```
-- FlowExpression.mesa, modified by Sweet, August 2, 1978  10:37 AM

DIRECTORY
    AltoDefs: FROM "altodefs" USING [BYTE, wordlength],
    Code: FROM "code" USING [acstack, CodeNotImplemented, firstcaseselread, mwcaseseltlex, ZEROlexeme],
**
    CodeDefs: FROM "codedefs" USING [BDOIndex, ChunkBase, CompareClass, JumpType, LabelCCIndex, Lexeme,
** lTOS, topostack],
    ComData: FROM "comdata",
    FOpCodes: FROM "fopcodes" USING [qAND, qDCOMP, qDSUB, qEXCH, qLI, qLINT, qLP, qNEG, qOR, qPOP, qPUS
**H, qXOR],
    P5ADefs: FROM "p5adefs" USING [adjustacstack, Cfin, Cflow, Ciout0, Ciout1, Cload, CompareFn, copyBD
**OItem, Coutjump, Csyscall,

    Csyscalln, dumpstack, easilyaddressed, genanonlex, gentemplex, incrstack, insertlabel, labelalloc,
**loadtsonaddress,
    loadtsonchars, LogHeapFree, LongTreeAddress, makeretlex, makeTOSlex, maketsonBDOItem, markstack, op
**erandtype, releasetemplex,

    RequireStack, resettomark, rmakeBDOItem, sCassign, stackoff, stackon, treeliteral, unmarkstack, wor
**dsforoperand],
    P5BDefs: FROM "p5bdefs" USING [Cexp, MWConstant, pushlex, pushlitval, pushrhs],
    SDDefs: FROM "sddefs" USING [sBLTE, sBYTBLTE, sFCOMP, sFLOAT, sFSUB],
    SymDefs: FROM "symdefs" USING [CSEIndex, CTXIndex, HTIndex, ISEIndex, SEIndex, setype],
    SymTabDefs: FROM "symtabdefs" USING [Cardinality, WordsForType],
    TableDefs: FROM "tabledefs" USING [TableBase, TableNotifier],
    TreeDefs: FROM "treedefs" USING [empty, listlength, NodeName, scanlist, TreeIndex, TreeLink, treety
**pe];

DEFINITIONS FROM FOpCodes, CodeDefs;

FlowExpression: PROGRAM
        IMPORTS CPtr: Code, P5ADefs, P5BDefs, SymTabDefs, TreeDefs
        EXPORTS CodeDefs, P5BDefs =
    BEGIN
    OPEN P5ADefs, P5BDefs;

    -- imported definitions

    BYTE: TYPE = AltoDefs.BYTE;
    wordlength: CARDINAL = AltoDefs.wordlength;

    sBLTE: BYTE = SDDefs.sBLTE;
    sBYTBLTE: BYTE = SDDefs.sBYTBLTE;

    CTXIndex: TYPE = SymDefs.CTXIndex;
    HTIndex: TYPE = SymDefs.HTIndex;
    ISEIndex: TYPE = SymDefs.ISEIndex;
    SEIndex: TYPE = SymDefs.SEIndex;

    empty: TreeLink = TreeDefs.empty;
    NodeName: TYPE = TreeDefs.NodeName;
    TreeIndex: TYPE = TreeDefs.TreeIndex;
    TreeLink: TYPE = TreeDefs.TreeLink;


    tb: TableDefs.TableBase;            -- tree base (local copy)
    seb: TableDefs.TableBase;           -- semantic entry base (local copy)
    cb: ChunkBase;              -- code base (local copy)

  FlowExpressionNotify: PUBLIC TableDefs.TableNotifier =
    BEGIN  -- called by allocator whenever table area is repacked
    seb ← base[SymDefs.setype];
    tb ← base[TreeDefs.treetype];
    cb ← LOOPHOLE[tb];
    RETURN
    END;

    JumpNN: ARRAY NodeName[relE..relLE] OF JumpType = [
                JumpE, JumpN, JumpL, JumpGE, JumpG, JumpLE];

    UJumpNN: ARRAY NodeName[relE..relLE] OF JumpType ← [
                JumpE, JumpN, UJumpL, UJumpGE, UJumpG, UJumpLE];

    CNN: ARRAY NodeName[relE..relLE] OF NodeName = [
                relN, relE, relGE, relL, relLE, relG];
```

```
    RNN: ARRAY NodeName[relE..relLE] OF NodeName = [
                relE, relN, relG, relLE, relL, relGE];

    PushOnly: PROCEDURE [t: TreeLink] =
        BEGIN
        IF t # empty THEN BEGIN RequireStack[0]; pushrhs[t] END
        ELSE BEGIN pushrhs[t]; RequireStack[1] END;
        RETURN
        END;

    Cflowexp: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [l: Lexeme] =
        BEGIN -- generates code for a flow expression

        SELECT (tb+node).name FROM
            ifexp => l ← Cifexp[node];
            or => l ← Cor[node];
