```
Poplar: CONFIGURATION
IMPORTS SystemDefs, StringDefs, IODefs, StreamDefs, SegmentDefs, MiscDefs, DiskKDDefs, BFSDefs, DiskDefs, DirectoryDefs,
ImageDefs, FrameDefs, DisplayDefs, ProcessDefs, CodeDefs,Stream, AllocDefs, CoreSwapDefs, LoadStateDefs
CONTROL pl = BEGIN
CheckPoint;
VM;
AFiles;
ThreeWayCore;
TimeConvert;
NWF;
disp;
pl;
sup;
eval;
parse;
string;
store;
route1;
route2;
stat;
BTree;
BTree2;
Storage;
BT;
EtherReport;
END.
```

```
BTDefs:DEFINITIONS = BEGIN
Setup: PROCEDURE;
END.
```

```
--B tree defs (maybe retrieval defs ?)
DIRECTORY FilePageUseDefs: FROM "FilePageUseDefs";
BTreeDefs:DEFINITIONS = BEGIN

Desc:TYPE = DESCRIPTOR FOR ARRAY OF WORD;
BTreeHandle:TYPE = PRIVATE RECORD[CARDINAL];
Call:TYPE = PROCEDURE[k:Desc,v:Desc] RETURNS[more:BOOLEAN];
TestKeys:TYPE = PROCEDURE[a,b:Desc] RETURNS[BOOLEAN];

CreateBTree:PROCEDURE[file:FilePageUseDefs.FileHandle,new:BOOLEAN]
        RETURNS[BTreeHandle];
Insert:PROCEDURE[bh:BTreeHandle,key:Desc,value:Desc];
Lookup:PROCEDURE[bh:BTreeHandle,key,value:Desc]
    RETURNS[lengthValue:CARDINAL];--177777B for not found
Delete:PROCEDURE[bh:BTreeHandle,key:Desc];
Enumerate:PROCEDURE[bh:BTreeHandle,key:Desc,c:Call];
ReleaseBTree:PROCEDURE[bh:BTreeHandle];
InitializeBTree:PROCEDURE
  [bh:BTreeHandle,
  isFirstBigger,areTheyEqual:TestKeys,
  dirtyCache:BOOLEAN];
FlushCache:PROCEDURE[bh:BTreeHandle];
AskDepth:PROCEDURE[bh:BTreeHandle] RETURNS[CARDINAL];
NullProc:TestKeys;

END.
```

A B Tree stores and retrieves Entries in a file system. A entry consists of a key and a value. Retrieval is based on the key, and returns the value. The user can insert, lookup or delete entries. He can also Enumerate entries starting at a particular key and getting called with successive entries until the call returns false. There is no provision for deleting the file once it is created; Release only closes it.

A Btree stores ordered Entries. If one attempts a lookup on a non-existent entry, one gets avalue length of -1, and an enumerate returns entries in numerical order (all based on the key). The ordering property is vital internally, since a binary search is going on to find the entry of interest.

InitializeBTree allows the user to supply his own ordering of items in the BTree, by supplying routines which define greater than and equal for the keys. InitializeBTree need never be called if the user is content with the default ordering, which is a simple cardinal compare of the elements one by one (runing out of elements is smaller). A user who builds a tree with one algorithm and searches it with another gets what he deserves. Normally the package operates in a "safe" mode, where it writes out changes as they occur. It is possible to change this mode to keep a dirty cache, and write it out only on closing the file, which goes quicker allows no recourse on an abort. If Initialize specifies NullProc , the defaults will be used.

It is improper for the user to store an entry with a zero length key. Enumerating on such a key dumps the whole tree.

Bug: The definition of Lookup is not good:
- a) it doesn't tell you the length of the value
- b) it has no way to change the entry
- c) it is awkward at best - there must be a "found" boolean.

Bug: The current implementation may get horribly lost if you change the tree during the course of an enumeration, and then try to continue the enumeration.

Bug: The enumeration actually returns pointers right into the pages of the tree. It would be quite desirable to change the value part of the entry, say during an enumeration or a lookup, and to have that reflected into the file. So long as the entry does not change length, there is no conceptual difficulty here. Unfortunately, the current implementation has no way of noticing that the page has become dirty, and will "cleverly" avoid rewriting the page, thereby preventing the change from happening (unless some other change marked it dirty later).

```
--btree2Defs

  BTree2Defs:DEFINITIONS = BEGIN

  VArray:TYPE =  POINTER TO DESCRIPTOR FOR ARRAY OF WORD;
  Entry :TYPE =  POINTER TO EntryR;
  EntryR:TYPE =  RECORD[k:Key,v:Value];
  Key   :TYPE =  DESCRIPTOR FOR ARRAY OF WORD;
  Value :TYPE =  DESCRIPTOR FOR ARRAY OF WORD;
  Index :TYPE =  CARDINAL;
  nullIndex:CARDINAL =  100000B;

  Initialize:PROCEDURE[VArray];
  Insert   :PROCEDURE[VArray,Index,Entry];
  Lookup   :PROCEDURE[VArray,Index] RETURNS[EntryR];
  Delete   :PROCEDURE[VArray,Index];
  Replace  :PROCEDURE[VArray,Index,Entry];

  GetAnother:SIGNAL RETURNS [VArray];
  Merger   :SIGNAL RETURNS [v:VArray,vOnRight:BOOLEAN];
  DeletePage:SIGNAL;

  Items:PROCEDURE[VArray] RETURNS[CARDINAL];

  END.
```

VArray stores in a compact way a variable number of variable length items, keyed off an index number. Index numbers start at 0 and run to Items-1. You may insert,lookup, delete,or replace at any index. Structurally, a Varray is a Pointer to a Descriptor for an Array of Word, and any client can create the structure. It must be initialized before use. An Item consists of two parts, a key and a Value. Each part of each item is described by a array descriptor, both for insertion and lookup. No copy is implied by a lookup: the descriptor points directly into the array. Insertion and deletion immediately renumber all of the stored items. Logically, replace is a delete followed by an insert, but by doing it all at once you avoid an ill advised attempt to merge the page (see the Merge signal below). Insertions occur after the index, so you must insert at -1 to put in the first entry.

There are two interesting events which trigger signals: trying to insert something which won't fit, and trying to delete something when the vArray is pretty empty.
If won't fit, the implementer signals GetAnother, which the client can resume with a new vArray for the overflow.
If the buffer gets rather empty, the implementer signals Merger. If the client Resumes with a new buffer, then the implementer will try to merge the current vArray with the new buffer. If successful, the implementer will signal DeletePage. It is possible to merge with a following or a preceeding buffer (vOnRight = TRUE or FALSE) but in either case, the modification will occur to the rightmost buffer. If the client resumes with NIL, the short array will remain.

```
-- DispDefs.Mesa Last edited by Morris November 29, 1978  5:58 PM
                                                        DIRECTORY
                                                        PLDefs: FROM "PLDefs";


DispDefs: DEFINITIONS = BEGIN

-- defined in disp.mesa
DispSetup: PROCEDURE;
DispReset: PROCEDURE;
Print: PROCEDURE[PLDefs.Node];
PrintString: PROCEDURE[PLDefs.Node];
Confirm: PROCEDURE RETURNS [BOOLEAN];
PrintChar: PROCEDURE[CHARACTER];
MyWriteProcedure: PROCEDURE[CHARACTER];
ClearLine: PROCEDURE;
ClearScreen: PROCEDURE;
ToggleMore: PROCEDURE RETURNS[BOOLEAN];
ToggleAllPrint: PROCEDURE RETURNS[BOOLEAN];

-- defined in wf.mesa
WF0:            PROCEDURE [STRING];
WF1:            PROCEDURE [STRING,UNSPECIFIED];
                    WF2:    PROCEDURE [STRING,UNSPECIFIED,UNSPECIFIED];
                    WF3:    PROCEDURE [STRING,UNSPECIFIED,UNSPECIFIED,UNSPECIFIED];
                    WF4:    PROCEDURE [STRING,UNSPECIFIED,UNSPECIFIED,UNSPECIFIED, UNSPECIFIED];
SetCode:        PROCEDURE[CHARACTER, PROCEDURE[UNSPECIFIED, STRING]];
ResetCode:      PROCEDURE [CHARACTER];
SetWriteProcedure: PROCEDURE [PROCEDURE[CHARACTER]] RETURNS [PROCEDURE[CHARACTER]];
WriteToString: PROCEDURE[STRING];
WFError:        SIGNAL[STRING,CARDINAL];
END.
```

```
EtherReportDefs: DEFINITIONS = BEGIN

Socket: TYPE = RECORD[high,low: CARDINAL];
Port: TYPE = RECORD[net, host: [0..256), socket: Socket];
Result: TYPE = {OK, NoEtherNet, HostIs0, RouteRequestFailed, SendFailed, NoReply};
Report: PROCEDURE[
        Port,
        POINTER TO ARRAY[0..1) OF WORD,
        CARDINAL
        ] RETURNS [Result];
END.
```

```
-- PLDefs.Mesa Last edited by Morris October 26, 1978 5:42 PM
-- modified by Wadler for lazy eval, 7/6/79
                                                    DIRECTORY
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    FilePageUseDefs: FROM "FilePageUseDefs",
                                                    ControlDefs: FROM "controldefs";


PLDefs: DEFINITIONS = BEGIN
TokType: TYPE = {ZERO,LP,RP,LB,RB,COMMA,EOF, SEP,LC,RC,COLON,
HOLE,DIV,STR,ID,RARR,ASS,PROG,FCN,SEQOF,OPT, DELETE,SEQOFC,MAPPLY, UNARY,PLUS,MINUS,ZARY,
GOBBLE,PFUNC,TILDE,PALT,PFUNC1, SEQUENCE,CATL,WILD,FAIL,ITER,SCREEN,EQUAL};
NodeType: TYPE = {ZERO,ID,STR,HOLE,LIST,CAT,MATCH,APPLY,FCN, ASS, PROG,FAIL,SEQOF,OPT,  SEQOFC, DELETE,
MAPPLY,
UNARY,PLUS,MINUS,ZARY,GOBBLE,PFUNC,PALT,PFUNC1, SEQUENCE, CATL,WILD,PATTERN,ITER,TILDE,EQUAL, ENV,
CLOSURE,UNDEFINED};
SType: TYPE = {zero,unk,string,proc,patproc};

Node: TYPE = POINTER TO NodeRecord;
pNode: TYPE = POINTER TO Node;
NodeRecord: TYPE = MACHINE DEPENDENT RECORD[
        Type: NodeType,
        Des: DesRecord,
        Var: SELECT OVERLAID NodeType FROM
        -- base
        ID => [name: Symbol],
        STR => [str: StringRecord],
        ZARY => [zary: Symbol],
        PFUNC => [pfunc: Symbol],
        FAIL, WILD, HOLE,UNDEFINED => NULL,
        -- inductive
        UNARY => [unary: Symbol, uexp: Node],
        PFUNC1 => [pfunc1: Symbol, pexp: Node],
        LIST =>         [listhead,listtail: Node],
        CAT,CATL,EQUAL => [left,right: Node],
        MATCH => [div,patt: Node], -- actually >, for historical reasons
        PALT => [alt1,alt2: Node],      -- alt1, alt2 = lists
        APPLY, MAPPLY, GOBBLE, ITER => [object,target: Node],
        FCN => [parms,fcn: Node],
        ASS => [lhs,rhs: Node],
        PROG => [prog1,prog2: Node],
        SEQOF, SEQOFC =>        [seqof: Node],
        OPT => [opt: Node],
        SEQUENCE => [from,to: Node],
        DELETE => [delete: Node],
        PLUS,MINUS => [arg1,arg2: Node],
        PATTERN => [pattern:Node],
        TILDE =>        [not:  Node],
        ENV => [val, next: Node],
        CLOSURE => [exp, env: Node],
        ENDCASE
        ];

Symbol: TYPE = POINTER TO SymbolRecord;
SymbolRecord: TYPE = RECORD[
        name: STRING,
        tok: TokType,
        type: SType,
        var: SELECT OVERLAID SType FROM
        zero,unk,string => [val:Node],
        proc =>         [proc: PROCEDURE[Symbol,Node,Node] RETURNS[Node]],
        patproc =>      [patproc: PROCEDURE[Symbol,Stream,Node] RETURNS[Node]],
        ENDCASE
        ];
StringType: TYPE = {simp,cat};
String: TYPE = POINTER TO StringRecord;
StringRecord: TYPE = RECORD[
        var: SELECT OVERLAID StringType FROM
        simp => [start,length: CARDINAL],
        cat => [n1,n2: Node],
        ENDCASE
        ];
```

```
DesRecord: TYPE = RECORD[
        g: BOOLEAN,     -- mark bit for garbage collector
        e: BOOLEAN,     -- if TRUE, this node has been eval'ed
        s: StringType,  -- if STR, which type
        b: BOOLEAN,     -- if TRUE, this STR has been balanced,
        fi: FileIndex   -- if type = simp (0 is default)
        ];
FileIndex: TYPE = [0..64);

Register: TYPE = CARDINAL;
Stream: TYPE = POINTER TO StreamRecord;
SBSize: CARDINAL = 40;
StreamRecord: TYPE = RECORD[
        -- this can be loopholed into string
        len: CARDINAL, -- no. of good characters in buff
        cp: CARDINAL,
            -- next character avail in buff (cp = len => empty)
        buff: PACKED ARRAY [0..SBSize) OF CHARACTER,
        node: Node, -- source string (NIL<=>empty)
        posn: CARDINAL -- next char in source string
        -- if node is a cat then it is right linear and left most sub-string is of length greater than posn.
        ];
        -- the buffer is redundant, because its characters can also be found in the len characters starting at posn.
--
NumLines: CARDINAL = 35;
NumPages: CARDINAL = 35;
sSize: CARDINAL = 500;
Unbound: UNSPECIFIED = ControlDefs.ControlLinkTag[unbound];
cdebug: BOOLEAN = FALSE;
--

PBug: SIGNAL[est: STRING];
SErr: SIGNAL[est: STRING];
RErr: SIGNAL[est: STRING];
EndDisplay: SIGNAL;
Interrupt: SIGNAL;
--
--
-- defined in pl.mesa
IsDebug: PROCEDURE RETURNS[BOOLEAN];
Fixup: PROCEDURE[Node] RETURNS[Node];
--
-- defined in parse.mesa
ParseSetup: PROCEDURE;
Dist: PROCEDURE[Node] RETURNS[Node];
ErrorMsg: PROCEDURE[STRING,STRING];
ErrorMsg1: PROCEDURE[STRING,UNSPECIFIED];
SetCurrentNode: PROCEDURE[Node];
--
-- defined in eval.mesa
Eval: PROCEDURE[Node,Node] RETURNS[Node];
ZEval: PROCEDURE[Node] RETURNS[Node];
Map: PROCEDURE[Node,PROCEDURE[Node]] ;
LengthList: PROCEDURE[Node] RETURNS[CARDINAL];
--
-- defined in sup.mesa
SupSetup: PROCEDURE;
SupReset: PROCEDURE;

-- defined in Route2.mesa
WriteRoutine: PROCEDURE[Symbol,Node,Node] RETURNS[Node];

-- defined in string.mesa
StringSetup: PROCEDURE;
StringCleanup: PROCEDURE;
MakeString: PROCEDURE[STRING,Node];
MakeSTR: PROCEDURE[STRING] RETURNS[Node];
Coerce: PROCEDURE[Node] RETURNS[Node];
MakeNUM: PROCEDURE[LONG INTEGER] RETURNS [Node];
MakeInteger: PROCEDURE[Node] RETURNS [LONG INTEGER];
StringConcat: PROCEDURE[Node,Node] RETURNS [Node];
LayOutBits: PROCEDURE[Node];
```

```
DoTransfer: PROCEDURE;
Transfer: PROCEDURE[Node];
StringDebugging: PROCEDURE;
StringGC: PROCEDURE;
EndStringGC: PROCEDURE;
GetSin: PROCEDURE RETURNS[CARDINAL];
FileRoutine: PROCEDURE[Symbol,Node,Node] RETURNS[Node];
Sub: PROCEDURE[Node,LONG INTEGER] RETURNS[CHARACTER];
SubString: PROCEDURE[Node,LONG INTEGER,LONG INTEGER] RETURNS [Node];
NewStream: PROCEDURE[Node] RETURNS[StreamRecord];
Item: PROCEDURE[Stream] RETURNS[CHARACTER];

-- defined in store.mesa
Alloc:PROCEDURE [NodeRecord] RETURNS [Node];
FreeTree: PROCEDURE[Node];
CheckNode: PROCEDURE[Node];
IndexNode: PROCEDURE[Node] RETURNS[CARDINAL];
Insert: PROCEDURE[STRING,TokType,PLDefs.SType,
        PROCEDURE[Symbol,Node,Node] RETURNS[Node],
        PROCEDURE[Symbol,Stream,Node] RETURNS[Node]]
        RETURNS[Symbol];
Lookup: PROCEDURE[STRING] RETURNS[Symbol];
ResetCursor: PROCEDURE;
StoreSetup: PROCEDURE;
StoreReset: PROCEDURE;
StoreCleanup: PROCEDURE;
MRS: PROCEDURE RETURNS[Register];
RRS: PROCEDURE[Register];
R: PROCEDURE[pNode];
R2: PROCEDURE[pNode,pNode];
R3: PROCEDURE[pNode,pNode,pNode];
IgnoreRRS: PROCEDURE;
CopyTree: PROCEDURE[Node] RETURNS[Node];
Preorder: PROCEDURE[Node,PROCEDURE[Node] RETURNS[BOOLEAN]];
Postorder: PROCEDURE[Node,PROCEDURE[Node] RETURNS[BOOLEAN]];
SnapShot: PROCEDURE;
GarbageRoutine: PROCEDURE[Symbol,Node,Node] RETURNS[Node];
GetSpecialNodes: PROCEDURE RETURNS[Node,Node,Node];
ParseTree: PROCEDURE[Node];
GetCore: PROCEDURE[CARDINAL] RETURNS[POINTER];

-- defined in stat.mesa
StatSetup: PROCEDURE;
Empty: PROCEDURE[Node] RETURNS [BOOLEAN];
Length: PROCEDURE[Node] RETURNS[LONG INTEGER];
Skip: PROCEDURE[Node,LONG INTEGER];
LinearSTR: PROCEDURE[Node] RETURNS[Node,LONG INTEGER];
Detail: PROCEDURE[Node];
FixRep: PROCEDURE[Node] RETURNS[Node];
SubStream: PROCEDURE[Stream,LONG INTEGER] RETURNS[CHARACTER];
SkipStream: PROCEDURE[Stream,LONG INTEGER];
LengthStream: PROCEDURE[Stream] RETURNS[LONG INTEGER];
ConvertStream: PROCEDURE[Stream] RETURNS[Node];
SubStringStream: PROCEDURE[Stream,LONG INTEGER,LONG INTEGER] RETURNS [Node];

END.
```

```
DIRECTORY PLDefs: FROM "PLDefs";

RouteDefs: DEFINITIONS = BEGIN

Route1Setup: PROCEDURE;
Route2Setup: PROCEDURE;
KeyRoutine: PROCEDURE[PLDefs.Symbol,PLDefs.Node,PLDefs.Node] RETURNS[PLDefs.Node];
Route3Setup: PROCEDURE;
END.
```

```
                    DIRECTORY
                    FileSystemDefs: FROM "FileSystemDefs";

VMDefs: DEFINITIONS = BEGIN

FileIndex: TYPE = [0..64);
Cardinal: PROCEDURE[in: LONG INTEGER] RETURNS [CARDINAL];
FS: PROCEDURE RETURNS [FileSystemDefs.FileSystem];
VMSetup: PROCEDURE;
VMCleanup: PROCEDURE;
GetSChar: PROCEDURE[LONG INTEGER, FileIndex] RETURNS[CHARACTER];
SetSChar: PROCEDURE[LONG INTEGER, FileIndex, CHARACTER];
GetString: PROCEDURE[LONG INTEGER,CARDINAL, FileIndex, STRING];
SetString: PROCEDURE[LONG INTEGER, FileIndex, STRING];
Open: PROCEDURE[STRING] RETURNS [FileIndex, LONG INTEGER];
Close: PROCEDURE[FileIndex];
OpenRW: PROCEDURE[STRING] RETURNS [FileIndex, LONG INTEGER];
CloseRW: PROCEDURE[FileIndex, LONG INTEGER];
END.
```

```
                                        DIRECTORY
                                        PLDefs: FROM "PLDefs",
                                        DispDefs: FROM "DispDefs",
                                        SystemDefs: FROM "SystemDefs",
                                        IODefs: FROM "IODefs",
                                        StreamDefs: FROM "StreamDefs",
                                        RouteDefs: FROM "RouteDefs",
                                        InlineDefs: FROM "InlineDefs",
                                        StringDefs: FROM "StringDefs";

-- written by Morris and Schmidt
-- modified by Wadler for lazy eval, 7/13/79

-- functions return a lazy value, which is
--    a. a list with expressions for its head and tail  (any necessary parts are in closures)
-- or b. a completely evaluated expression
-- in either case the e bit is set.

-- parameters and returned values of procedures are registered only
-- if they are used for accumulating intermediate answers.
-- a parameter variable is not registered if it is only set to a component of
-- itself, or the ZEval of a component of itself, e.g. val ← ZEval[val.listail].
-- all other node variables are always registered.

eval: PROGRAM IMPORTS P: PLDefs, DispDefs, StreamDefs, RouteDefs EXPORTS PLDefs = BEGIN
--
TokType: TYPE = PLDefs.TokType;
NodeType: TYPE = PLDefs.NodeType;
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
Register: TYPE = PLDefs.Register;
GlobalEnv: Node ← NIL;
--
Fail,MTSt,Nail: Node;

Apply: PROCEDURE[val, func: Node] RETURNS [ret: Node] =
-- apply func to val
-- assumes val is not evaluated, but func is
BEGIN
CK;
IF func.Type = UNARY OR func.Type = ZARY THEN
        ret ← func.unary.proc[func.unary,ZEval[val],func.uexp]
ELSE IF func.Type = CLOSURE AND func.exp.Type = FCN THEN
        BEGIN
        rr : Register = P.MRS[];
        nev: Node ← NIL;
        P.R[@nev];
        nev ← ExtendEnv[func.env, func.exp.parms, val]; -- bind parameters
        ret ← Eval[func.exp.fcn, nev];
        P.RRS[rr];
        END
ELSE IF func.Type = CLOSURE AND func.exp.Type = PATTERN AND ZEval[val].Type = STR THEN
        ret ← Match[val, func.exp.pattern, func.env]
ELSE IF func.Type = LIST THEN
        BEGIN
        rr: Register = P.MRS[];
        r1, r2: Node ← NIL;
        P.R2[@r1, @r2];
        IF func.listhead = NIL THEN
                ret ← Nail
        ELSE
                BEGIN
                ret ← P.Alloc[[LIST,,LIST[
                                r1 ← P.Alloc[[APPLY,,APPLY[val, func.listhead]]],
                                r2 ← P.Alloc[[APPLY,,APPLY[val, func.listtail]]]]]];
                ret.Des.e ← TRUE;
                END;
        P.RRS[rr];
        END
ELSE IF ZEval[val].Type = LIST AND func.Type = STR THEN
```

```
        BEGIN -- subscript  (list may be in lazy form)
        rr: Register  =  P.MRS[];
        i: LONG INTEGER ← P.MakeInteger[func];
        neg: BOOLEAN ← FALSE;
        P.R[@val];
        IF i = 0 THEN P.RErr["Zero subscript"];
        IF i<0 THEN BEGIN i ← -i; neg← TRUE END;
        UNTIL i = 1
                DO
                IF val.Type # LIST THEN P.PBug["Malformed list"];
                IF val.listhead = NIL THEN P.RErr["Subscript too big"];
                val ← ZEval[val.listtail];
                i ← i-1;
                ENDLOOP;
        IF val.Type # LIST THEN P.PBug["Malformed list"];
        IF val.listhead = NIL THEN P.RErr["Subscript too big"];
        ret ← IF neg THEN ZEval[val.listtail] ELSE ZEval[val.listhead];
        P.RRS[rr];
        END
ELSE P.RErr["Bad application"];
END;

Binary: PROCEDURE[vi, vj: Node, p: NodeType] RETURNS[ans: Node] =
-- return p applied to vi and vj
-- p is one of PLUS, MINUS, CAT
-- pays special attention to cases where vi or vj are lists
-- doesn't assume vi and vj are zevaled
BEGIN
-- This routine registers its parameters, so callers needn't.
rr: Register ← P.MRS[];
r1, r2: Node ← NIL;
P.R2[@r1, @r2];
[] ← ZEval[vi]; [] ← ZEval[vj];
IF vi.Type = STR AND vj.Type = STR THEN
        BEGIN
        SELECT p FROM
                PLUS =>
                        RETURN[P.MakeNUM[P.MakeInteger[vi] + P.MakeInteger[vj]]];
                MINUS =>
                        RETURN[P.MakeNUM[P.MakeInteger[vi]-P.MakeInteger[vj]]];
                CAT =>
                        RETURN[P.StringConcat[vi, vj]];
                ENDCASE => P.PBug["Non Plus, Minus or Cat to Binary"];
        END
ELSE IF vi.Type = STR AND vj.Type = LIST THEN
        IF vj.listhead = NIL THEN
                ans ← Nail
        ELSE
                BEGIN
                ans ← P.Alloc[[LIST,,LIST[
                                r1 ← P.Alloc[[p,,PLUS[vi, vj.listhead]]],
                                r2 ← P.Alloc[[p,,PLUS[vi, vj.listtail]]]]]];
                        -- PLUS here acts like a loophole for MINUS and CAT
                END
ELSE IF vi.Type = LIST AND vj.Type = STR THEN
        IF vi.listhead = NIL THEN
                ans ← Nail
        ELSE
                BEGIN
                ans ← P.Alloc[[LIST,,LIST[
                                r1 ← P.Alloc[[p,,PLUS[vi.listhead, vj]]],
                                r2 ← P.Alloc[[p,,PLUS[vi.listtail, vj]]]]]];
                END
ELSE IF vi.Type = LIST AND vj.Type = LIST THEN
        IF vi.listhead = NIL AND vj.listhead = NIL THEN
                ans ← Nail
        ELSE
                BEGIN
```

```
                IF vi.listhead = NIL OR vj.listhead = NIL THEN P.RErr["List lengths differ"];
                ans ← P.Alloc[[LIST,,LIST[
                                r1 ← P.Alloc[[p,,PLUS[vi.listhead, vj.listhead]]],
                                r2 ← P.Alloc[[p,,PLUS[vi.listtail, vj.listtail]]]]];
                END
ELSE P.RErr["Binary operation on improper arguments"];
ans.Des.e ← TRUE;
P.RRS[rr];
END;

CK: PROCEDURE = BEGIN
IF StreamDefs.ControlDELtyped[] THEN
        BEGIN
        StreamDefs.ResetControlDEL;
        P.Interrupt;
        END;
END;

EmptyString: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
        RETURN[n.Type = STR AND n.Des.s = simp AND n.str.length = 0];
        END;

ZEval: PUBLIC PROCEDURE[n: Node] RETURNS[ans: Node] =
-- evaluate node in global environment
-- it probably contains a closure
-- the answer should overwrite the node (ans = n)
BEGIN
IF n.Des.e THEN ans←n
ELSE
        BEGIN
        n1: Node; -- don't register n1 because Eval always registers its arg
        n1 ← P.Alloc[n↑];
        ans ← Eval[n1, NIL];
        n↑ ← ans↑;
        ans ← n;
        END;
END;

Eval: PUBLIC PROCEDURE[n,env: Node] RETURNS[ans: Node] =
-- evaluate node in given environment
BEGIN
IF n.Des.e THEN RETURN[n] ELSE
BEGIN
rr: Register = P.MRS[];
r1, r2: Node ← NIL;
lop: Node ← Nail; -- not NIL, so Poplar debugger can print
rop: Node ← Nail;

ErrorProcess: PROCEDURE[mess, est: STRING] RETURNS [proceed: BOOLEAN] =
        BEGIN m: Node ← NIL;
        DispDefs.WF2["%s: %s, in:*n",mess,est];
        DispDefs.Print[P.Alloc[[CLOSURE,,CLOSURE[n, NIL]]]];
          -- closure is to keep from calling ZEval during print
        P.R[@m];
        DO
                m ← RouteDefs.KeyRoutine[NIL, P.MakeSTR["$"],NIL];
                IF m.Type = FAIL THEN P.EndDisplay
                ELSE IF EmptyString[m] THEN RETURN[FALSE]
                ELSE IF EQ[m,"pr"] THEN RETURN[TRUE]
                ELSE IF EQ[m,"left"] THEN DispDefs.Print[lop]
                ELSE IF EQ[m,"right"] THEN DispDefs.Print[rop]
                ELSE
                        BEGIN
                        m← P.Dist[m !
                        P.SErr =>
                                BEGIN
                                DispDefs.WF0["Syntax Error!*n"];
                                LOOP
                                END];
```

```
                        m ← Eval[m, env];
                        DispDefs.Print[m];
                        END;
                ENDLOOP;
        END; -- ErrorProcess

P.R2[@n, @env];
P.R2[@r1, @r2];
P.R2[@lop, @rop];

BEGIN ENABLE BEGIN
                        P.RErr => IF ErrorProcess["Run-time error", est] THEN RETRY;
                        P.PBug => IF ErrorProcess["Poplar Bug!!", est] THEN RETRY;
                        P.Interrupt => IF ErrorProcess["***Interrupt***", ""]
                        THEN RESUME
                        END;

DE: PROCEDURE =
        BEGIN
        lop ← Eval[n.arg1,env];
        rop ← Eval[n.arg2,env];
        END;

Enclose: PROCEDURE[n1: Node] RETURNS [Node] =
        BEGIN
        IF env = NIL THEN RETURN[n1]
        ELSE RETURN[P.Alloc[[CLOSURE,,CLOSURE[n1, env]]]];
        END;

CK;
DO
SELECT n.Type FROM
CLOSURE =>
        BEGIN
        env ← n.env;
        n ← n.exp;
        LOOP;
        END;
ID =>
        ans ← ZEval[Look[n.name,env]];
HOLE, PFUNC, ZARY, FAIL, STR, WILD =>
        ans ←n;
PATTERN =>
        ans ← P.Alloc[[CLOSURE,,CLOSURE[n,env]]];
FCN => -- "x: E"
        BEGIN
        IF n.parms.Type = EQUAL THEN
                -- handle an equality assertion
                BEGIN
                ElimEq: PROCEDURE[n:Node]
                                RETURNS[BOOLEAN] =
                        BEGIN
                        IF n.Type = EQUAL THEN n↑ ← n.left↑;
                        RETURN[TRUE];
                        END;
                lop ← P.Alloc[[APPLY,,APPLY[n.parms.right, n]]];
                n.parms↑ ← n.parms.left↑;
                [] ← Eval[lop, env];
                P.Preorder[n, ElimEq];
                END;
        ans ← P.Alloc[[CLOSURE,,CLOSURE[n,env]]];
        END;
UNARY => -- e.g., "write x"
        ans ← P.Alloc[[UNARY,,UNARY[n.unary,Eval[n.uexp,env]]]];
          -- to be really lazy, take eval of unary parameter out
PLUS,MINUS => -- "m + n", "m-n"
        BEGIN
        DE;
```

```
        ans ← Binary[r1 ← Enclose[lop], r2 ← Enclose[rop], n.Type];
        END;
SEQUENCE => -- "m - - n"
        BEGIN
                DE;
                BEGIN
                from: LONG INTEGER = P.MakeInteger[lop];
                to: LONG INTEGER ← P.MakeInteger[rop];
                inc: LONG INTEGER = IF from<to THEN 1 ELSE -1;
                IF from = to THEN
                        ans ← P.Alloc[[LIST,,LIST[
                                        lop,
                                        Nail]]]
                ELSE
                        BEGIN
                        ans ← P.Alloc[[LIST,,LIST[
                                        lop,
                                        P.Alloc[[SEQUENCE,,SEQUENCE[
                                                        P.MakeNUM[from + inc],
                                                        rop]]]]]];
                        END;
                END;
        END;
APPLY => -- "x/f"
        BEGIN
        lop ← Enclose[n.arg1];
        rop ← Eval[n.arg2, env];
        IF rop.Type = CLOSURE AND rop.exp.Type = FCN THEN
                -- handle tail recursion as a special case
                BEGIN
                env ← ExtendEnv[rop.env, rop.exp.parms, lop];
                n ← rop.exp.fcn;
                LOOP;
                END
        ELSE ans ← Apply[lop,rop];
        END;
MAPPLY => -- "x//f"
        BEGIN
        DE;
