```
(*************************************************************************)
(* VERSION 1.2                                                           *)
(* DECRULES.INT                                    TIMED RISKLESS SIM    *)
(*                                                 10/24/85              *)
(*************************************************************************)

INTERFACE;
UNIT DECRULES(DoRand,DoEBA,DoLexiSemi,DoLexi,DoEqualWeight,DoMCD,
              DoEBAEU,DoEBAMCD,DoAddUtility,DoTimedAU,DoSatisficing,
              DoARule);
USES SIM;


PROCEDURE DoRand(VAR Decision: INTEGER);
(* Choose randomly from the alternatives in the list. *)

PROCEDURE DoEBA(VAR Decision,EBARulCnt,EBARndCnt,EBALimit: INTEGER);

PROCEDURE DoLexiSemi(VAR Decision,LSORulCnt,LSORndCnt,LexLimit: INTEGER);

PROCEDURE DoLexi(VAR Decision,LxLimit: INTEGER);

PROCEDURE DoEqualWeight(VAR Decision,EqWLimit: INTEGER);
(* Based on Risky version of EquiProb. *)

PROCEDURE DoMCD(VAR Decision,MCDTieCnt,MCDLimit: INTEGER);

PROCEDURE DoEBAEU(VAR Decision,EEULimit: INTEGER);

PROCEDURE DoEBAMCD(VAR Decision,ECDLimit: INTEGER);

PROCEDURE DoAddUtility(VAR Decision: INTEGER);

PROCEDURE DoTimedAU(VAR Decision,AdULimit: INTEGER);

PROCEDURE DoSatisficing (VAR Decision,SatRulCnt,SatRndCnt,SATLimit:INTEGER);

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*  Run the rules.                                                         *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

PROCEDURE DoARule(Rule: Rules);

END;
```

```
(*************************************************************************)
(* VERSION 1.2                                    TIMED RISKLESS SIM      *)
(* DECRULES.PAS                                   11/22/85                *)
(*************************************************************************)
(* Time up processing:                                                   *)
(*       1. Doesn't apply  - RAND                                        *)
(*       2. Pick at random - EBA, Satisficing, LexiSemi, Lexi            *)
(*       3. Current best   - AddUtility, MCD, Equalweight                *)
(*          Current best if available, else random choice.               *)
(*       4. Combination    - EBAMCD, EBAEU                               *)
(*          If rule in EBA mode, random choice.  If in MCD or EU mode    *)
(*          when time is up, use their current best procedures.          *)
(*************************************************************************)
(* 9/27 - AU rule divided into untimed original AU code and timed        *)
(* code, now known as rule "TimedAU".  Untimed used to collect values    *)
(* used as baseline for calc of JPIndex.  Note that rule numbers are     *)
(* different in output than for riskless SIM... there's one more rule    *)
(* here: AU.  Invoke the same way however: Rules 0-9.                    *)
(*************************************************************************)

{$include:'fformsuu.int'}
{$include:'screenvv.int'}
{$include:'sim.int'}
{$include:'lstprims.int'}
{$include:'stmprims.int'}
{$include:'random.int'}
{$include:'decrules.int'}
IMPLEMENTATION OF DECRULES;
USES sim,lstprims,stmprims,random;
USES fformsuu,screenvv;
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*  The timed riskless decision rules:                                   *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

PROCEDURE DoRand;
(* Choose randomly from the alternatives. *)

    VAR   TT, II: INTEGER;
        RealTemp: AttVec;

    BEGIN (* DoRand *)
       TT:= IntRand(0,Cardinality(Altern) - 1); (* Choose index of choice *)

        IF AtEnd(Altern) THEN GetNext(Altern);
        FOR II:= 1 TO TT DO BEGIN       (* Loop to the choice. *)
            GetNext(Altern);
            IF AtEnd(Altern) THEN GetNext(Altern);
        END;
        Decision:= CurrValue(Altern);

    END; (* DoRand *)



PROCEDURE DoEBA;

VAR Deciding: BOOLEAN;

