```
The WoodCutting Problem
=======================

(the translation of "WC5.txt" to a mathematical problem,
see "WC5.txt" for constant and type declarations; the translation
may contain errors, compare with the original definitions in "WC5.txt")

// translation of filtering conditions (if not available in target language):
// "∀x:T with p(x). q(x)" ~> "∀x:T. p(x) ⇒ q(x)"
// "∃x:T with p(x). q(x)" ~> "∃x:T. p(x) ∧ q(x)"

// can mostly use enumeration types for the decisions
type ReorderDecision = { none, forward, moveout, movein };
type CutDecision = Record[cnum:CutIndex, cuts: Cuts];
type FilterDecision = { none, keep, discard };
type AssemblyDecision = { none, accept };

The Problem
===========

Given:

  inboards: InBoards, // the list of input boards
  bempty: Bool,       // is the board buffer empty?
  buffer: Board,      // the board buffer
  pempty: Bool,       // is the pieces buffer empty?
  pbuffer: Piece,     // the pieces buffer
  fints: ForbiddenIntervals // the forbidden zones for cuts

Find:

  rbds: Array[RBDNUM,ReorderDecision], // the reorder board decisions
  cbds: Array[CBDNUM,CutDecision],     // the cut board decisions
  fpds: Array[DPNUM,FilterDecision],   // the filter piece decisions
  rpds: Array[RPDNUM,ReorderDecision], // the reorder piece decisions

Where:

  rbds,cbds,fpds, rpds = argmin_(
      rbds: Array[RBDNUM,ReorderDecision],
      cbds: Array[CBDNUM,CutDecision],
      fpds: Array[DPNUM,FilterDecision],
      rpds: Array[RPDNUM,ReorderDecision]).
    choose cost': Cost.
      Main(rbds, cbds, fpds, rpds,
        inboards, bempty, buffer, pempty, pbuffer, fints, cost')

The Predicates
==============

// the production line (consisting of multiple "stages")
Main(
  rbds: Array[RBDNUM,ReorderDecision],
  cbds: Array[CBDNUM,CutDecision],
```

```
  fpds: Array[DPNUM,FilterDecision],
  rpds: Array[RPDNUM,ReorderDecision]).
  inboards: InBoards,
  bempty: Bool, buffer: Board, pempty: Bool, pbuffer: Board,
  fints: ForbiddenIntervals,
  // "cost" dropped and replaced by fixed value 0
  cost':Cost
) ⇔
(
  // the value sequences for the "inout" parameters
  ∃costs: Array[FPDNUM+1,Cost].

  // their initial and final values ("cost" replaced by fixed value 0)
  costs[0] = 0 ∧ costs[FPDNUM] = cost' ∧

  // requirement on input (add it to solver?)
  // ∀i:BoardIndex.
  //   let board: Board = inboards[i] in
  //   ∀j:BoardIntervalIndex with j < board.bintnum.
  //     let bint:BoardInterval = board.bints[j] in
  //     bint.interval.from < bint.interval.to ∧ // intervals are not empty
  //     if j+1 < board.bintnum
  //       then bint.interval.to ≤ board.bints[j+1].interval.from
  //       else bint.interval.to ≤ board.length;

  // the local "val" values and the value sequences for the "var" variables
  ∃bemptys: Array[RBDNUM+1,Bool].
  ∃buffers: Array[RBDNUM+1,Board].
  ∃pemptys: Array[RPDNUM+1,Bool].
  ∃pbuffers: Array[RPDNUM+1,Piece].
  ∃outboards: Boards.
  ∃obnums: Array[RBDNUM+1,BoardIndex].
  ∃inpieces: InPieces.
  ∃ipnums: Array[CBDNUM+1,PieceIndex].
  ∃outpieces: OutPieces.
  ∃opnums: Array[FPDNUM+1,PieceIndex].
  ∃apieces: OutPieces.
  ∃apnums: Array[APDNUM+1,PieceIndex].

