KNOWN COMPILER BUGS
*** Compilers 3/7/76

1. Labels in manifest procedures are liable not to work.

2. let F[...] be valof ...
   produces a confusing report.

3. manifest
   $m
   a[x] = 0
   b[y] = y+1
   c = a[b[0]]
   $m
   produces report about invalid constant expression.

4. Some rather silly reports are produced, especially when inserted
   semicolons and dos are involved.

5. get file ending with single unused block definition loses GET
   symbol and reports from Pass 2 and later are reported as
   from wrong file.

6. Error reports sometimes get lost (ignord because Char is ERROR)
   when they shouldn't.

7. Error on line immediately following global definition causes
   global definition to be lost and silly reports about which
   globals set.

****

PROPOSED COMPILER CHANGES
*** Compilers 3/7/76

1. Remove inaccessible code (between exit or unconditional hop and
   next hop point).

2. Remove inaccessible (non-manifest) procedures.
   (This would confuse 'Map', so wait until Type 3 code
   is well established.)

3. Combine consecutive INC instructions.

4. Deal with getting the same file of manifest declarations twice.

5. Optimise code in situations like:
   NO;   INCO
   FNEXIT; INCO

6. Match strings like names to avoid duplication in code.

7. Report global clashes.

8. Allow `` to denote null character.

9. String quotes to appear at beginning of second and subsequent
   lines of strings.

****

```
Common Declarations
 *** Compilers 15/2/77


    manifest
    §m
        CG1 = 400
 5      CG2 = 475
    $m

    global
    §g
10      Ch             : CG1
        ChType         : CG1 + 1
        PassNo         : CG1 + 2
        LineNo         : CG1 + 3
        GetNo          : CG1 + 4
15      ReportNo       : CG1 + 5
        SN             : CG1 + 6
        GN             : CG1 + 7
        HN             : CG1 + 8
        MFN            : CG1 + 9
20      TableMax       : CG1 + 10
        StructureVec   : CG1 + 11
        PresentLevel   : CG1 + 12
        NILNAME        : CG1 + 13
        OUTSIDE        : CG1 + 14
25      NameVec        : CG1 + 15
        LocalGenNo     : CG1 + 16
        Send           : CG1 + 17
        SWN            : CG1 + 18
        StructureLPtr  : CG1 + 19
30      StructureNPtr  : CG1 + 20
        Peep           : CG1 + 21
        NameMap        : CG1 + 22
        SpecSend       : CG1 + 23
        NullNameNo     : CG1 + 24
35  $g

    global
    §g
        PP             : CG2
40      Pass1          : CG2 + 1
        Pass2          : CG2 + 2
        Pass3          : CG2 + 3
        Pass4          : CG2 + 4
        PrintReports   : CG2 + 5
45      Error          : CG2 + 6
        CompilerError  : CG2 + 7
        SCI            : CG2 + 8
        ErrorStream    : CG2 + 9
        GlobList       : CG2 + 10
50      Mode           : CG2 + 11

        NameSize       : CG2 + 12
        NumberSize     : CG2 + 13
        TextSize       : CG2 + 14
55      StringSize     : CG2 + 15
        StructureSize  : CG2 + 16
        AppSize        : CG2 + 18
        CodeSize       : CG2 + 19
        ReportSize     : CG2 + 20
60      GlobSize       : CG2 + 21
    $g
```

```
   manifest
   §m
65     DUMMY      = 0
       ERROR      = 81000
       ENDPROG    = 81001

       ENDCH      = 8377
70     ENDOFSTRCH = -1

       UNSET      = 32766
       ENORMOUS   = 32767

75     COMPERROR  = 4
       VERYHARD   = 3
       HARD       = 2
       SOFT       = 1
       NOERROR    = 0
80
       COMMASK    = 8160000
       TYPEMASK   = 8174000
       VALMASK    = 8003777

85     TypePart[C] = (C ∧ COMMASK) = 0 → C, C ∧ TYPEMASK

       ValPart[C]  = C ∧ VALMASK

       WorkFile[S] = LookUp[S, 'Work', SCI]
90 $m

****
```

```
Declarations for PP
 *** Compilers 18/9/76


    manifest
    §m
        G = 430
 5      NOTVEC = -30000
    $m

    global
    §g
10      NumFn         : G
        StringRt      : G + 1
        CharacterRt   : G + 2
        NLRt          : G + 3
        GetRt         : G + 4
15      FillProcVec   : G + 5
        NormalSend    : G + 6
        SendTables    : G + 7
        DenestGets    : G + 8
    $g
20
    static
    §s
        NameTable     = NOTVEC
        NameTableHash = NOTVEC
25      NameTableLink = NOTVEC
        NumTable      = NOTVEC
        StringTable   = NOTVEC
        GetList       = NOTVEC
        ProcVec       = NOTVEC
30      TextVec       = NOTVEC
        Stack         = NOTVEC
        StackPtr      = 0
        EndofPP       = false
        Symb          = 0
35      SaveLN        = 0
        OldLN         = 0
        OldGet        = 0
        DeclIndex     = 0
    $s
40
    manifest
    §m
        ENDCH       = 8377
        MAXLENGTH   = 255
45      MAXCH       = 8377
        NULL        = 0

        SIZE        = 0
        PTR         = 1
50      REP         = 2
        FIRSTEL     = 3
        OFFSET      = 2

        SAVEIN      = 2
55      SAVELINENO  = 3
        FILENO      = 4
        GETPRE      = 5
        GETSIZE     = 5
        GETMAX      = 30
60      GETLISTSIZE = GETMAX × GETSIZE
    $m
```

```
    table
    §   CatMap =
 65 8000006, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,
    8000004, 8000004, 8000000, 8000004, 8000006, 8000004, 8000004, 8000006,
    8000004, 8000004, 8000004, 8000004, 8000204, 8000004, 8000004, 8000004,
    8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,

 70 8000006, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000204,
    8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,
    8000055, 8000055, 8000055, 8000055, 8000055, 8000055, 8000055, 8000055,
    8000015, 8000015, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,

 75 8000004, 8000405, 8000405, 8000405, 8000405, 8000405, 8000405, 8000405,
    8000405, 8000405, 8000405, 8000405, 8000405, 8000405, 8000405, 8000405,
    8000405, 8000405, 8000405, 8000405, 8000405, 8000405, 8000405, 8000405,
    8000405, 8000405, 8000405, 8000004, 8000004, 8000004, 8000004, 8000004,

 80 8000104, 8000025, 8000025, 8000025, 8000025, 8000025, 8000025, 8000025,
    8000025, 8000025, 8000025, 8000025, 8000025, 8000025, 8000025, 8000025,
    8000025, 8000025, 8000025, 8000025, 8000025, 8000025, 8000025, 8000025,
    8000025, 8000025, 8000025, 8000004, 8000004, 8000004, 8000004, 8000004,

 85 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,
    8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,
    8000004, 8000004, 8000004, 8000004, 8000104, 8000004, 8000004, 8000004,
    8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,

 90 8000007, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,
    8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,
    8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,
    8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004, 8000004,

 95 8000004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004,
    8001004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004,
    8001004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004,
    8001004, 8001004, 8001004, 8000204, 8000004, 8000004, 8000004, 8000004,

100 8000004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004,
    8001004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004,
    8001004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004, 8001004,
    8001004, 8001004, 8001004, 8000004, 8000004, 8000004, 8000004, 8000000
    $
105
    manifest
    §m
        Is[x, Cat] = (CatMap↓x ∧ Cat) ≠ 0

110     PreProc[] is (ProcVec↓Ch)[]

        StackHash[] = StackPtr × Stack↓1 × Stack↓StackPtr

        Find[h, v] = DSwitch[v + FIRSTEL, v + v↓PTR + 1, -1, h]
115
        Pop[] is StackPtr := StackPtr - 1

        Push[x] is
        §P
120         if StackPtr < MAXLENGTH + 2 do StackPtr := StackPtr + 1
            Stack↓StackPtr := x
        $P

        NextCh[] is Ch := Next[In]
125
        PushCh_NextCh[] is
```

```
        §PN
            Push[Ch]
            NextCh[]
130     §PN

        PackStack[] is
        §   Stack↓0 := StackPtr
            PackString[Stack, Stack]
135     §

        ResetStack[] is StackPtr := 0

        TooBig[] = StackPtr > MAXLENGTH
140
        Top[] = Stack↓StackPtr

        IsRoomforInputStream[] = (MaxVecSize[] ≥ 320)
            || ensures there is room for
145         || either BytesfromWords[InfromFile[f]]
            || or (hopefully) a LookUp

        CanStartCom[x] = (x ∧ CSC) ≠ 0

150     CanEndCom[x] = (x ∧ (CEC ∨ CEE)) ≠ 0

        CanEndExp[x] = (x ∧ CEE) ≠ 0

        MSCRt[] is
155         if CanEndExp[Symb] do Send[DO, 0]

        DecimalFn[] = NumFn[10, CAT_d]

        OctalFn[] = NumFn[8, CAT_o]
160 §m

****
```

```
Declarations for Pass 1
 *** Compilers 16/2/77


    manifest GA = 430

    global
 5  §g
        Block           : GA
        Command         : GA + 1
        Expression      : GA + 2
        DeleteCommand   : GA + 3
 10     RestoreLevel    : GA + 4
        NewLevel        : GA + 5
        Name            : GA + 6
        DelComRestLevel : GA + 7
    $g
 15
    static
    §s
        LastNL   = 0
        ExpError = false
 20 $s

    manifest
    §m
        ScanName[] = valof
 25     §SN
            Scan[]
            resultis Name[]
        $SN

 30     ScanCommand[] is
        §SC
            Scan[]
            Command[]
        $SC
 35
        ScanExpression[] is
        §SE
            Scan[]
            Expression[]
 40     $SE

        FormSecKet[C] = SECTKET ∨ ValPart[C]

        NullSecKet[] = SECTKET ∨ NullNameNo
 45
        IsSecKet[C] = IsCType[C, SECTKET]
    $m


****
```

```
Declarations for Pass 3
 *** Compilers 31/7/76

    manifest GC = 430

    global
    §g
 5      Command             : GC
        Expression          : GC + 1
        LeftExpression      : GC + 2
        ConstantExpression  : GC + 3
        ConstOp             : GC + 4
10      Application         : GC + 5
        DealwithAss         : GC + 6
        FindName            : GC + 7
        Prec                : GC + 8
        StackOutput         : GC + 9
15      ClearStack          : GC + 10
        Addto               : GC + 11
        Subtractfrom        : GC + 12
        NormalOut           : GC + 13
        ChangeOutput        : GC + 14
20      DeleteCommand       : GC + 15
        CheckScope          : GC + 16
        SetUp3b             : GC + 17
        CloseDown3b         : GC + 18
        ManFunDefinition    : GC + 19
25      ManFunDefParams     : GC + 20

        NormalOutput        : GC + 21
        CellOutput          : GC + 22
        OutputVec           : GC + 23
30      ShuntVec            : GC + 24
        NonRecVec           : GC + 25
        NonRecDef           : GC + 26
        PresentOutput       : GC + 27
        FHN                 : GC + 28
35      LastRelExt          : GC + 29
        SSP                 : GC + 30
    $g

    manifest
40  §m
        BOTTOM     = 8366
        SUBEXP     = 8367

        NORMAL     = 0
45      MFAPP      = 1
        MFDEF      = 2
        CONSTANT   = 3
        IGNORE     = 4

50      SIZE       = 0
        PTR        = 1
        FIRSTEL    = 2

        SHUNTSIZE  = 50
55      DEPTHSIZE  = 20
        STACKSIZE  = 20
        BUFFERSIZE = 200
    $m

60  manifest
    §m
```

```
    UpdateSSP[T] is
        SSP := SSP + StackIncrement[T]
65
    Code[Type, n] is
    §C
        ClearStack[]
        NormalOutput[2, Type, n]
70      UpdateSSP[Type]
    §C

    Inst[Type] is
    §I
75      ClearStack[]
        NormalOutput[1, Type]
        UpdateSSP[Type]
    §I

80  ForHop[n] is
    §FH
        ClearStack[]
        NormalOutput[2, FORHOP, n]
    §FH
85
    ForHopPt[n] is
        if n > 0 do
        §   ClearStack[]
            NormalOutput[2, FORHOPPT, n]
90      §

    BackHop[n] is
    §BH
        ClearStack[]
95      NormalOutput[2, BACKHOP, n]
    §BH

    BackHopPt[n] is
    §BHP
100     ClearStack[]
        NormalOutput[2, BACKHOPPT, n]
    §BHP

    CondForHop[Type, n] is
105 §CFH
        ClearStack[]
        NormalOutput[3, CONDFORHOP, n, Type]
        UpdateSSP[Type]
    §CFH
110
    SetLitGlobal[No, Loc, Val] is
        NormalOut[4, SETLITGLOBAL, No, Loc, Val]

    SetLitStatic[No, Val] is
115     NormalOut[3, SETLITSTATIC, No, Val]

    SetDSegGlobal[No, Loc, l] is
        NormalOut[4, SETDSEGGLOBAL, No, Loc, l]

120 SetDSegStatic[No, l] is
        NormalOut[3, SETDSEGSTATIC, No, l]

    SetCSegGlobal[No, Loc, Id] is
    §SCG
125     ClearStack[]
        NormalOutput[4, SETCSEGGLOBAL, No, Loc, Id]
```

```
        §SCG

        SetCSegStatic[No, Id] is
130     §SCS
            ClearStack[]
            NormalOutput[3, SETCSEGSTATIC, No, Id]
        §SCS

135     SetManFun[No, Val] is
        §SMF
            ClearStack[]
            NormalOutput[3, SETMANFUN, No, Val]
        §SMF
140
        LoadManFun[No, Code] is
        §LMF
            ClearStack[]
            NormalOutput[3, LOADMANFUN, No, Code]
145         UpdateSSP[RAsm]
        §LMF

        LoadStatic[Type, No] is
        §LS
150         ClearStack[]
            NormalOutput[3, LOADSTATIC, Type, No]
            UpdateSSP[Type]
        §LS

155     LoadString[No] is
        §LS
            ClearStack[]
            NormalOutput[2, LOADSTRING, No]
            UpdateSSP[Nsm]
160     §LS

        UpdateS[n] is
        §US
            ClearStack[]
165         NormalOutput[2, Ssm, n]
            SSP := n
        §US

        StartOutputMode[Mode] is
170     §   Addto[OutputVec, Mode]
            ChangeOutput[]
        §

        EndOutputMode[] is
175     §   Subtractfrom[OutputVec]
            ChangeOutput[]
        §

        StartNonRecDef[] is
180     §   NonRecDef := true
            Addto[NonRecVec, Generation[PresentLevel]]
        §

        EndNonRecDef[Old] is
185     §   Subtractfrom[NonRecVec]
            NonRecDef := Old
        §

        Top[Vec] = Vec↓(Vec↓PTR)
190
        SwitchNo[] = valof
```

```
        §SWN
            SWN := SWN + 1
            resultis SWN
195     §SWN

        HopNo[] = valof
        §HN
            HN := HN + 1
200         resultis HN
        §HN

        UnusedHopNo[] = -HopNo[]

205     HopNoUsed[h] = (h ≥ 0)

        CellLength[p] = LengthField[p↓0]

        CellType[p] = p↓0
210
        CellNo[p] = p↓1

        CellNo2[p] = p↓2
    §m
215

****
```

```
Declarations for Pass 3a
 *** Compilers 31/7/76


    static
    §s
        NumTable      = 0
 5      Stack         = 0
        StackPtr      = 0
        StackBase     = 0

        Buffer        = DUMMY
10      BufferType    = DUMMY
        BufferLine    = 0
        VHN           = UNSET
        VSSP          = -1

15      SwitchVec     = 0
        CaseGen       = UNSET
        InRt          = false
        BHN           = UNSET
        CHN           = UNSET
20      EHN           = UNSET
        LHN           = UNSET

                    || setting globals:
        NormalOutput  = 0
25      CellOutput    = 0
        OutputVec     = UNSET
        ShuntVec      = UNSET
        NonRecVec     = UNSET
        NonRecDef     = false
30      PresentOutput = 0
        FHN           = UNSET
        LastRelExt    = false
        SSP           = -1
    $s
35
    manifest
    §m
        RAND  = 0
        RATOR = -1
40
        RIGHT = false
        LEFT  = true

        Abs[x] = (x < 0 → -x, x)
45
        StackSize[] = (StackPtr - StackBase)

        IsBool[x] = (x = true ∨ x = false)

50      BoolValueonStack[] =
            (FHN = UNSET ∧ StackSize[] > 0 ∧ IsBool[Stack↓StackPtr])
    $m


****
```

```
Declarations for Pass 3b
 *** Compilers 24/6/76


    static
    §s
        BuffStr       = 0
 5      SaveIn        = 0
        InPtr         = 0
        InEnd         = 0

        MFDefBuff     = 0
10      MFAppBuff     = 0
        PresMFDef     = 0
        PresMFApp     = 0
        MFCode        = 0
        MFDefPtr      = 0
15      MFAppPtr      = 0
        MFDefEnd      = 0
        MFAppEnd      = 0
        Store         = 0
        Link          = 0
20      MFDefBuffFull = 0
        MaxAppUsage   = 0
        Hop           = 0
        NoParams      = 0
        HopShift      = 0
25      ParamsSaved   = 0
        ParamBlock    = 0
        StackNo       = 0
        P1            = 0
        P2            = 0
30
        ManApp        = DUMMY
    $s

    manifest
35  §m
        HEADTYPE      = 0
        CODELENGTH    = 1
        PREV          = 2
        CODETYPE      = 3
40      INITHOPNO     = 4
        NOHOPS        = 5
        NOPARAMS      = 6
        NAMEBLOCK     = 7
        PARAMSSAVED   = 8
45      DEFHEADSIZE   = 9
        CODEPTR       = 4
        APPLYNO       = 5
        APPHEADSIZE   = 6
        NEWMANFUNDEF  = 80101
50      NEWMANFUNAPP  = 80102
        UNKNOWN       = 877776
        DIRECT        = 0
        SEMIDIRECT    = 1
        INDIRECT      = 2
55      LINKSIZE      = LengthField[LINKsm]
        RPSIZE        = LengthField[RPsm]
        LMFSIZE       = LengthField[LOADMANFUN]
        CHKJMPSIZE    = LengthField[JUMPPTsm]
        APSIZE        = LengthField[APPLYsm]
60  $m
```

```
    manifest
    §m
        UpdateCellNo[p, n] is p↓1 := p↓1 + n
65      UpdateCellNo2[p, n] is p↓2 := p↓2 + n

        GenCellLength[p] = valof
        §GL
            let l = CellLength[p]
70          resultis l = 0 → p↓1, l
        $GL

        NextCell[p] = (p + GenCellLength[p])

75      FindCodefromApp[] = rv (MFAppPtr - 1)

        ReportNonApp[] is
            Error[353, SOFT, NameField[FindCodefromApp[]↓NAMEBLOCK],
                NullProgram]
80  $m

****
```

```
Declarations for Pass 4
 *** Compilers 3/7/76


    manifest GD = 430

    global
 5  §g
        GenerateCode : GD
        SetUpCode    : GD + 1
        ProperVec    : GD + 2
    $g
 10
    static
    §s
        GlobalType    = 0
        GlobalLoc     = 0
 15     GlobalVal     = 0
        StaticType    = 0
        StaticLastUse = 0
        StaticVal     = 0
        MFVal         = 0
 20     MFLastUse     = 0
        StringUse     = 0
        StringData    = 0
        TableData     = 0
        HopList       = 0
 25     BackHopList   = 0
        SwitchList    = 0
        DiagAddr      = 0
        DiagGlobNo    = 0
        DiagName      = 0
 30     DiagString    = 0
        SL            = 0
        TDPtr         = 0
        DN            = 0
        Vector        = 0
 35     Ptr           = 0
        BodyVec       = 0
        NIn           = 0
        CSize         = 0
        FirstCode     = 0
 40     FirstWord     = 0
        LastWord      = 0
        CodeFull      = 0
        SpareByte     = 0
        SNec          = false
 45     ExitNec       = false
        HopNext       = false
        HopPtNext     = false
        FnExitNext    = false
        NextSS        = UNSET
 50     HopNo         = 0
        HopPtNo       = 0
    $s

    manifest
 55 §m
        SetUpBody[Size] is
        §   Vector := ProperVec[Size]
            BodyVec := Vector
        §
 60
        SetCodePtrs[] is
```

```
        §   FirstWord := BodyVec
            LastWord := BodyVec + CodeSize
            FirstCode := LastWord + 1
 65     §

        AlterBody[] is
        §   BodyVec := FirstCode
            Ptr := LastWord - FirstCode + 1
 70     §

        ResetBody[] is
            BodyVec := Vector

 75     ReturnBody[Size] is
            ReturnVec[Vector, Size]

        CodeWord[Loc] = rv Loc
        UpdateCode[Loc, v] is rv Loc := v
 80     UpdateBody[Ptr, v] is BodyVec↓Ptr := v
        CopytoBody[Vec, n] is
        §   Copy[Vec, BodyVec + Ptr, n]
            Ptr := Ptr + n
        §
 85
        CopyStringtoBody[S] is
            CopytoBody[S, LHalf[S↓0]/2 + 1]

        TransferIntoBody[S, p, n] is
 90         TransferIn[S, BodyVec + p, n]

        OutputSection[Type, n] is
        §   Write[Type]
            Write[n]
 95         TransferOut[Output, BodyVec, n]
        §

        HOPSUM       = HOPIFZEsm + HOPIFNZsm
        FIRSTMEDINST = 8260
100     LASTMEDINST  = 8317

        RevCondType[CT] = HOPSUM - CT

        CodePos[] = FirstCode - 1
105     NextCodePos[] = FirstCode
        LastCodePos[] = FirstCode - 2

        WordafterHop[n] = valof
        §   let l = HopList↓n
110         resultis l < 0 → 0, CodeWord[l]
        §

        DiagNo[] = valof
        §   DN := DN + 1
115         resultis DN
        §
    $m


****
```

```
   manifest
   §m
       PreviousLevel[L] = L↓0
       LevelType[L] = L↓1
       Generation[L] = L↓2
       NameList[L] = L↓3

       UpdatePreviousLevel[L, P] is L↓0 := P
       UpdateLevelType[L, T] is L↓1 := T
       UpdateGeneration[L, G] is L↓2 := G
       UpdateNameList[L, N] is L↓3 := N

       PreviousNameBlock[N] = N↓0
       NameType[N] = N↓1
    || Generation[N] = N↓2
       NextName[N] = N↓3
       NameVal[N] = N↓4
       NameNo[N] = N↓5
       NameField[N] = N↓6

       UpdatePreviousNameBlock[N, P] is N↓0 := P
       UpdateNameType[N, T] is N↓1 := T
    || UpdateGeneration[N, G] is N↓2 := G
       UpdateNextName[N, V] is N↓3 := V
       UpdateNameVal[N, V] is N↓4 := V
       UpdateNameNo[N, V] is N↓5 := V
    || UpdateNameField[N, F] is N↓6 := F

       NameBlock[N] = NameVec↓(ValPart[N])
       UpdateNameBlock[N, NB] is NameVec↓(ValPart[N]) := NB

       IsCType[C, T] = (C ∧ TYPEMASK) = T

       UpdateCh[C] is
       §U
           Ch := C
           ChType := TypePart[C]
       $U

       BackUp[C] is
       §BU
           PutBack[In, Ch]
           UpdateCh[C]
       $BU

       Insert[C] is
       §C
           Send[Ch]
           UpdateCh[C]
       $C


       MayPrecede[x, y] = Generation[x] < Generation[y]
$m

   let SetLevel[L] be
   §SL
       let A = CommonAncestor[PresentLevel, L]
       until PresentLevel = A do UpOneLevel[]
       DowntoLevel[L]
```

```
    $SL

    and UpOneLevel[] be
65  §UOL
        let p = NameList[PresentLevel]
        until p = NILNAME do
        §   UpdateNameBlock[NameField[p], PreviousNameBlock[p]]
            p := NextName[p]
70      §
        PresentLevel := PreviousLevel[PresentLevel]
    $UOL

    and DowntoLevel[L] be
75      unless PresentLevel = L do
        §DL
            DowntoLevel[PreviousLevel[L]]
          § let p = NameList[L]
            until p = NILNAME do
80          §   UpdatePreviousNameBlock[p, NameBlock[NameField[p]]]
                UpdateNameBlock[NameField[p], p]
                p := NextName[p]
            §
            PresentLevel := L
85      $DL

    and CommonAncestor[x, y] = (x = y) → x,
        MayPrecede[x, y] → CommonAncestor[x, PreviousLevel[y]],
                           CommonAncestor[PreviousLevel[x], y]
90

    let Read[] be
    §R
        UpdateCh[Next[In]]
95      switchon ChType into
        §S
            case NEWLINE:
                    LineNo := ValPart[Ch]
                    endcase
100
            case GET:
                    GetNo := ValPart[Ch]
                    endcase

105         case NEWLEVEL:
                    SetLevel[StructureVec + ValPart[Ch]]
                    endcase

            case TYPEMASK:
110                 Error[5, HARD, DUMMY, Finish]

            case DUMMY:
                    loop

115         default:return
        $S
        SpecSend[Ch]
    $R repeat

120 and Scan[] be
    §S
        Send[Ch]
        Read[]
    $S
125
```

