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

(the translation of "WC4.txt" to a mathematical problem,
see "WC4.txt" for constant and type declarations; the translation
may contain errors, compare with the original definitions in "WC4.txt")

// 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,
  bempty: Bool,
  buffer: Board,
  pempty: Bool,
  pbuffer: Piece,
  ipnum: InPieceIndex,
  fints: ForbiddenIntervals

Find:

  rbds: Array[RBDNUM,ReorderDecision],
  cbds: Array[CBDNUM,CutDecision],
  fpds: Array[DPNUM,FilterDecision]
  rpds: Array[RPDNUM,ReorderDecision],

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.

      ∃bempty': Bool, buffer': Board, pempty': Bool, pbuffer': Piece.
      Main(rbds, cbds, fpds, rpds,
        inboards,
        bempty, bempty', buffer, buffer',
        pempty, pempty', pbuffer, pbuffer',
        fints,
        0, 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, bempty': Bool, buffer: Board, buffer': Board,
  pempty: Bool, pempty': Bool, pbuffer: Board, pbufffer': Board,
  fints: ForbiddenIntervals,
  cost: Cost, cost':Cost,
) ⇔
(
  // the value sequences for the "inout" parameters
  ∃bemptys: Array[RBDNUM+1,Bool].
  ∃buffers: Array[RBDNUM+1,Board].
  ∃pemptys: Array[RPDNUM+1,Bool].
  ∃pbuffers: Array[RPDNUM+1,Piece].
  ∃costs: Array[FPDNUM+1,Cost].

  // their initial and final values
  bemptys[0] = bempty ∧ bemptys[RBDNUM] = bempty' ∧
  buffers[0] = buffer ∧ buffers[RBDNUM] = buffer' ∧
  pemptys[0] = pempty ∧ pemptys[RPDNUM] = pempty' ∧
  pbuffers[0] = pbuffer ∧ pbuffers[RPDNUM] = pbuffer' ∧
  costs[0] = cost ∧ costs[FPDNUM] = cost' ∧

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

  // the local "val" values and the value sequences for the "var" variables
  ∃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
  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, 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
stage Reorder(d: ReorderDecision,
  i: BoardIndex,
  inboards: Boards,
  outboards: Boards, // unconstrained at indices >= obnum
  obnum: BoardIndex, obnum': BoardIndex,
  bempty: Bool, bempty': Bool,
  buffer: Board, buffer': Board
) ⇔
(
  (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
stage Cut(d: CutDecision,
  i: BoardIndex,
  outboards: OutBoards,
  inpieces: InPieces, // unconstrained at indices >= ipnum
  ipnum: InPieceIndex, ipnum': InPieceIndex
) ⇔
(
  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.
    let start: Length = if j = 0 then 0 else cut[j-1] in
    start < cut[j] ∧ (j = cnum-1 ⇒ cut[j] < board.length) ∧
    ∃k: BoardIntervalIndex with k < board.bintsnum.
      let bint: BoardInterval = boards.bints[k] in
      let from: Length = bint.interval.from in
      let to: Length = bint.interval.to in
      if bint.type = curved then
        from ≤ cut[j] && cut[j] ≤ to
      else // if bint.type = bad then
        (cut[j] = from ∧ j+1 < cnum ∧ cut[j+1] = to) ∨
        (cut[j] = to ∧ j-1 > 0 ∧ cut[j-1] = from)) ∧
  // 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
Filter(d: FilterDecision,
  i: PieceIndex,
  ipnum: InPieceIndex,
  inpieces: InPieces,
  outpieces: OutPieces; // unconstrained at indices >= opnum
  opnum: PieceIndex, opnum': PieceIndex
  cost: Cost, cost': Cost
) ⇔
(
  (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;
  )
);

ReorderPieces(
  d: ReorderDecision,
  i: PieceIndex,
  outpieces: OutPieces,
  opnum: PieceIndex,
  apieces: OutPieces, // unconstrained at indices >= apnum
  apnum: OutPieceIndex, apnum': OutPieceIndex,
  pempty: Bool, pempty': Bool,
  pbuffer: Piece, pbuffer': Piece
) ⇔
(
  (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
Assembly(d: AssemblyDecision, // actually superfluous, can be removed
  i: PieceIndex,
  apieces: OutPieces,
  apnum: PieceIndex,
  fints: ForbiddenIntervals,
  blens: BeamLengths, // unconstrained at indices >= i
  blen: BeamLength, blen': BeamLength,
  bnum: BeamIndex, bnum': BeamIndex,
  bdepth: BeamDepth, bdepth': BeamIndex,
  bnum0: BeamIndex, bnum0': BeamIndex
```

```
) ⇔
(
  (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 then
        bdepth' = 0 ∧
        bnum0' = 0
      else
        bdepth' = bdepth+1 ∧
        bnum0' = bnum
  )
);
```