BEGIN  (* DoEBA *)
```

```
Deciding   := TRUE;
TimeUp     := FALSE;
Time[Rule] := 0.0;  (* Reinitialize counter for this iteration. *)


WHILE ((Deciding) AND (TimeUp=FALSE)) DO

   (* Sets out-of-time condition in Timed Riskless SIM: *)
   IF Time[Rule] >= TimeLimit THEN
   BEGIN
      Deciding := FALSE;
      TimeUp   := TRUE;


END ELSE IF (Cardinality(Altern) = 1) AND (NOT AtEnd(Altern)) THEN
   BEGIN
      Deciding:= FALSE;              (* We have eliminated down to 1, *)
      Decision:= CurrValue(Altern); (*  so return that.               *)
      EBARulCnt:=EBARulCnt + 1;      (* Counter external to rule.     *)
      (* For debug; external to rule:                                 *)
      IF (WinsOut = TRUE) AND (q < 10) THEN  (* Put winner in array  *)
      BEGIN                                  (* and output to Trace- *)
         q := q + 1;                         (* Out file.            *)
         EBADec[q] := CurrValue(Altern);
      END;

   END ELSE IF (Cardinality(Attrib) < 1) THEN
 (* We checked all attributes and more than one alternative is left, *)
 (*    so choose at random.                                          *)
   BEGIN
      DoRand(Decision);
      Deciding:= FALSE;
      EBARndCnt:=EBARndCnt + 1;      (* Counter external to rule.     *)
      (* For debug; external to rule:                                 *)
      IF (WinsOut = TRUE) AND (q < 10) THEN
      BEGIN
         q := q + 1;
         EBADec[q] := Decision;
      END;

   END ELSE IF (AtEnd(Altern) ) THEN
   (* We have eliminated all we can with this attribute.            *)
   BEGIN
      Elim(Attrib);
      SetPoint(5);                   (* Go back to start.            *)
      Forget(6);                     (* Marks that we need a new most *)
                                     (* important attribute.         *)
      Forget(5);
      Forget(2);                     (* Marks need for new cutoff.   *)

   END ELSE IF (AtEnd(Attrib) AND NOT IsEmpty(6)) THEN
   (* We have the most important attribute, so start processing:    *)
   BEGIN
      SetPoint(6);
      Remember(1);
      GetCut(1,2);
      GetMin(1,2,3);                  (* Get min of cutoff or value. *)

   END ELSE IF NOT IsEmpty(3) AND (STM[3].Alt<>CutOffVal) THEN
```

```
    (* The current attribute did not meet the cutoff; it was the min. *)
    BEGIN
       Elim(Altern);
       Forget(3);

    END ELSE IF NOT IsEmpty(3) AND (STM[3].Alt = CutOffVal) THEN
    (* The current attribute met the cutoff; cutoff was the min.      *)
    BEGIN
       GetNext(Altern);
       Forget(3);

    END ELSE IF (IsEmpty(2) AND IsEmpty(4) AND IsEmpty(5)) THEN
    (* Initialize search for the most important attribute.            *)
    BEGIN
       GetWeight(5);
       GetNext(Attrib);              (* Riskless GetNext skips wgts.   *)
       GetMax(5,5,6);


    END ELSE IF (IsEmpty (2) AND IsEmpty(4) AND NOT IsEmpty(6) ) THEN
    (* Continue search for the most important attribute.              *)
    BEGIN
       GetWeight(4);
       GetMax(4,6,6);                (* Current winner.                *)
       Forget(4);                    (* Mark that we still need one.   *)
       GetNext(Attrib);              (* Riskless GetNext skips wgts.   *)


    END ELSE
    (* Check this attribute against the cutoff.                       *)
    BEGIN
       Remember(1);
       GetMin(1,2,3);                (* Min of cutoff and value.       *)

    END;    (* WHILE *)


    IF (TimeUp=TRUE) THEN
    BEGIN
       DoRand(Decision);
       EBALimit := EBALimit + 1;

       EBARndCnt:=EBARndCnt + 1;      (* Counter external to rule.     *)
       (* For debug; external to rule:                                *)
       IF (WinsOut = TRUE) AND (q < 10) THEN
       BEGIN
          q := q + 1;
          EBADec[q] := Decision;
       END;
    END;

    END; (* DoEBA *)



PROCEDURE DoLexiSemi;

(* Lexicographic SemiOrder.  On most-important att find the absolute  *)