        IF lop.Type # LIST THEN P.RErr["Non-list before //"];
        IF lop.listhead = NIL THEN
                ans ← Nail
        ELSE
                ans ← P.Alloc[[LIST,,LIST[
                                r1 ← Enclose[P.Alloc[[APPLY,,APPLY[lop.listhead,rop]]]],
                                r2 ← Enclose[P.Alloc[[MAPPLY,,MAPPLY[lop.listtail,rop]]]]]]];
        END;
GOBBLE => -- "x///f"
        BEGIN
        DE;
        IF lop.Type # LIST THEN P.RErr["Non-list before ///"]
        ELSE IF lop.listhead = NIL THEN P.RErr["[] before ///"]
        ELSE
                BEGIN
                List2: PROCEDURE [a,b: Node] RETURNS [Node] =
                        -- return a list with a and b as elements
                        BEGIN
                        RETURN[P.Alloc[[LIST,,LIST[a, P.Alloc[[LIST,,LIST[b, Nail]]]]]]];
                        END;
                ans ← NIL; P.R[@ans];
                ans ← ZEval[lop.listhead];
                lop ← ZEval[lop.listtail];
                UNTIL lop.listhead = NIL DO
                        IF lop.Type # LIST THEN P.PBug["Malformed List"];
                        ans ← Apply[List2[ans, ZEval[lop.listhead]], rop];
                        lop ← ZEval[lop.listtail];
                        ENDLOOP;
                END;
```

```
        END;
ITER => -- "x%f"
        BEGIN
        ans ← NIL; P.R[@ans];
        lop ← Enclose[n.arg1];
        rop ← Eval[n.arg2, env];
                DO
                ans ← lop;
                lop ← Apply[lop, rop];
                IF lop.Type = FAIL THEN EXIT;
                ENDLOOP;
        ans ← ZEval[ans]; -- in case rop failed immediately
        END;
ASS => -- "x←E"
        BEGIN
        e: Node ← env;
        -- Parser has checked that s is an ID
        ans ← Eval[n.rhs, env];
        DO
                IF e = NIL THEN
                        BEGIN n.lhs.name.val ← ans; EXIT END; -- assume global
                IF e.val.unary = n.lhs.name THEN
                        BEGIN e.val.uexp ← ans; EXIT END;
                e ← e.next;
                ENDLOOP;
        END;
TILDE => -- "~E"
        BEGIN
        ans ← Eval[n.not,env];
        ans ← IF ans.Type = FAIL THEN MTSt ELSE Fail;
        END;
PROG => -- "E1; E2"
        BEGIN
        [] ← Eval[n.arg1,env];
        n ← n.arg2;
        LOOP;
        END;
MATCH => -- "E1 > E2"
        BEGIN
        ans ← Eval[n.div,env];
        IF ans.Type # FAIL THEN BEGIN n ← n.patt; LOOP END;
        END;
PALT => -- "E1 | E2"
        BEGIN
        ans ← Eval[n.alt1,env];
        IF ans.Type = FAIL THEN BEGIN n ← n.alt2; LOOP END;
        END;
CAT => -- "E1 E2"
        BEGIN
        DE;
        ans ← IF EmptyString[lop] THEN rop
                ELSE IF EmptyString[rop] THEN lop
                ELSE Binary[r1 ← Enclose[lop], r2 ← Enclose[rop], CAT];
        END;
LIST => -- "[x, ...]" the heart of lazy eval!
        IF n.listhead = NIL THEN ans ← n -- nail
        ELSE -- close elements and return, without evaluating
                ans ← P.Alloc[[LIST,,LIST[
                                r1 ← Enclose[n.listhead],
                                r2 ← Enclose[n.listtail]]]];
CATL => -- "x,,y"
        BEGIN
        lop ← Eval[n.left, env];
        IF lop.Type # LIST THEN P.RErr["The operation ,, is applied to a non-list"];
        IF lop.listhead = NIL THEN
                BEGIN
                ans ← Eval[n.right, env];
                IF ans.Type # LIST THEN P.RErr["The operation ,, is applied to a non-list"];
```

```
                END
        ELSE
                BEGIN
                ans ← P.Alloc[[LIST,,LIST[
                                r1 ← Enclose[lop.listhead],
                                r2 ← Enclose[P.Alloc[[CATL,,CATL[
                                                lop.listtail,
                                                n.right]]]]]];
                END;
        END;
EQUAL => -- "E1 = E2" equality comment
        BEGIN -- this will occur only in checking mode
        DE;
        IF ~Equal[lop,rop] THEN P.RErr["Equality Check"];
        n↑ ← n.left↑; -- avoid multiple checks
        ans ← lop;
        END;
ENDCASE => P.PBug["Unknown type"];
ans.Des.e ← TRUE;
P.RRS[rr];
RETURN;
ENDLOOP;
END;
END;
END; -- Eval

Map: PUBLIC PROCEDURE [l: Node, p: PROCEDURE[Node]] =
BEGIN
-- Map registers l, so caller needn't.
rr: Register ← P.MRS[];
P.R[@l];
FOR l ← l, ZEval[l.listtail] UNTIL l.listhead = NIL DO
        IF l.Type # LIST THEN P.PBug["Malformed List"];
        p[ZEval[l.listhead]];
        ENDLOOP;
P.RRS[rr];
END;

EQ: PROCEDURE[n: Node, s: STRING] RETURNS[BOOLEAN] = BEGIN i: CARDINAL;
IF P.Length[n] # s.length THEN RETURN[FALSE];
FOR i IN [0..s.length) DO
        IF s[i] # P.Sub[n,i] THEN RETURN[FALSE];
        ENDLOOP;
RETURN[TRUE];
END;

Equal: PROCEDURE[lx,rx: Node] RETURNS [res: BOOLEAN] =
        BEGIN
        DO
                IF lx.Type # rx.Type THEN res ← FALSE
                ELSE IF lx.Type = LIST THEN
                        BEGIN
                        IF lx.listhead = NIL THEN res ← rx.listhead = NIL
                        ELSE IF rx.listhead = NIL THEN res ← FALSE
                        ELSE IF ~Equal[ZEval[lx.listhead], ZEval[rx.listhead]] THEN
                                res ← FALSE
                        ELSE BEGIN
                                lx ← ZEval[lx.listtail];
                                rx ← ZEval[rx.listtail];
                                LOOP;
                                END
                        END
                ELSE IF lx.Type = STR THEN
                        BEGIN
                        rr: Register ← P.MRS[];
                        a,b: PLDefs.StreamRecord;
                        a.node ← b.node ← NIL;
                        P.R2[@a.node,@b.node];
```

```
                    a ← P.NewStream[lx];
                    b ← P.NewStream[rx];
                    DO
                        IF a.node = NIL THEN res ← b.node = NIL
                        ELSE IF b.node = NIL THEN res ← FALSE
                        ELSE IF P.Item[@a] ~ = P.Item[@b] THEN
                            res ← FALSE
                        ELSE LOOP;
                        EXIT;
                        ENDLOOP;
                    P.RRS[rr];
                    END
                ELSE P.RErr["Illegal equality check"];
                EXIT;
                ENDLOOP;
        END;

ExtendEnv: PROCEDURE[e,bv,val: Node] RETURNS [nev: Node] =
        -- val is not evaluated; e,bv fully evaluated
        BEGIN
        rr : Register = P.MRS[];
        nev ← e;
        P.R[@nev];
        IF bv.Type = ID THEN
            nev ← P.Alloc[[ENV,,ENV[
                    P.Alloc[[UNARY,,UNARY[bv.name, val]]],
                    nev]]]
        ELSE
            DO
                IF bv.Type # LIST THEN P.RErr["Malformed function"];
                IF ZEval[val].Type # LIST THEN P.RErr["Non-list before /"];
                IF bv.listhead = NIL THEN
                    BEGIN
                    IF val.listhead = NIL THEN EXIT;
                    P.RErr["Too many parameters for function"];
                    END;
                IF val.listhead = NIL THEN P.RErr["Too few parameters for function"];
                IF bv.listhead.Type # ID THEN P.RErr["Malformed function"];
                nev ← P.Alloc[[ENV,,ENV[
                        P.Alloc[[UNARY,,UNARY[
                                bv.listhead.name,
                                val.listhead -- evaluated at access time -- ]]],
                        nev]]];
                val ← ZEval[val.listtail];
                bv ← bv.listtail;
                ENDLOOP;
        P.RRS[rr];
        END;

LengthList: PUBLIC PROCEDURE[n: Node] RETURNS[i:CARDINAL] = BEGIN
IF n.Type ~ = LIST THEN P.PBug["LengthList expects a list"];
i ← 0;
WHILE n.listhead ~ = NIL DO
        i ← i + 1;
        n ← ZEval[n.listtail];
        ENDLOOP;
END;

Look: PROCEDURE[n: Symbol, e: Node] RETURNS[Node] = BEGIN
FOR e ← e, e.next UNTIL e = NIL DO
        IF e.val.unary = n THEN RETURN[e.val.uexp]
        ENDLOOP;
IF ~(n.type = string AND n.val.Type # UNDEFINED) THEN P.RErr["Undefined variable"];
RETURN[n.val];
END;

Match: PROCEDURE[subject, pattern, env: Node] RETURNS [struc: Node] = BEGIN
```

```
rr: Register = P.MRS[];
s: PLDefs.StreamRecord;
HolePassed: SIGNAL [holeString: Node] = CODE;
M2: PROCEDURE[p: Node, discarding: BOOLEAN,env: Node] RETURNS [ans: Node] =
        -- The state of s is an implict input and output of M2
        BEGIN
        rr: Register ← P.MRS[];
        CK;
        ans ← NIL; P.R3[@p, @env, @ans];
                DO
                IF p.Type = PATTERN THEN p ← p.pattern
                ELSE IF p.Type = ID THEN p ← ZEval[Look[p.name, env]]
                ELSE IF p.Type = CLOSURE THEN
                        BEGIN env ← p.env; p ← p.exp END
                ELSE EXIT;
                ENDLOOP;
        SELECT p.Type FROM
                PFUNC => BEGIN
                        sym: Symbol = p.pfunc1;
                        ans ← sym.patproc[sym, @s, NIL];
                        ans.Des.e ← TRUE;
                        END;
                PFUNC1 => BEGIN
                        sym: Symbol = p.pfunc1;
                        ans ← Eval[p.pexp, env];
                        ans ← sym.patproc[sym, @s, ans];
                        ans.Des.e ← TRUE;
                        END;
                MATCH, APPLY, MAPPLY, GOBBLE, ITER => BEGIN
                                ans ← M2[p.div, discarding, env];
                                IF ans.Type # FAIL AND ~discarding THEN
                                        BEGIN
                                        ans ← P.Alloc[[p.Type,, MATCH[ans,p.patt]]];
                                        IF ans.div.Des.e THEN ans ← Eval[ans,env];
                                        END;
                                END;
                TILDE => BEGIN
                                s1: PLDefs.StreamRecord ← s;
                                P.R[@s1.node];
                                ans ← M2[p.not, TRUE, env];
                                s ← s1;
                                ans ← IF ans.Type = FAIL THEN MTSt ELSE Fail;
                                END;
                DELETE => BEGIN
                                ans ← M2[p.delete, TRUE, env];
                                IF ans.Type # FAIL THEN ans ← MTSt;
                                END;
                STR => BEGIN
                                s1: PLDefs.StreamRecord ← s;
                                I1, i: LONG INTEGER ← 0;
                                I1 ← P.Length[p];
                                P.R[@s1.node];
                                UNTIL i = I1
                                        DO
                                        IF s.node = NIL OR P.Item[@s] # P.Sub[p, i] THEN
                                                BEGIN ans ← Fail; EXIT END;
                                        i ← i + 1;
                                        REPEAT
                                        FINISHED =>
                                                IF discarding THEN ans ← MTSt
                                                ELSE BEGIN
                                                        ans ← P.SubStringStream[@s1,0,i]; ans.Des.e ← TRUE;
                                                        END;
                                        ENDLOOP;
                                END;
                WILD => BEGIN
                                IF s.node = NIL THEN ans ← Fail
                                ELSE BEGIN
```

```
                    IF discarding THEN ans ← MTSt
                    ELSE BEGIN
                            ans ← P.SubStringStream[@s,0,1];
                            ans.Des.e ← TRUE;
                            END;
                    [] ← P.Item[@s];
                    END;
                END;
HOLE => BEGIN
                ans ← P.Alloc[MTSt↑]; -- empty string, to be overwritten
                ans.Des.e ← FALSE;
                SIGNAL HolePassed[ans];
                END;
CAT, PLUS, MINUS, LIST, CATL => BEGIN
                IF p.Type = LIST AND p.listhead = NIL THEN
                        ans ← p
                ELSE IF p.Type = LIST
                        AND p.listtail.listhead = NIL THEN
                        BEGIN -- not really binary
                        ans ← M2[p.left, discarding, env];
                        IF ans.Type # FAIL AND ~discarding THEN
                                BEGIN
                                ans ← P.Alloc[[LIST,,LIST[ans,Nail]]];
                                ans.Des.e ← ans.left.Des.e;
                                END;
                        END
                ELSE BEGIN
                holeNode, struc2: Node ← NIL;
                P.R2[@holeNode, @struc2];
                ans ← M2[p.left, discarding, env!
                                HolePassed =>
                                                BEGIN
                                                holeNode ← holeString;
                                                RESUME
                                                END];
                IF ans.Type # FAIL THEN
                        BEGIN
                        IF holeNode = NIL THEN struc2 ← M2[p.right, discarding, env]
                        ELSE BEGIN -- unanchored match allowed
                                i: CARDINAL ← 0;
                                s1, s2: PLDefs.StreamRecord ← s;
                                P.R[@s1.node];
                                        DO
                                        struc2 ← M2[p.right, discarding, env];
                                        IF struc2.Type # FAIL THEN
                                                BEGIN
                                                IF discarding THEN EXIT;
                                                holeNode↑ ← P.SubStringStream[@s1,0,i]↑;
                                                ans←Eval[ans,env];
                                                EXIT;
                                                END;
                                        IF s2.node = NIL THEN EXIT;
                                        [] ← P.Item[@s2];
                                        s ← s2;
                                        i ← i + 1;
                                        ENDLOOP;
                                END;
                        IF struc2.Type = FAIL THEN ans ← Fail
                        ELSE IF discarding THEN ans ← MTSt
                        ELSE BEGIN
                                ans ← P.Alloc[[p.Type,,CAT[ans,struc2]]];
                                IF ans.left.Des.e
                                AND ans.right.Des.e THEN
                                        ans←Eval[ans,env];
                                END;
                        END;
                END;
                END;
```

```
                SEQOF => BEGIN
                        pans: POINTER TO Node ← @ans;
                        tans: Node;
                        s1: PLDefs.StreamRecord;
                        P.R2[@tans, @s1.node];
                        ans ← Fail;
                                DO
                                s1 ← s;
                                tans ← M2[p.seqof, discarding, env !
                                        HolePassed => BEGIN holeString.Des.e ← TRUE; RESUME END]; -- The
                                        user probably didn't mean to put "..." at the end of a seq, but what can we
                                        do?
                                IF tans.Type = FAIL THEN BEGIN s ← s1; EXIT END;
                                IF discarding THEN ans ← MTSt
                                ELSE IF pans↑.Type = FAIL THEN pans↑ ← tans
                                ELSE pans↑ ← Binary[pans↑, tans, CAT];
                                pans.Des.e ← TRUE;
                                IF pans.Type = STR AND pans.Des.s = cat THEN pans ← @pans.str.n2;
                                -- This last statement should encourage right linear trees.
                                ENDLOOP;
                        END;

                SEQOFC => BEGIN
                        pans: POINTER TO Node ← @ans;
                        tans: Node;
                        s1: PLDefs.StreamRecord;
                        P.R2[@tans, @s1.node];
                        ans ← Fail;
                                DO
                                s1 ← s;
                                tans ← M2[p.seqof, discarding, env !
                                        HolePassed => BEGIN holeString.Des.e ← TRUE; RESUME END];
                                IF tans.Type = FAIL THEN BEGIN s ← s1; EXIT END;
                                IF discarding THEN ans ← MTSt
                                ELSE BEGIN
                                        pans↑ ← P.Alloc[[LIST,,LIST[tans, Nail]]];
                                        pans.Des.e ← TRUE;
                                        pans ← @pans.listtail;
                                        END;
                                ENDLOOP;
                        END;

                OPT => BEGIN -- (optional part) has single pattern as part
                        s1: PLDefs.StreamRecord ← s;
                        P.R[@s1.node];
                        ans ← M2[p.opt, discarding, env];
                        IF ans.Type = FAIL THEN BEGIN ans ← MTSt; s ← s1 END;
                        END;

                PALT => BEGIN
                        s1: PLDefs.StreamRecord ← s;
                        P.R[@s1.node];
                        ans ← M2[p.alt1, discarding, env];
                        IF ans.Type = FAIL THEN BEGIN s ← s1; ans ← M2[p.alt2, discarding, env] END;
                        END;
                FAIL => ans ← Fail;
                ENDCASE => P.RErr["Unknown pattern type"];
        P.RRS[rr];
        END;
holeNode: Node ← NIL;
struc ← NIL;
s ← P.NewStream[subject];
P.R3[@struc, @holeNode, @s.node];
struc ← M2[pattern, FALSE, env ! HolePassed => BEGIN holeNode ← holeString; RESUME END];
IF holeNode # NIL THEN
        BEGIN
        holeNode↑ ← P.ConvertStream[@s]↑;
        struc ← Eval[struc,env];
```

```
        END
ELSE IF s.node # NIL THEN struc ← Fail;
struc.Des.e ← TRUE;
P.RRS[rr];
END; -- Match

[Fail,MTSt,Nail] ← P.GetSpecialNodes[];
END.
```

```
--To avoid WF in ThreeWayCore

DIRECTORY
    ControlDefs: FROM "ControlDefs",
      IODefs: FROM "IODefs" USING [WriteChar],
    StringDefs: FROM "StringDefs" USING [AppendChar, StringToDecimal],
    VMDefs: FROM "VMDefs",
    InlineDefs: FROM "InlineDefs",
    DispDefs: FROM "DispDefs";

NWF: PROGRAM IMPORTS IODefs, StringDefs, VM: VMDefs EXPORTS DispDefs =
BEGIN
--
Unbound: UNSPECIFIED = ControlDefs.ControlLinkTag[unbound];
LongCARDINAL: TYPE = InlineDefs.LongCARDINAL;

procArray: ARRAY[1 .. 26] OF PROCEDURE[UNSPECIFIED,STRING];
saveProcArray: ARRAY[1 .. 26] OF PROCEDURE[UNSPECIFIED,STRING];
WC: PROCEDURE[CHARACTER] ← IODefs.WriteChar;
TheString: STRING;
Temp: CARDINAL;
fillchar: CHARACTER ← ' ;

nparam: ARRAY [1 .. 5] OF UNSPECIFIED;

WFError: PUBLIC SIGNAL[STRING,CARDINAL] = CODE;

WFG: PROCEDURE [s: STRING, c: CARDINAL] =
    BEGIN
    form: STRING ← [10];
    n,z,i,pnum: CARDINAL;
    ch: CHARACTER;
    f: CARDINAL;
    p: PROCEDURE[UNSPECIFIED,STRING];
    param: ARRAY [1 .. 5] OF UNSPECIFIED;

    FOR i IN [1 .. c] DO param[i] ← nparam[i]; ENDLOOP;
    pnum ← 0;
        BEGIN
        FOR i IN [0 .. s.length) DO
            SELECT s[i] FROM
            '% => BEGIN
                i ← i + 1;
                pnum ← pnum + 1;
                f ← 0;
                WHILE s[i] = '- OR s[i] IN ['0 .. '9] DO
                    form[f] ← s[i]; i ← i + 1; f←f + 1; ENDLOOP;
                form.length ← f;
                -- s[i] is a control character, and form is the stuff beteen % and s[i]
                ch ← s[i];
                IF ch IN ['A .. 'Z] THEN ch← ch + 40B;
                IF ch IN ['a .. 'z] THEN BEGIN
                    p ← (procArray[LOOPHOLE[ch,CARDINAL]-140B]);
                    IF LOOPHOLE[p, UNSPECIFIED] ~= Unbound THEN p[param[pnum],form]
                    ELSE WC[ch];
                    END;
                IF ch = '% THEN BEGIN
                        WC['%];
                        pnum ← pnum - 1;
                        END
                ELSE IF pnum > c THEN GOTO bad;
                END;
            '* => BEGIN
                i ← i + 1;
                SELECT s[i] FROM
                    'N, 'R, 'n,'r => WC[15C];
                    'B,'b => WC[10C];
                    'T,'t => WC[11C];
                    'F,'f => WC[14C];
                    IN ['0..'9] => BEGIN   --- octal constant, exactly 3 digits
                        IF s[i + 1] IN ['0 .. '9] AND s[i + 2] IN ['0 .. '9] THEN BEGIN
                            z ← LOOPHOLE['0];
```

```
                                n ← (LOOPHOLE[s[i],CARDINAL]-z)*64;
                                n←n + (LOOPHOLE[s[i + 1],CARDINAL]-z)*8;
                                n←n + LOOPHOLE[s[i + 2],CARDINAL]-z;
                                WC[LOOPHOLE[n]];
                                i ← i + 2;
                                END
                            ELSE SIGNAL WFError["Bad character to WF",c];
                            END;
                        ENDCASE => WC[s[i]];
                    END;
                ENDCASE => WC[s[i]];
                ENDLOOP;
            IF pnum < c THEN GOTO bad;
            EXITS
            bad => SIGNAL WFError["Wrong # of parameters to WF",c];
            END;
        RETURN;
        END;


SetCode: PUBLIC PROCEDURE[char: CHARACTER, p: PROCEDURE[d: UNSPECIFIED, form: STRING]] =
    BEGIN
    IF char IN ['A .. 'Z] THEN char ← char + 40B;
    IF char ~IN ['a .. 'z] THEN SIGNAL WFError["Invalid SetCode",0];
    procArray[LOOPHOLE[char,CARDINAL]-140B] ← p;
    END;

ResetCode: PUBLIC PROCEDURE[char: CHARACTER] =
    BEGIN
    i: [1 .. 26];
    i ← LOOPHOLE[char,CARDINAL] - 140B;
    procArray[i] ← saveProcArray[i];
    END;

WriteToString: PUBLIC PROCEDURE[s: STRING] =
    BEGIN
    WC ← GoToString;
    TheString ← s;
    END;

GoToString: PROCEDURE[ch: CHARACTER] =
    BEGIN
    StringDefs.AppendChar[TheString,ch];
    END;

ostring: PROCEDURE[d: CARDINAL,s:STRING,i:CARDINAL,b: CARDINAL,p: CARDINAL] RETURNS[CARDINAL] =
    BEGIN
    z: CARDINAL;
    j: CARDINAL ← p;
    IF d >= b THEN j ← ostring[d/b,s,i + 1,b,p]
    ELSE s.length ← i + 1;
    z ← d MOD b;
    s[j] ← IF z IN [10 .. 15] THEN (z-10) + 'A ELSE z + '0;
    RETURN[j + 1];
    END;

lostring: PROCEDURE[d: LONG INTEGER,s:STRING,i:CARDINAL,b: CARDINAL,p: CARDINAL] RETURNS[CARDINAL] =
    BEGIN
    z: CARDINAL;
    zz: LONG INTEGER;
    j: CARDINAL ← p;
    IF d >= b THEN j ← lostring[d/b,s,i + 1,b,p]
    ELSE s.length ← i + 1;
    zz ← d MOD LONG[b];
    z ← VM.Cardinal[zz];
    s[j] ← IF z IN [10 .. 15] THEN (z-10) + 'A ELSE z + '0;
    RETURN[j + 1];
    END;

nstring: PROCEDURE[d: INTEGER,s: STRING,b: CARDINAL] =
    BEGIN
    c: CARDINAL;
```

```
    i: CARDINAL ← 0;
    IF d < 0 THEN BEGIN
        s[0] ← '-;
        c ← -d;
        i ← i + 1;
        END
    ELSE c ← d;
    [] ← ostring[c,s,i,b,i];
    END;

Instring: PROCEDURE[dd: POINTER TO LONG INTEGER,s: STRING,b: CARDINAL] =
    BEGIN
    d: LONG INTEGER;
    c: LONG INTEGER;
    i: CARDINAL ← 0;
    d ← dd↑;
    IF d < 0 THEN BEGIN
        s[0] ← '-;
        c ← -d;
        i ← i + 1;
        END
    ELSE c ← d;
    [] ← lostring[c,s,i,b,i];
    END;

BRoutine: PROCEDURE[d: UNSPECIFIED, form: STRING] =
    BEGIN
    s: STRING ← [20];
    [] ← ostring[d,s,0,8,0];
    printit[s,form];
    END;

DRoutine: PROCEDURE[d: UNSPECIFIED, form: STRING] =
    BEGIN
    s: STRING ← [20];
    nstring[d,s,10];
    printit[s,form];
    END;

XRoutine: PROCEDURE[d: UNSPECIFIED, form: STRING] =
    BEGIN
    s: STRING ← [20];
    [] ← ostring[d,s,0,16,0];
    printit[s,form];
    END;

CRoutine: PROCEDURE[d: UNSPECIFIED, form: STRING] =
    BEGIN
    ch: CHARACTER ← d;
    s: STRING ← [20];
    s[0] ← ch;
    s.length ← 1;
    printit[s,form];
    END;

URoutine: PROCEDURE[d: UNSPECIFIED, form: STRING] =
    BEGIN
    s: STRING ← [20];
    [] ← ostring[d,s,0,10,0];
    printit[s,form];
    END;

IRoutine: PROCEDURE[d: UNSPECIFIED, form: STRING] = BEGIN
    s: STRING ← [20];
    Instring[d,s,10];
    printit[s,form];
    END;

LRoutine: PROCEDURE[d: UNSPECIFIED, form: STRING] =
    BEGIN
    D: num InlineDefs.LongNumber;
    p: POINTER TO num InlineDefs.LongNumber;
```

```
    i,j,k: INTEGER;
    a: CARDINAL;
    form ← form;
    p ← d;
    D ← p↑;
    i ← D.lowbits;
    IF i < 0 THEN BEGIN a ← 1; i ← i - 100000B END ELSE a ← 0;
    j ← D.highbits;
    k ← 0;
    IF j < 0 THEN k ← 2;
    j ← D.highbits*2 + a;
    IF j < 0 THEN BEGIN
            j ← j - 100000B;
            k ← k + 1;
            END;
    fillchar ← '0;                          -- kludge
    IF k = 0 THEN BEGIN
            IF j = 0 THEN WF1["%b",i]
                    ELSE WF2["%b%5b",j,i]
            END
    ELSE WF3["%b%5b%5b",k,j,i];
    fillchar ← ' ;                          -- unkludge
    END;

printit: PROCEDURE[d: UNSPECIFIED,form: STRING] =
    BEGIN
    ladj: BOOLEAN ← FALSE;
    w: CARDINAL;
    k: INTEGER;
    s: STRING ← d;
    j: CARDINAL;
    IF s = NIL THEN BEGIN WC['{]; WC['N]; WC['I]; WC['L]; WC['}]; RETURN END;
    IF form.length > 0 THEN BEGIN
        IF form[0] = '- THEN BEGIN form[0] ← '0; ladj ← TRUE; END;
        w ← StringDefs.StringToDecimal[form];
        END
    ELSE w ← s.length;
    -- w is field width
    k ← w - s.length;
    k ← MAX[0, k];
    IF ~ladj THEN THROUGH [1..k] DO WC[fillchar] ENDLOOP;
    FOR j IN [0 .. MIN[w,s.length]) DO WC[s[j]] ENDLOOP;
    IF ladj THEN THROUGH [1..k] DO WC[fillchar] ENDLOOP;
    END;

WF0: PUBLIC PROCEDURE [s: STRING] =
    BEGIN
    WFG[s,0];
    END;

WF1: PUBLIC PROCEDURE [s: STRING, a: UNSPECIFIED] =
    BEGIN
    nparam[1] ← a;
    WFG[s,1];
    END;

WF2: PUBLIC PROCEDURE [s: STRING, a,b: UNSPECIFIED] =
    BEGIN
    nparam[1] ← a; nparam[2] ← b;
    WFG[s,2];
    END;

WF3: PUBLIC PROCEDURE [s: STRING, a,b,c: UNSPECIFIED] =
    BEGIN
    nparam[1] ← a; nparam[2] ← b; nparam[3] ← c;
    WFG[s,3];
    END;

WF4: PUBLIC PROCEDURE [s: STRING, a,b,c,d: UNSPECIFIED] =
    BEGIN
    nparam[1] ← a; nparam[2] ← b; nparam[3] ← c; nparam[4] ← d;
    WFG[s,4];
```

```
    END;


-- INITIALIZATION CODE
FOR Temp IN [1 .. 26]
    DO
    saveProcArray[Temp] ← procArray[Temp] ← Unbound;
    ENDLOOP;
SetCode['b,BRoutine];
saveProcArray[LOOPHOLE['b,CARDINAL]-140B] ← BRoutine;
SetCode['c,CRoutine];
saveProcArray[LOOPHOLE['c,CARDINAL]-140B] ← CRoutine;
SetCode['d,DRoutine];
saveProcArray[LOOPHOLE['d,CARDINAL]-140B] ← DRoutine;
SetCode['i,IRoutine];
saveProcArray[LOOPHOLE['i,CARDINAL]-140B] ← IRoutine;
SetCode['l,LRoutine];
saveProcArray[LOOPHOLE['l,CARDINAL]-140B] ← LRoutine;
SetCode['s,printit];
saveProcArray[LOOPHOLE['s,CARDINAL]-140B] ← printit;
SetCode['u,URoutine];
saveProcArray[LOOPHOLE['u,CARDINAL]-140B] ← URoutine;
SetCode['x,XRoutine];
saveProcArray[LOOPHOLE['x,CARDINAL]-140B] ← XRoutine;
END.
```

