'hermes_link')),
                                                New(PAssociationType, Init(NoName,
'hermes_link_tree'))));
  DBPutNamedObject(TheAssociation, HermesUniversalContextId);

  {set up for critics}
  CreateHypertextNode('hermes_main_critique', 'Critiques of the current design:',
HermesUniversalContext);
  AddRationale('hermes_main_critique', 'summary 1', 'summary', HermesUniversalContext,
    'No problems were found with the current design');
  AddRationale('hermes_main_critique', 'critique 1', 'critique', 'Clayton Lewis context',
    'The private areas are not separated from the public areas.');
  AddRationale('hermes_main_critique', 'critique 2', 'critique', 'Mike Eisenberg context',
    'The layout of the sleep compartments is not modular for expansion.');
  TheDataList := New(PDataListOf, Init('hermes_critique', New(PAssociationType, Init(NoName,
'critique')),
                                        New(PDataListId, Init(NoName, DBIdOf('hermes_main_critique',
ot_Character)))));
  DBPutNamedObject(TheDataList, HermesUniversalContextId);
  TheDataList := New(PDataListOf, Init('hermes_null_critique', New(PAssociationType, Init(NoName,
'summary')),
                                        New(PDataListId, Init(NoName, DBIdOf('hermes_main_critique',
ot_Character)))));
  DBPutNamedObject(TheDataList, HermesUniversalContextId);
end;

       {****************************************
       *
       *   procedure DefineGraphics;
       *
       ****************************************}

procedure DefineGraphics;
var
  NewChair, NewTable, NewHabitat             : PHermesGraphic;
  NewTransform                               : PTransform;
  NewFixed, NewDisplace, NewScale, NewRotate : PHermesPoint;
  NewCube                                    : PHermesGraphic;
  NewMin, NewMax                             : PHermesPoint;
  TestContext                                : LongInt;
begin
  TestContext := HermesUniversalContextId;                  {define context for test graphics}

  NewHabitat := New(PHermesGraphic, Init('Simple Test Habitat'));
  NewHabitat^.SetKind('habitat');
  DBPutNamedObject(NewHabitat, TestContext);                {create test habitat graphic}

  NewMin := New(PHermesPoint, Init(0,0,0));
  NewMax := New(PHermesPoint, Init(500, 300, 300));
  NewCube := New(PHermesGraphic, Init('Habitat Shell'));
  DBPutNamedObject(NewCube, TestContext);
  NewFixed := New(PHermesPoint, Init(0,0,0));
  NewDisplace := New(PHermesPoint, Init(0,0,0));
  NewScale := New(PHermesPoint, Init(0,0,0));
  NewRotate := New(PHermesPoint, Init(0,0,0));
  NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
  NewHabitat^.AddContentNode(NewCube, NewTransform);         {place shell around habitat}

  NewChair := New(PHermesGraphic, Init(NoName));
  NewChair^.SetKind('chair');
  DBPutNamedObject(NewChair, TestContext);
  NewDisplace := New(PHermesPoint, Init(250,50,50));
  NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
  NewHabitat^.AddContentNode(NewChair, NewTransform);         {place chair 1 in habitat}
```

```
NewDisplace := New(PHermesPoint, Init(350,50,50));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewHabitat^.AddContentNode(NewChair, NewTransform);         {place chair 2 in habitat}

NewDisplace := New(PHermesPoint, Init(250,250,50));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewHabitat^.AddContentNode(NewChair, NewTransform);         {place chair 3 in habitat}

NewDisplace := New(PHermesPoint, Init(350,250,50));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewHabitat^.AddContentNode(NewChair, NewTransform);         {place chair 4 in habitat}

NewTable := New(PHermesGraphic, Init(NoName));
NewTable^.SetKind('table');
DBPutNamedObject(NewTable, TestContext);
NewDisplace := New(PHermesPoint, Init(200,100,50));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewHabitat^.AddContentNode(NewTable, NewTransform);         {place table in habitat}