****

```
Level Procedures 2
 *** Compilers 20/12/75


    static
    §s
        Ignoring = false
  5 $s

    manifest
    §m
        LEVELMAX = 2047
 10
        CheckStructureFull[] is
            unless StructureNPtr > StructureLPtr ≤ LEVELMAX do
                Error[2, HARD, DUMMY, Finish]

 15     GlobalNo[] = valof
        §GN
            GN := GN + 1
            resultis GN
        $GN
 20
        StaticNo[] = valof
        §SN
            SN := SN + 1
            resultis SN
 25     $SN

        ManFunNo[] = valof
        §MFN
            MFN := MFN + 1
 30         resultis MFN
        $MFN

        IsName[C] = IsCType[C, NAME]

 35     Ignoreif[Cond] is
        §   Ignoring := Cond
            Send := Cond → NullProgram, Write
        $

 40     OutputDef[Name, Type] is
        §OD
            Send[Type]
            Send[Name]
        $OD
 45 $m


    let FormLevel[Prev, Type, Gen, NameL] = valof
    §FL
 50     let L = StructureVec + StructureLPtr
        StructureLPtr := StructureLPtr + 4
        CheckStructureFull[]
        Copy[lv Prev, L, 4]
        resultis L
 55 $FL


    and FormNameBlock[Prev, Type, Gen, N, Val, No, Fld] = valof
    §FNB
 60     StructureNPtr := StructureNPtr - 7
        CheckStructureFull[]
```

```
      § let N = StructureVec + StructureNPtr
        Copy[lv Prev, N, 7]
        resultis N
 65 $FNB


    and AddDef[N, AddType, DefType] be
    §AD
 70     let F = 0
        switchon AddType into
        §S
            case SIMPLE:
            case GLOBAL:
 75         case MANIFEST:
            case MANFN:
                    AddName[N, AddType]
                    endcase

 80         case STATIC:
            case TABLE:
                    F := NameBlock[N]
                    test NameType[F] = GLOBAL
                      ifso  UpdateGlobNN[F]
 85                   ifnot AddName[N, AddType]
                    endcase

            default:CompilerError[1001]
        §S
 90     OutputDef[N, DefType]
    §AD


    and UpdateGlobNN[F] be
 95     test NameNo[F] = UNDEFINED
          ifso  UpdateNameNo[F, GlobalNo[]]
          ifnot Error[23, HARD, NameField[F], NullProgram]


100 and AddName[N, Type] be
    unless Ignoring do
    §AN
        let No = (Type = STATIC ∨ Type = LABEL ∨ Type = TABLE) → StaticNo[],
                 (Type = MANFN) → ManFunNo[], UNDEFINED
105     let v = FormNameBlock[NameBlock[N], Type, Generation[PresentLevel],
                NameList[PresentLevel], UNUSED, No, N]
        UpdateNameList[PresentLevel, v]
        UpdateNameBlock[N, v]
    §AN
110
****
```

```
Manifests 0
 *** Compilers 31/7/76


    || BCPL PP (C.M.J.)   ******* 30/8/73 17.45
    || Manifests

  5 manifest
    §10
        CEC         = 8 002000
        CEE         = 8 004000
        CSC         = 8 010000
 10 §10

    manifest
    §20
        NAME        = 8 020000 ∨ CSC ∨ CEE
 15     NUMBER      = 8 040000 ∨ CSC ∨ CEE
        STRING      = 8 060000 ∨ CSC ∨ CEE
        CHARACTER   = 8 100000 ∨ CSC ∨ CEE
        SECTBRA     = 8 060000 ∨ CSC
        SECTKET     = 8 060000 ∨ CEE
 20     COUNT       = 8 100000
        NEWLEVEL    = 8 120000
        NEWLINE     = 8 140000
        GET         = 8 160000

 25     ASS         = 8 000001
        COLON       = 8 000002
        COMMA       = 8 000003
        COND        = 8 000004
        DIV         = 8 000005
 30     EQ          = 8 000006
        EQV         = 8 000007
        GE          = 8 000010
        GT          = 8 000011
        LE          = 8 000012
 35     LOGAND      = 8 000013
        LOGOR       = 8 000014
        LT          = 8 000015
        MINUS       = 8 000016
        MULT        = 8 000017
 40     NE          = 8 000020
        NEQV        = 8 000021
        NOT         = 8 000022
        PLUS        = 8 000023
        RBRA        = 8 000024 ∨ CSC
 45     RKET        = 8 000025 ∨ CEE
        SBRA        = 8 000026
        SEMICOLON   = 8 000027
        SKET        = 8 000030 ∨ CEE
        VECAP       = 8 000031
 50     VECOP       = 8 000032
        AND         = 8 000033
        BE          = 8 000034
        BY          = 8 000035
        BREAK       = 8 000036 ∨ CSC ∨ CEC
 55     CASE        = 8 000037 ∨ CSC
        DEFAULT     = 8 000040 ∨ CSC
        DO          = 8 000041
        ELSE        = 8 000062
        ENDCASE     = 8 000043 ∨ CSC ∨ CEC
 60     FALSE       = 8 000044 ∨ CSC ∨ CEE
        FOR         = 8 000045 ∨ CSC
```

```
        GLOBAL      = 8 000046
        GOTO        = 8 000047 ∨ CSC
        IF          = 8 000050 ∨ CSC
 65     IFNOT       = 8 000051
        IFSO        = 8 000052
        INTO        = 8 000053
        IS          = 8 000054
        LET         = 8 000055
 70     LOOP        = 8 000056 ∨ CSC ∨ CEC
        LSHIFT      = 8 000057
        LV          = 8 000060 ∨ CSC
        MANIFEST    = 8 000061
        OR          = ELSE
 75     REM         = 8 000063
        REPEAT      = 8 000064 ∨ CEC
        REPEATUNTIL = 8 000065
        REPEATWHILE = 8 000066
        RESULTIS    = 8 000067 ∨ CSC
 80     RETURN      = 8 000070 ∨ CSC ∨ CEC
        RSHIFT      = 8 000071
        RV          = 8 000072 ∨ CSC
        STATIC      = 8 000073
        SWITCHON    = 8 000074 ∨ CSC
 85     TABLE       = 8 000075
        TEST        = 8 000076 ∨ CSC
        THEN        = DO
        TO          = 8 000100
        TRUE        = 8 000101 ∨ CSC ∨ CEE
 90     UNLESS      = 8 000102 ∨ CSC
        UNTIL       = 8 000103 ∨ CSC
        VALOF       = 8 000104 ∨ CSC
        VEC         = 8 000105
        WHILE       = 8 000106 ∨ CSC
 95 §20
    manifest
    §30
        CAT_a = 8 000001
        CAT_b = 8 000002
100     CAT_c = 8 000004
        CAT_d = 8 000010
        CAT_l = 8 000020
        CAT_o = 8 000040
        CAT_q = 8 000100
105     CAT_s = 8 000200
        CAT_u = 8 000400
        CAT_v = 8 001000
    §30

****
```

```
Manifests 1
 *** Compilers 29/5/76


    manifest
    §m
        CCOLON    = 8140
 5      UPLUS     = 8142
        UMINUS    = 8143
        DEF       = 8144
        LOCDEF    = 8145

 10     ENDBODY   = 8200
        ENDVALOF  = 8215
        ENDFOR    = 8230
        ENDSWITCH = 8201
        FNBODY    = 8203
 15     FNDF      = 8204
        RTBODY    = 8205
        RTDF      = 8206
        VALDF     = 8207
        VECDF     = 8210
 20     LASS      = 8211
        REPSTART  = 8212
        MANFNDF   = 8216
        MANRTDF   = 8217
        FNEXIT    = 8231
 25     RTEXIT    = 8232

        LOCAL     = 8220
        PARAM     = 8221
        SIMPLE    = 8222
 30     VALBODY   = 8224
        LABEL     = 8225
        MANFN     = 8226
        MANRT     = 8227

 35     EMPTY     = 0
        NOTNAME   = 0
        UNDEFINED = -10000
        UNUSED    = -20000
        SET       = -1
 40
        TESTIFS   = DO + OR
    $m

****
```

```
Pass 1.1
 *** Compilers 16/2/77


    let Pass1[] be
    §P1
        Send := Write
 5      SpecSend := Write
        SetupStructure[]
        SetupNameVec[]
        Peep[1]
        PassNo := 1
10      LineNo := 1
        GetNo := 0
        ReportNo := 0
        GN, SN, MFN := 0, 0, 0
        Send[ENDPROG]
15      NewLevel[RTBODY]
        ExpError := false
        UpdateCh[SECTBRA]
        Block[]
        unless Ch = ENDPROG do
20          Error[6, HARD, Ch, NullProgram]
        ReturnMap[NameMap]
        Peep[3]
    §P1

25  and SetupStructure[] be
    §SLV
        if StructureSize < 20 do
            CompilerError[1101]
        StructureVec := TopVec[StructureSize]
30      StructureLPtr := 0
        StructureNPtr := StructureSize
        NILNAME := FormNameBlock[UNDEFINED, UNDEFINED, 0, UNDEFINED,
                UNDEFINED, UNDEFINED, UNDEFINED]
        UpdateNextName[NILNAME, NILNAME]
35      UpdatePreviousNameBlock[NILNAME, NILNAME]
        OUTSIDE := FormLevel[UNDEFINED, UNDEFINED, 0, NILNAME]
        UpdatePreviousLevel[OUTSIDE, OUTSIDE]
        PresentLevel := OUTSIDE
        LocalGenNo := 0
40      LastNL := OUTSIDE - StructureVec
    §SLV

    and SetupNameVec[] be
    §SNV
45      let Size = NameSize + 26
        NameVec := TopVec[Size]
        for i = 0 to Size do NameVec↓i := NILNAME
    §SNV

50
    and TopVec[n] = valof
    §TV
        let m = MaxVecSize[]
        let v = NewVec[m]
55      ReturnVec[v, m - n - 1]
        resultis v + m - n
    §TV

    and LevelChange[] be
60  §LC
        let PresNL = PresentLevel - StructureVec
```

```
        Send[NEWLEVEL \/ LastNL]
        Send[NEWLEVEL \/ PresNL]
        LastNL := PresNL
65  $LC

    and Block[] be
    §B
        unless ChType = SECTBRA do
70          Error[111, HARD, Ch, BackUp, SECTBRA]
      § let SecKet, Level = FormSecKet[Ch], PresentLevel
        and DefBlock = false
        Scan[]

75      §r
            switchon ChType into
            §s
                case GLOBAL:
                        DefBlock := true
80                      ReadDefBlock[Ch, COLON]
                        endcase
                case MANIFEST:
                case STATIC:
                case TABLE:
85                      DefBlock := true
                        ReadDefBlock[Ch, EQ]
                        endcase
                case AND:
                        Error[105, HARD, Ch, UpdateCh, LET]
90              case LET:
                        DefBlock := true
                        ReadDef[]
                        endcase
                case SEMICOLON:
95                      Scan[]
                default:
                        §r
                            Command[]
                            if ChType = SECTKET  \/  Ch = ENDPROG break
100                         if StartsDef[] do
                            §   Error[142, HARD, Ch, NullProgram]
                                if Ch = AND do UpdateCh[LET]
                                BackUp[SECTBRA]
                                BackUp[SEMICOLON]
105                         §
                            unless Ch = SEMICOLON do
                            §   Send[SEMICOLON]
                                Error[142, HARD, Ch, DeleteCommand]
                                unless Ch = SEMICOLON break
110                         §
                            Scan[]
                        $r repeat
                case SECTKET:
                case ENDPROG:
115                     test Ch = SecKet
                          ifso  Scan[]
                          ifnot InsertSecKet[SecKet]
                        until PresentLevel = Level do UpOneLevel[]
                        if DefBlock do LevelChange[]
120                     return
            $s
        $r repeat
    $B

125 and ReadDefBlock[DefType, Delim] be
    §RDB
```

```
        let SecKet = SECTKET
        and SaveI = Ignoring
        NewLevel[LOCAL]
130     Read[]
        test ChType = SECTBRA
          ifso  §   SecKet := FormSecKet[Ch]
                    Scan[]
                §
135       ifnot Send[SECTBRA]

        §r
            let N = Name[]
            Ignoreif[(DefType = MANIFEST ∨ DefType = GLOBAL) ∧
140                  EntryinMap[NameMap, ValPart[N]]]
            test N = NOTNAME
              ifso  Error[201, HARD, Ch, DeleteCommand]
              ifnot switchon Ch into
                    §s
145                     case COLON:
                        case EQ:
                                unless Ch = Delim do
                                §   Error[105, HARD, Ch, DeleteCommand]
                                    endcase
150                             §
                                UpdateCh[DUMMY]
                                AddDef[N, DefType, DefType]
                                ScanExpression[] repeatwhile Ch = COMMA ∧
                                        DefType = TABLE
155                             endcase

                        case SBRA:
                                unless DefType = MANIFEST do
                                §   Error[105, HARD, Ch, DeleteCommand]
160                                 endcase
                                §
                                ProcDef[N, MANFN, MANFNDF, IS]
                                endcase

165                     default:Error[105, HARD, Ch, DeleteCommand]
                    §s

            if SecKet = SECTKET do BackUp[SECTKET]
            Ignoreif[SaveI]
170         if ChType = SECTKET  ∨  Ch = ENDPROG break
            unless Ch = SEMICOLON do
            §   test StartsDef[]
                  ifso  Error[142, HARD, Ch, BackUp, SecKet]
                  ifnot §   Send[SEMICOLON]
175                         Error[142, HARD, Ch, DeleteCommand]
                        §
                unless Ch = SEMICOLON break
            §
            Scan[]
180     $r repeat

        test Ch = SecKet
          ifso  Scan[]
          ifnot InsertSecKet[SecKet]
185 §RDB


    and InsertSecKet[SecKet] be
    §ISK
190     unless SecKet = SECTKET do
          test Ch = ENDPROG
```

```
            ifso  Error[5, HARD, Ch, NullProgram]
            ifnot unless SecKet = NullSecKet[] do
                    Error[107, SOFT, SecKet, NullProgram]
195     Send[SecKet]
    §ISK


    and ReadDef[] be
200 §RD
        NewLevel[LOCAL]

        §r1
            let n = 1
205         §r2
                let N = ScanName[]
                if N = NOTNAME do
                §   Error[201, HARD, Ch, DeleteCommand]
                    break
210             §
                switchon Ch into
                §s
                    case COMMA:
                            AddDef[N, SIMPLE, VALDF]
215                         n := n+1
                            endcase

                    case EQ:
                            Read[]
220                         if Ch = VEC do
                            §   AddDef[N, SIMPLE, VECDF]
                                unless n = 1 do
                                §   Error[303, HARD, Ch, DeleteCommand]
                                    break
225                             §
                                Read[]
                                Expression[]
                                break
                            §
230                         AddDef[N, SIMPLE, VALDF]
                            Send[DEF]
                            §r3
                                Expression[]
                                n := n-1
235                             unless Ch = COMMA break
                                Scan[]
                            §r3 repeat
                            unless n = 0 do
                                Error[301, HARD, Ch, DeleteCommand]
240                         break

                    case SBRA:
                            ProcDef[N, STATIC, FNDF, BE]
                            break
245
                    default:
                            Error[105, HARD, Ch, DeleteCommand]
                            break
                §s
250         §r2 repeat

            if StartsDef[] loop
            if Ch = SEMICOLON ∨ ChType = SECTKET ∨ Ch = ENDPROG break
            Error[105, HARD, Ch, DeleteCommand]
255
        §r1 repeatwhile Ch = AND
```

```
    $RD

260 and ProcDef[N, AddType, DefType, Delim] be
    §PD
        AddDef[N, AddType, DefType]
        NewLevel[FNBODY]
        ParameterList[]
265     unless Ch = SKET do
        §   Error[116, HARD, Ch, DelComRestLevel]
            return
        $
        Read[]
270     switchon Ch into
        §s
            case EQ:
                    Send[FNBODY]
                    Read[]
275                 Expression[]
                    unless ExpError do Send[ENDBODY]
                    endcase

            case BE:
280         case IS:
                    unless Ch = Delim do
                        Error[105, HARD, Ch, NullProgram]
                    Send[RTBODY]
                    Read[]
285                 UpdateLevelType[PresentLevel, RTBODY]
                    Command[]
                    Send[ENDBODY]
                    endcase

290         default:
                    Error[105, HARD, Ch, DeleteCommand]
        $s
        RestoreLevel[]
    $PD
295
    and Name[] = valof
    §N
        unless IsName[Ch] resultis NOTNAME
      § let N = Ch
300     let P = NameBlock[N]
        unless Ignoring do
            if Generation[P] = Generation[PresentLevel] do
                Error[202, HARD, Ch, NullProgram]
        Read[]
305     resultis N
    $N

    and NewLevel[LType] be
        unless Ignoring do
310     §NL
            let L = FormLevel[PresentLevel, LType,
                    Generation[PresentLevel] + 2, NILNAME]
            PresentLevel := L
            LevelChange[]
315     $NL

    and RestoreLevel[] be
        unless Ignoring do
        §RL
320         UpOneLevel[]
            LevelChange[]
```

```
        $RL

    and ParameterList[] be
325 §PL
        Read[]
        if Ch = SKET return
        §r
            let N = Name[]
330         if N = NOTNAME do
            §   Error[201, HARD, Ch, BackUp, ERROR]
                break
            §
            AddName[N, SIMPLE]
335         OutputDef[N, PARAM]
            unless Ch = COMMA break
            Scan[]
        $r repeat
    $PL
340
    and StartsDef[] = valof
    §SD
        switchon ChType into
        §s
345         case LET:
            case AND:
            case GLOBAL:
            case MANIFEST:
            case TABLE:
350         case STATIC:
                    resultis true

            default:resultis false
        $s
355 $SD

****
```

```
   let EndsExp[] = valof
   §E
       switchon ChType into
       §   case SECTKET:
        5      case REPEAT:
               case REPEATUNTIL:
               case REPEATWHILE:
               case DO:
               case OR:
       10      case IFSO:
               case IFNOT:
               case TO:
               case BY:
               case SEMICOLON:
       15      case INTO:
               case LET:
               case AND:
               case ENDPROG:
               case GLOBAL:
       20      case MANIFEST:
               case TABLE:
               case STATIC: resultis true
               default: resultis false
   $E
       25
   and Command[] be
   §C
       §R  switchon ChType into
           §S  case SECTBRA:
       30              Block[]
                       break
               case CASE:
                       ScanExpression[]
                       unless Ch = COLON do
       35              §   Error[113, HARD, Ch, Send, CCOLON]
                           loop
                       $
                       Send[CCOLON]
                       Read[]
       40              loop
               case DEFAULT:
                       Scan[]
                       unless Ch = COLON do
                       §   Error[113, HARD, Ch, Send, CCOLON]
       45                  loop
                       $
                       Send[CCOLON]
                       Read[]
                       loop
       50          case IF:
                   case UNLESS:
                   case WHILE:
                   case UNTIL:
                           ScanExpression[]
       55                  unless Ch = DO do
                           §   Error[130, HARD, Ch, DeleteCommand]
                               break
                           $
                           endcase
       60          case TEST:
                           ScanExpression[]
```

```
                    unless Ch = DO ∨ Ch = IFSO ∨ Ch = IFNOT do
                    §   Error[144, HARD, Ch, DeleteCommand]
                        break
                    §
                  § let A = TESTIFS - Ch
                    ScanCommand[]
                    unless Ch = A do
                    §   Error[145, HARD, Ch, DeleteCommand]
                        Send[A]
                        Send[SEMICOLON]
                        break
                    §
                    endcase
                  §
              case GOTO:
              case RESULTIS:
                    ScanExpression[]
                    unless EndsExp[] do
                    §   Error[102, HARD, Ch, DeleteCommand]
                        break
                    §
                    break
              case REPEAT:
              case REPEATUNTIL:
              case REPEATWHILE:
                    break
              case BREAK:
              case LOOP:
              case ENDCASE:
              case RETURN:
                    Scan[]
                    break
              case FOR:
                    NewLevel[LOCAL]
                  § let N = ScanName[]
                    if N = NOTNAME do
                    §   Error[201, HARD, Ch, DelComRestLevel]
                        break
                    §
                    AddDef[N, SIMPLE, VALDF]
                    unless Ch = EQ do
                    §   Error[122, HARD, Ch, DelComRestLevel]
                        break
                    §
                    Read[]
                    Expression[]
                    unless Ch = TO do
                    §   Error[132, HARD, Ch, DelComRestLevel]
                        break
                    §
                    ScanExpression[]
                    if Ch = BY do
                        ScanExpression[]
                    unless Ch = DO do
                    §   Error[130, HARD, Ch, DelComRestLevel]
                        break
                    §
                    ScanCommand[]
                    Send[ENDFOR]
                    RestoreLevel[]
                    break
                  §
              case NAME:
              case NUMBER:
              case STRING:
```

```
            case CHARACTER:
            case TRUE:
            case FALSE:
130         case RV:
            case RBRA:
            case VALOF:
                    §A  let n = 1
                        Expression[]
135                     if Ch = COLON endcase
                        unless Ch = ASS ∨ Ch = COMMA do
                        §   unless EndsExp[] do
                                Error[102, HARD, Ch, DeleteCommand]
                            break
140                     §
                        while Ch = COMMA do
                        §   ScanExpression[]
                            n := n+1
                        §
145                     unless Ch = ASS do
                        §   Error[112, HARD, Ch, DeleteCommand]
                            break
                        §
                        §   ScanExpression[]
150                         n := n-1
                        § repeatwhile Ch = COMMA
                        unless n = 0 do
                            Error[302, HARD, Ch, DeleteCommand]
                        unless EndsExp[] do
155                         Error[102, HARD, Ch, DeleteCommand]
                        break
                    §A
            case SWITCHON:
                    ScanExpression[]
160                 unless Ch = INTO do
                    §   Error[131, HARD, Ch, DeleteCommand]
                        break
                    §
                    Scan[]
165                 Block[]
                    Send[ENDSWITCH]
                    break
            case OR:
            case IFSO:
170         case IFNOT:
            case SEMICOLON:
            case SECTKET:
            case ENDPROG:
            case LET:
175         case AND:
            case GLOBAL:
            case MANIFEST:
            case TABLE:
            case STATIC:
180                 return
            default:
                    Error[103, HARD, Ch, DeleteCommand]
                    break
        §S
185     Scan[]
    §R repeat

    switchon ChType into
    §S
190     default:return
```

```
            case REPEAT:
                    Scan[]
                    endcase
195
            case REPEATUNTIL:
            case REPEATWHILE:
                    ScanExpression[]
                    unless EndsExp[] do
200                 §   Error[102, HARD, Ch, DeleteCommand]
                        endcase
                    §
                    endcase
        $S repeat
205 $C

    and Expression[] be
    §E
        §R
210         switchon ChType into
            §S
                case VALOF:
                        NewLevel[VALBODY]
                        ScanCommand[]
215                     RestoreLevel[]
                        Send[ENDVALOF]
                        loop
                case RBRA:
                        ScanExpression[]
220                     unless Ch = RKET do
                            Error[115, HARD, Ch, ExpressionError]
                        endcase
                case VECAP:
                        ScanExpression[]
225                     unless Ch = SKET do
                            Error[116, HARD, Ch, ExpressionError]
                        endcase
                case SBRA:
                        ScanExpression[] repeatwhile Ch = COMMA
230                     unless Ch = SKET do
                            Error[116, HARD, Ch, ExpressionError]
                        endcase
                case COND:
                        ScanExpression[]
235                     unless Ch = COMMA do
                            Error[114, HARD, Ch, ExpressionError]
                        endcase

                case NAME:
240                   § let F = NameBlock[Ch]
                        until F = NILNAME do
                        §   if NameVal[F] = UNUSED do
                                UpdateNameVal[F, UNDEFINED]
                            F := PreviousNameBlock[F]
245                     §
                        endcase
                      §

                case SECTBRA:
250             case IF:
                case UNLESS:
                case WHILE:
                case UNTIL:
                case TEST:
255             case FOR:
                case LOOP:
```

```
                case BREAK:
                case RETURN:
                case RESULTIS:
260             case SWITCHON:
                case CASE:
                case DEFAULT:
                case ENDCASE:
                case BE:
265             case GOTO:
                case IS:
                case VEC:
                        Error[101, HARD, Ch, ExpressionError]
                case SECTKET:
270             case REPEAT:
                case REPEATUNTIL:
                case REPEATWHILE:
                case DO:
                case OR:
275             case IFSO:
                case IFNOT:
                case TO:
                case BY:
                case SEMICOLON:
280             case INTO:
                case LET:
                case AND:
                case ENDPROG:
                case GLOBAL:
285             case MANIFEST:
                case TABLE:
                case STATIC:
                case RKET:
                case SKET:
290             case COLON:
                case COMMA:
                case ASS:
                        return
                case ERROR:
295                     ExpError := true
                        return
                default:
                        endcase              || all other symbols
            $S
300         if ExpError return
            Scan[]
        $R repeat
    $E

305 and DeleteCommand[] be
    §DC
        ExpError := false
        Send[ERROR]
        until IsEndofCom[] do Read[]
310 $DC

    and IsEndofCom[] = valof
    §IEC
        switchon ChType into
315     §S
            case SEMICOLON:
            case SECTKET:
            case LET:
            case AND:
320         case GLOBAL:
            case MANIFEST:
```

```
            case STATIC:
            case TABLE:
            case ENDPROG: resultis true
325         case SECTBRA: DeleteBlock[]
            default:      resultis false
        $S
    $IEC

330 and DeleteBlock[] be
    §D
        let SectKet = FormSecKet[Ch]
        Read[]
        switchon ChType into
335     §S
            case SECTBRA: DeleteBlock[]
                          Read[]
                          endcase
            case SECTKET: if Ch = SectKet return
340                       unless SectKet = NullSecKet[] do
                            Error[107, SOFT, SectKet, NullProgram]
                          BackUp[SectKet]
                          return
            case ENDPROG: Error[5, HARD, Ch, BackUp, SectKet]
345                       endcase
            default:      Read[]
        $S
        repeat
    $D
350
    and DelComRestLevel[] be
    §DCRL
        DeleteCommand[]
        RestoreLevel[]
355 $DCRL

    and ExpressionError[] be
    §E
        BackUp[ERROR]
360     ExpError := true
    $E

****
```

```
Pass 1
 *** Compilers 31/7/76


    get 'GLOBALS'
    get 'BitMap globals'

  5 get 'Manifests 0'
    get 'Manifests 1'

    get 'Common Decls'
    get 'Level Procs 1'
 10 get 'Level Procs 2'
    get 'Decls 1'

    get 'Pass 1.1'
    get 'Pass 1.2'
 15

****
```

```
Pass 2.1
 *** Compilers 31/7/76


   static
   §s
       RtDf    = false
 5     CaseNo  = 0
   $s

   let Pass2[] be
   §P2
10     Send := Write
       SpecSend := Write
       ReportNo := 0
       PassNo := 2
       UpdateCh[DUMMY]
15     RtDf := false
       Send[ENDPROG]
       BackPass[]
       CalculateStructureSize[]
   $P2
20
   and BackPass[] be
   §BP
       let Assignment = false
       and Definition = false
25     and AssNo = 1
       and RCaseNo = 0
       §R  Scan[]
           switchon ChType into
           §S
30             case COLON:
                       if Assignment do
                       §   Send[AssNo ∨ COUNT]
                           Send[LASS]
                           Assignment := false
35                     $
                       Read[]
                       unless IsName[Ch] do
                           Error[201, HARD, Ch, DeleteCommand]
                       FindLabel[]
40                     Insert[LABEL]
                       endcase

               case CASE:
                       CaseNo := CaseNo+1
45                     endcase

               case ENDSWITCH:
                       RCaseNo := CaseNo
                       CaseNo := 0
50                     BackPass[] repeatuntil Ch = SWITCHON
                       Send[CaseNo ∨ COUNT]
                       CaseNo := RCaseNo
                       endcase

55             case FNDF:
                       if RtDf do Ch := RTDF
                       RtDf := false
                       endcase

60             case MANFNDF:
                       if RtDf do Ch := MANRTDF
```

```
                RtDf := false
                endcase

    case ASS:
                AssNo := 1
                Assignment := true
                endcase

    case DEF:
                Definition := true
                endcase

    case COMMA:
                AssNo := AssNo + 1
                endcase

    case COND:
                AssNo := AssNo-1
                endcase

    case REPEAT:
    case REPEATUNTIL:
    case REPEATWHILE:
                BackPass[]
                Send[REPSTART]
                return

    case RTBODY:
                RtDf := true
    case DO:
    case OR:
    case IFSO:
    case IFNOT:
    case VALOF:
    case SEMICOLON:
    case CCOLON:
    case SECTBRA:
                if Assignment do
                §   Send[AssNo ∨ COUNT]
                    Send[LASS]
                §
                return

    case SWITCHON:
    case SBRA:
    case VECAP:
                return

    case SKET:
    case ENDVALOF:
                BackPass[]
                endcase

    case LET:
    case AND:
                if Definition do
                §   Send[LOCDEF]
                    Definition := false
                §
                endcase

    case SECTKET:
                BackPass[] repeatuntil IsSecBra[Ch]
                endcase
```

```
                case ENDPROG:
                        return

130             case ERROR:
                        DeleteCommand[]
                        BackUp[DUMMY]
                case default:
                        endcase
135
            $S
        $R repeat
    $BP

140 and FindLabel[] be
    §FL
        let OldNameBlock = NameBlock[Ch]
        unless LevelType[PresentLevel] = LABEL do
        §   let L = FormLevel[PreviousLevel[PresentLevel],