MODULE HISTORY

Created by Schmidt, July 1977

Changed by Schmidt, August 19, 1977 8:06 PM
Reason: to delete wf5 - wf9, put in setwriteprocedure, and add a test for a NIL string

Changed by Schmidt, August 19, 1977 8:23 PM
Reason: deconvert from dboss

Changed by Mitchell, June 13, 1978 9:48 PM
Reason: Convert to Mesa 4.0

Chnaged by Schmidt, June 26, 1978 11:21 PM
Reason: add %i, %l to handle 32-bit integers

```
                                                    DIRECTORY
                                                    PLDefs: FROM "PLDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    ImageDefs: FROM "ImageDefs",
                                                    IODefs: FROM "IODefs",
                                                    StringDefs: FROM "StringDefs",
                                                    MiscDefs: FROM "miscdefs",
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    SystemDefs: FROM "SystemDefs";

parse: PROGRAM IMPORTS DispDefs, P:PLDefs EXPORTS PLDefs = BEGIN
--
TokType: TYPE = PLDefs.TokType;
NodeType: TYPE = PLDefs.NodeType;
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
Register: TYPE = PLDefs.Register;
--
progstr: PLDefs.StreamRecord;
currn: Node;
TokTok,PeekTok: TokType;
TokVal, PeekVal: UNSPECIFIED;
nchar: CHARACTER;
stcnt, linecnt, charcnt: CARDINAL;
Fail,MTSt,Nail: Node;
savinput: PLDefs.StreamRecord;
checking: BOOLEAN ← FALSE; -- set by CheckRoutine, used by BinaryExp and FunctionBody
-- dist: prog EOF
-- prog: stmt { SEP prog } | stmt SEP
-- stmt: var ASS stmt | exp
-- ChoiceExp: ThenExp {PALT ThenExp}
-- ThenExp: BinaryExp {RARR BinaryExp}
-- BinaryExp: BinaryExp {(DIV | MAPPLY| GOBBLE| ITER | PLUS | MINUS | CATL | SEQUENCE | EQUAL) PrefixExp} | BinaryExp
PrefixExp
-- PrefixExp: UNARY PrefixExp | PFUNC1 PrefixExp | MINUS PrefixExp | TILDE PrefixExp | SimpleExp {SEQOF | SEQOFC | OPT |
DELETE}
-- SimpleExp: STR | ID | ZARY | PFUNC | SCREEN | WILD | FAIL | HOLE | LB RB | LB prog RB | LC stmt RC | ID COLON { =
PrefixExp ;}prog | LB prog RB COLON { = PrefixExp ;} prog
--     | LP stmt RP
--
-- Convention: peektok is the first token for each of the routines, e.g.
--  peektok = STR for the Base

CheckRoutine: PROCEDURE[sym: Symbol, prog, n2: Node]
            RETURNS [ans: Node] =
        BEGIN
        rr: Register ← P.MRS[];
        Ch: PROCEDURE[n:Node] RETURNS[BOOLEAN] =
            BEGIN
            IF n.Type = FCN AND n.parms.Type = EQUAL THEN
                BEGIN
                [] ← P.Eval[n, NIL];
                END;
            RETURN[TRUE];
            END;
        Ch1: PROCEDURE[n:Node] RETURNS[BOOLEAN] =
            BEGIN
            IF n.Type = EQUAL THEN
                BEGIN
                DispDefs.WF0["Missed Equality Check*n"];
                DispDefs.Print[n];
                END;
            RETURN[TRUE];
            END;
        P.R[@prog];
        IF prog.Type # STR THEN P.RErr["input to check not string"];
        checking ← TRUE;
        prog ← Dist[prog ! UNWIND => checking ← FALSE];
        checking ← FALSE;
        [] ← P.Eval[prog, NIL];
        P.Preorder[prog, Ch];
        P.Preorder[prog, Ch1];
```

```
        ans ← Nail;
        P.RRS[rr];
        END;


Dist: PUBLIC PROCEDURE [p: Node] RETURNS[Node] = BEGIN
-- this is the kickoff routine - call only once
-- p is the node to which has the string to be compiled
n: Node ← NIL;
rr: Register ← P.MRS[];
progstr ← P.NewStream[p];
P.R3[@progstr.node,@n,@savinput.node];
savinput ← progstr;
charcnt ← stcnt ← linecnt ← 1;
nchar ← ' ;
GetTok;                                    -- set up peek vals
n ← Prog[];
IF PeekTok ~ = EOF THEN P.SErr["Parser expected EOF"L];
P.Preorder[n,CheckPattern];
P.RRS[rr];
RETURN[n];
END;

CheckPattern: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
t: PLDefs.NodeType;
t ← n.Type;
IF t = PATTERN THEN RETURN[FALSE];              -- no sons are searched
IF t = PFUNC OR t = PFUNC1 OR t = SEQOF OR t = SEQOFC OR t = OPT OR t = WILD OR t = HOLE THEN P.SErr["Pattern
operator not surrounded by { }"L];
RETURN[TRUE];
END;

Prog: PROCEDURE RETURNS [res: Node] = BEGIN
rr: Register ← P.MRS[];
res ← NIL;
P.R[@res];
res ← Stmt[];
IF PeekTok = SEP THEN BEGIN
        GetTok;
        IF PeekTok ~ = EOF THEN res ← P.Alloc[[PROG,,PROG[res,Prog[]]]];
        END;
P.RRS[rr];
END;

Stmt: PROCEDURE RETURNS[i: Node] = BEGIN
rr: Register ← P.MRS[];
i ← ChoiceExp[];
P.R[@i];
IF PeekTok = ASS THEN BEGIN
        IF i.Type # ID THEN P.SErr["Missing ; or assignment to non-variable"L];
        GetTok;
        i ← P.Alloc[[ASS,,ASS[i,Stmt[]]]];
        END;
P.RRS[rr];
RETURN[i];
END;

-- this parses left-assoc instead of right assoc.
ChoiceExp: PROCEDURE RETURNS[Node] = BEGIN
i,j: Node ← NIL;
rr: Register ← P.MRS[];
P.R2[@i,@j];
IF PeekTok = PALT THEN i ← currn
ELSE i ← ThenExp[];
WHILE PeekTok = PALT DO
        GetTok;
        j ← ThenExp[];
        i ← P.Alloc[[PALT,,PALT[i,j]]]
        ENDLOOP;
P.RRS[rr];
RETURN[i];
```

```
END;

ThenExp: PROCEDURE RETURNS[Node] = BEGIN
i,j: Node ← NIL;
rr: Register ← P.MRS[];
P.R2[@i,@j];
IF PeekTok = RARR THEN i ← currn
ELSE i ← BinaryExp[];
WHILE PeekTok = RARR DO
        GetTok;
        j ← BinaryExp[];
        i ← P.Alloc[[MATCH,,MATCH[i,j]]]
        ENDLOOP;
P.RRS[rr];
RETURN[i];
END;

BinaryExp: PROCEDURE RETURNS[i: Node] = BEGIN
loop: BOOLEAN ← TRUE;
rr: Register ← P.MRS[];
p: PLDefs.TokType ← PeekTok;
i ← IF p = DIV OR p = MAPPLY OR p = GOBBLE OR p = ITER OR p = PLUS OR p = CATL OR p = SEQUENCE OR p = PLUS OR p
 = CATL OR p = SEQUENCE THEN currn ELSE PrefixExp[];
P.R[@i];
WHILE loop DO
        loop ← TRUE;
        SELECT PeekTok FROM
        DIV => BEGIN
                GetTok;
                i ← P.Alloc[[APPLY,,APPLY[i,PrefixExp[]]]];
                END;
        MAPPLY => BEGIN
                GetTok;
                i ← P.Alloc[[MAPPLY,,MAPPLY[i,PrefixExp[]]]];
                END;
        GOBBLE => BEGIN
                GetTok;
                i ← P.Alloc[[GOBBLE,,GOBBLE[i,PrefixExp[]]]];
                END;
        ITER => BEGIN
                GetTok;
                i ← P.Alloc[[ITER,,ITER[i,PrefixExp[]]]];
                END;
        PLUS => BEGIN
                GetTok;
                i ← P.Alloc[[PLUS,,PLUS[i,PrefixExp[]]]];
                END;
        MINUS => BEGIN
                GetTok;
                i ← P.Alloc[[MINUS,,MINUS[i,PrefixExp[]]]];
                END;
        CATL => BEGIN
                GetTok;
                i ← P.Alloc[[CATL,,CATL[i,PrefixExp[]]]];
                END;
        SEQUENCE => BEGIN
                GetTok;
                i ← P.Alloc[[SEQUENCE,,SEQUENCE[i,PrefixExp[]]]];
                END;
        EQUAL => BEGIN
                GetTok;
                IF checking THEN
                        i ← P.Alloc[[EQUAL,,EQUAL[i,PrefixExp[]]]]
                ELSE [] ← PrefixExp[];
                END;
        ENDCASE => BEGIN
                -- check to see if this is a cat
                -- the list below must be kept up to date.
                -- It is those things in First[PreFixExp]
                p ← PeekTok;
                IF p = ID OR p = STR OR p = UNARY OR p = PFUNC1 OR p = LB OR p = LC OR p = LP OR p =
```

```
                TILDE OR p = MINUS OR p = ZARY OR p = PFUNC OR p = SCREEN OR p = HOLE OR p = WILD
                OR p = FAIL THEN i ← P.Alloc[[CAT,,CAT[i,PrefixExp[]]]]
                ELSE loop ← FALSE;
                END;
        ENDLOOP;
P.RRS[rr];
RETURN[i];
END;

PrefixExp: PROCEDURE RETURNS[i:Node] = BEGIN
s: Symbol;
rr: Register;
IF PeekTok = UNARY THEN BEGIN
        GetTok;
        s ← TokVal;
        RETURN[P.Alloc[[UNARY,,UNARY[s,PrefixExp[]]]]];
        END;
IF PeekTok = PFUNC1 THEN BEGIN
        GetTok;
        s ← TokVal;
        RETURN[P.Alloc[[PFUNC1,,PFUNC1[s,PrefixExp[]]]]];
        END;
IF PeekTok = TILDE THEN BEGIN
        GetTok;
        RETURN[P.Alloc[[TILDE,,TILDE[PrefixExp[]]]]];
        END;
IF PeekTok = MINUS THEN BEGIN
        GetTok;
        i ← P.MakeNUM[0];
        rr ← P.MRS[];
        P.R[@i];
        i ← P.Alloc[[MINUS,,MINUS[i,PrefixExp[]]]];
        P.RRS[rr];
        RETURN;
        END;
i ← SimpleExp[];
WHILE PeekTok = SEQOF OR PeekTok = SEQOFC OR PeekTok = OPT OR PeekTok = DELETE DO
        IF PeekTok = SEQOF THEN BEGIN
                GetTok;
                i ← P.Alloc[[SEQOF,,SEQOF[i]]];
                END
        ELSE IF PeekTok = SEQOFC THEN BEGIN
                GetTok;
                i ← P.Alloc[[SEQOFC,,SEQOFC[i]]];
                END
        ELSE IF PeekTok = OPT THEN BEGIN
                GetTok;
                i ← P.Alloc[[OPT,,OPT[i]]];
                END
        ELSE IF PeekTok = DELETE THEN BEGIN
                GetTok;
                i ← P.Alloc[[DELETE,,DELETE[i]]];
                END;
        ENDLOOP;
END;

SimpleExp: PROCEDURE RETURNS[t: Node] = BEGIN
pans: POINTER TO Node;
SELECT PeekTok FROM
STR =>      BEGIN GetTok; RETURN[TokVal]; END;
ID =>       BEGIN
        rr: Register← P.MRS[];
        GetTok;
        t ← P.Alloc[[ID,,ID[TokVal]]];
        P.R[@t];
        IF PeekTok = COLON THEN t ← FunctionBody[t];
        P.RRS[rr];
        RETURN;
        END;
ZARY => BEGIN GetTok; RETURN[P.Alloc[[ZARY,,ZARY[TokVal]]]]; END;
PFUNC => BEGIN GetTok; RETURN[P.Alloc[[PFUNC,,PFUNC[TokVal]]]]; END;
```

```
SCREEN => BEGIN GetTok; RETURN[currn] END;
HOLE => BEGIN GetTok; RETURN[P.Alloc[[HOLE,,HOLE[]]]]; END;
WILD => BEGIN GetTok; RETURN[P.Alloc[[WILD,,WILD[]]]]; END;
FAIL => BEGIN GetTok; RETURN[Fail]; END;
LB =>       BEGIN
          rr: Register← P.MRS[];
          GetTok;
          IF PeekTok = RB THEN BEGIN
                    GetTok;
                    RETURN[Nail];
                    END;
          t ← P.Alloc[[LIST,,LIST[Prog[],Nail]]];
          P.R[@t];
          pans ← @t.listtail;
          WHILE PeekTok = COMMA DO
                    GetTok;
                    pans↑ ← P.Alloc[[LIST,,LIST[Prog[],Nail]]];
                    pans ← @pans.listtail;
                    ENDLOOP;
          IF PeekTok ~ = RB THEN P.SErr["Missing ']'"L];
          GetTok;
          IF PeekTok = COLON THEN t ← FunctionBody[t];
          P.RRS[rr];
          RETURN;
          END;
LP =>       BEGIN          -- used solely for parenthesization
          GetTok;
          t ← Prog[];
          IF PeekTok ~ = RP THEN P.SErr["Parser expected ')'"L];
          GetTok;
          RETURN[t];
          END;
LC =>       BEGIN
          GetTok;
          t ← Prog[];
          IF PeekTok ~ = RC THEN P.SErr["Parser expected '}'"L];
          GetTok;
          RETURN[P.Alloc[[PATTERN,,PATTERN[t]]]];
          END;
ENDCASE;
P.SErr["Parser did not recognize Simple Expression"L];
END;

FunctionBody: PROCEDURE[bv: Node] RETURNS [b: Node] =
          BEGIN
          GetTok;
          IF PeekTok = EQUAL THEN
                    BEGIN
                    rr: Register = P.MRS[];
                    testVal: Node ← NIL;
                    b ←NIL;
                    P.R2[@testVal,@b];
                    GetTok;
                    testVal ← PrefixExp[];
                    IF PeekTok # SEP THEN P.SErr["Missing ; after : = "];
                    GetTok;
                    IF checking THEN
                              bv ← P.Alloc[[EQUAL,,EQUAL[bv,testVal]]];
                    P.RRS[rr];
                    END;
          b ← P.Alloc[[FCN,,FCN[bv, Prog[]]]];
          END;

GetTok: PROCEDURE = BEGIN
wk: STRING ← [100];
i: CARDINAL;
sym: Symbol;
got: BOOLEAN ← FALSE;
c: CHARACTER;
loop: BOOLEAN ← TRUE;
uc: BOOLEAN;
```

```
TokTok ← PeekTok;
TokVal ← PeekVal;
WHILE loop DO
        loop ← FALSE;
        WHILE nchar = ' OR nchar = IODefs.TAB OR nchar = IODefs.CR DO
                [] ← GetNChar[]
                ENDLOOP;
        SELECT nchar FROM
        0C => PeekTok ← EOF;
        '" => BEGIN st: Node ← NIL;
                rr: Register = P.MRS[];
                PeekTok ← STR;
                PeekVal ← MTSt;
                P.R2[@PeekVal, @st];
                i ← 0;
                DO
                        IF i = wk.maxlength THEN
                                BEGIN
                                wk.length ← i;
                                st ← P.MakeSTR[wk];
                                PeekVal ← P.StringConcat[PeekVal, st];
                                i ← 0;
                                END;
                        wk[i] ← GetNChar[];
                        IF wk[i] = 0C THEN
                                P.SErr["String ran off end, probably omitted quote"L];
                        IF wk[i] = '" THEN
                                BEGIN
                                wk.length ← i;
                                st ← P.MakeSTR[wk];
                                PeekVal ← P.StringConcat[PeekVal, st];
                                EXIT;
                                END;
                        IF wk[i] = '↑ THEN
                                wk[i] ← Usual[GetNChar[]];
                        i ← i + 1;
                        ENDLOOP;
                P.RRS[rr];
                END;
        IN ['0..'9] => BEGIN
                wk[0] ← nchar;
                i ← 1;
                WHILE i < wk.maxlength DO
                        wk[i] ← GetNChar[];
                        IF wk[i] = 0C THEN EXIT;
                        IF wk[i] ~IN ['0..'9] THEN EXIT;
                        i ← i + 1;
                        ENDLOOP;
                IF i >= wk.maxlength THEN P.SErr["Number too long for parser"L];
                wk.length ← i;
                PeekTok ← STR;
                PeekVal ← P.MakeSTR[wk];
                got ← TRUE;
                END;
        '' => BEGIN                     -- single quote, just like " except terminated diff.
                i ← 0;
                WHILE i < wk.maxlength AND (GetNChar[] IN ['A..'Z] OR nchar IN ['a..'z] OR nchar IN ['0..'9] OR nchar = '. OR
                nchar = '↑) DO
                        wk[i] ← IF nchar = '↑ THEN Usual[GetNChar[]] ELSE nchar;
                        i ← i + 1;
                        ENDLOOP;
                IF i >= wk.maxlength THEN P.SErr["String too long for parser"L];
                wk.length ← i;
                PeekTok ← STR;
                PeekVal ← P.MakeSTR[wk];
                got ← TRUE;
                END;
        '( => PeekTok ← LP;
        ') => PeekTok ← RP;
        '[ => PeekTok ← LB;
        '] => PeekTok ← RB;
        '{ => PeekTok ← LC;
```

```
'} => PeekTok ← RC;
': => PeekTok ← COLON;
'~ => PeekTok ← TILDE;
'% => PeekTok ← ITER;
'@ => PeekTok ← SCREEN;
'/ => BEGIN
        [] ← GetNChar[];
        IF nchar ~ = '/ THEN BEGIN
                PeekTok ← DIV;
                got ← TRUE;
                END
        ELSE BEGIN
                [] ← GetNChar[];
                IF nchar ~ = '/ THEN BEGIN
                        PeekTok ← MAPPLY;
                        got ← TRUE;
                        END
                ELSE PeekTok ← GOBBLE;
                END;
        END;
', => BEGIN
        [] ← GetNChar[];
        IF nchar = ', THEN PeekTok ← CATL
        ELSE IF nchar = '! THEN BEGIN
                PeekTok ← SEQOFC;
                END
        ELSE IF nchar ~ = ', THEN BEGIN
                PeekTok ← COMMA;
                got ← TRUE;
                END;
        END;
'| => PeekTok ← PALT;
'. => BEGIN
        [] ← GetNChar[];
        IF nchar ~ = '. THEN P.SErr["Unknown character '.'"L]
        ELSE BEGIN
                [] ← GetNChar[];
                IF nchar ~ = '. THEN P.SErr["Unknown character '.'"L]
                ELSE PeekTok ← HOLE;
                END;
        END;
'+ => PeekTok ← PLUS;
'* => PeekTok ← DELETE;
'> => PeekTok ← RARR;
'# => PeekTok ← WILD;
'; => BEGIN
        stcnt ← stcnt + 1;
        PeekTok ← SEP;
        END;
'← => PeekTok ← ASS;
'= => PeekTok ← EQUAL;
'- => BEGIN
        [] ← GetNChar[] ;
        IF nchar = '- THEN BEGIN
                [] ← GetNChar[];
                IF nchar ~ = '- THEN BEGIN
                        PeekTok ← SEQUENCE;
                        got ← TRUE;
                        END
                ELSE BEGIN
                        DO
                                c ← GetNChar[];
                                IF c = 0C OR c = IODefs.CR THEN EXIT;
                                IF c ~ = '- THEN LOOP;
                                c ← GetNChar[];
                                IF c = 0C OR c = IODefs.CR THEN EXIT;
                                IF c ~ = '- THEN LOOP;
                                c ← GetNChar[];
                                IF c = '- OR c = 0C OR c = IODefs.CR THEN EXIT;
                                ENDLOOP;
                        loop ← TRUE;
                        got ← TRUE;
```

```
                                END
                        END
                ELSE BEGIN
                        PeekTok ← MINUS;
                        got ← TRUE;
                        END;
                END;
        '? => PeekTok ← OPT;
        '! => PeekTok ← SEQOF;
        IN ['a..'z], IN ['A..'Z] => BEGIN
                i ← 0;
                uc ← FALSE;
                WHILE nchar IN ['a..'z] OR nchar IN ['A..'Z] OR nchar IN ['0..'9] DO
                        wk[i] ← nchar;
                        uc ← uc OR nchar IN['A..'Z];
                        i ← i + 1;
                        [] ← GetNChar[];
                        ENDLOOP;
                wk.length ← i;
                sym ← P.Lookup[wk];
                IF sym = NIL THEN BEGIN
                        IF ~uc AND i > 1 THEN P.SErr["Unknown primitive function name"L];
                        PeekTok ← ID;
                        PeekVal ← P.Insert[wk,ID,string,PLDefs.Unbound,PLDefs.Unbound];
                        END
                ELSE BEGIN
                        PeekTok ← sym.tok;
                        PeekVal ← sym;
                        END;
                got ← TRUE;
                END;
        ENDCASE => P.SErr["Unknown character"L];
        IF ~got THEN [] ← GetNChar[];
        ENDLOOP;
END;


SetCurrentNode: PUBLIC PROCEDURE[n:Node] = BEGIN
currn ← n;
END;

GetNChar: PROCEDURE RETURNS [CHARACTER] = BEGIN
IF nchar = IODefs.CR THEN savinput ← progstr;
nchar ← P.Item[@progstr];
IF nchar = IODefs.ControlZ THEN
        WHILE (nchar ← P.Item[@progstr]) ~= IODefs.CR DO ENDLOOP;
charcnt ← IF nchar = IODefs.CR THEN 1 ELSE charcnt + 1;
IF nchar = IODefs.CR THEN linecnt ← linecnt + 1;
RETURN[nchar];
END;

ErrorMsg: PUBLIC PROCEDURE[str,str1: STRING] = BEGIN
c: CHARACTER;
DispDefs.WF4["Line: %d Stmt: %d Char: %d, %s "L,linecnt,stcnt,charcnt,str];
IF str1 ~= NIL THEN DispDefs.WF1["%s"L,str1];
DispDefs.WF0["*n"L];
DispDefs.WF0["In command "L];
WHILE savinput.node ~= NIL DO
        c ← P.Item[@savinput];
        IF c = IODefs.CR THEN EXIT;
        DispDefs.WF1["%c"L,c];
        ENDLOOP;
DispDefs.WF0["*n"L];
END;

ErrorMsg1: PUBLIC PROCEDURE[str: STRING, a: UNSPECIFIED] = BEGIN
i: INTEGER ← a;
DispDefs.WF4["Line: %d Stmt: %d Char: %d, %s "L,linecnt,stcnt,charcnt,str];
DispDefs.WF1["%d"L,i];
END;

Usual: PROCEDURE[c: CHARACTER] RETURNS[CHARACTER] = BEGIN
```

```
j: CARDINAL;
SELECT c FROM
' =>            RETURN[' ];
'" =>           RETURN['"];
'↑ =>           RETURN['↑];
IN ['0..'9] =>BEGIN
            j ← (c - '0)*64;
            j ← j + (GetNChar[] - '0) * 8;
            j ← j + (GetNChar[] - '0);
            RETURN[LOOPHOLE[j]];
            END;
ENDCASE => RETURN[nchar - 100B];
END;

ParseSetup: PUBLIC PROCEDURE = BEGIN
SetCurrentNode[Nail];
[] ← P.Insert["check"L,ZARY,proc,CheckRoutine,PLDefs.Unbound];
[] ← P.Insert["fail"L,FAIL,unk,PLDefs.Unbound,PLDefs.Unbound];
END;

[Fail,MTSt,Nail] ← P.GetSpecialNodes[];
END.
```

```
                                                    DIRECTORY
                                                    PLDefs: FROM "PLDefs",
                                                    VMDefs: FROM "VMDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    ImageDefs: FROM "ImageDefs",
                                                    IODefs: FROM "IODefs",
                                                    StringDefs: FROM "StringDefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    MiscDefs: FROM "miscdefs",
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    StreamDefs: FROM "StreamDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    TimeDefs: FROM "TimeDefs",
                                                    DisplayDefs: FROM "DisplayDefs",
                                                    EtherReportDefs: FROM "EtherReportDefs",
                                                    RouteDefs: FROM "RouteDefs",
                                                    OsStaticDefs: FROM "OsStaticDefs",
                                                    BTDefs: FROM "BTDefs",
                                                    SegmentDefs: FROM "SegmentDefs";

pl: PROGRAM IMPORTS D1:DispDefs, DL1:DisplayDefs, IODefs, StringDefs, FileSystemDefs, ImageDefs, P:PLDefs, VM: VMDefs,
TimeDefs, SystemDefs, StreamDefs, SegmentDefs, EtherReportDefs, RouteDefs, BTDefs, InlineDefs EXPORTS PLDefs
SHARES StringDefs = BEGIN
--
TokType: TYPE = PLDefs.TokType;
NodeType: TYPE = PLDefs.NodeType;
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
Register: TYPE = PLDefs.Register;
cdebug: BOOLEAN = PLDefs.cdebug;
--
tt: Node;
debug: BOOLEAN ← cdebug;
edfile: STRING ← [30];
IMax: CARDINAL = 2;
IStack: ARRAY[1..IMax] OF Node;
numnodes: CARDINAL;
Abort: BOOLEAN ← FALSE;
MTSt,Fail,Nail: Node;

-- for AmbushKeyStream
comStream, keyStream: StreamDefs.StreamHandle;
savedKeyStreamGet: PROCEDURE [StreamDefs.StreamHandle] RETURNS[UNSPECIFIED];
charWaiting: BOOLEAN;
firstChar: CHARACTER;

AmbushKeyStream: PROCEDURE =
    BEGIN
      -- Points to where the system stopped reading the command line.
      comfh: SegmentDefs.FileHandle = SegmentDefs.NewFile["com.cm"L, SegmentDefs.Read, SegmentDefs.OldFileOnly];
      ch: CHARACTER;
      comStream ← StreamDefs.CreateByteStream[comfh, SegmentDefs.Read];
        BEGIN
          --skip over first word
          FOR ch ← 'x , comStream.get[comStream] UNTIL ch = '
            DO
            IF comStream.endof[comStream] THEN GOTO noMore;
            ENDLOOP;
        FOR ch ← ' , comStream.get[comStream] WHILE ch = '      -- Skip leading blanks, if any.
            DO
            IF comStream.endof[comStream] THEN GOTO noMore;
            ENDLOOP;
        charWaiting ← TRUE;
          firstChar ← ch;
          EXITS
            noMore => BEGIN comStream.destroy[comStream]; RETURN END;
        END;

      keyStream ← StreamDefs.GetCurrentKey[];
      savedKeyStreamGet ← keyStream.get;
      keyStream.get ← AmbushedGet;
```

```
        END;

AmbushedGet: PROCEDURE[StreamDefs.StreamHandle]
        RETURNS[c: UNSPECIFIED] =
        BEGIN
        IF charWaiting THEN
                BEGIN
                charWaiting ← FALSE;
                c ← firstChar;
                END
        ELSE c ← comStream.get[comStream];
        IF comStream.endof[comStream] THEN
                BEGIN
                comStream.destroy[comStream];
                keyStream.get ← savedKeyStreamGet;
                END;
        END;

DayTimeRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[Node] =
BEGIN OPEN TimeDefs;
str: STRING ← [40];
AppendDayTime[str,UnpackDT[CurrentDayTime[]]];
RETURN[P.MakeSTR[str]];
END;

SetDisplaySizeRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[Node] =
BEGIN
n: CARDINAL = VM.Cardinal[P.MakeInteger[n1]];
DL1.SetSystemDisplaySize[n,n];
RETURN[Nail];
END;

DefaultName: PROCEDURE[st: STRING] = BEGIN
-- add .pl if no period is in filename
i: CARDINAL;
FOR i IN [0..st.length) DO
        IF st[i] = '. THEN RETURN;
        ENDLOOP;
StringDefs.AppendString[st,".pl"L];
END;

Interactive: PROCEDURE[str: STRING] = BEGIN
prog,tt: Node ← NIL;
wk: STRING ← [PLDefs.sSize];
i: CARDINAL;
Setup[];
FOR i IN [1.. IMax] DO
        IStack[i] ← Nail;
        ENDLOOP;
DO
        ENABLE BEGIN
                UNWIND => BEGIN D1.WF0["XXX*n"L]; RETRY END;
                IODefs.Rubout => BEGIN D1.WF0["XXX*n"L]; RETRY END;
                P.EndDisplay => RETRY;
                P.Interrupt => BEGIN
                        D1.WF0["**** Interrupt *****n"L];
                        RETRY;
                        END;
                FileSystemDefs.FileDoesNotExist => BEGIN
                        D1.WF0["File Does Not Exist - Try Again*n"L];
                        RETRY;
                        END;
                P.PBug => BEGIN
                        P.ErrorMsg["Poplar Bug!!"L,est];
                        RETRY;
                        END;
                P.SErr => BEGIN
                        P.ErrorMsg["Syntax Error - "L,est];
                        RETRY;
                        END;
```

```
        P.RErr => BEGIN
                  D1.WF1["Run-Time Error - %s *n"L,est];
                  RETRY;
                  END;
        ANY => BEGIN
                  D1.WF0["Mysterious Poplar Bug!!!*n"];
                  RETRY;
                  END
        END;
[] ← SystemDefs.PruneHeap[];
Reset[];
P.R2[@tt,@prog];
FOR i IN [1.. IMax] DO
        P.R[@IStack[i]];
        ENDLOOP;
IF str = NIL THEN BEGIN
        IF prog ~= NIL AND prog.Type = STR AND P.Length[prog] < wk.maxlength THEN P.MakeString[wk,prog];
        [] ← IODefs.ReadEditedString[wk,REString,TRUE];
        D1.WF0["*n"L];
        END
ELSE BEGIN
        StringDefs.AppendString[wk,str];
        str ← NIL;
        END;
IF StringDefs.EquivalentString[wk,"un"L] THEN BEGIN
        Pop![];
        PrintTop;
        LOOP
        END
ELSE IF StringDefs.EquivalentString[wk,"reset"L] THEN EXIT
ELSE IF StringDefs.EquivalentString[wk,"?"L] THEN BEGIN
        D1.WF0["Any Poplar stmt, reset for reset, un for backup, ESC for last command,*n"L];
        D1.WF0["percent for the debugging interface, quit to return to the O.S.*n"L];
        D1.WF0["$file to run file*n"L];
        LOOP;
        END
ELSE IF StringDefs.EquivalentString[wk,"%"L] THEN BEGIN
        prog ← NIL;
        Main[];
        EXIT;
        END
ELSE IF StringDefs.EquivalentString[wk,"quit"L] THEN BEGIN
        Abort ← TRUE;
        EXIT;
        END
ELSE IF StringDefs.EquivalentString[wk,"more"L] THEN BEGIN
        [] ← D1.ToggleMore[];
        PrintTop;
        [] ← D1.ToggleMore[];
        LOOP;
        END
ELSE IF wk[0] = '$ THEN BEGIN
        FOR i IN [0.. wk.length - 1) DO
                wk[i] ← wk[i + 1];
                ENDLOOP;
        wk.length ← wk.length - 1;
        DefaultName[wk];
        prog ← P.MakeSTR[wk];
        prog ← P.FileRoutine[NIL,prog,NIL];
        IF prog.Type = FAIL THEN LOOP;
        Reset[];
        P.R2[@tt,@prog];
        FOR i IN [1.. IMax] DO
                P.R[@IStack[i]];
                ENDLOOP;
        END
ELSE IF wk.length = 0 THEN LOOP
ELSE prog ← P.MakeSTR[wk];
IF IStack[1].Type = FAIL THEN IStack[1] ← MTSt;                    -- kludge
P.SetCurrentNode[IStack[1]];
tt ← P.Dist[prog];
IF tt.Type = ASS THEN []←P.Eval[tt,NIL]
```

```
        ELSE BEGIN
                Pushl[P.Eval[tt,NIL]];
                PrintTop;
                END;
        ENDLOOP;
Cleanup[];
END;

Main: PROCEDURE = BEGIN
wk: STRING ← [PLDefs.sSize];
prog: Node ← NIL;
c: CHARACTER;
DO
        IODefs.WriteChar['%];
        c ← IODefs.ReadChar[];
        SELECT c FROM
        't,'T => BEGIN
                D1.WF0["Toggle debug, now "L];
                debug ← ~debug;
                IF debug THEN D1.WF0["TRUE*n"L] ELSE D1.WF0["FALSE*n"L];
                END;
        'q,'Q => BEGIN
                D1.WF0["Quit [Confirm] "L];
                IF D1.Confirm[] THEN BEGIN
                        D1.WF0["*n"L];
                        RETURN;
                        END;
                D1.WF0["*n"L];
                END;
        'a,'A => BEGIN
                D1.WF0["Abort [Confirm] "L];
                IF D1.Confirm[] THEN ImageDefs.AbortMesa[];
                END;
        ' ,'p,'P => BEGIN
                IF c ~= ' THEN D1.WF0["Program:"L];
                D1.WF0[" "L];
                IF prog ~= NIL THEN P.MakeString[wk,prog];
                IODefs.ReadLine[wk];
                Setup[];
                prog ← P.MakeSTR[wk];
                D1.WF0["Start:*n"L];
                IF P.Length[prog] ~= 0 THEN tt ← P.Dist[prog];
                P.R[@tt];
                P.ParseTree[tt];
                END;
        ENDCASE => D1.WF0["*n"L];
        ENDLOOP;
END;

PrintTop: PROCEDURE =
        BEGIN
        D1.ClearScreen[];
        [] ← D1.ToggleMore[];
        IF IStack[1].Type = STR THEN
                BEGIN
                D1.PrintString[IStack[1]];
                D1.WF0["*n"L];
                END
        ELSE D1.Print[IStack[1]];
        [] ← D1.ToggleMore[];
        END;

IsDebug: PUBLIC PROCEDURE RETURNS[BOOLEAN] = BEGIN
RETURN[debug];
END;

Pushl: PROCEDURE[n: Node] = BEGIN
i: CARDINAL;
FOR i DECREASING IN [1..IMax) DO
        IStack[i + 1] ← IStack[i];
        ENDLOOP;
IStack[1] ← n;                  -- this is the top of stack
```

```
END;

Popl: PROCEDURE = BEGIN
i: CARDINAL;
FOR i IN [2..IMax] DO
        IStack[i-1] ← IStack[i];
        ENDLOOP;
IStack[IMax] ← Nail;
END;

REString: PROCEDURE[c: CHARACTER] RETURNS[a: BOOLEAN] = BEGIN
a ← c = IODefs.CR;
END;

CountDepth: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
numnodes ← numnodes + 1;
RETURN[TRUE];
END;

StartUp: PROCEDURE = BEGIN
DL1.SetSystemDisplaySize[0,0];
DL1.SetSystemDisplaySize[35,35];
IF InlineDefs.BITAND[LOOPHOLE[177035B, POINTER]↑,1] = 0 THEN -- Spare1 is down
        BEGIN
        ImageDefs.MakeImage["Poplar.Image"L];
        [] ←EtherReportDefs.Report[[3,200B,[0,32B]],
                                                -- my socket on Maxc