```

```
(* value of the difference between the 1st and next altern.  If the   *)
(* difference is greater than a given "just-noticeable-difference"    *)
(* value, elim the altern w/the smaller value.  If difference is less *)
(* than the JND, then compare the next altern value on that att vs    *)
(* the greater of the 2 survivors from the previous operation.        *)
(* Continue for all alterns on that att; then move to next most impt  *)
(* att and repeat.  When all but one altern have been eliminated,     *)
(* select that altern.  If all atts are considered and several alterns*)
(* remain, select at random.  ** ELIM modified for riskless att case; *)
(* riskless att cardinality reflects decrement of NContents by 2s.    *)

VAR Deciding: BOOLEAN;

BEGIN  (* DoLexiSemi *)

   Deciding    := TRUE;
   TimeUp      := FALSE;
   TIme[Rule] := 0.0; (* Reinitialize counter for this iteration. *)


WHILE ((Deciding) AND (TimeUp=FALSE)) DO

   (* Sets out-of-time condition in Timed Riskless SIM: *)
   IF Time[Rule] >= TimeLimit THEN
   BEGIN
      Deciding := FALSE;
      TimeUp   := TRUE;
   END

(* * * * * * * * *  At-End Processing:  * * * * * * * * * * * * * * * * *)

(* 1 *)
   ELSE IF Cardinality(Altern) = 1 THEN
   (* Only one altern left; some or all atts considered:              *)
   BEGIN
      Deciding  := FALSE;
      Decision  := CurrValue(Altern);
      LSORulCnt := LSORulCnt + 1;     (* Counter external to rule...   *)
      (* For debug; also external to rule:                            *)
      IF (WinsOut=TRUE) AND (p < 10) THEN
      BEGIN
         p := p + 1;
         LexDec[p] := CurrValue(Altern);
      END;
   END

(* 2 *)
   ELSE IF (Cardinality(Attrib)=0) THEN
   (*       Cardinality(Altern) > 1 implied                           *)
   (* Finished with all atts/columns, but there are still several     *)
   (* valid alterns/rows.                                             *)
   BEGIN
      Deciding  := FALSE;
      DoRand(Decision);
      LSORndCnt := LSORndCnt + 1;     (* Counter external to rule...   *)
      (* For debug; also external to rule:                            *)
      IF (WinsOut=TRUE) AND (p < 10) THEN
      BEGIN
         p := p + 1;
         LexDec[p] := Decision;
```

```
      END;
   END

(* * * * * * *  Move to Consider Another Att/Column * * * * * * * * * *)

(* 3A *)
   ELSE IF (Cardinality(Attrib) > 0) AND (AtEnd(Attrib))
           AND (NOT IsEmpty(6)) THEN
   (* Where an att has been considered and eliminated, and GetNext has*)
   (* cycled to the AtEnd marker for atts, but there are still valid  *)
   (* atts to consider.  Moves pointer to 1st att left in list.       *)

   (* 1st search for most-impt att is 1 to r search thru atts.  Loc 6 *)
   (* filled during compare set processing.  Subsequent searches for  *)
   (* most-impt att may start at the right of (an) eliminated att(s)  *)
   (* and may need to cycle back to start to check atts at left of the*)
   (* elim att(s).  STM Loc 6 is used both in processing & as a marker*)
   (* of when that cycle-back-to-start is used, to avoid endless loop.*)
   (* An empty Loc 6 means cycle-back has occurred once and need not  *)
   (* occur again before starting compare on this att.                *)
   BEGIN
      GetNext(Attrib);
      Forget(6);                        (* Empty 6 serves as marker.     *)
   END

(* 3B *)
   ELSE IF ( (Cardinality(Attrib) = 2)  AND (AtEnd(Altern)) ) THEN
   (* Elims last att and att pointer = 0; sends directly to random    *)
   (* decision in paragraph 2.  More efficient than 3C for this case. *)

   Elim(Attrib)

(* 3C *)
   ELSE IF (Cardinality(Attrib) > 2) AND (AtEnd(Altern)) THEN
   (*      Cardinality(Altern) > 1 implied                            *)
   (* Must eliminate current most-likely attrib and find the          *)
   (* next-most-likely.                                               *)
   BEGIN
      Elim(Attrib);                     (* Finished with this one.       *)