NewMin := New(PHermesPoint, Init(0, 0, 0));
NewMax := New(PHermesPoint, Init(20, 20, 5));
NewCube := New(PHermesGraphic, Init(NoName));
NewCube^.SetKind('seat');
DBPutNamedObject(NewCube, TestContext);
NewDisplace := New(PHermesPoint, Init(0, 0, 30));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewChair^.AddContentNode(NewCube, NewTransform);         {place seat on chair}

NewMin := New(PHermesPoint, Init(0, 0, 0));
NewMax := New(PHermesPoint, Init(4, 4, 30));
NewCube := New(PHermesGraphic, Init(NoName));
NewCube^.SetKind('leg');
DBPutNamedObject(NewCube, TestContext);
NewDisplace := New(PHermesPoint, Init(0, 0, 0));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewChair^.AddContentNode(NewCube, NewTransform);          {place leg 1 on chair}

NewDisplace := New(PHermesPoint, Init(0, 16, 0));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewChair^.AddContentNode(NewCube, NewTransform);          {place leg 2 on chair}

NewDisplace := New(PHermesPoint, Init(16, 0, 0));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewChair^.AddContentNode(NewCube, NewTransform);          {place leg 3 on chair}

NewDisplace := New(PHermesPoint, Init(16, 16, 0));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewChair^.AddContentNode(NewCube, NewTransform);          {place leg 4 on chair}

NewMin := New(PHermesPoint, Init(0, 0, 0));
NewMax := New(PHermesPoint, Init(200, 100, 5));
NewCube := New(PHermesGraphic, Init(NoName));
NewCube^.SetKind('top');
DBPutNamedObject(NewCube, TestContext);
NewDisplace := New(PHermesPoint, Init(0, 0, 50));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewTable^.AddContentNode(NewCube, NewTransform);         {place top on table}

NewMin := New(PHermesPoint, Init(0, 0, 0));
NewMax := New(PHermesPoint, Init(4, 4, 50));
NewCube := New(PHermesGraphic, Init(NoName));
NewCube^.SetKind('leg');
DBPutNamedObject(NewCube, TestContext);
NewDisplace := New(PHermesPoint, Init(0, 0, 0));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewTable^.AddContentNode(NewCube, NewTransform);          {place leg 1 on table}

NewDisplace := New(PHermesPoint, Init(0, 96, 0));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewTable^.AddContentNode(NewCube, NewTransform);          {place leg 2 on table}

NewDisplace := New(PHermesPoint, Init(196, 0, 0));
NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
NewTable^.AddContentNode(NewCube, NewTransform);          {place leg 3 on table}
```

```
  NewDisplace := New(PHermesPoint, Init(196, 96, 0));
  NewTransform := New(PTransform, Init(NewFixed, NewDisplace, NewScale, NewRotate));
  NewTable^.AddContentNode(NewCube, NewTransform);          {place leg 4 on table}
end;

       {*************************************
       *
       *   procedure CreateSeed;
       *
       **************************************}

procedure CreateSeed;
begin
  DefineTypes;
  DefineContextTree;
  DefineHypertextTree;
  DefineRationale;
  (* DefineGraphics; *)
end;


end.                {HerSeeds.pas}
```

18. HerPrivs.pas authoring privileges

```
 {*****************************************
 *
 *      HerPrivs.pas
 *
 *      version 2.0 -- Spring 1994
 *      copyright (c) 1994 by Gerry Stahl
 *      all rights reserved
 ******************************************}

unit HerPrivs;

 { This unit defines security rules for the Hermes system.}

interface

  procedure PrivsSwitchOn;
  {Set PrivsSwitch to True}

  procedure PrivsSwitchOff;
  {Set PrivsSwitch to False}

  function PrivsSwitchGet : Boolean;
  {Return value of PrivsSwitch}

  procedure PrivsErrorMsg;
  {Display security message}

  function PrivsCheckLogin : Boolean;
  {Process login security checking.
  This function is called from HerWorld's login procedures.}

  function PrivsCreateLink(TheContextId : LongInt) : Boolean;
  {Rule for Annotation.
  This function is called from CreateHypertextSwitch.}

  function PrivsEditNode(TheNodeOriginalContextId, TheEditContextId : LongInt) : Boolean;
  {Rule for Editing.
  This function is called from Delete Node and ModifyNode.}

  function PrivsCreateChildContext(ParentContextId : LongInt) : Boolean;
  {Rule for Creating Contexts.
  This function is called from HerWorld's context creation procedures.}

  function PrivsEditContext(OldContextId, NewContextId : LongInt) : Boolean;
  {Rule for Promoting Contexts.
  This function is called from HerWorld's context promotion procedures.}