  // the initial values of the "var" variables
  bemptys[0] = bempty ∧ buffers[0] = buffer ∧
  pemptys[0] = pempty ∧ pbuffers[0] = pbuffer ∧
  obnums[0] = 0 ∧ ipnums[0] = 0 ∧ opnums[0] = 0 ∧ apnums[0] = 0 ∧

  ∀i:Int[0,RBDNUM-1].
  (
    if (i < NUM ∧ obnum < OBNUM) ∨
       (i < NUM ∧ (bempty ∨ obnum < OBNUM)) ∨
       (i = NUM ∧ ¬bempty) then
      Reorder(rbds[i], i, inboards, outboards,
        obnums[i], obnums[i+1], bemptys[i], bemptys[i+1], buffers[i], buffers[i+1])
    else
      rdbs[i] = none ∧
      obnums[i+1] = obnums[i] ∧ bemptys[i+1] = bemptys[i] ∧ buffers[i+1] = buffers[i]
```

```
) ∧

// should be ensured by above
// obnums[RBDNUM] = NUM ∧

∀i:Int[0,CBDNUM-1].
(
  Cut(cdbs[i],outboards,inpieces,ipnums[i],ipnums[i+1])
) ∧

∀i:Int[0,FPDNUM-1].
(
  if (i < ipnum ∧ opnum < OPNUM) ∨ (i < ipnum) then // can be simplified to "i < ipnum"
    Filter(dds[i], i,
      inpieces, ipnums[CBDNUM-1], outpieces, opnums[i], opnums[i+1],
      costs[i], costs[i+1])
  else
    dds[i] = none ∧
    opnums[i+1] = opnums[i] ∧ costs[i+1] = costs[i]
) ∧

∀i:Int[0,RPDNUM-1].
(
  if (i < opnum ∧ apnum < PNUM) ∨
     (i < opnum ∧ (pempty ∨ apnum < PNUM)) ∨
     (i = opnum ∧ ¬pempty) then
    ReorderPieces(rpds[i], i,
      outpieces, opnum, apieces, apnums[i], apnums[i+1],
      pemptys[i], pemptys[i+1], pbuffers[i], pbuffers[i+1])
  else
    rpds[i] = none ∧
    apnums[i+1] = apnums[i] ∧
    pemptys[i+1] = pemptys[i] ∧ pbuffers[i+1] = pbuffers[i]
) ∧

// should be ensured by above
// apnums[RPDNUM] = opnums[FPDNUM];

// the value sequences for the "var" variables
∃blens: BeamLengths.
∃blen: Array[APDNUM+1,BeamLength].
∃bnum: Array[APDNUM+1,BeamIndex];
∃bdepth: Array[APNUM+1,BeamDepth];
∃bnum0: Array[APNUM+1,BeamIndex];

// the initial values of the "var" variables
blen[0] = 0 ∧ bnum[0] = 0 ∧ bdepth[0] = 0 ∧ bnum0[0] = 0 ∧

∀i:Int[0,APDNUM-1].