                                LOOPHOLE[OsStaticDefs.OsStatics.UserName],
                                (OsStaticDefs.OsStatics.UserName.length + 2)/2];
        AmbushKeyStream[];
        END;
D1.WF0["Poplar Version 0.2 Welcomes You*n"L];
END;

Setup: PROCEDURE = BEGIN
P.StoreSetup[];                                 -- inserts don't work till this is done
VM.VMSetup[];
P.ParseSetup[];
P.SupSetup[];
P.StringSetup[];
RouteDefs.Route1Setup[];
RouteDefs.Route2Setup[];
P.StatSetup[];
BTDefs.Setup;
D1.DispSetup[];
[] ← P.Insert["daytime"L,ZARY,proc,DayTimeRoutine,PLDefs.Unbound];
[] ← P.Insert["setdisplaysize"L,ZARY,proc,SetDisplaySizeRoutine,PLDefs.Unbound];
LOOPHOLE[424B,POINTER]↑ ← 500;-- mouse X coord.
LOOPHOLE[425B,POINTER]↑ ← 650;-- mouse Y coord.
END;

Cleanup: PROCEDURE = BEGIN
P.StringCleanup[];
P.StoreCleanup[];
VM.VMCleanup[];
END;

Reset: PROCEDURE = BEGIN
P.StoreReset[];
P.SupReset[];
D1.DispReset[];
END;

-- main program
StartUp[];
[Fail,MTSt, Nail] ← P.GetSpecialNodes[];
WHILE ~Abort DO Interactive[NIL] ENDLOOP;
ImageDefs.StopMesa[];
END.
```

```
-- route1.mesa last edited by Morris, October 30, 1978
-- modified by Wadler for lazy eval, 7/13/79

                                                DIRECTORY
                                                PLDefs: FROM "PLDefs",
                                                DispDefs: FROM "DispDefs",
                                                ImageDefs: FROM "ImageDefs",
                                                IODefs: FROM "IODefs",
                                                InlineDefs: FROM "InlineDefs",
                                                MiscDefs: FROM "miscdefs",
                                                AltoFileDefs: FROM "AltoFileDefs",
                                                DirectoryDefs: FROM "DirectoryDefs",
                                                FileSystemDefs: FROM "FileSystemDefs",
                                                StreamDefs: FROM "StreamDefs",
                                                SystemDefs: FROM "SystemDefs",
                                                StringDefs: FROM "StringDefs",
                                                RouteDefs: FROM "RouteDefs",
                                                VMDefs: FROM "VMDefs",
                                                SegmentDefs: FROM "SegmentDefs";

route1: PROGRAM IMPORTS D2: DispDefs, P:PLDefs, FileSystemDefs, StringDefs, DirectoryDefs,ImageDefs, VM: VMDefs EXPORTS
RouteDefs = BEGIN
--
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
Stream: TYPE = PLDefs.Stream;
Register: TYPE = PLDefs.Register;
--
Fail,MTSt,Nail: Node;

Route1Setup: PUBLIC PROCEDURE = BEGIN
[] ← P.Insert["chop"L,ZARY,proc,ChopRoutine,PLDefs.Unbound];
[] ← P.Insert["conc"L,ZARY,proc,ConcRoutine,PLDefs.Unbound];
[] ← P.Insert["delete"L,ZARY,proc,DeleteRoutine,PLDefs.Unbound];
[] ← P.Insert["differ"L,ZARY,proc,DifferRoutine,PLDefs.Unbound];
[] ← P.Insert["dir"L,ZARY,proc,DirRoutine,PLDefs.Unbound];
[] ← P.Insert["display"L,ZARY,proc,DisplayRoutine,PLDefs.Unbound];
[] ← P.Insert["divide"L,ZARY,proc,DivideRoutine,PLDefs.Unbound];
[] ← P.Insert["edit"L,ZARY,proc,EditRoutine,PLDefs.Unbound];
[] ← P.Insert["exec"L,ZARY,proc,ExecRoutine,PLDefs.Unbound];
[] ← P.Insert["ident"L,ZARY,proc,IdentRoutine,PLDefs.Unbound];
[] ← P.Insert["islist"L,ZARY,proc,IsListRoutine,PLDefs.Unbound];
[] ← P.Insert["isnull"L,ZARY,proc,IsNullRoutine,PLDefs.Unbound];
[] ← P.Insert["isstring"L,ZARY,proc,IsStringRoutine,PLDefs.Unbound];
[] ← P.Insert["length"L,ZARY,proc,LengthRoutine,PLDefs.Unbound];
[] ← P.Insert["lines"L,ZARY,proc,LinesRoutine,PLDefs.Unbound];
[] ← P.Insert["quit"L,ZARY,proc,QuitRoutine,PLDefs.Unbound];
[] ← P.Insert["stop"L,ZARY,proc,StopRoutine,PLDefs.Unbound];
[] ← P.Insert["append"L,UNARY,proc,AppendRoutine,PLDefs.Unbound];
END;

ChopRoutine: PROCEDURE[s:Symbol,input,n2: Node] RETURNS[ans:Node] = BEGIN --/z
-- zary
rr: Register ← P.MRS[];
ns: PLDefs.StreamRecord;
str: STRING ← [2];
r1, r2, r3, r4: Node ← NIL;
P.R2[@r1, @r2]; P.R2[@r3, @r4];
ans ← NIL;
ns ← P.NewStream[input];
P.R2[@ns.node,@ns];
IF input.Type ~ = STR THEN P.RErr["Input to chop must be a string"L];
str.length ← 1;
IF ns.node = NIL THEN
        ans ← Nail
ELSE
        BEGIN
        str[0] ← P.Item[@ns];
        ans ← P.Alloc[[LIST,,LIST[
                        r1 ← P.MakeSTR[str],
                        r2 ← P.Alloc[[APPLY,,APPLY[
                                        r3 ← P.ConvertStream[@ns],
```

```
                                        r4 ← P.Alloc[[ZARY,,ZARY[s]]] -- chop --]]]]]];
        END;
P.RRS[rr];
END;

ConcRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[ans:Node] = BEGIN
rr: Register ← P.MRS[];
-- zary
-- take a list and squish all its elements to return a single string.
-- No punctuation or delimeters are added
IF ~n1.Type = LIST THEN P.RErr["conc expects a list."L];
ans ← MTSt;
P.R[@ans];
WHILE n1.listhead # NIL DO
        IF P.ZEval[n1.listhead].Type ~ = STR THEN P.RErr["conc expects a list of strings"L];
        ans ← P.StringConcat[ans,n1.listhead]; --/z
        n1 ← P.ZEval[n1.listtail]; --/z
        ENDLOOP;
P.RRS[rr];
END;

DeleteRoutine: PROCEDURE[sym:Symbol,name,n2: Node] RETURNS[Node] = BEGIN
-- zary
fs: FileSystemDefs.FileSystem;
fname: STRING ← [40];
IF name.Type ~ = STR THEN P.RErr["Delete expects a simple string as filename"L];
P.MakeString[fname,name];
fs ← FileSystemDefs.Login['a,NIL,NIL,NIL];
FileSystemDefs.Delete[fs,fname];
FileSystemDefs.Logout[fs];
RETURN[Nail];
END;

DifferRoutine: PROCEDURE[s:Symbol,node,other: Node] RETURNS[ans:Node] = BEGIN
-- unary
n1,n2: Node;
s1,s2: PLDefs.StreamRecord;
rr: Register ← P.MRS[];
IF node.Type ~ = LIST OR P.LengthList[node] ~ = 2 THEN P.RErr["differ expects a list of length 2"L];
ans ← NIL;
P.R[@ans];
n1 ← P.ZEval[node.listhead]; --/z
n2 ← P.ZEval[node.listtail.listhead]; --/z
IF n1.Type ~ = STR OR n2.Type ~ = STR THEN P.RErr["Differ expects both list elements to be strings"L];
s1 ← P.NewStream[n1];
s2 ← P.NewStream[n2];
P.R2[@s1.node,@s2.node];
WHILE s1.node ~ = NIL AND s2.node ~ = NIL DO
        IF P.Item[@s1] ~ = P.Item[@s2] THEN EXIT;
        ENDLOOP;
ans ← P.Alloc[[LIST,,LIST[P.ConvertStream[@s2],Nail]]];
ans ← P.Alloc[[LIST,,LIST[P.ConvertStream[@s1],ans]]];
P.RRS[rr];
END;

DirRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
rr: Register ← P.MRS[];
pans: POINTER TO Node;

        AddName: PROCEDURE[p:POINTER,s: STRING] RETURNS[BOOLEAN] = BEGIN
        s.length ← s.length - 1;
        pans↑ ← P.Alloc[[LIST,,LIST[P.MakeSTR[s],Nail]]];
        pans ← @pans.listtail;
        RETURN[FALSE];
        END;

ans ← Nail;
P.R[@ans];
pans ← @ans;
```

```
DirectoryDefs.EnumerateDirectory[AddName];
ans.Des.e ← TRUE;
P.RRS[rr];
END;

DisplayRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
-- zary
D2.Print[n1];
RETURN[n1];
END;

DivideRoutine: PROCEDURE[sym:Symbol,n,n2: Node] RETURNS[res:Node] = BEGIN
rr: Register ← P.MRS[];
i,j,a,b: LONG INTEGER;
-- zary
IF n.Type ~ = LIST OR P.LengthList[n] ~ = 2 THEN P.RErr["Divide expects a list of length 2"L];
 --/z this expands out the list
res ← NIL;
P.R[@res];
a ← P.MakeInteger[P.ZEval[n.listhead]];
b ← P.MakeInteger[P.ZEval[n.listtail.listhead]];
i ← a/b;
j ← a - (i*b);
res ← P.Alloc[[LIST,,LIST[P.MakeNUM[j],Nail]]];
res ← P.Alloc[[LIST,,LIST[P.MakeNUM[i],res]]];
P.RRS[rr];
END;

EditRoutine: PROCEDURE[sym:Symbol,str,n2: Node] RETURNS[Node] = BEGIN
-- zary
s: STRING ← [PLDefs.sSize];
IF str.Type ~ = STR THEN P.RErr["Input to Edit must be a string"L];
P.MakeString[s,str];
Editor[s];
-- no return
RETURN[Nail];
END;

Editor: PROCEDURE[edfile: STRING] = BEGIN
str: STRING ← [600];
t: Node ← NIL;
rr: Register ← P.MRS[];
P.R[@t];
D2.WriteToString[str];
D2.WF2["Bravo/n %s;poplar $%s'*n"L,edfile,edfile];
[] ← D2.SetWriteProcedure[D2.MyWriteProcedure];
t ← P.MakeSTR[str];
WriteRem[t];
ImageDefs.StopMesa[];
END;

ExecRoutine: PROCEDURE[sym:Symbol,str,n2: Node] RETURNS[Node] = BEGIN
s: STRING ← [PLDefs.sSize];
n: Node ← NIL;
rr: Register ← P.MRS[];
-- zary
IF str.Type ~ = STR THEN P.RErr["Exec routine takes a string as argument"L];
StringDefs.AppendString[s,";RunMesa PoplarCheckPoint.Image$"L];
StringDefs.AppendChar[s,IODefs.CR];
n ← P.MakeSTR[s];
P.R[@n];
n ← P.StringConcat[str,n];
WriteRem[str];
IF ~ImageDefs.MakeCheckPoint["PoplarCheckPoint.Image$"L] THEN
        ImageDefs.StopMesa[];
P.RRS[rr];
RETURN[Nail];
END;

IdentRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
-- zary
```

```
RETURN[n1];
END;

IsListRoutine: PROCEDURE[sym:Symbol,n,n2: Node] RETURNS[Node] = BEGIN
-- zary
RETURN[IF n.Type = LIST THEN n ELSE Fail];
END;

IsNullRoutine: PROCEDURE[sym:Symbol,list,n2: Node] RETURNS[Node] = BEGIN
-- zary
RETURN[IF list.Type = LIST AND list.listhead = NIL THEN list ELSE Fail];
END;

IsStringRoutine: PROCEDURE[sym:Symbol,n,n2: Node] RETURNS[Node] = BEGIN
-- zary
RETURN[IF n ~ = NIL AND n.Type = STR THEN n ELSE Fail];
END;

LengthRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
IF n1.Type = STR THEN BEGIN
        RETURN[P.MakeNUM[VM.Cardinal[P.Length[n1]]]];
        END;
IF n1.Type = LIST THEN RETURN[P.MakeNUM[P.LengthList[n1]]];
P.RErr["Can only take length of strings and lists"L];
END;

LinesRoutine: PROCEDURE[sym:Symbol,input,n2: Node] RETURNS[ans:Node] = BEGIN --/z
-- zary
rr: Register ← P.MRS[];
r1, r2, r3, r4: Node ← NIL;
s,a: PLDefs.StreamRecord;
i: CARDINAL;
IF input.Type ~ = STR THEN P.RErr["Lines expects a string as input"L];
P.R2[@r1,@r2]; P.R2[@r3, @r4];
s ← P.NewStream[input];
ans ← Nail;
P.R2[@ans,@s.node];
i ← 0;
a ← s;
DO
        IF s.node = NIL THEN GOTO StringEnd;
        i ← i + 1;
        IF P.Item[@s] = IODefs.CR THEN GOTO CRFound;
        REPEAT
                StringEnd =>
                        ans ← P.Alloc[[LIST,,LIST[P.SubStringStream[@a,0,i],Nail]]];
                CRFound =>
                        BEGIN
                        ans ← P.Alloc[[LIST,,LIST[
                                        r1 ← P.SubStringStream[@a,0,i],
                                        r2 ← P.Alloc[[APPLY,,APPLY[
                                                        r3 ← P.ConvertStream[@s],
                                                        r4 ← P.Alloc[[ZARY,,ZARY[sym]]] -- lines --]]]]]];
                        END;
        ENDLOOP;
ans.Des.e ← TRUE;
P.RRS[rr];
END;


QuitRoutine: PROCEDURE[sym:Symbol,str,n2: Node] RETURNS[Node] = BEGIN
-- zary
s: STRING ← [PLDefs.sSize];
n: Node;
rr: Register ← P.MRS[];
IF str.Type ~ = STR THEN P.RErr["Input to Quit must be a string"L];
StringDefs.AppendChar[s,IODefs.CR];
n ← P.MakeSTR[s];
```

```
P.R[@n];
n ← P.StringConcat[str,n];
WriteRem[n];
ImageDefs.StopMesa[];
RETURN[NIL]; -- pace compiler
END;

StopRoutine: PROCEDURE[sym:Symbol,str,n2: Node] RETURNS[Node] = BEGIN
ImageDefs.StopMesa[];
RETURN[NIL];-- pace compiler
END;

WriteRem: PROCEDURE[str: Node] = BEGIN
rr: Register ← P.MRS[];
n: Node;
n ← P.MakeSTR["rem.cm"L];
P.R[@n];
[] ← AppendRoutine[NIL,str,n];
P.RRS[rr];
END;

AppendRoutine: PROCEDURE[s:Symbol,input,name: Node] RETURNS[Node] = BEGIN
-- unary
fname: STRING ← [100];
ns: PLDefs.StreamRecord;
rr: Register;
end: LONG INTEGER;
fh: VMDefs.FileIndex;
IF input.Type ~ = STR THEN P.RErr["Input to append routine must be string"L];
IF name.Type ~ = STR THEN P.RErr["Filename for append routine must be string"L];
IF P.Length[name] > = 100 THEN P.RErr["File name too long (>100)"L];
P.MakeString[fname,name];
[fh, end] ← VM.OpenRW[fname];
ns ← P.NewStream[input];
rr ← P.MRS[];
P.R[@ns.node];
WHILE ns.node ~ = NIL DO
        VM.SetSChar[end, fh, P.Item[@ns]];
        end←end + 1
        ENDLOOP;
VM.CloseRW[fh, end];
P.RRS[rr];
RETURN[Nail];
END;


[Fail,MTSt,Nail] ← P.GetSpecialNodes[];
END.
```

```
-- route2.mesa last edited by Morris, October 30, 1978
-- modified by Wadler for Lazy Eval, 7/13/79
                                                    DIRECTORY
                                                    PLDefs: FROM "PLDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    ImageDefs: FROM "ImageDefs",
                                                    IODefs: FROM "IODefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    MiscDefs: FROM "miscdefs",
                                                    AltoFileDefs: FROM "AltoFileDefs",
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    FilePageUseDefs: FROM "FilePageUseDefs",
                                                    StreamDefs: FROM "StreamDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    RouteDefs: FROM "RouteDefs",
                                                    VMDefs: FROM "VMDefs",
                                                    SegmentDefs: FROM "SegmentDefs";

route2: PROGRAM IMPORTS D2: DispDefs, IODefs,  P:PLDefs, FileSystemDefs, FilePageUseDefs, VM: VMDefs, InlineDefs EXPORTS
RouteDefs, PLDefs  = BEGIN
--
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
Stream: TYPE = PLDefs.Stream;
Register: TYPE = PLDefs.Register;
--
Fail,MTSt,Nail: Node;

Route2Setup: PUBLIC PROCEDURE = BEGIN
[] ← P.Insert["key"L,ZARY,proc,KeyRoutine,PLDefs.Unbound];
[] ← P.Insert["listin"L,ZARY,proc,ListInRoutine,PLDefs.Unbound];
[] ← P.Insert["listout"L,UNARY,proc,ListOutRoutine,PLDefs.Unbound];
[] ← P.Insert["marry"L,ZARY,proc,MarryRoutine,PLDefs.Unbound];
[] ← P.Insert["max"L,ZARY,proc,MaxRoutine,PLDefs.Unbound];
[] ← P.Insert["min"L,ZARY,proc,MinRoutine,PLDefs.Unbound];
[] ← P.Insert["minus"L,ZARY,proc,MinusRoutine,PLDefs.Unbound];
[] ← P.Insert["plus"L,ZARY,proc,PlusRoutine,PLDefs.Unbound];
[] ← P.Insert["reverse"L,ZARY,proc,ReverseRoutine,PLDefs.Unbound];
[] ← P.Insert["run"L,ZARY,proc,RunRoutine,PLDefs.Unbound];
[] ← P.Insert["subst"L,ZARY,proc,SubstRoutine,PLDefs.Unbound];
[] ← P.Insert["times"L,ZARY,proc,TimesRoutine,PLDefs.Unbound];
[] ← P.Insert["tolower"L,ZARY,proc,ToLowerRoutine,PLDefs.Unbound];
[] ← P.Insert["toupper"L,ZARY,proc,ToUpperRoutine,PLDefs.Unbound];
[] ← P.Insert["write"L,UNARY,proc,WriteRoutine,PLDefs.Unbound];
[] ← P.Insert["zip"L,ZARY,proc,ZipRoutine,PLDefs.Unbound];
END;

KeyRoutine: PUBLIC PROCEDURE[sym:Symbol,n1,n2: Node]
        RETURNS[Node] = BEGIN

        REString: PROCEDURE[c: CHARACTER] RETURNS[a: BOOLEAN] = BEGIN
        Bad ← c = IODefs.DEL;
        a ← Bad OR c = IODefs.CR;
        END;

-- zary
-- the output is a pre-quoted string
-- this has the same effect as if the characters typed were in a file
s: STRING ← [PLDefs.sSize];
ns: PLDefs.StreamRecord;
Bad: BOOLEAN ← FALSE;
-- should use UsualEscape
IF n1.Type ~= STR THEN P.RErr["key requires a string as input"L];
ns ← P.NewStream[n1];
WHILE ns.node # NIL DO IODefs.WriteChar[P.Item[@ns]] ENDLOOP;
[] ← IODefs.ReadEditedString[s,REString,FALSE];
IODefs.WriteChar[IODefs.CR];
RETURN[IF Bad THEN Fail ELSE P.MakeSTR[s]];
END;

ListInRoutine: PROCEDURE[sym:Symbol,name,n2: Node] RETURNS[ans:Node] = BEGIN
```

```
-- zary
rr: Register ← P.MRS[];
IF name.Type ~ = STR THEN P.RErr["Listin expects a string for file name"L];
ans ← NIL;
P.R[@ans];
ans ← P.FileRoutine[sym,name,n2];
ans ← P.ZEval[ans]; --/z
IF ans.Type = STR THEN ans ← P.Dist[ans];
P.RRS[rr];
END;

ListOutRoutine: PROCEDURE[sym:Symbol,list,name: Node] RETURNS[ans:Node] = BEGIN
-- unary

        ListWriteProc: PROCEDURE[c: CHARACTER] = BEGIN
        IF i >= pgsize THEN BEGIN
                FilePageUseDefs.WritePage[fh,pno,buf + 2];
                i ← 0;
                pno ← pno + 1;
                END;
        buf[i] ← c;
        i ← i + 1;
        END;

fname: STRING ← [40];
fs: FileSystemDefs.FileSystem;
fh: FilePageUseDefs.FileHandle;
i,pgsize,pno,lp,bp: CARDINAL;
buf: STRING ← [600];
OP: PROCEDURE[CHARACTER];
rr: Register ← P.MRS[];
IF name.Type ~ = STR THEN P.RErr["Listout expects a string for file name"L];
P.MakeString[fname,name];
fs ← FileSystemDefs.Login['a,NIL,NIL,NIL];
fh ← FileSystemDefs.Open[fs,fname,FileSystemDefs.OpenMode[create]
        ! FileSystemDefs.FileAlreadyExists => RESUME];
[lp,bp,pgsize] ← FilePageUseDefs.Measure[fh];
OP ← D2.SetWriteProcedure[ListWriteProc];
pno ← 0;
i ← 0;
D2.Print[list];
FilePageUseDefs.WritePage[fh,pno,buf + 2];                    -- last page
[] ← D2.SetWriteProcedure[OP];
FilePageUseDefs.SetLength[fh,pno,i];
FilePageUseDefs.Close[fh];
FileSystemDefs.Logout[fs];
P.RRS[rr];
RETURN[Nail];
END;

List2: PROCEDURE[a,b: Node] RETURNS [Node] =
-- return the list with a and b as elements
BEGIN
RETURN[P.Alloc[[LIST,,LIST[a, P.Alloc[[LIST,,LIST[b, Nail]]]]]]];
END;

MarryRoutine: PROCEDURE[sym:Symbol,node,other: Node] RETURNS[ans:Node] = BEGIN --/z
-- unary
rr: Register = P.MRS[];
n1, n2: Node;
r1, r2, r3, r4: Node ← NIL;
IF node.Type ~ = LIST OR P.LengthList[node] ~ = 2 THEN
        P.RErr["marry expects a list of length 2"L];
P.R2[@r1, @r2]; P.R2[@r3, @r4];
n1 ← P.ZEval[node.listhead];
n2 ← P.ZEval[node.listtail.listhead];
IF ~(n1.Type = LIST AND n2.Type = LIST) THEN P.RErr["marry expects two lists"L];
IF n1.listhead = NIL AND n2.listhead = NIL THEN
        ans ← Nail
```

```
ELSE IF n1.listhead = NIL OR n2.listhead = NIL THEN
        P.RErr["marry expects equal length lists"L]
ELSE
        ans ← P.Alloc[[LIST,,LIST[
                        r1 ← List2[P.ZEval[n1.listhead], P.ZEval[n2.listhead]],
                        r2 ← P.Alloc[[APPLY,,APPLY[
                                        r3 ← List2[P.ZEval[n1.listtail], P.ZEval[n2.listtail]],
                                        r4 ← P.Alloc[[ZARY,,ZARY[sym]]] -- marry -- ]]]]]];
P.RRS[rr];
END;

MaxRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
-- zary
a:Node;
max,ii: LONG INTEGER;
IF n1.Type ~ = LIST THEN P.RErr["max expects a list "L];
a ← P.ZEval[n1.listhead]; --/z
max ← P.MakeInteger[a];
WHILE n1.listhead ~ = NIL DO
        IF max < (ii ← P.MakeInteger[P.ZEval[n1.listhead]]) THEN --/z
                BEGIN
                a ← n1.listhead;
                max ← ii;
                END;
        n1 ← P.ZEval[n1.listtail]; --/z
        ENDLOOP;
RETURN[a];
END;

MinRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
-- zary
a:Node;
min,ii: LONG INTEGER;
IF n1.Type ~ = LIST THEN P.RErr["min expects a list "L];
a ← P.ZEval[n1.listhead]; --/z
min ← P.MakeInteger[a];
WHILE n1.listhead ~ = NIL DO
        IF min > (ii ← P.MakeInteger[P.ZEval[n1.listhead]]) THEN --/z
                BEGIN
                a ← n1.listhead;
                min ← ii;
                END;
        n1 ← P.ZEval[n1.listtail]; --/z
        ENDLOOP;
RETURN[a];
END;

MinusRoutine: PROCEDURE[sym:Symbol,n,n2: Node] RETURNS[Node] = BEGIN
-- zary
IF n.Type ~ = LIST OR P.LengthList[n] ~ = 2 THEN P.RErr["Minus expects a list of length 2"L];
RETURN[P.MakeNUM[P.MakeInteger[P.ZEval[n.listhead]]-P.MakeInteger[P.ZEval[n.listtail.listhead]]]]; --/z
END;

PlusRoutine: PROCEDURE[sym:Symbol,n,n2: Node] RETURNS[Node] = BEGIN
-- zary
IF n.Type ~ = LIST OR P.LengthList[n] ~ = 2 THEN P.RErr["Plus expects a list of length 2"L];
RETURN[P.MakeNUM[P.MakeInteger[P.ZEval[n.listhead]] + P.MakeInteger[P.ZEval[n.listtail.listhead]]]]; --/z
END;

ReverseRoutine: PROCEDURE[sym:Symbol,n,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
rr: Register ← P.MRS[];
s: PLDefs.StreamRecord;
t: Node ← NIL;
P.R2[@ans,@t];
IF n.Type = LIST THEN
        BEGIN
        ans ← Nail;
        WHILE n.Type = LIST AND n.listhead ~ = NIL DO
```

```
                ans ← P.Alloc[[LIST,,LIST[P.ZEval[n.listhead],ans]]]; --/z
                n ← P.ZEval[n.listtail]; --/z
                ENDLOOP;
        END
ELSE IF n.Type = STR THEN BEGIN
        s ← P.NewStream[n];
        P.R[@s.node];
        ans ← MTSt;
        WHILE s.node ~ = NIL DO
                t ← Rev[@s];
                ans ← P.StringConcat[t, ans];
                ENDLOOP;
        END
ELSE P.RErr["reverse expects a list or string as input"L];
P.RRS[rr];
END;

Rev: PROCEDURE[s: Stream] RETURNS[Node] = BEGIN
wk: STRING ← [PLDefs.sSize];
i,j,k: CARDINAL;
i ← wk.maxlength;
WHILE s.node ~ = NIL AND i > 0 DO
        i ← i - 1;
        wk[i] ← P.Item[s];
        ENDLOOP;
k ← 0;
FOR j IN [i..wk.maxlength) DO
        wk[k] ← wk[j];
        k ← k + 1;
        ENDLOOP;
wk.length ← k;
RETURN[P.MakeSTR[wk]];
END;

RunRoutine: PROCEDURE[sym:Symbol,prog,n2: Node] RETURNS[tt:Node] = BEGIN
-- zary
rr: Register ← P.MRS[];
IF prog.Type ~ = STR THEN P.RErr["Input to Run must be string"L];
tt ← P.Dist[prog];
P.R[@tt];
tt ← P.Eval[tt,NIL];
P.RRS[rr];
RETURN[tt];
END;

SubstRoutine: PROCEDURE[sym:Symbol,inputnode,n2: Node] RETURNS[ans:Node] = BEGIN

        match: PROCEDURE RETURNS[BOOLEAN] = BEGIN
        pat: PLDefs.StreamRecord ← pattern;
        inp: PLDefs.StreamRecord ← input;
        WHILE inp.node ~ = NIL DO
                IF pat.node = NIL THEN RETURN[TRUE];
                IF P.Item[@pat] ~ = P.Item[@inp] THEN RETURN[FALSE];
                ENDLOOP;
        IF pat.node = NIL AND inp.node = NIL THEN RETURN[TRUE];
        RETURN[FALSE];
        END;

rr: Register ← P.MRS[];
lenpattern,k: LONG INTEGER;
input,pattern,sav: PLDefs.StreamRecord;
output,t: Node ← NIL;
IF inputnode.Type ~ = STR THEN P.RErr["Subst takes as input a string"L];
sav ← input ← P.NewStream[inputnode];
P.R3[@output,@t,@input.node];
P.R[@sav.node];
output ← KeyRoutine[NIL,P.MakeSTR["replacement string: "L],NIL];
IF output.Type = FAIL THEN P.Interrupt;
t ← KeyRoutine[NIL,P.MakeSTR["pattern string: "L],NIL];
```

```
IF t.Type = FAIL THEN P.Interrupt;
pattern ← P.NewStream[t];
ans ← MTSt;
P.R2[@pattern.node,@ans];
lenpattern ← P.Length[t];
IF lenpattern = 0 THEN P.RErr["Pattern must be non-empty"L];
k ← 0;
WHILE input.node ~ = NIL DO
        IF match[] THEN BEGIN
                t ← IF k > 0 THEN P.SubStringStream[@sav,0,k] ELSE MTSt;
                t ← P.StringConcat[t,output];
                ans ← P.StringConcat[ans,t];
                P.SkipStream[@input,lenpattern];
                sav ← input;
                k ← 0;
                END
        ELSE BEGIN
                [] ← P.Item[@input];
                k ← k + 1;
                END;
        ENDLOOP;
t ← IF sav.node ~ = NIL THEN P.SubStringStream[@sav,0,k] ELSE MTSt;
ans ← P.StringConcat[ans,t];
P.RRS[rr];
END;

TimesRoutine: PROCEDURE[sym:Symbol,n,n2: Node] RETURNS[Node] = BEGIN
-- zary
IF n.Type ~ = LIST OR P.LengthList[n] ~ = 2 THEN P.RErr["Times expects a list of length 2"L];
RETURN[P.MakeNUM[P.MakeInteger[P.ZEval[n.listhead]]*P.MakeInteger[P.ZEval[n.listtail.listhead]]]]; --/z
END;

ToCase: PROCEDURE[inputnode: Node, tolower: BOOLEAN] RETURNS[ans:Node] = BEGIN
rr: Register ← P.MRS[];
k: LONG INTEGER;
input,sav: PLDefs.StreamRecord;
t,z: Node ← NIL;
s: STRING ← [2];
c: CHARACTER;
IF inputnode.Type ~ = STR THEN P.RErr["tolower and toupper take as input a string"L];
sav ← input ← P.NewStream[inputnode];
ans ← MTSt;
P.R3[@ans,@t,@input.node];
P.R2[@sav.node,@z];
k ← 0;
s.length ← 1;
WHILE input.node ~ = NIL DO
        c ← P.Item[@input];
        IF (tolower AND c IN ['A..'Z]) OR (~tolower AND c IN ['a..'z]) THEN BEGIN
                t ← IF k > 0 THEN P.SubStringStream[@sav,0,k] ELSE MTSt;
                s[0] ← LOOPHOLE[InlineDefs.BITXOR[LOOPHOLE[c,CARDINAL],40B]];
                z ← P.MakeSTR[s];
                t ← P.StringConcat[t,z];
                ans ← P.StringConcat[ans,t];
                sav ← input;
                k ← 0;
                END
        ELSE k ← k + 1;
        ENDLOOP;
t ← IF sav.node ~ = NIL THEN P.SubStringStream[@sav,0,k] ELSE MTSt;
ans ← P.StringConcat[ans,t];
P.RRS[rr];
END;

ToLowerRoutine: PROCEDURE[sym:Symbol,inputnode,n2: Node] RETURNS[Node] = BEGIN
RETURN[ToCase[inputnode,TRUE]];
END;

ToUpperRoutine: PROCEDURE[sym:Symbol,inputnode,n2: Node] RETURNS[Node] = BEGIN
RETURN[ToCase[inputnode,FALSE]];
```

```
END;

WriteRoutine: PUBLIC PROCEDURE[s:Symbol,input,name: Node] RETURNS[Node] = BEGIN
-- unary
fname: STRING ← [100];
ns: PLDefs.StreamRecord;
rr: Register;
i: LONG INTEGER;
fh: VMDefs.FileIndex;
IF input.Type ~ = STR THEN P.RErr["Input to write routine must be string"L];
IF name.Type ~ = STR THEN P.RErr["Filename for write routine must be string"L];
IF P.Length[name] > = 100 THEN P.RErr["File name too long (>100)"L];
P.MakeString[fname, name];
[fh,] ← VM.OpenRW[fname];
i ← 0;
ns ← P.NewStream[input];
rr ← P.MRS[];
P.R[@ns.node];
WHILE ns.node ~ = NIL DO
        VM.SetSChar[i, fh, P.Item[@ns]];
        i ← i+1
        ENDLOOP;
VM.CloseRW[fh,i];
P.RRS[rr];
RETURN[Nail];
END;

ZipRoutine: PROCEDURE[sym:Symbol,node,other: Node] RETURNS[ans:Node] = BEGIN --/z
-- unary
rr: Register = P.MRS[];
r1, r2: Node ← NIL;
n1, n2: Node;
IF node.Type ~ = LIST OR P.LengthList[node] ~ = 2 THEN P.RErr["zip expects a list of length 2"L];
P.R2[@r1, @r2];
n1 ← P.ZEval[node.listhead];
n2 ← P.ZEval[node.listtail.listhead];
IF n1.Type # LIST OR n2.Type # LIST THEN P.RErr["zip expects a list of lists"L];
IF n1.listhead = NIL THEN
        ans ← P.ZEval[n2]
ELSE IF n2.listhead = NIL THEN
        ans ← P.ZEval[n1]
ELSE
        ans ← P.Alloc[[LIST,,LIST[
                        P.ZEval[n1.listhead],
                        P.Alloc[[LIST,,LIST[
                                        P.ZEval[n2.listhead],
                                        P.Alloc[[APPLY,,APPLY[
                                                        r1 ← List2[P.ZEval[n1.listtail], P.ZEval[n2.listtail]],
                                                        r2 ← P.Alloc[[ZARY,,ZARY[sym]]] -- zip --]]]]]]]]];

P.RRS[rr];
END;

[Fail,MTSt,Nail] ← P.GetSpecialNodes[];
END.
```

```
-- stat.mesa last edited by Morris, October 27, 1978  11:47 AM
                                                        DIRECTORY
                                                        DispDefs: FROM "DispDefs",
                                                        PLDefs: FROM "PLDefs",
                                                        IODefs: FROM "IODefs",
                                                        InlineDefs: FROM "InlineDefs",
                                                        MiscDefs: FROM "MiscDefs",
                                                        FileSystemDefs: FROM "FileSystemDefs",
                                                        FilePageUseDefs: FROM "FilePageUseDefs",
                                                        SystemDefs: FROM "SystemDefs",
                                                        VMDefs: FROM "VMDefs",
                                                        StringDefs: FROM "StringDefs";

stat: PROGRAM IMPORTS DispDefs, P:PLDefs, SystemDefs, VM: VMDefs EXPORTS PLDefs  = BEGIN
--
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
String: TYPE = PLDefs.String;
Stream: TYPE = PLDefs.Stream;
Register: TYPE = PLDefs.Register;
LongCARDINAL: TYPE = InlineDefs.LongCARDINAL;
cdebug: BOOLEAN = PLDefs.cdebug;
--
numnodes,numcat,numfile,numsimp,spacesimp,numlist: CARDINAL;
maxlength: LONG INTEGER;
Bal: POINTER TO ARRAY OF Node;
nleaves: CARDINAL;
bin: CARDINAL;
MinCoerceSize: CARDINAL = 300;
MTSt,Fail,Nail: Node;
dnum: CARDINAL = 150;
len: LONG INTEGER;
defaultBal: ARRAY[0..dnum) OF Node;

Empty: PUBLIC PROCEDURE[n: Node] RETURNS [BOOLEAN] =
        BEGIN
        DO
                IF n.Type # STR THEN P.PBug["Non-String"L];
                IF n.Des.s = simp THEN RETURN [n.str.length = 0];
                --n.Des.s = cat
                IF ~Empty[n.str.n1]THEN RETURN[FALSE];
                n ← n.str.n2;
                ENDLOOP;
        END;

Length: PUBLIC PROCEDURE[n: Node] RETURNS [i:LONG INTEGER] =
        BEGIN
        i ← 0;
        DO
                IF n.Type # STR THEN P.PBug["Non-String"];
                IF n.Des.s = simp THEN RETURN[i + n.str.length];
                i ← i + Length[n.str.n1];
                n ← n.str.n2;
                ENDLOOP;
        END;


-- used in SubString and ConvertStream only
Skip: PUBLIC PROCEDURE[n:Node,i: LONG INTEGER] = BEGIN
a: CARDINAL;
ii: LONG INTEGER;
s: String;
-- advance n's position by i chars
-- equivalent to THROUGH[1..i] DO [] ← Next[n] ENDLOOP, but much faster
IF n.Type ~ = STR THEN P.PBug["must be STR"L];
IF n.Des.s = simp THEN BEGIN
        a ← VM.Cardinal[i];
        s ← @n.str;
        s.length ← IF a > = s.length THEN 0 ELSE s.length - a;
        s.start ← s.start + a;
        END
```

```
ELSE IF n.Des.s = cat THEN BEGIN
        ii ← Length[n.str.n1];
        IF ii > i THEN Skip[n.str.n1,i]
        ELSE BEGIN
                -- this free is possibly the most dangerous FreeTree
                P.FreeTree[n.str.n1];
                n1 ← n.str.n2t;
                Skip[n,i-ii];
                END;
        END
ELSE P.PBug["unknown STR type"L];
END;

LinearSTR: PUBLIC PROCEDURE[n: Node] RETURNS[Node,LONG INTEGER] = BEGIN
i: CARDINAL;
rr: Register;
b,ns: Node ← NIL;
IF n = NIL OR n.Type ~ = STR THEN RETURN[n,0];
IF n.Des.s ~ = cat OR n.Des.b THEN RETURN[n,Length[n]];
IF n.str.n1.Des.s ~ = cat THEN BEGIN            -- may already be in desired form
        rr ← P.MRS[];
        b ← n;
        ns ← n.str.n2;
        len ← Length[n.str.n1];
        P.R[@b];
        WHILE ns.Des.s = cat AND ns.str.n1.Des.s ~ = cat DO
                IF cdebug THEN P.CheckNode[ns];
                b ← ns;
                len ← len + Length[n.str.n1];
                ns ← ns.str.n2;
                ENDLOOP;
        IF b.str.n2.Des.s = cat THEN [b.str.n2,] ← LinearSTR[b.str.n2];
        len ← len + Length[b.str.n2];
        P.RRS[rr];
        n.Des.b ← TRUE;
        RETURN[n,len];
        END;
rr ← P.MRS[];
P.R[@n];
nleaves ← 0;
P.Preorder[n,CountLeaves];
-- nleaves must be 1 or higher
Bal ← IF nleaves > = dnum THEN P.GetCore[nleaves*SIZE[Node]] ELSE BASE[defaultBal];
bin ← 0;
-- the nodes in Bal need not be registered as they are pointed to by n
len ← 0;
P.Preorder[n,InsertLeaves];
-- bin must be 1 or higher
n ← P.StringConcat[Bal[bin-2],Bal[bin-1]];
