```
-- Poplar.Config - configuration module for Poplar
-- convention DispDefs,IODefs,DisplayDefs,other

Poplar: CONFIGURATION
IMPORTS SystemDefs, StringDefs, IODefs, StreamDefs, SegmentDefs, MiscDefs, DiskKDDefs, BFSDefs, DiskDefs,
DirectoryDefs, ImageDefs, AllocDefs, CoreSwapDefs, FrameDefs, LoadStateDefs, DisplayDefs, ProcessDefs, BootDefs,
LoaderBcdUtilDefs, FontDefs
CONTROL pl  = BEGIN
[strdefs: StreamDefs,FileSystemDefs,FilePageUseDefs,ImageDefs,RandomDefs, FontDefs, KeyDefs, TimeDefs] ←
Support[strdefs,SystemDefs,StringDefs,SegmentDefs,MiscDefs,DiskKDDefs,
BFSDefs,DiskDefs,DirectoryDefs,ImageDefs,AllocDefs,CoreSwapDefs,FrameDefs,LoadStateDefs, BootDefs,
LoaderBcdUtilDefs, ProcessDefs];
-- display stuff
[DLDefs1: DisplayDefs,SDefs1: StreamDefs] ← SD1: SystemDisplay[ImageDefs,SegmentDefs,strdefs];
[DLDefs2: DisplayDefs,SDefs2: StreamDefs] ← SystemDisplay[ImageDefs,SegmentDefs,strdefs];
stot1 ← strdefs PLUS SDefs1;
stot2 ← strdefs PLUS SDefs2;
[DLDefs1,ForgotDefs] ← DisplayControl[DirectoryDefs,DLDefs1,DLDefs2,FontDefs,ImageDefs, ProcessDefs, SegmentDefs,
stot1,StringDefs,SD1];
--
[DDefs1: DispDefs,IDefs1: IODefs] ← DispInstance[IDefs1,stot1,PLDefs,StringDefs];
[DDefs2: DispDefs,IDefs2: IODefs] ← DispInstance[IDefs2,stot2,PLDefs,StringDefs];
-- high level
pl[DDefs1,DDefs2,IDefs1,DLDefs1,DLDefs2,StringDefs,FileSystemDefs,ImageDefs,PLDefs,TimeDefs];
sup[DDefs1,stot1,SegmentDefs,MiscDefs,StringDefs,ImageDefs, PLDefs,FrameDefs,TimeDefs,RandomDefs,SystemDefs];
eval[PLDefs, DDefs1];
parse[DDefs1,PLDefs];
string[DDefs2,PLDefs,SystemDefs,MiscDefs,StringDefs,FileSystemDefs,FilePageUseDefs];
store[DDefs2,MiscDefs,PLDefs,StringDefs,SystemDefs];
route[DDefs1,DDefs2,IDefs2,PLDefs,FileSystemDefs,FilePageUseDefs, StringDefs,DirectoryDefs];
stat[DDefs2,PLDefs,SystemDefs];
VM;
END.
```

```
-- Support.Config - configuration module for Support routines for Poplar

Support: CONFIGURATION
IMPORTS StreamDefs, SystemDefs, StringDefs, SegmentDefs, MiscDefs, DiskKDDefs, BFSDefs, DiskDefs, DirectoryDefs,
ImageDefs, AllocDefs, CoreSwapDefs, FrameDefs, LoadStateDefs, BootDefs, LoaderBcdUtilDefs, ProcessDefs
EXPORTS StreamDefs, FileSystemDefs, FilePageUseDefs, ImageDefs, RandomDefs, FontDefs, KeyDefs, TimeDefs
 = BEGIN
Keyboard;
KeyStreams;
AlFont;
AltoFiles;
CheckPoint;
RandomProg;
ImageMaker;
StreamsA[];
StreamsB[];
TimeConvert;
END.
```

```
-- DispDefs.Mesa Last edited by Schmidt September 17, 1978  3:15 PM
                                                        DIRECTORY
                                                        PLDefs: FROM "PLDefs";


DispDefs: DEFINITIONS = BEGIN

-- defined in disp.mesa
DispSetup: PROCEDURE;
DispReset: PROCEDURE;
Print: 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.
```

```
-- PLDefs.Mesa Last edited by Schmidt September 23, 1978  9:06 PM
                                                    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};
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};
SType: TYPE = {zero,unk,string,proc,patproc};

Node: TYPE = POINTER TO NodeRecord;
pNode: TYPE = POINTER TO Node;
NodeRecord: TYPE = RECORD[
        Type: NodeType,
        Des: DesRecord,
        Var: SELECT OVERLAID NodeType FROM
        -- base
        ID = >          [name: Symbol],
        STR = >         [str: StringRecord],
        ZARY = >        [zary: Symbol],
        PFUNC = >       [pfunc: Symbol],
        FAIL = >        NULL,
        WILD = >        NULL,
        HOLE = >        NULL,
        -- inductive
        UNARY = >       [unary: Symbol, uexp: Node],
        PFUNC1 = >      [pfunc1: Symbol, pexp: Node],
        LIST = >        [listhead,listtail: Node],
        CAT,CATL = > [left,right: Node],
        MATCH = >       [div,patt: Node],                                   -- div = list, patt = pattern
        PALT = > [alt1,alt2: Node],                                 -- alt1, alt2 = lists
        APPLY, MAPPLY, GOBBLE, ITER = >      [object,target: Node],  -- object = list, target = pattern
        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],
        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 = {zero,simp,file,cat};
String: TYPE = POINTER TO StringRecord;
StringRecord: TYPE = RECORD[
        var: SELECT OVERLAID StringType FROM
        simp = > [start,length: CARDINAL],
        file = > [f: StringFile,of: OpenFile],
        cat = > [n1,n2: Node],
        ENDCASE
```

```
        ];
StringFile: TYPE = POINTER TO StringFileRecord;
StringFileRecord: TYPE = RECORD[
        pgno: CARDINAL,
        inx: CARDINAL,
        len: LONG INTEGER,
        bp: Node                                        -- back pointer for debugging
        ];
OpenFile: TYPE = POINTER TO OpenFileRecord;
OpenFileRecord: TYPE = RECORD[
        fs: FileSystemDefs.FileSystem,
        fh: FilePageUseDefs.FileHandle,
        lastpage: CARDINAL,
        pgsize: CARDINAL,
        lastchar: CARDINAL,
        filename: STRING
        ];
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
        ];
Register: TYPE = CARDINAL;
Stream: TYPE = POINTER TO StreamRecord;
StreamRecord: TYPE = RECORD[
        node: Node,
        posn: LONG INTEGER
        ];
--
NumLines: CARDINAL = 35;
NumPages: CARDINAL = 35;
sSize: CARDINAL = 500;
Unbound: UNSPECIFIED = ControlDefs.ControlLinkTag[unbound];
cdebug: BOOLEAN = TRUE;
--

PBug: SIGNAL[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] RETURNS[Node];
MapList: PROCEDURE[Node,PROCEDURE[Node] RETURNS[Node]] RETURNS[Node];
--
-- defined in sup.mesa
AmbushKeyStream: PROCEDURE;
Execute: PROCEDURE[Node];
WriteRem: PROCEDURE[Node];
Editor: PROCEDURE[STRING];
SupSetup: PROCEDURE;
SupReset: PROCEDURE;
DefaultName: PROCEDURE[STRING];
NonZero: PROCEDURE[POINTER, CARDINAL] RETURNS[BOOLEAN];
SetCursorAmt: PROCEDURE[CARDINAL,CARDINAL];
ResetCursor: PROCEDURE;

-- defined in route.mesa
```

```
RouteSetup: PROCEDURE;
WriteRoutine: PROCEDURE[Symbol,Node,Node] RETURNS[Node];
LengthList: PROCEDURE[Node] RETURNS[CARDINAL];

-- 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];
NewStringFile: PROCEDURE RETURNS[StringFile];
FreeStringFile: PROCEDURE[StringFile];
Sub: PROCEDURE[Node,LONG INTEGER] RETURNS[CHARACTER];
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];
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];
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];
SubString: PROCEDURE[Node,LONG INTEGER,LONG INTEGER] RETURNS [Node];
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];
```

```
-- defined in VM.Mesa
VMSetup: PROCEDURE;
VMCleanup: PROCEDURE;
GetSChar: PROCEDURE[CARDINAL] RETURNS[CHARACTER];
SetSChar: PROCEDURE[CARDINAL,CHARACTER];
GetString: PROCEDURE[CARDINAL,CARDINAL,STRING];
SetString: PROCEDURE[CARDINAL,STRING];
GetMaxStr: PROCEDURE RETURNS[CARDINAL];
END.
```

```
DispInstance: CONFIGURATION
IMPORTS IODefs, StreamDefs, PLDefs, StringDefs
EXPORTS DispDefs,IODefs
CONTROL disp  = BEGIN
StreamIO;
WF;
disp;
END.
```

```
-- disp.mesa last edited by schmidt, September 17, 1978  9:17 PM
                                                        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 DispDefs, IODefs, P:PLDefs, StreamDefs 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;
--
DispCnt: INTEGER ← 0;
AskEnd: BOOLEAN ← TRUE;
ConfirmChar: CHARACTER;
LineLength: CARDINAL  =  60;
MoreFlag: BOOLEAN;
AllPrint: BOOLEAN ← FALSE;

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;


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

ToggleAllPrint: PUBLIC PROCEDURE RETURNS[BOOLEAN]  =  BEGIN
RETURN[AllPrint ← ~AllPrint];
END;

MyWriteProcedure: PUBLIC PROCEDURE[c: CHARACTER]  =  BEGIN
b: BOOLEAN;
IODefs.WriteChar[c];
IF c ~=  IODefs.CR OR ~AskEnd THEN RETURN;
DispCnt ← DispCnt  +  1;
IF DispCnt  >  =  PLDefs.NumLines - 5 THEN BEGIN
        IF MoreFlag THEN BEGIN
                DispCnt ← 0;
                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
                DispCnt ← 0;
                IODefs.WriteLine[" ... more ..."];
                P.EndDisplay;
                END;
        END;
END;

PrintChar: PUBLIC PROCEDURE[c: CHARACTER] = BEGIN
OPEN DispDefs;
SELECT c FROM
' ,'",'↑ = > WF1["↑%c",c];
IN ['! .. IODefs.DEL] = > WF1["%c",c];
ENDCASE = > BEGIN
        IF c = 36C THEN WF0["↑036"]
        ELSE BEGIN
                c ← c + 100B;
                WF1["↑%c",c];
                END;
        END;
END;

Print: PUBLIC PROCEDURE[n: Node] = BEGIN
PrintList[n,n.Type = LIST OR n.Type = STR,0,n.Type = STR];
END;

PrintList: PROCEDURE[n: Node,p: BOOLEAN,in: CARDINAL,str: BOOLEAN] = BEGIN
-- This routine never calls Alloc
-- if p is TRUE then the lists and brackets have CR's between them
-- if str is true then print string without ...
OPEN DispDefs;
i: CARDINAL;
FOR i IN [0..in) DO
        WF0[" "];
        ENDLOOP;
IF n = NIL THEN WF0["[]"]
ELSE SELECT n.Type FROM
FAIL = > WF0["fail"];
WILD = > WF0["#"];
STR = >
        IF AllPrint THEN WF1["""%f""",n]
        ELSE IF str THEN WF1["""%a""",n]
        ELSE WF1["""%e""",n];
ID = > WF1["%s",n.name.name];
ZARY = > WF1["%s",n.zary.name];
PFUNC = > WF1["%s",n.pfunc.name];
HOLE = > WF0["..."];
PATTERN = > BEGIN WF0["{"]; PrintList[n.pattern,p,in,str]; WF0["}"]; END;
LIST = > BEGIN
        WF0["["];
        IF p THEN WF0["*n"];
        WHILE n ~= NIL AND n.Type = LIST DO
                PrintList[n.listhead,p,in,str];
                IF n.listtail ~= NIL THEN WF0[","];
                IF p AND n.listhead ~= NIL THEN WF0["*n"];
                n ← n.listtail;
                ENDLOOP;
        WF0["]"];
        END;
PFUNC1 = > BEGIN WF1["%s ",n.pfunc1.name]; PrintList[n.pexp,p,in,str] END;
UNARY = > BEGIN
        WF1["%s ",n.unary.name];
        PrintList[n.uexp,p,in,str];
        END;
CAT = > BEGIN PrintList[n.left,p,in,str]; WF0[" "]; PrintList[n.right,p,in,str]; END;
CATL = > BEGIN PrintList[n.left,p,in,str]; WF0[",,"]; PrintList[n.right,p,in,str]; END;
MATCH = > BEGIN PrintList[n.div,p,in,str]; WF0[" > "]; PrintList[n.patt,p,in,str]; END;
PALT = > BEGIN PrintList[n.alt1,p,in,str]; WF0["|"]; PrintList[n.alt2,p,in,str]; END;
APPLY = > BEGIN PrintList[n.object,p,in,str]; WF0["/"]; PrintList[n.target,p,in,str]; END;
MAPPLY = > BEGIN PrintList[n.object,p,in,str]; WF0["//"]; PrintList[n.target,p,in,str]; END;
GOBBLE = > BEGIN PrintList[n.object,p,in,str]; WF0["///"]; PrintList[n.target,p,in,str]; END;
ITER = > BEGIN PrintList[n.object,p,in,str]; WF0[" iter "]; PrintList[n.target,p,in,str]; END;
```

```
FCN = > BEGIN WF0["("]; PrintList[n.parms,p,in,str]; WF0[":"]; PrintList[n.fcn,p,in,str]; WF0[")"]; END;
ASS = > BEGIN PrintList[n.lhs,p,in,str]; WF0["←"]; PrintList[n.rhs,p,in,str]; END;
PROG = > BEGIN PrintList[n.prog1,p,in,str]; WF0[";*n"]; PrintList[n.prog2,p,in,str]; END;
PLUS = > BEGIN PrintList[n.arg1,p,in,str]; WF0[" + "]; PrintList[n.arg2,p,in,str]; END;
MINUS = > BEGIN PrintList[n.arg1,p,in,str]; WF0["-"]; PrintList[n.arg2,p,in,str]; END;
SEQUENCE = > BEGIN PrintList[n.from,p,in,str]; WF0[" -- "]; PrintList[n.to,p,in,str]; END;
SEQOF = > BEGIN PrintList[n.seqof,p,in,str]; WF0[" ! "]; END;
SEQOFC = > BEGIN PrintList[n.seqof,p,in,str]; WF0[" ,! "]; END;
OPT = > BEGIN PrintList[n.opt,p,in,str]; WF0[" ? "]; END;
DELETE = > BEGIN  PrintList[n.opt,p,in,str]; WF0[" ** "] END;
TILDE = > BEGIN  WF0[" ~ "]; PrintList[n.not,p,in,str] END;
ENDCASE = > P.PBug["Unknown type printlist"];
END;

PrintNode: PROCEDURE[d: UNSPECIFIED,form: STRING] = BEGIN
OPEN DispDefs;
t: Node ← d;
p: PROCEDURE[Node] RETURNS[CARDINAL] ← P.IndexNode;
IF cdebug THEN BEGIN
WF1["%-3d ",p[t]];
SELECT t.Type FROM
WILD = > WF0["WILD"];
HOLE = > WF0["HOLE"];
ID = > WF1["ID ""%s""",t.name.name];
STR = > WF1["STR ""%a""",t];
ZARY = > WF1["ZARY ""%s""",t.zary.name];
PFUNC = > WF1["PFUNC ""%s""",t.pfunc.name];
PATTERN = > WF1["PATTERN %d",p[t.pattern]];
PFUNC1 = > WF2["PFUNC1 ""%s"" %d",t.pfunc1.name,p[t.pexp]];
UNARY = > WF2["UNARY ""%s"" %d",t.unary.name,p[t.uexp]];
PROG = > WF2["PROG %d %d",p[t.prog1],p[t.prog2]];
CAT = > WF2["CAT %d %d",p[t.left],p[t.right]];
CATL = > WF2["CATL %d %d",p[t.left],p[t.right]];
LIST = > WF2["LIST %d %d",p[t.listhead],p[t.listtail]];
ASS = > WF2["ASS %d %d",p[t.lhs],p[t.rhs]];
FCN = > WF2["FCN %d %d",p[t.parms],p[t.fcn]];
MATCH = > WF2["MATCH %d %d",p[t.div],p[t.patt]];
PALT = > WF2["PALT %d %d",p[t.alt1],p[t.alt2]];
APPLY = > WF2["APPLY %d %d",p[t.object],p[t.target]];
MAPPLY = > WF2["MAPPLY %d %d",p[t.object],p[t.target]];
GOBBLE = > WF2["GOBBLE %d %d",p[t.object],p[t.target]];
ITER = > WF2["ITER %d %d",p[t.object],p[t.target]];
PLUS = > WF2["PLUS %d %d",p[t.arg1],p[t.arg2]];
MINUS = > WF2["MINUS %d %d",p[t.arg1],p[t.arg2]];
SEQUENCE = > WF2["SEQUENCE %d %d",p[t.from],p[t.to]];
SEQOF = > WF1["SEQOF %d",p[t.seqof]];
SEQOFC = > WF1["SEQOFC %d",p[t.seqof]];
OPT = > WF1["OPT %d",p[t.opt]];
DELETE = > WF1["DELETE %d",p[t.delete]];
TILDE = > WF1["TILDE %d",p[t.not]];
ENDCASE = > P.PBug["Unknown Type Printnode"];
END;
END;


PrintPLSTR: PROCEDURE[d: UNSPECIFIED,form: STRING] = BEGIN
n: Node ← d;
ns: PLDefs.StreamRecord;
rr: Register ← P.MRS[];
i: LONG INTEGER;
m: CARDINAL;
IF n.Type ~= STR THEN P.PBug["Bad print str"];
i ← P.Length[n];
IF i < LineLength THEN DispDefs.WF1["%a",n]
ELSE BEGIN
        ns ← P.NewStream[n];
        P.R[@ns.node];
        m ← (LineLength - 5)/2;
        THROUGH [1..m] DO
                DispDefs.WF1["%c",P.Item[@ns]];
                ENDLOOP;
```

```
        i ← i - (2*m);
        DispDefs.WF0[" ..... "];
        P.SkipStream[@ns,i];
        WHILE ns.node ~= NIL DO
                DispDefs.WF1["%c",P.Item[@ns]];
                ENDLOOP;
        END;
P.RRS[rr];
END;

PrintSTR: PROCEDURE[d: UNSPECIFIED,form: STRING] = BEGIN
n: Node ← d;
ns: PLDefs.StreamRecord;
rr: Register ← P.MRS[];
IF n.Type ~= STR THEN P.PBug["Bad print str"];
ns ← P.NewStream[n];
P.R[@ns.node];
WHILE ns.node ~= NIL DO
        DispDefs.WF1["%c",P.Item[@ns]];
        ENDLOOP;
P.RRS[rr];
END;

PrintUniqSTR: PROCEDURE[d: UNSPECIFIED,form: STRING] = BEGIN
n: Node ← d;
ns: PLDefs.StreamRecord;
rr: Register ← P.MRS[];
IF n.Type ~= STR THEN P.PBug["Bad print str"];
ns ← P.NewStream[n];
P.R[@ns.node];
WHILE ns.node ~= NIL DO
        PrintChar[P.Item[@ns]];
        ENDLOOP;
P.RRS[rr];
END;

ClearLine: PUBLIC PROCEDURE = BEGIN
StreamDefs.ClearCurrentLine[StreamDefs.GetDefaultDisplayStream[]];
END;

ClearScreen: PUBLIC PROCEDURE = BEGIN
i: CARDINAL;
FOR i IN [0 .. PLDefs.NumLines) DO
        IODefs.WriteChar[IODefs.CR];
        ENDLOOP;
END;

DispSetup: PUBLIC PROCEDURE = BEGIN
DispReset[];
END;

DispReset: PUBLIC PROCEDURE = BEGIN
DispCnt ← 0;
AskEnd ← TRUE;
MoreFlag ← TRUE;
AllPrint ← FALSE;
END;



[] ← DispDefs.SetWriteProcedure[MyWriteProcedure];
DispDefs.SetCode['a,PrintSTR];
DispDefs.SetCode['e,PrintPLSTR];
DispDefs.SetCode['f,PrintUniqSTR];
DispDefs.SetCode['n,PrintNode];
END.
```

```
-- eval.mesa last edited by schmidt, September 17, 1978  9:34 PM
                                                    DIRECTORY
                                                    PLDefs: FROM "PLDefs",
                                                    WFDefs: FROM "WFDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    IODefs: FROM "IODefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    StringDefs: FROM "StringDefs";

eval: PROGRAM IMPORTS P: PLDefs, DispDefs EXPORTS PLDefs = BEGIN
--
TokType: TYPE = PLDefs.TokType;
NodeType: TYPE = PLDefs.NodeType;
Node: TYPE = PLDefs.Node;
Symbol: TYPE = PLDefs.Symbol;
Register: TYPE = PLDefs.Register;
--
Fail,MTSt: Node;

Apply: PROCEDURE[val, func: Node] RETURNS [ret: Node] = BEGIN
        -- It is not necessary to register the vals in the bvStack, because the are all pointed to by val.
        rr : Register = P.MRS[];
        bvStack: ARRAY [0..30) OF RECORD[bv: PLDefs.Symbol, val: Node];
        bvp: CARDINAL ← 0;
        StackBvs: PROCEDURE[bv, vl: Node] =
                BEGIN
                i, j: CARDINAL;
                stb: PROCEDURE[x: Node] =
                        BEGIN
                        IF bvp + i > = 30 THEN P.RErr["Parameter stack exeeded"];
                        IF x.Type # ID THEN P.RErr["Malformed Parameter List"];
                        bvStack[bvp + i].bv ← x.name;
                        i ← i + 1;
                        END;
                stv: PROCEDURE[x: Node] =
                        BEGIN
                        IF j = i THEN P.RErr["Too many parameters"];
                        bvStack[bvp + j].val ← x;
                        j ← j + 1;
                        END;
                IF bv.Type = ID THEN BEGIN i ← 0; stb[bv]; j ← 0; stv[vl]; bvp ← bvp + 1 END
                ELSE
                        BEGIN
                        i ← 0;
                        Map[bv, stb];
                        j ← 0;
                        Map[vl, stv];
                        IF i > j THEN P.RErr["Too few parameters"];
                        bvp ← bvp + i;
                        END;
                END;

        Subst: PROCEDURE[n: Node] RETURNS [ret: Node] =
                BEGIN
                rr: Register ← P.MRS[];
                i, j: Node ← NIL;
                k: CARDINAL;
                oldBvp: CARDINAL;
                P.R2[@i, @j];
                CK;
                IF n = NIL THEN ret ← n ELSE
                IF n.Des.e THEN ret ← n ELSE
                SELECT n.Type FROM
                        ID = > BEGIN
                                FOR k DECREASING IN [0..bvp)