            and => l ← Cand[node];
            not => l ← Cnot[node];
            relE,relN,relL,relGE,relG,relLE => l ← Crel[node, TRUE];
            in => l ← Cin[node, TRUE];
            notin => l ← Cin[node, FALSE];
            abs => l ← Cabs[node];
            lengthen => l ← Clengthen[node];
            min => l ← Cmin[node];
            max => l ← Cmax[node];
            ENDCASE =>
                BEGIN
                    SIGNAL CPtr.CodeNotImplemented;
                    l ← CPtr.ZEROlexeme;
                END;
        RETURN
        END;


    Cabs: PROCEDURE [node: TreeIndex] RETURNS [Lexeme] =
        BEGIN -- generate code for ABS
        poslabel, donelabel: LabelCCIndex;

        IF (tb+node).attr1 THEN RETURN [CLabs[node]];
        poslabel ← labelalloc[];
        donelabel ← labelalloc[];
        markstack[];
        PushOnly[(tb+node).son1];
        pushlitval[0];
        Coutjump[JumpGE, poslabel];
        Ciout0[qPUSH];
        Ciout0[qNEG];
        adjustacstack[-1];
        resettomark[];
        Coutjump[Jump, donelabel];
        insertlabel[poslabel];
        Ciout0[qPUSH];
        unmarkstack[];
        insertlabel[donelabel];
        RETURN[topostack]
        END;


    CLabs: PROCEDURE [node: TreeIndex] RETURNS [Lexeme] =
        BEGIN
        pos, done: LabelCCIndex;
        tlex: se Lexeme;
        r: BDOIndex;
        IF (tb+node).attr2 THEN RETURN[CRabs[node]];
        r ← rmakeBDOItem[Cexp[(tb+node).son1]];
        IF ~easilyaddressed[r] THEN
            BEGIN
            Cload[r];
            tlex ← gentemplex[2];
            sCassign[tlex.lexsei];
            r ← rmakeBDOItem[tlex];
            END;
        RequireStack[0];
        markstack[];
        Cload[copyBDOItem[r]];
```

```
        Ciout1[qLI, 0];
        Ciout1[qLI, 0];
        Ciout0[qDCOMP];
        pushlitval[0];
        Coutjump[JumpGE, pos ← labelalloc[]];
        Ciout1[qLI, 0];
        Ciout1[qLI, 0];
        Cload[copyBDOItem[r]];
        Ciout0[qDSUB];
        adjustacstack[-2]; resettomark[];
        Coutjump[Jump, done ← labelalloc[]];
        insertlabel[pos];
        Cload[r];
        unmarkstack[];
        insertlabel[done];
        RETURN [makeTOSlex[2]]
        END;

    CRabs: PROCEDURE [node: TreeIndex] RETURNS [Lexeme] =
        BEGIN
        pos, done: LabelCCIndex;
        tlex, zlex: se Lexeme;
        r: BDOIndex;
        RequireStack[0];
        markstack[];  -- float
        pushlitval[0];
        pushlitval[0];
        Csyscalln[SDDefs.sFLOAT,2];
        zlex ← gentemplex[2];
        sCassign[zlex.lexsei];
        markstack[];  -- conditional
        markstack[];  -- fcomp
        r ← rmakeBDOItem[Cexp[(tb+node).son1]];
        IF ~easilyaddressed[r] THEN
            BEGIN
            Cload[r];
            tlex ← gentemplex[2];
            sCassign[tlex.lexsei];
            r ← rmakeBDOItem[tlex];
            END;
        Cload[copyBDOItem[r]];
        pushlex[zlex];
        Csyscalln[SDDefs.sFCOMP,1];
        pushlitval[0];
        Coutjump[JumpGE, pos ← labelalloc[]];
        markstack[];  -- fsub
        pushlex[zlex];
        Cload[copyBDOItem[r]];
        Csyscalln[SDDefs.sFSUB,2];
        adjustacstack[-2]; resettomark[];
        Coutjump[Jump, done ← labelalloc[]];
        insertlabel[pos];
        Cload[r];
        unmarkstack[];
        insertlabel[done];
        RETURN [makeTOSlex[2]]
        END;

    Clengthen: PROCEDURE [node: TreeIndex] RETURNS [Lexeme] =
        BEGIN
        r: BDOIndex;
        r ← maketsonBDOItem[(tb+node).son1].lexbdoi;