IF bin > 2 THEN FOR i DECREASING IN [0.. bin-3] DO
        n ← P.StringConcat[Bal[i],n];
        ENDLOOP;
IF Bal ~ = BASE[defaultBal] THEN SystemDefs.FreeSegment[Bal];
P.RRS[rr];
n.Des.b ← TRUE;
RETURN[n,len];
END;

CountLeaves: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
IF n.Type ~ = STR OR n.Des.s = cat THEN RETURN[TRUE];
-- n is a file STR or simp STR
nleaves ← nleaves + 1;
RETURN[FALSE];
END;

InsertLeaves: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
IF n.Type ~ = STR OR n.Des.s = cat THEN RETURN[TRUE];
-- n is a file STR or simp STR
len ← len + Length[n];
Bal[bin] ← n;
```

```
bin ← bin + 1;
RETURN[FALSE];
END;

Detail: PUBLIC PROCEDURE[n: Node] = BEGIN
ii: LONG INTEGER;
IF cdebug THEN BEGIN
        -- print out important things about STRS
        IF n = NIL OR (n.Type ~ = STR AND n.Type ~ = LIST) THEN RETURN;
        numnodes ← numcat ← numfile ← numsimp ← spacesimp ← numlist ← 0;
        maxlength ← 0;
        ii ← IF n.Type = STR THEN Length[n] ELSE 0;
        P.Preorder[n,strstat];
        DispDefs.WF4["Len %i,maxLen %i,nNode %u,Cat %u,"L, @ii, @maxlength, numnodes,numcat];
        DispDefs.WF4["File %u,Smp %u,spSmp %u,nest %u,"L,numfile, numsimp, spacesimp, CatDepth[n]];
        DispDefs.WF2["SF %u, nlist %u*n"L,spacesimp + 2*(numcat + numfile + numlist + numsimp), numlist];
        END;
END;

strstat: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
ii: LONG INTEGER;
IF cdebug THEN BEGIN
        IF n.Type = LIST THEN numlist ← numlist + 1
        ELSE IF n.Type = STR THEN BEGIN
                numnodes ← numnodes + 1;
                IF n.Des.s = cat THEN numcat ← numcat + 1
                ELSE BEGIN
                        ii ← Length[n];
                        maxlength ← IF ii > maxlength THEN ii ELSE maxlength;
                        IF n.Des.s = simp THEN BEGIN
                                numsimp ← numsimp + 1;
                                spacesimp ← spacesimp + n.str.length;
                                END;
                        END;
                END;
        END;
RETURN[TRUE];
END;

CatDepth: PROCEDURE[n: Node] RETURNS[CARDINAL] = BEGIN
i,j: CARDINAL;
IF cdebug THEN BEGIN
        IF n = NIL OR n.Type ~ = STR OR n.Des.s ~ = cat THEN RETURN[0];
        i ← CatDepth[n.str.n1];
        j ← CatDepth[n.str.n2];
        RETURN[IF i > j THEN i + 1 ELSE j + 1 ];
        END;
RETURN[0];--pace compiler
END;

FixRep: PUBLIC PROCEDURE[n: Node] RETURNS[Node] = BEGIN
-- returns a right-linear STR node
ii: LONG INTEGER;
rr: Register ← P.MRS[];
a: Node ← NIL;
pans: POINTER TO Node;
P.R[@n];
IF cdebug THEN P.CheckNode[n];
IF n = NIL THEN n←n
ELSE IF n.Type = LIST THEN BEGIN
        a ← n;
        WHILE a.listhead ~ = NIL DO
                a.listhead ← FixRep[a.listhead];
                a ← a.listtail;
                ENDLOOP;
        END
ELSE IF n.Type = STR AND n.Des.s ~ = simp THEN BEGIN
        [n,ii] ← LinearSTR[n];
        IF ii < MinCoerceSize THEN BEGIN
```

```
                n ← P.Coerce[n];
                END
        ELSE BEGIN
                a ← n;
                P.R[@a];
                pans ← @n;
                WHILE a.Des.s = cat DO
                        ii ← Length[a.str.n1];
                        IF ii < MinCoerceSize AND a.Des.s ~ = simp THEN a.str.n1 ← P.Coerce[a.str.n1];
                        pans ← @a.str.n2;
                        a ← a.str.n2;
                        ENDLOOP;
                ii ← Length[a];
                IF ii < MinCoerceSize AND a.Des.s ~ = simp THEN pans↑ ← P.Coerce[a];
                END;
        END;
P.RRS[rr];
RETURN[n];
END;

SubStream: PUBLIC PROCEDURE[s:Stream,i: LONG INTEGER] RETURNS [CHARACTER] = BEGIN
RETURN[IF s.node = NIL THEN 0C ELSE P.Sub[s.node,i + (s.posn-s.len + s.cp)]];
END;

LengthStream: PUBLIC PROCEDURE[s: Stream] RETURNS [LONG INTEGER] = BEGIN
RETURN[IF s.node = NIL THEN 0 ELSE Length[s.node]-(s.posn-s.len + s.cp)];
END;

ConvertStream: PUBLIC PROCEDURE[s: Stream] RETURNS [ans: Node] = BEGIN
IF s.node ~ = NIL THEN BEGIN
        ans ← P.CopyTree[s.node];
        Skip[ans,(s.posn-s.len + s.cp)];
        END
ELSE ans ← MTSt;
END;

SubStringStream: PUBLIC PROCEDURE[s: Stream,i,j: LONG INTEGER] RETURNS [Node] = BEGIN
RETURN[IF s.node = NIL THEN MTSt ELSE P.SubString[s.node,(s.posn-s.len + s.cp) + i,(s.posn-s.len + s.cp) + j]];
END;


-- Pattern Procedures

WordRoutine: PROCEDURE[sym: Symbol,n:Stream, n2: Node] RETURNS[ans: Node] = BEGIN
s: PLDefs.StreamRecord ← n↑;
c: CHARACTER;
a: LONG INTEGER ← 0;
DO
        IF n.node = NIL THEN EXIT;
        c ← SubStream[n,0];
        IF c NOT IN ['A..'Z] AND c NOT IN ['a..'z] THEN EXIT;
        [] ← P.Item[n];
        a ← a + 1;
        ENDLOOP;
ans ← IF a = 0 THEN Fail ELSE SubStringStream[@s,0,a];
END;

ThingRoutine: PROCEDURE[sym: Symbol,n:Stream, n2: Node] RETURNS[ans: Node] = BEGIN
s: PLDefs.StreamRecord ← n↑;
c: CHARACTER;
a: LONG INTEGER ← 0;
DO
        IF n.node = NIL THEN EXIT;
        c ← SubStream[n,0];
        IF c NOT IN ['A..'Z] AND c NOT IN ['a..'z] AND c NOT IN ['0..'9] THEN EXIT;
        [] ← P.Item[n];
        a ← a + 1;
        ENDLOOP;
```

```
ans ← IF a = 0 THEN Fail ELSE SubStringStream[@s,0,a];
END;

IntegerRoutine: PROCEDURE[sym: Symbol,n: Stream, n2: Node] RETURNS[ans: Node] = BEGIN
c: CHARACTER;
s: PLDefs.StreamRecord ← n↑;
sign: BOOLEAN ← FALSE;
a: LONG INTEGER ← 0;
c ← SubStream[n, 0];
IF c = '- OR c = '+ THEN BEGIN sign←TRUE; a ← a + 1; [] ← P.Item[n] END;
DO
        IF n.node = NIL THEN EXIT;
        c ← SubStream[n, 0];
        IF c ~IN ['0..'9] THEN EXIT;
        [] ← P.Item[n];
        a ← a + 1;
        ENDLOOP;
ans ← IF a = 0 OR sign AND a = 1 THEN Fail ELSE SubStringStream[@s,0,a];
END;

NumberRoutine: PROCEDURE[sym: Symbol,n: Stream, n2: Node] RETURNS[ans: Node] = BEGIN
per: BOOLEAN ← FALSE;
c: CHARACTER;
s: PLDefs.StreamRecord ← n↑;
sign: BOOLEAN ← FALSE;
a: LONG INTEGER ← 0;
c ← SubStream[n, 0];
IF c = '- OR c = '+ THEN BEGIN sign←TRUE; a ← a + 1; [] ← P.Item[n] END;
DO
        IF n.node = NIL THEN EXIT;
        c ← SubStream[n, 0];
        IF c ~IN ['0..'9] AND (c ~= '. OR (c = '. AND per))THEN EXIT;
        per ← per OR c = '.;
        [] ← P.Item[n];
        a ← a + 1;
        ENDLOOP;
ans ← IF a = 0 OR sign AND a = 1 THEN Fail ELSE SubStringStream[@s,0,a];
END;

ItemRoutine: PROCEDURE[sym: Symbol,n: Stream, n2: Node] RETURNS[ans: Node] = BEGIN
per: BOOLEAN ← FALSE;
c: CHARACTER;
s: PLDefs.StreamRecord ← n↑;
sign: BOOLEAN ← FALSE;
a: LONG INTEGER ← 0;
c ← SubStream[n, 0];
IF c = '- OR c = '+ THEN BEGIN sign←TRUE; a ← a + 1; [] ← P.Item[n] END;
DO
        IF n.node = NIL THEN EXIT;
        c ← SubStream[n, 0];
        IF c ~IN ['0..'9] AND c ~IN ['a..'z] AND c ~IN ['A..'Z] AND (c ~= '. OR (c = '. AND per))THEN EXIT;
        per ← per OR c = '.;
        [] ← P.Item[n];
        a ← a + 1;
        ENDLOOP;
ans ← IF a = 0 OR sign AND a = 1 THEN Fail ELSE SubStringStream[@s,0,a];
END;

SpaceRoutine: PROCEDURE[sym: Symbol,string:Stream, n2: Node] RETURNS[ans: Node] = BEGIN
c: CHARACTER;
c ← SubStream[string,0];
IF c = ' OR c = IODefs.TAB THEN BEGIN
        ans ← SubStringStream[string,0,1];
        [] ← P.Item[string];
        END
ELSE ans ← Fail;
END;

DigitRoutine: PROCEDURE[sym: Symbol,string:Stream, n2: Node] RETURNS[ans: Node] = BEGIN
c: CHARACTER;
```

```
c ← SubStream[string,0];
IF c IN ['0..'9] THEN BEGIN
        ans ← SubStringStream[string,0,1];
        [] ← P.Item[string];
        END
ELSE ans ← Fail;
END;

SmallLetterRoutine: PROCEDURE[sym: Symbol,string:Stream, n2: Node] RETURNS[ans: Node] = BEGIN
c: CHARACTER;
c ← SubStream[string,0];
IF c IN ['a..'z] THEN BEGIN
        ans ← SubStringStream[string,0,1];
        [] ← P.Item[string];
        END
ELSE ans ← Fail;
END;

BigLetterRoutine: PROCEDURE[sym: Symbol,string:Stream, n2: Node] RETURNS[ans: Node] = BEGIN
c: CHARACTER;
c ← SubStream[string,0];
IF c IN ['A..'Z] THEN BEGIN
        ans ← SubStringStream[string,0,1];
        [] ← P.Item[string];
        END
ELSE ans ← Fail;
END;

LetterRoutine: PROCEDURE[sym: Symbol,string:Stream, n2: Node] RETURNS[ans: Node] = BEGIN
c: CHARACTER;
c ← SubStream[string,0];
IF c IN ['A..'Z] OR c IN ['a..'z] THEN BEGIN
        ans ← SubStringStream[string,0,1];
        [] ← P.Item[string];
        END
ELSE ans ← Fail;
END;

LenRoutine: PROCEDURE[sym: Symbol,string: Stream,count: Node] RETURNS[ans: Node] = BEGIN
-- len(n) skips n chars
n,a: LONG INTEGER;
n ← P.MakeInteger[count];
a ← LengthStream[string];
IF a < n THEN ans ← Fail
ELSE BEGIN
        ans ← SubStringStream[string,0,n];
        P.SkipStream[string,n];
        END;
END;

BlanksRoutine: PROCEDURE[sym: Symbol,string: Stream,count: Node] RETURNS[ans: Node] = BEGIN
rr: Register ← P.MRS[];
s: PLDefs.StreamRecord ← string↑;
-- blanks(n) skips n blanks, fails otherwise
n,m: LONG INTEGER;
m ← n ← P.MakeInteger[count];
WHILE n > 0 DO
        IF string.node = NIL THEN EXIT;
        IF SubStream[string, 0] ~ = ' ' THEN EXIT;
        [] ← P.Item[string];
        n ← n - 1;
        ENDLOOP;
IF n > 0 THEN ans ← Fail
ELSE ans ← SubStringStream[@s,0,m];
P.RRS[rr];
END;

StatSetup: PUBLIC PROCEDURE = BEGIN
-- pattern routines
```

```
[] ← P.Insert["word"L,PFUNC,patproc,PLDefs.Unbound,WordRoutine];
[] ← P.Insert["thing"L,PFUNC,patproc,PLDefs.Unbound,ThingRoutine];
[] ← P.Insert["integer"L,PFUNC,patproc,PLDefs.Unbound,IntegerRoutine];
[] ← P.Insert["number"L,PFUNC,patproc,PLDefs.Unbound,NumberRoutine];
[] ← P.Insert["item"L,PFUNC,patproc,PLDefs.Unbound,ItemRoutine];
[] ← P.Insert["digit"L,PFUNC,patproc,PLDefs.Unbound,DigitRoutine];
[] ← P.Insert["smallletter"L,PFUNC,patproc,PLDefs.Unbound,SmallLetterRoutine];
[] ← P.Insert["bigletter"L,PFUNC,patproc,PLDefs.Unbound,BigLetterRoutine];
[] ← P.Insert["letter"L,PFUNC,patproc,PLDefs.Unbound,LetterRoutine];
[] ← P.Insert["space"L,PFUNC,patproc,PLDefs.Unbound,SpaceRoutine];
[] ← P.Insert["len"L,PFUNC1,patproc,PLDefs.Unbound,LenRoutine];
[] ← P.Insert["blanks"L,PFUNC1,patproc,PLDefs.Unbound,BlanksRoutine];
END;

[Fail,MTSt,Nail] ← P.GetSpecialNodes[];
END.
```

```
-- storage.Mesa

DIRECTORY SystemDefs:FROM "SystemDefs",
      StorageDefs:FROM "StorageDefs";
Storage: PROGRAM IMPORTS SystemDefs EXPORTS StorageDefs =
 PUBLIC BEGIN

Allocate:PROCEDURE[size:CARDINAL] RETURNS[POINTER] =
 BEGIN  RETURN[SystemDefs.AllocateHeapNode[size]];  END;

Free:PROCEDURE[p:POINTER] =
 BEGIN  SystemDefs.FreeHeapNode[p];  END;


END.
```

```
-- store.mesa last edited by Morris, October 30, 1978  1:34 PM
                                                    DIRECTORY
                                                    PLDefs: FROM "PLDefs",
                                                    VMDefs: FROM "VMDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    ImageDefs: FROM "ImageDefs",
                                                    IODefs: FROM "IODefs",
                                                    MiscDefs: FROM "miscdefs",
                                                    AltoFileDefs: FROM "AltoFileDefs",
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    FilePageUseDefs: FROM "FilePageUseDefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    StreamDefs: FROM "StreamDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    StringDefs: FROM "StringDefs",
                                                    SegmentDefs: FROM "SegmentDefs";

store: PROGRAM IMPORTS DispDefs, MiscDefs, P:PLDefs, StringDefs, SystemDefs, VM: VMDefs, InlineDefs EXPORTS PLDefs =
BEGIN
--
Node: TYPE = PLDefs.Node;
pNode: TYPE = PLDefs.pNode;
Symbol: TYPE = PLDefs.Symbol;
Register: TYPE = PLDefs.Register;
cdebug: BOOLEAN = PLDefs.cdebug;
--
MaxNodes: CARDINAL = 256*(60/3); -- 60 pages
nSyms: CARDINAL = 250;
RSize: CARDINAL = 500;
RSIndex: CARDINAL;
sptr: CARDINAL;
PleaseGarbageCollect: BOOLEAN = FALSE;
NeverActuallyFree: BOOLEAN ← FALSE;
NeverGarbageCollect: BOOLEAN ← FALSE;
AlwaysGarbageCollect: BOOLEAN ← FALSE;
--
FreeList: Node;
FreeCount: CARDINAL;
--
GF: ARRAY PLDefs.FileIndex OF BOOLEAN; -- Garbage bits for File indices
RegisterStack: ARRAY[0..RSize] OF pNode;
SymbolArray: ARRAY[0..nSyms] OF PLDefs.SymbolRecord;
NodeArray: POINTER TO ARRAY OF PLDefs.NodeRecord;
CheckOK: BOOLEAN;
ZeroOK: BOOLEAN;
FailRecord: PLDefs.NodeRecord ← [FAIL,[TRUE,TRUE,,FALSE,],FAIL[]];
Fail: Node = @FailRecord;
MTStRecord: PLDefs.NodeRecord ← [STR,[TRUE,TRUE,simp,FALSE,0],STR[[simp[0,0]]]];
MTSt: Node = @MTStRecord;
NailRecord: PLDefs.NodeRecord ← [LIST,[TRUE,TRUE,,FALSE,],LIST[NIL,NIL]];
Nail: Node = @NailRecord;
UndefinedRecord: PLDefs.NodeRecord ← [UNDEFINED,[TRUE,TRUE,,FALSE,],UNDEFINED[]];
Undefined: Node = @UndefinedRecord;

SaveCurmap: ARRAY[0..15] OF WORD;
Curmap: POINTER TO ARRAY OF WORD = LOOPHOLE[431B];
cntr: CARDINAL;
oldnl: CARDINAL ← 60000;                    -- invalid val

-- A node may disappear if it is in ones procedure frame
-- (in a local variable or parameter list) and user-subroutine is called.
-- Nodes reachable from the symbol table, parse tree, and the specific
-- Alloc call which caused the GC will be found.
-- Thus a node need be registered only if it is a partial evaluation.

Alloc: PUBLIC PROCEDURE[r: PLDefs.NodeRecord] RETURNS[n: Node] = BEGIN
-- r must be a well formed NodeRecord
IF FreeList = NIL OR AlwaysGarbageCollect THEN GarbageCollect[@r];
n ← FreeList;
FreeList ← (LOOPHOLE[n,POINTER] + 1)↑;
n↑ ← r;
n.Des.e ← FALSE;
```

```
n.Des.b ← FALSE;
FreeCount ← FreeCount - 1;
SetCursorAmt[MaxNodes - FreeCount];
END;

CheckNode: PUBLIC PROCEDURE[n: Node] = BEGIN
p: CARDINAL ← LOOPHOLE[n];
IF cdebug THEN BEGIN
        IF n = NIL THEN P.PBug["CheckNode - n is NIL"L];
        IF (p < LOOPHOLE[@NodeArray[1],CARDINAL] OR LOOPHOLE[@NodeArray[MaxNodes-1], CARDINAL] < p) AND
        ~CheckOK THEN BEGIN
                IF n ~= Fail AND n ~= MTSt AND n~= Nail THEN P.PBug["CheckNode - n is not valid"L];
                END;
        IF n.Type = ZERO AND ~ZeroOK THEN P.PBug["CheckNode - n has type zero"L];
        END;
END;

CopyTree: PUBLIC PROCEDURE[n: Node] RETURNS[new: Node] = BEGIN
new ← NIL;
new ← Alloc[n↑];
IF new.Type = STR THEN BEGIN
        IF new.Des.s = cat THEN BEGIN
                rr: Register ← MRS[];
                R[@new];
                new.str.n1 ← CopyTree[new.str.n1];
                new.str.n2 ← CopyTree[new.str.n2];
                RRS[rr]
                END;
        END;
END;

-- be sure to avoid dangling references!!!!
FreeNode: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
p: POINTER ← n;
IF n = NIL THEN RETURN[TRUE];
IF cdebug THEN CheckNode[n];
MiscDefs.Zero[p,SIZE[PLDefs.NodeRecord]];
(p + 1)↑ ← FreeList;            -- this is a loophole
FreeList ← p;
FreeCount ← FreeCount + 1;
RETURN[TRUE];
END;

FreeTree: PUBLIC PROCEDURE[n: Node] = BEGIN
-- beware of dangling references!
-- be sure n in caller in not registered
-- the only time this may be safely used is in n in the result of a CopyTree (Start)
-- and n in not saved in any other Node
Postorder[n,FreeNode];
END;

GarbageCollect: PROCEDURE[spec: Node] =
        BEGIN
        i: CARDINAL;
        IF NeverGarbageCollect THEN P.PBug["Ran out of node space"L];
        spec.Des.g ← FALSE;
        FOR i IN[1..MaxNodes) DO
                NodeArray[i].Des.g ← FALSE;
                ENDLOOP;
        FOR i IN PLDefs.FileIndex DO GF[i]←FALSE ENDLOOP;
        P.StringGC[];
        Mark[spec];
        IF ~NeverActuallyFree THEN BEGIN
                -- this procedure builds a free list to be used by alloc
                -- it reclaims storage so is dangerous
                P.DoTransfer[];
                FreeList ← NIL;
                FreeCount ← 0;
                ZeroOK ← TRUE;
                FOR i DECREASING IN [1..MaxNodes) DO
```

```
                    IF ~NodeArray[i].Des.g THEN [] ← FreeNode[@NodeArray[i]]
                    ELSE IF NodeArray[i].Type = STR THEN P.Transfer[@NodeArray[i]];
                    ENDLOOP;
                FOR i IN [1..62] DO
                    IF ~GF[i] THEN VM.Close[i] ENDLOOP;
                ZeroOK ← FALSE;
                IF cdebug THEN DispDefs.WF1["GC: %d are free*n"L,FreeCount];
                P.Transfer[spec];
                END;
        P.EndStringGC[];
        IF FreeList = NIL THEN P.RErr["Ran out of node space"L];
        END;

GarbageRoutine: PUBLIC PROCEDURE[s: Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
-- zary
GarbageCollect[n1];
RETURN[Nail];
END;

GetCore: PUBLIC PROCEDURE[n: CARDINAL] RETURNS[p: POINTER] = BEGIN
i: CARDINAL;
p ← SystemDefs.AllocateSegment[n];
i ← SystemDefs.SegmentSize[p];
IF cdebug THEN DispDefs.WF2["Wanted %d, got %d*n"L,n,i];
IF i < n THEN P.PBug["Out of core"L];
END;

GetSpecialNodes: PUBLIC PROCEDURE RETURNS[Node,Node,Node] = BEGIN
RETURN[Fail,MTSt,Nail];
END;

IndexNode: PUBLIC PROCEDURE[n: Node] RETURNS [i: CARDINAL] = BEGIN
FOR i IN [1..MaxNodes) DO
        IF n = @NodeArray[i] THEN RETURN[i];
        ENDLOOP;
RETURN[0];
END;

Insert: PUBLIC PROCEDURE[
        n: STRING,tok: PLDefs.TokType, t: PLDefs.SType,
        p1: PROCEDURE[Symbol,Node,Node] RETURNS[Node],
        p2: PROCEDURE[Symbol,PLDefs.Stream,Node] RETURNS[Node]]
            RETURNS[s: Symbol] = BEGIN
-- these are never freed
sym: STRING ← SystemDefs.AllocateHeapString[n.length];
StringDefs.AppendString[sym,n];
SymbolArray[sptr] ← IF t = proc THEN [sym,tok,proc,proc[p1]]
        ELSE IF t = patproc THEN [sym,tok,patproc,patproc[p2]]
        ELSE [sym,tok,string,string[Undefined]];
s ← @SymbolArray[sptr];
sptr ← sptr + 1;
IF sptr > nSyms THEN P.SErr["Too many symbol table entries"L];
RETURN[s];
END;

Lookup: PUBLIC PROCEDURE[s: STRING] RETURNS[Symbol] = BEGIN
i: CARDINAL;
FOR i IN [1..sptr) DO
        IF StringDefs.EqualString[SymbolArray[i].name,s] THEN RETURN[@SymbolArray[i]];
        ENDLOOP;
RETURN[NIL];
END;

Mark: PROCEDURE[spec: Node] = BEGIN
i: CARDINAL;
f: Node;
-- this procedure marks all accessible nodes
-- it looks at all nodes in the symbol table and all nodes in the RegisterStack
-- a node not in one of those two places will be unmarked
-- spec is a special node, usually in a stack frame
MT: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
```

```
            IF n.Des.g THEN RETURN[FALSE];
            n.Des.g ← TRUE;
            IF n.Type = STR AND n.Des.s = simp THEN
                    BEGIN
                    IF n.Des.fi = 0 THEN P.LayOutBits[n];
                    GF[n.Des.fi] ← TRUE;
                    END;
            RETURN[TRUE];
            END;
IF cdebug THEN DispDefs.WF1["Marking, %d registered, "L,RSIndex-1];
CheckOK ← TRUE;
Preorder[spec,MT];
CheckOK ← FALSE;
FOR i IN [1.. sptr) DO
            IF SymbolArray[i].type = string THEN Preorder[SymbolArray[i].val,MT];
            ENDLOOP;
FOR i IN [1.. RSIndex) DO
            Preorder[RegisterStack[i]↑,MT];
            ENDLOOP;
IF ~cdebug THEN RETURN;
-- now check free list
f ← FreeList;
WHILE f ~ = NIL DO
            IF f.Des.g THEN DispDefs.WF1["Marked but on free list %b*n"L,f];
            f ← (LOOPHOLE[f,POINTER] + 1)↑;
            ENDLOOP;
END;

MRS: PUBLIC PROCEDURE RETURNS[Register] = BEGIN
RETURN[RSIndex];
END;

NonZero: PROCEDURE[p: POINTER, m: CARDINAL] RETURNS[BOOLEAN] = BEGIN
i: CARDINAL;
FOR i IN [0..m) DO
            IF (p + i)↑ ~ = 0 THEN RETURN[TRUE];
            ENDLOOP;
RETURN[FALSE];
END;

ParseTree: PUBLIC PROCEDURE[top:Node] = BEGIN
i: CARDINAL;
Preorder[top,PN];
FOR i IN [1..sptr) DO
            IF SymbolArray[i].type ~ = string THEN LOOP;
            DispDefs.WF1["%-10s:*n"L,SymbolArray[i].name];
            ENDLOOP;
END;

PN: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
DispDefs.WF1["%n*n"L,n];
RETURN[TRUE];
END;

Postorder: PUBLIC PROCEDURE[n: Node,p: PROCEDURE[Node] RETURNS[BOOLEAN]] = BEGIN
-- this proc applies p recursively to node n and in postorder all nodes accessible from it
-- p's return value is ignored
IF n = NIL THEN RETURN;
IF cdebug THEN CheckNode[n];
SELECT n.Type FROM
ZARY,FAIL,ID,PFUNC,WILD,HOLE => NULL;
STR => IF n.Des.s = cat THEN BEGIN Postorder[n.str.n1,p]; Postorder[n.str.n2,p] END;
UNARY => Postorder[n.uexp,p];
PFUNC1 => Postorder[n.pexp,p];
SEQOF,SEQOFC => Postorder[n.seqof,p];
OPT => Postorder[n.opt,p];
DELETE => Postorder[n.delete,p];
PATTERN => Postorder[n.pattern,p];
LIST => BEGIN Postorder[n.listhead,p]; Postorder[n.listtail,p] END;
CAT,CATL => BEGIN Postorder[n.left,p]; Postorder[n.right,p] END;
MATCH => BEGIN Postorder[n.div,p]; Postorder[n.patt,p] END;
PALT => BEGIN Postorder[n.alt1,p]; Postorder[n.alt2,p] END;
```

```
APPLY,MAPPLY,GOBBLE,ITER => BEGIN Postorder[n.object,p]; Postorder[n.target,p] END;
FCN => BEGIN Postorder[n.parms,p]; Postorder[n.fcn,p] END;
ASS => BEGIN Postorder[n.lhs,p]; Postorder[n.rhs,p] END;
PROG => BEGIN Postorder[n.prog1,p]; Postorder[n.prog2,p] END;
SEQUENCE,EQUAL => BEGIN Postorder[n.from,p]; Postorder[n.to,p] END;
PLUS,MINUS => BEGIN Postorder[n.arg1,p]; Postorder[n.arg2,p] END;
TILDE => Postorder[n.not,p];
ENDCASE => P.PBug["Unknown variant"L];
[] ← p[n];
END;

Preorder: PUBLIC PROCEDURE[n: Node,p: PROCEDURE[Node] RETURNS[BOOLEAN]] = BEGIN
-- this proc applies p recursively to node n and in preorder all nodes accessible from it
-- if p returns false node n's descendants are not searched
DO
IF n = NIL THEN RETURN;
IF cdebug THEN CheckNode[n];
IF ~p[n] THEN RETURN;
SELECT n.Type FROM
ZARY,FAIL,ID,PFUNC,WILD,HOLE,UNDEFINED => RETURN;
STR => IF n.Des.s = cat THEN BEGIN
                    Preorder[n.str.n1,p];
                    n ← n.str.n2;
                    END
          ELSE RETURN;
UNARY,PFUNC1 => n ← n.uexp;
SEQOF,SEQOFC,OPT,DELETE,PATTERN,TILDE => n ← n.seqof;
LIST,CAT,CATL,MATCH,PALT,APPLY,MAPPLY,GOBBLE,ITER,FCN,ASS,PROG,SEQUENCE,PLUS,MINUS,CLOSURE,ENV,EQUAL =>
BEGIN
          Preorder[n.listhead,p];
          n ← n.listtail;
          END;
ENDCASE => P.PBug["Unknown variant"];
ENDLOOP;
END;

R: PUBLIC PROCEDURE[n: pNode] = BEGIN
RegisterStack[RSIndex] ← n;
RSIndex ← RSIndex + 1;
IF RSIndex >= RSize THEN P.RErr["Internal overflow - register stack"L];
END;

ResetCursor: PUBLIC PROCEDURE = BEGIN
i: CARDINAL;
oldnl ← 60000;                          -- invalid val
FOR i IN [0..15] DO
          Curmap[i] ← SaveCurmap[i];
          ENDLOOP;
END;

RRS: PUBLIC PROCEDURE[r: Register] = BEGIN
RSIndex ← r;
END;

R2: PUBLIC PROCEDURE[n1,n2: pNode] = BEGIN
RegisterStack[RSIndex] ← n1;
RegisterStack[RSIndex + 1] ← n2;
RSIndex ← RSIndex + 2;
IF RSIndex >= RSize THEN P.RErr["Internal overflow - register stack"L];
END;

R3: PUBLIC PROCEDURE[n1,n2,n3: pNode] = BEGIN
RegisterStack[RSIndex] ← n1;
RegisterStack[RSIndex + 1] ← n2;
RegisterStack[RSIndex + 2] ← n3;
RSIndex ← RSIndex + 3;
IF RSIndex >= RSize THEN P.RErr["Internal overflow - register stack"L];
END;

SetCursorAmt: PROCEDURE[val: CARDINAL] = BEGIN
nl,i: CARDINAL;
nl ← (val + (MaxNodes/28))/(MaxNodes/14);
```

```
IF oldnl = nl THEN RETURN;
oldnl ← nl;
Curmap[0] ← Curmap[15] ← 177777B;
FOR i IN [1..14] DO
        Curmap[i] ← IF (14-i) < nl THEN
                InlineDefs.BITOR[Curmap[i],107777B]
                ELSE InlineDefs.BITAND[Curmap[i],174001B];
        ENDLOOP;
END;

CARDINALFromLI: PROCEDURE[x: LONG INTEGER] RETURNS[CARDINAL] = BEGIN
t: RECORD[low, high: CARDINAL] = LOOPHOLE[x];
RETURN[t.low];
END;

SnapRoutine: PROCEDURE[s: Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
SnapShot[];
RETURN[Nail];
END;

SnapShot: PUBLIC PROCEDURE = BEGIN
f: Node;
i: CARDINAL ← 0;
j: CARDINAL;
p: POINTER;
f ← FreeList;
WHILE f ~ = NIL DO
        i ← i + 1;
        p ← f;
        IF NonZero[p + 2,SIZE[PLDefs.NodeRecord]-2] THEN DispDefs.WF1["Is non zero %bB*n"L,f];
        f ← (p + 1)↑;
        ENDLOOP;
DispDefs.WF2["Out of %d, %d are on free list*n"L,MaxNodes-1,i];
Mark[NIL];
i ← 0;
FOR j IN [1..MaxNodes) DO
        IF NodeArray[j].Des.g THEN i ← i + 1;
        ENDLOOP;
DispDefs.WF2["%d marked*nVM file size = %d bytes*n"L,i, P.GetSin[]];
END;

StoreCleanup: PUBLIC PROCEDURE = BEGIN
s: STRING;
i: CARDINAL;
FOR i IN [1..sptr) DO
        s ← SymbolArray[i].name;
        IF s ~ = NIL THEN SystemDefs.FreeHeapString[s];
        ENDLOOP;
P.ResetCursor[];
END;

StoreReset: PUBLIC PROCEDURE = BEGIN
RSIndex ← 1;
END;

StoreSetup: PUBLIC PROCEDURE = BEGIN
p:POINTER;
i: CARDINAL;
StoreReset[];
sptr ← 1;
FreeList ← NIL;
FreeCount ← MaxNodes;
FOR i DECREASING IN [1 .. MaxNodes) DO
        p ← @NodeArray[i];
        MiscDefs.Zero[p,SIZE[PLDefs.NodeRecord]];
        (p + 1)↑ ← FreeList;                                    -- this is a loophole
        FreeList ← p;
        ENDLOOP;
[] ← Insert["snap"L,ZARY,proc,SnapRoutine,PLDefs.Unbound];
[] ← Insert["garbage"L,ZARY,proc,GarbageRoutine,PLDefs.Unbound];
[] ← Insert["symbol"L,ZARY,proc,SymbolRoutine,PLDefs.Unbound];
SetCursorAmt[0];
```

```
Curmap[0] ← Curmap[15] ← 177777B;
FOR i IN [1..14] DO
        Curmap[i] ← 104001B;
        ENDLOOP;
CheckOK ← ZeroOK ← FALSE;
END;

SymbolRoutine: PUBLIC PROCEDURE[s: Symbol,n1,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
t: PLDefs.SType;
n: Node ← NIL;
i: CARDINAL;
rr: Register ← P.MRS[];
ans ← Nail;
P.R2[@ans,@n];
FOR i IN [1..sptr) DO
        t ← SymbolArray[i].type;
        IF t ~ = string THEN LOOP;
        n ← P.MakeSTR[SymbolArray[i].name];
        ans ← P.Alloc[[LIST,,LIST[n,ans]]];
        ENDLOOP;
P.RRS[rr];
END;


NodeArray ← P.GetCore[MaxNodes * SIZE[PLDefs.NodeRecord]];
FOR cntr IN [0..15] DO
        SaveCurmap[cntr] ← Curmap[cntr];
        ENDLOOP;
END.
```

```
                                                    DIRECTORY
                                                    VMDefs: FROM "VMDefs",
                                                    PLDefs: FROM "PLDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    IODefs: FROM "IODefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    MiscDefs: FROM "MiscDefs",
                                                    SystemDefs: FROM "SystemDefs";

string: PROGRAM IMPORTS VM: VMDefs, DispDefs, P:PLDefs, SystemDefs, MiscDefs, InlineDefs EXPORTS PLDefs = BEGIN
--
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
String: TYPE = PLDefs.String;
Stream: TYPE = PLDefs.Stream;
Register: TYPE = PLDefs.Register;
LongCARDINAL: TYPE = InlineDefs.LongCARDINAL;
cdebug: BOOLEAN ← PLDefs.cdebug;
--
StringSize: LONG INTEGER ← 177777B;
sin: CARDINAL;
Bit: ARRAY[0..15] OF CARDINAL = [
1B,2B,4B,10B,20B,40B,100B,200B,
400B,1000B,2000B,4000B,10000B,20000B,40000B,100000B];
StringBitMap: POINTER TO ARRAY OF CARDINAL;
RangeSize: CARDINAL = 384;
Range: POINTER TO ARRAY[0..RangeSize) OF RangeRecord;
RangeRecord: TYPE = RECORD[
    old,new: CARDINAL
    ];
inx: CARDINAL;
StringGarbageCollect: BOOLEAN ← TRUE;
StringChanged: BOOLEAN;
MinFileStringSize: CARDINAL = 256;
MinCatSize: CARDINAL = 100;
MinCoerceSize: CARDINAL = 100;
LayMax: CARDINAL;
Def: PLDefs.DesRecord = [FALSE, FALSE, simp, FALSE, 0]; --default
-- one may not assume that for cat STRs n1 is of non-zero length; it may be empty
MTSt: Node;

AppendDigits: PROCEDURE [of: LONG INTEGER] =
    -- of will be non-neg
    BEGIN
    ii: LONG INTEGER;
    Check[1];
    IF of<10 THEN
                BEGIN
                VM.SetSChar[sin,0,VM.Cardinal[of] + '0];
                sin ← sin + 1;
                RETURN;
                END;
    AppendDigits[of/10];
    ii ← of - ((of/10)*10);
    VM.SetSChar[sin,0,VM.Cardinal[ii] + '0];
    sin ← sin + 1;
    END;

Check: PROCEDURE[i: LONG INTEGER] = BEGIN
-- There is space for i more chars in VMFile
IF sin + i >= StringSize THEN P.RErr["Too many strings in memory for program"L];
END;

Coerce: PUBLIC PROCEDURE[n: Node] RETURNS[ans: Node] = BEGIN
-- n is a STR - make sure it is represented as a simple string in memory
-- i.e., convert from file or cat to simp
rr: Register;
ns: PLDefs.StreamRecord;
ii: LONG INTEGER;
j,m: CARDINAL;
IF n = NIL OR n.Type ~= STR THEN P.PBug["coerce wants string"L];
IF n.Des.s = simp AND n.Des.fi=0 THEN RETURN[n];
```

```
rr ← P.MRS[];
ii ← P.Length[n];
Check[ii];
ns ← NewStream[n];
P.R[@ns.node];
m ← VM.Cardinal[ii];
FOR j IN [0..m) DO
    VM.SetSChar[sin + j, 0, Item[@ns]];
    ENDLOOP;
ans ← P.Alloc[[STR,Def,STR[[simp[sin,m]]]]];
sin ← sin + m;
StringChanged ← TRUE;
P.RRS[rr];
END;

FileRoutine: PUBLIC PROCEDURE[s: Symbol,name,n2: Node] RETURNS[Node] = BEGIN
RETURN[CommonFileRoutine[name, FALSE]];
END;

CFileRoutine: PUBLIC PROCEDURE[s: Symbol,name,n2: Node] RETURNS[Node] = BEGIN
RETURN[CommonFileRoutine[name, TRUE]];
END;

CommonFileRoutine: PROCEDURE[name: Node, coerce:BOOLEAN] RETURNS[ans:Node] = BEGIN
fname: STRING ← [100];
rr: Register ← P.MRS[];
fi: PLDefs.FileIndex;
len: LONG INTEGER;
IF name = NIL OR name.Type ~ = STR THEN P.RErr["File name must be string"L];
IF P.Length[name] > = 100 THEN P.RErr["File name too long (>100)"L];
ans ← NIL;
P.R[@ans];
MakeString[fname,name];
[fi, len] ← VM.Open[fname];
IF fi = 0 THEN ans ← P.Alloc[[FAIL,,FAIL[]]]
ELSE IF len>177777B THEN P.RErr["File to big (>64K bytes)"]
ELSE BEGIN
    ans ← P.Alloc[[STR,[,,simp,,fi],STR[[simp[0,VM.Cardinal[len]]]]]];
    IF coerce OR ans.str.length < MinCoerceSize THEN ans ← P.Coerce[ans];
    END;
P.RRS[rr];
END;

Item: PUBLIC PROCEDURE[s: Stream] RETURNS[c: CHARACTER] = BEGIN
-- the string in s must have been linearized!!!
-- i.e. at worst, it is a right linear tree of cat nodes.
IF s.node = NIL THEN RETURN[0C];
c ← s.buff[s.cp];
s.cp←s.cp + 1;
IF s.cp = s.len THEN LoadBuff[s];
END;

LoadBuff: PUBLIC PROCEDURE[s: Stream] =
    BEGIN
    n: Node ← s.node;
    slength,bite: CARDINAL;
    WHILE n.Des.s = cat DO  --iterate only to discard empty strings
                            IF s.posn # n.str.n1.str.length THEN
                                BEGIN n←n.str.n1; EXIT END;
                            s.node←n←n.str.n2;
                            s.posn ← 0;
                            ENDLOOP;
    -- n is simp