                                        DO
                                        IF n.name = bvStack[k].bv THEN
                                                BEGIN ret ← bvStack[k].val;
                                                EXIT;
                                                END;
                                        REPEAT
```

```
                                FINISHED => ret ← n;
                                ENDLOOP;
                        END;
                STR, FAIL, PFUNC, ZARY, WILD, HOLE => ret ← n;
                LIST, CAT, MATCH, PALT, APPLY, MAPPLY, PROG, PLUS, MINUS, CATL, SEQUENCE, GOBBLE,
                ITER => BEGIN -- really a big loophole here!
                        i ← Subst[n.listhead];
                        j ← Subst[n.listtail];
                        ret ← IF i = n.listhead AND j = n.listtail THEN n ELSE P.Alloc[[n.Type, ,LIST[i, j]]];
                        END;
                FCN => BEGIN -- check bound vars
                                oldBvp ← bvp;
                                StackBvs[n.parms, n.parms];
                                i ← Subst[n.fcn];
                                bvp ← oldBvp;
                                ret ← IF i = n.fcn THEN n ELSE P.Alloc[[FCN, ,FCN[n.parms, i]]];
                                END;
                UNARY, PFUNC1 =>
                                BEGIN
                                i ← Subst[n.uexp];
                                ret ← IF i = n.uexp THEN n ELSE P.Alloc[[n.Type, ,UNARY[n.unary,i]]];
                                END;
                ASS => BEGIN
                                i ← Subst[n.rhs];
                                ret ← IF i = n.rhs THEN n ELSE P.Alloc[[n.Type, ,ASS[n.lhs, i]]];
                                END;
                SEQOF, OPT, SEQOFC, DELETE, PATTERN, TILDE =>
                                BEGIN
                                i ← Subst[n.seqof];
                                ret ← IF i = n.seqof THEN n ELSE P.Alloc[[n.Type, ,SEQOF[i]]];
                                END;
                ENDCASE => P.PBug["Unknown type"];
        P.RRS[rr];
        END;
CK;
ret ← NIL;
P.R[@ret];
IF func.Type = ZARY THEN BEGIN
        s: Symbol ← func.zary;
        ret ← s.proc[s,val,NIL];
        END
ELSE IF func.Type = UNARY THEN BEGIN
        s: Symbol ← func.unary;
        ret ← s.proc[s,val,func.uexp];
        END
ELSE IF func.Type = FCN THEN
        BEGIN
        StackBvs[func.parms, val];
        ret ← Subst[func.fcn];
        ret ← Eval[ret];
        END
ELSE IF func.Type = PATTERN AND val.Type = STR THEN
        BEGIN
        ret ← Match[val, func.pattern];
        ret ← Eval[ret];
        END
ELSE IF func.Type = LIST THEN ret ← Subscript[func, val]
ELSE P.RErr["Bad application"];
P.RRS[rr];
END;

Binary: PROCEDURE[vi, vj: Node, p: PROCEDURE[Node, Node] RETURNS [Node]] RETURNS[Node] = BEGIN
C1: PROCEDURE [j: Node] RETURNS [Node] =
        BEGIN
        RETURN [p[vi, j]];
        END;
C2: PROCEDURE [i: Node] RETURNS [Node] =
        BEGIN
        RETURN [p[i, vj]];
        END;
IF vi.Type = STR AND P.Empty[vi] THEN RETURN[vj];
IF vj.Type = STR AND P.Empty[vj] THEN RETURN[vi];
```

```
IF vi = NIL AND vj.Type # LIST THEN RETURN[NIL];
IF vj = NIL AND vi.Type # LIST THEN RETURN[NIL];
IF vi.Type = STR AND vj.Type = STR THEN RETURN[p[vi, vj]];
IF vi.Type = STR AND vj.Type = LIST THEN RETURN[MapList[vj, C1]];
IF vi.Type = LIST AND vj.Type = STR THEN RETURN[MapList[vi, C2]];
IF vi.Type = LIST AND vj.Type = LIST THEN
        BEGIN
        rr: Register ← P.MRS[];
        ans: Node ← NIL;
        pans: POINTER TO Node ← @ans;
        P.R[@ans];
        UNTIL vi = NIL AND vj = NIL
                DO
                IF vi = NIL OR vj = NIL THEN P.RErr["List lengths differ"];
                IF vi.Type # LIST OR vj.Type # LIST  THEN P.PBug["Malformed List"];
                pans↑ ← P.Alloc[[LIST, ,LIST[p[vi.listhead, vj.listhead], NIL]]];
                pans ← @pans.listtail;
                vi ← vi.listtail;
                vj ← vj.listtail;
                ENDLOOP;
        P.RRS[rr];
        RETURN[ans];
        END;
P.RErr["Illegal binary operation"];
END;

ConcL: PROCEDURE[vi, vj: Node] RETURNS [Node] =
        BEGIN
        IF vj = NIL THEN RETURN[vi];
        BEGIN
        rr: Register ← P.MRS[];
        ans: Node ← NIL;
        pans: POINTER TO Node ← @ans;
        P.R[@ans];
        UNTIL vi = NIL
                DO
                IF vi.Type # LIST THEN P.PBug["Malformed List"];
                pans↑ ← P.Alloc[[LIST, ,LIST[vi.listhead, NIL]]];
                pans ← @pans.listtail;
                vi ← vi.listtail; -- someone is pointing at first vi, so don't need to register
                ENDLOOP;
        pans↑ ← vj;
        P.RRS[rr];
        RETURN[ans];
        END;
        END;

Eval: PUBLIC PROCEDURE[n: Node] RETURNS[ret: Node] = BEGIN
rr: Register ← P.MRS[];
IF n.Des.e THEN ret ← n ELSE
BEGIN ENABLE BEGIN
                P.RErr = > BEGIN
                        DispDefs.WF1["Error: %s, in:*n ",est];
                        DispDefs.Print[n];
                        DispDefs.WF0["*nBigger context?"];
                        IF ~DispDefs.Confirm[] THEN BEGIN
                                DispDefs.WF0["*n"];
                                P.EndDisplay;
                                END;
                        DispDefs.WF0["*n"];
                        END;
                        END;
i, j: Node ← NIL;
DE: PROCEDURE =
        BEGIN
        i ← Eval[n.arg1];
        j ← Eval[n.arg2]
        END;
ret ← NIL;
CK;
P.R3[@ret, @i, @j];
IF n # NIL THEN SELECT n.Type FROM
```

```
HOLE, PFUNC, ZARY, FCN,FAIL, STR, WILD, PATTERN = > ret ← n;
ID = > ret ← n.name.val;
UNARY, PFUNC1 = >
        ret ← P.Alloc[[n.Type,,UNARY[n.unary, Eval[n.uexp]]]];
DELETE, SEQOF, SEQOFC, OPT = > BEGIN
        ret ← P.Alloc[[n.Type,,DELETE[Eval[n.delete]]]];
        END;
PLUS,MINUS = > BEGIN
                F: PROCEDURE[i,j: Node] RETURNS[Node] =
                        BEGIN
                        a: LONG INTEGER = P.MakeInteger[i];
                        b: LONG INTEGER = P.MakeInteger[j];
                        RETURN[P.MakeNUM[SELECT n.Type FROM
                                        PLUS = > a+b,
                                        MINUS = > a-b,
                                        ENDCASE = > 0]];
                        END;
        DE;
        ret ← Binary[i, j, F];
        END;
SEQUENCE = > BEGIN
                DE;
                BEGIN
                from: LONG INTEGER = P.MakeInteger[i];
                to: LONG INTEGER ← P.MakeInteger[j];
                inc: LONG INTEGER = IF from > to THEN 1 ELSE -1;
                ret ← P.Alloc[[LIST,,LIST[P.MakeNUM[to], NIL]]];
                UNTIL from = to
                        DO
                        to ← to + inc;
                        ret ← P.Alloc[[LIST,,LIST[P.MakeNUM[to], ret]]];
                        ENDLOOP;
                END;
        END;
APPLY = > BEGIN
                DE;
                i ← Apply[i, j]; j ← NIL;
                ret ← P.FixRep[i];
                END;
MAPPLY = > BEGIN
                Map: PROCEDURE[x: Node] RETURNS [Node] =
                        BEGIN RETURN[Apply[x, j]] END;
                DE;
                IF i # NIL AND i.Type # LIST THEN P.RErr["Non-list before //"];
                i ← MapList[i, Map]; j← NIL;
                ret ← P.FixRep[i];
                END;
GOBBLE = > BEGIN
                GMap: PROCEDURE[x: Node] =
                        BEGIN i ← P.Alloc[[LIST,,LIST[ret, P.Alloc[[LIST,,LIST[x, NIL]]]]]];
                        ret ← Apply[i, j]
                        END;
                DE;
                IF i = NIL THEN ret ← NIL
                ELSE IF i.Type # LIST THEN P.RErr["Non-list before ///"]
                ELSE BEGIN
                        ret ← i.listhead;
                        Map[i.listtail, GMap];
                        END;
                i ← j ← NIL;
                ret ← P.FixRep[ret];
                END;
ITER = > BEGIN
                DE;
                ret ← Fail;
                UNTIL i = Fail
                        DO
                        ret ← i;
                        i ← Apply[i, j];
                        ENDLOOP;
                END;
ASS = > BEGIN
```

```
        s: Symbol = n.lhs.name;
        IF n.lhs.Type ~= ID THEN P.RErr["lhs not ID"];
        ret ← Eval[n.rhs];
        IF ret#Fail THEN s.val ← ret;
        END;
TILDE = >
        ret ← IF Eval[n.not]=Fail THEN MTSt ELSE Fail;
PROG = > BEGIN
        DE;
        ret ← j;
        END;
MATCH = > ret ← IF Eval[n.div] = Fail THEN Fail ELSE Eval[n.patt];
PALT = > BEGIN
        ret ← Eval[n.alt1];
        IF ret=Fail THEN ret ← Eval[n.alt2];
        END;
CAT, LIST, CATL = > BEGIN
                DE;
                ret ← IF n.Type=CATL THEN ConcL[i, j]
                ELSE IF n.Type=LIST THEN
                        (IF i=n.left AND j=n.left THEN n ELSE P.Alloc[[LIST,,LIST[i, j]]])
                ELSE Binary[i, j, P.StringConcat];
                END;
ENDCASE = > P.PBug["Unknown type"];
ret.Des.e ← TRUE;
END;
P.RRS[rr];
END;

Map: PUBLIC PROCEDURE[l: Node, p: PROCEDURE[Node]] =
        BEGIN
        FOR l ← l, l.listtail UNTIL l=NIL
                DO IF l.Type#LIST THEN P.PBug["Malformed List"];
        p[l.listhead]; ENDLOOP;
        END;

MapList: PUBLIC PROCEDURE[l: Node, p: PROCEDURE[Node] RETURNS [Node]] RETURNS [ans: Node] =
        BEGIN
        rr: Register ← P.MRS[];
        pans: POINTER TO Node ← @ans;
        tans: Node ← NIL;
        ans ← NIL;
        P.R[@ans];
        UNTIL l=NIL
                DO
                IF l.Type#LIST THEN P.PBug["Malformed List"];
                tans ← p[l.listhead];
                IF tans#Fail THEN
                        BEGIN
                        pans↑ ← P.Alloc[[LIST, ,LIST[tans, NIL]]];
                        pans ← @pans.listtail;
                        END;
                l ← l.listtail;
                ENDLOOP;
        P.RRS[rr];
        END;

Match: PROCEDURE[subject, pattern: Node] RETURNS [struc: Node] = BEGIN
rr: Register = P.MRS[];
s: PLDefs.StreamRecord;
HolePassed: SIGNAL [holeString: Node] = CODE;
M2: PROCEDURE[p: Node] RETURNS [ans: Node] =
        -- The state of s is an implict input and output of M2
        BEGIN
        CK;
        IF p=NIL THEN ans ← NIL ELSE
        BEGIN
        rr: Register ← P.MRS[];
        ans ← NIL;
        P.R[@ans];
                DO
```

```
        IF p.Type = PATTERN THEN p ← p.pattern
        ELSE IF p.Type = ID THEN p ← p.name.val
        ELSE EXIT;
        ENDLOOP;
SELECT p.Type FROM
        PFUNC, PFUNC1 = > BEGIN
                sym: Symbol = p.pfunc1;
                ans ← sym.patproc[sym, @s, p.pexp];
                ans.Des.e ← TRUE;
                END;
        MATCH, APPLY, MAPPLY, GOBBLE, ITER = > BEGIN
                        ans ← M2[p.div];
                        IF ans # Fail THEN
                                BEGIN
                                ans ← P.Alloc[[p.Type,,MATCH[ans,p.patt]]];
                                IF ans.div.Des.e THEN ans ← Eval[ans];
                                END;
                        END;
        TILDE = > BEGIN
                        s1: PLDefs.StreamRecord ← s;
                        P.R[@s1.node];
                        ans ← M2[p.not];
                        s ← s1;
                        ans ← IF ans = Fail THEN MTSt ELSE Fail;
                        END;
        DELETE = > BEGIN
                        ans ← M2[p.delete];
                        IF ans # Fail THEN ans ← MTSt;
                        END;
        STR = > BEGIN
                        s1: PLDefs.StreamRecord ← s;
                        l1, i: LONG INTEGER ← 0;
                        l1 ← P.Length[p];
                        P.R[@s1.node];
                        UNTIL i = l1
                                DO
                                IF s.node = NIL OR P.Item[@s] # P.Sub[p, i] THEN
                                        BEGIN ans ← Fail; EXIT END;
                                i ← i + 1;
                                REPEAT
                                FINISHED = > 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
                                ans ← P.SubStringStream[@s,0,1];
                                ans.Des.e ← TRUE;
                                [] ← 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
                        holeNode, struc2: Node ← NIL;
                        P.R2[@holeNode, @struc2];
                        ans ← M2[p.left !
                                HolePassed = > BEGIN
                                        holeNode ← holeString; RESUME END];
                        IF ans # Fail THEN
                                BEGIN
                                IF holeNode = NIL THEN struc2 ← M2[p.right]
                                ELSE BEGIN -- unanchored match allowed
                                        i: CARDINAL ← 0;
                                        s1, s2: PLDefs.StreamRecord ← s;
                                        tNode: Node;
                                        P.R[@s1.node];
                                                DO
```

```
                                        struc2 ← M2[p.right];
                                        IF struc2 # Fail THEN
                                                BEGIN
                                                tNode ← P.SubStringStream[@s1,0,i];
                                                holeNode↑ ← tNode↑;
                                                tNode↑ ← MTSt↑;
                                                ans ← Eval[ans];
                                                EXIT;
                                                END;
                                        IF s2.node = NIL THEN EXIT;
                                        [] ← P.Item[@s2];
                                        s ← s2;
                                        i ← i + 1;
                                        ENDLOOP;
                                END;
                        IF struc2 = Fail THEN ans ← Fail
                        ELSE BEGIN
                                ans ← P.Alloc[[p.Type,,CAT[ans,struc2]]];
                                IF struc2.Des.e THEN ans ← Eval[ans];
                                END;
                        END;
                END;

SEQOF = > BEGIN
                pans: POINTER TO Node ← @ans;
                tans: Node;
                s1: PLDefs.StreamRecord;
                P.R[@s1.node];
                ans ← Fail;
                        DO
                        s1 ← s;
                        tans ← M2[p.seqof !
                                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 = Fail THEN BEGIN s ← s1; EXIT END;
                        IF pans↑ = Fail THEN pans↑ ← tans
                        ELSE pans↑ ← Binary[pans↑, tans, P.StringConcat];
                        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.R[@s1.node];
                ans ← Fail;
                        DO
                        s1 ← s;
                        tans ← M2[p.seqof !
                                HolePassed = > BEGIN holeString.Des.e ← TRUE; RESUME END];
                        IF tans = Fail THEN BEGIN s ← s1; EXIT END;
                        pans↑ ← P.Alloc[[LIST,,LIST[tans, NIL]]];
                        pans.Des.e ← TRUE;
                        pans ← @pans.listtail;
                        ENDLOOP;
                END;

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

PALT = > BEGIN
                s1: PLDefs.StreamRecord ← s;
                P.R[@s1.node];
                ans ← M2[p.alt1];
```

```
                                        IF ans = Fail THEN BEGIN s ← s1; ans ← M2[p.alt2] END;
                                        END;
                        FAIL = > ans ← Fail;
                        ENDCASE = > P.RErr["Unknown pattern type"];
                P.RRS[rr];
                END;
                END;
holeNode, tNode: Node ← NIL;
s ← P.NewStream[subject];
P.R2[@holeNode, @s.node];
struc ← M2[pattern ! HolePassed = > BEGIN holeNode ← holeString; RESUME END];
IF holeNode # NIL THEN
        BEGIN
        tNode ← P.ConvertStream[@s];
        holeNode↑ ← tNode↑;
        tNode↑ ← MTSt↑;  -- to keep garbage collector from seeing same file node twice
        END
ELSE IF s.node # NIL THEN struc ← Fail;
P.RRS[rr];
END;

Subscript: PROCEDURE[list, sub: Node] RETURNS [ret: Node] =
        BEGIN
        i: LONG INTEGER ← P.MakeInteger[sub];
        neg: BOOLEAN ← FALSE;
        IF i = 0 THEN P.RErr["Zero subscript"];
        IF i < 0 THEN BEGIN i ← 1-i; neg← TRUE END;
        IF list = NIL THEN P.RErr["Subscripting empty list"];
        UNTIL i = 1
                DO
                IF list = NIL THEN P.RErr["Subscript too big"];
                IF list.Type # LIST THEN P.PBug["Malformed list"];
                list ← list.listtail;
                i ← i-1;
                ENDLOOP;
        ret ← IF neg THEN list ELSE list.listhead;
        END;

CK: PROCEDURE = BEGIN
-- blank top key on Alto-I
-- doesn't work right on Alto-II
-- = 0 means key depressed
k: POINTER ← LOOPHOLE[177034B];
IF InlineDefs.BITAND[(k + 1)↑,100000B] = 0 THEN P.Interrupt;
END;



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