      GetNext(Altern);                  (* Get back to first avail alt.  *)
      Forget(1);                        (* and mark that a new search    *)
      Forget(2);
      Forget(3);                        (* for a new most-likely att can *)
      Forget(5);                        (* begin.                        *)
   END

(* * * * * * * * *  Get Most-Likely Attribute * * * * * * * * * * * * *)

(* 4 *)
   ELSE IF (Cardinality(Attrib) > 2) AND (IsEmpty(1)) AND
           (IsEmpty(3)) AND (IsEmpty(5)) THEN
   (* No compare yet in progress; no need to search if only one att   *)
   (* left.  Initial search for most-important attribute.             *)
   BEGIN
      GetWeight(1);
      GetMax(1,3,3);                    (* Current most-important in 3.  *)
      GetNext(Attrib);
   END

(* 5 *)
```

```
   ELSE IF (Cardinality(Attrib) > 2) AND (NOT AtEnd(Attrib))
           AND (NOT IsEmpty(1)) AND (IsEmpty(2)) AND (IsEmpty(5)) THEN
   (* Continue search for most-impt attrib, by testing challengers.   *)
   (* If only 1 attribute is left in the list; no need to search.     *)
   BEGIN
      GetWeight(2);                    (* Challenger.                  *)
      GetMax(2,3,3);
      Forget(2);                       (* Mark that more attributes can *)
                                       (* challenge, if available.     *)
      GetNext(Attrib);
   END

(* * * *  Begin Processing: ie,-Comparing Alt Values in This Att  * * *)

(* 6A *)
   ELSE IF (Cardinality(Attrib) > 2) AND (AtEnd(Attrib))
           AND (IsEmpty(2)) THEN
   (* All atts have challenged, and we have the most impt, so begin.  *)
   (* Positions most impt att when several have been considered.  Not *)
   (* for special case of only 1 att left; see paragraph 6B.          *)
   BEGIN
      GetJND(4);                       (* Just Noticeable Difference.  *)
      SetPoint(3);                     (* 1st avail altern and most-impt*)
                                       (* att to current pointers.     *)
      Forget(3);
      Remember(1);                     (* Get 2 altern in same att for  *)
      GetNext(Altern);                 (* comparison in STM Loc 1 and 2.*)
      Remember(2);
   END

(* 6B *)
   ELSE IF (Cardinality(Attrib) = 2) AND (IsEmpty(2)) THEN
   (* Special case of only 1 att left; commence processing.           *)
   BEGIN
      GetJND(4);
      Remember(1);
      GetNext(Altern);
      Remember(2);
   END

(* 7 *)
   ELSE IF (NOT IsEmpty(2)) AND (IsEmpty(5)) THEN
   (* Initial compare with results into STM Loc 3. Subsequent looping *)
   (* compares occur in paragraph 10.  Impt to identify the minimum   *)
   (* value (loser) so it can be eliminated in paragraph 8 if needed. *)
   BEGIN
      Subt(1,2,3);
      GetMax(1,2,5);
      GetMin(1,2,6);
   END

(* 8AA *)
   ELSE IF (Cardinality(Altern) > 2) AND (NOT IsEmpty(3))
           AND (IsEmpty(7))
           AND (ABS(STM[3].AValue) > STM[4].AValue) THEN
   (* Failed; difference of 2 values was > JND.                       *)
   BEGIN
      SetPoint(2);                     (* Find last challenger, then    *)
      GetNext(Altern);                 (* set up to get next, if any.   *)
      Remember(7);                     (* Mark that setup has occurred. *)
```

```
   END

(* 8AB *)
ELSE IF (Cardinality(Altern) > 2) AND (NOT IsEmpty(3))
        AND (ABS(STM[3].AValue) > STM[4].AValue)
        AND (NOT AtEnd(Altern)) THEN
     BEGIN
        Forget(3);
        Remember(2);       (* Since NOT at end, get this value in STM. *)
        SetPoint(6);
        Elim(Altern);      (* Get rid of loser.                        *)
        Forget(7);         (* Clear out marker.                        *)
     END

(* 8AC *)
ELSE IF (Cardinality(Altern) > 2) AND (NOT IsEmpty(3))
        AND (ABS(STM[3].AValue) > STM[4].AValue)
        AND (AtEnd(Altern)) AND (NOT IsEmpty(7)) THEN
     BEGIN
        SetPoint(6);
        Elim(Altern);