    slength←n.str.length;
    IF s.posn = slength THEN -- n must have been simp at start
                s.node←NIL
    -- now old nodes have be thrown away if necessary
    -- slength is size of n
    ELSE BEGIN
                bite ← MIN[slength-s.posn, PLDefs.SBSize];
                s.cp ← PLDefs.SBSize; -- pretending it's a string
                VM.GetString[n.str.start + s.posn, bite, n.Des.fi, LOOPHOLE[s]];
```

```
                s.posn←s.posn + bite;
                s.cp←0;
                END;
    END;

MakeInteger: PUBLIC PROCEDURE[n: Node] RETURNS[cnt: LONG INTEGER] = BEGIN
i,j: CARDINAL;
a,b: LONG INTEGER;
s: String;
wk: STRING ← [20];
IF n.Type ~= STR THEN P.RErr["Expecting integer as a string"L];
IF P.Length[n] = 0 THEN P.RErr["String is not a number"L];
IF P.Length[n] >20 THEN P.RErr["Number too big (>20 digits)"L];
s ← @n.str;
j ← 0;
cnt ← 0;
IF n.Des.s = simp THEN BEGIN
    VM.GetString[s.start,s.length,n.Des.fi,wk];
    IF wk[0] = '- THEN j ← 1;
    FOR i IN [j..s.length) DO
                IF wk[i] ~IN ['0..'9] THEN P.RErr["String is not a number"L];
                cnt ← cnt *10 + (wk[i]-'0);
                ENDLOOP;
    END
ELSE IF n.Des.s = cat THEN BEGIN
    a ← MakeInteger[s.n1];
    b ← MakeInteger[s.n2];
    THROUGH[1..VM.Cardinal[P.Length[s.n2]]] DO
                a ← a * 10;
                ENDLOOP;
    cnt ← a + b;
    END
ELSE P.PBug["unknown variant"L];
IF j = 1 THEN cnt ← -cnt;
END;

MakeNUM: PUBLIC PROCEDURE[i: LONG INTEGER] RETURNS [n: Node] =
    BEGIN
    rr: Register;
    n ← P.Alloc[[STR,Def,STR[[simp[sin, 0]]]]];
    n.str.start ← sin; -- shoudn't be necesary, GC bug
    rr ← P.MRS[];
    P.R[@n];
    IF i < 0 THEN BEGIN
                i ← -i;
                VM.SetSChar[sin, 0, '-];
                sin ← sin + 1;
                END;
    AppendDigits[i];
    n.str.length ← sin - n.str.start;
    StringChanged ← TRUE;
    P.RRS[rr];
    END;

MakeSTR: PUBLIC PROCEDURE[s: STRING] RETURNS[n:Node] = BEGIN
IF s.length = 0 THEN RETURN[MTSt];
Check[s.length];
VM.SetString[sin, 0, s];
n ← P.Alloc[[STR,Def,STR[[simp[sin,s.length]]]]];
sin ← sin + s.length;
StringChanged ← TRUE;
END;

MakeString: PUBLIC PROCEDURE[s: STRING,n: Node] = BEGIN
i: CARDINAL ← 0;
ns: PLDefs.StreamRecord;
rr: Register ← P.MRS[];
ns ← P.NewStream[n];
P.R[@ns.node];
WHILE ns.node ~= NIL DO
    IF i >= s.maxlength THEN P.RErr["String too big for makestring"L];
    s[i] ← P.Item[@ns];
```

```
    i ← i + 1;
    ENDLOOP;
s.length ← i;
P.RRS[rr];
END;

NewStream: PUBLIC PROCEDURE[n: Node] RETURNS[s: PLDefs.StreamRecord] = BEGIN
-- assume STR w/cat node is non-empty
-- the value returned node must be registered by caller:
IF n = NIL OR n.Type ~ = STR THEN P.PBug["NewStream expects a STR"L];
IF n.Des.s ~ = cat AND P.Length[n] = 0 THEN n ← NIL
ELSE IF ~n.Des.b THEN [n,] ← P.LinearSTR[n];
s ← [len: 0, cp: 0, buff:, node: n, posn: 0];
LoadBuff[@s];
END;

SimpConcat: PROCEDURE[i, j: Node] RETURNS [n: Node] = BEGIN
-- i, j are STRs, i.Des.s = simp  AND j.Des.s = simp
a, in: CARDINAL;
s: String;
g,h: CARDINAL;
g ← i.str.length;
h ← j.str.length;
IF g = 0 THEN RETURN[j];
IF h = 0 THEN RETURN[i];
a ← i.str.start + g;
IF i.Des.fi = j.Des.fi AND a = j.str.start THEN
    n ← P.Alloc[[STR,[,,simp,,i.Des.fi],STR[[simp[i.str.start,g + h]]]]]
ELSE IF i.Des.fi = 0 AND a = sin AND h<MinCatSize THEN  -- can just bang onto end
    BEGIN
    -- too small, make a new STR
    wk: STRING = SystemDefs.AllocateHeapString[h];
    BEGIN ENABLE UNWIND => SystemDefs.FreeHeapString[wk];
    Check[h];
    s ← @j.str;
    VM.GetString[s.start,h,j Des.fi,wk];
    VM.SetString[sin, 0, wk];
    sin ← sin + h;
    StringChanged ← TRUE;
    n ← P.Alloc[[STR,Def,STR[[simp[i.str.start,g + h]]]]];
    SystemDefs.FreeHeapString[wk];
    END;
    END
ELSE IF g + h < MinCatSize THEN BEGIN
    -- too small, make a new STR
    wk: STRING = SystemDefs.AllocateHeapString[MAX[g,h]];
    BEGIN ENABLE UNWIND => SystemDefs.FreeHeapString[wk];
    Check[g + h];
    in ← sin;
    s ← @i.str;
    VM.GetString[s.start,g,i.Des.fi,wk];
    VM.SetString[sin, 0, wk];
    sin ← sin + g;
    s ← @j.str;
    VM.GetString[s.start,h,j.Des.fi,wk];
    VM.SetString[sin, 0, wk];
    sin ← sin + h;
    StringChanged ← TRUE;
    n ← P.Alloc[[STR,Def,STR[[simp[in,sin-in]]]]];
    SystemDefs.FreeHeapString[wk];
    END;
    END
ELSE n ← P.Alloc[[STR,[,,cat,,],STR[[cat[i,j]]]]];
END;

SkipStream: PUBLIC PROCEDURE[s: Stream,inx: LONG INTEGER] = BEGIN
n: Node;
ii: LONG INTEGER;
-- discard buffer
s.posn ← s.posn - (s.len-s.cp);
s.cp ← s.len ← 0;
WHILE s.node ~ = NIL DO
```

```
    n ← s.node;
    IF n.Des.s = simp THEN BEGIN
                ii ← n.str.length;
                s.posn ← s.posn + VM.Cardinal[inx];
                IF s.posn > ii THEN P.PBug["bad string length skip"L];
                IF s.posn = ii THEN s.node ← NIL;
                RETURN;
                END
    ELSE IF n.Des.s = cat THEN BEGIN
                ii ← P.Length[n.str.n1];
                IF s.posn + inx >= ii THEN BEGIN
                            inx ← inx - (ii-s.posn);
                            s.posn ← 0;
                            s.node ← s.node.str.n2;
                            END
                ELSE BEGIN
                            s.posn ← s.posn + VM.Cardinal[inx];
                            RETURN;
                            END
                END
    ELSE P.PBug["unknown str type - skipstream"L];
    ENDLOOP;
IF inx > 0 THEN P.PBug["skip past end"L];
LoadBuff[s];
END;

StringConcat: PUBLIC PROCEDURE[i, j: Node] RETURNS [n: Node] = BEGIN
IF i.Type # STR OR j.Type # STR THEN P.PBug["Bad concat"L];
-- try to avoid making CAT node
IF i.Des.s = simp THEN
    BEGIN
    IF j.Des.s = simp THEN RETURN[SimpConcat[i,j]];
    IF j.str.n1.Des.s = simp AND
                i.str.length + j.str.n1.str.length<MinCatSize THEN
                            RETURN[P.Alloc[[STR,[,,cat,,],
                                STR[[cat[SimpConcat[i, j.str.n1],j.str.n2]]]]]];
    END;
IF j.Des.s = simp THEN
    BEGIN
    -- i.Des.s = cat
    IF i.str.n2.Des.s = simp AND
                j.str.length + i.str.n2.str.length<MinCatSize THEN
                            RETURN[P.Alloc[[STR,[,,cat,,],
                                STR[[cat[i.str.n1, SimpConcat[i.str.n2, j]]]]]]];
    END;
RETURN[P.Alloc[[STR,[,,cat,,],STR[[cat[i,j]]]]]];
END;

Sub: PUBLIC PROCEDURE[n: Node,inx: LONG INTEGER] RETURNS[CHARACTER] = BEGIN
i: CARDINAL;
len: LONG INTEGER;
DO
    IF n = NIL OR n.Type ~= STR OR inx < 0 THEN P.PBug["bad sub"L];
    IF n.Des.s = simp THEN BEGIN
                IF inx >= n.str.length THEN P.RErr["bad length - sub"L];
                i ← VM.Cardinal[inx];
                RETURN[VM.GetSChar[n.str.start + i, n.Des.fi]];
                END
    ELSE IF n.Des.s = cat THEN BEGIN
                len ← P.Length[n.str.n1];
                -- recursion
                IF len > inx THEN n ← n.str.n1
                ELSE BEGIN
                            n ← n.str.n2;
                            inx ← inx - len;
                            END;
                END
    ELSE P.PBug["invalid str"L];
    ENDLOOP;
END;

SubString: PUBLIC PROCEDURE[n: Node, ii, jj: LONG INTEGER] RETURNS [ans: Node] =
```

```
BEGIN
-- ii if the first char, jj-1 is the last char returned
rr: Register ← P.MRS[];
ns1, ns2: Node ← NIL;
str: String;
P.R2[@ns1, @ns2];
str ← @n.str;
IF ii> = jj THEN ans←MTSt
ELSE IF ii = 0 AND P.Length[n] = jj THEN ans ← n
ELSE IF n.Des.s = simp THEN BEGIN
        i, j: CARDINAL;
        i ← VM.Cardinal[ii];
        j ← VM.Cardinal[jj];
        IF j > str.length THEN P.PBug["SubString out of bounds"L];
        ans ←P.Alloc[[STR,[,,simp,,n.Des.fi],STR[[simp[str.start + i, j-i]]]]];
         -- disable this for a while
        IF FALSE AND ans.Des.fi # 0 AND ans.str.length<MinFileStringSize THEN
                BEGIN
                wk: STRING ← [256];
                VM.GetString[ans.str.start, ans.str.length, n.Des.fi, wk];
                Check[ans.str.length];
                VM.SetString[sin, 0, wk];
                ans.Des.fi←0;
                ans.str.start ← sin;
                sin ← sin + ans.str.length;
                StringChanged ← TRUE;
                END;
        END
ELSE IF n.Des.s = cat THEN
        BEGIN
        kk: LONG INTEGER ← P.Length[str.n1];
        IF kk < = ii THEN ans ← SubString[str.n2,ii-kk,jj-kk]
        ELSE IF jj<kk THEN ans ← SubString[str.n1, ii, jj]
        ELSE BEGIN
                ns1 ← SubString[str.n1,ii,kk];
                ns2 ← SubString[str.n2,0,jj-kk];
                ans ← P.StringConcat[ns1, ns2];
                END
        END
ELSE P.PBug["unknown variant"L];
P.RRS[rr];
END;


-- String Garbage Collection

Mask: PROCEDURE[w1,b1,w2,b2: CARDINAL] = BEGIN
ONES: CARDINAL = 177777B;
i: CARDINAL;
top: CARDINAL;
IF w1 > w2 OR (w1 = w2 AND b1 > b2) THEN P.RErr["bit map bad"L];
top ← IF w1 = w2 THEN b2 ELSE 15;
FOR i IN [b1..top] DO
    StringBitMap[w1] ← InlineDefs.BITOR[StringBitMap[w1],Bit[i]];
    ENDLOOP;
IF w1 = w2 THEN RETURN;
FOR i IN (w1..w2) DO
    StringBitMap[i] ← ONES;
    ENDLOOP;
FOR i IN [0..b2] DO
    StringBitMap[w2] ← InlineDefs.BITOR[StringBitMap[w2],Bit[i]];
    ENDLOOP;
END;

BitSet: PROCEDURE[i: CARDINAL] RETURNS[BOOLEAN] = BEGIN
w,b,m: CARDINAL;
w ← i/16;
b ← i MOD 16;
-- is the bit b in word w set?
m ← Bit[b];
RETURN[InlineDefs.BITAND[StringBitMap[w],m] ~ = 0];                -- if set will be true
```

```
END;

LayOutBits: PUBLIC PROCEDURE[n: Node] = BEGIN
i: CARDINAL;
i1,i2,j1,j2: CARDINAL;
s: String;
IF ~StringGarbageCollect OR ~StringChanged THEN RETURN;
IF n = NIL OR n.Type ~ = STR OR n.Des.s ~ = simp THEN RETURN;
s ← @n.str;
i1 ← s.start/16;
i2 ← s.start MOD 16;
i ← IF s.length > 0 THEN s.start + s.length - 1 ELSE s.start;
j1 ← i/16;
j2 ← i MOD 16;
Mask[i1,i2,j1,j2];
IF LayMax < i THEN LayMax ← i;
-- IF i > = sin THEN P.PBug["i > sin"L];
END;

StringGC: PUBLIC PROCEDURE = BEGIN
StringBitMap ← P.GetCore[(sin + 15)/16];
Range ← P.GetCore[RangeSize * SIZE[RangeRecord]];
MiscDefs.Zero[StringBitMap,(sin + 15)/16];
LayMax ← 0;
END;

EndStringGC: PUBLIC PROCEDURE = BEGIN
SystemDefs.FreeSegment[StringBitMap];
SystemDefs.FreeSegment[Range];
END;

DoTransfer: PUBLIC PROCEDURE = BEGIN
i,j: CARDINAL;
run: BOOLEAN;
-- i is the old subscript, j the new one
Range[0] ← [0,0];
inx ← 1;
IF ~StringGarbageCollect OR ~StringChanged THEN RETURN;
IF cdebug THEN DispDefs.WF1["SOlde: %u, "L,sin];
run ← FALSE;
j ← 1;
FOR i IN [1..LayMax] DO
    IF BitSet[i] THEN BEGIN
                IF ~run THEN BEGIN
                            Range[inx] ← [i,j];
                            inx ← inx + 1;
                            IF inx > = RangeSize THEN P.RErr["Too many small strings"L];
                            run ← TRUE;
                            END;
                IF i # j THEN VM.SetSChar[j, 0, VM.GetSChar[i,0]];
                j ← j + 1;
                END
    ELSE run ← FALSE;
    ENDLOOP;
Range[inx] ← [LayMax + 1,j];
inx ← inx + 1;
-- IF LayMax > = sin THEN P.PBug["Layout touched bit > old sin"L];
sin ← j;
IF cdebug THEN DispDefs.WF1["SNewe %u, "L,sin];
StringChanged ← FALSE;
END;

-- put in new values
Transfer: PUBLIC PROCEDURE[n:Node] = BEGIN
s: String;
j,d: CARDINAL;
IF ~StringGarbageCollect OR inx < = 2 THEN RETURN;
IF n = NIL OR n.Type ~ = STR OR n.Des.s ~ = simp OR n.Des.fi # 0 THEN RETURN;
s ← @n.str;
j ← 1;
-- note that this is linear search
WHILE j < inx DO
```

```
    IF s.start < Range[j].old THEN EXIT;
    j ← j + 1;
    ENDLOOP;
IF s.start + s.length > StringSize THEN P.PBug["bad string index"L];
IF j >= inx THEN P.PBug["transfer cant happen"L];
-- this run begins at Range[j-1].old
d ← s.start - Range[j-1].old;
s.start ← Range[j-1].new + d;
IF s.start + s.length > StringSize THEN P.PBug["bad string index1"L];
END;

StringDebugging: PUBLIC PROCEDURE = BEGIN
END;

StringSetup: PUBLIC PROCEDURE = BEGIN
-- must follow vmsetup
sin ← 1;
[] ← P.Insert["file"L,ZARY,proc,FileRoutine,PLDefs.Unbound];
[] ← P.Insert["cfile"L,ZARY,proc,CFileRoutine,PLDefs.Unbound];
StringChanged ← FALSE;
END;

StringCleanup: PUBLIC PROCEDURE = BEGIN
END;

GetSin: PUBLIC PROCEDURE RETURNS[CARDINAL] = BEGIN
RETURN[sin];
END;

[,MTSt,] ← P.GetSpecialNodes[];
END.
```

```
-- sup.mesa last edited by morris, November 7, 1978  2:43 PM
                                                        DIRECTORY
                                                        PLDefs: FROM "PLDefs",
                                                        SystemDefs: FROM "SystemDefs",
                                                        InlineDefs: FROM "InlineDefs";

sup: PROGRAM IMPORTS P:PLDefs, SystemDefs EXPORTS PLDefs  = BEGIN
--
TokType: TYPE = PLDefs.TokType;
NodeType: TYPE = PLDefs.NodeType;
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
Register: TYPE = PLDefs.Register;
--
PBug: PUBLIC SIGNAL[STRING] = CODE;
SErr: PUBLIC SIGNAL[est: STRING] = CODE;
RErr: PUBLIC SIGNAL[est: STRING] = CODE;
EndDisplay: PUBLIC SIGNAL = CODE;
Interrupt: PUBLIC SIGNAL = CODE;
--

ListRecord: TYPE = RECORD[old,new: Node, pref: LONG INTEGER,number: BOOLEAN];
List: POINTER TO ARRAY OF ListRecord;
lin: CARDINAL;
ascii: BOOLEAN;
Nail: Node;

SortRoutine: PROCEDURE[sym:Symbol,input,func: Node] RETURNS[Node] = BEGIN
ascii ← FALSE;
RETURN[sort[input,func]];
END;

ASortRoutine: PROCEDURE[sym:Symbol,input,func: Node] RETURNS[Node] = BEGIN
ascii ← TRUE;
RETURN[sort[input,func]];
END;

sort: PROCEDURE[input,func: Node] RETURNS[j:Node] = BEGIN
-- zary
rr: Register ← P.MRS[];
i,m: CARDINAL;
j ← NIL;
P.R2[@j,@input];
i ← P.LengthList[input];
IF i <= 1 THEN j ← input
ELSE BEGIN
        input ← P.FixRep[input];
        List ← P.GetCore[i*SIZE[ListRecord]];
        lin ← 0;
        FOR m IN [0..i) DO
                List[m].number ← FALSE;
                ENDLOOP;
        P.Map[input,SR];
        -- setting up simple random number generaor
        TreeSort[lin];                          -- sort the list
        j ← P.Alloc[[LIST,,LIST[List[lin-1].old,Nail]]];
        IF lin > 1 THEN FOR i DECREASING IN [0.. (lin-2)] DO
                j ← P.Alloc[[LIST,,LIST[List[i].old,j]]];
                ENDLOOP;
        SystemDefs.FreeSegment[List];
        END;
P.RRS[rr];
RETURN[j];
END;

SR: PROCEDURE[n: Node] = BEGIN
rr: Register ← P.MRS[];
ns: PLDefs.StreamRecord;
List[lin].old ← n;
IF n.Type = LIST THEN n ← n.listhead;
```

```
List[lin].new ← n;
IF List[lin].new.Type ~ = STR THEN P.RErr["sort expects a list of strings"L];
ns ← P.NewStream[List[lin].new];
P.R[@ns.node];
List[lin].number ← FALSE;
IF ~ascii THEN
        BEGIN
        c: CHARACTER;
        neg: BOOLEAN ← FALSE;
        n: LONG INTEGER ← 0;
        c ← P.Item[@ns];
        IF c = '- THEN BEGIN neg ← TRUE; c ← P.Item[@ns] END;
        IF c = 0C THEN GOTO NotNum;
        UNTIL c = 0C DO
                IF c NOT IN ['0..'9] THEN GOTO NotNum;
                n ← n*10 + (c-'0);
                c ← P.Item[@ns];
                ENDLOOP;
        IF neg THEN n ← -n;
        List[lin].number ← TRUE;
        List[lin].pref ← n
        EXITS
        NotNum => NULL;
        END;
IF ~List[lin].number THEN
        BEGIN
        s: STRING;
        ns ← P.NewStream[List[lin].new];
        s ← LOOPHOLE[@List[lin].pref - 2];
        s[2] ← P.Item[@ns];
        s[3] ← P.Item[@ns];
        s[0] ← P.Item[@ns];
        s[1] ← P.Item[@ns];
        END;
lin ← lin + 1;
P.RRS[rr];
END;

TreeSort: PROCEDURE[N: INTEGER] = BEGIN
t: ListRecord;
siftUp: PROCEDURE[low, high: INTEGER] =
        BEGIN k, son: INTEGER;
        k ← low;
        DO
                IF 2*k>high THEN EXIT;
                IF 2*k + 1>high OR LessThan[2*k + 1-1, 2*k-1] THEN son ← 2*k ELSE son ← 2*k + 1;
                IF LessThan[son-1, k-1] THEN EXIT;
                t ← List[son-1]; List[son-1] ← List[k-1]; List[k-1] ← t;
                k ← son;
                ENDLOOP;
        END;
i: INTEGER;
FOR i DECREASING IN [1..N/2] DO siftUp[i,N] ENDLOOP;
FOR i DECREASING IN [1..N) DO
        t ← List[1-1]; List[1-1] ← List[i + 1-1]; List[i + 1-1] ← t;
        siftUp[1,i]
        ENDLOOP;
END;

LessThan: PROCEDURE[i,j: INTEGER] RETURNS[BOOLEAN] = BEGIN
res: INTEGER;
rr: Register;
n1, n2: PLDefs.StreamRecord;
c1,c2: CHARACTER;
IF List[i].number AND List[j].number THEN BEGIN
        IF List[i].pref < List[j].pref THEN RETURN[TRUE]
        ELSE RETURN[FALSE];
        END;
IF ~List[i].number AND ~List[j].number THEN BEGIN
        res ← USC[List[i].pref,List[j].pref];
```

```
        IF res ~ = 0 THEN RETURN[res = -1];
        END;
rr ← P.MRS[];
n1.node ← n2.node ← NIL;
P.R2[@n1.node, @n2.node];
n1 ← P.NewStream[List[i].new];
n2 ← P.NewStream[List[j].new];
DO
        c1 ← P.Item[@n1];
        c2 ← P.Item[@n2];
        IF c1 > c2 THEN BEGIN
                res ← 1;
                EXIT;
                END;
        IF c1 < c2 THEN BEGIN
                res ← -1;
                EXIT;
                END;
        IF c1 = 0C THEN BEGIN
                res ← 0;
                EXIT;
                END;
        ENDLOOP;
P.RRS[rr];
RETURN[res = -1];
END;

USC: PROCEDURE[a,b: LONG INTEGER] RETURNS[INTEGER] = BEGIN
-- unsigned compare, treat a and b as LONG CARDINALS
-- return -1 if a<b, 0 if a = b, 1 if a>b
i: InlineDefs.LongCARDINAL ← LOOPHOLE[a];
j: InlineDefs.LongCARDINAL ← LOOPHOLE[b];
IF i>j THEN RETURN[1];
IF i<j THEN RETURN[-1];
RETURN[0];
END;

USortRoutine: PROCEDURE[sym:Symbol,input,func: Node] RETURNS[ans:Node] = BEGIN
-- unary
rr: Register ← P.MRS[];
n: Node;
ans ← NIL;
P.R[@ans];
ans ← ASortRoutine[sym,input,func];
IF ans.listhead = NIL THEN RETURN;
n ← ans;
WHILE n.listtail.listhead ~ = NIL DO
        IF Equal[n.listhead,n.listtail.listhead] THEN n↑ ← n.listtail↑
        ELSE n ← n.listtail
        ENDLOOP;
P.RRS[rr];
END;

FactorRoutine: PROCEDURE[sym:Symbol,input,func: Node] RETURNS[ans:Node] = BEGIN
-- unary
rr: Register ← P.MRS[];
key, n: Node;
target: POINTER TO Node;
Ch: PROCEDURE[n: Node] =
        BEGIN
        IF n.Type # LIST AND n.listhead.Type # STR THEN P.RErr["Illegal input to factor"];
        END;
ans ← NIL;
P.R[@ans];
P.Map[input,Ch];
ans ← n ← SortRoutine[sym,input,func];
UNTIL n.listhead = NIL DO
        key ← n.listhead.listhead;
        n.listhead ← P.Alloc[[LIST,,LIST[key,
```

```
                                                P.Alloc[[LIST,,LIST[n.listhead.listtail,Nail]]]]];
        target ← @n.listhead.listtail.listtail;
        n←n.listtail;
        WHILE n.listhead # NIL AND Equal[key,n.listhead.listhead] DO
                targett ← P.Alloc[[LIST,,LIST[n.listhead.listtail,Nail]]];
                target ← @target.listtail;
                nt ← n.listtailt
                ENDLOOP;
        ENDLOOP;
P.RRS[rr];
END;

Equal: PROCEDURE[i,j: Node] RETURNS[res: BOOLEAN] = BEGIN
rr: Register ← P.MRS[];
a,b: PLDefs.StreamRecord;
IF i.Type ~ = STR OR j.Type ~ = STR THEN P.RErr["usort only compares lists of strings"L];
a.node ← b.node ← NIL;
P.R2[@a.node,@b.node];
a ← P.NewStream[i];
b ← P.NewStream[j];
DO
        IF a.node = NIL THEN BEGIN res ← b.node = NIL; EXIT END;
        IF b.node = NIL THEN BEGIN res ← FALSE; EXIT END;
        IF P.Item[@a] ~ = P.Item[@b] THEN
                BEGIN res ← FALSE; EXIT END;
        ENDLOOP;
P.RRS[rr];
END;

SupSetup: PUBLIC PROCEDURE = BEGIN
[] ← P.Insert["sort"L,ZARY,proc,SortRoutine,PLDefs.Unbound];
[] ← P.Insert["usort"L,ZARY,proc,USortRoutine,PLDefs.Unbound];
[] ← P.Insert["asort"L,ZARY,proc,ASortRoutine,PLDefs.Unbound];
[] ← P.Insert["factor"L,ZARY,proc,FactorRoutine,PLDefs.Unbound];
SupReset[];
END;

SupReset: PUBLIC PROCEDURE = BEGIN
END;

[,,Nail] ← P.GetSpecialNodes[];

END.
```

```
                                                    DIRECTORY
                                                    VMDefs: FROM "VMDefs",
                                                    PLDefs: FROM "PLDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    IODefs: FROM "IODefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    FilePageUseDefs: FROM "FilePageUseDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    StringDefs: FROM "StringDefs";

VM: PROGRAM IMPORTS P:PLDefs, FSys: FileSystemDefs, FPU: FilePageUseDefs, ST: StringDefs, InlineDefs EXPORTS VMDefs =
BEGIN
--
FileIndex: TYPE = VMDefs.FileIndex;
bSize: CARDINAL = 512;
NPages: CARDINAL = 128;
nbuf: CARDINAL = 14;
FilePage: TYPE = RECORD[f: FileIndex, p: [0..256)];
AC: TYPE = {dead, alive};
InP: ARRAY [0..nbuf) OF RECORD[mod: BOOLEAN, active: AC, fp: FilePage];
StringArray: ARRAY [0..nbuf) OF PACKED ARRAY [0..bSize) OF CHARACTER;
FTable: ARRAY FileIndex OF FilePageUseDefs.FileHandle;
RWTable: ARRAY FileIndex OF BOOLEAN;
fs: FileSystemDefs.FileSystem ← NIL;
bptr: CARDINAL;
lastval: FilePage;
lastinx: CARDINAL ← 60000;
CurMap: POINTER TO ARRAY OF UNSPECIFIED = LOOPHOLE[431B];

Cardinal: PUBLIC PROCEDURE[in: LONG INTEGER] RETURNS [CARDINAL] = BEGIN
IF in<0 OR in > 177777B THEN P.PBug["integer too big"];
RETURN[LOOPHOLE[in, num InlineDefs.LongNumber].lowbits];
END;

GetSChar: PUBLIC PROCEDURE[in: LONG INTEGER, fi: FileIndex] RETURNS[CHARACTER] = BEGIN
n,p,b: CARDINAL;
p ← Cardinal[in/bSize];
n ← Cardinal[in MOD bSize];
-- p is page # [0..NPages), n is char on page [0..bSize)
b ← GetPage[[fi, p]];
RETURN[(StringArray[b])[n]];
END;

FS: PUBLIC PROCEDURE RETURNS [FileSystemDefs.FileSystem] =
        BEGIN RETURN[fs] END;

SetSChar: PUBLIC PROCEDURE[in: LONG INTEGER, fi: FileIndex, c: CHARACTER] = BEGIN
b,n,p: CARDINAL;
p ← Cardinal[in/bSize];
n ← Cardinal[in MOD bSize];
-- p is page # [0..NPages), n is char on page [0..bSize)
b ← GetPage[[fi, p]];
StringArray[b][n] ← c;
InP[b].mod ← TRUE;
END;

VMSetup: PUBLIC PROCEDURE = BEGIN
i: CARDINAL;
FOR i IN FileIndex DO
        FTable[i]←NIL;
        ENDLOOP;
IF fs = NIL THEN BEGIN
        fs ← FSys.Login['a,NIL,NIL,NIL];
        FTable[0] ← FSys.Open[fs,"Poplar.VMFile$"L, FSys.OpenMode[create] ! FSys.FileAlreadyExists => RESUME];
        RWTable[0] ← TRUE;
        END;
FOR i IN [0..nbuf) DO
        InP[i] ← [FALSE,dead,[63, 255]];
        ENDLOOP;
bptr ← 0;
```

```
lastval ← [63, 255];
END;

VMCleanup: PUBLIC PROCEDURE = BEGIN
i: FileIndex;
FOR i IN FileIndex DO
        IF FTable[i] # NIL THEN
                FPU.Close[FTable[i]];
        FTable[i]←NIL;
        ENDLOOP;
FSys.Logout[fs];
fs ← NIL;
END;

Open: PUBLIC PROCEDURE [name: STRING]
        RETURNS [f: FileIndex, len: LONG INTEGER] =
        BEGIN
        free: FileIndex;
        lp: LONG INTEGER;
        bp,ps: CARDINAL;
        free ← FindFree[name];
        -- We open the file in write mode to keep core from bitching if we later decide to write it.
        IF FTable[free] = NIL THEN
                FTable[free] ← FSys.Open[fs,name, FSys.OpenMode[write]
                        ! FSys.FileDoesNotExist = >CONTINUE];
        IF FTable[free] = NIL THEN RETURN[0,0];
        RWTable[free] ← FALSE;
        [lp, bp, ps] ← FPU.Measure[FTable[free]];
        RETURN[free, lp*ps + bp];
        END;

FindFree: PROCEDURE [name: STRING] RETURNS [free: FileIndex] =
        BEGIN
        i: FileIndex;
        free ← 63;
        FOR i IN [1..62] --fix if FileIndex ever changes
                DO
                IF FTable[i] = NIL THEN free←i
                ELSE IF ST.EquivalentString[name,FPU.Name[FTable[i]]] THEN BEGIN free ← i; RETURN END;
                ENDLOOP;
        IF free = 63 THEN P.RErr["Too many files open(>62)"L];
        END;

OpenRW: PUBLIC PROCEDURE [name: STRING]
        RETURNS [f: FileIndex, len: LONG INTEGER] =
        BEGIN
        free: FileIndex;
        lp: LONG INTEGER;
        bp,ps: CARDINAL;
        free ← FindFree[name];
        IF FTable[free] = NIL THEN
                FTable[free] ← FSys.Open[fs,name, FSys.OpenMode[create]
                        ! FSys.FileAlreadyExists = >RESUME];
        RWTable[free] ← TRUE;
        [lp, bp, ps] ← FPU.Measure[FTable[free]];
        RETURN[free, lp*ps + bp];
        END;

Close: PUBLIC PROCEDURE [f: FileIndex] =
        BEGIN i: CARDINAL;
        IF RWTable[f] THEN RETURN; -- must be closed by CloseRW
        IF FTable[f] # NIL THEN FPU.Close[FTable[f]];
        FOR i IN [0..nbuf) DO
                IF InP[i].fp.f = f THEN InP[i] ← [FALSE,dead,[63,255]];
                ENDLOOP;
        IF lastval.f = f THEN lastval ← [63,255];
        FTable[f] ← NIL;
        END;

CloseRW: PUBLIC PROCEDURE [f: FileIndex, size: LONG INTEGER] =
        BEGIN i: CARDINAL;
        IF FTable[f] = NIL THEN RETURN;
```

```
        FOR i IN [0..nbuf) DO
                IF InP[i].fp.f = f THEN
                        BEGIN
                        IF InP[i].mod THEN -- file must be 0
                                FPU.WritePage[
                                                FTable[f],
                                                InP[i].fp.p,
                                                BASE[StringArray[i]]];
                        InP[i] ← [FALSE,dead,[63,255]];
                        END
                        ENDLOOP;
        FPU.SetLength[FTable[f],
                Cardinal[size/bSize], Cardinal[size MOD bSize]];
        FPU.Close[FTable[f]];
        IF lastval.f = f THEN lastval ← [63,255];
        FTable[f] ← NIL;
        END;

GetPage: PROCEDURE[p: FilePage] RETURNS[CARDINAL] = BEGIN
-- return the buffer # with page p in it
i: CARDINAL;
IF lastval = p THEN RETURN[lastinx]; -- active was already set to alive
FOR i IN [0..nbuf) DO
        IF p = InP[i].fp THEN BEGIN
                InP[i].active ← alive;
                lastval ← p;
                lastinx ← i;
                RETURN[i];
                END;
        ENDLOOP;
-- not in core, must go get it
DO
CurMap[(bptr MOD 14) + 1] ← InlineDefs.BITAND[CurMap[(bptr MOD 14) + 1],107777B];
bptr ← (bptr + 1) MOD nbuf;
IF InP[bptr].active = alive THEN InP[bptr].active ← dead
ELSE EXIT;
ENDLOOP;
IF InP[bptr].mod THEN
        FPU.WritePage[
                        FTable[InP[bptr].fp.f],
                        InP[bptr].fp.p,
                        BASE[StringArray[bptr]]]; -- extends if necessary
FPU.ReadPage[FTable[p.f], p.p, BASE[StringArray[bptr]]
                        ! FPU.EndOfFile => CONTINUE];
-- If file page isn't there, who cares about content?
CurMap[(bptr MOD 14) + 1] ← InlineDefs.BITOR[CurMap[(bptr MOD 14) + 1],70000B];
InP[bptr] ← [FALSE,alive,p];
lastval←p;
lastinx ← bptr;
RETURN[bptr];
END;

GetString: PUBLIC PROCEDURE[st: LONG INTEGER, len: CARDINAL, f: FileIndex, s: STRING] = BEGIN
ii: LONG INTEGER;
i,b,p,p1,p2,n1,n2,k,m: CARDINAL;
IF len > s.maxlength THEN P.PBug["bad GetString"L];
s.length ← len;
ii ← st + len - 1;
p1 ← Cardinal[st/bSize];
p2 ← Cardinal[ii/bSize];
n1 ← Cardinal[st MOD bSize];
n2 ← Cardinal[ii MOD bSize];
b ← GetPage[[f, p1]];
k ← 0;
m ← IF p1 = p2 THEN n2 ELSE bSize - 1;
FOR i IN [n1..m] DO
        s[k] ← StringArray[b][i];
        k ← k + 1;
        ENDLOOP;
IF p1 = p2 THEN RETURN;
```

```
FOR p IN (p1 .. p2) DO
        b ← GetPage[[f, p]];
        FOR i IN [0..bSize) DO
                s[k] ← StringArray[b][i];
                k ← k + 1;
                ENDLOOP;
        ENDLOOP;
b ← GetPage[[f, p2]];
FOR i IN [0..n2] DO
        s[k] ← StringArray[b][i];
        k ← k + 1;
        ENDLOOP;
END;

SetString: PUBLIC PROCEDURE[st: LONG INTEGER, f: FileIndex, s: STRING] = BEGIN
ii: LONG INTEGER;
i,b,p,p1,p2,n1,n2,k,m: CARDINAL;
IF s.length = 0 THEN P.PBug["Attempt to store empty string"L];
IF ~RWTable[f] THEN P.PBug["Attempt to write read-only file"L];
ii ← st + s.length - 1;
p1 ← Cardinal[st/bSize];
p2 ← Cardinal[ii/bSize];
n1 ← Cardinal[st MOD bSize];
n2 ← Cardinal[ii MOD bSize];
b ← GetPage[[f, p1]];
k ← 0;
m ← IF p1 = p2 THEN n2 ELSE bSize - 1;
FOR i IN [n1..m] DO
        StringArray[b][i] ← s[k];
        k ← k + 1;
        ENDLOOP;
InP[b].mod ← TRUE;
IF p1 = p2 THEN RETURN;
FOR p IN (p1 .. p2) DO
        b ← GetPage[[f,p]];
        FOR i IN [0..bSize) DO
                StringArray[b][i] ← s[k];
                k ← k + 1;
                ENDLOOP;
        InP[b].mod ← TRUE;
        ENDLOOP;
b ← GetPage[[f, p2]];
FOR i IN [0..n2] DO
        StringArray[b][i] ← s[k];
        k ← k + 1;
        ENDLOOP;
InP[b].mod ← TRUE;
END;

END.
```

```
                                                DIRECTORY
                                                ovD: FROM "OverviewDefs",
                                                crD: FROM "CoreDefs",
                                                SystemDefs: FROM "SystemDefs",
                                                InlineDefs: FROM "InlineDefs",
                                                FilePageUseDefs: FROM "FilePageUseDefs",
                                                FileSystemDefs: FROM "FileSystemDefs",
                                                PLDefs: FROM "PLDefs",
                                                StringDefs: FROM "StringDefs";

AFiles: PROGRAM
    IMPORTS crD, SY: SystemDefs, ST: StringDefs, PLDefs
    EXPORTS FilePageUseDefs, FileSystemDefs
    SHARES crD
    = BEGIN
NN: TYPE = CARDINAL;

Transaction: TYPE =  POINTER TO TR;

TR: TYPE = RECORD[

    dmsu: crD.DMSUser,

    prefix: STRING];

File: TYPE = POINTER TO FH;

FH: TYPE = RECORD[ ufh: crD.UFileHandle,

                name: STRING];
FileHandle: TYPE = FilePageUseDefs.FileHandle;
FileSystem: TYPE = FileSystemDefs.FileSystem;

PageNumber: TYPE = FilePageUseDefs.PageNumber;

OpenMode: TYPE = FileSystemDefs.OpenMode;

BytesPerPage: NN = 512; -- bytes
ReadOnly: PUBLIC ERROR = CODE;
EndOfFile: PUBLIC ERROR = CODE;
FileAlreadyExists: PUBLIC SIGNAL = CODE;
FileDoesNotExist: PUBLIC SIGNAL = CODE;
IllegalFileName: PUBLIC ERROR = CODE;
DiskError: PUBLIC ERROR [erc: NN] = CODE;

Check: PROCEDURE  [erc: NN] =
BEGIN
SELECT erc FROM
ovD.ok => RETURN;