```
-- parse.mesa last edited by schmidt, September 22, 1978  10:01 PM
                                                    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: Node;
savinput: PLDefs.StreamRecord;

-- dist: prog EOF
-- prog: stmt { SEP prog } | stmt SEP
-- stmt: exp { ASS exp }
-- exp: factor {DIV exp | MAPPLY exp | GOBBLE exp | ITER exp}
-- factor: term {PALT factor | RARR factor | COLON prog | PLUS factor | MINUS factor | CATL factor | SEQUENCE factor |
term}
-- term:  UNARY term | PFUNC1 term | MINUS term | TILDE term | primary {SEQOF | SEQOFC | OPT | DELETE} +
-- primary: STR | ID | ZARY | PFUNC | SCREEN | WILD | FAIL | HOLE | LB RB | LB stmt RB | LC stmt RC
--       | LP stmt RP
--
-- Convention: peektok is the first token for each of the routines, e.g.
--   peektok = STR for the Base

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"];
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 { }"];
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 ← Exp[];
P.R[@i];
IF PeekTok = ASS THEN BEGIN
        [] ← GetTok[];
        i ← P.Alloc[[ASS,,ASS[i,Exp[]]]];
        END;
P.RRS[rr];
RETURN[i];
END;

-- this parses left-assoc instead of right assoc.
Exp: PROCEDURE RETURNS[Node] = BEGIN
v: TokType;
i,j: Node ← NIL;
rr: Register ← P.MRS[];
P.R2[@i,@j];
IF PeekTok = DIV OR PeekTok = MAPPLY OR PeekTok = GOBBLE OR PeekTok = ITER THEN i ← currn
ELSE i ← Factor[];
WHILE PeekTok = DIV OR PeekTok = MAPPLY OR PeekTok = GOBBLE OR PeekTok = ITER DO
        v ← GetTok[];
        j ← Factor[];
        IF v = DIV THEN i ← P.Alloc[[APPLY,,APPLY[i,j]]]
        ELSE IF v = MAPPLY THEN i ← P.Alloc[[MAPPLY,,MAPPLY[i,j]]]
        ELSE IF v = ITER THEN i ← P.Alloc[[ITER,,ITER[i,j]]]
        ELSE i ← P.Alloc[[GOBBLE,,GOBBLE[i,j]]];
        ENDLOOP;
P.RRS[rr];
RETURN[i];
END;

Factor: PROCEDURE RETURNS[i: Node] = BEGIN
loop: BOOLEAN ← TRUE;
rr: Register ← P.MRS[];
p: PLDefs.TokType ← PeekTok;
i ← IF p = PALT OR p = RARR OR p = COLON OR p = PLUS OR p = CATL OR p = SEQUENCE THEN currn ELSE Term[];
P.R[@i];
WHILE loop DO
        loop ← TRUE;
        SELECT PeekTok FROM
        PALT = > BEGIN
                [] ← GetTok[];
                i ← P.Alloc[[PALT,,PALT[i,Term[]]]];
                END;
        RARR = > BEGIN
                [] ← GetTok[];
                i ← P.Alloc[[MATCH,,MATCH[i,Term[]]]];
                END;
        COLON = > BEGIN
                [] ← GetTok[];
                i ← P.Alloc[[FCN,,FCN[i,Prog[]]]];
                END;
        PLUS = > BEGIN
                [] ← GetTok[];
                i ← P.Alloc[[PLUS,,PLUS[i,Term[]]]];
                END;
        MINUS = > BEGIN
                [] ← GetTok[];
                i ← P.Alloc[[MINUS,,MINUS[i,Term[]]]];
                END;
        CATL = > BEGIN
```

```
                [] ← GetTok[];
                i ← P.Alloc[[CATL,,CATL[i,Term[]]]];
                END;
        SEQUENCE = > BEGIN
                [] ← GetTok[];
                i ← P.Alloc[[SEQUENCE,,SEQUENCE[i,Term[]]]];
                END;
        ENDCASE = > BEGIN
                -- check to see if this is a cat
                -- the list below must be kept up to date.
                -- it is those things not in First[factor]
                p ← PeekTok;
                IF p ~= SEP AND p ~= ASS AND p ~= DIV AND p ~= MAPPLY AND p ~= GOBBLE AND p
                ~= ITER AND p ~= RB AND p ~= RC AND p ~= RP AND p ~= EOF AND p ~= COMMA
                THEN i ← P.Alloc[[CAT,,CAT[i,Term[]]]]
                ELSE loop ← FALSE;
                END;
        ENDLOOP;
P.RRS[rr];
RETURN[i];
END;

Term: PROCEDURE RETURNS[i:Node] = BEGIN
s: Symbol;
rr: Register;
IF PeekTok = UNARY THEN BEGIN
        [] ← GetTok[];
        s ← TokVal;
        RETURN[P.Alloc[[UNARY,,UNARY[s,Term[]]]]];
        END;
IF PeekTok = PFUNC1 THEN BEGIN
        [] ← GetTok[];
        s ← TokVal;
        RETURN[P.Alloc[[PFUNC1,,PFUNC1[s,Term[]]]]];
        END;
IF PeekTok = TILDE THEN BEGIN
        [] ← GetTok[];
        RETURN[P.Alloc[[TILDE,,TILDE[Term[]]]]];
        END;
IF PeekTok = MINUS THEN BEGIN
        [] ← GetTok[];
        i ← P.MakeNUM[0];
        rr ← P.MRS[];
        P.R[@i];
        i ← P.Alloc[[MINUS,,MINUS[i,Term[]]]];
        P.RRS[rr];
        RETURN;
        END;
i ← Primary[];
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;

Primary: PROCEDURE RETURNS[t: Node] = BEGIN
pans: POINTER TO Node;
```

```
rr: Register;
SELECT PeekTok FROM
STR = >     BEGIN [] ← GetTok[]; RETURN[TokVal]; END;
ID = >      BEGIN [] ← GetTok[]; RETURN[P.Alloc[[ID,,ID[TokVal]]]]; 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
        [] ← GetTok[];
        IF PeekTok = RB THEN BEGIN
                [] ← GetTok[];
                RETURN[NIL];
                END;
        t ← P.Alloc[[LIST,,LIST[Prog[],NIL]]];
        rr ← P.MRS[];
        P.R[@t];
        pans ← @t.listtail;
        WHILE PeekTok = COMMA DO
                [] ← GetTok[];
                pans↑ ← P.Alloc[[LIST,,LIST[Prog[],NIL]]];
                pans ← @pans.listtail;
                ENDLOOP;
        IF PeekTok ~= RB THEN P.SErr["Parser expected ']'"];
        [] ← GetTok[];
        P.RRS[rr];
        RETURN;
        END;
LP = >      BEGIN       -- used solely for parenthesization
        [] ← GetTok[];
        t ← Prog[];
        IF PeekTok ~= RP THEN P.SErr["Parser expected ')'"];
        [] ← GetTok[];
        RETURN[t];
        END;
LC = >      BEGIN
        [] ← GetTok[];
        t ← Prog[];
        IF PeekTok ~= RC THEN P.SErr["Parser expected '}'"];
        []← GetTok[];
        RETURN[P.Alloc[[PATTERN,,PATTERN[t]]]];
        END;
ENDCASE;
P.SErr["Parser did not recognize primary statement"];
END;

GetTok: PROCEDURE RETURNS [TokType] = BEGIN
wk: STRING ← [100];
i: CARDINAL;
sym: Symbol;
got: BOOLEAN ← FALSE;
c: CHARACTER;
loop: BOOLEAN ← TRUE;
uc,num: 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;
        '", IN ['0..'9] = > BEGIN
                num ← nchar IN ['0..'9];
                wk[0] ← nchar;
                i ← IF num THEN 1 ELSE 0;
                got ← num;
                WHILE i < wk.maxlength DO
```

```
                wk[i] ← GetNChar[];
                IF wk[i] = '" OR wk[i] = 0C THEN EXIT;
                IF num AND wk[i] ~IN ['0..'9] THEN EXIT;
                IF wk[i] = '↑ THEN wk[i] ← Usual[GetNChar[]];
                i ← i + 1;
                ENDLOOP;
        IF wk[i] = 0C AND ~num THEN P.SErr["String ran off end, probably omitted quote"];
        IF i > = wk.maxlength THEN P.SErr["String too long for parser"];
        wk.length ← i;
        PeekTok ← STR;
        PeekVal ← P.MakeSTR[wk];
        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"];
        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
                sym ← P.Lookup["seqofc"];
                PeekTok ← sym.tok;
                PeekVal ← sym;
                END
        ELSE IF nchar ~ = ', THEN BEGIN
                PeekTok ← COMMA;
                got ← TRUE;
                END;
        END;
'| = > PeekTok ← PALT;
'. = > BEGIN
        [] ← GetNChar[];
        IF nchar ~ = '. THEN P.SErr["Unknown character '.'"]
        ELSE BEGIN
                [] ← GetNChar[];
                IF nchar ~ = '. THEN P.SErr["Unknown character '.'"]
                ELSE PeekTok ← HOLE;
                END;
```

```
        END;
'+ = > PeekTok ← PLUS;
'* = > PeekTok ← DELETE;
'> = > PeekTok ← RARR;
'# = > PeekTok ← WILD;
'; = > BEGIN
        stcnt ← stcnt + 1;
        PeekTok ← SEP;
        END;
'← = > PeekTok ← ASS;
'- = > 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;
'? = > BEGIN
        sym ← P.Lookup["opt"];
        PeekTok ← sym.tok;
        PeekVal ← sym;
        END;
'! = > BEGIN
        sym ← P.Lookup["seqof"];
        PeekTok ← sym.tok;
        PeekVal ← sym;
        END;
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 built-in function name"];
                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"];
IF ~got THEN [] ← GetNChar[];
```

```
        ENDLOOP;
RETURN[TokTok];
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 ",linecnt,stcnt,charcnt,str];
IF str1 ~= NIL THEN DispDefs.WF1["%s",str1];
DispDefs.WF0["*n"];
DispDefs.WF0["In command "];
WHILE savinput.node ~= NIL DO
        c ← P.Item[@savinput];
        IF c = IODefs.CR THEN EXIT;
        DispDefs.WF1["%c",c];
        ENDLOOP;
DispDefs.WF0["*n"];
END;

ErrorMsg1: PUBLIC PROCEDURE[str: STRING, a: UNSPECIFIED] = BEGIN
i: INTEGER ← a;
DispDefs.WF4["Line: %d Stmt: %d Char: %d, %s ",linecnt,stcnt,charcnt,str];
DispDefs.WF1["%d",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[NIL];
[] ← P.Insert["seqof",SEQOF,unk,PLDefs.Unbound,PLDefs.Unbound];
[] ← P.Insert["seqofc",SEQOFC,unk,PLDefs.Unbound,PLDefs.Unbound];
[] ← P.Insert["opt",OPT,unk,PLDefs.Unbound,PLDefs.Unbound];
[] ← P.Insert["fail",FAIL,unk,PLDefs.Unbound,PLDefs.Unbound];
END;