        Forget(7);
        SetPoint(6);        (* Reverse the setpoint operation.         *)
     END

(* 8B *)
   ELSE IF (Cardinality(Altern) = 2) AND (NOT IsEmpty(3))
           AND (ABS(STM[3].AValue) > STM[4].AValue)
           AND (NOT AtEnd(Altern)) THEN
   (* Failed as above, but a special case where elim the loser leaves *)
   (* only 1 alt left: the choice.  So don't set up for next compare. *)
   (* Next production to fire will be the decision selector.          *)
   (* NB- 3B and 3C take care AtEnd(Altern) cases; they either end    *)
   (* & pick at random, or go on to another attribute's alterns.      *)
   BEGIN
      SetPoint(6);
      Elim(Altern);
      SetPoint(5);                    (* Position to winning alt.     *)
   END

(* 9A *)
   ELSE IF (NOT IsEmpty(3))
           AND (ABS(STM[3].AValue) < STM[4].AValue)
           AND (IsEmpty(7)) THEN
   (* Passed; difference of 2 values was < JND.  Don't eliminate      *)
   (* either alternative.                                             *)
   BEGIN
      SetPoint(2);                    (* Return to place in list.     *)
      GetNext(Altern);                (* Another challenger?          *)
      Remember(7);                    (* Mark that we've done this 1/2.*)
   END

(* 9B *)
   ELSE IF (NOT IsEmpty(3))
           AND (ABS(STM[3].AValue) < STM[4].AValue)
           AND (NOT AtEnd(Altern)) AND (Not IsEmpty(7)) THEN
   BEGIN
      Forget(3);
      Remember(2);       (* Since not at end, get value set up in     *)
      Forget(7);         (* previous paragraph.  Clear marker in 7.   *)
```

```
   END

(* 10 *)
   ELSE IF (NOT AtEnd(Altern)) AND (NOT IsEmpty(1)) THEN
   (* Continue processing begun in earlier paragraphs (8, 9)...       *)
   BEGIN
      Subt(2,5,3);                    (* Get difference bet. the two.  *)
      GetMin(2,5,6);                  (* Get loser in case elim needed.*)
      GetMax(2,5,5);                  (* Statement order impt here.    *)

   END;  (* WHILE *)


   IF (TimeUp=TRUE) THEN
   BEGIN
      DoRand(Decision);
      LexLimit := LexLimit +1;

      LSORndCnt := LSORndCnt + 1;     (* Counter external to rule...   *)
      (* For debug; also external to rule:                            *)
      IF (WinsOut=TRUE) AND (p < 10) THEN
      BEGIN
         p := p + 1;
         LexDec[p] := Decision;
      END;
   END;

 END;  (* LexiSemi *)




PROCEDURE DoLexi;

(* Lexicographic Rule.  On most-important att compare the values of   *)
(* two alternatives and elim the altern w/the lesser value.  Use the  *)
(* altern with the larger, winning value as the basis for comparison  *)
(* with the next altern on that att.  Continue for all alterns on     *)
(* that att.  When all but one altern have been eliminated, return    *)
(* the remaining altern as the winner.  **ELIM modified for riskless  *)
(* att case; riskless att cardinality reflects decrement of NContents *)
(* by 2s instead of by 1s.  This means that to indicate more than 1   *)
(* att, for example, code reads: "cardinality(attrib)>2".             *)


VAR Deciding: BOOLEAN;

BEGIN  (* DoLexi *)

   Deciding    := TRUE;
   TimeUp      := FALSE;
   TIme[Rule] := 0.0; (* Reinitialize counter for this iteration. *)


WHILE ((Deciding) AND (TimeUp=FALSE)) DO

   (* Sets out-of-time condition in Timed Riskless SIM: *)
   IF Time[Rule] >= TimeLimit THEN
```

```
   BEGIN
      Deciding := FALSE;
      TimeUp   := TRUE;

(* * * * * * * * *  At-End Processing:  * * * * * * * * * * * * * * * * *)

(* 1 *)
   END ELSE IF Cardinality(Altern) = 1 THEN
   (* Only one altern left; all alterns on this att have been checked:*)
   BEGIN
      Deciding  := FALSE;
      Decision  := CurrValue(Altern);
      (* For debug; also external to rule:                              *)
      IF (WinsOut=TRUE) AND (s < 10) THEN
      BEGIN
         s := s + 1;
         LxDec[s] := CurrValue(Altern);
      END;
   END


(* * * * * * * * *  Get Most-Likely Attribute * * * * * * * * * * * * * *)