        IF cb[r].offset.size = 2*wordlength THEN -- array descriptor
            BEGIN
            IF cb[r].tag = o AND cb[r].offset.level # lTOS THEN
                BEGIN
                copyr: BDOIndex = copyBDOItem[r];
                cb[copyr].offset.size ← wordlength;
                Cload[copyr]; -- base
                Ciout0[qLP];
                cb[r].offset.size ← wordlength;
                cb[r].offset.posn.wd ← cb[r].offset.posn.wd+1;
                Cload[r]; -- length
                END
            ELSE
                BEGIN
```

```
            tlex: se Lexeme = gentemplex[1];
            Cload[r];
            sCassign[tlex.lexsei]; -- length
            Ciout0[qLP];
            pushlex[tlex]; -- length
            END;
        RETURN[makeTOSlex[3]]
        END;
    Cload[r];
    IF (tb+node).attr1 THEN Ciout0[qLP]
    ELSE IF (tb+node).attr2 THEN Ciout1[qLI, 0]
    ELSE Ciout0[qLINT];
    RETURN [makeTOSlex[2]]
    END;


Cand: PROCEDURE [node: TreeIndex] RETURNS [Lexeme] =
    BEGIN -- generate code for "AND"
    t1: TreeLink ← (tb+node).son1;
    t2: TreeLink ← (tb+node).son2;

    WITH t1 SELECT FROM
        subtree => NULL;
        ENDCASE =>
            WITH t2 SELECT FROM
                subtree => NULL;
                ENDCASE =>
                    BEGIN
                    pushrhs[t1];
                    pushrhs[t2];
                    Ciout0[qAND];
                    RETURN[topostack]
                    END;
    RETURN[sCand[TRUE, t1, t2]]
    END;


sCand: PROCEDURE [tf: BOOLEAN, t1, t2: TreeLink] RETURNS [Lexeme] =
    BEGIN -- main subroutine for Cand
    label, elabel: LabelCCIndex;

    label ← labelalloc[];
    elabel ← labelalloc[];
    markstack[];
    BEGIN ENABLE LogHeapFree => RESUME[FALSE, topostack];
        Cflow[t1, FALSE, label];
        Cflow[t2, FALSE, label];
    END;
    pushlitval[IF tf THEN 1 ELSE 0];
    adjustacstack[-1];
    unmarkstack[];
    Coutjump[Jump,elabel];
    insertlabel[label];
    stackoff[];
    pushlitval[IF tf THEN 0 ELSE 1];
    stackon[];
    insertlabel[elabel];
    RETURN[topostack]
    END;


Cor: PROCEDURE [node: TreeIndex] RETURNS [Lexeme] =
    BEGIN -- generate code for "OR"
    t1: TreeLink ← (tb+node).son1;
    t2: TreeLink ← (tb+node).son2;

    WITH t1 SELECT FROM
        subtree => NULL;
        ENDCASE =>
            WITH t2 SELECT FROM
                subtree => NULL;
                ENDCASE =>
                    BEGIN
                    pushrhs[t1];
                    pushrhs[t2];
                    Ciout0[qOR];
```

```
                    RETURN[topostack]
                    END;
        RETURN[sCor[TRUE, t1, t2]]
        END;


    sCor: PROCEDURE [tf: BOOLEAN, t1,t2: TreeLink] RETURNS [Lexeme] =
        BEGIN -- main subroutine for Cor
        labelt, labelf, elabel: LabelCCIndex;

        labelt ← labelalloc[];
        labelf ← labelalloc[];
        elabel ← labelalloc[];
        markstack[];
        BEGIN ENABLE LogHeapFree => RESUME[FALSE, topostack];
            Cflow[t1, TRUE, labelt];
            Cflow[t2, FALSE, labelf];
        END;
        insertlabel[labelt];
        pushlitval[IF tf THEN 1 ELSE 0];
        adjustacstack[-1];
        unmarkstack[];
        Coutjump[Jump,elabel];
        insertlabel[labelf];
        stackoff[];
        pushlitval[IF tf THEN 0 ELSE 1];
        stackon[];
        insertlabel[elabel];
        RETURN[topostack]
        END;


    Cnot: PROCEDURE [node: TreeIndex] RETURNS [l: Lexeme] =
        BEGIN -- generate code for "NOT"
        node1: TreeIndex;

        BEGIN
        WITH (tb+node).son1 SELECT FROM