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

```
-- pl.mesa last edited by schmidt, September 23, 1978  7:42 PM
                                                       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",
                                                       TimeDefs: FROM "TimeDefs",
                                                       DisplayDefs: FROM "DisplayDefs";

pl: PROGRAM IMPORTS D1:DispDefs, D2:DispDefs, IODefs, DL1:DisplayDefs, DL2:DisplayDefs, StringDefs, FileSystemDefs,
ImageDefs, P:PLDefs, TimeDefs EXPORTS PLDefs = 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: Node;

Interactive: PROCEDURE[str: STRING] = BEGIN
prog,tt: Node ← NIL;
wk: STRING ← [PLDefs.sSize];
i: CARDINAL;
Setup[];
FOR i IN [1.. IMax] DO
        IStack[i] ← NIL;
        ENDLOOP;
DO
        ENABLE BEGIN
                UNWIND = > BEGIN D1.WF0["XXX*n"]; RETRY END;
                IODefs.Rubout = > BEGIN D1.WF0["XXX*n"]; RETRY END;
                P.EndDisplay = > RETRY;
                P.Interrupt = > BEGIN
                        D2.WF0["**** Interrupt *****n"];
                        RETRY;
                        END;
                FileSystemDefs.FileDoesNotExist = > BEGIN
                        D1.WF0["File Does Not Exist - Try Again*n"];
                        RETRY;
                        END;
                -- P.PBug = > BEGIN
                        -- P.ErrorMsg["Internal Error",NIL];
                        -- RETRY;
                        -- END;
                P.SErr = > BEGIN
                        P.ErrorMsg["Syntax Error - ",est];
                        RETRY;
                        END;
                P.RErr = > BEGIN
                        D1.WF1["Run-Time Error - %s*n",est];
                        RETRY;
                        END;
                END;
        Reset[];
        P.R2[@tt,@prog];
        FOR i IN [1.. IMax] DO
                P.R[@IStack[i]];
                ENDLOOP;
        IF str = NIL THEN BEGIN
```

```
        D1.WF0["$"];
        IF prog ~= NIL AND prog.Type = STR AND P.Length[prog] < wk.maxlength THEN
        P.MakeString[wk,prog];
        [] ← IODefs.ReadEditedString[wk,REString,TRUE];
        D1.WF0["*n"];
        END
ELSE BEGIN
        StringDefs.AppendString[wk,str];
        str ← NIL;
        END;
IF StringDefs.EquivalentString[wk,"un"] THEN BEGIN
        Popl[];
        D2.ClearScreen[];
        [] ← D2.ToggleMore[];
        D2.Print[IStack[1]];
        D2.WF0["*n"];
        [] ← D2.ToggleMore[];
        LOOP
        END
ELSE IF StringDefs.EquivalentString[wk,"reset"] THEN EXIT
ELSE IF StringDefs.EquivalentString[wk,"?"] THEN BEGIN
        D2.WF0["Any Poplar stmt, reset for reset, un for backup, ESC for last command,*n"];
        D2.WF0["percent for the debugging interface, quit to return to the O.S.*n"];
        D2.WF0["$file to run file*n"];
        LOOP;
        END
ELSE IF StringDefs.EquivalentString[wk,"%"] THEN BEGIN
        prog ← NIL;
        Main[];
        EXIT;
        END
ELSE IF StringDefs.EquivalentString[wk,"quit"] THEN BEGIN
        Abort ← TRUE;
        EXIT;
        END
ELSE IF StringDefs.EquivalentString[wk,"more"] THEN BEGIN
        D2.ClearScreen[];
        D2.Print[IStack[1]];
        D2.WF0["*n"];
        LOOP;
        END
ELSE IF StringDefs.EquivalentString[wk,"debug"] THEN BEGIN
        MiscDefs.CallDebugger["string"];
        LOOP;
        END
ELSE IF StringDefs.EquivalentString[wk,"make"] THEN BEGIN
        D1.WF0["*nMakeImage [Confirm]"];
        IF D1.Confirm[] THEN BEGIN
                Cleanup[];
                ImageDefs.MakeImage["Poplar.Image"];
                D1.WF0["Poplar Version 0.0 Welcomes You*n"];
                Setup[];
                FOR i IN [1.. IMax] DO
                        IStack[i] ← NIL;
                        ENDLOOP;
                END;
        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;
        P.DefaultName[wk];
        prog ← P.MakeSTR[wk];
        prog ← P.FileRoutine[NIL,prog,NIL];
        D2.Print[prog ! P.EndDisplay = > CONTINUE];
        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];
        tt ← Fixup[tt];
        PushI[P.Eval[tt]];
        IF tt.Type ~= ASS THEN BEGIN
                D2.ClearScreen[];
                [] ← D2.ToggleMore[];
                IF cdebug AND IsDebug[] THEN BEGIN
                        P.Detail[IStack[1]];
                        IStack[1] ← P.FixRep[IStack[1]];
                        P.Detail[IStack[1]];
                        numnodes ← 0;
                        P.Preorder[IStack[1],CountDepth];
                        D2.WF1["Total nodes %u*n",numnodes];
                        END;
                D2.Print[IStack[1]];
                D2.WF0["*n"];
                [] ← D2.ToggleMore[];
                END;
        NoteTime[];
        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 "];
                debug ← ~debug;
                IF debug THEN D1.WF0["TRUE*n"] ELSE D1.WF0["FALSE*n"];
                END;
        'q,'Q = > BEGIN
                D1.WF0["Quit [Confirm] "];
                IF D1.Confirm[] THEN BEGIN
                        D1.WF0["*n"];
                        RETURN;
                        END;
                D1.WF0["*n"];
                END;
        'a,'A = > BEGIN
                D1.WF0["Abort [Confirm] "];
                IF D1.Confirm[] THEN ImageDefs.AbortMesa[];
                END;
        ' ,'p,'P = > BEGIN
                IF c ~= ' THEN D1.WF0["Program:"];
                D1.WF0[" "];
                IF prog ~= NIL THEN P.MakeString[wk,prog];
                IODefs.ReadLine[wk];
                Setup[];
                prog ← P.MakeSTR[wk];
                D1.WF0["Start:*n"];
                IF P.Length[prog] ~= 0 THEN tt ← P.Dist[prog];
                P.R[@tt];
                P.ParseTree[tt];
                END;
        ENDCASE = > D1.WF0["*n"];
        ENDLOOP;
END;

StatRoutine: PROCEDURE[s:Symbol,obj,n2: Node] RETURNS[Node] = BEGIN
P.SnapShot[];
```

```
numnodes ← 0;
P.Preorder[obj,CountDepth];
D2.WF1["Total nodes %u, ",numnodes];
D2.WF1["Str lim %u*n",P.GetSin[]];
P.Detail[obj];
RETURN[NIL];
END;

Fixup: PUBLIC PROCEDURE[n: Node] RETURNS[Node] = BEGIN
-- fixes up things so zary and unary functions do not need slashes
-- if they are leftmost in the tree
IF n = NIL THEN RETURN[NIL];
SELECT n.Type FROM
PROG = > BEGIN
        n.prog1 ← Fixup[n.prog1];
        n.prog2 ← Fixup[n.prog2];
        END;
LIST = >    n.listhead ← Fixup[n.listhead];
CAT,CATL = >          n.left ← Fixup[n.left];
MATCH = > n.div ← Fixup[n.div];
PALT = > n.alt1 ← Fixup[n.alt1];
APPLY,MAPPLY,GOBBLE,ITER = > n.object ← Fixup[n.object];
FCN = >    n.parms ← Fixup[n.parms];
ASS = >    BEGIN
        n.lhs ← Fixup[n.lhs];
        n.rhs ← Fixup[n.rhs];
        END;
DELETE = > n.delete ← Fixup[n.delete];
SEQUENCE = > n.from ← Fixup[n.from];
PLUS,MINUS = > n.arg1 ← Fixup[n.arg1];
ZARY,UNARY = > BEGIN
        RETURN[P.Alloc[[APPLY,,APPLY[IStack[1],n]]]];
        END;
ENDCASE;                                    -- others fall through
RETURN[n];
END;

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

PushI: 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;

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

REString: PROCEDURE[c: CHARACTER] RETURNS[BOOLEAN] = BEGIN
RETURN[c = IODefs.ESC];
END;

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

NoteTime: PROCEDURE = BEGIN
OPEN TimeDefs;
str: STRING ← [40];
IF ~cdebug THEN RETURN;
AppendDayTime[str,UnpackDT[CurrentDayTime[]]];
D2.WF1["%s*n",str];
```

```
END;

StartUp: PROCEDURE = BEGIN
-- DL1.SetTypescript[NIL];
DL2.SetSystemDisplaySize[PLDefs.NumLines,PLDefs.NumPages];
DL1.SetSystemDisplaySize[6,6];
DL1.SetDummyDisplaySize[50];
P.AmbushKeyStream[];
D1.WF0["Poplar Version 0.0 Welcomes You*n"];
END;

Setup: PROCEDURE = BEGIN
P.StoreSetup[];                                  -- inserts don't work till this is done
P.VMSetup[];
P.ParseSetup[];
P.SupSetup[];
P.StringSetup[];
P.RouteSetup[];
P.StatSetup[];
D2.DispSetup[];
D1.DispSetup[];
[] ← P.Insert["stat",ZARY,proc,StatRoutine,PLDefs.Unbound];
LOOPHOLE[424B,POINTER]↑ ← 500;                   -- mouse X coord.
LOOPHOLE[425B,POINTER]↑ ← 650;                   -- mouse Y coord.
END;

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

Reset: PROCEDURE = BEGIN
P.StoreReset[];
P.SupReset[];
D2.DispReset[];
D1.DispReset[];
END;

-- main program
START DL1.DisplayControl;
StartUp[];
[Fail,MTSt] ← P.GetSpecialNodes[];
WHILE ~Abort DO Interactive[NIL] ENDLOOP;
ImageDefs.StopMesa[];
END.
```

```
-- route.mesa last edited by schmidt, September 22, 1978  11:10 PM
                                                    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",
                                                    FilePageUseDefs: FROM "FilePageUseDefs",
                                                    RandomDefs: FROM "RandomDefs",
                                                    StreamDefs: FROM "StreamDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    StringDefs: FROM "StringDefs",
                                                    SegmentDefs: FROM "SegmentDefs";

route: PROGRAM IMPORTS D1: DispDefs, D2: DispDefs, IODefs,  P:PLDefs, FileSystemDefs, FilePageUseDefs, StringDefs,
DirectoryDefs  EXPORTS PLDefs SHARES RandomDefs  = BEGIN
--
Node: TYPE  =  PLDefs.Node;
Symbol: TYPE  =  PLDefs.Symbol;
Stream: TYPE  =  PLDefs.Stream;
Register: TYPE  =  PLDefs.Register;
--
Fail,MTSt: Node;

RouteSetup: PUBLIC PROCEDURE  =  BEGIN
[] ← P.Insert["chop",ZARY,proc,ChopRoutine,PLDefs.Unbound];
[] ← P.Insert["conc",ZARY,proc,ConcRoutine,PLDefs.Unbound];
[] ← P.Insert["confirm",ZARY,proc,ConfirmRoutine,PLDefs.Unbound];
[] ← P.Insert["debravo",ZARY,proc,DeBravoRoutine,PLDefs.Unbound];
[] ← P.Insert["delete",ZARY,proc,DeleteRoutine,PLDefs.Unbound];
[] ← P.Insert["differ",ZARY,proc,DifferRoutine,PLDefs.Unbound];
[] ← P.Insert["dir",ZARY,proc,DirRoutine,PLDefs.Unbound];
[] ← P.Insert["dirlength",ZARY,proc,DirLengthRoutine,PLDefs.Unbound];
[] ← P.Insert["display",ZARY,proc,DisplayRoutine,PLDefs.Unbound];
[] ← P.Insert["divide",ZARY,proc,DivideRoutine,PLDefs.Unbound];
[] ← P.Insert["edit",ZARY,proc,EditRoutine,PLDefs.Unbound];
[] ← P.Insert["exec",ZARY,proc,ExecRoutine,PLDefs.Unbound];
[] ← P.Insert["ident",ZARY,proc,IdentRoutine,PLDefs.Unbound];
[] ← P.Insert["isfail",ZARY,proc,IsFailRoutine,PLDefs.Unbound];
[] ← P.Insert["islist",ZARY,proc,IsListRoutine,PLDefs.Unbound];
[] ← P.Insert["isnull",ZARY,proc,IsNullRoutine,PLDefs.Unbound];
[] ← P.Insert["isstring",ZARY,proc,IsStringRoutine,PLDefs.Unbound];
[] ← P.Insert["key",ZARY,proc,KeyRoutine,PLDefs.Unbound];
[] ← P.Insert["last",ZARY,proc,LastRoutine,PLDefs.Unbound];
[] ← P.Insert["length",ZARY,proc,LengthRoutine,PLDefs.Unbound];
[] ← P.Insert["lines",ZARY,proc,LinesRoutine,PLDefs.Unbound];
[] ← P.Insert["listin",ZARY,proc,ListInRoutine,PLDefs.Unbound];
[] ← P.Insert["listout",UNARY,proc,ListOutRoutine,PLDefs.Unbound];
[] ← P.Insert["loop",ZARY,proc,LoopRoutine,PLDefs.Unbound];
[] ← P.Insert["marry",ZARY,proc,MarryRoutine,PLDefs.Unbound];
[] ← P.Insert["max",ZARY,proc,MaxRoutine,PLDefs.Unbound];
[] ← P.Insert["min",ZARY,proc,MinRoutine,PLDefs.Unbound];
[] ← P.Insert["minus",ZARY,proc,MinusRoutine,PLDefs.Unbound];
[] ← P.Insert["plus",ZARY,proc,PlusRoutine,PLDefs.Unbound];
[] ← P.Insert["print",ZARY,proc,PrintRoutine,PLDefs.Unbound];
[] ← P.Insert["quit",ZARY,proc,QuitRoutine,PLDefs.Unbound];
[] ← P.Insert["reverse",ZARY,proc,ReverseRoutine,PLDefs.Unbound];
[] ← P.Insert["run",ZARY,proc,RunRoutine,PLDefs.Unbound];
[] ← P.Insert["step",ZARY,proc,StepRoutine,PLDefs.Unbound];
[] ← P.Insert["subst",ZARY,proc,SubstRoutine,PLDefs.Unbound];
[] ← P.Insert["times",ZARY,proc,TimesRoutine,PLDefs.Unbound];
[] ← P.Insert["tolower",ZARY,proc,ToLowerRoutine,PLDefs.Unbound];
[] ← P.Insert["toupper",ZARY,proc,ToUpperRoutine,PLDefs.Unbound];
[] ← P.Insert["uniq",ZARY,proc,UniqRoutine,PLDefs.Unbound];
[] ← P.Insert["write",UNARY,proc,WriteRoutine,PLDefs.Unbound];
[] ← P.Insert["zip",ZARY,proc,ZipRoutine,PLDefs.Unbound];
END;
```

```
ChopRoutine: PROCEDURE[s:Symbol,input,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
rr: Register ← P.MRS[];
pn: POINTER TO Node;
ns: PLDefs.StreamRecord;
str: STRING ← [2];
ans ← NIL;
IF input = NIL OR input.Type ~= STR THEN P.RErr["Input to chop must be a string"];
ns ← P.NewStream[input];
P.R2[@ns.node,@ans];
str.length ← 1;
pn ← @ ans;
WHILE ns.node ~= NIL DO
        str[0] ← P.Item[@ns];
        pn↑ ← P.Alloc[[LIST,,LIST[P.MakeSTR[str],NIL]]];
        pn ← @((pn↑).listtail);
        ENDLOOP;
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 ~= STR AND n1.Type ~= LIST THEN P.RErr["Conc expects a list or string"];
ans ← n1;
P.R[@ans];
WHILE n1 ~= NIL AND n1.Type = LIST DO
        IF n1.listhead.Type ~= STR THEN P.RErr["Conc expects a list of strings"];
        ans ← IF ans ~= n1 THEN P.StringConcat[ans,n1.listhead] ELSE n1.listhead;
        n1 ← n1.listtail;
        ENDLOOP;
P.RRS[rr];
END;

ConfirmRoutine: PROCEDURE[sym:Symbol,n,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
pans: POINTER TO Node;
rr: Register ← P.MRS[];
IF n.Type ~= LIST THEN P.RErr["confirm expects a list"];
ans ← NIL;
pans ← @ans;
P.R[@ans];
WHILE n ~= NIL AND n.Type = LIST DO
        D1.Print[n.listhead];
        D1.WF0[" ? "];
        IF D1.Confirm[] THEN BEGIN
                pans↑ ← P.Alloc[[LIST,,LIST[n.listhead,NIL]]];
                pans ← @pans.listtail;
                END;
        D1.WF0["*n"];
        n ← n.listtail;
        ENDLOOP;
P.RRS[rr];
END;

DeBravoRoutine: PROCEDURE[sym:Symbol,inputnode,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
rr: Register ← P.MRS[];
k: LONG INTEGER;
input,sav: PLDefs.StreamRecord;
t: Node ← NIL;
IF inputnode.Type ~= STR THEN P.RErr["debravo expects a string as input"];
sav ← input ← P.NewStream[inputnode];
```

```
ans ← MTSt;
P.R3[@ans,@input.node,@t];
P.R[@sav.node];
k ← 0;
WHILE input.node ~= NIL DO
        IF P.Item[@input] = IODefs.ControlZ THEN BEGIN
                t ← IF k > 0 THEN P.SubStringStream[@sav,0,k] ELSE MTSt;
                ans ← P.StringConcat[ans,t];
                sav ← input;
                WHILE input.node ~= NIL AND P.Item[@input] ~= IODefs.CR DO
                        sav ← input
                        ENDLOOP;
                k ← 1;
                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;

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"];
P.MakeString[fname,name];
fs ← FileSystemDefs.Login['a,NIL,NIL,NIL];
FileSystemDefs.Delete[fs,fname];
FileSystemDefs.Logout[fs];
RETURN[name];
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 LengthList[node] ~= 2 THEN P.RErr["differ expects a list of length 2"];
ans ← NIL;
P.R[@ans];
n1 ← node.listhead;
n2 ← node.listtail.listhead;
IF n1.Type ~= STR OR n2.Type ~= STR THEN P.RErr["Differ expects both list elements to be strings"];
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],NIL]]];
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;
panst ← P.Alloc[[LIST,,LIST[P.MakeSTR[s],NIL]]];
pans ← @pans.listtail;
RETURN[FALSE];
```

```
END;

ans ← NIL;
P.R[@ans];
pans ← @ans;
DirectoryDefs.EnumerateDirectory[AddName];
ans.Des.e ← TRUE;
P.RRS[rr];
END;

DirLengthRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
rr: Register ← P.MRS[];
temp: Node ← NIL;
pans: POINTER TO Node;
fs: FileSystemDefs.FileSystem;

AddName: PROCEDURE[p:POINTER,s: STRING] RETURNS[BOOLEAN] = BEGIN
len: LONG INTEGER;
fh: FilePageUseDefs.FileHandle;
lp,bp,pg: CARDINAL;
s.length ← s.length - 1;
len ← 0;
BEGIN
fh ← FileSystemDefs.Open[fs,s,FileSystemDefs.OpenMode[read]
        ! ANY = > GOTO nop];
[lp,bp,pg] ← FilePageUseDefs.Measure[fh ! ANY = > GOTO nop];
FilePageUseDefs.Close[fh!ANY = > CONTINUE];
len ← lp;
len ← len*pg + bp;
EXITS
nop = > NULL;
END;
temp ← P.Alloc[[LIST,,LIST[P.MakeNUM[len],NIL]]];
pans↑ ← P.Alloc[[LIST,,LIST[P.Alloc[[LIST,,LIST[P.MakeSTR[s],temp]]],NIL]]];
pans ← @pans.listtail;
RETURN[FALSE];
END;

ans ← NIL;
P.R2[@ans,@temp];
pans ← @ans;
fs ← FileSystemDefs.Login['a,NIL,NIL,NIL];
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 LengthList[n] ~= 2 THEN P.RErr["Divide expects a list of length 2"];
res ← NIL;
P.R[@res];
a ← P.MakeInteger[n.listhead];
b ← P.MakeInteger[n.listtail.listhead];
i ← a/b;
j ← a - (i*b);
```

```
res ← P.Alloc[[LIST,,LIST[P.MakeNUM[j],NIL]]];
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"];
P.MakeString[s,str];
P.Editor[s];
-- no return
RETURN[NIL];
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"];
StringDefs.AppendString[s,"; PoplarCheckPoint.Image$"];
StringDefs.AppendChar[s,IODefs.CR];
n ← P.MakeSTR[s];
P.R[@n];
n ← P.StringConcat[str,n];
P.Execute[n];
P.RRS[rr];
RETURN[str];
END;

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

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

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

IsNullRoutine: PROCEDURE[sym:Symbol,list,n2: Node] RETURNS[Node] = BEGIN
-- zary
IF list = NIL OR (list.Type = LIST AND list.listhead = NIL AND list.listtail = NIL) THEN RETURN[MTSt];
RETURN[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;

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

REString: PROCEDURE[c: CHARACTER] RETURNS[BOOLEAN] = BEGIN
Bad ← c = IODefs.DEL;
RETURN[c = IODefs.DEL OR c = IODefs.ESC];
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];
Bad: BOOLEAN ← FALSE;
-- should use UsualEscape
[] ← IODefs.ReadEditedString[s,REString,FALSE];
D2.WF0["*n"];
RETURN[IF Bad THEN Fail ELSE P.MakeSTR[s]];
END;

LastRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
-- zary
IF n1 = NIL THEN P.RErr["Last expects a list as argument"];
IF n1.Type ~= LIST THEN RETURN[n1];
WHILE n1.listtail ~= NIL DO
        n1 ← n1.listtail;
        ENDLOOP;
RETURN[n1.listhead];
END;

LengthRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
i: InlineDefs.LongCARDINAL;
IF n1 = NIL THEN RETURN[P.MakeNUM[0]];
IF n1.Type = STR THEN BEGIN
        i ← LOOPHOLE[P.Length[n1]];
        RETURN[P.MakeNUM[i.lowbits]];
        END;
IF n1.Type = LIST THEN RETURN[P.MakeNUM[LengthList[n1]]];
P.RErr["Can only take length of strings and lists"];
END;

LinesRoutine: PROCEDURE[sym:Symbol,input,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
rr: Register ← P.MRS[];
pans: POINTER TO Node;
s,a: PLDefs.StreamRecord;
i: CARDINAL;
IF input.Type ~= STR THEN P.RErr["Lines expects a string as input"];
s ← P.NewStream[input];
ans ← NIL;
P.R2[@ans,@s.node];
pans ← @ans;
WHILE s.node ~= NIL DO
        i ← 0;
        a ← s;
        WHILE s.node ~= NIL AND P.Item[@s] ~= IODefs.CR DO
                i ← i + 1;
                ENDLOOP;
        pans↑ ← P.Alloc[[LIST,,LIST[P.SubStringStream[@a,0,i + 1],NIL]]];
        pans ← @pans.listtail;
        ENDLOOP;
P.RRS[rr];
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"];
ans ← NIL;
P.R[@ans];
ans ← P.FileRoutine[sym,name,n2];
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"];
P.MakeString[fname,name];
fs ← FileSystemDefs.Login['a,NIL,NIL,NIL];
fh ← FileSystemDefs.Open[fs,fname,FileSystemDefs.OpenMode[create]];
[lp,bp,pgsize] ← FilePageUseDefs.Measure[fh];
OP ← D2.SetWriteProcedure[ListWriteProc];
[] ← D2.ToggleAllPrint[];
pno ← 0;
i ← 0;
D2.Print[list];
FilePageUseDefs.WritePage[fh,pno,buf + 2];                    -- last page
[] ← D2.SetWriteProcedure[OP];
[] ← D2.ToggleAllPrint[];
FilePageUseDefs.SetLength[fh,pno,i];
FilePageUseDefs.Close[fh];
FileSystemDefs.Logout[fs];
P.RRS[rr];
RETURN[list];
END;

LoopRoutine: PROCEDURE[sym:Symbol,prog,n2: Node] RETURNS[Node] = BEGIN
-- zary
-- remember the string must be quoted to avoid being evaluated by interactive
rr: Register ← P.MRS[];
tt: Node ← NIL;
IF prog.Type ~ = STR THEN P.RErr["Input to loop must be string"];
P.R[@tt];
D2.ClearScreen[];
DO
        tt ← P.Dist[prog];
        tt ← P.Fixup[tt];
        tt ← P.Eval[tt];
        D2.Print[tt];
        D2.WF0["*n"];
        ENDLOOP;
-- no loop exit
RETURN[NIL];
END;

MarryRoutine: PROCEDURE[sym:Symbol,node,other: Node] RETURNS[ans:Node] = BEGIN
n1,n2,j: Node ← NIL;
pans: POINTER TO Node;
rr: Register;
-- unary
IF node.Type ~ = LIST OR LengthList[node] ~ = 2 THEN P.RErr["Marry expects a list of length 2"];
```

```
n1 ← node.listhead;
n2 ← node.listtail.listhead;
IF LengthList[n1] ~= LengthList[n2] THEN P.RErr["Both lists must have the same length"];
IF n1 = NIL THEN RETURN[P.Alloc[[LIST,,LIST[n2,NIL]]]];
IF n2 = NIL THEN RETURN[P.Alloc[[LIST,,LIST[n1,NIL]]]];
ans ← NIL;
rr ← P.MRS[];
P.R2[@ans,@j];
pans ← @ans;
WHILE n1 ~= NIL AND n2 ~= NIL DO
        j ← P.Alloc[[LIST,,LIST[n2.listhead,NIL]]];
        j ← P.Alloc[[LIST,,LIST[n1.listhead,j]]];
        pans↑ ← P.Alloc[[LIST,,LIST[j,NIL]]];
        pans ← @pans.listtail;
        n1 ← n1.listtail;
        n2 ← n2.listtail;
        ENDLOOP;
P.RRS[rr];
END;

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

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

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

PlusRoutine: PROCEDURE[sym:Symbol,n,n2: Node] RETURNS[Node] = BEGIN
-- zary
IF n.Type ~= LIST OR LengthList[n] ~= 2 THEN P.RErr["Plus expects a list of length 2"];
```

```
RETURN[P.MakeNUM[P.MakeInteger[n.listhead] + P.MakeInteger[n.listtail.listhead]]];
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"];
D2.WF1["%a",n1];
RETURN[n1];
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"];
StringDefs.AppendChar[s,IODefs.CR];
n ← P.MakeSTR[s];
P.R[@n];
n ← P.StringConcat[str,n];
P.WriteRem[n];
ImageDefs.StopMesa[];
-- no return
RETURN[NIL];
END;

ReverseRoutine: PROCEDURE[sym:Symbol,n,n2: Node] RETURNS[ans:Node] = BEGIN
-- zary
rr: Register ← P.MRS[];
s: PLDefs.StreamRecord;
t: Node ← NIL;
ans ← NIL;
P.R2[@ans,@t];
IF n ~= NIL AND n.Type ~= LIST AND n.Type ~= STR THEN P.RErr["Reverse expects a list or string as
input"];
WHILE n ~= NIL AND n.Type = LIST DO
        ans ← P.Alloc[[LIST,,LIST[n.listhead,ans]]];
        n ← n.listtail;
        ENDLOOP;
IF n~= NIL AND 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[ans,t];
                ENDLOOP;
        END;
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"];
tt ← P.Dist[prog];
P.R[@tt];
tt ← P.Fixup[tt];
tt ← P.Eval[tt];
P.RRS[rr];
RETURN[tt];
END;

StepRoutine: PROCEDURE[sym:Symbol,progstr,n2: Node] RETURNS[ret:Node] = BEGIN

PEval: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
IF n.Type = PROG THEN RETURN[TRUE];
D2.Print[n];
D2.WF0["*nEval? "];
IF ~D2.Confirm[] THEN P.Interrupt;
D2.WF0["*n"];
ret ← P.Eval[n];
D2.Print[ret];
D2.WF0["*n"];
RETURN[FALSE];
END;

rr: Register ← P.MRS[];
prog:Node ← NIL;
IF progstr.Type ~= STR THEN P.RErr["Input to step must be string"];
ret ← NIL;
D2.ClearScreen[];
[] ← D2.ToggleMore[];
P.R2[@prog,@ret];
prog ← P.Dist[progstr];
prog ← P.Fixup[prog];
P.Preorder[prog,PEval];
P.RRS[rr];
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"];
sav ← input ← P.NewStream[inputnode];
P.R3[@output,@t,@input.node];
P.R[@sav.node];
```

```
D2.WF0["replacement string: "];
output ← KeyRoutine[NIL,NIL,NIL];
IF output.Type = FAIL THEN P.Interrupt;
D2.WF0["pattern string: "];
t ← KeyRoutine[NIL,NIL,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"];
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 LengthList[n] ~= 2 THEN P.RErr["Times expects a list of length 2"];
RETURN[P.MakeNUM[P.MakeInteger[n.listhead]*P.MakeInteger[n.listtail.listhead]]];
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"];
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;

UniqRoutine: PROCEDURE[s:Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
-- zary
-- print on terminal the incoming string, with special characters highlighted
IF n1.Type ~= STR THEN P.RErr["Uniq requires a string as input"];
D2.WF1["%f",n1];
RETURN[n1];
END;

WriteRoutine: PUBLIC PROCEDURE[s:Symbol,input,name: Node] RETURNS[Node] = BEGIN
-- unary
fname: STRING ← [40];
ns: PLDefs.StreamRecord;
rr: Register;
str: STRING ←[600];
i,j,lp,bp,pg: CARDINAL;
fs: FileSystemDefs.FileSystem;
fh: FilePageUseDefs.FileHandle;
IF input.Type ~= STR THEN P.RErr["Input to write routine must be string"];
IF name.Type ~= STR THEN P.RErr["Filename for write routine must be string"];
P.MakeString[fname,name];
fs ← FileSystemDefs.Login['a,NIL,NIL,NIL];
fh ← FileSystemDefs.Open[fs,fname,FileSystemDefs.OpenMode[create]
        ! FileSystemDefs.FileAlreadyExists => RESUME];
[lp,bp,pg] ← FilePageUseDefs.Measure[fh];
i ← 0;
ns ← P.NewStream[input];
rr ← P.MRS[];
P.R[@ns.node];
WHILE ns.node ~= NIL DO
        j ← 0;
        DO
                IF ns.node = NIL OR j >= pg THEN EXIT;
                str[j] ← P.Item[@ns];
                j ← j + 1;
                ENDLOOP;
        str.length ← j;
        FilePageUseDefs.WritePage[fh,i,str + 2];
        i ← i + 1;
        ENDLOOP;
FilePageUseDefs.SetLength[fh,IF i = 0 THEN 0 ELSE i-1,str.length];
FilePageUseDefs.Close[fh];
FileSystemDefs.Logout[fs];
P.RRS[rr];
RETURN[input];
END;

ZipRoutine: PROCEDURE[sym:Symbol,node,other: Node] RETURNS[ans:Node] = BEGIN
-- unary
pans: POINTER TO Node;
n1,n2: Node ← NIL;
rr: Register;
IF node.Type ~= LIST OR LengthList[node] ~= 2 THEN P.RErr["Zip expects a list of length 2"];
n1 ← node.listhead;
n2 ← node.listtail.listhead;
```

```
IF n1 = NIL THEN RETURN[n2];
IF n2 = NIL THEN RETURN[n1];
IF n1.Type#LIST OR n2.Type#LIST THEN RETURN[P.Alloc[[LIST,,LIST[n1,n2]]]];
rr ← P.MRS[];
ans ← NIL;
P.R3[@ans,@n1,@n2];
pans ← @ans;
WHILE n1 ~= NIL AND n2 ~= NIL DO
        pans↑ ← P.Alloc[[LIST,,LIST[n1.listhead,NIL]]];
        pans ← @pans.listtail;
        pans↑ ← P.Alloc[[LIST,,LIST[n2.listhead,NIL]]];
        pans ← @pans.listtail;
        n1 ← n1.listtail;
        n2 ← n2.listtail;
        ENDLOOP;
pans↑ ← IF n1 = NIL THEN n2 ELSE n1;
P.RRS[rr];
END;

LengthList: PUBLIC PROCEDURE[n: Node] RETURNS[i:CARDINAL] = BEGIN
IF n = NIL THEN RETURN[0];
IF n.Type ~= LIST THEN RETURN[1];
i ← IF n.listhead = NIL THEN 0 ELSE 1;
WHILE n.listtail ~= NIL DO
        i ← i + 1;
        n ← n.listtail;
        ENDLOOP;
END;

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