{==================================================================================}

implementation

uses Strings, OStdDlgs, OWindows, WinTypes, WinProcs,
     HerLists, HerBasic, HerNodes, HerDataB, HerWorld;

const
  PrivsSwitch : Boolean = False;

        {***************************************
        *
        *    function PrivsSwitchOn
        *
        ****************************************}

procedure PrivsSwitchOn;
  {Set PrivsSwitch to True}
begin
  PrivsSwitch := True;
end;

        {***************************************
        *
        *    function PrivsSwitchOff
```

```
        *
        ****************************************}

procedure PrivsSwitchOff;
  {Set PrivsSwitch to False}
begin
  PrivsSwitch := False;
end;

        {****************************************
        *
        *    function PrivsSwitchGet
        *
        ****************************************}

function PrivsSwitchGet : Boolean;
  {Return value of PrivsSwitch}
begin
  PrivsSwitchGet := PrivsSwitch;
end;

        {****************************************
        *
        *    function PrivsErrorMsg
        *
        ****************************************}

procedure PrivsErrorMsg;
  {Display security message}
begin
  ErrorMsg(et_NOTICE, 'You are not authorized to make this change.');
end;

        {****************************************
        *
        *    function PrivsCheckLogin
        *
        ****************************************}

function PrivsCheckLogin : Boolean;
  {Process login security checking.
   This function is called from HerWorld's login procedures.}
var
  EditText      : array[0..41] of Char;
  TheLogin : PLogin;
begin
  if PrivsSwitchGet
    then
      begin
        PrivsCheckLogin := False;           {set up default guest permissions}
        HermesApp.SetDBAuthor('guest');
        StrCopy(EditText, 'guest');
        if (Application^.ExecDialog(New(PInputDialog, Init(Application^.MainWindow,
                      'Login Procedure', 'Enter your user name: ',
                      EditText, SizeOf(EditText)))) = id_OK) then
          begin                                 {if got a Login name}
            TheLogin := PLogin(DBGetName(EditText, ot_Login));
            if assigned(TheLogin) then
              begin                             {and if a Login is defined for the Login name}
                StrCopy(EditText, '                                        ');
                if (Application^.ExecDialog(New(PInputDialog, Init(Application^.MainWindow,
                      'Login Procedure', 'Enter your password: ',
                      EditText, SizeOf(EditText)))) = id_OK) then
                  begin                         {and if correct password entered}
                    PrivsCheckLogin := (StrComp(EditText, TheLogin^.GetPassword) = 0);
                    HermesApp.SetDBAuthor(TheLogin^.GetName);
                  end;                          {then return True and save Login as DBAuthor}
              end;
          end;
      end
    else PrivsCheckLogin := True;
end;

        {****************************************
        *
        *    function PrivsCreateLink
```

```
        *
        ****************************************}

function PrivsCreateLink(TheContextId : LongInt) : Boolean;
begin
  PrivsCreateLink := True;
end;

        {****************************************
        *
        *    function PrivsEditNode
        *
        ****************************************}

function PrivsEditNode(TheNodeOriginalContextId, TheEditContextId : LongInt) : Boolean;
begin
  PrivsEditNode := True;
end;

        {****************************************
        *
        *    function PrivsCreateChildContext
        *
        ****************************************}

function PrivsCreateChildContext(ParentContextId : LongInt) : Boolean;
begin
  PrivsCreateChildContext := True;
end;

        {****************************************
        *
        *    function PrivsEditContext
        *
        ****************************************}

function PrivsEditContext(OldContextId, NewContextId : LongInt) : Boolean;
begin
  PrivsEditContext := True;
end;

end.    {HerPrivs}
```

* end of Hermes source code and documentation