145                           LevelType[PresentLevel],
                              Generation[PresentLevel],
                              NameList[PresentLevel]]
            UpdatePreviousLevel[PresentLevel, L]
            UpdateNameList[PresentLevel, NILNAME]
150         UpdateLevelType[PresentLevel, LABEL]
            UpdateGeneration[PresentLevel, Generation[PresentLevel] + 1]
        §
        if Generation[PresentLevel] = Generation[OldNameBlock] do
            Error[202, HARD, Ch, NullProgram]
155     AddName[Ch, LABEL]
    §FL

    and IsSecBra[C] = IsCType[C, SECTBRA]

160 and CalculateStructureSize[] be
        ReportS['Structure Size = *N',
            StructureSize - StructureNPtr + StructureLPtr]

    and DeleteCommand[] be
165 §DC
        Read[]
        switchon ChType into
        §S
            case SEMICOLON:
170         case RTBODY:
            case DO:
            case OR:
            case IFSO:
            case IFNOT:
175         case VALOF:
            case CCOLON:
            case SECTBRA:
            case ENDBODY:
                    return
180
            case ENDPROG:
                    CompilerError[2101]

            case SECTKET:
185                 DeleteUntil[SECTBRA]
                    endcase

            case ENDVALOF:
                    DeleteUntil[VALOF]
190                 endcase
        $S
```

```
    §DC repeat

    and DeleteUntil[x] be
195 §DU
        Read[]
        switchon ChType into
        §S
            case SECTKET:
200                 DeleteUntil[SECTBRA]
                    endcase

            case ENDVALOF:
                    DeleteUntil[VALOF]
205                 endcase

            case ENDPROG:
                    CompilerError[2102]

210         default:if ChType = x return
        §S
    §DU repeat

****
```

```
Pass 2
 *** Compilers 31/7/76


    get 'GLOBALS'

    get 'Manifests 0'
  5 get 'Manifests 1'

    get 'Common Decls'
    get 'Level Procs 1'
    get 'Level Procs 2'
 10
    get 'Pass 2.1'


****
```

```
   let Pass3[] be
   §P3
       PassNo := 3
       Send := FailSend
       SpecSend := NullProgram
       LineNo := 1
       GetNo := 0
       ReportNo := 0
       LocalGenNo := 0
       HN := 0
       SWN := 0
       TableMax := 0
       NumTable := SetUpNumTable[]
       Stack := NewVec[STACKSIZE]
       Stack↓SIZE := STACKSIZE
       SetUp3b[]
       Peep[1]

       Write[ENDPROG]
       Command[]

       CloseDown3b[]
       ReturnVec[Stack, STACKSIZE]
       ReturnVec[NumTable, NumTable↓SIZE]
       ReturnVec[StructureVec, StructureSize]
       ReturnVec[NameVec, NameSize + 26]
       Peep[3]
   §P3


   and LeftName[C] be
   §LN
       let F = FindName[C]
       switchon NameType[F] into
       §s
           case PARAM:
           case SIMPLE:
                   Code[LPsm, NameVal[F]]
                   return

           case GLOBAL:
                   Code[LGsm, NameVal[F]]
                   return

           case STATIC:
           case TABLE:
           case LABEL:
                   LoadStatic[Nsm, NameNo[F]]
                   return

           case MANIFEST:
           case MANFN:
           case MANRT:
                   Error[204, HARD, C, NullProgram]

           case UNDEFINED:
                   CodeNsm[0]
                   return

           default:CompilerError[3101]
```

```
        §s
    §LN

65
    and RightName[C] be
    §RN
        let F = FindName[C]
        switchon NameType[F] into
70      §s
            case PARAM:
            case SIMPLE:
                    Code[RPsm, NameVal[F]]
                    return
75
            case GLOBAL:
                    Code[RGsm, NameVal[F]]
                    return

80          case STATIC:
            case TABLE:
            case LABEL:
                    LoadStatic[RAsm, NameNo[F]]
                    return
85
            case MANIFEST:
                    CodeNsm[NameVal[F]]
                    return

90          case MANFN:
            case MANRT:
                    if UNDEFINED ≠ NameVal[F] ≠ UNSET do
                    §   ClearStack[]
                        Addto[OutputVec, MFAPP]
95                      ChangeOutput[]
                    §
                    LoadManFun[NameNo[F], NameVal[F]]
                    return

100         case UNDEFINED:
                    CodeNsm[0]
                    return

            default:CompilerError[3102]
105     §s
    §RN


    and StoreName[C] be
110 §SN
        let F = FindName[C]
        switchon NameType[F] into
        §s
            case PARAM:
115         case SIMPLE:
                    Code[SPsm, NameVal[F]]
                    return

            case GLOBAL:
120                 Code[SGsm, NameVal[F]]
                    return

            case STATIC:
            case TABLE:
125         case LABEL:
                    LoadStatic[SAsm, NameNo[F]]
```

```
                    return

            case MANIFEST:
130         case MANFN:
            case MANRT:
                    Error[204, HARD, C, NullProgram]

            case UNDEFINED:
135                 Code[SPsm, 0]
                    return

            default:CompilerError[3103]
        $s
140 $SN


    and LeftVersion[C, CType] be
    §LV
145     switchon CType into
        §s
            case VECAP:
            case VECOP:
                    InstP[ADDsm]
150
            case RV:return

            case NAME:
                    LeftName[C]
155                 return

            default:Error[181, HARD, C, NullProgram]
        $s
    $LV
160

    and RightVersion[C, CType] be
    §RV
        switchon CType into
165     §s
            case NAME:
                    RightName[C]
                    return

170         case NUMBER:
                    CodeNsm[NumberVal[C]]
                    return

            case CHARACTER:
175                 CodeNsm[ValPart[C]]
                    return

            case STRING:
                    LoadString[ValPart[C]]
180                 return

            case TRUE:
                    CodeNsm[true]
                    return
185
            case FALSE:
                    CodeNsm[false]
                    return

190         case UPLUS:
            case DUMMY:
```

```
                return

        case NOT:
195             InstM[NOTsm]
                return

        case UMINUS:
                InstM[NEGsm]
200             return

        case PLUS:
                InstP[ADDsm]
                return
205
        case MINUS:
                InstP[SUBsm]
                return

210     case MULT:
                InstD[MULTsm]
                return

        case DIV:
215             InstD[DIVsm]
                return

        case REM:
                InstD[REMsm]
220             return

        case LSHIFT:
                InstD[LSHsm]
                return
225
        case RSHIFT:
                InstD[RSHsm]
                return

230     case LOGAND:
                InstD[ANDsm]
                return

        case LOGOR:
235             InstD[ORsm]
                return

        case EQV:
                InstD[EQVsm]
240             return

        case NEQV:
                InstD[NEQVsm]
                return
245
        case RV:Inst[CONTSsm]
                return

        case VECAP:
250     case VECOP:
                Inst[VECAPsm]
                return

        case EQ:
255     case NE:
        case LT:
```

```
            case LE:
            case GT:
            case GE:InstRel[CType]
260                 return

            default:CompilerError[3104]

            case LV:
265         case ASS:
            case COND:

        $s
    $RV
270

    and StoreVersion[C, CType] be
    §SV
        switchon CType into
275     §s
            case VECAP:
            case VECOP:
                    Inst[VECSTsm]
                    return
280
            case RV:Inst[STOREsm]
                    return

            case NAME:
285                 StoreName[C]
                    return

            default:Error[181, HARD, C, NullProgram]
        §s
290 §SV


    and CodeNsm[n] be
    §CN
295     test StackPtr ≥ Stack↓SIZE
          ifso  §   NormalOutput[2, Nsm, Stack↓(StackBase + 1)]
                    Copy[Stack + StackBase + 2, Stack + StackBase + 1,
                            STACKSIZE - 1 - StackBase]
                    Stack↓STACKSIZE := n
300               §
          ifnot §   StackPtr := StackPtr + 1
                    Stack↓StackPtr := n
                §
        SSP := SSP + 1
305 §CN


    and InstD[Type] be
    §ID
310     test StackSize[] ≥ 2
          ifso  StackOp[Type]
          ifnot Inst[Type]
    §ID

315
    and InstM[Type] be
    §IM
        test StackSize[] ≥ 1
          ifso  §   let v = Stack↓StackPtr
320                 Stack↓StackPtr := Type = NOTsm → v, -v
                §
```

```
          ifnot Inst[Type]
    §IM

325
    and InstP[Type] be
    §IP
        switchon StackSize[] into
        §s
330         case 0: Inst[Type]
                    return

            case 1:
                  § let n = (Type = SUBsm → -Stack↓StackPtr, Stack↓StackPtr)
335                 StackPtr := StackBase
                    NormalOutput[2, INCsm, n]
                    SSP := SSP - 1
                    return
                  §
340
            default:StackOp[Type]
                    return
        §s
    §IP
345

    and InstRel[Type] be
    §IR
        unless StackSize[] ≥ 2 do CompilerError[3105]
350     StackOp[Type]
    §IR


    and StackOp[Type] be
355 §SO
        let p = StackPtr - 1
        Stack↓p := ConstOp[Type, Stack↓p, Stack↓StackPtr]
        StackPtr := p
        SSP := SSP - 1
360 §SO


    and NumberVal[C] = valof
    §   let n = ValPart[C]
365     resultis 0 ≤ n ≤ 500 → n, NumTable↓(n - 500)
    §


    and SetUpNumTable[] = valof
370 §SNT
        let Table = NewVec[NumberSize]
        let S = InfromFile[WorkFile['Numbers']]
        let n = Next[S]
        Table↓SIZE := NumberSize
375     TransferIn[S, Table + 1, n]
        Close[S]
        resultis Table
    §SNT

380
    and StackOutput[p] be
    §SO
        switchon CellType[p] into
        §s
385         case Nsm:
                    CodeNsm[CellNo[p]]
```

```
                endcase

        case ADDsm:
390     case SUBsm:
                InstP[CellType[p]]
                endcase

        case NOTsm:
395     case NEGsm:
                InstM[CellType[p]]
                endcase

        case MULTsm:
400     case DIVsm:
        case REMsm:
        case LSHsm:
        case RSHsm:
        case ANDsm:
405     case ORsm:
        case EQVsm:
        case NEQVsm:
                InstD[CellType[p]]
                endcase
410
        case INCsm:
                if StackSize[] > 0 do
                §   Stack↓StackPtr := Stack↓StackPtr + CellNo[p]
                    endcase
415             §

        default:ClearStack[]
                CellOutput[p]
                endcase
420 $s
$SO


and ClearStack[] be
425 if StackPtr > StackBase do
    §
        let p = StackBase
        StackBase := StackPtr
        for i = p + 1 to StackPtr do
430         test Stack↓i = -1
              ifso  NormalOutput[1, TRUEsm]
              ifnot NormalOutput[2, Nsm, Stack↓i]
        ClearStack[]
        StackPtr, StackBase := p, p
435 §


and FailSend[x] be
    CompilerError[3106]
440

****
```

```
   let Expression[] be
   §E
       SubExpression[0]
       OuttoBuffer[DUMMY, DUMMY]
       unless FHN = UNSET do CoercetoVal[]
   $E


   and LeftExpression[] be
   §LE
       SubExpression[0]
       OuttoBuffer[ASS, ASS]
       FHN := UNSET
   $LE


   and CondExpression[x] = valof
   §CE
       SubExpression[0]
       OuttoBuffer[DUMMY, DUMMY]
       test FHN = UNSET
         ifso  CoercetoBool[x]
         ifnot if IsRevBoolType[x] do ReverseBoolHops[]
     § let f = FHN
       FHN := UNSET
       resultis LastRelExt → f, -f
   $CE


   and ConstantExpression[] = valof
   §CE
       ClearStack[]
       StartOutputMode[CONSTANT]
       Expression[]
       unless NormalOutput = NullProgram do  || see ConstOut
           unless StackPtr = 1 do CompilerError[3201]
       EndOutputMode[]
       StackPtr := 0
       SSP := SSP - 1
       resultis Stack↓1
   $CE


   and SubExpression[p] be
   §E
       let ExpOp = RAND
       §r
           Read[]
         § let x = Ch
           switchon ChType into
           §s
               case SBRA:
                       unless ExpOp = RATOR do
                       §   Error[192, HARD, Ch, BackUp, TRUE]
                           endcase
                       $
                       OuttoBuffer[DUMMY, DUMMY]
                       Application[FNEXIT]
                       break
```

```
            case RBRA:
                    unless ExpOp = RAND do
                    §   Error[191, HARD, Ch, BackUp, PLUS]
                        endcase
                    §
                    Addto[ShuntVec, Ch]
                    SubExpression[0]
                    unless Ch = RKET do CompilerError[3202]
                    ExpOp := RATOR
                    break

            case PLUS:
                    if ExpOp = RAND do
                    §   UpdateCh[UPLUS]
                        endcase
                    §

            case MINUS:
                    if ExpOp = RAND do
                    §   UpdateCh[UMINUS]
                        endcase
                    §
            case MULT:
            case DIV:
            case REM:
            case LSHIFT:
            case RSHIFT:
            case VECOP:
            case EQV:
            case NEQV:
            case EQ:
            case NE:
            case LT:
            case LE:
            case GT:
            case GE:unless ExpOp = RATOR do
                    §   Error[192, HARD, Ch, BackUp, TRUE]
                        endcase
                    §
                    ExpOp := RAND
                    EmptyShunt[]
                    if Prec[LEFT, Ch] ≤ p return
                    unless FHN = UNSET do CoercetoVal[]
                    Addto[ShuntVec, Ch]
                    break

            case LOGAND:
            case LOGOR:
                  § let x = Ch
                    let DefVal = (x = LOGAND → FALSE, TRUE)
                    unless ExpOp = RATOR do
                    §   Error[192, HARD, Ch, BackUp, TRUE]
                        endcase
                    §
                    EmptyShunt[]
                    if Prec[LEFT, Ch] ≤ p return
                    Addto[ShuntVec, SUBEXP]
                    test FHN = UNSET
                      ifso
                        §   SubExpression[Prec[RIGHT, x]]
                            OuttoBuffer[DUMMY, DUMMY]
                            unless Top[ShuntVec] = SUBEXP do
                                CompilerError[3203]
                            Subtractfrom[ShuntVec]
                            test FHN = UNSET
```

```
                                ifso  Addto[ShuntVec, x]
                                ifnot
                                  §   if x = LOGAND do ReverseBoolHops[]
130                                   UpdateS[SSP - 1]
                                      RightVersion[DefVal, DefVal]
                                      ForHopPt[FHN]
                                      if LastRelExt do UpdateS[SSP]
                                      FHN := UNSET
135                               §
                            §
                          ifnot
                            §1  if x = LOGOR do ReverseBoolHops[]
                              § let f = FHN
140                             let LRE = LastRelExt
                                FHN := UNSET
                                SubExpression[Prec[RIGHT, x]]
                                OuttoBuffer[DUMMY, DUMMY]
                                unless Top[ShuntVec] = SUBEXP do
145                                 CompilerError[3204]
                                Subtractfrom[ShuntVec]
                                test FHN = UNSET
                                  ifso
                                    §   let HN1 = HopNo[]
150                                     ForHop[HN1]
                                        ForHopPt[f]
                                        test LRE
                                          ifso  UpdateS[SSP - 1]
                                          ifnot SSP := SSP - 1
155                                     RightVersion[DefVal, DefVal]
                                        ForHopPt[HN1]
                                    §
                                  ifnot
                                    §   if x = LOGAND do ReverseBoolHops[]
160                                     ForHopPt[f]
                                        if LRE do UpdateS[SSP]
                                        if x = LOGAND do ReverseBoolHops[]
                                    §
                            §1
165                   endcase
                    §

                case UPLUS:
                case UMINUS:
170             case LV:
                case RV:
                        unless ExpOp = RAND do
                        §   Error[191, HARD, Ch, BackUp, PLUS]
                            endcase
175                     §
                        Addto[ShuntVec, Ch]
                        break

                case NOT:
180                     unless ExpOp = RAND do
                        §   Error[191, HARD, Ch, BackUp, PLUS]
                            endcase
                        §
                        Addto[ShuntVec, SUBEXP]
185                     SubExpression[Prec[RIGHT, NOT]]
                        OuttoBuffer[DUMMY, DUMMY]
                        unless Top[ShuntVec] = SUBEXP do CompilerError[3205]
                        Subtractfrom[ShuntVec]
                        test FHN = UNSET
                          ifso  Addto[ShuntVec, NOT]
190                       ifnot ReverseBoolHops[]
```

```
                    ExpOp := RATOR
                    endcase

195             case DO:
                case LET:
                case AND:
                case COMMA:
                case RKET:
200             case SKET:
                case ASS:
                case TO:
                case BY:
                case INTO:
205             case CCOLON:
                case SECTBRA:|| in case a global type def follows a let type
                case SECTKET:
                case SEMICOLON:
                case OR:
210             case IFSO:
                case IFNOT:
                case REPEAT:
                case REPEATUNTIL:
                case REPEATWHILE:
215             case ENDBODY:
                case ENDFOR:
                case ENDVALOF:
                    unless ExpOp = RATOR ∨ Ch = SKET do
                    §   Error[192, HARD, Ch, BackUp, TRUE]
220                     endcase
                    §
                    EmptyShunt[]
                    return

225             case COND:
                    unless ExpOp = RATOR do
                    §   Error[192, HARD, Ch, BackUp, TRUE]
                        endcase
                    §
230                 EmptyShunt[]
                    if Prec[LEFT, Ch] ≤ p return
                    OuttoBuffer[DUMMY, DUMMY]
                    if FHN = UNSET do CoercetoBool[COND]
                  § let f, LRE, h = FHN, LastRelExt, HopNo[]
235                 let s = SSP
                    FHN := UNSET
                    Addto[ShuntVec, COND]
                    SubExpression[0]
                    OuttoBuffer[DUMMY, DUMMY]
240                 if BoolValueonStack[] do CoercetoBool[COND]
                    SSP := s    ||  reset after left branch
                    unless Ch = COMMA do CompilerError[3206]
                    ForHop[h]
                    ForHopPt[f]
245                 if LRE do UpdateS[SSP]
                    test FHN = UNSET
                      ifso
                        §   Expression[]
                            OuttoBuffer[DUMMY, DUMMY]
250                         ForHopPt[h]
                        §
                      ifnot
                        §   let F2, H2 = FHN, HopNo[]
                            and LRE2 = LastRelExt
255                         FHN := UNSET
                            SubExpression[0]
```

```
                                OuttoBuffer[DUMMY, DUMMY]
                                if BoolValueonStack[] do
                                    CoercetoBool[COND]
260                             ForHop[H2]
                                test FHN = UNSET
                                  ifso
                                    §   ForHopPt[h]
                                        FHN := F2
265                                     LastRelExt := LRE2
                                        SSP := SSP - 1
                                        CoercetoVal[]
                                    §
                                  ifnot
270                                 §   ForHopPt[F2]
                                        ForHop[FHN]
                                        LastRelExt := LastRelExt ∨ LRE2
                                        ForHopPt[h]
                                    §
275                             ForHopPt[H2]
                            §
                        unless Top[ShuntVec] = COND do
                            CompilerError[3207]
                        Subtractfrom[ShuntVec]
280                     ExpOp := RATOR
                        endcase
                    §

                case VECAP:
285                     unless ExpOp = RATOR do
                        §   Error[192, HARD, Ch, BackUp, TRUE]
                            endcase
                        §
                        Expression[]
290                     unless Ch = SKET do CompilerError[3208]
                        ExpOp := RATOR
                        break

                case VALOF:
295                     unless ExpOp = RAND do
                        §   Error[191, HARD, Ch, BackUp, PLUS]
                            endcase
                        §
                        OuttoBuffer[DUMMY, DUMMY]
300                     Valof[]
                        ExpOp := RATOR
                        break

                case NAME:
305             case NUMBER:
                case CHARACTER:
                case STRING:
                case TRUE:
                case FALSE:
310                     unless ExpOp = RAND do
                        §   Error[191, HARD, Ch, BackUp, PLUS]
                            endcase
                        §
                        ExpOp := RATOR
315                     OuttoBuffer[Ch, ChType]
                        break

                default:CompilerError[3209]
            $s repeat
320     $r repeat
    $E
```

```
    and Valof[] be
325 §V
        let RVHN = VHN
        and RVSSP = VSSP
        and l, g = LineNo, GetNo
        VSSP := SSP + 1
330     VHN := UnusedHopNo[]
        Addto[ShuntVec, Ch]
        Command[]
        unless Ch = ENDVALOF do CompilerError[3210]
        unless HopNoUsed[VHN] do
335     §   let Line, Get = LineNo, GetNo
            LineNo, GetNo := l, g
            Error[260, HARD, DUMMY, NullProgram]
            LineNo, GetNo := Line, Get
        $
340     ForHopPt[VHN]
        SSP := VSSP
        unless Top[ShuntVec] = VALOF do CompilerError[3211]
        Subtractfrom[ShuntVec]
        VSSP := RVSSP
345     VHN := RVHN
    $V


    and EmptyShunt[] be
350 §ES
        let n = Prec[LEFT, ChType]
        while n < Prec[RIGHT, Top[ShuntVec]] do
            test IsRelop[Top[ShuntVec]]
              ifso  BoolRelation[]
355           ifnot ShunttoBuffer[]
        if n = Prec[RIGHT, Top[ShuntVec]] do
            test Ch = RKET ∨ Ch = SKET
              ifso  Subtractfrom[ShuntVec]
              ifnot ShunttoBuffer[]
360 $ES


    and BoolRelation[] be
    §BR
365     let Relop = Top[ShuntVec]
        and Result = true
        Subtractfrom[ShuntVec]
        OuttoBuffer[DUMMY, DUMMY]
        FHN := HopNo[]
370     LastRelExt := false
        §r
            switchon StackSize[] into
            §s
                case 1: if Result = false do
375                     §   StackPtr := StackPtr - 1
                            SSP := SSP - 1
                            endcase
                        §

380             case 0: if Result = false do
                        §   UpdateS[SSP - 1]
                            endcase
                        §

385                     test IsRelop[Top[ShuntVec]]
                          ifso  §   Code[RPsm, SSP - 1]
```

```
                                CondForHop[BoolType[Relop], FHN]
                                LastRelExt := true
                            §
390                   ifnot §   SpecCondForHop[BoolType2[Relop], FHN]
                                return
                            §
                    endcase

395         default:
                    Result := Result ∧
                        ConstOp[Relop, Stack↓(StackPtr - 1),
                            Stack↓StackPtr]
                    StackPtr := StackPtr - 1
400                 SSP := SSP - 1
                    endcase
        §s

        Relop := Top[ShuntVec]
405     test IsRelop[Relop]
          ifso  Subtractfrom[ShuntVec]
          ifnot §   FHN := UNSET
                    test StackSize[] > 0
                      ifso  Stack↓StackPtr := Result
410                   ifnot §   UpdateS[SSP - 1]
                                CodeNsm[Result]
                            §
                    return
                §
415 §r repeat
§BR


and SpecCondForHop[Type, n] be
420 §SCFH
    if (Type = HOPIFEQsm ∨ Type = HOPIFNEsm) do
        if StackSize[] > 0 ∧ Stack↓StackPtr = 0 do
        §   Type := (Type = HOPIFEQsm → HOPIFZEsm, HOPIFNZsm)
            StackPtr := StackPtr - 1
425         SSP := SSP - 1
        §
    CondForHop[Type, n]
§SCFH

430
and IsRelop[CType] = (Prec[RIGHT, CType] = 11)


and BoolType[Op] = valof
435 §BT
    switchon Op into
    §s
        case EQ:resultis HOPIFNEsm
        case NE:resultis HOPIFEQsm
440     case LT:resultis HOPIFGEsm
        case GE:resultis HOPIFLTsm
        case LE:resultis HOPIFGTsm
        case GT:resultis HOPIFLEsm
    §s
445 §BT


and BoolType2[Op] = valof
§BT
450 switchon Op into
    §s
```

```
            case EQ:resultis HOPIFNEsm
            case NE:resultis HOPIFEQsm
            case LT:resultis HOPIFLEsm
            case GE:resultis HOPIFGTsm
            case LE:resultis HOPIFLTsm
            case GT:resultis HOPIFGEsm
        $s
    $BT

    and CoercetoBool[x] be
    §CB
        FHN := HopNo[]
        LastRelExt := false
        OuttoBuffer[DUMMY, DUMMY]
        test StackSize[] > 0
          ifso  §   let Bool = Stack↓StackPtr
                    StackPtr := StackPtr - 1
                    SSP := SSP - 1
                    if IsRevBoolType[x] ≠ (Bool = 0) do ForHop[FHN]
                §
          ifnot CondForHop[(IsRevBoolType[x] → HOPIFNZsm, HOPIFZEsm), FHN]
    $CB

    and CoercetoVal[] be
    §CV
        OuttoBuffer[DUMMY, DUMMY]
        ClearStack[]
        NormalOutput[3, COERCETOVAL, FHN, LastRelExt → SSP, UNSET]
        UpdateSSP[COERCETOVAL]
        FHN := UNSET
    $CV

    and ReverseBoolHops[] be
    §RBH
        let h = FHN
        FHN := HopNo[]
        ForHop[FHN]
        ForHopPt[h]
        if LastRelExt do
        §   UpdateS[SSP]
            LastRelExt := false
        §
    $RBH


and IsRevBoolType[x] = valof
    §IR
        switchon x into
        §s
            case IF:
            case WHILE:
            case COND:
            case REPEATWHILE:
                    resultis false

            case UNLESS:
            case UNTIL:
            case REPEATUNTIL:
                    resultis true

            case TEST:
                    resultis (Ch = IFNOT)
```

```
            default:CompilerError[3212]
        $s
$IR


    and ShunttoBuffer[] be
    §SB
        let t = Top[ShuntVec]
        unless FHN = UNSET do CoercetoVal[]
        OuttoBuffer[t, t]
        Subtractfrom[ShuntVec]
    $SB


    and OuttoBuffer[C, CType] be
    §OB
        let l = LineNo
        LineNo := BufferLine
        BufferLine := l
        switchon CType into
        §s
            case LV:LeftVersion[Buffer, BufferType]
                    endcase

            case ASS:
                    StoreVersion[Buffer, BufferType]
                    endcase

            default:RightVersion[Buffer, BufferType]
                    endcase
        $s
        Buffer := C
        BufferType := CType
        LineNo := BufferLine
    $OB

****
```

```
   let Command[] be
   §C
       Read[]
    5  §r
           let x = Ch
           and OldNRD = NonRecDef
           and HN1, F, C = 0, 0, 0
           and RBHN, RLHN = BHN, LHN
   10      switchon ChType into
           §s
               case IF:
               case UNLESS:
                       HN1 := CondExpression[Ch]
   15                  unless Ch = DO do CompilerError[3301]
                       Command[]
                       ForHopPt[Abs[HN1]]
                       if HopNoUsed[HN1] do UpdateS[SSP]
                       break
   20
               case WHILE:
               case UNTIL:
                       HN1 := HopNo[]
                       BackHopPt[HN1]
   25                  BHN := CondExpression[Ch]
                       unless Ch = DO do CompilerError[3302]
                       LHN := UnusedHopNo[]
                       Command[]
                       ForHopPt[LHN]
   30                  if HopNoUsed[LHN] do UpdateS[SSP]
                       BackHop[HN1]
                       ForHopPt[Abs[BHN]]
                       if HopNoUsed[BHN] do UpdateS[SSP]
                       BHN, LHN := RBHN, RLHN
   35                  break

               case REPSTART:
                     § let HN2 = -UNSET
                       HN1 := HopNo[]
   40                  BHN, LHN := UnusedHopNo[], UnusedHopNo[]
                       BackHopPt[HN1]
                       Command[]
                       ForHopPt[LHN]
                       if HopNoUsed[LHN] do UpdateS[SSP]
   45                  switchon Ch into
                       §s
                           case REPEAT:
                                   Read[]
                                   BackHop[HN1]
   50                              endcase

                           case REPEATWHILE:
                           case REPEATUNTIL:
                                   HN2 := CondExpression[Ch]
   55                              BackHop[HN1]
                                   ForHopPt[Abs[HN2]]
                                   endcase

                           default:CompilerError[3303]
   60                  $s
                       ForHopPt[BHN]
```

```
                if HopNoUsed[HN2] ∨ HopNoUsed[BHN] do UpdateS[SSP]
                BHN, LHN := RBHN, RLHN
                break
65            §

          case TEST:
            § let HN2 = HopNo[]
              HN1 := CondExpression[Ch]
70            unless Ch = DO ∨ Ch = IFSO ∨ Ch = IFNOT do
                  CompilerError[3304]
              Command[]
              unless Ch = OR ∨ Ch = IFSO ∨ Ch = IFNOT do
                  CompilerError[3305]
75            ForHop[HN2]
              ForHopPt[Abs[HN1]]
              if HopNoUsed[HN1] do UpdateS[SSP]
              Command[]
              ForHopPt[HN2]
80            break
            §

          case FOR:
              HN1 := HopNo[]
85            Read[]
              Read[]
              F := FindName[Ch]
              StartNonRecDef[]
              Expression[]
90            unless Ch = TO do CompilerError[3306]
              UpdateNameVal[F, SSP + 1]
              Expression[]
              Inst[SWAPsm]
              BackHopPt[HN1]
95            Inst[DDUPsm]
              C := (Ch = BY → ConstantExpression[], 1)
              EndNonRecDef[OldNRD]
              unless Ch = DO do CompilerError[3307]
              BHN, LHN := HopNo[], UnusedHopNo[]
100           CondForHop[C ≥ 0 → HOPIFGTsm, HOPIFLTsm, BHN]