```
-- stat.mesa last edited by schmidt, September 14, 1978  9:37 PM
                                                    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",
                                                    StringDefs: FROM "StringDefs";

stat: PROGRAM IMPORTS DispDefs, P:PLDefs, SystemDefs 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: Node;
dnum: CARDINAL = 150;
len: LONG INTEGER;
defaultBal: ARRAY[0..dnum) OF Node;

Empty: PUBLIC PROCEDURE[n: Node] RETURNS [BOOLEAN] =
        BEGIN
        str: String;
        IF n.Type # STR THEN P.PBug["Non-String"];
        str ← @n.str;
        IF n.Des.s = simp THEN RETURN [str.length = 0]
        ELSE IF n.Des.s = file THEN RETURN[str.f.len = 0]
        ELSE IF n.Des.s = cat THEN RETURN[Empty[str.n1] AND Empty[str.n2]]
        ELSE P.PBug["unknown variant"];
        END;

Length: PUBLIC PROCEDURE[n: Node] RETURNS [LONG INTEGER] =
        BEGIN
        str: String;
        i: LONG INTEGER;
        IF n = NIL THEN RETURN[0];
        IF n.Type # STR THEN P.PBug["Non-String"];
        str ← @n.str;
        IF n.Des.s = simp THEN BEGIN
                i ← str.length;
                RETURN[i];
                END
        ELSE IF n.Des.s = file THEN RETURN[str.f.len]
        ELSE IF n.Des.s = cat THEN RETURN[Length[str.n1] + Length[str.n2]]
        ELSE P.PBug["unknown variant"];
        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];
        IF ii > jj THEN jj ← ii;
        str ← @n.str;
        IF ii = 0 AND Length[n] = jj THEN ans ← n
        ELSE IF n.Des.s = simp THEN BEGIN
                i, j: CARDINAL;
```

```
                i ← LOOPHOLE[ii,InlineDefs.LongCARDINAL].lowbits;
                j ← LOOPHOLE[jj,InlineDefs.LongCARDINAL].lowbits;
                IF j > str.length THEN P.PBug["SubString out of bounds"];
                ans ←P.Alloc[[STR,[,,simp,],STR[[simp[str.start + i, j-i]]]]]
                END
        ELSE IF n.Des.s = file THEN BEGIN
                IF jj > n.str.f.len THEN P.PBug["SubString out of bounds"];
                ns1 ← P.CopyTree[n];
                Skip[ns1, ii];
                ns1.str.f.len ← jj-ii;
                ans ← IF jj-ii < MinCoerceSize THEN P.Coerce[ns1] ELSE ns1;
                END
        ELSE IF n.Des.s = cat THEN
                BEGIN
                kk: LONG INTEGER ← 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"];
        P.RRS[rr];
        END;


-- used in SubString and ConvertStream only
Skip: PUBLIC PROCEDURE[n:Node,i: LONG INTEGER] = BEGIN
a: CARDINAL;
ii: LONG INTEGER;
s: String;
f: PLDefs.StringFile;
-- 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"];
IF n.Des.s = simp THEN BEGIN
        a ← LOOPHOLE[i,InlineDefs.LongCARDINAL].lowbits;
        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 = file THEN BEGIN
        f ← n.str.f;
        ii ← f.inx + i;
        f.pgno ← f.pgno + LOOPHOLE[ii/n.str.of.pgsize,LongCARDINAL].lowbits;
        f.inx ← LOOPHOLE[ii,LongCARDINAL].lowbits MOD n.str.of.pgsize;
        f.len ← IF i >= f.len THEN 0 ELSE f.len - i;
        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];
                n↑ ← n.str.n2↑;
                Skip[n,i-ii];
                END;
        END
ELSE P.PBug["unknown STR type"];
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,", @ii, @maxlength, numnodes,numcat];
        DispDefs.WF4["File %u,Smp %u,spSmp %u,nest %u,",numfile, numsimp, spacesimp, CatDepth[n]];
        DispDefs.WF2["SF %u, nlist %u*n",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 = file THEN numfile ← numfile + 1
                        ELSE IF n.Des.s = simp THEN BEGIN
                                numsimp ← numsimp + 1;
                                spacesimp ← spacesimp + n.str.length;
                                END;
                        END;
                END;
        RETURN[TRUE];
        END;
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;
END;