(
  if i < apnum then
    Assembly(accept, i,  // "accept" actually superfluous, can be removed
      apieces, apnums[APNUM], fints, blens,
      blen[i], blen[i+1], bnum[i], bnum[i+1],
```

```
        bdepth[i], bdepth[i+1], bnum0[i], bnum0[i+1])
    else
      blen[i+1] = blen[i] ∧ bnum[i+1] = bnum[i] ∧
      bdepth[i+1] = bdepth[i] ∧ bnum0[i+1] = bnum0[i]
  )
);

// the reordering stage (boards in, boards out)
stage Reorder(d: ReorderDecision,
  i: BoardIndex,     // the index of the board to process next
  inboards: Boards,  // the input boards
  outboards: Boards, // the output boards, unconstrained at indices >= obnum
  obnum: BoardIndex, obnum': BoardIndex, // the number of output boards produced
  bempty: Bool, bempty': Bool,           // the state of the board buffer
  buffer: Board, buffer': Board          // the board buffer
) ⇔
(
  (d = forward ∧
    i < NUM ∧ obnum < NUM ∧
    let board: Board = inboards[i] in
    outboards[obnum] = board ∧
    obnum' = obnum+1 ∧
    bempty' = bempty ∧ buffer' = buffer
  ) ∨
  (d = moveout ∧
    ibnum < NUM ∧ (bempty ∨ obnum < NUM) ∧
    let board: Board = inboards[i] in
    (¬bempty ⇒ outboards[obnum] = buffer) ∧
    obnum' = if bempty then obnum else obnum+1 ∧
    bempty' = false ∧
    buffer' = board;
  ) ∨
  (d = movein ∧
    i = NUM ∧ ¬bempty ∧
    outboards[obnum] = buffer ∧
    obnum' = obnum+1 ∧
    bempty' = true
    // buffer' can be arbitrary
  )
);

// the cutting stage (boards in, pieces out)
stage Cut(d: CutDecision,
  i: BoardIndex,        // the index of the board to be processed next
  outboards: OutBoards, // the boards to be processed
  inpieces: InPieces,   // the pieces generated, unconstrained at indices >= ipnum
  ipnum: InPieceIndex, ipnum': InPieceIndex // the number of pieces generated
) ⇔
(
  let cnum:CutIndex = d.cnum in
  let cuts:Cuts = d.cuts in
  let board: Board = outboards[i] in
  // only allowed cuts
  (∀j: CutIndex with j < cnum.
```

```
    // cuts are strictly ordered
    let start: Length = if j = 0 then 0 else cut[j-1] in
    start < cut[j] ∧ (j = cnum-1 ⇒ cut[j] < board.length) ∧
    // cuts are not strictly within bad intervals
    !∃k: BoardIntervalIndex with k < board.bintsnum.
       val bint: BoardInterval = boards.bints[k].
       bint.type = bad ∧
       bint.interval.from < cut[j] ∧ cut[j] < bint.interval.to) ∧
  // all necessary cuts
  (∀k: BoardIntervalIndex with k < board.bintsnum.
    let bint: BoardInterval = board.bints[k] in
    if bint.type = curved then
      ∃j: CutIndex with j < cnum.
        bint.interval.from ≤ cuts[j] ∧ cuts[j] ≤ bint.interval.to
    else // if cint.type = bad then
      ∃j: CutIndex with j < cnum-1.
        bint.interval.from = cut[j] ∧ bint.interval.to = cut[j+1]) ∧
  // the resulting piece (can be combined with "only allowed cuts")
  (∀j: CutIndex with j < cnum.
    let inpiece: InPiece = inpieces[ipnum+j] in
    let start: Length = if j = 0 then 0 else cut[j-1] in
    inpiece.length = cut[j]-start ∧
    inpiece.good =
      ¬∃k: BoardIndex with k < board.bintsnum.
        let bint: BoardInterval = boards.bints[k] in
        bint.type = bad ∧ start = bint.interval.from) ∧
  ipnum' = ipnum+cnum
);

// the filtering stage (pieces in, pieces out)
Filter(d: FilterDecision,
  i: PieceIndex,        // the index of the next piece to be processed
  ipnum: InPieceIndex,  // the number of pieces available
  inpieces: InPieces,   // the pieces to be processed
  outpieces: OutPieces; // the pieces forwarded, unconstrained at indices >= opnum
  opnum: PieceIndex, opnum': PieceIndex // the number of pieces forward
  cost: Cost, cost': Cost // the accumulated cost of filtering good pieces
) ⇔
(
  (d = keep ∧
    i < ipnum ∧
    let piece: Piece = inpieces[i] in
    piece.good ∧ piece.length >= PLEN ∧
    outpieces[opnum] = piece ∧
    opnum' = opnum+1 ∧
    cost' = cost
  ) ∨
  (d = discard ∧
    i < ipnum ∧
    let piece: Piece = inpieces[i] in
    cost' = if piece.good then cost+piece.length else cost; ∧
    opnum' = opnum;
  )
);
```

```
// the reordering stage (pieces in, pieces out)
ReorderPieces(
  d: ReorderDecision,
  i: PieceIndex, // the index of the next piece to be processed
  outpieces: OutPieces, // the pieces to be processed
  opnum: PieceIndex,    // the number of pieces available
  apieces: OutPieces,   // the pieces forwarded, unconstrained at indices >= apnum
  apnum: OutPieceIndex, apnum': OutPieceIndex, // the number of pieces forwarded
  pempty: Bool, pempty': Bool,    // the state of the piece buffer
  pbuffer: Piece, pbuffer': Piece // the piece buffer
) ⇔
(
  (d = forward ∧
     i < opnum ∧ apnum < PNUM ∧
     let piece: OutPiece = outpieces[i] in
     apieces[apnum] = piece ∧ // equality, not assignment!