ovD.storageNotAvailable => PLDefs.PBug["Disk error- requested storage could not be obtained"L];
ovD.notAStamp => PLDefs.PBug["Disk error- notAStamp"L];
ovD.badFieldName => PLDefs.PBug["Disk error- in a message header"L];
ovD.badFieldBody => PLDefs.PBug["Disk error- in a message header"L];
ovD.badMailFile => PLDefs.PBug["Disk error- badMailFile"L];
ovD.tOCNotFound => PLDefs.PBug["Disk error- VirtualizeTOC couldn't find your file. (VirtualMgr)"L];
ovD.notATOCFile => PLDefs.PBug["Disk error- VirtualizeTOC didn't like your file. (VirtualMgr)"L];
ovD.tOCOverflow => PLDefs.PBug["Disk error- Too many msgs. from VirtualizeTOC, ExtendTOC."L];
ovD.cMTooBig => PLDefs.PBug["Disk error-  Attemp to extend CM beyond cMOCharMapTableSize."L];
ovD.cantAccessMailbox => PLDefs.PBug["Disk error- from MailFileOps."L];
ovD.ftpError => PLDefs.PBug["Disk error- from Operations"L];
ovD.notAMailFile => PLDefs.PBug["Disk error-  from MailFileOps."L];
ovD.badDMSUser => PLDefs.PBug["Disk error- Operations, user and password provided didn't work."L];
ovD.messageSyntaxError => PLDefs.PBug["Disk error- Operations, error inparsing message to be sent."L];
ovD.invalidRecipient => PLDefs.PBug["Disk error- Operations, couldn't lookup specified recipient."L];
ovD.mailboxBusy => PLDefs.PBug["Disk error- Operations, attempt to connect to mail server failed for the indicated reason."L];
ovD.cantConnect => PLDefs.PBug["Disk error- Operations, generic connection failure."L];
ovD.noMessagesMoved => PLDefs.PBug["Disk error- Operations, AppendMailToFileOperation moved no messages."L];
ovD.cancelCode => PLDefs.PBug["Disk error- Operations, generic cancelled by user."L];
ovD.diskError => PLDefs.RErr["Disk error- An I/O data error (Core)."L];
ovD.diskCorrupted => PLDefs.RErr["Disk error- Disk structure is logically inconsistent; run Scavenger."L];
ovD.diskFull => PLDefs.RErr["Disk error- Deleted some files and try again."L];
ovD.fileInUse => PLDefs.RErr["Disk error- Attempt to open a file that has already been opened (with another UFileHandle), from
```

```
OpenFile."L];
ovD.illegalFilename => PLDefs.RErr["Disk error- Either (1) bad character(s) in filename, or (2) name too long.  From OpenFile."L];
ovD.fileNotFound => PLDefs.RErr["Disk error- Opening a non-existing file for input only.  From OpenFile."L];
ovD.fileTooBig => PLDefs.RErr["Disk error- Attempt to extend a file beyond max allowed (from WritePages) or to open a file that is
already longer than that max (OpenFile)."L];
ENDCASE => PLDefs.PBug["Disk error- Mystery"L];
END;

Close: PUBLIC PROCEDURE[fh: FileHandle] =
    BEGIN
    f: File = LOOPHOLE[fh];
    erc: ovD.ErrorCode = crD.CloseFile[f.ufh];
    Check[erc];
    SY.FreeHeapString[f.name];
    SY.FreeHeapNode[f];
    END;

Delete: PUBLIC PROCEDURE  [fs: FileSystem, fileName: STRING] =
    BEGIN
    t: Transaction = LOOPHOLE[fs];
    f: crD.UFileHandle;
    erc: NN;
    [erc, f] ← crD.OpenFile[t.dmsu, [fileName], update];
    Check[erc];
    Check[crD.DeleteFile[f]];
    END;


Login: PUBLIC PROCEDURE  [type: CHARACTER, serverMachine, name, password: STRING]
                RETURNS [fs: FileSystem] = BEGIN
    t: Transaction = SY.AllocateHeapNode[SIZE[TR]];
    t↑ ← [[["Morris"], ["Jim"]], ""];
    fs ← LOOPHOLE[t];
    END;

Logout: PUBLIC PROCEDURE  [fs: FileSystem] =
    BEGIN
    SY.FreeHeapNode[fs];
    END;

Measure: PUBLIC PROCEDURE  [fh: FileHandle] RETURNS [lastPage,nextByteOnPage,pageSize:CARDINAL] = BEGIN
    f: File = LOOPHOLE[fh];
    RETURN[f.ufh.lastFilePage, f.ufh.byteFF, BytesPerPage];
    END;


Name: PUBLIC PROCEDURE[fh: FileHandle] RETURNS [STRING] =
    BEGIN
    RETURN [LOOPHOLE[fh, File].name];
    END;

Open: PUBLIC PROCEDURE
    [fs: FileSystem, fileName: STRING, mode: OpenMode]
    RETURNS [f: FileHandle] =
    BEGIN
    t: Transaction = LOOPHOLE[fs];
    fh: File ← SY.AllocateHeapNode[SIZE[FH]];
    erc: NN;
    fh.name ← SY.AllocateHeapString[fileName.length];
    ST.AppendString[fh.name, fileName];
    [erc, fh.ufh] ← crD.OpenFile[t.dmsu, [fileName], read];
    IF erc = ovD.fileNotFound THEN
                BEGIN
                IF mode = read THEN ERROR FileDoesNotExist;
                IF mode = write THEN SIGNAL FileDoesNotExist;
                [erc, fh.ufh] ← crD.OpenFile[t.dmsu, [fileName], update];
                END
    ELSE IF erc = ovD.ok AND mode # read THEN
                BEGIN
                Check[crD.CloseFile[fh.ufh]];
                IF mode = create THEN SIGNAL FileAlreadyExists;
                [erc, fh.ufh] ← crD.OpenFile[t.dmsu, [fileName], update];
```

```
            END;
    Check[erc];
    f ← LOOPHOLE[fh];
    END;


ReadPage: PUBLIC PROCEDURE  [fh: FileHandle, p: PageNumber, b: POINTER]  =
    BEGIN
    f: File  = LOOPHOLE[fh];
    erc: NN;
    IF p> = (IF f.ufh.byteFF = 0 THEN f.ufh.lastFilePage ELSE f.ufh.lastFilePage + 1) THEN
                        ERROR EndOfFile;
    [erc, ] ← crD.ReadPages[b, 512, p, f.ufh];
    Check[erc];
    END;

SetLength: PUBLIC PROCEDURE
    [fh: FileHandle, lastPage:PageNumber, nextByteOnPage:CARDINAL]  =  BEGIN
    f: File  = LOOPHOLE[fh];
    IF f.ufh.access = read THEN ERROR ReadOnly;
    IF lastPage>f.ufh.lastFilePage
            OR lastPage = f.ufh.lastFilePage AND nextByteOnPage>f.ufh.byteFF THEN -- lengthen file by writing junk
            Check[crD.WritePages[LOOPHOLE[0], 512*(lastPage-f.ufh.lastFilePage) + nextByteOnPage, f.ufh.lastFilePage, f.ufh]]
    ELSE -- shorten file
            Check[crD.UFileTruncate[lastPage, nextByteOnPage, f.ufh]];
    END;


WritePage: PUBLIC PROCEDURE  [fh: FileHandle, p: PageNumber, b: POINTER]  =
    BEGIN
    f: File  = LOOPHOLE[fh];
    IF f.ufh.access = read THEN ERROR ReadOnly;
    Check[crD.WritePages[b, 512, p, f.ufh]];
    END;


END.
```

```
                                                    DIRECTORY
                                                    PLDefs: FROM "PLDefs",
                                                    FilePageUseDefs: FROM "FilePageUseDefs",
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    StringDefs: FROM "StringDefs",
                                                    BTDefs: FROM "BTDefs",
                                                    VMDefs: FROM "VMDefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    BTreeDefs: FROM "BTreeDefs";

BT: PROGRAM IMPORTS B: BTreeDefs, P:PLDefs, VM: VMDefs, FileSystemDefs, ST: StringDefs, SY: SystemDefs
EXPORTS BTDefs = BEGIN
--
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
Stream: TYPE = PLDefs.Stream;
Register: TYPE = PLDefs.Register;
--
Fail,Nail: Node;

OBT: TYPE = POINTER TO OBTR;
OBTR: TYPE = RECORD[
        password: CARDINAL,
        tree: BTreeDefs.BTreeHandle,
        stringFile: VMDefs.FileIndex,
        btsin: LONG INTEGER];
Password: CARDINAL = 10067;
WildNode: TYPE = POINTER TO WildR;
WildR: TYPE = RECORD[ig: CARDINAL, o: OBT];

BTOpenRoutine: PROCEDURE[s:Symbol,input,n2: Node]
RETURNS[n: Node] = BEGIN
        rr: PLDefs.Register ← P.MRS[];
        st: STRING ← [100];
        o: OBT = SY.AllocateHeapNode[SIZE[OBTR]];
        file: FilePageUseDefs.FileHandle;
        new: BOOLEAN ← TRUE;
        stringFile: VMDefs.FileIndex;
        btsin: LONG INTEGER;
        wild: Node ← P.Alloc[[FAIL,,FAIL[]]];
        wn: WildNode = LOOPHOLE[wild];
        n ← Nail;
        P.R2[@n,@wild];
        o.password ← Password;
        P.MakeString[st, input];
        file ← FileSystemDefs.Open[VM.FS[], st, create !
        FileSystemDefs.FileAlreadyExists => 
                BEGIN new ← FALSE; RESUME END];
        o.tree ← B.CreateBTree[file, new];
        B.InitializeBTree[o.tree, FirstGE, Eq, TRUE];
        ST.AppendString[st, ".leaves"];
        [stringFile, btsin] ←
                VM.OpenRW[st ! FileSystemDefs.FileDoesNotExist => RESUME];
        o.stringFile ← stringFile;
        o.btsin ← btsin;
        wn.o ← o;
        n ← Nail;
        n ← P.Alloc[[LIST,,LIST[P.Alloc[[UNARY,,UNARY[P.Lookup["close"], wild]]], n]]];
        n ← P.Alloc[[LIST,,LIST[P.Alloc[[UNARY,,UNARY[P.Lookup["forget"], wild]]], n]]];
        n ← P.Alloc[[LIST,,LIST[P.Alloc[[UNARY,,UNARY[P.Lookup["next"], wild]]], n]]];
        n ← P.Alloc[[LIST,,LIST[P.Alloc[[UNARY,,UNARY[P.Lookup["store"], wild]]], n]]];
        n ← P.Alloc[[LIST,,LIST[P.Alloc[[UNARY,,UNARY[P.Lookup["fetch"], wild]]], n]]];
        P.RRS[rr];
        END;

ErSt: STRING = "illegal call on B-tree routine";

CloseRoutine: PROCEDURE[s:Symbol,input,n2: Node]
RETURNS[Node] = BEGIN
        w: WildNode = LOOPHOLE[n2];
```

```
        IF w.o.password # Password THEN P.RErr[ErSt];
        B.ReleaseBTree[w.o.tree];
        VM.CloseRW[w.o.stringFile, w.o.btsin];
        SY.FreeHeapNode[w.o];
        RETURN[Nail];
        END;

Eq: BTreeDefs.TestKeys =
        BEGIN
        RETURN[ST.EqualString[St[a],St[b]]];
        END;

FirstGE: BTreeDefs.TestKeys =
        BEGIN
        as:STRING = St[a];
        bs:STRING = St[b];
        pointer: CARDINAL←0;
        DO
                IF as.length = pointer THEN RETURN[bs.length = pointer];
                IF bs.length = pointer THEN RETURN[TRUE];
                IF as[pointer]<bs[pointer] THEN RETURN[FALSE];
                IF as[pointer]>bs[pointer] THEN RETURN[TRUE];
                pointer←pointer + 1;
                ENDLOOP;
        END;

FetchRoutine: PROCEDURE[s:Symbol,input,n2: Node] RETURNS[n: Node] = BEGIN
K: STRING ← [100];
V: RECORD[st: LONG INTEGER, len: CARDINAL];
w: WildNode = LOOPHOLE[n2];
IF w.o.password # Password THEN P.RErr[ErSt];
IF input.Type ~ = STR THEN P.RErr["fetch expects string"L];
IF P.Length[input]>100 THEN P.RErr["key to long (>100)"];
P.MakeString[K, input];
IF B.Lookup[w.o.tree,
        DESCRIPTOR[K,(K.length + 1)/2 + 2],
        DESCRIPTOR[@V,3]] = 177777B THEN RETURN[Fail];
IF V.st + V.len< = 177777B THEN
        n ← P.Alloc[[STR,
                [FALSE, TRUE, simp, FALSE, w.o.stringFile],
                STR[[simp[VM.Cardinal[V.st], V.len]]]]]
ELSE BEGIN s: STRING = SY.AllocateHeapString[V.len];
        VM.GetString[V.st, V.len, w.o.stringFile, s];
        n ← P.MakeSTR[s];
        SY.FreeHeapNode[s];
        END;
END;

ForgetRoutine: PROCEDURE[s: Symbol,input,n2: Node] RETURNS[Node] = BEGIN
K: STRING ← [100];
w: WildNode = LOOPHOLE[n2];
IF w.o.password # Password THEN P.RErr[ErSt];
IF input.Type ~ = STR THEN P.RErr["forget expects string"L];
IF P.Length[input]>100 THEN P.RErr["key to long (>100)"];
P.MakeString[K, input];
B.Delete[w.o.tree, DESCRIPTOR[K,(K.length + 1)/2 + 2]];
RETURN[Nail];
END;

NextRoutine: PROCEDURE[s:Symbol,input,n2: Node] RETURNS[Node] =
        BEGIN
        K: STRING ← [100];
        N: STRING ← [100];
        Grab: BTreeDefs.Call =
                BEGIN
                IF ST.EqualString[St[k], K] THEN RETURN[TRUE];
                ST.AppendString[N, St[k]];
                RETURN[FALSE];
                END;
```

```
        w: WildNode = LOOPHOLE[n2];
        IF w.o.password # Password THEN P.RErr[ErSt];
        IF input.Type ~ = STR THEN P.RErr["next expects string"L];
        IF P.Length[input]>100 THEN P.RErr["key to long (>100)"];
        P.MakeString[K, input];
        B.Enumerate[w.o.tree, DESCRIPTOR[K,(K.length + 1)/2 + 2], Grab];
        RETURN[IF N.length = 0 THEN Fail ELSE P.MakeSTR[N]];
        END;

StoreRoutine: PROCEDURE[s:Symbol,input,n2: Node] RETURNS[Node] = BEGIN
K: STRING ← [100];
V: RECORD[st: LONG INTEGER, len: CARDINAL];
st: STRING;
w: WildNode = LOOPHOLE[n2];
IF w.o.password # Password THEN P.RErr[ErSt];
IF ~(input.Type = LIST AND P.LengthList[input] = 2 AND input.left.Type = STR AND input.right.left.Type = STR ) THEN
P.RErr["store expects a list of two strings"L];
IF P.Length[input.left]>100 THEN P.RErr["key to long (>100)"];
P.MakeString[K, input.left];
st ← SY.AllocateHeapString[VM.Cardinal[P.Length[input.right.left]]];
P.MakeString[st, input.right.left];
VM.SetString[w.o.btsin, w.o.stringFile, st];
V.st ← w.o.btsin;
V.len ← st.length;
w.o.btsin ← w.o.btsin + st.length;
SY.FreeHeapString[st];
IF K.length MOD 2 = 1 THEN K[K.length] ← 0C;
B.Insert[w.o.tree, DESCRIPTOR[K, (K.length + 1)/2 + 2], DESCRIPTOR[@V, 3]];
RETURN[Nail];
END;

Setup: PUBLIC PROCEDURE =
        BEGIN
        [] ← P.Insert["store"L,UNARY,proc,StoreRoutine,PLDefs.Unbound];
        [] ← P.Insert["fetch"L,UNARY,proc,FetchRoutine,PLDefs.Unbound];
        [] ← P.Insert["close"L,UNARY,proc,CloseRoutine,PLDefs.Unbound];
        [] ← P.Insert["forget"L,UNARY,proc,ForgetRoutine,PLDefs.Unbound];
        [] ← P.Insert["next"L,UNARY,proc,NextRoutine,PLDefs.Unbound];
        [] ← P.Insert["btopen"L,ZARY,proc,BTOpenRoutine,PLDefs.Unbound];
        END;

St: PROCEDURE [x: BTreeDefs.Desc] RETURNS [STRING] =
        BEGIN
        RETURN[LOOPHOLE[BASE[x]]];
        END;


[Fail,,Nail] ← P.GetSpecialNodes[];
END.
```

```
--BTree.mesa  Last edit:  February 19, 1979  10:05 AM

DIRECTORY
InlineDefs: FROM "InlineDefs",
StorageDefs: FROM "StorageDefs",
FilePageUseDefs: FROM "FilePageUseDefs",
BTreeDefs: FROM "BTreeDefs",
BTree2Defs: FROM "BTree2Defs";
BTree:PROGRAM
 IMPORTS bV:BTree2Defs,bB:BTree2Defs,FilePageUseDefs,StorageDefs
 EXPORTS BTreeDefs = PUBLIC BEGIN
OPEN StorageDefs,FilePageUseDefs,BTree2Defs,BTreeDefs;
EntryTooSmall:SIGNAL = CODE;
cacheLimit:CARDINAL = 4;
CallTree:TYPE = PROCEDURE[p:Page,i:Index] RETURNS[BOOLEAN];
NN:TYPE = CARDINAL;
Page:TYPE = POINTER TO BTreePageRecord;
BTreePageRecord:TYPE = RECORD
   [b:Tree,pa:NN,ca:VArray,c:Desc,level:NN,
    cache:Page,age:NN,dirty,deleted,rekey:BOOLEAN];
Tree:TYPE = POINTER TO BTreeRecord;
BTreeRecord:TYPE = RECORD[version,logPageSizeInBytes,maxIS,time,
   rootPA,freePA,lastPA,pageS,depth,entries:NN,
   file:FileHandle,self:BTreePageRecord,
   IsFirstBigger,AreTheyEqual:TestKeys,
   dirtyCache,dirtyTree:BOOLEAN];
nullPA:NN = 177777B;--from coredefs?
Bits:ARRAY[0..16) OF CARDINAL = [1,2,4,10B,20B,40B,100B,200B,400B,
 1000B,2000B,4000B,10000B,20000B,40000B,100000B];

debug: BOOLEAN ← TRUE;

CreateBTree:PROCEDURE[file:FileHandle,new:BOOLEAN]
  RETURNS[BTreeHandle] =
 BEGIN temp:Tree; I:NN = 8;
 b:Tree←Allocate[SIZE[BTreeRecord]];
 b↑←[1,I,20,0,nullPA,0,1,Bits[I],177777B,0,,
    BTreePageRecord[b,1,,,1,NIL,0,TRUE,FALSE,FALSE],
    MyIsFirstBigger,MyAreTheyEqual,FALSE,FALSE];
 IF new THEN BEGIN b.file←file; BuildARoot[b,NIL]; END
  ELSE BEGIN temp←Allocate[b.pageS]; ReadPage[file,1-1,temp];
     b↑←temp↑; Free[temp]; b.file←file; b.dirtyTree←FALSE;
     b.IsFirstBigger←MyIsFirstBigger; b.AreTheyEqual←MyAreTheyEqual; END;
 RETURN[LOOPHOLE[b]];
 END;

InitializeBTree:PROCEDURE
 [bh:BTreeHandle,isFirstBigger,areTheyEqual:TestKeys,
 dirtyCache:BOOLEAN] = BEGIN
 b:Tree = LOOPHOLE[bh];
 IF (b.IsFirstBigger←isFirstBigger) = NullProc
  THEN b.IsFirstBigger←MyIsFirstBigger;
 IF (b.AreTheyEqual←areTheyEqual) = NullProc
  THEN b.AreTheyEqual←MyAreTheyEqual;
 b.dirtyCache←dirtyCache;
 END;

ReleaseBTree:PROCEDURE[bh:BTreeHandle] = BEGIN
 b:Tree = LOOPHOLE[bh];
 FlushCache[bh];
 Close[b.file]; Free[b];
 END;

FlushCache:PROCEDURE[bh:BTreeHandle] = BEGIN
 b:Tree = LOOPHOLE[bh]; p:Page;
 UNTIL (p←b.self.cache) = NIL
  DO b.self.cache←p.cache; WriteOutPage[p,TRUE]; ENDLOOP;
 IF b.dirtyTree THEN BEGIN b.dirtyTree ← FALSE; WritePage[b.file,0,b]; END;
 END;

AskDepth:PROCEDURE[bh:BTreeHandle] RETURNS[CARDINAL] =
 BEGIN b:Tree = LOOPHOLE[bh]; RETURN[b.entries]; END;
```

```
Insert:PROCEDURE[bh:BTreeHandle,key:Desc,value:Desc] = BEGIN
 b:Tree = LOOPHOLE[bh];
 eR:EntryR←[key,value];
 InsertC:CallTree =
   BEGIN InsertP[p,i,@eR]; b.entries←b.entries + 1; b.dirtyTree←TRUE; RETURN[FALSE]; END;
 IF LENGTH[key] = 0 THEN ERROR;
 StartEnumerate[bh,key,InsertC];
 END;

Lookup:PROCEDURE[bh:BTreeHandle,key,value:Desc]
    RETURNS[lengthValue:CARDINAL] = BEGIN
 LookupC:CallTree = BEGIN
  b:Tree = LOOPHOLE[bh];
  eR:EntryR←LookupP[p,i];
  lengthValue←LENGTH[eR.v];
  IF b.AreTheyEqual[key,eR.k]
   THEN Copy[from:eR.v,to:value]
   ELSE lengthValue←177777B;
  RETURN[FALSE];
  END;
 StartEnumerate[bh,key,LookupC];
 END;

Next:PROCEDURE[bh:BTreeHandle,e:Entry] = BEGIN
 seen:CARDINAL←0;
 LookupC:CallTree = BEGIN
  b:Tree = LOOPHOLE[bh];
  eR:EntryR;
  IF (seen←seen + 1) = 1 THEN RETURN[TRUE];
  eR←LookupP[p,i];
  IF LENGTH[eR.v]>LENGTH[e.v] OR LENGTH[eR.k]>LENGTH[e.k]
    THEN SIGNAL EntryTooSmall;
  Copy[from:eR.v,to:e.v];
  Copy[from:eR.k,to:e.k];
  e.k←DESCRIPTOR[BASE[e.k],LENGTH[eR.k]];
  e.v←DESCRIPTOR[BASE[e.v],LENGTH[eR.v]];
  RETURN[FALSE];
  END;
 StartEnumerate[bh,e.k,LookupC];
 END;

Delete:PROCEDURE[bh:BTreeHandle,key:Desc] = BEGIN
 DeleteC:CallTree = BEGIN
 b:Tree = LOOPHOLE[bh];
  IF b.AreTheyEqual[key,LookupP[p,i].k] THEN
   BEGIN
   DeleteP[p,i];
   IF i = 0 THEN p.rekey←TRUE;
   b.entries←b.entries-1;
   b.dirtyTree←TRUE;
   END;
  RETURN[FALSE];
  END;
 IF LENGTH[key] # 0 THEN StartEnumerate[bh,key,DeleteC];
 END;

Enumerate:PROCEDURE[bh:BTreeHandle,key:Desc,c:Call] = BEGIN
 b:Tree = LOOPHOLE[bh];
 seen:CARDINAL←0;
 EnumC:CallTree = BEGIN
  eR:EntryR←LookupP[p,i];
  seen←seen + 1;
  RETURN[IF seen = 1 AND ~b.AreTheyEqual[key,eR.k]
      THEN TRUE ELSE c[eR.k,eR.v]];
  END;
 StartEnumerate[bh,key,EnumC];
 END;

StartEnumerate:PROCEDURE[bh:BTreeHandle,key:Desc,c:CallTree] = BEGIN
 b:Tree = LOOPHOLE[bh];
 this:Page←SwapInPage[b,b.rootPA,b.depth];
```

```
 brother:Page←NIL;
 Silli:PROCEDURE RETURNS[VArray] = BEGIN
  RETURN[(brother←SwapInPage[b,nullPA,this.level]).ca]; END;
 []←EnumerateFromTree[this,key,c
    !bV.GetAnother = >RESUME[Silli[]];
     bB.GetAnother = >RESUME[Silli[]];
     bV.Merger = >RESUME[NIL,TRUE];
     bB.Merger = >RESUME[NIL,TRUE]
     ];
 IF brother # NIL THEN
   BEGIN brother.dirty←TRUE; BuildARoot[b,brother]; END;
 IF ItemsP[this] = 1  AND b.deptn # 0 THEN BEGIN
  b.rootPA←LookupPA[ihis,0]; this.deleted←TRUE; b.depth←b.depth-1; b.dirtyTree←TRUE;
  END;
 SwapOutPage[this];
 END;

EnumerateFromTree:PROCEDURE[this:Page,key:Desc,c:CallTree]
 RETURNS[go:BOOLEAN] =  BEGIN
 index:Index; son,brother,right:Page; onRight:BOOLEAN;
 end:CARDINAL←ItemsP[this];  low:NN = this.level-1; b:Tree = this.b;
 delta:NN;
 Silly:PROCEDURE RETURNS[VArray] = BEGIN
    delta←IF onRight←(end>index + 1) THEN 1 ELSE -1;
    IF index = 0 AND NOT onRight THEN RETURN[NIL];
    brother←SwapInPage[b,LookupPA[this,index + delta],low];
    (right←IF onRight THEN brother ELSE son).rekey←TRUE;
    RETURN[brother.ca];
    END;
 FOR index←FindIndexOnPage[this,key],index + 1
 UNTIL index> = end DO
 IF this.level = 0 THEN go←c[this,index]
 ELSE BEGIN
  brother←NIL;
  onRight←FALSE;
  delta←0;
  son←SwapInPage[b,LookupPA[this,index],low];
  go←EnumerateFromTree[son,key,c
   !bB.GetAnother = >
      RESUME[(brother←SwapInPage[b,nullPA,low]).ca];
    bV.GetAnother = >
      RESUME[(brother←SwapInPage[b,nullPA,low]).ca];
    bB.Merger = >RESUME[Silly[],onRight];
    bV.Merger = >RESUME[Silly[],onRight];
    bB.DeletePage = >BEGIN right.deleted←TRUE; RESUME; END;
    bV.DeletePage = >BEGIN right.deleted←TRUE; RESUME; END
   ];
  IF brother # NIL THEN BEGIN
    brother.dirty←TRUE;
    ChangeKeys[this,index + delta,brother,onRight];
    IF delta = 1 THEN ChangeKeys[this,index,son,TRUE];
    END
  ELSE IF son.rekey  =  TRUE THEN
       ChangeKeys[this,index,son,TRUE];
  SwapOutPage[brother];
  SwapOutPage[son];
  END;
 IF NOT go THEN EXIT;
 ENDLOOP;
 END;

ChangeKeys:PROCEDURE[this:Page,i:Index,son:Page,replace:BOOLEAN] =
 BEGIN eR:EntryR;
 this.dirty ← TRUE;
 IF son.deleted THEN DeleteP[this,i] ELSE
  IF son.rekey THEN BEGIN
    son.rekey←FALSE;
    eR←LookupP[son,0];
    eR.v←DESCRIPTOR[@son.pa,1];
    IF replace THEN bB.Replace[this.ca,i,@eR] ELSE InsertP[this,i,@eR];
    END;
 END;
```

```
FindIndexOnPage:PROCEDURE[p:Page,key:Desc] RETURNS[m:Index] = BEGIN
 Bigger:TestKeys←p.b.IsFirstBigger;
 h:Index; n:Index←ItemsP[p]; t:BOOLEAN; h←m←128;
 DO t←IF m> = n THEN FALSE ELSE Bigger[key,LookupP[p,m].k];
   IF (h←h/2) = 0 THEN EXIT ELSE m←IF t THEN m + h ELSE m-h;
   ENDLOOP;
 IF NOT t THEN m←m-1;
 END;

MyIsFirstBigger:PROCEDURE[a,b:Desc] RETURNS[BOOLEAN] = BEGIN
 i:CARDINAL; t:CARDINAL = MIN[LENGTH[b],LENGTH[a]]; g:INTEGER←0;
 FOR i IN [0..t) UNTIL (g←a[i]-b[i]) # 0 DO ENDLOOP;
 RETURN[g>0 OR (g = 0 AND LENGTH[a]> = LENGTH[b])];
 END;

MyAreTheyEqual,NullProc:PROCEDURE[a.b:Desc] RETURNS[BOOLEAN] = BEGIN
 i:CARDINAL; t:CARDINAL = LENGTH[a];
 IF LENGTH[b] # t THEN RETURN[FALSE];
 FOR i IN [0..t) DO IF a[i] # b[i] THEN RETURN[FALSE]; ENDLOOP;
 RETURN[TRUE];
 END;

Copy:PROCEDURE[from:Desc,to:Desc] = BEGIN i,I:CARDINAL;
 I←MIN[LENGTH[from],LENGTH[to]];
 FOR i IN [0..I) DO to[i]←from[i]; ENDLOOP;
 END;

SwapInPage:PROCEDURE[b:Tree,p:NN,I:CARDINAL]
 RETURNS[bp:Page] = BEGIN
 place:POINTER; back:Page; new:BOOLEAN←p = nullPA;
  reuse:BOOLEAN←new AND (b.freePA # 0);
 IF new THEN BEGIN
    p←IF reuse THEN b.freePA ELSE b.lastPA←b.lastPA + 1;
    b.dirtyTree←TRUE;
    END;
 FOR back←@b.self, bp UNTIL (bp←back.cache) = NIL DO
  IF bp.pa = p AND bp.b = b THEN BEGIN back.cache←bp.cache; EXIT; END;
--something new in line above--
  REPEAT FINISHED = >BEGIN
    place←Allocate[b.pageS];
    IF NOT new OR reuse THEN ReadPage[b.file,p-1,place];
    bp←Allocate[SIZE[BTreePageRecord]];
    bp.pa←p; bp.b←b; bp.ca←@bp.c; bp.dirty←FALSE;
    bp.c←DESCRIPTOR[place,b.pageS];
    END;
  ENDLOOP;
 bp.level←I; bp.age←b.time←b.time + 1;
 bp.rekey←new; bp.deleted←FALSE;
 IF reuse THEN b.freePA←bp.ca[0];
 IF new THEN BEGIN
  bp.dirty←FALSE;
  (IF I # 0 THEN bB.Initialize ELSE bV.Initialize)[bp.ca];
  END;
 IF debug THEN Check[bp];
 END;

SwapOutPage:PROCEDURE[p:Page] = BEGIN
 b:Tree = p.b; bp:Page; back:Page←@b.self; t:NN; a:NN = p.pa;
 IF p = NIL THEN RETURN;
 IF debug THEN Check[p];
 IF p.deleted
   THEN BEGIN p.ca[0]←b.freePA; b.freePA←a; b.dirtyTree←p.dirty←TRUE; END;
 IF (p.dirty AND ~b.dirtyCache) THEN WriteOutPage[p,FALSE];
 FOR t IN [0..cacheLimit) DO
    IF back.cache = NIL THEN BEGIN back.cache←p; p.cache←NIL; RETURN; END;
    back←back.cache;
    ENDLOOP;
 IF cacheLimit # 0 THEN
    FOR back←@b.self, bp UNTIL (bp←back.cache) = NIL DO
    t←b.time-bp.age; t←p.level + t/8;
    IF bp.level< = t THEN
```

```
      BEGIN back.cache←p; p.cache←bp.cache; p←bp; EXIT; END;
    ENDLOOP;
 WriteOutPage[p,TRUE];
 END;

WriteOutPage:PROCEDURE[p:Page,free:BOOLEAN] =
 BEGIN
 IF p.dirty THEN BEGIN
  WritePage[p.b.file,p.pa-1,BASE[p.c]];
  p.dirty←FALSE;
  END;
 IF free THEN BEGIN Free[BASE[p.c]]; Free[p]; END;
 END;

BuildARoot:PROCEDURE[b:Tree,brother:Page] = BEGIN eR:EntryR;
 root:Page←SwapInPage[b,nullPA,b.depth←b.depth + 1];
 IF brother # NIL THEN BEGIN
    eR←LookupP[brother,0];
    eR.v←DESCRIPTOR[@brother.pa,1];
    InsertP[root,177777B,@eR];
    SwapOutPage[brother];
    END;
 eR←[DESCRIPTOR[NIL,0],DESCRIPTOR[@b.rootPA,1]];
 InsertP[root,177777B,@eR];
 b.rootPA←root.pa;
 b.dirtyTree←TRUE;
 SwapOutPage[root];
 END;

LookupPA:PROCEDURE[p:Page,i:Index] RETURNS[NN] =
 BEGIN RETURN[LookupP[p,i].v[0]]; END;

InsertP:PROCEDURE[p:Page,i:Index,e:Entry] = BEGIN p.dirty←TRUE;
 (IF p.level = 0 THEN bV.Insert ELSE bB.Insert)[p.ca,i,e];
 END;

LookupP:PROCEDURE[p:Page,i:Index] RETURNS[EntryR] = BEGIN
 RETURN[(IF p.level = 0 THEN bV.Lookup ELSE bB.Lookup)[p.ca,i]];
 END;

DeleteP:PROCEDURE[p:Page,i:Index] = BEGIN p.dirty←TRUE;
 (IF p.level = 0 THEN bV.Delete ELSE bB.Delete)[p.ca,i];
 END;

ItemsP:PROCEDURE[p:Page] RETURNS[CARDINAL] = BEGIN
 RETURN[(IF p.level = 0 THEN bV.Items ELSE bB.Items)[p.ca]]; END;

Check:PROCEDURE[p:Page] = BEGIN
 b:Tree = p.b; i:NN; n:NN = ItemsP[p]; e:EntryR;
 thisIsPageOne:BOOLEAN = p = @b.self;
 SELECT thisIsPageOne FROM
  TRUE => IF p.pa # 1 THEN ERROR;
  ENDCASE => IF p.pa NOT IN [2..b.lastPA] THEN ERROR;
 IF p.ca # @p.c OR LENGTH[p.c] # b.pageS THEN ERROR;
 IF p.level>b.depth AND NOT p.deleted THEN ERROR;
 IF b.rootPA NOT IN [2..b.lastPA] AND ~(b.rootPA = nullPA AND p.level = b.depth) THEN ERROR;
 IF b.freePA>b.lastPA OR b.freePA = 1 THEN ERROR;
 --IF n> = 2 THEN IF b.IsFirstBigger[LookupP[p,0].k,
  --LookupP[p,n-1].k] THEN ERROR;
 --IF n> = 4 THEN IF b.AreTheyEqual[LookupP[p,n/2].k,
   --LookupP[p,n/2-1].k] THEN ERROR;
 IF p.level>0 THEN FOR i IN [0..n)
   DO
   e ← LookupP[p,i];
   IF LENGTH[e.v] # 1 OR e.v[0] NOT IN [2..b.lastPA] THEN ERROR;
   IF LENGTH[e.k] = 0 AND i>0 THEN ERROR;
   ENDLOOP;
  END;

 END.
```

```
DIRECTORY BTree2Defs : FROM "BTree2Defs";
BTree2: PROGRAM EXPORTS BTree2Defs = PUBLIC BEGIN
OPEN BTree2Defs;

NN:TYPE = CARDINAL;
MinusOne: CARDINAL ← 177777B;-- horrible kludge to evade range checker, see ReplaceX and Check (JHM, 5/8/79)
Desc:TYPE = DESCRIPTOR FOR ARRAY OF WORD;
RealJob:TYPE = {insert,replace,delete};
w,FixedOverhead:CARDINAL = 2;
OverheadPerItem:CARDINAL = 1;
nullER:EntryR←[DESCRIPTOR[NIL,0],DESCRIPTOR[NIL,0]];

debug: BOOLEAN ← TRUE;

Form:TYPE = RECORD[k:[0..2048),v:[0..32)];
Foo:TYPE = POINTER TO ARRAY [0..3) OF Form;

IndexOutOfBounds:SIGNAL = CODE;
GetAnother:SIGNAL RETURNS[VArray] = CODE;
ReKey:SIGNAL = CODE;
Merger:SIGNAL RETURNS [VArray,BOOLEAN] = CODE;
DeletePage:SIGNAL = CODE;

Initialize:PROCEDURE[v:VArray] = BEGIN i:CARDINAL;
 FOR i IN [0..LENGTH[v↑]) DO v[i]←0; ENDLOOP;
 v[1]←LOOPHOLE[Form[LENGTH[v↑],0]];
 END;

Items:PROCEDURE[v:VArray] RETURNS[NN] = BEGIN RETURN[v[0]]; END;

Lookup:PROCEDURE[this:VArray,item:Index] RETURNS[EntryR] = BEGIN
 foo:Foo; startK,endK,startV,lengthK:NN;
 [foo,,,startK,endK]←Unpack[this,item];
 IF item> = this[0] THEN SIGNAL IndexOutOfBounds;
 startV←startK + (lengthK←foo[item].v);
 RETURN[[DESCRIPTOR[@this[startK],lengthK],
      DESCRIPTOR[@this[startV],endK-startV]]];
 END;

Insert:PROCEDURE[this:VArray,item:Index,e:Entry] =
 BEGIN
 IF LENGTH[e.k]> = 32 OR LENGTH[e.k] + LENGTH[e.v]>80 THEN ERROR;
 ReplaceX[this,item,e,insert];
 END;

Delete:PROCEDURE[this:VArray,item:Index] =
 BEGIN ReplaceX[this,item,@nullER,delete]; END;

Replace:PROCEDURE[this:VArray,item:Index,e:Entry] =
 BEGIN ReplaceX[this,item,e,replace]; END;

ReplaceX:PROCEDURE[this:VArray,item:Index,e:Entry,r:RealJob] =
BEGIN
diddle:INTEGER = SELECT r FROM replace = >0,insert = >1,ENDCASE = >-1;
foo:Foo; other:VArray; shrinkage:INTEGER;
nextIndex,startLast,startItem,page,halfPage,sizeKey,sizeVal:NN;
sizeNew,sizeOld,start,extra,used,endItem,i:NN;
[foo,nextIndex,startLast,startItem,endItem]←Unpack[this,item];
page←foo[MinusOne].k;-- Horrible kludge (JHM)
halfPage ←page/2;
sizeNew←(sizeKey←LENGTH[e.k]) + (sizeVal←LENGTH[e.v]);
sizeOld←IF r = insert THEN 0 ELSE endItem-startItem;
shrinkage←sizeOld-sizeNew;
start←startItem + shrinkage;
extra←halfPage-w-shrinkage;
used←page-startLast-shrinkage + nextIndex-1 + OverheadPerItem + w;
IF debug THEN Check[this]; IF nextIndex<item + 1 THEN ERROR IndexOutOfBounds;
IF used + 4>page THEN BEGIN
   other←SIGNAL GetAnother;
   FOR i IN [0..nextIndex) DO
    IF foo[i].k<(IF i> = item THEN extra ELSE halfPage) + i
    THEN BEGIN Move[this,i,other];
```

```
      IF i>item THEN ReplaceX[this,item,e,r !Merger = >ERROR]
            ELSE ReplaceX[other,item-i,e,r !Merger = >ERROR];
      RETURN; END;
    ENDLOOP;
   ERROR;--oops
   END;
IF shrinkage # 0 THEN
  Slip[this,this,item + 1,item + 1 + diddle,startLast + shrinkage];
IF r # delete THEN BEGIN
  foo[item + diddle]←[start,sizeKey];
  Copy[@e.k[0],@this[start],sizeKey,0];
  Copy[@e.v[0],@this[start + sizeKey],sizeVal,0];
  END;
this[0]←nextIndex + diddle;
IF used<halfPage AND shrinkage>0 THEN Balance[this,used];
IF debug THEN Check[this];
END;


Balance:PROCEDURE[this:VArray,used:NN] = BEGIN
 oFoo:Foo; other:VArray; onRight:BOOLEAN;
 page,halfPage,nextIndex,thisEnd:NN;
 I,here,nextOther,otherEnd,i,carry:NN;
 page←LOOPHOLE[this[1],Form].k;
 halfPage←page/2;
 [other,onRight]←SIGNAL Merger;
 IF other = NIL THEN RETURN;
 [,nextIndex,thisEnd,,]←Unpack[this,0];
 [oFoo,nextOther,otherEnd,,]←Unpack[other,0];
 IF used + nextOther<otherEnd THEN BEGIN
   IF onRight THEN Move[other,0,this] ELSE Move[this,0,other];
   SIGNAL DeletePage; RETURN; END;
 carry←halfPage + w + (IF onRight THEN used ELSE 0);
 --magic to leave the page on the left just over half full
 FOR i IN [0..nextOther] UNTIL carry + i>(here←oFoo[i-1].k)
   DO ENDLOOP;
 IF carry + i< = here THEN ERROR;
 IF onRight THEN BEGIN
    other[0]←i;
    Move[other,0,this];
    other[0]←nextOther;
    Slip[other,other,i,0,otherEnd + page-here];
    other[0]←nextOther-i;
    END
 ELSE BEGIN
    I←nextOther-i;
    Slip[this,this,0,I,otherEnd + thisEnd-here];
    this[0]←0;
    Move[other,i,this];
    this[0]←I + nextIndex;
    END;
 END;

Unpack:PROCEDURE[v:VArray,item:Index] RETURNS[Foo,Index,NN,NN,NN] =
 BEGIN next:NN = v[0]; foo:Foo = LOOPHOLE[@v[w]];
 RETURN[foo,next,foo[next-1].k,foo[item].k,foo[item-1].k];
 END;

Move:PROCEDURE[from:VArray,at:Index,to:VArray] = BEGIN
 toNext,fromNext,fromEnd,atEnd,toEnd:NN;
 [,fromNext,fromEnd,,atEnd]←Unpack[from,at];
 [,toNext,toEnd,,]←Unpack[to,0];
 Slip[from,to,at,toNext,toEnd-atEnd + fromEnd];
 to[0]←toNext + fromNext-at;
 from[0]←at;
 END;

Slip:PROCEDURE[this,other:VArray,startIndex,newIndex,newData:Index] =
  BEGIN foo,toFoo:Foo;  endIndex,endData,startData,q:NN;
  [foo,endIndex,endData,,startData]←Unpack[this,startIndex];
  toFoo←LOOPHOLE[@other[w]];
  q←LOOPHOLE[Form[newData-endData,0]];
```

```
 Copy[@foo[startIndex],@toFoo[newIndex],endIndex-startIndex,q];
 Copy[@this[endData],@other[newData],startData-endData,0];
 END;

Copy:PROCEDURE[from,to:POINTER,len,a:NN] = BEGIN i:NN;
 IF len>256 THEN ERROR;
 IF LOOPHOLE[to,CARDINAL]>LOOPHOLE[from,CARDINAL]
 THEN FOR i DECREASING IN [0..len) DO (to + i)↑←(from + i)↑ + a; ENDLOOP
 ELSE FOR i           IN [0..len) DO (to + i)↑←(from + i)↑ + a; ENDLOOP;
 END;

Check:PROCEDURE [v:VArray] = BEGIN
 foo:Foo = LOOPHOLE[@v[w]];
 i: NN;
 IF v[0] > = LENGTH[v↑] THEN ERROR;
 IF foo[MinusOne].k # LENGTH[v↑] THEN ERROR; -- Horrible kludge (JHM)
 FOR i IN [0..v[0]) DO IF foo[i].k> = foo[i-1].k OR foo[i].v > foo[i-1].k - foo[i].k THEN ERROR; ENDLOOP;
 END;

END.
```