            subtree =>
                BEGIN
                node1 ← index;
                SELECT (tb+node1).name FROM
                    or => l ← sCor[FALSE, (tb+node1).son1, (tb+node1).son2];
                    and => l ← sCand[FALSE, (tb+node1).son1, (tb+node1).son2];
                    relE,relN,relL,relGE,relG,relLE => l ← Crel[node1, FALSE];
                    in => l ← Cin[node1, FALSE];
                    notin => l ← Cin[node1, TRUE];
                    not => pushrhs[(tb+node1).son1];
                    ENDCASE => GOTO VanillaNot;
                RETURN [topostack]
                END;
            ENDCASE;
        EXITS
            VanillaNot => NULL;
        END;
        pushrhs[(tb+node).son1];
        pushlitval[1];
        CioutO[qXOR];
        RETURN[topostack]
        END;


    Crel: PROCEDURE [node: TreeIndex, tf: BOOLEAN] RETURNS [Lexeme] =
        BEGIN -- produces code for relationals outside flow
        t1: TreeLink ← (tb+node).son1;
        t2, tt: TreeLink;
        n: NodeName ← (tb+node).name;
        tlabel: LabelCCIndex ← labelalloc[];
        elabel: LabelCCIndex ← labelalloc[];
        nwords: CARDINAL;
        sei: SymDefs.CSEIndex;
        function: CompareClass ← word;
        plength: [1..2];
        code: BOOLEAN ← FALSE;
        real: BOOLEAN;
        t1addrsize, t2addrsize: CARDINAL;
```

```
BEGIN
IF treeliteral[t1] THEN
    BEGIN
    n ← RNN[n];
    t2 ← t1; t1 ← (tb+node).son2;
    END
ELSE t2 ← (tb+node).son2;
IF ~tf THEN n ← CNN[n];
nwords ← wordsforoperand[t2];

    BEGIN
    IF t2.tag = literal THEN GO TO notpacked;
    sei ← operandtype[t2];
    WITH (seb+sei) SELECT FROM
        array =>
            BEGIN
            IF ~packed OR SymTabDefs.Cardinality[componenttype] > 8 THEN
                GO TO notpacked;
            nwords ← SymTabDefs.Cardinality[indextype];
            IF nwords <= 4 THEN
                BEGIN
                IF t1 # empty THEN RequireStack[0];
                loadtsonchars[t1, nwords];
                IF t1 = empty THEN RequireStack[(nwords+1)/2];
                loadtsonchars[t2, nwords];
                IF nwords <= 2 THEN Coutjump[UJumpNN[n], tlabel]
                ELSE
                    BEGIN
                    Ciout0[qDCOMP];
                    pushlitval[0];
                    Coutjump[JumpNN[n], tlabel];
                    END;
                GO TO loadresult
                END
            ELSE function ← byte;
            END;
        ENDCASE => GO TO notpacked;
    EXITS
        notpacked => nwords ← wordsforoperand[t2];
    END;
IF nwords > 1 THEN
    IF nwords = 2 THEN
        BEGIN
        RequireStack[0];
        IF (real ← (tb+node).attr1 AND (tb+node).attr2) THEN markstack[];
        IF t1 = empty THEN
            BEGIN
            CPtr.firstcaseselread←FALSE;
            pushlex[CPtr.mwcaseseltlex]
            END
        ELSE pushrhs[t1];
        pushrhs[t2];
        IF real THEN Csyscalln[SDDefs.sFCOMP,1] ELSE Ciout0[FOpCodes.qDCOMP];
        pushlitval[0];
        Coutjump[JumpNN[n], tlabel];
        END
    ELSE
        BEGIN
        dumpstack[]; markstack[];
        WITH t1 SELECT FROM
            subtree => IF (tb+index).name = mwconst THEN
                BEGIN tt ← t1; t1 ← t2; t2 ← tt END;
            ENDCASE;
        WITH t1 SELECT FROM
            subtree => IF (tb+index).name = mwconst THEN
                SIGNAL CPtr.CodeNotImplemented;
            ENDCASE;
        t1addrsize ← loadtsonaddress[t1];
        IF t1addrsize = wordlength AND LongTreeAddress[t2] THEN Ciout0[qLP];