              Command[]
              ForHopPt[LHN]
              if HopNoUsed[LHN] do UpdateS[SSP]
              Code[INCsm, C]
105           BackHop[HN1]
              ForHopPt[BHN]
              BHN, LHN := RBHN, RLHN
              UpdateS[SSP - 2]
              unless Ch = ENDFOR do CompilerError[3308]
110           Read[]
              break

          case LASS:
              DealwithAss[]
115           break

          case SWITCHON:
            § let RSVec = SwitchVec
              and RSSP = SSP
120           and RCHN, REHN = CHN, EHN
              and RCaseGen = CaseGen
              CaseGen := Generation[PresentLevel]
              Read[]
              SetupSwitchVec[]
125           Expression[]
              unless Ch = INTO do CompilerError[3309]
```

```
                    Code[SWITCHsm, SwitchVec↓3]
                    Command[]
                    CellOutput[SwitchVec + 1]
130                 ForHopPt[EHN]
                    ReturnVec[SwitchVec, SwitchVec↓2]
                    SwitchVec := RSVec
                    UpdateS[RSSP]
                    CHN, EHN := RCHN, REHN
135                 CaseGen := RCaseGen
                    unless Ch = ENDSWITCH do CompilerError[3310]
                    Read[]
                    break
                  §
140
                case CASE:
                    C := ConstantExpression[]
                    test CHN=UNSET ∨ CaseGen≠Generation[PresentLevel]
                      ifso  Error[251, HARD, DUMMY, NullProgram]
145                   ifnot §   UpdateSwitchVec[C]
                                CHN := CHN + 1
                                ForHopPt[CHN]
                            §
                    Read[]
150                 loop

                case DEFAULT:
                    Read[]
                    test EHN=UNSET ∨ CaseGen≠Generation[PresentLevel]
155                   ifso  Error[255, HARD, DUMMY, NullProgram]
                      ifnot §   unless SwitchVec↓5 = EHN do
                                    Error[258, HARD, PresentLevel,
                                        NullProgram]
                                SwitchVec↓5 := EHN - 1
160                             ForHopPt[EHN - 1]
                            §
                    Read[]
                    loop

165             case ENDCASE:
                    if EHN = UNSET do
                        Error[252, HARD, DUMMY, NullProgram]
                    ForHop[EHN]
                    Read[]
170                 break

                case GLOBAL:
                case STATIC:
                case MANIFEST:
175             case TABLE:
                    Read[]
                  § let l, g = LineNo, GetNo
                    F := FindName[Ch]
                    StartNonRecDef[]
180                 C := ConstantExpression[]
                    switchon x into
                    §s
                        case STATIC:
                                test NameType[F] = GLOBAL
185                               ifso  §   SetLitGlobal[NameNo[F],
                                                NameVal[F], C]
                                            RememberGlobal[F]
                                        §
                                  ifnot §   SetLitStatic[NameNo[F], C]
190                                         CheckScope[F, l, g]
                                        §
```

```
                        endcase

                    case GLOBAL:
195                     unless 0 ≤ C ≤ 500 ∨
                                NameVal[F] = UNUSED do
                            Error[261, HARD, C, NullProgram]
                        UpdateNameVal[F, C]
                        CheckScope[F, l, g]
200                     endcase

                    case MANIFEST:
                        UpdateNameVal[F, C]
                        UpdateNameNo[F, SET]
205                     CheckScope[F, l, g]
                        endcase

                    case TABLE:
                          § let n = 1
210                       Write[C]
                          while Ch = COMMA do
                          §   C := ConstantExpression[]
                              n := n + 1
                              Write[C]
215                       §
                          TableMax := TableMax + n
                          test NameType[F] = GLOBAL
                            ifso   §   SetDSegGlobal[NameNo[F],
                                           NameVal[F], n]
220                                    RememberGlobal[F]
                                   §
                            ifnot  §   SetDSegStatic[NameNo[F], n]
                                       CheckScope[F, l, g]
                                   §
225                       endcase
                        §
                $s
                EndNonRecDef[OldNRD]
                loop
230         §

            case LOCDEF:
                  § let RSSP = SSP
                    Read[]
235                 §r
                        Read[]
                        F := FindName[Ch]
                        Read[]
                        SSP := SSP + 1
240                     UpdateNameVal[F, SSP]
                        unless Ch = COMMA break
                        Read[]
                    $r repeat
                    unless Ch = DEF do CompilerError[3311]
245                 SSP := RSSP
                    StartNonRecDef[]
                    Expression[] repeatuntil Ch ≠ COMMA
                    EndNonRecDef[OldNRD]
                    loop
250             §

            case VECDF:
                    Read[]
                    F := FindName[Ch]
255                 SSP := SSP + 1
                    UpdateNameVal[F, SSP]
```

```
                StartNonRecDef[]
                C := ConstantExpression[]
                if C < 0 do Error[262, HARD, DUMMY, NullProgram]
                UpdateNameNo[F, C]
                Code[LPsm, SSP + 1]
                SSP := SSP + C
                EndNonRecDef[OldNRD]
                UpdateS[SSP]
                loop

            case PARAM:
                §r
                    Read[]
                    F := FindName[Ch]
                    SSP := SSP + 1
                    UpdateNameVal[F, SSP]
                    Read[]
                    unless Ch = COMMA break
                    Read[]
                $r repeat
                if PresentOutput = MFDEF do ManFunDefParams[]
                loop

            case FNDF:
            case RTDF:
                  § let RSSP = SSP
                    and G = LocalGenNo
                    ClearStack[]
                    HN1 := HopNo[]
                    Read[]
                    F := FindName[Ch]
                    StartOutputMode[NORMAL]
                    ForHop[HN1]
                    SetCSegGloborStat[F]
                    SSP := -1
                    LocalGenNo := Generation[PresentLevel]
                    Command[]
                    unless Ch = ENDBODY do CompilerError[3312]
                    LocalGenNo := G
                    SSP := RSSP
                    ForHopPt[HN1]
                    EndOutputMode[]
                    Read[]
                    break
                  §

            case MANFNDF:
            case MANRTDF:
                    Read[]
                    ManFunDefinition[x]
                    Read[]
                    break

            case FNBODY:
            case RTBODY:
                  § let RVHN, RCHN, REHN = VHN, CHN, EHN
                    let RInRt = InRt
                    InRt := (Ch = RTBODY)
                    BHN, LHN, VHN, CHN, EHN :=
                        UNSET, UNSET, UNSET, UNSET, UNSET
                    Code[JUMPPTsm, SSP]
                    test Ch = FNBODY
                      ifso  §   Expression[]
                                Code[FNEXITsm, SSP]
                            §
```

```
                    ifnot §   Command[]
                              Code[RTEXITsm, SSP]
                          §
                BHN, LHN, VHN, CHN, EHN :=
                    RBHN, RLHN, RVHN, RCHN, REHN
                InRt := RInRt
                break
              §

        case BREAK:
                if BHN = UNSET do
                    Error[253, HARD, DUMMY, NullProgram]
                SpecForHop[lv BHN]
                Read[]
                break

        case LOOP:
                if LHN = UNSET do
                    Error[257, HARD, DUMMY, NullProgram]
                SpecForHop[lv LHN]
                Read[]
                break

        case RESULTIS:
                if VHN = UNSET do
                    Error[254, HARD, DUMMY, NullProgram]
                Expression[]
                unless VSSP = SSP do
                §   Code[SPsm, VSSP]
                    Code[Ssm, VSSP]
                    SSP := SSP + 1
                §
                SpecForHop[lv VHN]
                SSP := SSP - 1
                break

        case GOTO:
                Expression[]
                Inst[GOTOsm]
                break

        case RETURN:
                test InRt
                  ifnot Error[259, HARD, DUMMY, NullProgram]
                  ifso  Code[RETURNsm, SSP]
                Read[]
                break

        case VALOF:
                Valof[]
                RoutineApp[]
                break

        case RBRA:
                Addto[ShuntVec, Ch]
                Expression[]
                RoutineApp[]
                break

        case NAME:
        case NUMBER:
        case CHARACTER:
        case STRING:
        case TRUE:
        case FALSE:
```

```
                    RightVersion[Ch, ChType]
                    RoutineApp[]
                    break
390
            case LABEL:
                    Read[]
                    F := FindName[Ch]
                    SetCSegGloborStat[F]
395                 Code[JUMPPTsm, SSP]
                    Read[]
                    loop

            case SECTBRA:
400                 § let RSSP = SSP
                    Command[] repeatwhile Ch = SEMICOLON
                    unless ChType = SECTKET ∧
                            ValPart[Ch] = ValPart[x] do
                        CompilerError[3313]
405                 unless SSP = RSSP do UpdateS[RSSP]
                    Read[]
                    break
                  §

410         case LET:
            case AND:
                    Read[]
                    loop

415         default:
                    break
        $s
    $r repeat

420 switchon ChType into
    §s
        case LET:
        case AND:
        case SECTBRA:
425             BackUp[SEMICOLON]

        case SECTKET:
        case SEMICOLON:
        case OR:
430     case IFSO:
        case IFNOT:
        case REPEAT:
        case REPEATUNTIL:
        case REPEATWHILE:
435     case ENDBODY:
        case ENDFOR:
        case ENDVALOF:
        case ENDSWITCH:
        case ENDPROG:
440             return

        default:Error[105, HARD, Ch, DeleteCommand]
    $s
$C
445

and SpecForHop[Loc] be
§SFH
    if rv Loc ≤ 0 do rv Loc := - rv Loc
450 ForHop[rv Loc]
$SFH
```

```
    and SetCSegGloborStat[F] be
455     test NameType[F] = GLOBAL
          ifso  §   SetCSegGlobal[NameNo[F], NameVal[F], NameField[F]]
                    RememberGlobal[F]
                §
          ifnot §   UpdateNameVal[F, UNSET]
460               SetCSegStatic[NameNo[F], NameField[F]]
                  CheckScope[F, LineNo, GetNo]
                §


465 and SetupSwitchVec[] be
    §SSV
        let n = ValPart[Ch]
        let m = MaxVecSize[]
        unless n + 5 ≤ m do
470     §   Peep[2]
            Error[9, HARD, DUMMY, Finish]
        §
        SwitchVec := NewVec[n + 5]
        SwitchVec↓0 := 0
475     SwitchVec↓1 := SWITCH
        SwitchVec↓2 := n + 5
        SwitchVec↓3 := SwitchNo[]
        CHN := HN
        HN := HN + n + 1
480     EHN := HopNo[]
        SwitchVec↓4 := CHN + 1
        SwitchVec↓5 := EHN
    $SSV

485
    and UpdateSwitchVec[a] be
    §USV
        let p = SwitchVec↓0 + 6
        if p > SwitchVec↓2 do CompilerError[3314]
490     SwitchVec↓0 := p - 5
        SwitchVec↓p := a
        for i = 6 to p - 1 do
            if SwitchVec↓i = SwitchVec↓p do
            §   Error[256, HARD, PresentLevel, NullProgram]
495             break
            §
    $USV


500 and RoutineApp[] be
    §RA
        Read[]
        unless Ch = SBRA do
        §   Error[118, HARD, Ch, BackUp, ERROR]
505         return
        §
        Application[RTEXIT] repeatwhile Ch = SBRA
    $RA

510
    and RememberGlobal[F] be
    §RG
        let n = GlobList↓0 + 1
        and v = NameVal[F]
515     and i = ValPart[NameField[F]]
        let Sys = (0 ≤ v ≤ 99 ∧ v ≠ 1) ∨ (300 ≤ v ≤ 399)
```

```
        test n > GlobSize
          ifso  §   if n = GlobSize + 1 do
                        Error[20, SOFT, DUMMY, NullProgram]
520                 if Sys do Error[21, SOFT, i, NullProgram]
                  §
          ifnot GlobList↓(2 × n - 1), GlobList↓(2 × n) :=
                      v ∨ (Sys → 8100000, 0), i
        GlobList↓0 := n
525     if Sys do GlobList↓(-1) := GlobList↓(-1) + 1
    $RG

****
```

```
   let SetUpRA[] be
   §SRA
       SaveIn := In
       BuffStr := ConstructStream[FASTBILATERAL]
       SetNextFn[BuffStr, ErrorNext]
       SetOutBlockRt[BuffStr, NullProgram]
       SetCloseRt[BuffStr, DestroyStream]
       SetUpVector[BuffStr, BUFFERSIZE]
     § let Buffer = Vector[BuffStr] + 1
       SetInBuff[BuffStr, Buffer, Buffer]
       SetOutBuff[BuffStr, Buffer, Buffer + BUFFERSIZE]
   $SRA


   and CloseRA[] be
   §CRA
       Close[BuffStr]
   $CRA


   and ErrorNext[S] = valof
   §   In := SaveIn
       CompilerError[3401]
       resultis 0
   §


   and RestoreIn[S] = valof
   §RI
       SwapIn[]
       SetNextFn[S, ErrorNext]
       resultis Next[In]
   $RI


   and SwapIn[] be
   §S
       let IP, IE = InPtr, InEnd
       test In = SaveIn
         ifso  InPtr, In := 0, BuffStr
         ifnot InPtr, InEnd := InBuffPtr[BuffStr], InBuffEnd[BuffStr]
       test IP = 0
         ifso  In := SaveIn
         ifnot SetInBuff[BuffStr, IP, IE]
   $S


   and DealwithAss[] be
   §DA
       let IP, IE = InPtr, InEnd
       and OBP, OBE = OutBuffPtr[BuffStr], OutBuffEnd[BuffStr]
       and Count = ValPart[Next[In]]
       and LeftLN, RightLN = LineNo, LineNo
       and LeftGet, RightGet = GetNo, GetNo
       and LASSCt = 1
       until OutBuffPtr[BuffStr] ≥ OBE ∨ LASSCt = 0 do
       §u
           Ch := Next[In]
           switchon TypePart[Ch] into
           §s
```

```
            case NEWLINE:
                    RightLN := ValPart[Ch]
                    endcase
65
            case GET:
                    RightGet := ValPart[Ch]
                    endcase

70          case LASS:
                    LASSCt := LASSCt + 1
                    endcase

            case ASS:
75                  LASSCt := LASSCt - 1
                    endcase
        $s
        Out[BuffStr, Ch]
    $u
80  InPtr, InEnd := OBP, OutBuffPtr[BuffStr]
    test Ch = ASS
      ifso  §   let Delim = 0
                for i = 1 to Count do
                §f
85                  LineNo, GetNo := RightLN, RightGet
                    Expression[]
                    test i = Count
                      ifso  Delim := Ch
                      ifnot unless Ch = COMMA do CompilerError[3402]
90                  RightLN, RightGet := LineNo, GetNo
                    SwapIn[]
                    LineNo, GetNo := LeftLN, LeftGet
                    LeftExpression[]
                    unless Ch = (i = Count → ASS, COMMA) do
95                      CompilerError[3403]
                    LeftLN, LeftGet := LineNo, GetNo
                    SwapIn[]
                §f
                LineNo, GetNo := RightLN, RightGet
100             Ch := Delim
                ChType := TypePart[Delim]
            §
      ifnot §   let AssStart = HN
                and RSSP = SSP
105             and AN = HN
                SetNextFn[BuffStr, RestoreIn]
                SwapIn[]
                HN := HN + 2 × Count + 1
                §r
110                 ForHop[GenHopNo[lv AN]]
                    BackHopPt[GenHopNo[lv AN]]
                    UpdateS[RSSP + 1]
                    LeftExpression[]
                §r repeatwhile Ch = COMMA
115             unless Ch = ASS do CompilerError[3404]
                ForHop[GenHopNo[lv AN]]
                AN := AssStart
                ForHopPt[GenHopNo[lv AN]]
                §r
120                 UpdateS[RSSP]
                    Expression[]
                    BackHop[GenHopNo[lv AN]]
                    ForHopPt[GenHopNo[lv AN]]
                §r repeatwhile Ch = COMMA
125             UpdateS[RSSP]
                SetNextFn[BuffStr, ErrorNext]
```

```
                §
        InPtr, InEnd := IP, IE
        SetOutBuff[BuffStr, OBP, OBE]
130 §DA


    and GenHopNo[Loc] = valof
    §GH
135     rv Loc := rv Loc + 1
        resultis rv Loc
    §GH


****
```

```
   let SetUpManFunBuffs[] be
   §SMFB
       MFDefBuff := lv (StructureVec↓StructureLPtr)
       MFAppBuff := NewVec[AppSize]
       PresMFDef := UNKNOWN
       PresMFApp := UNKNOWN
       MFDefPtr := MFDefBuff
       MFAppPtr := MFAppBuff
       MFDefEnd := lv (StructureVec↓StructureNPtr) - 1
       MFAppEnd := MFAppBuff + AppSize
       Store := false
       Link := false
       MFDefBuffFull := false
       MaxAppUsage := 0
   §SMFB


   and CloseManFunBuffs[] be
   §CMFB
       unless (Mode ∧ 1) = 0 do
           ReportS['Def, app sizes = *N, *N', MFDefPtr - MFDefBuff,
               MaxAppUsage]
       ReturnVec[MFAppBuff, AppSize]
   §CMFB


   and ManFunDefinition[Type] be
   §MFD
       let RHop = Hop
       and RSSP = SSP
       and F, G = FindName[Ch], LocalGenNo
       and OldNRD = NonRecDef
       if Type = MANRTDF do UpdateNameType[F, MANRT]
       ClearStack[]
       if NameVal[F] = UNUSED do
       §   StartOutputMode[IGNORE]
           Command[]
           ClearStack[]
           SSP := RSSP
           EndOutputMode[]
           return
       §
       Hop := HopNo[]
       CheckScope[F, LineNo, GetNo]
       SSP := -1
       LocalGenNo := Generation[PresentLevel]
       StartNonRecDef[]
       test MFDefBuffFull
         ifso  §   StartOutputMode[NORMAL]
                   ForHop[Hop]
                   SetManFun[NameNo[F], NameVal[F]]
               §
         ifnot §   StartOutputMode[MFDEF]
                   SetupNewMFDef[F]
               §
       Command[]
       ClearStack[]
       test MFDefBuffFull
         ifso  UpdateNameVal[F, UNSET]
         ifnot
```

```
            §   EndOutputMode[]
                StartOutputMode[NORMAL]
                ForHop[Hop]
                SetManFun[NameNo[F], NameVal[F]]
                OutputMFDef[]
                UpdateNameVal[F, PresMFDef]
                CloseMFDef[]
            §
        NormalOutput[2, ENDMFDEF, NameNo[F]]
        EndNonRecDef[OldNRD]
        LocalGenNo := G
        SSP := RSSP
        ForHopPt[Hop]
        Hop := RHop
        EndOutputMode[]
    $MFD


    and SetupNewMFDef[F] be
    §SNMFD
        let PrevMFDef = PresMFDef
        PresMFDef := MFDefPtr
        if PresMFDef + DEFHEADSIZE ≥ MFDefEnd do
        §   CloseMFDefBuff[F]
            PresMFDef := PrevMFDef
            return
        §
        PresMFDef↓PREV := PrevMFDef
        PresMFDef↓CODETYPE := DIRECT
        PresMFDef↓INITHOPNO := HopNo[]
        PresMFDef↓NOPARAMS := 0
        PresMFDef↓HEADTYPE := NEWMANFUNDEF
        PresMFDef↓NAMEBLOCK := F
        PresMFDef↓PARAMSSAVED := 0
        MFDefPtr := PresMFDef + DEFHEADSIZE
    $SNMFD


    and OutputMFDef[] be
    §OMFD
        let p = PresMFDef + DEFHEADSIZE
        until p ≥ MFDefPtr do
        §   unless CellType[p] = NEWMANFUNDEF do CellOutput[p]
            p := NextCell[p]
        §
    $OMFD


    and CloseMFDef[] be
    §CMFD
        PresMFDef↓CODELENGTH := MFDefPtr - PresMFDef
        PresMFDef↓NOHOPS := HN - PresMFDef↓INITHOPNO + 1
        if PresMFDef↓NOPARAMS = 0 do PresMFDef↓CODETYPE := DIRECT
        IsDefPerfect[]
        PresMFDef := PresMFDef↓PREV
    $CMFD


    and ManFunDefParams[] be
    §MFDP
        let n = SSP + 1
        PresMFDef↓NOPARAMS := n
    $MFDP
```

```
    and OutputtoMFDef[p] be
    §OMFD
        switchon CellType[p] into
130     §s
            case SPsm:
            case LPsm:
                    if CellNo[p] ≤ PresMFDef↓NOPARAMS - 1 do
                        PresMFDef↓CODETYPE := INDIRECT
135                 OuttoMFDef[p]
                    endcase

            case SAsm:
            case SGsm:
140         case LGsm:
            case APPLYsm:
                    Store := true
                    OuttoMFDef[p]
                    endcase
145
            case LOADSTATIC:
                    unless p↓1 = RAsm do Store := true
                    OuttoMFDef[p]
                    endcase
150
            default:
                    if Store do
                        unless PresMFDef↓CODETYPE = INDIRECT do
                            PresMFDef↓CODETYPE := SEMIDIRECT
155
            case RTEXITsm:
            case FNEXITsm:
            case RETURNsm:
                    Store := false
160
            case FORHOPPT:
            case BACKHOPPT:
                    OuttoMFDef[p]
        $s
165 $OMFD


    and OuttoMFDef[p] be
    §OMFD
170     test (MFDefPtr + GenCellLength[p] ≤ MFDefEnd) ∧  MFDefBuffFull
          ifso  §   let q = MFDefPtr
                    CopyCell[p, q]
                    MFDefPtr := NextCell[q]
                §
175       ifnot §   CloseMFDefBuff[PresMFDef↓NAMEBLOCK]
                    CellOutput[p]
                §
    $OMFD

180
    and IsDefPerfect[] be
    §IP
        let m, n, i, p = 0, 0, 0, 0
        if (PresMFDef↓CODETYPE = INDIRECT) ∨ (PresMFDef↓NOPARAMS = 0) return
185     p := PresMFDef + (DEFHEADSIZE + CHKJMPSIZE)
        m := CellNo[p]
        n := PresMFDef↓NOPARAMS - 1
        i := m
        §   unless (CellNo[p] = i) ∧ (CellType[p] = RPsm) return
190         i := i + 1
            p := NextCell[p]
```

```
        § repeatuntil i > n
        until (CellType[p] = FNEXITsm) ∨ (CellType[p] = RTEXITsm) do
        §   if (CellType[p] = RPsm) ∧ (m ≤ CellNo[p] ≤ n) return
195         p := NextCell[p]
        §
        PresMFDef↓PARAMSSAVED := n - m + 1
    $IP

200
    and CloseMFDefBuff[F] be
    §CMFDB
        ClearStack[]
        EndOutputMode[]
205     StartOutputMode[NORMAL]
        ForHop[Hop]
        SetManFun[NameNo[F], NameVal[F]]
        OutputMFDef[]
        MFDefBuffFull := true
210     unless (Mode ∧ 4) = 0 do Error[351, SOFT, NameField[F], NullProgram]
        PresMFDef := PresMFDef↓PREV
    $CMFDB


215 and OutputtoMFApp[p] be
    §OMFA
        switchon CellType[p] into
        §s
            case LINKsm:
220                 test Link
                      ifso  §   NewMFApp[]
                                Link := false
                                (PresentOutput = MFAPP → OuttoMFAppBuff,
                                    CellOutput)[p]
225                       §
                      ifnot OuttoMFAppBuff[p]
                    endcase

            case LOADMANFUN:
230                 if UNDEFINED ≠ CellNo2[p] ≠ UNSET do
                    §   if Link do
                        §   EndOutputMode[]
                            ReportNonApp[]
                            CloseMFApp[]
235                         StartOutputMode[MFAPP]
                        §
                        Link := true
                        OuttoMFAppBuff[p]
                        endcase
240                 §

            default:test Link
                      ifso  §   ReportNonApp[]
                                CloseMFApp[]
245                             Link := false
                                CellOutput[p]
                          §
                      ifnot OuttoMFApp[p]
        §s
250 §OMFA


    and OuttoMFApp[p] be
    §OMFA
255     OuttoMFAppBuff[p]
        if CellType[p] ≠ APPLYsm ∨ PresMFApp = UNKNOWN return
```

```
        if PresMFApp↓APPLYNO = CellNo[p] do SubstituteCode[]
    $OMFA

260
    and NewMFApp[] be
    §NMFA
        let PrevMFApp = PresMFApp
        MFCode := rv( MFAppPtr - 1)
265     if MFAppPtr + APPHEADSIZE ≥ MFAppEnd do
        §   DefaultCloseMFAppBuff[]
            return
        §
        PresMFApp := MFAppPtr
270     PresMFApp↓CODELENGTH := APPHEADSIZE
        PresMFApp↓HEADTYPE := NEWMANFUNAPP
        PresMFApp↓CODEPTR := MFCode
        PresMFApp↓PREV := PrevMFApp
        PresMFApp↓CODETYPE := MFCode↓CODETYPE
275     PresMFApp↓APPLYNO := SSP
        MFAppPtr := PresMFApp + APPHEADSIZE
    $NMFA


280 and OuttoMFAppBuff[p] be
    §OMFAB
        test MFAppPtr + GenCellLength[p] ≤ MFAppEnd
          ifso  §   let q = MFAppPtr
                    CopyCell[p, q]
285                 MFAppPtr := NextCell[q]
                §
          ifnot §   DefaultCloseMFAppBuff[]
                    CellOutput[p]
                §
290 $OMFAB


    and SubstituteCode[] be
    §SC
295     MFCode := PresMFApp↓CODEPTR
        NoParams := MFCode↓NOPARAMS
        HopShift := HN - MFCode↓INITHOPNO + 1
        StackNo := PresMFApp↓APPLYNO
        ParamsSaved := MFCode↓PARAMSSAVED
300     test CanSubstituteCode[]
          ifso  §   let PrevApp = PresMFApp↓PREV
                    Params[PresMFApp↓CODETYPE]
                    P1 := MFCode + (DEFHEADSIZE + CHKJMPSIZE) +
                        RPSIZE × MFCode↓PARAMSSAVED
305                 Substitute[]
                    HN := HN + MFCode↓NOHOPS
                    ManApp := NameType[MFCode↓NAMEBLOCK]
                    CloseMFApp[]
                    PresMFApp := PrevApp
310             §
          ifnot DefaultCloseMFAppBuff[]
    $SC


315 and CanSubstituteCode[] = valof
    §CSC
        let CodeLength = MFCode↓CODELENGTH
        and ParamLength = MFAppPtr - 1 - PresMFApp
        let AppUsage = MFAppPtr - MFAppBuff + 4 × (NoParams - ParamsSaved) +
320             (CodeLength ≥ ParamLength → CodeLength, ParamLength)
        if MaxAppUsage < AppUsage do MaxAppUsage := AppUsage
```

```
        resultis (AppUsage < AppSize)
    $CSC

325
    and Params[Type] be
    §P
        let MaxP = NoParams - ParamsSaved
        and ParamEnd = MFAppPtr - APSIZE
330     and i, r = 0, 0
        ParamBlock := MFAppEnd + 1 - 4 × MaxP
        P1 := PresMFApp + (APPHEADSIZE + LINKSIZE)
        P2 := PresMFApp - LMFSIZE
        until P1 ≥ ParamEnd do
335     §u
            test CellType[NextCell[P1]] = ENDPARAM
              ifnot §   ModTransferCell[i - r - 2] repeatuntil
                            CellType[P1] = ENDPARAM ∧ CellNo[P1] = StackNo
                        unless r ≥ MaxP do
340                         ParamBlock↓(4 × r), ParamBlock↓(4 × r + 1) :=
                                RPsm, i + StackNo
                        i := i + 1
                    §
              ifso  test Type = INDIRECT ∨ r ≥ MaxP
345                   ifso  §   ModTransferCell[i - r - 2]
                                unless r ≥ MaxP do
                                §   ParamBlock↓(4 × r) := RPsm
                                    ParamBlock↓(4 × r + 1) := i + StackNo
                                §
350                             i := i + 1
                            §
                      ifnot switchon CellType[P1] into
                            §s
                                case RPsm:
355                                     if CellNo[P1] > StackNo do
                                            UpdateCellNo[P1, i - r - 2]
                                case Nsm:
                                case LOADMANFUN:
                                case LGsm:
360                                     CopyCell[P1, ParamBlock + 4 × r]
                                        P1 := NextCell[P1]
                                        endcase

                                case RAsm:
365                             case RGsm:
                                case LOADSTATIC:
                                case LOADSTRING:
                                        if Type = DIRECT do
                                        §   CopyCell[P1, ParamBlock + 4 × r]
370                                         P1 := NextCell[P1]
                                            endcase
                                        §

                                default:ModTransferCell[i - r - 2]
375                                     ParamBlock↓(4 × r) := RPsm
                                        ParamBlock↓(4×r + 1) := i + StackNo
                                        i := i + 1
                            §s
            P1 := NextCell[P1]
380         r := r + 1
        §u
        ParamsSaved := ParamsSaved + r - i
        unless r = NoParams do
        §   Error[350, SOFT, r - NoParams, NullProgram]
385         OutputtoMFAppBuff[Ssm, NoParams - r + i + StackNo - 1]
            for s = r to MaxP - 1 do
```

```
            ParamBlock↓(4 × s), ParamBlock↓(4 × s + 1) := Nsm, 0
        $
    $P
390

    and ModTransferCell[n] be
        switchon CellType[P1] into
        §s
395         case APPLYsm:
            case Ssm:
            case JUMPPTsm:
            case RPsm:
            case LPsm:
400         case SPsm:
                if CellNo[P1] > StackNo do UpdateCellNo[P1, n]

            default:CopyCell[P1, P2]
                P2 := NextCell[P2]
405
            case ENDPARAM:
                P1 := NextCell[P1]
        $s

410
    and Substitute[] be
    §S
        let Returning, ReturnSS = false, false
        switchon CellType[P1] into
415     §s
            case RPsm:
                test 0 ≤ CellNo[P1] < NoParams
                  ifso  TransferParam[CellNo[P1]]
                  ifnot EditCell[StackShift[CellNo[P1] + 1]]
420               endcase

            case Ssm:
            case JUMPPTsm:
            case APPLYsm:
425             EditCell[StackNo - ParamsSaved]
                endcase

            case LPsm:
            case SPsm:
430             EditCell[StackShift[CellNo[P1] + 1]]
                endcase

            case COERCETOVAL:
                if CellType[NextCell[P1]] = FNEXITsm do
435             §   FHN := CellNo[P1] + HopShift
                    SSP := SSP - 1
                    LastRelExt := (CellNo2[P1] ≠ UNSET)
                    P1 := NextCell[P1]
                    endcase
440             §
                EditCVCell[HopShift, StackNo - ParamsSaved]
                endcase

            case FORHOPPT:
445         case BACKHOPPT:
            case FORHOP:
            case BACKHOP:
            case CONDFORHOP:
                EditCell[HopShift]
450             endcase
```

```
            case SWITCH:
                    TransferSwitch[]
                    endcase
455
            default:TransferCell[]
                    endcase

            case RETURNsm:
460               § let n = MFCode↓INITHOPNO + HopShift
                    OutputtoMFAppBuff[FORHOP, n]
                    Returning := true
                    ReturnSS := (CellNo[P1] + 1 ≠ ParamsSaved)
                    P1 := NextCell[P1]
465                 endcase
                  §

            case FNEXITsm:
                    if ParamsSaved ≠ CellNo[P1] do
470                     test FHN = UNSET
                          ifso  §   OutputtoMFAppBuff[SPsm, StackNo]
                                    OutputtoMFAppBuff[Ssm, StackNo]
                                §
                          ifnot §   OutputtoMFAppBuff[Ssm, StackNo - 1]
475                               LastRelExt := true
                                §
                    MFAppPtr := P2
                    return

480         case RTEXITsm:
                    if Returning do
                        OutputtoMFAppBuff[FORHOPPT,
                            MFCode↓INITHOPNO + HopShift]
                    if ParamsSaved ≠ CellNo[P1] + 1 ∨ ReturnSS do
485                     OutputtoMFAppBuff[Ssm, StackNo - 1]
                    MFAppPtr := P2
                    return

            case NEWMANFUNDEF:
490                 P1 := NextCell[P1]
                    endcase
        $s repeat
    $S

495
    and StackShift[n] = n ≤ NoParams → StackNo, StackNo - ParamsSaved


    and OutputtoMFAppBuff[a, b, c, d] be
500 §OMAB
        CopyCell[lv a, P2]
        P2 := NextCell[P2]
    §OMAB

505
    and CloseMFApp[] be
    §CMFA
        ClearStack[]
        EndOutputMode[]
510     if PresentOutput ≠ MFAPP do
        §   let p = PresMFApp = UNKNOWN → MFAppBuff, PresMFApp - LMFSIZE
            and q = MFAppPtr
            and RSSP = SSP
            MFAppPtr := p
515         until p ≥ q do
            §   StackOutput[p]
```

```
                p := NextCell[p]
            §
            SSP := RSSP
520     §
    §CMFA


    and DefaultCloseMFAppBuff[] be
525 §DCMFAB
        ClearStack[]
        unless (Mode ∧ 4) = 0 do
            Error[352, SOFT, NameField[MFCode↓NAMEBLOCK], NullProgram]
        until PresentOutput ≠ MFAPP do
530     §   EndOutputMode[]
            unless PresMFApp = UNKNOWN do PresMFApp := PresMFApp↓PREV
        §
      § let p = PresMFApp = UNKNOWN → MFAppBuff, PresMFApp - LMFSIZE
        and q = MFAppPtr
535     MFAppPtr := p
        until p ≥ q do
        §   switchon CellType[p] into
            §s
                case LOADMANFUN:
540                     p↓2 := UNSET

                default:CellOutput[p]

                case NEWMANFUNAPP:
545             case ENDPARAM:
            §s
            p := NextCell[p]
        §
    §DCMFAB
550


    and TransferCell[] be
    §TC
        CopyCell[P1, P2]
555     P1 := NextCell[P1]
        P2 := NextCell[P2]
    §TC


560 and EditCell[n] be
    §EC
        CopyCell[P1, P2]
        UpdateCellNo[P2, n]
        P1 := NextCell[P1]
565     P2 := NextCell[P2]
    §EC


    and EditCVCell[h, s] be
570 §ECC
        CopyCell[P1, P2]
        UpdateCellNo[P2, h]
        unless CellNo2[P2] = UNSET do UpdateCellNo2[P2, s]
        P1 := NextCell[P1]
575     P2 := NextCell[P2]
    §ECC


    and TransferParam[n] be
580 §TP
        CopyCell[ParamBlock + 4 × n, P2]
```

```
        P1 := NextCell[P1]
        P2 := NextCell[P2]
    §TP
585

    and TransferSwitch[] be
    §TS
        CopyCell[P1, P2]
590     P2↓3, P2↓4 := P2↓3 + HopShift, P2↓4 + HopShift
        P1 := NextCell[P1]
        P2 := NextCell[P2]
    §TS

595
    and CopyCell[p, q] be
        for i = 0 to GenCellLength[p] do q↓i := p↓i

****
```

```
   let ConstOp[Op, a, b] = valof
   §CO
       switchon Op into
 5     §s
           case ADDsm:  resultis a + b
           case SUBsm:  resultis a - b
           case MULTsm: resultis a × b
           case DIVsm:  resultis a / b
10         case REMsm:  resultis a rem b
           case ORsm:   resultis a ∨ b
           case ANDsm:  resultis a ∧ b
           case EQVsm:  resultis a = b
           case NEQVsm: resultis a ≠ b
15         case LSHsm:  resultis a lshift b
           case RSHsm:  resultis a rshift b
           case EQ:     resultis a = b
           case NE:     resultis a ≠ b
           case LT:     resultis a < b
20         case LE:     resultis a ≤ b
           case GT:     resultis a > b
           case GE:     resultis a ≥ b
           default:     Error[106, HARD, Op, NullProgram]
                        resultis ERROR
25     §s
   §CO


   and Application[Exit] be
30 §A
       let RSSP = SSP
       Inst[LINKsm]
       Addto[ShuntVec, Ch]
       §   Expression[]
35         if PresentOutput = MFAPP ∧ SSP > RSSP + 1 do
               Code[ENDPARAM, RSSP]
       § repeatwhile Ch = COMMA
       unless Ch = SKET do CompilerError[3601]
       ManApp := DUMMY
40     Code[APPLYsm, RSSP]
       if Exit = RTEXIT do
       §   Read[]
           if Ch = SBRA do Exit := FNEXIT
       §
45     SSP := (Exit = RTEXIT ∨ FHN ≠ UNSET) → RSSP - 1, RSSP
       switchon ManApp into
       §s
           case DUMMY:
                   Code[JUMPPTsm, SSP]
50                 endcase

           case MANFN:
                   unless Exit = FNEXIT do
                       Error[270, HARD, DUMMY, NullProgram]
55                 endcase

           case MANRT:
                   unless Exit = RTEXIT do
                       Error[271, HARD, DUMMY, NullProgram]
60                 endcase
```

```
            default:CompilerError[3602]
        $s
    $A
65

    and FindName[C] = valof
    §FN
        let N = NameBlock[C]
70      if NonRecDef do N := NonRecName[N]
        switchon NameType[N] into
        §s
            case SIMPLE:
                    if Generation[N] > LocalGenNo resultis N
75
            case UNDEFINED:
                    unless Ignoring[] do
                        Error[203, HARD, C, NullProgram]

80          case GLOBAL:
            case MANIFEST:
            case STATIC:
            case TABLE:
            case MANFN:
85          case MANRT:
            case LABEL:
                    resultis N

            default:CompilerError[3603]
90      $s
    $FN


    and NonRecName[N] = valof
95  §NRN
        for i = FIRSTEL to NonRecVec↓PTR do
            if Generation[N] = NonRecVec↓i do
            §   if NameType[N] = MANIFEST ∧ NameNo[N] = SET loop
                if (NameType[N] = MANFN ∨ NameType[N] = MANRT)
100                     ∧ (UNDEFINED ≠ NameVal[N] ≠ UNUSED) loop
                resultis NonRecName[PreviousNameBlock[N]]
            §
        resultis N
    $NRN
105

    and Ignoring[] = valof
    §I
        for i = OutputVec↓PTR to FIRSTEL by -1 do
110         switchon OutputVec↓i into
            §s
                case IGNORE: resultis true

                case NORMAL: resultis false
115         $s
        CompilerError[3604]
    $I


120 and CheckScope[F, Line, Get] be
    §CS
        let G = PreviousNameBlock[F]
        let l, g = LineNo, GetNo
        if G = NILNAME return
125     switchon NameType[F] into
        §s
```

```
            case GLOBAL:
            case MANIFEST:
                    if NameType[F] = NameType[G] do
130                     if NameVal[F] = NameVal[G] return

            case LABEL:
            case STATIC:
            case TABLE:
135         case MANFN:
            case MANRT:
                    if Generation[F] = Generation[G] return
                    LineNo, GetNo := Line, Get
                    Error[206, SOFT, NameField[F], NullProgram]
140                 LineNo, GetNo := l, g
                    return

            default:CompilerError[3506]
        $s
145 $CS


    and DeleteCommand[] be
    §DC
150     switchon ChType into
        §s
            case SECTKET:
            case SEMICOLON:
            case OR:
155         case IFSO:
            case IFNOT:
            case REPEAT:
            case REPEATUNTIL:
            case REPEATWHILE:
160         case ENDBODY:
            case ENDFOR:
            case ENDVALOF:
                    return

165         case SECTBRA:
                    DeleteBlock[]

            default:Read[]
        $s repeat
170 §DC


    and DeleteBlock[] be
    §DB
175     let SectKet = SECTKET ∨ ValPart[Ch]
        Read[]
        switchon ChType into
        §s
            case SECTBRA:
180                 DeleteBlock[]
                    Read[]
                    endcase

            case SECTKET:
185                 unless Ch = SectKet do CompilerError[3606]
                    return

            case ENDPROG:
                    CompilerError[3607]
190
            default:Read[]
```

```
        $s repeat
    $DB

195
    and Prec[Left, CType] = valof
    $P
        switchon CType into
        $s
200         case BOTTOM:
            case VALOF:
            case SUBEXP:
                    resultis 0

205         case TO:
            case BY:
            case DO:
            case ASS:
            case OR:
210         case IFSO:
            case IFNOT:
            case INTO:
            case REPEAT:
            case REPEATWHILE:
215         case REPEATUNTIL:
            case SEMICOLON:
            case LET:
            case AND:
            case CCOLON:
220         case SECTBRA:
            case SECTKET:
            case ENDBODY:
            case ENDFOR:
            case ENDVALOF:
225                 resultis 1

            case RBRA:
            case SBRA:
                    resultis Left → 20, 2
230
            case RKET:
            case SKET:
                    resultis 2

235         case COND:
                    resultis Left → 3, 0

            case COMMA:
                    resultis 4
240
            case NEQV:
                    resultis 5

            case EQV:
245                 resultis 6

            case LOGOR:
                    resultis 7

250         case LOGAND:
                    resultis 8

            case NOT:
                    resultis Left → 20, 9
255
            case LSHIFT:
```

```
            case RSHIFT:
                    resultis Left → 10, 13

260         case EQ:
            case GT:
            case LT:
            case NE:
            case GE:
265         case LE:resultis Left → 12, 11

            case PLUS:
            case MINUS:
                    resultis 14
270
            case UPLUS:
            case UMINUS:
                    resultis Left → 20, 15

275         case MULT:
            case DIV:
            case REM:
                    resultis Left → 17, 16

280         case LV:
            case RV:resultis Left → 20, 18

            case VECAP:
            case VECOP:
285                 resultis 19

            default:CompilerError[3608]
        $s
    $P
290

    and ChangeOutput[] be
    §CO
        let n = OutputVec↓PTR
295     PresentOutput := OutputVec↓n
        switchon PresentOutput into
        §s
            case NORMAL:
                    NormalOutput := NormalOut
300                 CellOutput := CellOut
                    return

            case MFAPP:
                    NormalOutput := AppOut
305                 CellOutput := OutputtoMFApp
                    return

            case MFDEF:
                    NormalOutput := DefOut
310                 CellOutput := OutputtoMFDef
                    return

            case CONSTANT:
                    NormalOutput := ConstOut
315                 CellOutput := ConstOut
                    return

            case IGNORE:
                    NormalOutput := NullProgram
320                 CellOutput := NullProgram
                    return
```

```
            default:CompilerError[3609]
        $s
325 $CO


    and NormalOut[n, a, b, c, d] be
    §NO
330     switchon n into
        §s
            case 4: Write[d]
            case 3: Write[c]
            case 2: Write[b]
335         case 1: Write[a]
                    return

            default:CompilerError[3610]
        $s
340 $NO


    and AppOut[n, a, b, c, d] be
        OutputtoMFApp[lv a]
345

    and DefOut[n, a, b, c, d] be
        OutputtoMFDef[lv a]

350
    and CellOut[p] be
    §CO
        test p↓0 = SWITCH
          ifso  §   let n = p↓1 - 5
355                 TransferOut[Output, p + 5, n]
                    for i = 4 to 0 by -1 do Write[p↓i]
                §
          ifnot NormalOut[CellLength[p], p↓0, p↓1, p↓2, p↓3]
    $CO
360

    and ConstOut[] be
    §CO
        Error[205, HARD, DUMMY, NullProgram]
365     NormalOutput := NullProgram
        CellOutput := NullProgram
    $CO


370 and SetUpVec[Size] = valof
    §SUV
        let v = NewVec[Size]
        v↓SIZE := Size
        v↓PTR := FIRSTEL - 1
375     resultis v
    $SUV


    and Addto[Vec, El] be
380 §A
        let n = Vec↓PTR + 1
        unless n ≤ Vec↓SIZE do Error[8, HARD, DUMMY, Finish]
        Vec↓PTR := n
        Vec↓n := El
385 $A
```

```
    and Subtractfrom[Vec] be
    §S
390     let n = Vec↓PTR - 1
        unless n ≥ (FIRSTEL - 1) do CompilerError[3611]
        Vec↓PTR := n
    $S

395
    and CloseDownVec[Vec] be
    §CDV
        unless Vec↓PTR = (FIRSTEL - 1) do CompilerError[3612]
        ReturnVec[Vec, Vec↓SIZE]
400 $CDV


    and SetUp3b[] be
    §SU3b
405     SetUpManFunBuffs[]
        SetUpRA[]
        OutputVec := SetUpVec[DEPTHSIZE]
        NonRecVec := SetUpVec[DEPTHSIZE]
        ShuntVec := SetUpVec[SHUNTSIZE]
410     StartOutputMode[NORMAL]
        Addto[ShuntVec, BOTTOM]
    $SU3b


415 and CloseDown3b[] be
    §CD3b
        CloseManFunBuffs[]
        CloseRA[]
        Subtractfrom[ShuntVec]
420     EndOutputMode[]
        CloseDownVec[OutputVec]
        CloseDownVec[NonRecVec]
        CloseDownVec[ShuntVec]
    $CD3b
425

****
```

```
Pass 3a
 *** Compilers 26/6/76


    get 'GLOBALS'

    get 'Manifests 0'
  5 get 'Manifests 1'
    get 'Stack Machine Manifests'

    get 'Common Decls'
    get 'Level Procs 1'
 10 get 'Decls 3'
    get 'Decls 3a'

    get 'Pass 3.1'
    get 'Pass 3.2'
 15 get 'Pass 3.3'

****
```

```
Pass 3b
 *** Compilers 26/6/76


    get 'GLOBALS'

    get 'Stream declarations'
  5
    get 'Manifests 0'
    get 'Manifests 1'
    get 'Stack Machine Manifests'

 10 get 'Common Decls'
    get 'Level Procs 1'
    get 'Decls 3'
    get 'Decls 3b'

 15 get 'Pass 3.4'
    get 'Pass 3.5'
    get 'Pass 3.6'


****
```

```
Pass 4.1
 *** Compilers 3/7/76


    let Pass4[] be
    §P4
        SetupPass4[]
 5      Peep[1]
        NewSection[]
        Code[]
        Data[]
        Labels[]
10      Globals[]
        Diagnostics[]
        EndLoad[]
        ClosePass4[]
        Peep[3]
15  §P4


    and NewSection[] be
    §NS
20      Ptr := 0
        OuttoBody[STACKMACHINECODE]
        OutputSection[NEWSECTION, Ptr]
    §NS

25
    and Code[] be
    §C
        SetUpCode[]
        GenerateCode[]
30      Relocate[]
        AlterBody[]
        OutputSection[CODE, Ptr]
        CSize := Ptr
        ResetBody[]
35  §C


    and Data[] be
    §D
40      Ptr := 0
        for i = 1 to SN do
            unless StaticType↓i = CSEG do
                OuttoBody[StaticType↓i = LIT → StaticVal↓i, UNSET]
        for i = 1 to SN do
45          if StaticType↓i = CSEG do
                OuttoBody[UNSET]
        CopytoBody[TableData, TableMax]
        TransferInStrings[]
        unless Ptr = 0 do OutputSection[DATA, Ptr]
50      ReportS['C, D = *N, *N', CSize, Ptr]
    §D


    and Labels[] be
55  §B
        let j = 0
        Ptr := 0
        for i = 1 to SN do
            unless StaticType↓i = CSEG do
60          §   if StaticType↓i = DSEG do
                    Pair[DSEG, DSEG, StaticVal↓i, j]
```

```
                unless StaticLastUse↓i = UNSET do
                    Pair[DSEG, CSEG, j, StaticLastUse↓i]
                j := j + 1
        §
    for i = 1 to SN do
        if StaticType↓i = CSEG do
        §   Pair[CSEG, DSEG, StaticVal↓i, j]
            unless StaticLastUse↓i = UNSET do
                Pair[DSEG, CSEG, j, StaticLastUse↓i]
            j := j + 1
        §
    for i = 1 to MFN do
        unless MFLastUse↓i = UNSET do
            Pair[CSEG, CSEG, MFVal↓i, MFLastUse↓i]
    for i=1 to SL do
       unless StringUse↓i = UNSET do
        Pair[DSEG, CSEG, StringData↓i, StringUse↓i]
    OutputSection[LABELS, Ptr]
    unless (Mode ∧ 1) = 0 do
        ReportS['Labels Size = *N',Ptr]
§B


and Globals[] be
§G
    let RepeatedGlob = false
    Ptr := 0
    for i = 1 to GN do
    §   for j = i + 1 to GN do
            if GlobalLoc↓i = GlobalLoc↓j do
            §   Error[22, HARD, GlobalLoc↓i, NullProgram]
                RepeatedGlob := true
                break
            §
        OuttoBody[GlobalLoc↓i]
        OuttoBody[GlobalType↓i]
        OuttoBody[GlobalVal↓i]
    §
    if RepeatedGlob do Finish[]
    OutputSection[GLOBALS, Ptr]
    unless (Mode ∧ 1) = 0 do
        ReportS['Globals Size = *N',Ptr]
§G


and Diagnostics[] be
§D
    let S = InfromFile[WorkFile['Names']]
    let TotalNames = Next[S]
    for n = 1 to TotalNames do
    §f
        let N = ReadString[S]
        and Used = false
        for i = 1 to DN do
            if DiagName↓i = n do
            §   DiagString↓i := N
                Used := true
            §
        unless Used do ReturnString[N]
    §f
    Close[S]

    Ptr := 0
    OuttoBody[DN]
    for i = DN to 1 by -1 do
```

```
        §f
            OuttoBody[DiagAddr↓i]
            OuttoBody[DiagGlobNo↓i]
            CopyStringtoBody[DiagString↓i]
        §f
        OutputSection[DIAGNOSTICS, Ptr]
        unless (Mode ∧ 1) = 0 do
            ReportS['Diagnostics size = *N', Ptr]
$D


    and EndLoad[] be
    §EL
        OutputSection[ENDLOAD, 0]
    $EL


    and SetupPass4[] be
§SCG
        PassNo := 4
        ReportNo := 0
        if SN + TableMax > CodeSize do
            Error[10, HARD, DUMMY, Finish]
        GlobalType := ProperVec[GN]
        GlobalLoc := ProperVec[GN]
        GlobalVal := ProperVec[GN]
        StaticType := ProperVec[SN]
        StaticLastUse := ProperVec[SN]
        StaticVal := ProperVec[SN]
        MFVal := ProperVec[MFN]
        MFLastUse := ProperVec[MFN]
        SetupStrings[]
        StringUse := ProperVec[SL]
        StringData := ProperVec[SL]
        TableData := ProperVec[TableMax]
        TDPtr := TableMax
        HopList := ProperVec[HN]
        BackHopList := ProperVec[HN]
        SwitchList := ProperVec[SWN]
        DiagAddr := ProperVec[GN + SN]
        DiagGlobNo := ProperVec[GN + SN]
        DiagName := ProperVec[GN + SN]
        DiagString := ProperVec[GN + SN]
        DN := 0
        SetUpBody[CodeSize]
        for i=1 to GN do GlobalType↓i := UNSET
        for i=1 to SN do StaticType↓i := UNSET
        for i=1 to SN do StaticLastUse↓i := UNSET
        for i = 1 to MFN do MFLastUse↓i := UNSET
        for i=1 to SL do StringUse↓i := UNSET
        for i=1 to HN do HopList↓i := UNSET
    $SCG


    and SetupStrings[] be
    §SS
        NIn := InfromFile[WorkFile['Strings']]
        SL := Next[NIn]
$SS


    and TransferInStrings[] be
    §TIS
        for i = 1 to SL do
        §   let x = Next[NIn]
```

```
            let l = (x rshift 8) / 2
            StringData↓i := Ptr
            if Ptr + l ≥ CodeSize do Error[10, HARD, DUMMY, Finish]
195         UpdateBody[Ptr, x]
            TransferIntoBody[NIn, Ptr + 1, l]
            unless StringUse↓i = UNSET do Ptr := Ptr + l + 1
        $
        Close[NIn]
200 $TIS


    and ClosePass4[] be
    §CCG
205     ReturnVec[GlobalType, GN]
        ReturnVec[GlobalLoc, GN]
        ReturnVec[GlobalVal, GN]
        ReturnVec[StaticType, SN]
        ReturnVec[StaticLastUse, SN]
210     ReturnVec[StaticVal, SN]
        ReturnVec[MFVal, MFN]
        ReturnVec[MFLastUse, MFN]
        ReturnVec[StringUse, SL]
        ReturnVec[StringData, SL]
215     ReturnVec[TableData, TableMax]
        ReturnVec[HopList, HN]
        ReturnVec[BackHopList, HN]
        ReturnVec[SwitchList, SWN]
        ReturnVec[DiagAddr, GN + SN]
220     ReturnVec[DiagGlobNo, GN + SN]
        ReturnVec[DiagName, GN + SN]
        ReturnVec[DiagString, GN + SN]
        ReturnBody[CodeSize]
        unless (Mode ∧ 1) = 0 do
225         ReportS['*N hops', HN]
    $CCG


    and ReadString[S] = valof
230 §RS
        let x = Next[S]
        let l = LHalf[x]/2
        let v = ProperVec[l]
        v↓0 := x
235     TransferIn[S, v + 1, l]
        resultis v
    $RS


240 and ReturnString[N] be
        ReturnVec[N, LHalf[N↓0]/2]


    and ProperVec[n] = valof
245 §PV
        let m = MaxVecSize[]
        unless n ≤ m do
        §   Peep[2]
           Error[9,HARD,DUMMY,Finish]
250     §
      § let V = NewVec[n]
        V↓0 := n
        resultis V
    $PV
255
```

```
    and OuttoBody[C] be
    §OV
        if Ptr > CodeSize do
260        Error[7,HARD,DUMMY,Finish]
        UpdateBody[Ptr, C]
        Ptr := Ptr + 1
    $OV

265
    and Pair[ASeg, PSeg, AEntry, PEntry] be
    §P
        OuttoBody[AEntry ∨ ASeg]
        OuttoBody[PEntry ∨ PSeg]
270 $P


    and Relocate[] be
    §R
275     let Shift = FirstCode
        for i = 1 to SN do
            if StaticType↓i = CSEG do
                StaticVal↓i := StaticVal↓i - Shift
        for i = 1 to GN do
280         if GlobalType↓i = CSEGg do
                GlobalVal↓i := GlobalVal↓i - Shift
        for i = 1 to MFN do
            MFVal↓i := MFVal↓i - Shift
        for i = 1 to DN do
285         DiagAddr↓i := DiagAddr↓i - Shift
        for i = 1 to SN do RelocateChain[lv StaticLastUse↓i, Shift]
        for i = 1 to MFN do RelocateChain[lv MFLastUse↓i, Shift]
        for i = 1 to SL do RelocateChain[lv StringUse↓i, Shift]
    $R
290

    and RelocateChain[Lv, Shift] be
    unless rv Lv = UNSET do
    §u
295     let Loc = rv Lv
        rv Lv := Loc - Shift
        until CodeWord[Loc] = UNSET do
        §   let TempLoc = CodeWord[Loc]
            UpdateCode[Loc, TempLoc - Shift]
300         Loc := TempLoc
        §
    $u


****
```

```
Pass 4.2
 *** Compilers 31/7/76


   let SetUpCode[] be
   §SC
       SetCodePtrs[]
 5     CodeFull := true
       SNec := false
       ExitNec := false
       HopNext := false
       HopPtNext := false
10     FnExitNext := false
       NextSS := UNSET
       Inst[RTEXITsm]
       LineNo, GetNo := 0, 0
   $SC
15

   and GenerateCode[] be
   §GC
       Ch := NextNecCh[]
20     ForHopCode[Ch]
   $GC repeatuntil Ch = ENDPROG


   and InstCode[Ch] be
25 §IC
       switchon Ch into
       §s
           case Ssm:
                   SMLCode[Ch, Next[In] + 1]
30                 endcase

           case APPLYsm:
                   StartNewWord[]

35         case INCsm:
           case Nsm:
           case RPsm:
           case SPsm:
                   SMLCode[Ch, Next[In]]
40                 endcase

           case LGsm:
           case RGsm:
           case SGsm:
45         case RAsm:
           case SAsm:
           case LPsm:
                   MLCode[Ch, Next[In]]
                   endcase
50
           case JUMPPTsm:
                   JumpPt[]
                   endcase

55         default:unless LengthField[Ch] = 1 do
                       CompilerError[4201]

           case RTEXITsm:
           case FNEXITsm:
60                 Inst[Ch]
                   endcase
```

```
            case SETCSEGGLOBAL:
                    SetCSegGlobal[]
                    endcase

            case SETCSEGSTATIC:
                    SetCSegStatic[]
                    endcase

            case SETMANFUN:
                    SetManFun[]
                    endcase

            case BACKHOP:
                    BackHop[]
                    endcase

            case BACKHOPPT:
                    BackHopPt[]
                    endcase

            case LOADSTATIC:
                    LoadStatic[]
                    endcase

            case LOADMANFUN:
                    LoadManFun[]
                    endcase

            case LOADSTRING:
                    LoadString[]
                    endcase

            case SWITCH:
                    ReadSwitchBlock[]
                    endcase

            case SWITCHsm:
                    Switch[]
                    endcase

            case ENDPROG:
                    StartNewWord[]
                    return
        $s
    $IC


and NextNecCh[] = valof
    §NNC
        §r
            let Ch = Next[In]
            switchon Ch into
            §s
                case RTEXITsm:
                case RETURNsm:
                      § let x = Next[In]  §
                        unless ExitNec endcase
                        ExitNec := false
                        SNec := false
                        resultis RTEXITsm

                case Ssm:
                        unless SNec do
                        §   Ch := Next[In]
```

```
                        endcase
                    §
                    SNec := false
130                 NextSS := Next[In]
                    endcase

            case FNEXITsm:
                  § let x = Next[In]  §
135
            default:SNec := true
                    ExitNec := true

            case FORHOPPT:
            case BACKHOPPT:
                    resultis Ch

            case ENDMFDEF:
                    Ch := Next[In]
145                 if MFLastUse↓Ch = UNSET do SkipMFDef[Ch]
                    endcase

            case ENDPARAM:
                    Ch := Next[In]
150                 endcase
        $s
    $r repeat
$NNC

155
and SkipMFDef[m] be
§SD
    let Type = Next[In]
    switchon Type into
160 §s
        case SETMANFUN:
              § let n = Next[In]
                let Ch = Next[In]
                if m = n return
165             endcase
              §

        case SWITCH:
                for i = 3 to Next[In] do
170             §   let x = Next[In]  §
                endcase

        default:
                for i = 2 to LengthField[Type] do
175             §   let x = Next[In]  §
                endcase

        case ENDPROG:
        case ENDOFSTRCH:
180             CompilerError[4202]
    $s
$SD repeat


185 and ForHopCode[Ch] be
§FHC
    switchon Ch into
    §s
        case FNEXITsm:
190             FnExitNext := true
                HopNext := false
```

```
                HopPtNext := false
                NextSS := UNSET
                InstCode[Ch]
195             return

        case FORHOP:
                HopNo := Next[In]
                NextSS := UNSET
200             SNec := (HopList↓HopNo > 0)
                if (HopPtNo = HopNo) ∧ HopPtNext return
                HopNext := true
                FnExitNext := false
                HopPtNext := false
205             return

        case CONDFORHOP:
                §   let n = Next[In]
                    let CondType = Next[In]
210                 if HopNext ∧ NextSS = UNSET ∧ CodeFull ∧
                            Abs[HopList↓n] = NextCodePos[] do
                    §   HopNext := false
                        n := HopNo
                        CondType := RevCondType[CondType]
215                 §
                    if HopNext do InsertForHop[]
                    SetStackifNec[]
                    ResetPtrs[]
                    MLCode[CondType, HopLength[n]]
220                 return
                §

        case SETLITGLOBAL:
                SetLitGlobal[]
225             return

        case SETLITSTATIC:
                SetLitStatic[]
                return
230
        case SETDSEGGLOBAL:
                SetDSegGlobal[]
                return

235     case SETDSEGSTATIC:
                SetDSegStatic[]
                return

        case SPsm:
240             Ch := Next[In]
                if ((WordafterHop[HopNo] = (FNEXITsm lshift 8) ∧
                        HopNext) ∨ FnExitNext) ∧ Ch = NextSS do
                §   NextSS := UNSET
                    return
245             §
                if HopNext do InsertForHop[]
                SetStackifNec[]
                ResetPtrs[]
                SMLCode[SPsm, Ch]
250             return

        default:if HopNext do InsertForHop[]
                SetStackifNec[]

255     case RTEXITsm:
                ResetPtrs[]
```

```
                InstCode[Ch]
                return

        case COERCETOVAL:
                if HopNext do InsertForHop[]
                SetStackifNec[]
                StartNewWord[]
                SCode[Nsm, 0]
                HopPtNo := Next[In]
                NextSS := Next[In]
                SetStackifNec[]
                StartNewWord[]
                HopList↓HopPtNo := NextCodePos[]
                test HopNext
                  ifso  InsertForHop[]
                  ifnot SCode[HOPsm, 1]
                Inst[TRUEsm]
                ResetPtrs[]
                return

        case FORHOPPT:
                HopPtNo := Next[In]
                HopPtNext := true
                if HopNext ∧ NextSS = UNSET do
                §   HopList↓HopPtNo := HopList↓HopNo
                    return
                $
                if HopNext do InsertForHop[]
                SetStackifNec[]
                StartNewWord[]
                HopList↓HopPtNo := (NextSS = UNSET → NextCodePos[],
                    -NextCodePos[])
                NextSS := UNSET
                HopNext := false
                return
    $s
$FHC


and ResetPtrs[] be
§RP
    HopNext := false
    HopPtNext := false
    FnExitNext := false
    NextSS := UNSET
$RP


and InsertForHop[] be
§IFH
    let l = HopLength[HopNo]
    if l = 0 ∧ CodeFull return
    switchon WordafterHop[HopNo] into
    §s
        case FNEXITsm lshift 8:
                Inst[FNEXITsm]
                return

        case RTEXITsm lshift 8:
                Inst[RTEXITsm]
                return

        default:SMLCode[HOPsm, l]
    $s
$IFH
```

```
    and JumpPt[] be
325 §JP
        let n = Next[In] + 1
        unless 0 ≤ n ≤ 255 do
        §   LCode[Ssm, n]
            n := 0
330     §
        StartNewWord[]
        MCode[JUMPPTsm, n]
    §JP

335
    and SetLitStatic[] be
    §SLS
        let n = Next[In]
        StaticType↓n := LIT
340     StaticVal↓n := Next[In]
    §SLS


    and SetLitGlobal[] be
345 §SLG
        let n = Next[In]
        GlobalType↓n := ACTUALg
        GlobalLoc↓n := Next[In]
        GlobalVal↓n := Next[In]
350 §SLG


    and SetDSegGlobal[] be
    §SDG
355     let n = Next[In]
        GlobalType↓n := DSEGg
        GlobalLoc↓n := Next[In]
        AddTable[Next[In]]
        GlobalVal↓n := SN + TDPtr
360 §SDG


    and SetDSegStatic[] be
    §SDS
365     let n = Next[In]
        StaticType↓n := DSEG
        AddTable[Next[In]]
        StaticVal↓n := SN + TDPtr
    §SDS
370

    and AddTable[n] be
    §AT
        TDPtr := TDPtr - n
375     if TDPtr < 0 do CompilerError[4203]
        for i = TDPtr + n - 1 to TDPtr by -1 do TableData↓i := Next[In]
    §AT


380 and SetCSegStatic[] be
    §SCS
        let n = Next[In]
        StartNewWord[]
        StaticType↓n := CSEG
385     StaticVal↓n := NextCodePos[]
      § let d = DiagNo[]
```

```
        DiagAddr↓d := NextCodePos[]
        DiagGlobNo↓d := -1
        DiagName↓d := ValPart[Next[In]]
390     DiagString↓d := UNSET
    §SCS


    and SetCSegGlobal[] be
395 §SCG
        let n = Next[In]
        StartNewWord[]
        GlobalType↓n := CSEGg
        GlobalVal↓n := NextCodePos[]
400     GlobalLoc↓n := Next[In]
      § let d = DiagNo[]
        DiagAddr↓d := NextCodePos[]
        DiagGlobNo↓d := GlobalLoc↓n
        DiagName↓d := ValPart[Next[In]]
405     DiagString↓d := UNSET
    §SCG


    and SetManFun[] be
410 §SMF
        let n = Next[In]
        StartNewWord[]
        MFVal↓n := NextCodePos[]
        n := Next[In]
415 §SMF


    and LoadManFun[] be
    §LMF
420     let p = CodePos[]
        and n = Next[In]
        LCode[Nsm, MFLastUse↓n]
        MFLastUse↓n := p
        n := Next[In]
425 §LMF


    and LoadStatic[] be
    §LS
430     let Type = Next[In]
        and p = CodePos[]
        let n = Next[In]
        LCode[Type, StaticLastUse↓n]
        StaticLastUse↓n := p
435 §LS


    and LoadString[] be
    §LS
440     let p = CodePos[]
        let n = Next[In]
        LCode[Nsm, StringUse↓n]
        StringUse↓n := p
    §LS
445

    and BackHop[] be
    §BH
        let n = Next[In]
450     HopList↓n := NextCodePos[]
        BackHopList↓n := CodePos[]
```

```
        LCode[HOPsm, DUMMY]
    $BH

455
    and BackHopPt[] be
    §BHP
        let n = Next[In]
        StartNewWord[]
460     UpdateCode[BackHopList↓n, -HopLength[n]]
    $BHP


    and ReadSwitchBlock[] be
465 §RSB
        let n = Next[In]
        let v = ProperVec[n - 2]
        TransferIn[In, v + 1, n - 2]
        SwitchList↓(v↓1) := v
470 $RSB


    and Switch[] be
    §S
475     let v = SwitchList↓(Next[In])
        let h, n = v↓2, v↓0
        StartNewWord[]
        WordtoCode[HopLength[v↓3] + 2]
        for i = 4 to n do
480     §   WordtoCode[v↓i]
            WordtoCode[HopLength[h + n - i] + 2]
        §
        WordtoCode[3 - n]
        Inst[SWITCHsm]
485     ReturnVec[v, n]
    $S


    and StartNewWord[] be
490     unless CodeFull do
        §   DecCodeBodyPtrs[]
            CodeFull := true
            UpdateCode[FirstCode, (FIRSTMEDINST ≤ SpareByte ≤ LASTMEDINST →
                                   SpareByte, SpareByte lshift 8)]
495     §


    and SetStackifNec[] be
        unless NextSS = UNSET do SMLCode[Ssm, NextSS + 1]
500

    and HopLength[n] = (Abs[HopList↓n] - NextCodePos[])


505 and SMLCode[Type, Length] be
        test 0 ≤ Length ≤ 15
          ifso  SCode[Type, Length]