A Varray consists of:
1) the number of items (n) stored in the array.
2) a mumble, described below.
3) n packed descriptors for the items, each consisting of a pointer to the starting place and a length for the key.
4) some empty space perhaps
5) the items themselves, stored backward from the end of array

The length of the value part of the item can be deduced from the start of the preceeding item. The mumble is a start for a phony item so it all works without special end tests.

The Copy routines make the code smaller by 10%
The use of Entry makes the code smaller by 10%

A Copy is A[i]←B[i] + x for all i

A Slip moves entries startIndex and beyond from this array to newIndex and newData in array other, copying both index and data.

A Move appends entries startIndex and beyond from this array to the other array. move updates the entry count, slip doesn't.

```
-- modified by Wadler for lazy eval, 7/13/79
                                                    DIRECTORY
                                                    DispDefs: FROM "DispDefs",
                                                    PLDefs: FROM "PLDefs",
                                                    ImageDefs: FROM "ImageDefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    IODefs: FROM "IODefs",
                                                    MiscDefs: FROM "miscdefs",
                                                    AltoFileDefs: FROM "AltoFileDefs",
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    FilePageUseDefs: FROM "FilePageUseDefs",
                                                    StreamDefs: FROM "StreamDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    StringDefs: FROM "StringDefs",
                                                    SegmentDefs: FROM "SegmentDefs";

disp: PROGRAM IMPORTS IODefs, P:PLDefs EXPORTS DispDefs  = BEGIN
--
TokType: TYPE = PLDefs.TokType;
NodeType: TYPE = PLDefs.NodeType;
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
Register: TYPE = PLDefs.Register;
String: TYPE = PLDefs.String;
Stream: TYPE = PLDefs.Stream;
cdebug: BOOLEAN = PLDefs.cdebug;
CR: CHARACTER = IODefs.CR;
WP: TYPE = PROCEDURE[CHARACTER];
--
WC: WP;
charCount: CARDINAL ← 0;
DispCnt: INTEGER ← 0;
AskEnd: BOOLEAN ← TRUE;
ConfirmChar: CHARACTER;
LineLength: CARDINAL = 60;
MoreFlag: BOOLEAN;

Precedence: TYPE = [0..256);
Operator: ARRAY NodeType OF STRING;

CharsPerLine: CARDINAL = 72;
cIn: CARDINAL; -- current indentation of the line to be printed;
cPos: CARDINAL; -- current character position;
broken: [0..1]; -- 1 iff lines at current indentation should be put on multiple lines
Line: ARRAY [0..CharsPerLine) OF RECORD[break: BOOLEAN, indent: [0..128), c: CHARACTER];
-- Invariant:
        -- Line[0...cIn).c = SP
        -- For i in [cIn...cPos)Line[i].c contains characters
        -- If Line[i].break then we can start a new line there indented by Line[i].indent.  This indent value us always greater than cIn.
        -- cPos < CharsPerLine (i.e. we empty immediately when full)

ClearLine: PUBLIC PROCEDURE = BEGIN
END;

ClearScreen: PUBLIC PROCEDURE = BEGIN
charCount ← DispCnt ← 0;
END;

Confirm: PUBLIC PROCEDURE RETURNS [BOOLEAN] =
    BEGIN
     DO
        ConfirmChar ← IODefs.ReadChar[];
        SELECT ConfirmChar FROM
            'Y, 'y, IODefs.CR, '& => BEGIN IODefs.WriteString[" Yes."]; RETURN[TRUE]; END;
            'N, 'n, IODefs.DEL => BEGIN IODefs.WriteString[" No."]; RETURN[FALSE]; END;
            ENDCASE => IODefs.WriteString[" ?? "];
        ENDLOOP;
    END;

DispReset: PUBLIC PROCEDURE = BEGIN
DispCnt ← 0;
```

```
AskEnd ← TRUE;
MoreFlag ← TRUE;
WC ← MyWriteProcedure;
END;

DispSetup: PUBLIC PROCEDURE = BEGIN
DispReset[];
[] ← P.Insert["print"L,ZARY,proc,PrintRoutine,PLDefs.Unbound];
END;

MyWriteProcedure: PUBLIC PROCEDURE[c: CHARACTER] = BEGIN
b: BOOLEAN;
IODefs.WriteChar[c];
charCount ← IF c = IODefs.CR THEN 0 ELSE charCount + 1;
IF ~(charCount>CharsPerLine OR c = IODefs.CR) OR ~AskEnd THEN RETURN;
charCount ← 0;
DispCnt ← DispCnt + 1;
IF DispCnt >= PLDefs.NumLines - 5 THEN BEGIN
        IF MoreFlag THEN BEGIN
                DispCnt ← 0;
                IODefs.WriteChar[IODefs.CR];
                IODefs.WriteString["More? "];
                b ← Confirm[];
                IF ConfirmChar = '& THEN AskEnd ← FALSE;
                IF ~b THEN BEGIN
                        IODefs.WriteChar[IODefs.CR];
                        P.EndDisplay;
                        END;
                ClearLine[];
                END
        ELSE BEGIN
                IODefs.WriteChar[IODefs.CR];
                IODefs.WriteLine[" ... more ..."];
                P.EndDisplay;
                END;
        END;
END;

NL: PROCEDURE[i: CARDINAL] =
        BEGIN
        IF i<cIn + broken THEN -- flush out line
                BEGIN j: CARDINAL;
                FOR j IN [0..cPos)
                        DO
                        WC[Line[j].c];
                        Line[j]←[FALSE,,' ];
                        ENDLOOP;
                WC[CR];
                IF i<cIn THEN broken ← 0;
                cIn ← cPos ← i;
                END
        ELSE BEGIN
                Line[cPos].break ← TRUE;
                Line[cPos].indent ← i;
                END;
        END;

PC: PROCEDURE[c: CHARACTER] = BEGIN
        Line[cPos].c ← c;
        cPos ← cPos + 1;
        IF cPos = CharsPerLine THEN -- Time to flush line
                BEGIN
                i, j, k, l: CARDINAL;
                mindent: CARDINAL ← 77777B;
                FOR i IN [cIn + 1 ..CharsPerLine) DO
                        IF Line[i].break AND Line[i].indent<mindent THEN
                        mindent ← Line[i].indent;
                        ENDLOOP;
                FOR i ← 0, i + 1 UNTIL
                                i>cIn AND Line[i].break AND Line[i].indent = mindent OR i = CharsPerLine
                        DO
                        WC[Line[i].c]
```

```
                ENDLOOP;
            WC[CR];
            IF mindent = 77777B THEN cPos ← cIn
            ELSE BEGIN
                -- We're now at greater indentation than before
                --Add blanks
                FOR j IN [cIn..mindent) DO
                    Line[j] ← [FALSE,,' ]; ENDLOOP;
                j ← cIn ← mindent;
                -- shift characters to the left
                k ← i;
                    DO
                    Line[j] ← Line[k];
                    j ← j + 1;
                    k ← k + 1;
                    IF k = CharsPerLine THEN EXIT;
                    IF Line[k].break AND Line[k].indent = cIn THEN
                        BEGIN -- flush again
                        FOR I IN [0..j)
                            DO
                            WC[Line[I].c] ·
                            ENDLOOP;
                        WC[CR];
                        j ← cIn;
                        END;
                    ENDLOOP;
                cPos ← j;
                broken ← 1;
                END;
            FOR i IN [cPos..CharsPerLine) DO
                Line[i].break ← FALSE ENDLOOP;
            END;
    END;

Print: PUBLIC PROCEDURE[n: Node] = BEGIN
BEGIN ENABLE P.EndDisplay => CONTINUE;
i: CARDINAL;
    cIn ← cPos ← broken ← 0;
    FOR i IN [0..CharsPerLine) DO
        Line[i].break ← FALSE;
        ENDLOOP;
PrintExp[n,0,0,0,FALSE];broken ← 1;NL[0];
END;
END;

PrintExp: PROCEDURE[n: Node,in: CARDINAL,lp,rp: Precedence, de: BOOLEAN] = BEGIN --/z
-- This routine never calls Alloc (except in ZEval)
-- de is true when inside a closure, so Dont Eval lists
DO -- to eliminate tail recursion
IF n = NIL THEN PS["NIL(error)"]
ELSE SELECT n.Type FROM
FAIL, WILD, HOLE => PS[Operator[n.Type]];
STR =>
    BEGIN
    -- No Allocs done in here, by called procedures either!
    ns, ns1: PLDefs.StreamRecord;
    ns ← P.NewStream[n];
    ns1 ← ns;
    WHILE ns1.node ~ = NIL DO
        IF P.Item[@ns1] NOT IN ['0..'9] THEN GOTO NonNum;
        REPEAT
        NonNum =>
            BEGIN
            PC['"];
            WHILE ns.node ~ = NIL DO
                BEGIN
                c: CHARACTER = P.Item[@ns];
                SELECT c FROM
                '", '↑ => BEGIN PC['↑]; PC[c] END;
                36C => PS["↑036"];
                IODefs.DEL => PS["↑177"];
```

```
                                    IN [' ..IODefs.DEL) => PC[c];
                                    ENDCASE => BEGIN
                                            PC['↑]; PC[c + 100B];
                                            END;
                                    END;
                                    ENDLOOP;
                            PC['"];
                            END;
                    FINISHED =>
                            IF ns.node = NIL OR ns.node.str.length = 0 THEN
                                    BEGIN PC['"]; PC['"] END
                            ELSE WHILE ns.node ~ = NIL DO
                                    PC[P.Item[@ns]];
                                    ENDLOOP;
                    ENDLOOP;
            END;
CLOSURE =>
            BEGIN
            IF n.env = NIL THEN
                    PrintExp[n.exp,in,0,0,TRUE] --/z
            ELSE
                    BEGIN
                    PC['(];
                    in ← in + 1;
                    PrintExp[n.exp,in,0,0,TRUE]; --/z
                    NL[in];
                    PS[" where "];
                    n ← n.env;
                    UNTIL n = NIL DO
                            PS[n.val.unary.name];
                            PC[' = ];
                            PrintExp[n.val.uexp,in,0,0,de]; NL[in];
                            n ← n.next;
                            IF n # NIL THEN PS[" and "]
                            ENDLOOP;
                    PC[')];
                    END;
            END;
ID, ZARY, PFUNC => PS[n.name.name];
PATTERN => BEGIN
            PC['{]; PrintExp[n.pattern,in + 1,0,0,de]; PC['}]
            END;
LIST =>
            IF n.listhead = NIL THEN PS["[]"]
            ELSE
                    BEGIN
                    PC['[];
                    DO
                            PrintExp[(IF de THEN n.listhead ELSE P.ZEval[n.listhead]),in + 1,0,0,de]; --/z
                            n ← (IF de THEN n.listtail ELSE P.ZEval[n.listtail]); --/z
                            IF n.Type ~ = LIST THEN
                                    IF de THEN --/z
                                            BEGIN
                                            PC['.]; NL[in + 1];
                                            PrintExp[n,in + 1,0,0,de];
                                            EXIT;
                                            END
                                    ELSE
                                            BEGIN
                                            PS["TRASHED LIST TAIL!!!"];
                                            EXIT
                                            END;
                            IF n.listhead = NIL THEN EXIT;
                            PC[',]; NL[in + 1];
                            ENDLOOP;
                    PC[']];
                    END;
PFUNC1, UNARY =>
            IF rp>12 THEN
                    BEGIN
                    PC['(];
```

```
                PS[n.pfunc1.name];
                PC[' ];
                NL[in + 1];
                PrintExp[n.pexp,in + 1,12, 0,de];
                PC[')];
                END
        ELSE
                BEGIN
                PS[n.pfunc1.name];
                PC[' ];
                NL[in];
                n←n.pexp;
                lp ← 12;
                LOOP;
                END;
TILDE =>
        IF rp>12 THEN
                BEGIN
                PS["(~"];NL[in + 1];
                PrintExp[n.not,in + 1,12, 0,de];
                PC[')];
                END
        ELSE
                BEGIN
                PC['~];NL[in];
                n←n.not;
                lp ← 12;
                LOOP;
                END;
FCN =>
        BEGIN
        IF lp>16 OR 0<rp THEN
                BEGIN
                PC['(];
                PrintExp[n.left,in + 1,0,16,de];
                PS[": "];
                NL[in + 1];
                PrintExp[n.right,in + 1,0, 0,de];
                PC[')];
                END
        ELSE BEGIN
                PrintExp[n.left,in,lp,16,de];
                PS[": "];
                NL[in];
                n ← n.right;
                lp ← 0;
                LOOP;
                END;
        END;
CAT, CATL, MATCH, PALT, APPLY, MAPPLY, GOBBLE, ITER, ASS, PROG, PLUS, MINUS, SEQUENCE,EQUAL =>
        BEGIN
        lprec, rprec: Precedence;
        SELECT n.Type FROM
                PROG => BEGIN lprec←2; rprec ← 2 END;
                ASS => BEGIN lprec←16; rprec ← 4 END;
                PALT => BEGIN lprec←6; rprec ← 6 END;
                MATCH => BEGIN lprec←8; rprec ← 8 END;
                ENDCASE => BEGIN lprec←10; rprec ← 11 END;
        IF lp>lprec OR rprec<rp THEN
                BEGIN
                PC['(];
                PrintExp[n.left,in + 1,0,lprec,de];
                NL[in + 1];
                PS[Operator[n.Type]];
                IF n.Type = PALT OR n.Type = MATCH THEN in ← in + 1;
                PrintExp[n.right,in + 1,rprec, 0, de];
                PC[')];
                END
        ELSE BEGIN
                PrintExp[n.left,in,lp,lprec,de];
                NL[in];
                PS[Operator[n.Type]];
```

```
                    IF n.Type = PALT OR n.Type = MATCH THEN in ← in + 1;
                    n ← n.right;
                    lp ← rprec;
                    LOOP;
                    END;
            END;
OPT, DELETE, SEQOF, SEQOFC =>
            BEGIN
            PrintExp[n.opt,in,lp, 14, de];
            PS[Operator[n.Type]];
            END;
ENDCASE => BEGIN
            PS["TRASHED NODE!!! Type = "];
            PC[LOOPHOLE[n.Type]];
            END;
RETURN;
ENDLOOP;
END;

PrintRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
-- zary
-- print on terminal the entire incoming string, as is
-- must be string coming in
IF n1.Type ~ = STR THEN P.RErr["print requires a string as input"L];
PrintString[n1];
RETURN[n1];
END;

PrintString: PUBLIC PROCEDURE[s: Node] = BEGIN
            ns: PLDefs.StreamRecord;
            ns ← P.NewStream[s];
            WHILE ns.node ~ = NIL DO WC[P.Item[@ns]]; ENDLOOP;
END;

PS: PROCEDURE[s: STRING] = BEGIN
i: CARDINAL;
FOR i IN [0..s.length) DO PC[s[i]] ENDLOOP;
END;


SetWriteProcedure: PUBLIC PROCEDURE[p: WP] RETURNS [old: WP]
= BEGIN
            old ← WC;
            WC ← p;
            END;

ToggleAllPrint: PUBLIC PROCEDURE RETURNS[BOOLEAN] = BEGIN
RETURN[TRUE];
END;

ToggleMore: PUBLIC PROCEDURE RETURNS[BOOLEAN] = BEGIN
RETURN[MoreFlag ← ~MoreFlag];
END;

t: NodeType;
FOR t IN NodeType DO Operator[t] ← "???" ENDLOOP;
Operator[HOLE] ← "...";
Operator[CAT] ← " ";
Operator[FAIL] ← "fail";
Operator[WILD] ← "#";
Operator[CATL] ← ",,";
Operator[EQUAL] ← "=";
Operator[MATCH] ← ">";
Operator[PALT] ← "|";
Operator[APPLY] ← "/";
Operator[MAPPLY] ← "//";
Operator[GOBBLE] ← "///";
Operator[ITER] ← "%%";
Operator[FCN] ← ":";
Operator[ASS] ← "←";
Operator[PROG] ← ";";
```

```
Operator[SEQOF] ← "!";
Operator[SEQOFC] ← ",!";
Operator[DELETE] ← "*";
Operator[PLUS] ← " + ";
Operator[MINUS] ← "-";
Operator[TILDE] ← "~";
Operator[SEQUENCE] ← "--";
Operator[OPT] ← "?";

END.
```

```
                                        DIRECTORY
                                        EtherReportDefs: FROM "EtherReportDefs",
                                        NovaOps: FROM "NovaOps",
                                        TimeDefs: FROM "TimeDefs",
                                        InlineDefs: FROM "InlineDefs";

EtherReport: PROGRAM IMPORTS TimeDefs, InlineDefs, NovaOps
      EXPORTS EtherReportDefs =
BEGIN
-- Report events in common format to logging host. Copied from bcpl version

-- Last modified May 11, 1979  6:02 PM

-- This source file is intended to be self-contained not requiring
-- any declarations files - hence if Pup definitions change, this
-- file will need to be changed as well.

-- Standard Pup definitions:

Byte: TYPE = [0..256);
Socket: TYPE = EtherReportDefs.Socket;
Port: TYPE = EtherReportDefs.Port;
NetAddress: TYPE = RECORD[net, host: Byte];
PackedTime: TYPE = num InlineDefs.LongNumber;

EtherPup: TYPE = POINTER TO EtherPupRecord;
EtherPupRecord: TYPE = RECORD
          [
          eDest: Byte,
          eSrc: Byte,
          eType: WORD,          -- 1000B => Pup
          -- the Pup begins here, stuff above is Ethernet encapsulation
          length: WORD,         -- Bytes of pup contents
          transport: Byte,
          type: Byte,
          id: Socket,
          dPort: Port,
          sPort: Port,
          contents: ARRAY [0..300-pupOvWords) OF WORD,
          checksum: WORD        -- position varies, follows contents
          ];
pupOvBytes: CARDINAL = 22;      -- Pup header overhead
pupOvWords: CARDINAL = pupOvBytes/2;

ptEventReport: CARDINAL = 240B;     -- Event Report
ptEventReply: CARDINAL = 241B;      -- Event Report Reply

ptRouteRequest: CARDINAL = 200B;  -- Routing table info request
ptRouteReply: CARDINAL = 201B;

socketRouteInfo: Socket = [0,2];  -- Well known socket

RTC:POINTER TO CARDINAL = LOOPHOLE[430B];--Real Time Clock

-- Ether definitions
etherPup: CARDINAL = 1000B;    -- Ethernet type = Pup
etherOvWords: CARDINAL = 2;    -- Ether encapsulation overhead

ePLoc: POINTER TO CARDINAL = LOOPHOLE[600B];-- Post location
eBLoc: POINTER TO CARDINAL = LOOPHOLE[601B];-- Interrupts
eLLoc: POINTER TO CARDINAL = LOOPHOLE[603B];-- Load location
eICLoc: POINTER TO CARDINAL = LOOPHOLE[604B];-- Input count
eIPLoc: POINTER TO EtherPup = LOOPHOLE[605B];      -- Input
eOCLoc:POINTER TO CARDINAL = LOOPHOLE[606B];--Output count
eOPLoc: POINTER TO EtherPup = LOOPHOLE[607B];-- Output
eHLoc: POINTER TO Byte = LOOPHOLE[610B];-- Host address

etherReset: CARDINAL = 3;       -- interface SIO command bits
etherInput: CARDINAL = 2;
etherOutput: CARDINAL = 1;
```

```
Report: PUBLIC PROCEDURE[
        eventPort: Port,
        eventV: POINTER TO ARRAY[0..1) OF WORD,-- event
        eventVLength: CARDINAL
        ] RETURNS [EtherReportDefs.Result] =
BEGIN
retryCount: CARDINAL = 3;
timeOut: CARDINAL = 3*27;
try: CARDINAL;
r: INTEGER;
t: NetAddress = StartIO[0];   -- Find my serial number
buf: EtherPupRecord;    -- Packets live here
origTime: PackedTime ← LOOPHOLE[TimeDefs.CurrentDayTime[]];
id: Socket = [origTime.highbits, origTime.lowbits];
sPort: Port;
pdh: Byte ← eventPort.host;   -- Physical Destination Host

IF InlineDefs.BITAND[LOOPHOLE[t,UNSPECIFIED], 77777B] = 77777B THEN
        RETURN[NoEtherNet]; -- No Ethernet interface!
IF pdh = 0 THEN RETURN[HostIs0];    -- Don't broadcast!

eBLoc↑ ← 0;         -- No interrupts please
eHLoc↑ ← t.host;    -- Our host address
ePLoc↑ ← 0;
[] ← StartIO[etherReset];

-- Source port
sPort.net ← 0;
sPort.host ← eHLoc↑;
sPort.socket ← id;

FOR try IN [1..retryCount] DO
        -- If we don't know our network, get some routing info
        IF sPort.net = 0 THEN
            BEGIN
            routePort: Port ← [0, 0, socketRouteInfo];
            IF ~SendPup[@buf, routePort, routePort, id,
                ptRouteRequest, 0, 0] THEN
                                RETURN[RouteRequestFailed];
            r ← ReceivePup[@buf, routePort, routePort, [0,0],
                  ptRouteReply, timeOut];
            WHILE r > 0       -- Examine the routing info reply
              DO
              r ← r-2;
              sPort.net ← buf.dPort.net;
              IF eventPort.net # sPort.net
                    AND eventPort.net = LB[buf.contents[r]] THEN
                          pdh ← buf.eSrc;
                    ENDLOOP;
            END;

        -- Send the event report
        InlineDefs.COPY[to: @buf.contents, from: eventV,
                nwords: eventVLength];
        IF ~SendPup[@buf, sPort, eventPort, id, ptEventReport,
          eventVLength, pdh] THEN RETURN[SendFailed];

        -- don't wait for response
        -- r ← ReceivePup[@buf, eventPort, sPort, id, ptEventReply,
            -- timeOut];
        --IF r > = 0 THEN--RETURN[OK];
        ENDLOOP;
RETURN[NoReply]
END;

LB: PROCEDURE[x: WORD] RETURNS [[0..400B)] =
        BEGIN y: RECORD[l, r: [0..400B)] = LOOPHOLE[x];
        RETURN[y.l];
        END;

SendPup: PROCEDURE[buf:EtherPup,
                        sPort, dPort: Port,
```

```
                                id: Socket,
                                pupType, wordLength, dest:CARDINAL]
                        RETURNS[ok: BOOLEAN] =
-- Assume data already in buf.
-- Returns true IF packet sent OK.
BEGIN
i: CARDINAL;
buf.eSrc ← eHLoc↑;
buf.eDest ← dest;
buf.eType ← etherPup;   -- I'm a Pup in an Ether packet
buf.length ← wordLength*2 + pupOvBytes;
buf.transport ← 0;
buf.type ← pupType;
buf.id ← id;
buf.dPort ← dPort;
buf.sPort ← sPort;
buf.sPort.host ← eHLoc↑;
buf.contents[wordLength] ← 177777B;  -- No checksum

[] ← StartIO[etherReset];       -- Reset interface
eOCLoc↑ ← wordLength + pupOvWords + etherOvWords;
eOPLoc↑ ← buf;
eLLoc↑ ← 0;
ePLoc↑ ← 0;
[] ← StartIO[etherOutput];      -- Turn on transmitter
FOR i IN [1..30000] DO IF ePLoc↑ # 0 THEN EXIT ENDLOOP;
RETURN [ePLoc↑ # 0];
END;

ReceivePup: PROCEDURE[buf: EtherPup,
                                sPort, dPort: Port,
                                id: Socket,
                                pupType, timeOut :CARDINAL]
        RETURNS[INTEGER] =
-- Filter by sockets, id, and type.
-- Return length in words of data.
-- Return -1 IF no Pup received within specified time.
BEGIN
tim: CARDINAL = RTC↑ + timeOut; -- avoid short tim
status: CARDINAL;
  DO
  [] ← StartIO[etherReset];     -- Reset interface
  ePLoc↑ ← 0;
  eICLoc↑ ← 299;
  eIPLoc↑ ← buf;
  [] ← StartIO[etherInput];     -- Turn on receiver
  UNTIL ePLoc↑ # 0
        DO
        IF LOOPHOLE[RTC↑-tim, INTEGER]>0 THEN -- tricky
                BEGIN -- timeout
                [] ← StartIO[etherReset];
                RETURN[-1]
                END;
        ENDLOOP;
  status ← ePLoc↑;
  [] ← StartIO[etherReset];     -- Reset interface;

  IF status = 377B AND
   buf.eType = etherPup AND
   buf.eDest = eHLoc↑ AND  -- Discards broadcasts
   buf.sPort.socket = sPort.socket AND
   buf.dPort.socket = dPort.socket AND
   (id = [0,0] OR buf.id = id) AND
   buf.type = pupType THEN
    RETURN [(buf.length-22 + 1)/2];
  ENDLOOP;  -- Until good Pup received or timeOut;
END;

StartIO: PROCEDURE[ac0: UNSPECIFIED] RETURNS [UNSPECIFIED] =
BEGIN
code: ARRAY [0..2) OF CARDINAL ←
        [061004B, --SIO
```

```
        001400B]; --JMP 0,3
RETURN[NovaOps.NovaJSR[JSR, BASE[code], ac0]];
END;
END.
```