        pushlitval[nwords];
        t2addrsize ← loadtsonaddress[t2
            !MWConstant =>
                BEGIN
                code ← TRUE;
                pushlitval[cOffset];
```

```
                    t2addrsize ← t1addrsize;
                    CONTINUE;
                    END];
            IF t1addrsize # t2addrsize THEN
                BEGIN
                IF t1addrsize > wordlength THEN Ciout0[FOpCodes.qLP];
                plength ← 2;
                END
            ELSE plength ← t1addrsize/wordlength;
            Csyscall[CompareFn[function, code, plength]];
            incrstack[1]; CPtr.acstack ← 1;
            IF n = relN THEN
                BEGIN pushlitval[1]; Ciout0[qXOR]; END;
            RETURN[topostack]
            END
        ELSE
            BEGIN
            PushOnly[t1];
            pushrhs[t2];
            Coutjump[IF (tb+node).attr2 THEN UJumpNN[n] ELSE JumpNN[n], tlabel];
            END;
        EXITS loadresult => NULL;
        END;
        pushlitval[0];
        adjustacstack[-1];
        Coutjump[Jump, elabel];
        insertlabel[tlabel];
        stackoff[];
        pushlitval[1];
        stackon[];
        insertlabel[elabel];
        RETURN[topostack]
        END;


    Cin: PROCEDURE [node: TreeIndex, tf: BOOLEAN] RETURNS [Lexeme] =
        BEGIN -- generates code for IN expression (outside flow)
        label: LabelCCIndex ← labelalloc[];
        elabel: LabelCCIndex ← labelalloc[];

        markstack[];
        Cfin[TreeLink[subtree[node]], FALSE, label];
        pushlitval[IF tf THEN 1 ELSE 0];
        Coutjump[Jump, elabel];
        unmarkstack[];
        adjustacstack[-1];
        insertlabel[label];
        stackoff[];
        pushlitval[IF tf THEN 0 ELSE 1];
        stackon[];
        insertlabel[elabel];
        RETURN[topostack]
        END;


    Cifexp: PROCEDURE [node: TreeIndex] RETURNS [Lexeme] =
        BEGIN -- generates code for an IF expression
        ilabel, elabel: LabelCCIndex;
        nwords: INTEGER = SymTabDefs.WordsForType[(tb+node).info];
        tlex: se Lexeme ← topostack;
        thenpsize, elsepsize: CARDINAL ← wordlength;

        elabel ← labelalloc[];
        markstack[];
        Cflow[(tb+node).son1, FALSE, elabel];
        ilabel ← ilabel; -- copes with following ENABLE and its CATCH
        BEGIN ENABLE
            BEGIN
            LogHeapFree => RESUME[FALSE, topostack];
            MWConstant =>
                BEGIN
                IF tlex = topostack THEN tlex ← genanonlex[nwords];
                RESUME[tlex];
                END
            END;
            IF nwords > 2 THEN
```

```
            BEGIN
            thenpsize ← loadtsonaddress[(tb+node).son2];
            IF thenpsize = wordlength AND LongTreeAddress[(tb+node).son3] THEN
                BEGIN Ciout0[qLP]; thenpsize ← 2*wordlength END;
            resettomark[];
            adjustacstack[-thenpsize/wordlength]
            END
        ELSE
            BEGIN
            pushrhs[(tb+node).son2];
            resettomark[];
            adjustacstack[-nwords];
            END;
        Coutjump[Jump, ilabel ← labelalloc[]];
        insertlabel[elabel];
        IF nwords > 2 THEN elsepsize ← loadtsonaddress[(tb+node).son3]
        ELSE pushrhs[(tb+node).son3];
    END;
    IF thenpsize # elsepsize THEN Ciout0[FOpCodes.qLP];
    unmarkstack[];
    insertlabel[ilabel];
    IF tlex # topostack THEN releasetemplex[tlex];
    RETURN[makeretlex[nwords, thenpsize]]
    END;


Cmin: PROCEDURE [node: TreeIndex] RETURNS [Lexeme] =
    BEGIN -- generate code for "MIN[...]"