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

```
                END
        ELSE BEGIN
                a ← n;
                rr ← P.MRS[];
                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];
                P.RRS[rr];
                END;
        END;
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]];
END;

LengthStream: PUBLIC PROCEDURE[s: Stream] RETURNS [LONG INTEGER] = BEGIN
RETURN[IF s.node = NIL THEN 0 ELSE Length[s.node] - s.posn];
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];
        END
ELSE ans ← MTSt;
END;

SkipStream: PUBLIC PROCEDURE[s: Stream,inx: LONG INTEGER] = BEGIN
n: Node;
ii: LONG INTEGER;
WHILE s.node ~= NIL DO
        n ← s.node;
        IF n.Des.s = simp OR n.Des.s = file THEN BEGIN
                ii ← Length[n];
                s.posn ← s.posn + inx;
                IF s.posn > ii THEN P.PBug["bad string length skip"];
                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 + inx;
                        RETURN;
                        END
                END
        ELSE P.PBug["unknown str type - skipstream"];
        ENDLOOP;
IF inx > 0 THEN P.PBug["skip past end"];
END;
```

```
SubStringStream: PUBLIC PROCEDURE[s: Stream,i,j: LONG INTEGER] RETURNS [Node] = BEGIN
RETURN[IF s.node = NIL THEN MTSt ELSE SubString[s.node,s.posn + i,s.posn + 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];
        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",PFUNC,patproc,PLDefs.Unbound,WordRoutine];
[] ← P.Insert["thing",PFUNC,patproc,PLDefs.Unbound,ThingRoutine];
[] ← P.Insert["integer",PFUNC,patproc,PLDefs.Unbound,IntegerRoutine];
[] ← P.Insert["number",PFUNC,patproc,PLDefs.Unbound,NumberRoutine];
[] ← P.Insert["item",PFUNC,patproc,PLDefs.Unbound,ItemRoutine];
[] ← P.Insert["digit",PFUNC,patproc,PLDefs.Unbound,DigitRoutine];
[] ← P.Insert["smallletter",PFUNC,patproc,PLDefs.Unbound,SmallLetterRoutine];
[] ← P.Insert["bigletter",PFUNC,patproc,PLDefs.Unbound,BigLetterRoutine];
[] ← P.Insert["letter",PFUNC,patproc,PLDefs.Unbound,LetterRoutine];
[] ← P.Insert["space",PFUNC,patproc,PLDefs.Unbound,SpaceRoutine];
[] ← P.Insert["len",PFUNC1,patproc,PLDefs.Unbound,LenRoutine];
[] ← P.Insert["blanks",PFUNC1,patproc,PLDefs.Unbound,BlanksRoutine];
END;

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

```
-- store.mesa last edited by schmidt, September 19, 1978  1:36 AM
                                                    DIRECTORY
                                                    PLDefs: FROM "PLDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    ImageDefs: FROM "ImageDefs",
                                                    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";

store: PROGRAM IMPORTS DispDefs, MiscDefs, P:PLDefs, StringDefs, SystemDefs  EXPORTS PLDefs  = BEGIN
--
Node: TYPE = PLDefs.Node;
pNode: TYPE = PLDefs.pNode;
Symbol: TYPE = PLDefs.Symbol;
Register: TYPE = PLDefs.Register;
cdebug: BOOLEAN = PLDefs.cdebug;
--
maxnNodes: CARDINAL = 1536;
nNodes: CARDINAL;
nSyms: CARDINAL = 100;
RSize: CARDINAL = 500;
PLimit: CARDINAL = 300;
psptr: CARDINAL;
RSIndex: CARDINAL;
sptr: CARDINAL;
StringCollect: BOOLEAN;
PleaseGarbageCollect: BOOLEAN = FALSE;
NeverActuallyFree: BOOLEAN ← FALSE;
NeverGarbageCollect: BOOLEAN ← FALSE;
AlwaysGarbageCollect: BOOLEAN ← FALSE;
ignorerrs: BOOLEAN;
--
FreeList: Node;
FreeCount: CARDINAL;
--
PStack: ARRAY[0..PLimit] OF Register;
RegisterStack: ARRAY[0..RSize] OF PLDefs.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],STR[[simp[0,0]]]];
MTSt: Node = @MTStRecord;

MRS: PUBLIC PROCEDURE RETURNS[Register] = BEGIN
IF cdebug THEN BEGIN
        PStack[psptr] ← RSIndex;
        psptr ← psptr + 1;
        IF psptr > PLimit THEN P.PBug["too many proc calls"];
        END;
RETURN[RSIndex];
END;

RRS: PUBLIC PROCEDURE[r: Register] = BEGIN
IF cdebug THEN BEGIN
        IF ~ignorerrs THEN BEGIN
                psptr ← psptr - 1;
                IF psptr < 1 OR r ~= PStack[psptr] THEN P.PBug["Did not get RRS right"];
                END;
        END;
RSIndex ← r;
END;

IgnoreRRS: PUBLIC PROCEDURE = BEGIN
```

```
ignorerrs ← TRUE;
END;

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

R2: PUBLIC PROCEDURE[n1,n2: pNode] = BEGIN
R[n1];
R[n2];
END;

R3: PUBLIC PROCEDURE[n1,n2,n3: pNode] = BEGIN
R[n1];
R[n2];
R[n3];
END;

MT: PROCEDURE[n: Node] RETURNS[BOOLEAN] = BEGIN
IF n ~ = NIL THEN n.Des.g ← TRUE;
RETURN[TRUE];
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
IF cdebug THEN DispDefs.WF1["Marking, %d registered, ",RSIndex-1];
FOR i IN[1..nNodes) DO
        NodeArray[i].Des.g ← FALSE;
        ENDLOOP;
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",f];
        f ← (LOOPHOLE[f,POINTER] + 1)↑;
        ENDLOOP;
END;

-- 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.

MakeFree: PROCEDURE = BEGIN
i: CARDINAL;
-- 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..nNodes) DO
        IF ~NodeArray[i].Des.g THEN [] ← FreeNode[@NodeArray[i]]
        ELSE IF NodeArray[i].Type = STR THEN P.Transfer[@NodeArray[i]];
        ENDLOOP;
```

```
ZeroOK ← FALSE;
IF cdebug THEN DispDefs.WF1["GC: %d are free*n",FreeCount];
END;

Alloc: PUBLIC PROCEDURE[r: PLDefs.NodeRecord] RETURNS[n: Node] = BEGIN
-- r must be a well formed NodeRecord
IF FreeList = NIL OR AlwaysGarbageCollect THEN BEGIN
        IF NeverGarbageCollect THEN P.PBug["Ran out of node space"]
        ELSE BEGIN
                StringCollect ← TRUE;
                P.StringGC[];
                Mark[@r];
                IF ~NeverActuallyFree THEN BEGIN
                        MakeFree[];
                        P.Transfer[@r];
                        END;
                P.EndStringGC[];
                StringCollect ← FALSE;
                IF FreeList = NIL THEN P.RErr["Ran out of node space"];
                END;
        END;
n ← FreeList;
FreeList ← (LOOPHOLE[n,POINTER] + 1)↑;
n↑ ← r;
n.Des.e ← FALSE;
n.Des.b ← FALSE;
FreeCount ← FreeCount - 1;
P.SetCursorAmt[nNodes - FreeCount, nNodes];
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];
IF n.Type = STR AND n.Des.s = file THEN P.FreeStringFile[n.str.f];
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;

CopyTree: PUBLIC PROCEDURE[n: Node] RETURNS[new: Node] = BEGIN
rr: Register ← MRS[];
new ← NIL;
R[@new];
new ← Alloc[n↑];
IF new.Type = STR THEN BEGIN
        IF new.Des.s = file THEN BEGIN
                new.str.f ← P.NewStringFile[];
                new.str.f↑ ← n.str.f↑;
                new.str.f.bp ← new;
                END
        ELSE IF new.Des.s = cat THEN BEGIN
                new.str.n1 ← CopyTree[new.str.n1];
                new.str.n2 ← CopyTree[new.str.n2];
                END;
        END;
RRS[rr]
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[Fail]];
s ← @SymbolArray[sptr];
sptr ← sptr + 1;
IF sptr > nSyms THEN P.SErr["Too many symbol table entries"];
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;

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

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

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

GarbageRoutine: PUBLIC PROCEDURE[s: Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
-- zary
FreeList ← NIL;
[] ← Alloc[[HOLE,,HOLE[]]];                          -- guaranteed to raise GC
RETURN[n1];
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 ← NIL;
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;

CheckNode: PUBLIC PROCEDURE[n: Node] = BEGIN
p: CARDINAL ← LOOPHOLE[n];
IF cdebug THEN BEGIN
        IF n = NIL THEN P.PBug["CheckNode - n is NIL"];
        IF (p < LOOPHOLE[@NodeArray[1],CARDINAL] OR LOOPHOLE[@NodeArray[nNodes-1], CARDINAL] < p) AND
        ~CheckOK THEN BEGIN
                IF n ~= Fail AND n ~= MTSt THEN P.PBug["CheckNode - n is not valid"];
                END;
        IF n.Type = ZERO AND ~ZeroOK THEN P.PBug["CheckNode - n has type zero"];
        IF n.Type = STR AND n.Des.s = file THEN BEGIN
                IF n.str.f = NIL OR n.str.of = NIL OR n.str.f.len = -1 THEN P.PBug["CheckNode - file with NIL"]
                -- ELSE IF n.str.f.bp ~= n THEN P.PBug["CheckNode - bp failed"];
                END;
        END;
END;

GetSpecialNodes: PUBLIC PROCEDURE RETURNS[Node,Node] = BEGIN
RETURN[Fail,MTSt];
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 P.NonZero[p+2,SIZE[PLDefs.NodeRecord]-2] THEN DispDefs.WF1["Is non zero %bB*n",f];
        f ← (p+1)↑;
        ENDLOOP;
DispDefs.WF2["Out of %d, %d are on free list*n",nNodes-1,i];
Mark[NIL];
i ← 0;
FOR j IN [1..nNodes) DO
        IF NodeArray[j].Des.g THEN i ← i + 1;
        ENDLOOP;
DispDefs.WF1["%d marked*n",i];
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
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 => NULL;
STR => IF n.Des.s = cat THEN BEGIN
                -- WHILE n.Type = STR AND n.Des.s = cat DO
                        -- Preorder[n.str.n1,p];
                        -- n ← n.str.n2;
                        -- ENDLOOP;
                -- Preorder[n,p];
                Preorder[n.str.n1,p];
                Preorder[n.str.n2,p]
                END
        ELSE IF StringCollect AND n.Des.s = simp THEN P.LayOutBits[n];
UNARY => Preorder[n.uexp,p];
PFUNC1 => Preorder[n.pexp,p];
SEQOF,SEQOFC => Preorder[n.seqof,p];
OPT => Preorder[n.opt,p];
DELETE => Preorder[n.delete,p];
PATTERN => Preorder[n.pattern,p];
LIST => BEGIN
        -- WHILE n ~= NIL AND n.Type = LIST DO
                -- Preorder[n.listhead,p];
```

```
                -- n ← n.listtail;
                -- ENDLOOP;
        Preorder[n.listhead,p];
        Preorder[n.listtail,p]
        END;
CAT,CATL = > BEGIN Preorder[n.left,p]; Preorder[n.right,p] END;
MATCH = > BEGIN Preorder[n.div,p]; Preorder[n.patt,p] END;
PALT = > BEGIN Preorder[n.alt1,p]; Preorder[n.alt2,p] END;
APPLY,MAPPLY,GOBBLE,ITER = > BEGIN Preorder[n.object,p]; Preorder[n.target,p] END;
FCN = > BEGIN Preorder[n.parms,p]; Preorder[n.fcn,p] END;
ASS = > BEGIN Preorder[n.lhs,p]; Preorder[n.rhs,p] END;
PROG = > BEGIN Preorder[n.prog1,p]; Preorder[n.prog2,p] END;
SEQUENCE = > BEGIN Preorder[n.from,p]; Preorder[n.to,p] END;
PLUS,MINUS = > BEGIN Preorder[n.arg1,p]; Preorder[n.arg2,p] END;
TILDE = > Preorder[n.not,p];
ENDCASE = > P.PBug["Unknown variant"];
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 = > 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"];
[] ← p[n];
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",n,i];
IF i < n THEN P.PBug["Out of core"];
END;