          ifnot MLCode[Type, Length]

510
    and MLCode[Type, Length] be
        test 0 ≤ Length ≤ 255
          ifso  MCode[Type, Length]
          ifnot LCode[Type, Length]
515
```

```
    and SCode[Type, Length] be BytetoCode[ShortVersion[Type] ∨ Length]


520 and MCode[Type, Length] be
    §MC
        BytetoCode[Length]
        BytetoCode[MediumVersion[Type]]
    $MC
525

    and LCode[Type, Length] be
    §LC
        WordtoCode[Length]
530     BytetoCode[LongVersion[Type]]
    $LC


    and ShortVersion[Type] = InstField[Type] lshift 4
535

    and MediumVersion[Type] =
            0 ≤ InstField[Type] ≤ 817 → InstField[Type] ∨ MEDIUM,
                                        InstField[Type + 1]
540

    and LongVersion[Type] =
            0 ≤ InstField[Type] ≤ 817 → InstField[Type] ∨ LONG,
                                        InstField[Type]
545

    and Inst[T] be BytetoCode[InstField[T]]


550 and BytetoCode[W] be
        test CodeFull
          ifso  §   CodeFull := false
                    SpareByte := W
                §
555       ifnot §   DecCodeBodyPtrs[]
                    CodeFull := true
                    UpdateCode[FirstCode, SpareByte ∨ (W lshift 8)]
                §

560
    and WordtoCode[W] be
    §WC
        DecCodeBodyPtrs[]
        UpdateCode[FirstCode, W]
565 $WC


    and DecCodeBodyPtrs[] be
    §DBP
570     FirstCode := FirstCode - 1
        if FirstCode < FirstWord do Error[7, HARD, DUMMY, Finish]
    $DBP

****
```

```
Pass 4
 *** Compilers 3/7/76


    get 'GLOBALS'
    get 'Misc functions'

  5 get 'Stack Machine Manifests'

    get 'Common Decls'
    get 'Decls 4'

 10 get 'Pass 4.1'
    get 'Pass 4.2'


****
```

```
 *** Compilers 15/2/77


    let PP[] be
    §P
        let c = NewVec[2]
 5      c↓2 := ClosePP
        PassNo := 0
        LineNo := 1
        GetNo := 0
        ReportNo := 0
10      Send := NormalSend
        DeclIndex := LookUp['Decl', 'Index', SystemIndex]
        SetUpTables[]
        NullNameNo := InsertNullName[]
        Peep[1]
15      EnterinCUChain[c]
        NextCh[] repeatwhile Ch = '*n'
        PreProc[] repeatuntil EndofPP
        Write[ENDPROG]
        Send := NullProgram
20      until ProcVec↓ENDCH = P_End_of_Input do DenestGets[]
        Close[Output]
        ClosePP[]
        RemovefromCUChain[c]
        ReturnVec[c, 2]
25      ReturnTables[]
        Peep[3]
    §P


30  and SetUpTables[] be
    §SUT
        NameMap := BitMap[NameSize + 26]
        for i = 1 to 26 do EnterintoMap[NameMap, i]

35      NameTable := NewTable[NameSize, 51]
        NameTableHash := NewTable[NameSize × 2, 0]
        NameTableLink := NewTable[NameSize, 0]
        NumTable := NewTable[NumberSize, 52]
        StringTable := NewTable[StringSize, 53]
40
        SetUpProcVec[]
        GetList := NewVec[GETLISTSIZE]
        GetList↓0 := 0
        TextVec := NewVec[TextSize]
45      Stack := TextVec
        ResetStack[]
    §SUT


50  and ReturnTables[] be
    §RT
        ReturnVec[TextVec, TextSize]
        ReturnVec[GetList, GETLISTSIZE]
        ReturnVec[ProcVec, MAXCH]
55
        ReturnTable[NameTable]
        ReturnTable[NameTableHash]
        ReturnTable[NameTableLink]
        ReturnTable[NumTable]
60      ReturnTable[StringTable]
    §RT
```

```
   and PrintUsage[] be
65     unless (Mode ∧ 1) = 0 do
       §PU
           ReportTable['name', NameTable]
           ReportTable['number', NumTable]
           ReportTable['string', StringTable]
70         ReportS['*nText size = *N', Stack - TextVec]
       §PU


   and ReportTable[String, Table] be
75 §RT
       let n = Table↓PTR - OFFSET
       OutS[ReportStream, '*N *S*C ', n, String, n = 1 → NULL, `s`]
   §RT

80
   and ClosePP[] be
   §   PrintUsage[]
       SendTables[]
   §
85

   and TermPP[] be
   §TPP
       until ProcVec↓ENDCH = P_End_of_Input do DenestGets[]
90     Finish[]
   §TPP


   and NewTable[n, x] = valof
95 §NT
       let v = NewVec[n + OFFSET]
       v↓SIZE := n + OFFSET
       v↓PTR  := FIRSTEL - 1
       v↓REP := x
100    v↓FIRSTEL := NULL
       resultis v
   §NT


105 and ReturnTable[t] be
       ReturnVec[t, t↓SIZE]


   and SetUpProcVec[] be
110 §SUPV
       ProcVec := NewVec[MAXCH]
       for i = 0 to MAXCH do ProcVec↓i := P_Illegal_Ch
       FillProcVec[]
       ProcVec↓ENDCH := P_End_of_Input
115 §SUPV


   and P_Illegal_Ch[] be
   §I
120    Error[55, HARD, Ch, NullProgram]
       Send[ERROR, 0]
       NextCh[]
   §I

125
   and Delete[Cat] be
```

```
        while Is[Ch, Cat] do NextCh[]


130 and P_End_of_Input[] be EndofPP := true


    and Adjoin_to[Table] = valof
    §AD
135     let n = Table↓PTR + 1
        if n > Table↓SIZE do
            Error[Table↓REP, HARD, DUMMY, TermPP]
        Table↓n := Top[]
        Table↓PTR := n
140     ResetStack[]
        resultis n - OFFSET
    §AD


145 and AddVec_to[Table] = valof
    §AV
        let n = Table↓PTR + 1
        if n > Table↓SIZE do
            Error[Table↓REP, HARD, DUMMY, TermPP]
150     Table↓n := Stack
        Table↓PTR := n
        StepOnStack[]
        resultis n - OFFSET
    §AV
155

    and StepOnStack[] be
    §SOS
        Stack := Stack + StackPtr/2 + 1
160     if Stack + (MAXLENGTH + 2) > TextVec + TextSize do
            Error[60, HARD, DUMMY, TermPP]
        ResetStack[]
    §SOS

165
    and InsertNullName[] = valof
    §INN
        ResetStack[]
        resultis InsertName[]
170 §INN


    and InsertName[] = Insert_in[NameTable,NameTableHash,NameTableLink] + 26

175
    and Insert_in[Table, Hash, Link] = valof
    §IN
        let h = StackHash[]
        let n = Find[h, Hash]
180     PackStack[]
        test n = NULL
          ifso
            §   n := AddVec_to[Table] + OFFSET
                AddHash[n, h, Hash]
185             Link↓n := NULL
                resultis n - OFFSET
            §
          ifnot
            §r
190             test EqS[Table↓n, Stack]
                  ifso
```

```
                    §   if Table = NameTable do
                            EnterintoMap[NameMap, n + 26 - OFFSET]
                        ResetStack[]
195                     resultis n - OFFSET
                    §
                  ifnot test Link↓n = NULL
                          ifso
                            §   let p = AddVec_to[Table] + OFFSET
200                             Link↓n := p
                                Link↓p := NULL
                                resultis p - OFFSET
                            §
                          ifnot n := Link↓n
205         $r repeat
    §IN


    and AddHash[n, h, Hash] be
210 §AH
        let p = Hash↓PTR + 2
        Hash↓PTR := p
        Hash↓(p-1) := n
        Hash↓p :=h
215     Hash↓(p+1) := NULL
    $AH


    and Wrong[String]  = valof
220 §CI
        let n = ((String↓0) rshift 8)/2
        for i = 0 to n do
            test Ch = (8377 ∧ String↓i)
            ifso NextCh[]
225         ifnot resultis true
        resultis false
    $CI

****
```

```
   let FillProcVec[] be
   §10
       ProcVec↓0 := P_b
       ProcVec↓'*p' := P_b
       ProcVec↓'*4' := P_b
       ProcVec↓'*s' := P_b
       ProcVec↓'_' := P_b
       for i = '0' to '9' do ProcVec↓i := P_d
       for i = 'a' to 'z' do ProcVec↓i := P_l
       ProcVec↓'*'' := CharacterRt
       ProcVec↓148 := CharacterRt
       ProcVec↓20 := StringRt
       ProcVec↓'*'' := StringRt
       ProcVec↓'[' := StringRt
       for i = 'A' to 'Z' do ProcVec↓i := P_u
       for i = 'A' to 'Z' do ProcVec↓i := P_v
       for i = 'a' to 'z' do ProcVec↓i := P_v
       ProcVec↓'*n' := NLRt
       ProcVec↓'↪' := P_22
       ProcVec↓'∨' := P_26
       ProcVec↓'×' := P_27
       ProcVec↓'≠' := P_29
       ProcVec↓'↓' := P_31
       ProcVec↓'(' := P_40
       ProcVec↓')' := P_41
       ProcVec↓'**' := P_42
       ProcVec↓'+' := P_43
       ProcVec↓',' := P_44
       ProcVec↓'-' := P_45
       ProcVec↓'/' := P_47
       ProcVec↓':' := P_58
       ProcVec↓';' := P_59
       ProcVec↓'<' := P_60
       ProcVec↓'=' := P_61
       ProcVec↓'>' := P_62
       ProcVec↓'[' := P_91
       ProcVec↓']' := P_93
       ProcVec↓'∧' := P_94
       ProcVec↓'§' := P_123
       ProcVec↓'|' := P_124
       ProcVec↓'$' := P_125
       ProcVec↓' ' := P_126
       ProcVec↓'≠' := P_157
       ProcVec↓'8' := P_184
       ProcVec↓'≤' := P_188
       ProcVec↓'=' := P_189
       ProcVec↓'≥' := P_190
       ProcVec↓'a' := P_225
       ProcVec↓'b' := P_226
       ProcVec↓'c' := P_227
       ProcVec↓'d' := P_228
       ProcVec↓'e' := P_229
       ProcVec↓'f' := P_230
       ProcVec↓'g' := P_231
       ProcVec↓'i' := P_233
       ProcVec↓'l' := P_236
       ProcVec↓'m' := P_237
       ProcVec↓'o' := P_239
       ProcVec↓'r' := P_242
       ProcVec↓'s' := P_243
```

```
        ProcVec↓‘t‘ := P_244
        ProcVec↓‘u‘ := P_245
        ProcVec↓‘v‘ := P_246
 65     ProcVec↓‘w‘ := P_247
    §10


    and P_b[] be                                                || Ch is CAT_b
 70 §20
        NextCh[]
        while Is[Ch, CAT_b] do NextCh[]
    §20

 75
    and P_d[] be                                                || Ch is CAT_d
    §30
        unless DecimalFn[] return
        Send[NUMBER, Adjoin_to[NumTable] + 500]
 80 §30


    and P_l[] be                                                || Ch is CAT_l
    §40
 85     Send[NAME, Ch - (‘a‘ - 1)]
        NextCh[]
    §40


 90 and P_u[] be                                                || Ch is CAT_u
    §70
        PushCh_NextCh[]
        while Is[Ch, CAT_a] do PushCh_NextCh[]
        if TooBig[] do
 95     §   Error[61, HARD, DUMMY, NullProgram]
            Send[ERROR, 0]
            ResetStack[]
            return
        §
100     Send[NAME, InsertName[]]
    §70


    and P_v[] be                                                || Ch is CAT_v
105 §80
        Error[56, HARD, Ch, Delete, CAT_v]
        Send[ERROR, 0]
    §80

110
    and P_22[] be                                                || Ch is ‘↪‘
    §90
        NextCh[]
        Send[COND, 0]
115 §90


    and P_26[] be                                                || Ch is ‘∨‘
    §100
120     NextCh[]
        Send[LOGOR, 0]
    §100


125 and P_27[] be                                                || Ch is ‘×‘
    §110
```

```
        NextCh[]
        Send[MULT, 0]
    §110
130

    and P_29[] be                                           || Ch is '≠'
    §120
        NextCh[]
135     Send[NE, 0]
    §120


    and P_31[] be                                           || Ch is '↓'
140 §130
        NextCh[]
        Send[VECOP, 0]
    §130

145
    and P_40[] be                                           || Ch is '('
    §140
        NextCh[]
        Send[RBRA, 0]
150 §140


    and P_41[] be                                           || Ch is ')'
    §150
155     NextCh[]
        Send[RKET, 0]
    §150


160 and P_42[] be                                          || Ch is '**'
    §160
        NextCh[]
        if Ch = '[' do
        §161
165         NextCh[]
            Send[VECAP, 0]
            return
        §161
        Error[78, SOFT, DUMMY, NullProgram]
170     Send[MULT, 0]
    §160


    and P_43[] be                                           || Ch is '+'
175 §170
        NextCh[]
        Send[PLUS, 0]
    §170

180
    and P_44[] be                                           || Ch is ','
    §180
        NextCh[]
        Send[COMMA, 0]
185 §180


    and P_45[] be                                           || Ch is '-'
    §190
190     NextCh[]
        Send[MINUS, 0]
```

```
    §190


195 and P_47[] be                                           || Ch is '/'
    §200
        NextCh[]
        Send[DIV, 0]
    §200
200

    and P_58[] be                                           || Ch is ':'
    §210
        NextCh[]
205     switchon Ch into
        §211
            case '=':
                    NextCh[]
                    Send[ASS, 0]
210                 return

            default:Send[COLON, 0]
                    return
        §211
215 §210


    and P_59[] be                                           || Ch is ';'
    §220
220     NextCh[]
        Send[SEMICOLON, 0]
    §220


225 and P_60[] be                                           || Ch is '<'
    §230
        NextCh[]
        Send[LT, 0]
    §230
230

    and P_61[] be                                           || Ch is '='
    §240
        NextCh[]
235     Send[EQ, 0]
    §240


    and P_62[] be                                           || Ch is '>'
240 §250
        NextCh[]
        Send[GT, 0]
    §250

245
    and P_91[] be                                           || Ch is '['
    §260
        NextCh[]
        Send[SBRA, 0]
250 §260


    and P_93[] be                                           || Ch is ']'
    §270
255     NextCh[]
        Send[SKET, 0]
```

```
    §270


260 and P_94[] be                                              || Ch is '∧'
    §280
        NextCh[]
        Send[LOGAND, 0]
    §280
265

    and P_123[] be                                             || Ch is '§'
    §290
        NextCh[]
270     while Is[Ch, CAT_a] do PushCh_NextCh[]
        MSCRt[]
        if TooBig[] do
        §   Error[62, HARD, DUMMY, NullProgram]
            Send[ERROR, 0]
275         ResetStack[]
            return
        §
        Send[SECTBRA, InsertName[]]
    §290
280

    and P_124[] be                                             || Ch is '|'
    §300
        NextCh[]
285     if Wrong['|'] do
        §   Error[63, HARD, DUMMY, NullProgram]
            Send[ERROR, 0]
            return
        §
290     while Is[Ch, CAT_c] do NextCh[]
    §300


    and P_125[] be                                             || Ch is '§'
295 §310
        NextCh[]
        while Is[Ch, CAT_a] do PushCh_NextCh[]
        if TooBig[] do
        §   Error[62, HARD, DUMMY, NullProgram]
300         Send[ERROR, 0]
            ResetStack[]
            return
        §
        Send[SECTKET, InsertName[]]
305 §310


    and P_126[] be                                             || Ch is ' '
    §320
310     NextCh[]
        Send[NOT, 0]
    §320


315 and P_157[] be                                             || Ch is '≠'
    §330
        NextCh[]
        Send[NEQV, 0]
    §330
320
```

```
    and P_184[] be                                          || Ch is '8'
    §340
        NextCh[]
325     while Is[Ch, CAT_b] do NextCh[]
        unless Is[Ch, CAT_o] do
        §   Error[64, HARD, Ch, NullProgram]
            Send[ERROR, 0]
            return
330     §
        unless OctalFn[] return
        Send[NUMBER, Adjoin_to[NumTable] + 500]
    §340

335
    and P_188[] be                                          || Ch is '<'
    §350
        NextCh[]
        Send[LE, 0]
340 §350


    and P_189[] be                                          || Ch is '='
    §360
345     NextCh[]
        Send[EQV, 0]
    §360


350 and P_190[] be                                          || Ch is '>'
    §370
        NextCh[]
        Send[GE, 0]
    §370
355

****
```

```
   and P_225[] be                                              || Ch is ‘a‘
   §00
       NextCh[]
 5     if Wrong[’n d’] do
       §   Error[56, HARD, ‘a‘, Delete, CAT_v]
           Send[ERROR, 0]
           return
       §
10     Send[AND, 0]
   §00


   and P_226[] be                                              || Ch is ‘b‘
15 §10
       NextCh[]
       switchon Ch into
       §11
           case ‘e‘:
20                 NextCh[]
                   Send[BE, 0]
                   return

           case ‘r‘:
25                 NextCh[]
                   if Wrong[’e a k’] do
                   §   Error[56, HARD, ‘b‘, Delete, CAT_v]
                       Send[ERROR, 0]
                       return
30                 §
                   MSCRt[]
                   Send[BREAK, 0]
                   return

35         case ‘y‘:
                   NextCh[]
                   Send[BY, 0]
                   return

40         default:Error[56, HARD, ‘b‘, Delete, CAT_v]
                   Send[ERROR, 0]
       §11
   §10

45
   and P_227[] be                                              || Ch is ‘c‘
   §20
       NextCh[]
       if Wrong[’a s e’] do
50     §   Error[56, HARD, ‘c‘, Delete, CAT_v]
           Send[ERROR, 0]
           return
       §
       MSCRt[]
55     Send[CASE, 0]
   §20


   and P_228[] be                                              || Ch is ‘d‘
60 §30
       NextCh[]
```

```
        switchon Ch into
        §31
            case ‘e‘:
                    NextCh[]
                    if Wrong[’f a u l t’] do
                    §   Error[56, HARD, ‘d‘, Delete, CAT_v]
                        Send[ERROR, 0]
                        return
                    §
                    MSCRt[]
                    Send[DEFAULT, 0]
                    return

            case ‘o‘:
                    NextCh[]
                    Send[DO, 0]
                    return

            default:Error[56, HARD, ‘d‘, Delete, CAT_v]
                    Send[ERROR, 0]
        §31
    §30

    and P_229[] be                                          || Ch is ‘e‘
    §40
        NextCh[]
        switchon Ch into
        §41
            case ‘l‘:
                    NextCh[]
                    if Wrong[’s e’] do
                    §   Error[56, HARD, ‘e‘, Delete, CAT_v]
                        Send[ERROR, 0]
                        return
                    §
                    Send[ELSE, 0]
                    return

            case ‘n‘:
                    NextCh[]
                    if Wrong[’d c a s e’] do
                    §   Error[56, HARD, ‘e‘, Delete, CAT_v]
                        Send[ERROR, 0]
                        return
                    §
                    MSCRt[]
                    Send[ENDCASE, 0]
                    return

            default:Error[56, HARD, ‘e‘, Delete, CAT_v]
                    Send[ERROR, 0]
        §41
    §40


    and P_230[] be                                          || Ch is ‘f‘
    §50
        NextCh[]
        switchon Ch into
        §51
            case ‘a‘:
                    NextCh[]
                    if Wrong[’l s e’] do
                    §   Error[56, HARD, ‘f‘, Delete, CAT_v]
```

```
                    Send[ERROR, 0]
                    return
                §
130             Send[FALSE, 0]
                return

        case ‘o‘:
                NextCh[]
135             if Wrong[’r’] do
                §   Error[56, HARD, ‘f‘, Delete, CAT_v]
                    Send[ERROR, 0]
                    return
                §
140             MSCRt[]
                Send[FOR, 0]
                return

        default:Error[56, HARD, ‘f‘, Delete, CAT_v]
145             Send[ERROR, 0]
    §51
§50


150 and P_231[] be                                          || Ch is ‘g‘
§60
    NextCh[]
    switchon Ch into
    §61
155     case ‘e‘:
                NextCh[]
                if Wrong[’t’] do
                §   Error[56, HARD, ‘g‘, Delete, CAT_v]
                    Send[ERROR, 0]
160                 return
                §
                GetRt[]
                return

165     case ‘l‘:
                NextCh[]
                if Wrong[’o b a l’] do
                §   Error[56, HARD, ‘g‘, Delete, CAT_v]
                    Send[ERROR, 0]
170                 return
                §
                Send[GLOBAL, 0]
                return

175     case ‘o‘:
                NextCh[]
                if Wrong[’t o’] do
                §   Error[56, HARD, ‘g‘, Delete, CAT_v]
                    Send[ERROR, 0]
180                 return
                §
                MSCRt[]
                Send[GOTO, 0]
                return
185
        default:Error[56, HARD, ‘g‘, Delete, CAT_v]
                Send[ERROR, 0]
    §61
§60
190
```

```
    and P_233[] be                                          || Ch is ‘i‘
    §70
        NextCh[]
195     switchon Ch into
        §71
            case ‘f‘:
                    NextCh[]
                    switchon Ch into
200                 §711
                        case ‘n‘:
                                NextCh[]
                                if Wrong[’o t’] do
                                §   Error[56, HARD, ‘i‘, Delete, CAT_v]
205                                 Send[ERROR, 0]
                                    return
                                §
                                Send[IFNOT, 0]
                                return
210
                        case ‘s‘:
                                NextCh[]
                                if Wrong[’o’] do
                                §   Error[56, HARD, ‘i‘, Delete, CAT_v]
215                                 Send[ERROR, 0]
                                    return
                                §
                                Send[IFSO, 0]
                                return
220
                        default:MSCRt[]
                                Send[IF, 0]
                                return
                    §711
225                 return

            case ‘n‘:
                    NextCh[]
                    if Wrong[’t o’] do
230                 §   Error[56, HARD, ‘i‘, Delete, CAT_v]
                        Send[ERROR, 0]
                        return
                    §
                    Send[INTO, 0]
235                 return

            case ‘s‘:
                    NextCh[]
                    Send[IS, 0]
240                 return

            default:Error[56, HARD, ‘i‘, Delete, CAT_v]
                    Send[ERROR, 0]
        §71
245 §70


    and P_236[] be                                          || Ch is ‘l‘
    §80
250     NextCh[]
        switchon Ch into
        §81
            case ‘e‘:
                    NextCh[]
255                 if Wrong[’t’] do
                    §   Error[56, HARD, ‘l‘, Delete, CAT_v]
```

```
                    Send[ERROR, 0]
                    return
                §
260             Send[LET, 0]
                return

        case ‘o‘:
                NextCh[]
265             if Wrong[’o p’] do
                §   Error[56, HARD, ‘l‘, Delete, CAT_v]
                    Send[ERROR, 0]
                    return
                §
270             MSCRt[]
                Send[LOOP, 0]
                return

        case ‘s‘:
275             NextCh[]
                if Wrong[’h i f t’] do
                §   Error[56, HARD, ‘l‘, Delete, CAT_v]
                    Send[ERROR, 0]
                    return
280             §
                Send[LSHIFT, 0]
                return

        case ‘v‘:
285             NextCh[]
                Send[LV, 0]
                return

        default:Error[56, HARD, ‘l‘, Delete, CAT_v]
290             Send[ERROR, 0]
    §81
§80


295 and P_237[] be                                          || Ch is ‘m‘
§90
    NextCh[]
    if Wrong[’a n i f e s t’] do
    §   Error[56, HARD, ‘m‘, Delete, CAT_v]
300     Send[ERROR, 0]
        return
    §
    Send[MANIFEST, 0]
§90
305

and P_239[] be                                              || Ch is ‘o‘
§100
    NextCh[]
310 if Wrong[’r’] do
    §   Error[56, HARD, ‘o‘, Delete, CAT_v]
        Send[ERROR, 0]
        return
    §
315 Send[OR, 0]
§100


and P_242[] be                                              || Ch is ‘r‘
320 §110
    NextCh[]
```

```
        switchon Ch into
        §111
            case ‘e‘:
325                 NextCh[]
                    switchon Ch into
                    §1111
                        case ‘m‘:
                                NextCh[]
330                             Send[REM, 0]
                                return

                        case ‘p‘:
                                NextCh[]
335                             if Wrong[’e a t’] do
                                §   Error[56, HARD, ‘r‘, Delete, CAT_v]
                                    Send[ERROR, 0]
                                    return
                                §
340                             switchon Ch into
                                §11111
                                    case ‘u‘:
                                            NextCh[]
                                            if Wrong[’n t i l’] do
345                                         §   Error[56, HARD, ‘r‘, Delete,
                                                    CAT_v]
                                                Send[ERROR, 0]
                                                return
                                            §
350                                         Send[REPEATUNTIL, 0]
                                            return

                                    case ‘w‘:
                                            NextCh[]
355                                         if Wrong[’h i l e’] do
                                            §   Error[56, HARD, ‘r‘, Delete,
                                                    CAT_v]
                                                Send[ERROR, 0]
                                                return
360                                         §
                                            Send[REPEATWHILE, 0]
                                            return

                                    default:Send[REPEAT, 0]
365                                         return
                                §11111
                                return

                        case ‘s‘:
370                             NextCh[]
                                if Wrong[’u l t i s’] do
                                §   Error[56, HARD, ‘r‘, Delete, CAT_v]
                                    Send[ERROR, 0]
                                    return
375                             §
                                MSCRt[]
                                Send[RESULTIS, 0]
                                return

380                 case ‘t‘:
                                NextCh[]
                                if Wrong[’u r n’] do
                                §   Error[56, HARD, ‘r‘, Delete, CAT_v]
                                    Send[ERROR, 0]
385                                 return
                                §
```

```
                                MSCRt[]
                                Send[RETURN, 0]
                                return
390
                            default:Error[56, HARD, ‘r‘, Delete, CAT_v]
                                Send[ERROR, 0]
                        §1111
                        return
395
            case ‘s‘:
                        NextCh[]
                        if Wrong[’h i f t’] do
                        §   Error[56, HARD, ‘r‘, Delete, CAT_v]
400                         Send[ERROR, 0]
                            return
                        §
                        Send[RSHIFT, 0]
                        return
405
            case ‘v‘:
                        NextCh[]
                        Send[RV, 0]
                        return
410
            default:Error[56, HARD, ‘r‘, Delete, CAT_v]
                        Send[ERROR, 0]
        §111
    §110
415

    and P_243[] be                                              || Ch is ‘s‘
    §120
        NextCh[]
420     switchon Ch into
        §121
            case ‘t‘:
                        NextCh[]
                        if Wrong[’a t i c’] do
425                     §   Error[56, HARD, ‘s‘, Delete, CAT_v]
                            Send[ERROR, 0]
                            return
                        §
                        Send[STATIC, 0]
430                     return

            case ‘w‘:
                        NextCh[]
                        if Wrong[’i t c h o n’] do
435                     §   Error[56, HARD, ‘s‘, Delete, CAT_v]
                            Send[ERROR, 0]
                            return
                        §
                        MSCRt[]
440                     Send[SWITCHON, 0]
                        return

            default:Error[56, HARD, ‘s‘, Delete, CAT_v]
                        Send[ERROR, 0]
445     §121
    §120


    and P_244[] be                                              || Ch is ‘t‘
450 §130
        NextCh[]
```

```
        switchon Ch into
        §131
            case ‘a‘:
                    NextCh[]
                    if Wrong[’b l e’] do
                    §   Error[56, HARD, ‘t‘, Delete, CAT_v]
                        Send[ERROR, 0]
                        return
                    §
                    Send[TABLE, 0]
                    return

            case ‘e‘:
                    NextCh[]
                    if Wrong[’s t’] do
                    §   Error[56, HARD, ‘t‘, Delete, CAT_v]
                        Send[ERROR, 0]
                        return
                    §
                    MSCRt[]
                    Send[TEST, 0]
                    return

            case ‘h‘:
                    NextCh[]
                    if Wrong[’e n’] do
                    §   Error[56, HARD, ‘t‘, Delete, CAT_v]
                        Send[ERROR, 0]
                        return
                    §
                    Send[THEN, 0]
                    return

            case ‘o‘:
                    NextCh[]
                    Send[TO, 0]
                    return

            case ‘r‘:
                    NextCh[]
                    if Wrong[’u e’] do
                    §   Error[56, HARD, ‘t‘, Delete, CAT_v]
                        Send[ERROR, 0]
                        return
                    §
                    Send[TRUE, 0]
                    return

            default:Error[56, HARD, ‘t‘, Delete, CAT_v]
                    Send[ERROR, 0]
        §131
    §130

    and P_245[] be                                          || Ch is ‘u‘
    §140
        NextCh[]
        if Wrong[’n’] do
        §   Error[56, HARD, ‘u‘, Delete, CAT_v]
            Send[ERROR, 0]
            return
        §
        switchon Ch into
        §141
            case ‘l‘:
```

```
                    NextCh[]
                    if Wrong['e s s'] do
                    §   Error[56, HARD, 'u', Delete, CAT_v]
520                     Send[ERROR, 0]
                        return
                    §
                    MSCRt[]
                    Send[UNLESS, 0]
525                 return

            case 't':
                    NextCh[]
                    if Wrong['i l'] do
530                 §   Error[56, HARD, 'u', Delete, CAT_v]
                        Send[ERROR, 0]
                        return
                    §
                    MSCRt[]
535                 Send[UNTIL, 0]
                    return

            default:Error[56, HARD, 'u', Delete, CAT_v]
                    Send[ERROR, 0]
540     §141
    §140


    and P_246[] be                                          || Ch is 'v'
545 §150
        NextCh[]
        switchon Ch into
        §151
            case 'a':
550                 NextCh[]
                    if Wrong['l o f'] do
                    §   Error[56, HARD, 'v', Delete, CAT_v]
                        Send[ERROR, 0]
                        return
555                 §
                    Send[VALOF, 0]
                    return

            case 'e':
560                 NextCh[]
                    if Wrong['c'] do
                    §   Error[56, HARD, 'v', Delete, CAT_v]
                        Send[ERROR, 0]
                        return
565                 §
                    Send[VEC, 0]
                    return

            default:Error[56, HARD, 'v', Delete, CAT_v]
570                 Send[ERROR, 0]
        §151
    §150


575 and P_247[] be                                          || Ch is 'w'
    §160
        NextCh[]
        if Wrong['h i l e'] do
        §   Error[56, HARD, 'w', Delete, CAT_v]
580         Send[ERROR, 0]
            return
```

```
        $
        MSCRt[]
        Send[WHILE, 0]
585 $160
****
```

```
   let NormalSend[x, y] be
   §NS
       Symb := x
 5     Out[Output, x ∨ y]
   $NS


   and SendNL[x, y] be
10 §SN
       if x = GET do
       §   SaveLN := 1
           return
       $
15     if x = NEWLINE do