    sCminmax[relL,(tb+node).son1,(tb+node).attr1,(tb+node).attr2];
    RETURN[IF ~(tb+node).attr1 THEN topostack ELSE makeTOSlex[2]]
    END;


Cmax: PROCEDURE [node: TreeIndex] RETURNS [Lexeme] =
    BEGIN -- generates code for "MAX[...]"
    sCminmax[relG,(tb+node).son1,(tb+node).attr1,(tb+node).attr2];
    RETURN[IF ~(tb+node).attr1 THEN topostack ELSE makeTOSlex[2]]
    END;


sCminmax: PROCEDURE [n: NodeName, t: TreeLink, double, realorunsigned: BOOLEAN] =
    BEGIN -- common subroutine for Cmin and Cmax
    node: TreeIndex;
    label1, label2: LabelCCIndex;
    tlex: se Lexeme;
    t2lex: se Lexeme ← topostack;
    firstson: BOOLEAN ← TRUE;
    sscm: PROCEDURE [t: TreeLink] =
        BEGIN
        label3: LabelCCIndex;
        r: BDOIndex;
        IF firstson THEN BEGIN firstson ← FALSE; RETURN END;
        label3 ← labelalloc[];
        IF double THEN
            BEGIN
            IF realorunsigned THEN markstack[];
            r ← rmakeBDOItem[Cexp[t]];
            IF ~easilyaddressed[r] THEN
                BEGIN
                IF t2lex = topostack THEN t2lex ← gentemplex[2];
                Cload[r];
                sCassign[t2lex.lexsei];
                r ← rmakeBDOItem[t2lex];
                END;
            Cload[copyBDOItem[r]];
            END
        ELSE pushrhs[t];
        pushlex[tlex];
        IF double THEN
            BEGIN
            IF realorunsigned THEN Csyscalln[SDDefs.sFCOMP,1] ELSE Ciout0[qDCOMP];
            pushlitval[0];
            END;
        Coutjump[IF realorunsigned THEN UJumpNN[RNN[n]] ELSE JumpNN[RNN[n]], label3];
        IF double THEN Cload[r] ELSE Ciout0[qPUSH];
        sCassign[tlex.lexsei];
```

```
            insertlabel[label3];
            RETURN
            END;

        WITH t SELECT FROM
            subtree =>
                BEGIN
                node ← index;
                RequireStack[0];
                pushrhs[(tb+node).son1];
                IF ~double AND TreeDefs.listlength[t] = 2 THEN
                    BEGIN
                    pushrhs[(tb+node).son2];
                    label1 ← labelalloc[]; label2 ← labelalloc[];
                    Coutjump[IF realorunsigned THEN UJumpNN[n] ELSE JumpNN[n], label1];
                    Ciout0[qPUSH]; Ciout0[qPUSH]; Ciout0[qEXCH]; Ciout0[qPOP];
                    Coutjump[Jump,label2];
                    adjustacstack[-1];
                    insertlabel[label1];
                    stackoff[];
                    Ciout0[qPUSH];
                    stackon[];
                    insertlabel[label2];
                    RETURN;
                    END;
                tlex ← gentemplex[IF double THEN 2 ELSE 1];
                sCassign[tlex.lexsei];
                TreeDefs.scanlist[t, sscm];
                pushlex[tlex];
                END;
            ENDCASE;
        RETURN
        END;


    END...
```