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

```
[] ← Insert["symbol",ZARY,proc,SymbolRoutine,PLDefs.Unbound];
P.SetCursorAmt[0,maxnNodes];
StringCollect ← FALSE;
CheckOK ← ZeroOK ← FALSE;
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;
psptr ← 1;
ignorerrs ← FALSE;
END;

NodeArray ← P.GetCore[maxnNodes * SIZE[PLDefs.NodeRecord]];
END.
```

```
-- string.mesa last edited by schmidt, September 23, 1978  2:37 AM
                                                    DIRECTORY
                                                    PLDefs: FROM "PLDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    IODefs: FROM "IODefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    MiscDefs: FROM "MiscDefs",
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    FilePageUseDefs: FROM "FilePageUseDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    StringDefs: FROM "StringDefs";

string: PROGRAM IMPORTS DispDefs, P:PLDefs, SystemDefs, MiscDefs, StringDefs, FileSystemDefs, FilePageUseDefs
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;
--
nbuf: CARDINAL  = 4;
StringSize: CARDINAL;
sin: CARDINAL;
StringBuffer: ARRAY [0.. nbuf) OF POINTER TO PACKED ARRAY[0..511] OF CHARACTER;
StringPNo: ARRAY [0.. nbuf) OF CARDINAL;
StringFH: ARRAY [0.. nbuf) OF FilePageUseDefs.FileHandle;
lastfh: FilePageUseDefs.FileHandle;
lastp,lastb: CARDINAL;
bptr: CARDINAL;
FSize: CARDINAL  = 30;
FTable: ARRAY[0..FSize) OF PLDefs.OpenFileRecord;
fin: 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;
MinCatSize: CARDINAL  = 20;
MinCoerceSize: CARDINAL  = 100;
SFLimit: CARDINAL  = 512;
SFTable: POINTER TO ARRAY OF PLDefs.StringFileRecord;
sfin: CARDINAL;
LayMax: CARDINAL;
MaybeStringFile: BOOLEAN;

-- one may not assume that for cat STRs n1 is of non-zero length; it may be empty


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
        s[i] ← P.Item[@ns];
        i ← i + 1;
        IF i > = s.maxlength THEN P.RErr["String too big for makestring"];
        ENDLOOP;
s.length ← i;
P.RRS[rr];
```

```
END;

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


Check: PROCEDURE[i: LONG INTEGER] = BEGIN
-- on return, there is space for i more chars in StringArray
IF sin+i < StringSize THEN RETURN;
StringChanged ← TRUE;
[] ← P.GarbageRoutine[NIL,NIL,NIL];                    -- raises GC
IF sin + i >= StringSize THEN P.RErr["Too many strings in memory for program"];
END;

MakeNUM: PUBLIC PROCEDURE[i: LONG INTEGER] RETURNS [n: Node] =
        BEGIN
        rr: Register;
        n ← P.Alloc[[STR,[,,simp,],STR[[simp[sin, 0]]]]];
        rr ← P.MRS[];
        P.R[@n];
        IF i < 0 THEN BEGIN
                i ← -i;
                P.SetSChar[sin,'-];
                sin ← sin + 1;
                END;
        AppendDigits[i];
        n.str.length ← sin - n.str.start;
        StringChanged ← TRUE;
        P.RRS[rr];
        END;

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

MakeInteger: PUBLIC PROCEDURE[n: Node] RETURNS[cnt: LONG INTEGER] = BEGIN
-- this could be done more simply using Start and Next
i,j: CARDINAL;
a,b: LONG INTEGER;
mm: InlineDefs.LongCARDINAL;
s: String;
c: CHARACTER;
wk: STRING ← [PLDefs.sSize];
IF n.Type ~= STR THEN P.RErr["Expecting integer as a string"];
IF P.Length[n] <= 0 THEN RETURN[0];
s ← @n.str;
```

```
j ← 0;
cnt ← 0;
IF n.Des.s = simp THEN BEGIN
        P.GetString[s.start,s.length,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"];
                cnt ← cnt *10 + (wk[i]-'0);
                ENDLOOP;
        END
ELSE IF n.Des.s = file THEN BEGIN
        IF  (c ← get[s.f,TRUE,s.of]) = '- THEN BEGIN j ← 1; c ← get[s.f,TRUE,s.of] END;
        DO
                IF c ~IN ['0..'9] THEN P.RErr["String is not a number"];
                cnt ← cnt *10 + (c-'0);
                c ← get[s.f,TRUE,s.of];
                ENDLOOP;
        END
ELSE IF n.Des.s = cat THEN BEGIN
        a ← MakeInteger[s.n1];
        b ← MakeInteger[s.n2];
        mm ← LOOPHOLE[P.Length[s.n2]];
        THROUGH[1..mm.lowbits] DO
                a ← a * 10;
                ENDLOOP;
        cnt ← a*b;
        END
ELSE P.PBug["unknown variant"];
IF j = 1 THEN cnt ← -cnt;
END;

FileRoutine: PUBLIC PROCEDURE[s: Symbol,name,n2: Node] RETURNS[ans:Node] = BEGIN
fname: STRING ← [30];
curr: CARDINAL;
fb: PLDefs.StringFile;
fs: FileSystemDefs.FileSystem;
fh: FilePageUseDefs.FileHandle;
lp,bp,pg: CARDINAL;
rr: Register ← P.MRS[];
BEGIN
IF name = NIL OR name.Type ~= STR THEN P.RErr["File name must be string"];
IF P.Length[name] > = 30 THEN P.RErr["File name too long"];
ans ← NIL;
P.R[@ans];
MakeString[fname,name];
curr ← 0;
WHILE curr < fin DO
        IF StringDefs.EquivalentString[fname,FTable[curr].filename] THEN EXIT;
        curr ← curr + 1;
        ENDLOOP;
IF curr = fin THEN BEGIN
        fin ← fin + 1;
        IF fin > = FSize THEN P.RErr["Too many open files"];
        fs ← FileSystemDefs.Login['a,NIL,NIL,NIL];
        fh ← FileSystemDefs.Open[fs,fname,FileSystemDefs.OpenMode[read]
                ! FileSystemDefs.FileDoesNotExist = > GOTO end
                ];
        [lp,bp,pg] ← FilePageUseDefs.Measure[fh];
        FTable[curr] ← [fs,fh,lp,pg,bp,SystemDefs.AllocateHeapString[fname.length]];
        StringDefs.AppendString[FTable[curr].filename,fname];
        END;
fb ← NewStringFile[];
fbt ← [0,0,(FTable[curr].lastpage*FTable[curr].pgsize) + FTable[curr].lastchar,NIL];
ans ← P.Alloc[[STR,[,,file,],STR[[file[fb,@FTable[curr]]]]]];
fb.bp ← ans;
IF ans.str.f.len < MinCoerceSize THEN ans ← P.Coerce[ans];
EXITS
```

```
end => ans ← P.Alloc[[FAIL,,FAIL[]]];
END;
P.RRS[rr];
END;

get: PROCEDURE[f: PLDefs.StringFile,adv: BOOLEAN,of: PLDefs.OpenFile] RETURNS[c: CHARACTER] =
BEGIN
bindex: INTEGER;
-- does not call Alloc
IF f.len = 0 THEN RETURN[0C];
-- get the character
IF f.len = -1 THEN P.PBug["Bad get file"];
IF (bindex ← BlockNumber[of.fh,f.pgno]) < 0 THEN BEGIN
        bptr ← bptr + 1;
        IF bptr = nbuf THEN bptr ← 0;
        FilePageUseDefs.ReadPage[of.fh,f.pgno,StringBuffer[bptr]];
        StringFH[bptr] ← of.fh;
        StringPNo[bptr] ← f.pgno;
        bindex ← bptr;
        END;
c ← StringBuffer[bindex][f.inx];
IF ~adv THEN RETURN[c];
f.inx ← f.inx + 1;
f.len ← f.len - 1;
IF f.inx >= of.pgsize THEN BEGIN
        f.inx ← 0;
        f.pgno ← f.pgno + 1;
        END;
IF (f.pgno > of.lastpage OR (f.pgno = of.lastpage AND f.inx >= of.lastchar)) AND f.len ~= 0 THEN
P.PBug["length wrong"];
RETURN[c];
END;

BlockNumber: PROCEDURE[fh: FilePageUseDefs.FileHandle,p: CARDINAL] RETURNS[INTEGER] = BEGIN
i: CARDINAL;
IF lastfh = fh AND lastp = p THEN RETURN[lastb];
FOR i IN [0.. bptr] DO
        IF fh = StringFH[i] AND p = StringPNo[i] THEN BEGIN
                lastfh ← fh;
                lastp ← p;
                lastb ← i;
                RETURN[i];
                END;
        ENDLOOP;
lastfh ← NIL;
RETURN[-1];
END;

StringConcat: PUBLIC PROCEDURE[i, j: Node] RETURNS [n: Node] = BEGIN
f: PLDefs.StringFile;
a: CARDINAL;
aa,bb: LONG INTEGER;
g,h,in: CARDINAL;
wk: STRING ← [PLDefs.sSize];
s: String;
IF i.Type#STR OR j.Type#STR THEN P.PBug["Bad concat"];
-- try to avoid making CAT node
IF i.Des.s = simp AND j.Des.s = simp THEN BEGIN
        g ← i.str.length;
        h ← j.str.length;
        IF g = 0 THEN RETURN[j];
        IF h = 0 THEN RETURN[i];
        IF g + h < MinCatSize THEN BEGIN
                -- too small, make a new STR
                Check[g+h];
                in ← sin;
                s ← @i.str;
```

```
                P.GetString[s.start,g,wk];
                P.SetString[sin,wk];
                sin ← sin + g;
                s ← @j.str;
                P.GetString[s.start,h,wk];
                P.SetString[sin,wk];
                sin ← sin + h;
                StringChanged ← TRUE;
                RETURN[P.Alloc[[STR,[,,simp,],STR[[simp[in,sin-in]]]]]];
                END;
        a ← i.str.start + g;
        IF a = j.str.start THEN RETURN[P.Alloc[[STR,[,,simp,],STR[[simp[i.str.start,g + h]]]]]];
        END
ELSE IF i.Des.s = file AND j.Des.s = file THEN BEGIN
        f ← i.str.f;
        IF f.len = -1 THEN P.PBug["Bad strcat file"];
        aa ← (f.pgno*i.str.of.pgsize) + f.inx + f.len;
        f ← j.str.f;
        bb ← (f.pgno*i.str.of.pgsize) + f.inx;
        IF aa = bb THEN BEGIN
                n ← P.CopyTree[i];
                n.str.f.len ← n.str.f.len + j.str.f.len;
                RETURN[n];
                END;
        END;
n ← P.Alloc[[STR,[,,cat,],STR[[cat[i,j]]]]];
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"];
IF n.Des.s = simp THEN RETURN[n];
rr ← P.MRS[];
ii ← P.Length[n];
Check[ii];
ns ← NewStream[n];
P.R[@ns.node];
m ← LOOPHOLE[ii,InlineDefs.LongCARDINAL].lowbits;
FOR j IN [0..m) DO
        P.SetSChar[sin + j,Item[@ns]];
        ENDLOOP;
ans ← P.Alloc[[STR,[,,simp,],STR[[simp[sin,m]]]]];
sin ← sin + m;
StringChanged ← TRUE;
P.RRS[rr];
END;

Sub: PUBLIC PROCEDURE[n: Node,inx: LONG INTEGER] RETURNS[CHARACTER] = BEGIN
i: CARDINAL;
ii: LONG INTEGER;
f: PLDefs.StringFileRecord;
len: LONG INTEGER;
DO
        IF n = NIL OR n.Type ~= STR OR inx < 0 THEN P.PBug["bad sub"];
        IF n.Des.s = simp THEN BEGIN
                IF inx > = n.str.length THEN P.RErr["bad length - sub"];
                i ← LOOPHOLE[inx,InlineDefs.LongCARDINAL].lowbits;
                RETURN[P.GetSChar[n.str.start + i]];
                END
        ELSE IF n.Des.s = file THEN BEGIN
                f ← n.str.ft;
                IF f.len = -1 THEN P.PBug["Bad sub file"];
```

```
                IF inx > = f.len THEN P.PBug["bad length - sub"];
                ii ← f.inx + inx;
                f.pgno ← f.pgno + LOOPHOLE[ii/n.str.of.pgsize,LongCARDINAL].lowbits;
                f.inx ← LOOPHOLE[ii,LongCARDINAL].lowbits MOD n.str.of.pgsize;
                RETURN[get[@f,FALSE,n.str.of]];
                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"];
        ENDLOOP;
END;

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

Item: PUBLIC PROCEDURE[s: Stream] RETURNS[c: CHARACTER] = BEGIN
f: PLDefs.StringFileRecord;
ii: LONG INTEGER;
i: CARDINAL;
n:Node ← s.node;
IF n = NIL THEN RETURN[0C];
IF n.Type ~ = STR THEN P.PBug["Item expects a STR"];
WHILE n.Des.s = cat DO
        ii ← P.Length[n.str.n1];
        IF s.posn > = ii THEN BEGIN
                s.posn ← 0;
                n ← s.node ← n.str.n2;
                END
        ELSE BEGIN
                n ← n.str.n1;
                EXIT;
                END;
        ENDLOOP;
-- n is type file or simp
IF n.Des.s = simp THEN BEGIN
        i ← LOOPHOLE[s.posn,InlineDefs.LongCARDINAL].lowbits;
        c ← P.GetSChar[n.str.start + i];
        END
ELSE IF n.Des.s = file THEN BEGIN
        f ← n.str.ft;
        IF f.len = -1 THEN P.PBug["bad item file"];
        IF s.posn > = f.len THEN P.PBug["Bad length - item"];
        ii ← f.inx + s.posn;
        IF ii > 65000 THEN BEGIN
                f.pgno ← f.pgno + LOOPHOLE[(ii/n.str.of.pgsize),LongCARDINAL].lowbits;
                f.inx ← LOOPHOLE[ii,LongCARDINAL].lowbits MOD n.str.of.pgsize;
                END
        ELSE BEGIN
                i ← LOOPHOLE[ii,LongCARDINAL].lowbits;
                f.pgno ← f.pgno + i/n.str.of.pgsize;
                f.inx ← i MOD n.str.of.pgsize;
```

```
                END;
        c ← get[@f,FALSE,n.str.of];
        END
ELSE P.PBug["bad str type item"];
s.posn ← s.posn + 1;
IF s.node.Des.s ~= cat AND P.Length[s.node] <= s.posn THEN s.node ← NIL;
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"];
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"];
END;

StringGC: PUBLIC PROCEDURE = BEGIN
StringBitMap ← P.GetCore[StringSize/16];
Range ← P.GetCore[RangeSize * SIZE[RangeRecord]];
MiscDefs.Zero[StringBitMap,StringSize/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, ",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"];
                        run ← TRUE;
                        END;
                P.SetSChar[j,P.GetSChar[i]];
                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"];
sin ← j;
IF cdebug THEN DispDefs.WF1["SNewe %u, ",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 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"];
IF j > = inx THEN P.PBug["transfer cant happen"];
-- 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"];
END;

NewStringFile: PUBLIC PROCEDURE RETURNS[PLDefs.StringFile] = BEGIN
i: CARDINAL ← 1;
-- note that this is linear search which is excruciatingly slow
IF ~MaybeStringFile AND sfin + 2 > = SFLimit THEN
        [] ← P.GarbageRoutine[NIL,NIL,NIL];                    -- force G.C.
IF MaybeStringFile THEN WHILE i < sfin DO
        IF SFTable[i].len = -1 THEN EXIT;
        i ← i + 1;
        ENDLOOP
ELSE i ← sfin;
IF i = sfin THEN BEGIN
```

```
        sfin ← sfin + 1;
        MaybeStringFile ← FALSE;
        IF sfin > = SFLimit THEN P.RErr["Too many file pieces"];
        END;
RETURN[@SFTable[i]];
END;

FreeStringFile: PUBLIC PROCEDURE[f: PLDefs.StringFile] = BEGIN
IF f = NIL THEN P.PBug["f is nil - freestringfile"];
IF f.len = -1 THEN P.PBug["f.len is -1"];
f.len ← -1;
MaybeStringFile ← TRUE;
END;

StringDebugging: PUBLIC PROCEDURE = BEGIN
END;

StringSetup: PUBLIC PROCEDURE = BEGIN
-- must follow vmsetup
i: CARDINAL;
sin ← 1;
[] ← P.Insert["file",ZARY,proc,FileRoutine,PLDefs.Unbound];
[] ← P.Insert["close",ZARY,proc,CloseRoutine,PLDefs.Unbound];
fin ← 0;
FOR i IN [0..FSize) DO
        FTable[i].fh ← NIL;
        ENDLOOP;
FOR i IN [0..nbuf) DO
        StringFH[i] ← NIL;
        ENDLOOP;
StringSize ← P.GetMaxStr[];
StringChanged ← FALSE;
sfin ← 1;
MaybeStringFile ← FALSE;
lastfh ← NIL;
END;

CloseRoutine: PROCEDURE[sym:Symbol,n1,n2: Node] RETURNS[Node] = BEGIN
i: CARDINAL;
FOR i IN [0..fin) DO
        IF FTable[i].fh ~= NIL THEN FilePageUseDefs.Close[FTable[i].fh];
        FTable[i].fh ← NIL;
        ENDLOOP;
RETURN[NIL];
END;

StringCleanup: PUBLIC PROCEDURE = BEGIN
[] ← CloseRoutine[NIL,NIL,NIL];
END;

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



SFTable ← P.GetCore[SFLimit * SIZE[PLDefs.StringFileRecord]];
FOR bptr IN [0.. nbuf) DO
        StringBuffer[bptr] ← P.GetCore[256];
        ENDLOOP;
END.
```

```
-- sup.mesa last edited by schmidt, September 23, 1978  4:02 PM
                                                    DIRECTORY
                                                    PLDefs: FROM "PLDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    ImageDefs: FROM "ImageDefs",
                                                    FrameDefs: FROM "FrameDefs",
                                                    IODefs: FROM "IODefs",
                                                    MiscDefs: FROM "miscdefs",
                                                    AltoFileDefs: FROM "AltoFileDefs",
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    FilePageUseDefs: FROM "FilePageUseDefs",
                                                    StreamDefs: FROM "StreamDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    StringDefs: FROM "StringDefs",
                                                    TimeDefs: FROM "TimeDefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    DoubleDefs: FROM "DoubleDefs",
                                                    RandomDefs: FROM "RandomDefs",
                                                    SegmentDefs: FROM "SegmentDefs";

sup: PROGRAM IMPORTS D1:DispDefs, StreamDefs, SegmentDefs, MiscDefs, StringDefs, ImageDefs, P:PLDefs, FrameDefs,
TimeDefs, RandomDefs, SystemDefs EXPORTS PLDefs SHARES RandomDefs  = 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;
-- for AmbushKeyStream
comStream, keyStream: StreamDefs.StreamHandle;
savedKeyStreamGet: PROCEDURE [StreamDefs.StreamHandle] RETURNS[UNSPECIFIED];
charWaiting: BOOLEAN;
firstChar: CHARACTER;
--
SaveCurmap: ARRAY[0..15] OF CARDINAL;
Curmap: POINTER TO ARRAY OF CARDINAL = LOOPHOLE[431B];
cntr: CARDINAL;
oldnl: CARDINAL ← 60000;                              -- invalid val

ListRecord: TYPE = RECORD[old,new: Node, pref: LONG INTEGER,st: PLDefs.StreamRecord, number: BOOLEAN];
List: POINTER TO ARRAY OF ListRecord;
lin: CARDINAL;
sgs: RandomDefs.SimpleGeneratorState;
ascii: BOOLEAN;


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

SetCursorAmt: PUBLIC PROCEDURE[val,max: CARDINAL] = BEGIN
nl,i: CARDINAL;
nl ← (val*14)/max;
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 177777B ELSE 100001B;
        ENDLOOP;
END;
AmbushKeyStream: PUBLIC PROCEDURE =
```

```
    BEGIN
    cfa: POINTER TO AltoFileDefs.CFA = MiscDefs.CommandLineCFA[];
      -- Points to where the system stopped reading the command line.
    comfh: SegmentDefs.FileHandle ← SegmentDefs.InsertFile[@cfa.fp, SegmentDefs.Read];
    ch: CHARACTER;

    comStream ← StreamDefs.CreateByteStream[comfh, SegmentDefs.Read];
    StreamDefs.JumpToFA[comStream, @cfa.fa];
        BEGIN
        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 => charWaiting ← FALSE;
        END;

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

AmbushedGet: PROCEDURE[StreamDefs.StreamHandle] RETURNS[UNSPECIFIED] =
    BEGIN
    ch: CHARACTER;
    IF charWaiting THEN
        BEGIN
        charWaiting ← FALSE;
        RETURN[firstChar];
        END;
    WHILE NOT comStream.endof[comStream]
        DO
        ch ← comStream.get[comStream];
        IF ch # IODefs.CR THEN RETURN[ch];
        ENDLOOP;
    comStream.destroy[comStream];
    comStream ← NIL;
    keyStream.get ← savedKeyStreamGet;
    RETURN[keyStream.get[keyStream]];
    END;

Editor: PUBLIC PROCEDURE[edfile: STRING] = BEGIN
str: STRING ← [600];
t: Node ← NIL;
rr: Register ← P.MRS[];
P.R[@t];
D1.WriteToString[str];
D1.WF2["Bravo/n %s;basicmesa.image poplar $%s'*033*n",edfile,edfile];
[] ← D1.SetWriteProcedure[D1.MyWriteProcedure];
t ← P.MakeSTR[str];
WriteRem[t];
ImageDefs.StopMesa[];
END;

Execute: PUBLIC PROCEDURE[str: Node] = BEGIN
OPEN TimeDefs, InlineDefs;
pt,npt: PackedTime;
diff: LongCARDINAL;
i: CARDINAL;
WriteRem[str];
pt ← CurrentDayTime[];
ImageDefs.MakeCheckPoint["PoplarCheckPoint.Image$"];
npt ← CurrentDayTime[];
diff ← DoubleDefs.DSub[npt,pt];
i ← LOOPHOLE[diff,LongCARDINAL].lowbits;
D1.WF0["We think you should say "];
D1.WF0[IF i < 12 THEN "yes*n" ELSE "no*n"];
```

```
D1.WF0["Quit? "];
IF D1.Confirm[] THEN ImageDefs.StopMesa[];
D1.WF0["*n"];
END;

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

DefaultName: PUBLIC 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"];
END;

NonZero: PUBLIC 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;

FlushCodeSegments: PROCEDURE = BEGIN

ClearItOut: PROCEDURE[gfh: FrameDefs.GlobalFrameHandle] RETURNS[BOOLEAN] = BEGIN
IF gfh.codesegment.lock = 0 THEN FrameDefs.SwapOutCode[gfh];
RETURN[FALSE];
END;

[] ← FrameDefs.EnumerateGlobalFrames[ClearItOut];
END;

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];
IF input = NIL THEN P.RErr["Sort expects a list of strings"];
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].st.node ← NIL;
                List[m].number ← FALSE;
```

```
                P.R[@List[m].st.node];
                ENDLOOP;
        [] ← P.MapList[input,SR];
        -- setting up simple random number generaor
        sgs.x ← [0,1];
        sgs.c ← [0,0];
        sgs.a ← RandomDefs.RandomSeed[[0,0]];
        RandomDefs.InitializeSimpleRandom[@sgs];
        QuickSort[0,lin-1];                    -- sort the list
        j ← P.Alloc[[LIST,,LIST[List[lin-1].old,NIL]]];
        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] RETURNS[Node] = BEGIN
s: STRING;
i: CARDINAL;
num: BOOLEAN;
ns: PLDefs.StreamRecord;
c: CHARACTER;
IF n = NIL THEN RETURN[n];
List[lin].old ← n;
IF n.Type = LIST THEN n ← n.listhead;
List[lin].new ← n;
P.R[@List[lin].new];
IF List[lin].new.Type ~= STR THEN P.RErr["Sort expects a list of strings"];
ns ← List[lin].st  ← 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];
IF ~ascii THEN BEGIN
        num ← TRUE;
        FOR i IN [0..3] DO
                num ← num AND (s[i] IN ['0..'9] OR s[i] = 0C OR s[i] = '-);
                ENDLOOP;
        WHILE num AND ns.node ~= NIL DO
                c ← P.Item[@ns];
                num ← '0 < = c AND c < = '9;
                ENDLOOP;
        IF num THEN BEGIN
                List[lin].number ← TRUE;
                List[lin].pref ← P.MakeInteger[List[lin].new];
                END;
        END;
lin ← lin + 1;
RETURN[n];
END;

QuickSort: PROCEDURE[l,r: INTEGER] = BEGIN
s,i,j,pivot: INTEGER;
-- Quicksort from List[l].new to List[r].new
IF l > = r THEN RETURN;
IF r = l + 1 THEN BEGIN
        IF ncomp[l,r] > 0 THEN nexch[l,r];
        RETURN;
        END;
s ← RandomDefs.SimpleRandom[@sgs,r-l-1] + l + 1;
-- D2.WF3["Q %d %d %d*n",l,s,r];
```

```
i ← l;
j ← r;
pivot ← s;
WHILE i < j DO
        WHILE j > = l AND ncomp[j,pivot] > = 0 DO j ← j -1 ENDLOOP;
        WHILE i< =r AND ncomp[i,pivot] < 0 DO i ← i + 1 ENDLOOP;
        IF i < j THEN BEGIN
                nexch[i,j];
                IF i=s THEN s ← j
                ELSE IF j = s THEN s ← i;
                END;
        ENDLOOP;
-- strings from l to i are < s, from j+1 to r are < = s
IF i ~= l THEN BEGIN
        QuickSort[l,i];
        QuickSort[j + 1,r];
        END
ELSE BEGIN
        nexch[j + 1,s];
        QuickSort[j + 2,r];
        END;
END;

nexch: PROCEDURE[i,j: INTEGER] = BEGIN
a,b: ListRecord;
a ← List[i];
b ← List[j];
List[j] ← a;
List[i] ← b;
END;

ncomp: PROCEDURE[i,j: INTEGER] RETURNS[res: INTEGER] = BEGIN
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[1]
        ELSE IF List[i].pref < List[j].pref THEN RETURN[-1]
        ELSE RETURN[0];
        END;
IF ~List[i].number AND ~List[j].number THEN BEGIN
        res ← USC[List[i].pref,List[j].pref];
        IF res ~= 0 THEN RETURN;
        END;
n1 ← List[i].st;
n2 ← List[j].st;
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;
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.highbits > j.highbits THEN RETURN[1];
IF i.highbits < j.highbits THEN RETURN[-1];
-- test lsbits
IF i.lowbits > j.lowbits THEN RETURN[1];
IF i.lowbits < j.lowbits THEN RETURN[-1];
RETURN[0];
END;

USortRoutine: PROCEDURE[sym:Symbol,input,func: Node] RETURNS[ans:Node] = BEGIN
-- unary
rr: Register ← P.MRS[];
n: Node;
pans: POINTER TO Node;
ans ← NIL;
P.R2[@input,@ans];
input ← ASortRoutine[sym,input,func];
n ← input;
ans ← P.Alloc[[LIST,,LIST[n.listhead,NIL]]];
pans ← @ans.listtail;
WHILE n ~= NIL AND n.Type = LIST AND n.listtail.Type = LIST DO
        IF ~Equal[n.listhead,n.listtail.listhead] THEN BEGIN
                pans↑ ← P.Alloc[[LIST,,LIST[n.listtail.listhead,NIL]]];
                pans ← @pans.listtail;
                END;
        n ← n.listtail;
        ENDLOOP;
P.RRS[rr];
END;

Equal: PROCEDURE[i,j: Node] RETURNS[res: BOOLEAN] = BEGIN
rr: Register ← P.MRS[];
a,b: PLDefs.StreamRecord;
len: LONG INTEGER;
IF i.Type ~= STR OR j.Type ~= STR THEN P.RErr["USort only compares lists of strings"];
a ← P.NewStream[i];
b ← P.NewStream[j];
P.R2[@a.node,@b.node];
res ← TRUE;
len ← P.LengthStream[@a];
IF P.LengthStream[@b] ~= len THEN res ← FALSE
ELSE WHILE a.node ~= NIL DO
        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",ZARY,proc,SortRoutine,PLDefs.Unbound];
[] ← P.Insert["usort",ZARY,proc,USortRoutine,PLDefs.Unbound];
[] ← P.Insert["asort",ZARY,proc,ASortRoutine,PLDefs.Unbound];
SupReset[];
END;

SupReset: PUBLIC PROCEDURE = BEGIN
END;

FOR cntr IN [0..15] DO
        SaveCurmap[cntr] ← Curmap[cntr];
        ENDLOOP;
END.
```

```
-- VM.mesa last edited by schmidt, September 18, 1978  10:56 PM
                                                    DIRECTORY
                                                    PLDefs: FROM "PLDefs",
                                                    DispDefs: FROM "DispDefs",
                                                    IODefs: FROM "IODefs",
                                                    InlineDefs: FROM "InlineDefs",
                                                    MiscDefs: FROM "MiscDefs",
                                                    FileSystemDefs: FROM "FileSystemDefs",
                                                    FilePageUseDefs: FROM "FilePageUseDefs",
                                                    SystemDefs: FROM "SystemDefs",
                                                    StringDefs: FROM "StringDefs";

VM: PROGRAM IMPORTS P:PLDefs, FileSystemDefs, FilePageUseDefs, MiscDefs EXPORTS PLDefs  = BEGIN
--
bSize: CARDINAL = 512;
nPages: CARDINAL = 20;
nbuf: CARDINAL = 10;
lnPNo: ARRAY [0..nbuf) OF CARDINAL;
lnPMod: ARRAY [0..nbuf) OF BOOLEAN;
StringArray: ARRAY [0..nbuf) OF PACKED ARRAY [0..bSize] OF CHARACTER;
maxStringSize: CARDINAL;
vmfs: FileSystemDefs.FileSystem ← NIL;
vmfile: FilePageUseDefs.FileHandle ← NIL;
bptr: CARDINAL;
lastval,lastinx: CARDINAL ← 60000;

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

SetSChar: PUBLIC PROCEDURE[in: CARDINAL,c: CHARACTER] = BEGIN
b,n,p: CARDINAL;
p ← in/bSize;
n ← in MOD bSize;
-- p is page # [0..nPages), n is char on page [0..bSize)
b ← GetPage[p];
StringArray[b][n] ← c;
lnPMod[b] ← TRUE;
END;

VMSetup: PUBLIC PROCEDURE = BEGIN
i,lp,bp,bytespage: CARDINAL;
vmfs ← FileSystemDefs.Login['a,NIL,NIL,NIL];
vmfile ← FileSystemDefs.Open[vmfs,"Poplar.VMFile$", FileSystemDefs.OpenMode[create] !
FileSystemDefs.FileAlreadyExists => RESUME];
[lp,bp,bytespage] ← FilePageUseDefs.Measure[vmfile];
IF bytespage ~= bSize THEN P.PBug["wrong pagesize"];
IF lp ~= nPages - 1 OR bp ~= 0 THEN BEGIN
        MiscDefs.Zero[BASE[StringArray[0]],(bSize + 1)/2];
        FOR i IN [0..nPages) DO
                FilePageUseDefs.WritePage[vmfile,i,BASE[StringArray[0]]];
                ENDLOOP;
        END;
FilePageUseDefs.SetLength[vmfile,nPages,0];
FOR i IN [0..nbuf) DO
        lnPNo[i] ← 60000;
        lnPMod[i] ← FALSE;
        ENDLOOP;
bptr ← 0;
lastval ← 60000;
maxStringSize ← bSize*nPages;
END;

VMCleanup: PUBLIC PROCEDURE = BEGIN
FilePageUseDefs.Close[vmfile];
```

```
FileSystemDefs.Logout[vmfs];
END;

GetPage: PROCEDURE[p: CARDINAL] RETURNS[CARDINAL] = BEGIN
-- return the buffer # with page p in it
i: CARDINAL;
IF lastval = p THEN RETURN[lastinx];
FOR i IN [0..nbuf) DO
        IF p = InPNo[i] THEN BEGIN
                lastval ← p;
                lastinx ← i;
                RETURN[i];
                END;
        ENDLOOP;
-- not in core, must go get it
bptr ← bptr + 1;
IF bptr > = nbuf THEN bptr ← 0;
lastval ← 60000;
IF InPMod[bptr] THEN FilePageUseDefs.WritePage[vmfile,InPNo[bptr],BASE[StringArray[bptr]]];
FilePageUseDefs.ReadPage[vmfile,p,BASE[StringArray[bptr]]];
InPNo[bptr] ← p;
InPMod[bptr] ← FALSE;
RETURN[bptr];
END;

GetString: PUBLIC PROCEDURE[st,len: CARDINAL,s: STRING] = BEGIN
i,b,p,p1,p2,n1,n2,k,m: CARDINAL;
IF st+len > maxStringSize OR len > = s.maxlength THEN P.PBug["bad GetString"];
s.length ← len;
i ← st+len - 1;
p1 ← st/bSize;
p2 ← i/bSize;
n1 ← st MOD bSize;
n2 ← i MOD bSize;
b ← GetPage[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 (n1 .. n2) DO
        b ← GetPage[p];
        FOR i IN [0..bSize) DO
                s[k] ← StringArray[b][i];
                k ← k + 1;
                ENDLOOP;
        ENDLOOP;
b ← GetPage[p2];
FOR i IN [0..n2] DO
        s[k] ← StringArray[b][i];
        k ← k + 1;
        ENDLOOP;
END;

SetString: PUBLIC PROCEDURE[st:CARDINAL,s: STRING] = BEGIN
i,b,p,p1,p2,n1,n2,k,m: CARDINAL;
IF st+s.length > maxStringSize THEN P.PBug["bad SetString"];
i ← st+s.length - 1;
p1 ← st/bSize;
p2 ← i/bSize;
n1 ← st MOD bSize;
n2 ← i MOD bSize;
b ← GetPage[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;
```

```
InPMod[b] ← TRUE;
IF p1 = p2 THEN RETURN;
FOR p IN (n1 .. n2) DO
        b ← GetPage[p];
        FOR i IN [0..bSize) DO
                StringArray[b][i] ← s[k];
                k ← k + 1;
                ENDLOOP;
        InPMod[b] ← TRUE;
        ENDLOOP;
b ← GetPage[p2];
FOR i IN [0..n2] DO
        StringArray[b][i] ← s[k];
        k ← k + 1;
        ENDLOOP;
InPMod[b] ← TRUE;
END;

GetMaxStr: PUBLIC PROCEDURE RETURNS[CARDINAL] = BEGIN
-- vmsetup must have been previously called
RETURN[maxStringSize];
END;

END.
```

```
--In <mesalib>WF.DM, file WF.mesa - WriteFormatted implementation
-- last edit Schmidt, August 16, 1978  10:44 PM

DIRECTORY
    IODefs: FROM "IODefs" USING [WriteChar],
    StringDefs: FROM "StringDefs" USING [AppendChar, StringToDecimal],
    ForgotDefs: FROM "ForgotDefs" USING [LowHalf],
    InlineDefs: FROM "InlineDefs",
    DispDefs: FROM "DispDefs",
    WFDefs: FROM "WFDefs" USING [Unbound];

WF: PROGRAM IMPORTS IODefs, StringDefs EXPORTS DispDefs =
BEGIN
--
LongCARDINAL: TYPE = InlineDefs.LongCARDINAL;

procArray: ARRAY[1 .. 26] OF PROCEDURE[UNSPECIFIED,STRING];
saveProcArray: ARRAY[1 .. 26] OF PROCEDURE[UNSPECIFIED,STRING];
WC: PROCEDURE[CHARACTER];
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] ~= WFDefs.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;

SetWriteProcedure: PUBLIC PROCEDURE[p: PROCEDURE[CHARACTER]] RETURNS [op: PROCEDURE[CHARACTER]] =
    BEGIN
    op ← WC;
    WC ← p;
    RETURN[op];
    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 ← ForgotDefs.LowHalf[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;

lnstring: 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];
    lnstring[d,s,10];
```

```
    printit[s,form];
    END;

LRoutine: PROCEDURE[d: UNSPECIFIED, form: STRING] =
    BEGIN
    D: LongCARDINAL;
    p: POINTER TO LongCARDINAL;
    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] ← WFDefs.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;
[] ← SetWriteProcedure[IODefs.WriteChar];
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