     apnum' = apnum+1 ∧
     pempty' = pempty ∧ pbuffer' = pbuffer
  ) ∨
  (d = moveout ∧
    i < opnum ∧ (pempty ∨ apnum < PNUM) ∧
    let piece: OutPiece = outpieces[i] in
    (¬pempty ⇒ apieces[apnum] = pbuffer) ∧
    apnum' = if pempty then apnum else apnum+1 ∧
    pempty' = false ∧
    pbuffer' = piece
  ) ∨
  (d = movein ∧
    i = opnum ∧ ¬pempty ∧
    apieces[apnum] = pbuffer ∧
    apnum' = apnum+1 ∧
    pempty' = true
    // pbuffer' can be arbitrary
  )
);

// the assembly stage (pieces in, accepted or not)
Assembly(d: AssemblyDecision, // actually superfluous, can be removed
  i: PieceIndex,       // the index of the next piece to be processed
  apieces: OutPieces,  // the pieces to be processed
  apnum: PieceIndex,   // the number of pieces available
  fints: ForbiddenIntervals, // the forbidden zones
  blens: BeamLengths, // the sequence of beam lengths, unconstrained at indices >= i
  blen: BeamLength, blen': BeamLength, // the length of the current layer
  bnum: BeamIndex, bnum': BeamIndex,   // the number of pieces in the current layer
  bdepth: BeamDepth, bdepth': BeamIndex, // the number of completed layers
  bnum0: BeamIndex, bnum0': BeamIndex    // the number of pieces in the previous layer
) ⇔
  // for layers [2,5,3],[1,9],[5,3,2] we have:
  // blen =   0, 2, 7,  0, 1,  0, 5, 8
  // blens = [2, 7, 10, 1, 10, 5, 8, 10]
  // bnum =   0, 1, 2,  0, 1,  0, 1, 2
  // bdepth = 0, 0, 0,  1, 1,  2, 2, 2
```

```
  // bnum0 =  0, 0, 0,  3, 3,  2, 2, 2
(
  (d = accept ∧
    i < apnum ∧
    let blen0: BeamLength = blen+apieces[i] in
    blens[i] = blen0 ∧
    blen0 ≤ BLEN ∧
    (¬∃j:ForbiddenIndex. j < FNUM ∧
      fints[j].from ≤ blen0 ∧ blen0 ≤ fints[j].to) ∧
    (∀j:BeamIndex. j < bnum0 ⇒
      let diff: BeamLength = blen0-blens[i-bnum-bnum0+j] in
      DIFF ≤ if diff ≥ 0 then diff else -diff) ∧
    if blen0 < BLEN then
      blen' = blen0 ∧
      bnum' = bnum+1 ∧
      bdepth' = bdepth ∧ bnum0' = bnum0
    else
      blen' = 0 ∧
      bnum' = 0 ∧
      if bdepth < BDEPTH-1 then
        bdepth' = bdepth+1 ∧
        bnum0' = bnum
      else
        bdepth' = 0 ∧
        bnum0' = 0;
  )
);
```