(* 2 *)
   ELSE IF (IsEmpty(1)) THEN
   (* No compare yet in progress; initial search for most-important   *)
   (* attribute.                                                      *)
   BEGIN
      GetWeight(1);          (* Weight of this attribute.             *)
      GetMax(1,3,3);         (* Greatest weight stored in STM Loc 3.  *)
      GetNext(Attrib);
   END

(* 3 *)
   ELSE IF (NOT AtEnd(Attrib)) AND (NOT IsEmpty(1)) AND
           (IsEmpty(2)) AND (IsEmpty(5)) THEN
   (* Continue search for most-impt attrib, by testing challengers.   *)
   (* If only 1 attribute is left in the list; no need to search.     *)
   BEGIN
      GetWeight(2);                 (* Challenger.                    *)
      GetMax(2,3,3);
      Forget(2);                    (* Mark that more attributes can *)
                                    (* challenge, if available.       *)
      GetNext(Attrib);
   END

(* * * *  Begin Processing: ie,-Comparing Alt Values in This Att  * * *)

(* 4 *)
   ELSE IF (AtEnd(Attrib)) AND (IsEmpty(2)) THEN
   (* All atts have challenged, and we have the most impt, so begin.  *)
   (* Positions most impt att when several have been considered.      *)
   BEGIN
      SetPoint(3);                  (* 1st avail altern and most-impt*)
                                    (* att set to current pointers.  *)
      Forget(3);
      Remember(1);                  (* Get 2 altern in same att for  *)
      GetNext(Altern);              (* comparison in STM Loc 1 and 2.*)
      Remember(2);
   END
```

```
(* 5 *)
   ELSE IF (Cardinality(Altern) > 2) AND (NOT IsEmpty(2)) AND
           (IsEmpty(5)) THEN
   (* Case of more than 2 alternatives still left to consider.        *)
   (* Elim the altern w/smaller value and set up for next compare,    *)
   (* since after elim there are still at least 2 altern. STM5=best.  *)
   BEGIN
      GetMax(1,2,5);
      GetMin(1,2,6);
      SetPoint(5);                    (* Put winner in STM Loc 1.      *)
      Remember(1);
      SetPoint(2);                    (* Return to last challenger, &  *)
      GetNext(Altern);                (* then get the next challenger. *)
      Remember(2);                    (* New challenger in STM Loc 2.  *)
      SetPoint(6);                    (* Get rid of losing altern.     *)
      Elim(Altern);
      Forget(5);                      (* Mark we're ready for the      *)
                                      (*   next comparison.            *)
   END

(* 6 *)
   ELSE IF (Cardinality(Altern) = 2) AND (NOT IsEmpty(2)) AND
           (IsEmpty(5)) THEN
   (* Case of only 2 alternatives to consider.  When 1 eliminated,    *)
   (* don't go through the steps to set up another comparison.        *)
   (* Next production to fire will be the decision selector.          *)
   BEGIN
      GetMax(1,2,5);
      GetMin(1,2,6);
      SetPoint(6);                    (* Delete losing alternative.    *)
      Elim(Altern);
      SetPoint(5);                    (* Position to winning alt.      *)

   END;  (* WHILE *)


   IF (TimeUp=TRUE) THEN
   BEGIN
      DoRand(Decision);
      LxLimit := LxLimit +1;
      (* For debug; also external to rule:                            *)
      IF (WinsOut=TRUE) AND (s < 10) THEN
      BEGIN
         s := s + 1;
         LxDec[s] := CurrValue(Altern);
      END;

   END;

 END;  (* DoLexi *)



PROCEDURE DoEqualWeight;

VAR Deciding: BOOLEAN;

BEGIN    (* DoEqualWeight *)
```

```
   Deciding   := TRUE;
   TimeUp     := FALSE;
   Time[Rule] := 0.0;  (* Reinit counter for this iteration. *)


   WHILE ((Deciding) AND (TimeUp=FALSE))  DO

   (* Sets out-of-time condition in Timed Riskless SIM: *)
   IF Time[Rule] >= TimeLimit THEN
   BEGIN
      Deciding := FALSE;
      TimeUp   := TRUE;


     END ELSE IF AtEnd(Altern) THEN
     BEGIN
        Deciding := FALSE;          (* Exhausted all alternatives.     *)
        Decision := STM[5].Alt;