```
--FilePageUseDefs.Mesa


FilePageUseDefs: DEFINITIONS = BEGIN

FileHandle:TYPE = POINTER TO RECORD [CARDINAL,CARDINAL];
PageNumber:TYPE = CARDINAL;

ReadPage:PROCEDURE[FileHandle,PageNumber,POINTER];
WritePage:PROCEDURE[FileHandle,PageNumber,POINTER];
Close:PROCEDURE [FileHandle];
Measure:PROCEDURE[FileHandle]
   RETURNS [lastPage,nextByteOnPage,pageSize: CARDINAL];
SetLength: PROCEDURE
  [fh:FileHandle,lastPage:PageNumber,nextByteOnPage: CARDINAL];
Name: PROCEDURE[FileHandle] RETURNS [STRING];
         -- string good as long as file is open

EndOfFile: ERROR; -- comes only from reads
ReadOnly: ERROR; -- comes only from writes
BadFileHandle: ERROR; -- consistency check

END.
```

Notes:

Any implementation has a fixed page size (often 512 bytes) which may be discoverd by calling Measure.

Pages are numbered starting with 0.

Bytes on a page are numbered starting with 0.

Measure returns [n, c, bytesPerPage] for a file with n*bytesPerPage + c bytes.

Measure returns [n, 0, bytesPerPage] for a file with exactly n pages.

Measure returns [0, 0, bytesPerPage] for an empty file.

WritePage extends the file if necessary.

ReadPage[fh, n, p] fails iff n > lastPage or n = lastPage and nextByteOnPage = 0 where [lastPage, nextByteOnPage, ps] = MeasureFile[fh].

Name returns a string suitable for printing in the case of mishap. Typically it is the full file name of the file. The space for the string will be deallocated when the file is closed.

```
--FileSystemDefs.Mesa
                                        DIRECTORY FilePageUseDefs: FROM "FilePageUseDefs";

FileSystemDefs: DEFINITIONS = BEGIN

FileSystem:TYPE = POINTER TO RECORD[CARDINAL,CARDINAL]; -- just a unique type

Login: PROCEDURE[type: CHARACTER, serverMachine, name, password: STRING] RETURNS [FileSystem];
Logout: PROCEDURE[FileSystem];
Delete: PROCEDURE [FileSystem, STRING];
Open: PROCEDURE
      [fs: FileSystem, fileName: STRING, mode: OpenMode]
        RETURNS [FilePageUseDefs.FileHandle];
OpenMode: TYPE = {read, write, create};


NextDirectoryName: PROCEDURE [FileSystem, STRING];


FileDoesNotExist: SIGNAL; -- open for reading or writing, resuming creates file
FileAlreadyExists: SIGNAL; -- create and open on Alto, resuming causes overwrite
BadFileSystem: ERROR;

END...
```

Notes:
Login and Logout open and close Juniper transactions. If you forget to Logout, none of the writes you performed will take effect. However, if your Mesa program terminates normally (including a quit from the debugger) a logout will be issued automatically by a clean up procedure.

The file name passed to delete must have an explicit version number.

The effect of the OpenMode is as follows:

- read: The signal FileDoesNotExist may be raised. Resuming the signal will cause a file to be created. In any case, the open file is flagged as being read only which will cause WritePage (see FilePageUseDefs) to complain.
- write: The most recent version of the file will be sought. The signal FileDoesNotExist may be raised. Resuming the signal will cause a file to be created. The file may be read or written.
- create: A new version of the file will be created. If a new version cannot be created FileAlreadyExists will be raised. Resuming the signal will cause opening of the latest existing version. The file is open for writing.

NextDirectoryName overwrites its string parameter with the name from the directory which comes next after it. In Juniper, it means the alphabetically next file or the next lowest version; if no version number is given you get the highest. The empty string is defined to be the least file. FileDoesNotExist is used to signal the end of the directory. If it is ever implemented on the Alto, "next" means the file that happens to be next in the directory.

Unless the file name supplied to a procedure starts with " < " a standard prefix is assumed. Initially, it is the login name, but it may be altered with SetNamePrefix.

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

AFiles: PROGRAM
        IMPORTS crD, SY: SystemDefs, ST: StringDefs
        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
        IF erc = ovD.illegalFilename THEN ERROR IllegalFileName;
        IF erc = ovD.fileNotFound THEN ERROR FileDoesNotExist;
        IF erc # ovD.ok THEN ERROR DiskError[erc];
        END;

Close: PUBLIC PROCEDURE[fh: FileHandle] =
        BEGIN
        f: File = LOOPHOLE[fh];
        Check[crD.CloseFile[f.ufh]];
        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↑ ← [[[name], [password]], ""];
        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.