       §   SaveLN := y
           return
       $
       Send := NormalSend
20     if CanStartCom[x] do Send[SEMICOLON, 0]
       Send[NEWLINE, OldLN]
       unless GetNo = OldGet do
       §   Send[GET, OldGet]
           Send[GET, GetNo]
25     $
       Send[NEWLINE, SaveLN]
       Send[x, y]
   $SN

30
   and NLRt[] be
   §N
       OldLN := LineNo
       §   LineNo := LineNo + 1
35         NextCh[]
       $ repeatwhile Ch = '*n'
       if LineNo > 2000 do Error[3, HARD, DUMMY, TermPP]
       test CanEndCom[Symb]
         ifso  Send, SaveLN, Symb, OldGet := SendNL, LineNo, 0, GetNo
40               || Symb must not be able to end an exp or com
         ifnot §   Send[NEWLINE, OldLN]
                   Send[NEWLINE, LineNo]
   $N

45
   let GetRt[] be
   §G
       let i, g = 0, GetList↓0
       let v, End = GetList + g × GETSIZE, GetList + GETLISTSIZE
50     OldLN := LineNo
       if g ≥ GETMAX do Error[65, HARD, DUMMY, TermPP]
       §R
           Delete[CAT_b]
           if Ch = '|' do
55         §   NextCh[]
               if Wrong['|'] do
               §   Error[63, HARD, DUMMY, Delete, CAT_c]
                   return
               $
60             Delete[CAT_c]
           $
```

```
            if Ch = '*n' break
            unless Is[Ch, CAT_s] do
            §   Error[66, VERYHARD, DUMMY, Delete, CAT_c]
65              return
            §
            if v + i ≥ End do
            §   Error[67, VERYHARD, DUMMY, Delete, CAT_c]
                return
70          §
            i := i + 1
            v↓i := TreatString[Ch = '[' → ']', Ch]
        §R repeat
        if i = 0 do
75      §   Error[68, VERYHARD, DUMMY, NullProgram]
            return
        §
        unless IsRoomforInputStream[] do
        §   Error[69, VERYHARD, DUMMY, NullProgram]
80          return
        §
        test i = 1
          ifso  i, v↓2 := 2, 'Text'
          ifnot if (i ∧ 1) ≠ 0 do
85              §   i := i + 1
                    v↓i := 'Index'
                §
      § let File = FindFile[v, i]
        if File = 0 return
90      v↓FILENO, v↓SAVEIN, v↓SAVELINENO, v↓GETPRE :=
            File, In, LineNo, GetNo
        In := BytesfromWords[InfromFile[File]]
        if Endof[In] do
        §   Error[70, SOFT, DUMMY, NullProgram]
95          Close[In]
            In := v↓SAVEIN
            Ch := '*n'
            return
        §
100     Send[NEWLINE, OldLN]
        Send[GET, GetNo]
        GetList↓0 := g + 1
        GetNo := g + 1
        ProcVec↓ENDCH := DenestGets
105     NextCh[] repeatwhile Ch = '*n'
        LineNo := 1
        Send[GET, GetNo]
        Send[NEWLINE, 1]
    §G
110


    and FindFile[v, i] = valof
    §FF
        let Index = -1
115     while i > 2 do
        §   Index := SpecialLookUp[v↓(i - 1), v↓i, Index, true]
            i := i - 2
            if Index = 0 do resultis 0
        §
120   § let f = SpecialLookUp[v↓1, v↓2, Index, false]
        if f = 0 do resultis 0
      § let h = FindHeadingVector[f]
        if Deleted[h] do
        §   Error[76, VERYHARD, DUMMY, NullProgram]
125         f := 0
        §
```

```
        if Archived[h] do
        §   Error[77, VERYHARD, DUMMY, NullProgram]
            f := 0
130     §
        ReturnHeadVec[h]
        resultis f
    $FF

135
    and SpecialLookUp[S1, S2, i, IndReq] = valof
    §SLU
        let f = 0
        test i = -1
140       ifso  §   f := LookUp[S1, S2, CurrentIndex]
                    if f = 0 do
                        f := LookUp[S1, S2, IndReq → SystemIndex, DeclIndex]
              §
          ifnot f := LookUp[S1, S2, i]
145     if f = 0 do Error[IndReq → 71, 72, VERYHARD, DUMMY, NullProgram]
        resultis f
    $SLU


150 and DenestGets[] be
    §   let g = GetList + (GetNo - 1) × GETSIZE
        Close[In]
        Send[NEWLINE, LineNo]
        Send[GET, GetNo]
155     In, LineNo := g↓SAVEIN, g↓SAVELINENO
        GetNo := g↓GETPRE
        if GetNo = 0 do ProcVec↓ENDCH := P_End_of_Input
        Ch := '*n'                  || so next char sent gives correct line no
        Send[GET, GetNo]
160 §


    and CharacterRt[] be
    §CR
165     let RememberLN, SaveStack = LineNo, Stack
        let s = TreatString[Ch]
        if s = 0 return
        unless (s↓0 rshift 8) ≤ 1 do
        §   Error[73, HARD, DUMMY, NullProgram]
170         Send[ERROR, 0]
            return
        §
        Send[CHARACTER, s↓0 ∧ 8377]
        unless RememberLN = LineNo ∨ Ch = '*n' do Write[NEWLINE ∨ LineNo]
175                                     || Send without setting Symb
        Stack := SaveStack
    §CR


180 and StringRt[] be
    §SR
        let RememberLN = LineNo
        let s = TreatString[Ch = '[' → ']', Ch]
        if s = 0 do
185     §   Send[ERROR, 0]
            return
        §
        Push[s]
        Send[STRING, Adjoin_to[StringTable]]
190     unless RememberLN = LineNo ∨ Ch = '*n' do Write[NEWLINE ∨ LineNo]
                                        || Send without setting Symb
```

```
    $SR


195 and TreatString[EndQ] = valof
    §TS
        NextCh[]
        switchon Ch into
        §SW
200         case `**`:
                    Push[StarFn[Next[In]]]
                    endcase

            case `*n`:
205                 LineNo := LineNo + 1
                    NextCh[] repeatwhile Is[Ch, CAT_b]
                    endcase

            case ENDCH:
210                 if Endof[In] do
                    §   Error[59, VERYHARD, DUMMY, NullProgram]
                        ResetStack[]
                        resultis 0
                    §
215
            default:if Ch = EndQ break
                    PushCh_NextCh[]
        $SW repeat
        if TooBig[] do Error[58, VERYHARD, DUMMY, NullProgram]
220     NextCh[]
        resultis PackedString[]
    $TS


225 and StarFn[x] = valof
    §SF
        test x = `8`
          ifso  §8
                    let n = 0
230                 NextCh[] repeatwhile Is[Ch, CAT_b]
                    unless Is[Ch, CAT_o] do
                    §   Error[74, HARD, DUMMY, NullProgram]
                        resultis Ch
                    §
235                 for i = 1 to 3 do
                    §   n := (n lshift 3) + Ch - `0`
                        NextCh[]
                        unless Is[Ch, CAT_o] break
                    §
240                 resultis n
                $8
          ifnot §   NextCh[]
                    switchon x into
                    §S
245                     case `4`:
                                resultis `*4`

                        case `b`:
                                resultis `*b`
250
                        case `n`:
                                resultis `*n`

                        case `p`:
255                             resultis `*p`
```

```
                        case ‘q‘:
                                resultis ‘¿

260                     case ‘s‘:
                                resultis ‘*s‘

                        case ‘**‘:
                        case ‘*‘‘:
265                     case ‘*’‘:
                                resultis x

                        default:Error[57, HARD, x, NullProgram]
                                resultis x
270 $SF


    and PackedString[] = valof
    §PS
275     let s = Stack
        if TooBig[] do StackPtr := MAXLENGTH
        PackStack[]
        StepOnStack[]
        resultis s
280 $PS


    and NumFn[Base, Cat] = valof
    §NF
285     let n = 0
        while Is[Ch, Cat] do
        §   n := n × Base + Ch - ‘0‘
            NextCh[]
        §
290     if 0 ≤ n ≤ 500 do
        §   Send[NUMBER, n]
            resultis false
        §
        Push[n]
295     resultis true
    $NF


    let SendTables[] be
300 §ST
        SendT[’Numbers’, NumTable, Write]
        SendT[’Strings’, StringTable, SendS]
        SendGets[]
        SendIds[]
305 $ST


    and SendT[WF, Table, Rt] be
    §ST
310     Output := OuttoFile[WorkFile[WF]]
        Write[Table↓PTR - OFFSET]
        for i = FIRSTEL to Table↓PTR do Rt[Table↓i]
        Close[Output]
    $ST
315

    and SendS[s] be TransferOut[Output, s, (s↓0 rshift 9) + 1]


320 and SendGets[] be
    §SG
```

```
        Output := OuttoFile[WorkFile['Gets']]
        Write[GetList↓0]
        for i = 0 to GetList↓0 - 1 do
325     §   Write[GetList↓(i × GETSIZE + FILENO)]
            SendS[GetList↓(i × GETSIZE + 1)]
        §
        Close[Output]
    $SG
330

    and SendIds[] be
    §SI
        Output := OuttoFile[WorkFile['Names']]
335     Write[NameTable↓PTR - OFFSET + 26]
        for i = 0 to 25 do Write[(8400 ∨ 'a') + i]
        for i = FIRSTEL to NameTable↓PTR do SendS[NameTable↓i]
        Close[Output]
    $SI
340
****
```

```
PP
 *** Compilers 31/7/76


    get 'GLOBALS'
    get 'PRIVATE GLOBALS'
    get 'LIBRARY GLOBALS'
  5 get 'BitMap globals'

    get 'Manifests 0'

    get 'Common Decls'
 10 get 'Decls 0'

    get 'Heading declarations'

    get 'PP 1'
 15 get 'PP 2'
    get 'PP 3'
    get 'PP 4'


****
```

```
Print Reports 1
 *** Compilers 20/12/75


    static Room = 0


  5 let PrintReports[] be
    §PR
        SetupTables[]
        DoReports[WorkFile['Error Reports']]
        unless (Mode ∧ 2) = 0 do PrintGlobList[]
 10     ClosedownTables[]
    $PR


    and SetupTables[N] be
 15 §ST
        Output := ReportStream
        NameTable := SetUpTable['Names', 0]
        StringTable := SetUpTable['Strings', 0]
        GetTable := SetUpTable['Gets', 1]
 20     NumberTable := VecfromFile[WorkFile['Numbers']] + 1
    $ST


    and SetUpTable[WF, a] = valof
 25 §ST
        let v = VecfromFile[WorkFile[WF]] + 1
        let Ptr = v + 1
        and n = v↓0
        let Tab = ProperVec[n]
 30     for i = 1 to n do
        §   Tab↓i := Ptr
            Ptr := NextString[Ptr + a]
        §
        resultis Tab
 35 $ST


    and ClosedownTables[] be
    §CT
 40     ReturnTable[NumberTable]
        ReturnTable[StringTable]
        ReturnTable[GetTable]
        ReturnTable[NameTable]
    $CT
 45


    and DoReports[E] be
    §DR
        SetupVecs[E]
 50     OutputReports[]
    $DR


    and SetupVecs[E] be
 55 §SV
        let v = VecfromFile[E]
        NoReports := v↓0 / 5
        ErrorVec := ProperVec[NoReports]
        LineVec := ProperVec[ReportSize]
 60     for i = 1 to NoReports do ErrorVec↓i := v + (5 × i) - 4
        Quicksort[ErrorVec, 1, NoReports, Gtr]
```

```
   §SV


65 and Gtr[p, q] = p↓1 ≠ q↓1 → p↓1 > q↓1, p↓2 ≠ q↓2 → p↓2 > q↓2, p > q


   and OutputReports[] be
   §OR
70     let ReportsPrinted = 0
       if NoReports = 0 return
       WriteS['*n*n*N REPORT*C*n', NoReports, NoReports = 1 → ' ', 'S']
       §R
           let i = ReportsPrinted + 1
75         let G = GetVal[i]
           test G = 0
             ifso  WriteS['*nIn Main Text:*n']
             ifnot WriteS['*nIn get file *'*S*':*n', FileTitle[G]]
           SetupGIn[G]
80         while i ≤ NoReports ∧ GetVal[i] = G do
           §   unless AlreadyOutput[ErrorVal[i]] do PrintError[i]
               i := i + 1
           §
           WriteS['*n*n']
85         i := ReportsPrinted + 1
           while i ≤ NoReports ∧ GetVal[i] = G do
           §   OutputLine[LineVal[i]]
               i := i + 1
           §
90         ReportsPrinted := i - 1
           Close[GIn]
       §R repeatwhile ReportsPrinted < NoReports
   §OR

95
   and PrintError[i] be
   §MES
       let p, q = 1, 1
       and G, E, C, L = GetVal[i], ErrorVal[i], ChVal[i], LineVal[i]
100    and MultLine = false
       LineVec↓1 := L
       for j = i + 1 to NoReports do
           if GetVal[j] = G ∧ ErrorVal[j] = E ∧ ChVal[j] = C do
           §   p := p + 1
105            LineVec↓p := LineVal[j]
               UpdateErrorVal[j, 0]
               unless LineVal[j] = L do MultLine := true
           §
       LineVec↓(p + 1) := 0
110    ErrorString[i, p]
       if L = 0 return
       WriteS['on line']
       if MultLine do Write['s']
       while q ≤ p do
115    §   let l = LineVec↓q
           and r = 0
           while LineVec↓q = l do q, r := q + 1, r + 1
           if Room < (r = 1 → 5, 15) do
           §   WriteS['*n*4*4']
120            Room := 64
           §
           WriteS[' *N', l]
           Nce[r]
           if q ≤ p do Write[',']
125        Room := Room - 5
       §
```

```
    $MES


130 and Nce[r] be
    §N
        switchon r into
        §S
            case 1: return
135
            case 2: WriteS[' (twice)']
                    Room := Room - 8
                    return

140         default:WriteS[' (*N times)', r]
                    Room := Room - 10
        §S
    §N

145
    and SetupGIn[G] be
    §SI
        let Ch = 0
        GIn := G = 0 → In, BytesfromWords[InfromFile[FileNo[G]]]
150     Ch := Next[GIn] repeatwhile Ch = '*n'
        PutBack[GIn, Ch]
        LineNo := 1
    §SI

155
    and OutputLine[L] be
    §OL
        let Ch = 0
        until LineNo ≥ L do
160     §   Ch := Next[GIn]
            if Ch = '*n' do LineNo := LineNo + 1
            if Endof[GIn] do
            §   WriteS['End of text at line *N', LineNo]
                LineNo := ENORMOUS
165         §
        §
        if LineNo > L return
        WriteS['*I4. ', LineNo]
        §   Ch := Next[GIn]
170         Write[Ch]
        § repeatuntil Ch = '*n' ∨ Endof[GIn]
        LineNo := LineNo + 1
    §OL

175
    and VecfromFile[f] = valof
    §V
        let h = FindHeading[f]
        if Empty[h] do
180     §   ReturnHead[h]
            resultis ProperVec[0]
        §
        DisctoCore[DiscPage, LastPage[h]]
        §   let Unused = ENDOFDATAAREA - LastUsedWordPtr[DiscPage]
185         let n = (DATASIZE × NumberofPages[h]) - Unused
            ReturnHead[h]
          § let v = ProperVec[n]
            let S = InfromFile[f]
            TransferIn[S, v + 1, n]
190         Close[S]
            resultis v
```

```
    $V


195 and ProperVec[n] = valof
    §P
        let m = MaxVecSize[]
        if m < n do
        §   ReportS['Not enough free store for reports']
200         Finish[]
        $
        §   let v = NewVec[n]
            v↓0 := n
            resultis v
205 $P


    and PrintGlobList[] be
    §P
210     let n = GlobList↓0
        let Sys = GlobList↓(-1)
        let m = n > GlobSize → GlobSize, n
        ReportS['*n*N global*S set', n, AddS[n]]
        unless Sys = 0 do
215         ReportS['*4including *N system global*S', Sys, AddS[Sys]]
        for i = 1 to 2 × m by 2 do
        §   let v = GlobList↓i     || the sign bit is set for system globals
            ReportS['*I3: *S*S', (v ∧ 877777), NameTable↓(GlobList↓(i + 1)),
                 v < 0 → ' ************', '']
220     $
        if m < n do ReportS['*n  etc.']
    $P


225 and AddS[n] = n = 1 → '', 's'
****
```

```
   manifest
   §m
       Write2[Ch, S] is
 5     §   OutputSymbol[Ch]
           WriteS[S]
       §

       Write3[S1, Ch, S2] is
10     §   WriteS[S1]
           OutputSymbol[Ch]
           WriteS[S2]
       §

15     Write4[Ch1, Ch2] is
       §   OutputSymbol[Ch1]
           WriteS[' found where ']
           OutputSymbol[Ch2]
           WriteS[' expected']
20     §

       WriteTM[S] is
           WriteS['Too many *S', S]

25     WriteSR[S] is
           WriteS['Soft report: *S', S]

       WriteGF[S] is
           WriteS['get file *S', S]
30
       WriteSubs[n, S] is
       §   WriteS[n = 1 → '*n*4*4second *S ', '*n*4*4subsequent *Ss ', S]
           Room := (n = 1 → 44, 39)
       §
35
       Times[n] is
           WriteS[n = 2 → 'twice', '*N times', n]
   $m

40 let ErrorString[i, n] be
   §ES
       let Ch = ChVal[i]
       WriteS['*n*4']
       switchon ErrorVal[i] into
45     §S
           case 1: WriteS['Too many pass *N error reports', PassVal[i]]
                   endcase

           case 2: WriteS['Structure Size exceeded']
50                 endcase

           case 3: WriteTM['newlines in text']
                   endcase

55         case 5: WriteS['End of text found but not expected']
                   endcase

           case 6: WriteS['End of text expected but not found']
                   endcase
60
           case 7: WriteS['Too much code']
```

```
            endcase

        case 8: WriteS['Subsidiary vector overflow']
            endcase

        case 9: WriteS['Not enough free store']
            endcase

        case 10:WriteS['Too much data']
            endcase

        case 11:WriteS['Too many pass *N reports - soft reports
                *signored, starting', PassVal[i]]
            endcase

        case 20:WriteSR['too many globals set for list']
            endcase

        case 21:WriteSR['item with ']
            Write2[Ch, ' sets system global']
            endcase

        case 22:WriteS['global *N set ', Ch]
            Times[n + 1]
            return

        case 23:Write3['global with ', Ch, ' set ']
            Times[n + 1]
            WriteSubs[n, 'time']
            return

        case 51:WriteTM['names']
            endcase

        case 52:WriteTM['numbers']
            endcase

        case 53:WriteTM['strings']
            endcase

        case 55:WriteS['Invalid character *'*C*', Ch]
            endcase

        case 56:WriteS['Invalid symbol, starting *'*C*',', Ch]
            endcase

        case 57:WriteS['Invalid symbol *'***C*' in string', Ch]
            endcase

        case 58:WriteS['String too long']
            endcase

        case 59:WriteS['End of text in string']
            endcase

        case 60:WriteTM['characters in names']
            endcase

        case 61:WriteS['Name too long']
            endcase

        case 62:WriteS['Tag too long']
            endcase

        case 63:WriteS['Unpaired comment bar']
```

```
                endcase

            case 64:WriteS['Invalid character *'*C*' after 8', Ch]
130             endcase

            case 65:WriteTM['gets']
                endcase

135         case 66:WriteS['Invalid get param']
                endcase

            case 67:WriteTM['get params']
                endcase
140
            case 68:WriteS['No get params']
                endcase

            case 69:WriteTM['nested gets']
145             endcase

            case 70:WriteSR['get file empty']
                endcase

150         case 71:WriteS['get index not found']
                endcase

            case 72:WriteGF['not found']
                endcase
155
            case 73:WriteS['String in character quotes']
                endcase

            case 74:WriteS['Invalid char after **8']
160             endcase

            case 75:WriteS['StringVec full']
                endcase

165         case 76:WriteGF['deleted']
                endcase

            case 77:WriteGF['archived']
                endcase
170
            case 78:WriteSR['star used for multiplication']
                endcase

            case 101:
175             Write2[Ch, ' found out of context in expression']
                endcase

            case 102:
                Write2[Ch, ' not expected to delimit expression']
180             endcase

            case 103:
                Write2[Ch, ' not expected to start command']
                endcase
185
            case 104:
                Write2[Ch, ' not expected to delimit command']
                endcase

190         case 105:
                Write2[Ch, ' found out of context']
```

```
                endcase

            case 106:
195             Write2[Ch, ' out of context in manifest expression']
                endcase

            case 107:
                WriteSR['closing section bracket *']
200             Write2[Ch, '*' inserted']
                endcase

            case 111:
                Write4[Ch, SECTBRA]
205             endcase

            case 112:
                Write4[Ch, ASS]
                endcase
210
            case 113:
                Write4[Ch, COLON]
                endcase

215         case 114:
                Write4[Ch, COMMA]
                endcase

            case 115:
220             Write4[Ch, RKET]
                endcase

            case 116:
                Write4[Ch, SKET]
225             endcase

            case 118:
                Write4[Ch, SBRA]
                endcase
230
            case 122:
                Write4[Ch, EQ]
                endcase

235         case 130:
                Write4[Ch, DO]
                endcase

            case 131:
240             Write4[Ch, INTO]
                endcase

            case 132:
                Write4[Ch, TO]
245             endcase

            case 142:
                Write2[Ch, ' found where ; or § expected']
                endcase
250
            case 144:
                Write4[Ch, THEN]
                endcase

255         case 145:
                Write4[Ch, OR]
```

```
                endcase

            case 181:
260             WriteS['Invalid L-mode expression']
                endcase

            case 191:
                Write2[Ch, ' found where operator expected']
265             endcase

            case 192:
                Write2[Ch, ' found where operand expected']
                endcase
270
            case 201:
                Write2[Ch, ' found where name expected']
                endcase

275         case 202:
                Write2[Ch, ' declared more than once on the same level']
                Nce[n]
                WriteSubs[n, 'time']
                return
280
            case 203:
                Write2[Ch, ' used but not declared']
                endcase

285         case 204:
                Write3['manifest ', Ch, ' used in l-mode']
                endcase

            case 205:
290             WriteS['Invalid constant expression']
                endcase

            case 206:
                WriteSR['scope clash for items with ']
295             OutputSymbol[Ch]
                Nce[n]
                WriteSubs[n, 'defn']
                return

300         case 251:
                WriteS['case not in switch']
                endcase

            case 252:
305             WriteS['endcase not in switch']
                endcase

            case 253:
                WriteS['break not in loop']
310             endcase

            case 254:
                WriteS['resultis not in valof block']
                endcase
315
            case 255:
                WriteS['default not in switch']
                endcase

320         case 256:
                WriteS['*N identical case constants in switch', n + 1]
```

```
                WriteSubs[n, 'time']
                return

325         case 257:
                WriteS['loop not in loop']
                endcase

            case 258:
330             WriteS['*N defaults found in switch', n + 1]
                WriteSubs[n, 'time']
                return

            case 259:
335             WriteS['return used in function']
                endcase

            case 260:
                WriteS['No resultis in valof block*C ',
340                 n = 1 → DUMMY, `s`]
                Room := 21
                return

            case 261:
345             WriteS['Global number *N out of range', Ch]
                endcase

            case 262:
                WriteS['vec parameter negative']
350             endcase

            case 270:
                WriteS['Manifest function used as routine']
                endcase
355
            case 271:
                WriteS['Manifest routine used as function']
                endcase

360         case 301:
                WriteS['Unequal sides in local declaration']
                endcase

            case 302:
365             WriteS['Unequal sides in assignment command']
                endcase

            case 303:
                WriteS['More than one name in vec declaration']
370             endcase

            case 350:
                WriteSR['manifest fn called with']
                WriteS[' *N too *S params', Ch > 0 → Ch, -Ch,
375                 Ch > 0 → 'many', 'few']
                endcase

            case 351:
                WriteSR['manifest definition buffer full']
380             endcase

            case 352:
                WriteSR['manifest procedure with ']
                Write2[Ch, '*n*4*4too big for open code']
385             endcase
```

```
            case 353:
                    WriteSR['manifest procedure with ']
                    Write2[Ch, '*n*4*4used (not applied)']
390                 endcase

            default:WriteS['Unknown report *N ', ErrorVal[i]]
                    OutputSymbol[Ch]
                    Write[`,`]
395                 endcase
        $S
        Nce[n]
        WriteS[n = 1 → ' ', '*n*4*4']
        Room := 56
400 $ES


****
```

```
   let OutputSymbol[C] be
   §OS
       switchon TypePart[C] into
 5     §s
           case NAME:
                   OutputName[ValPart[C]]
                   endcase

10         case NUMBER:
                   OutputNumber[ValPart[C]]
                   endcase

           case STRING:
15                 OutputString[ValPart[C]]
                   endcase

           case CHARACTER:
                   OutputCharacter[ValPart[C]]
20                 endcase

           case SECTBRA:
                   OutputSectBra[ValPart[C]]
                   endcase
25
           case SECTKET:
                   OutputSectKet[ValPart[C]]
                   endcase

30         case ASS:
                   WriteS[':=']
                   endcase

           case COLON:
35                 Write[':']
                   endcase

           case COMMA:
                   Write[',']
40                 endcase

           case COND:
                   Write['→']
                   endcase
45
           case DIV:
                   Write['/']
                   endcase

50         case EQ:Write['=']
                   endcase

           case EQV:
                   Write['=']
55                 endcase

           case GE:Write['≥']
                   endcase

60         case GT:Write['>']
                   endcase
```

```
        case LE:Write[‘≤‘]
             endcase
65
        case LOGAND:
             Write[‘∧‘]
             endcase

70      case LOGOR:
             Write[‘∨‘]
             endcase

        case LT:Write[‘<‘]
75           endcase

        case MINUS:
             Write[‘-‘]
             endcase
80
        case MULT:
             Write[‘×‘]
             endcase

85      case NE:Write[‘≠‘]
             endcase

        case NEQV:
             Write[‘≠‘]
90           endcase

        case NOT:
             Write[‘ ‘]
             endcase
95
        case PLUS:
             Write[‘+‘]
             endcase

100     case RBRA:
             Write[‘(‘]
             endcase

        case RKET:
105          Write[‘)‘]
             endcase

        case SBRA:
             Write[‘[‘]
110          endcase

        case SEMICOLON:
             Write[‘;‘]
             endcase
115
        case SKET:
             Write[‘]‘]
             endcase

120     case VECAP:
             WriteS[’**[’]
             endcase

        case VECOP:
125          Write[‘↓‘]
             endcase
```

```
        case AND:
                WriteS['and']
130             endcase

        case BE:WriteS['be']
                endcase

135     case BY:WriteS['by']
                endcase

        case BREAK:
                WriteS['break']
140             endcase

        case CASE:
                WriteS['case']
                endcase
145
        case DEFAULT:
                WriteS['default']
                endcase

150     case DO:WriteS['do']
                endcase

        case ENDCASE:
                WriteS['endcase']
155             endcase

        case FALSE:
                WriteS['false']
                endcase
160
        case FOR:
                WriteS['for']
                endcase

165     case GLOBAL:
                WriteS['global']
                endcase

        case GOTO:
170             WriteS['goto']
                endcase

        case IF:WriteS['if']
                endcase
175
        case IFNOT:
                WriteS['ifnot']
                endcase

180     case IFSO:
                WriteS['ifso']
                endcase

        case INTO:
185             WriteS['into']
                endcase

        case IS:WriteS['is']
                endcase
190
        case LET:
```

```
                WriteS['let']
                endcase

195         case LOOP:
                WriteS['loop']
                endcase

            case LSHIFT:
200             WriteS['lshift']
                endcase

            case LV:WriteS['lv']
                endcase
205
            case MANIFEST:
                WriteS['manifest']
                endcase

210         case OR:WriteS['or']
                endcase

            case REM:
                WriteS['rem']
215             endcase

            case REPEAT:
                WriteS['repeat']
                endcase
220
            case REPEATUNTIL:
                WriteS['repeatuntil']
                endcase

225         case REPEATWHILE:
                WriteS['repeatwhile']
                endcase

            case RESULTIS:
230             WriteS['resultis']
                endcase

            case RETURN:
                WriteS['return']
235             endcase

            case RSHIFT:
                WriteS['rshift']
                endcase
240
            case RV:WriteS['rv']
                endcase

            case STATIC:
245             WriteS['static']
                endcase

            case SWITCHON:
                WriteS['switchon']
250             endcase

            case TABLE:
                WriteS['table']
                endcase
255
            case TEST:
```

```
                    WriteS['test']
                    endcase

        case TO:WriteS['to']
                    endcase

        case TRUE:
                    WriteS['true']
                    endcase

        case UNLESS:
                    WriteS['unless']
                    endcase

        case UNTIL:
                    WriteS['until']
                    endcase

        case VALOF:
                    WriteS['valof']
                    endcase

        case VEC:
                    WriteS['vec']
                    endcase

        case WHILE:
                    WriteS['while']
                    endcase

        case ENDBODY:
                    WriteS['End of procedure declaration']
                    endcase

        case ENDPROG:
                    WriteS['End of text']
                    endcase

        default:WriteS['symbol *O', C]
                    endcase
    $s
$OS


and OutputName[n] be WriteS['Name *'*S*'', NameTable↓n]


and OutputNumber[n] be
    WriteS['Number *N', n ≤ 500 → n, NumberTable↓(n - 500)]


and OutputString[n] be WriteS['*'*S*'', StringTable↓n]


and OutputCharacter[n] be WriteS['*`*C*`', n]


and OutputSectBra[n] be WriteS['§*S', n = 0 → '', NameTable↓n]


and OutputSectKet[n] be WriteS['$*S', n = 0 → '', NameTable↓n]

****
```

```
Print Reports
 *** Compilers 31/7/76


    get 'GLOBALS'
    get 'PRIVATE GLOBALS'
    get 'LIBRARY GLOBALS'
  5
    get 'Disc buffer declarations'
    get 'Heading declarations'

    get 'Manifests 0'
 10 get 'Manifests 1'

    get 'Common Decls'
    get 'Reports Decls'

 15 get 'Print Reports 1'
    get 'Print Reports 2'
    get 'Print Reports 3'


****
```

```
Declarations for Print Reports
 *** Compilers 31/7/76


    manifest GZ = 430

    global
  5 §g
        NumberTable : GZ
        StringTable : GZ + 1
        GetTable    : GZ + 2
        NameTable   : GZ + 3
 10     LineVec     : GZ + 4
        ErrorVec    : GZ + 5
        NoReports   : GZ + 6
        GIn         : GZ + 7
        ErrorString : GZ + 8
 15     OutputSymbol: GZ + 9
    $g

    manifest
    §m
 20     PassVal[i] = ErrorVec↓i↓0
        GetVal[i] = ErrorVec↓i↓1
        LineVal[i] = ErrorVec↓i↓2
        ErrorVal[i] = ErrorVec↓i↓3
        ChVal[i] = ErrorVec↓i↓4
 25
        UpdateErrorVal[i, e] is ErrorVec↓i↓3 := e

        AlreadyOutput[e] = e=0

 30     FileTitle[G] = GetTable↓G + 1
        FileNo[G] = GetTable↓G↓0

        NextString[p] = p + (rv p rshift 8)/2 + 1

 35     ReturnTable[Tab] is ReturnVec[Tab, Tab↓0]
    $m


****
```