        (* For debug; external to rule:                                *)
        IF (WinsOut=TRUE) AND (n < 10) THEN
        BEGIN
           n := n + 1;
           EqWDec[n] := STM[5].Alt;
        END;
     END

ELSE IF AtEnd(Attrib) THEN
     BEGIN                          (* End of this alt's atts.         *)
        GetMax(2,5,5);
        Forget(2);                  (* Initialize for new calculation  *)
        GetNext(Altern);            (*    and move on.                 *)
        GetNext(Attrib);
     END

ELSE IF IsEmpty(2) THEN
     BEGIN                          (* For 1st att in an altern; avoids*)
        Remember(2);                (*     an unnecessary addition.    *)
        GetNext(Attrib);
     END

ELSE BEGIN                          (* Get next payoff and sum.        *)
        Remember(1);                (* The att value.                  *)
        GetNext(Attrib);
        Add(1,2,2);

     END;  (* WHILE *)

     IF (TimeUp=TRUE) THEN
     BEGIN
        IF STM[5].Alt > 0 THEN        (* If a current best is avail *)
           Decision := STM[5].ALT     (* take it.                   *)
        ELSE
           DoRand(Decision);

           EqWLimit := EqWLimit + 1;

        (* For debug; external to rule:                                *)
        IF (WinsOut=TRUE) AND (n < 10) THEN
        BEGIN
```

```
            n := n + 1;
            EqWDec[n] := Decision;
          END;

      END;

END;  (* DoEqualWeight *)



PROCEDURE DoMCD;

(* Majority of Confirming Dimensions. In Timed Riskless SIM, if time  *)
(* limit reached and current best is available, it becomes the choice *)
(* else a choice is made at random.                                   *)
VAR Deciding: BOOLEAN;

BEGIN
   Deciding   := TRUE;
   TimeUp     := FALSE;
   Time[Rule] := 0.0; (* Reinit counter for this iteration. *)

   WHILE ((Deciding) AND (TimeUp=FALSE)) DO

   (* Sets out-of-time condition in Timed Riskless SIM: *)
   IF Time[Rule] >= TimeLimit THEN
   BEGIN
      Deciding := FALSE;
      TimeUp   := TRUE;
   END


(* 1 *)
 ELSE IF (Cardinality(Altern) = 1) THEN   (* Only best alt left. *)
      BEGIN
         Deciding := FALSE;
         Decision := CurrValue(Altern);
         (* For debug; external to rule:                          *)
         IF (WinsOut=TRUE) AND (m < 10) THEN
         BEGIN
            m := m + 1;
            MCDDec[m] := CurrValue(Altern);
         END;
      END

(* 2 *)
ELSE  IF (NOT AtEnd(Attrib)) AND (NOT AtEnd(Altern))
      AND (IsEmpty(3)) THEN
      BEGIN
         Remember(1);
         GetNext(Altern);
         Remember(2);
         GetMax(1,2,3);
       END

(* * * * * *  Register a vote for Winner  * * * * * * * * * * * * *)
(* The two alts are compared, att by att.  The winning value on ea  *)
(* compare either increments (for 1st alt) or decrements (for 2d    *)
(* alt) the neutral value in STM Loc 4.  If at end the value in 4   *)
```

```
(* is positive, then the 1st alt has won overall; if negative, the  *)
(* 2d alt has won; if zero, a tie.                                   *)

(* 3A *)
ELSE  IF (NOT IsEmpty(3)) AND (STM[1].AValue = STM[3].AValue) THEN
(* First alt wins; increment.                                        *)
      BEGIN
         Add1(4);          (* Add to STM Loc 4.                     *)
         Forget(3);        (* We can compare a new pair of alts.    *)
         GetNext(Attrib);
         GetPrev(Altern);
      END

(* 3B *)
ELSE  IF (NOT IsEmpty(3)) AND (STM[2].AValue = STM[3].AValue) THEN
(* Second alt wins; decrement.                                       *)
      BEGIN
         Subt1(4);         (* Subtract from STM Loc 4.              *)
         Forget(3);        (* We can compare a new pair of alts.    *)
         GetNext(Attrib);
         GetPrev(Altern);
      END


(*       Is Winner 1st Alt, 2d Alt or Tie; More or At End?          *)
(* All the productions below imply AtEnd(Attrib) is true.           *)
(* 4A *)
ELSE IF (Cardinality(Altern)>2) AND (STM[4].AValue > 0.0) THEN