Stack Machine Manifests
*** Compilers 3/7/76

```
   manifest
   §m
       SETLITGLOBAL  = 84001
 5     SETLITSTATIC  = 83002
       SETCSEGGLOBAL = 84003
       SETCSEGSTATIC = 83004
       LOADSTATIC    = 83005
       LOADSTRING    = 82006
10     FORHOP        = 82007
       BACKHOP       = 82010
       CONDFORHOP    = 83011
       FORHOPPT      = 82013
       BACKHOPPT     = 82014
15     SWITCH        = 80015
       LOADMANFUN    = 83016
       ENDMFDEF      = 82017
       SETMANFUN     = 83020
       SETDSEGGLOBAL = 84021
20     SETDSEGSTATIC = 83022
       ENDPARAM      = 8102023
       COERCETOVAL   = 8113024

       INCsm     = 8102000
25     Nsm       = 8112001
       Ssm       = 8102002
       RPsm      = 8112003
       SPsm      = 8072004
       APPLYsm   = 8102005
30     JUMPPTsm  = 8102006
       HOPsm     = 8102007
       SAsm      = 8072010
       LPsm      = 8112011
       RAsm      = 8112012
35     LGsm      = 8112015
       SGsm      = 8072016
       RGsm      = 8112017

       HOPIFZEsm = 8070300
40     HOPIFGEsm = 8060302
       HOPIFNEsm = 8060304
       HOPIFGTsm = 8060306
       HOPIFLEsm = 8060310
       HOPIFEQsm = 8060312
45     HOPIFLTsm = 8060314
       HOPIFNZsm = 8070316

       NOTsm     = 8101360
       NEGsm     = 8101361
50     MULTsm    = 8071362
       DIVsm     = 8071363
       REMsm     = 8071364
       ADDsm     = 8071365
       SUBsm     = 8071366
55     LSHsm     = 8071367
       RSHsm     = 8071370
       ANDsm     = 8071371
       ORsm      = 8071372
       EQVsm     = 8071373
60     NEQVsm    = 8071374
       CONTSsm   = 8101375
```

```
        STOREsm   = 8061376
        GOTOsm    = 8071377

 65     SWAPsm    = 8101340
        DUPsm     = 8111341
        DDUPsm    = 8121342
        LINKsm    = 8111343
        FNEXITsm  = 8072344
 70     RTEXITsm  = 8102345
        SWITCHsm  = 8072346
        TRUEsm    = 8111347
        VECAPsm   = 8071350
        VECSTsm   = 8051351
 75
        RETURNsm  = 8102357

        CODE        = 1
        DATA        = 4
 80     DIAGNOSTICS = 5
        NEWSECTION  = 6
        ENDLOAD     = 7
        LABELS      = 15
        GLOBALS     = 16
 85
        CSEG        = 0
        DSEG        = 8100000
        LIT         = -1

 90     CSEGg       = 1
        DSEG        = 2
        ACTUALg     = 3

        LONG        = 8240
 95     MEDIUM      = 8260

        STACKMACHINECODE = 3

        StackIncrement[T] = ((T rshift 12) - 8)
100     LengthField[T] = ((T rshift 9) ∧ 7)
        InstField[T] = (T ∧ 8777)
    $m
****
```

```
Stream Compiler Steering
 *** Compilers 12/5/78

    get 'GLOBALS'
    get 'PRIVATE GLOBALS'
    get 'LIBRARY GLOBALS'

  5 get 'Common Decls'


    static
    §s
 10     WF            = 0
        InputStream   = 0
        OutputStream  = 0
        Compiled      = false
        WorstError    = NOERROR
 15 $s

    static  || initialising globals
    §s
        Mode          = 6
 20     NameSize      = 1000
        NumberSize    = 500
        TextSize      = 7500
        StringSize    = 200
        StructureSize = 12000
 25     AppSize       = 300
        CodeSize      = 6000
        ReportSize    = 30
        GlobSize      = 50
    $s
 30
    manifest
    §m
        AddtoCUC[Rt] is
        §   let c = NewVec[2]
 35         c↓2 := Rt
            EnterinCUChain[c]
        §

        LoadF[S] is LoadFile[LookUp[S, 'IC', SCI]]
 40
        WorseErrorfoundthan[Type] = (WorstError > Type)
    $m


 45 let Prog[] be
    §P
        SetUpCompiler[]
        Run[Compiler]
        RunReports[]
 50     ReportS['*nCompilation *S', Compiled → 'succeeds', 'fails']
        if WorstError = COMPERROR do Run[PM]
    $P


 55 and Compiler[] be
    §C
        let c = CPtr
        LoadSystemFile['BitMap Rts']
        DoPass0[]
 60     if WorseErrorfoundthan[HARD] return
        DoPass1[]
```

```
        Unload[c]
        DoPass2[]
        DoPass3[]
65      if WorseErrorfoundthan[SOFT] return
        DoPass4[]
        unless WorseErrorfoundthan[SOFT] do Compiled := true
    $C

70
    and DoPass0[] be
    §P0
        let c = CPtr
        LoadF['PP']
75      Output := OuttoBFile[WF]

        PP[]

        Unload[c]
80  §P0


    and DoPass1[] be
    §P1
85      let c = CPtr
        LoadF['Pass 1']
        In := InfromBFile[WF]
        Output := OuttoBFile[WF]

90      Pass1[]

        Close[In]
        Close[Output]
        Unload[c]
95  §P1


    and DoPass2[] be
    §P2
100     let c = CPtr
        LoadF['Pass 2']
        In := RevInfromBFile[WF]
        Output := OuttoBFile[WF]

105     Pass2[]

        Close[In]
        Close[Output]
        Unload[c]
110 §P2


    and DoPass3[] be
    §P3
115     let c = CPtr
        LoadF['Pass 3a']
        LoadF['Pass 3b']
        In := RevInfromBFile[WF]
        Output := OuttoBFile[WF]
120
        Pass3[]

        Close[In]
        Close[Output]
125     Unload[c]
    §P3
```

```
    and DoPass4[] be
130 §P4
        let c = CPtr
        LoadF['Pass 4']
        In := RevInfromBFile[WF]
        Output := OutputStream
135
        Pass4[]

        Close[In]
        Close[Output]
140     Unload[c]
    $P4


    and RunReports[] be
145 §RR
        Close[ErrorStream]
        LoadSystemFile['Quicksort']
        LoadF['Print Reports']
        In := InputStream
150     Reset[In]
        Run[PrintReports]
    $RR


155 and SetUpCompiler[] be
    §SC
        SCI := LookUp['Compiler', 'Index', SystemIndex]
        WF := WorkFile['WorkFile']
        ErrorStream := OuttoFile[WorkFile['Error Reports']]
160     GlobList := NewVec[GlobSize × 2 + 1] + 1
        GlobList↓(-1) := 0
        GlobList↓0 := 0
        InputStream := In
        OutputStream := Output
165     Peep := NullProgram
        LoadSystemFile['Bilateral File Streams']
        Out[ReportStream, `*n`]
        AddtoCUC[ClearUpCompiler]
    $SC
170

    and ClearUpCompiler[] be
    §CC
        DeleteBody[WF]
175     DeleteBody[WorkFile['Error Reports']]
        DeleteBody[WorkFile['Names']]
        DeleteBody[WorkFile['Strings']]
        DeleteBody[WorkFile['Numbers']]
        DeleteBody[WorkFile['Gets']]
180 $CC


    and Error[n, Type, Char, Routine, P] be
    §E
185     let RepMax = (Type = SOFT → ReportSize/2, ReportSize)
        unless Char = ERROR ∨ ReportNo ≥ RepMax do
        §   ReportNo := ReportNo + 1
            if ReportNo = RepMax do
                test Type = SOFT
190               ifso  n := 11
                  ifnot n, Routine := 1, Finish
```

```
            Out[ErrorStream, PassNo]
            Out[ErrorStream, GetNo]
            Out[ErrorStream, LineNo]
195         Out[ErrorStream, n]
            Out[ErrorStream, Char]
        $
        unless WorseErrorfoundthan[Type] do WorstError := Type
        Routine[P]
200 $E


    and CompilerError[n] be
    §CE
205     ReportS['Compiler error *N', n]
        for i = 1 to 20 do
        §   OutS[ReportStream, '*O*C', Ch, i rem 10 = 0 → '*n', '*s']
            if Endof[In] do
            §   ReportS['End of Input']
210             break
            $
            Ch := Next[In]
        $
        Dump[]
215     WorstError := COMPERROR
        Finish[]
    §CE

    and PM[] be
220 §   Prog := DefaultProg
        LoadSystemFile['DumpPM']
        Run[Prog]
    $

****
```