     (* Winner is 1st alt considered; there are more alts left.     *)
     BEGIN
        SetPoint(2);
        Elim(Altern);      (* Mark off the loser in the list.       *)
        SetPoint(1);       (* Set to winner; att always last att.   *)
        GetNext(Attrib);   (* Sets to AtEnd mark for atts.          *)
        GetNext(Attrib);   (* Sets to first att.                    *)
        Forget(4);         (* Clear out for next compare of alts.   *)
     END

(* 4B *)
ELSE IF (Cardinality(Altern)=2) AND (STM[4].AValue > 0.0) THEN
     (* Winner is 1st alt considered; there are no alts left.       *)
     BEGIN
        SetPoint(2);
        Elim(Altern);      (* Leaves cardinality of altern at 1.    *)
        SetPoint(1);       (* Make 1st alt considered current.      *)
     END

(* 5A *)
ELSE IF (Cardinality(Altern) > 2) AND (STM[4].AValue < 0.0) THEN
     (* Winner is 2d alt considered; there are more alts left.      *)
     BEGIN
        SetPoint(1);
        Elim(Altern);
        SetPoint(2);       (* Sets to winner; att always last.      *)
        GetNext(Attrib);   (* Sets to AtEnd mark for atts.          *)
        GetNext(Attrib);   (* Sets to first att.                    *)
        Forget(4);
     END

(* 5B *)
```

```
ELSE IF (Cardinality(Altern) = 2) AND (STM[4].AValue < 0.0) THEN
     (* Winner is 2d alt considered; there are no alts left.          *)
     BEGIN
        SetPoint(1);
        Elim(Altern);
        SetPoint(2);       (* Make 2d alt considered the current alt.  *)
     END

(* 6A *)
ELSE IF (Cardinality(Altern) > 2) AND (STM[4].AValue = 0.0) THEN
     (* Tie; there are more alts left.                                 *)
     BEGIN
        GetMin(1,2,3);     (* Find the loser and mark it off on list. *)
        SetPoint(3);
        Elim(Altern);
        GetMax(1,2,3);     (* Find the winner and get a challenger.   *)
        SetPoint(3);       (* Sets to winner; att always last.        *)
        GetNext(Attrib);   (* Sets to AtEnd mark for atts.            *)
        GetNext(Attrib);   (* Sets to first att.                      *)
        Forget(4);
        MCDTieCnt:=MCDTieCnt + 1; (* Ctr external to rule...          *)
     END

(* 6B *)
ELSE IF (Cardinality(Altern) = 2) AND (STM[4].AValue = 0.0) THEN
     (* Tie; there are no alts left.                                   *)
     BEGIN
        GetMin(1,2,3);     (* Find the loser and mark it off on list. *)
        SetPoint(3);
        Elim(Altern);
        GetMax(1,2,3);     (* In case of a tie, overall winner is the *)
        SetPoint(3);       (*    winner of compare on last att.       *)
        MCDTieCnt:=MCDTieCnt+1; (* Ctr external to rule...            *)

     END;    (* WHILE *)


   IF (TimeUp=TRUE) THEN
   BEGIN
      IF STM[2].Alt <= 2 THEN      (* At least one complete comparison *)
            DoRand(Decision)       (* required for a designated current*)
      ELSE                         (* best; ie-STM[2] alt at least 3.  *)
         Decision := STM[1].Alt;

      MCDLimit := MCDLimit + 1;

      (* For debug; external to rule:                                  *)
      IF (WinsOut=TRUE) AND (m < 10) THEN
      BEGIN
         m := m + 1;
         MCDDec[m] := Decision;
      END;

   END;

   END;  (* DoMCD *)
```

```
PROCEDURE DoEBAEU;

VAR Deciding: BOOLEAN;

BEGIN  (* DoEBAEU *)

 Deciding   := TRUE;
 TimeUp     := FALSE;
 Time[Rule] := 0.0; (* Reinit for this iteration. *)

 WHILE ((Deciding) AND (TimeUp=FALSE)) DO

 (* Sets out-of-time condition in Timed Riskless SIM: *)
 IF Time[Rule] >= TimeLimit THEN
 BEGIN
    Deciding := FALSE;
    TimeUp   := TRUE;
 END

    ELSE IF (Cardinality(Altern) = 1) AND (NOT AtEnd(Altern)) THEN
    BEGIN
       Deciding := FALSE;   (* Only 1 alt left so take it.          *)
       Decision := CurrValue(Altern);
       (* For debug; external to rule: *)
       IF (WinsOut = TRUE) AND (l < 10) THEN
       BEGIN
          l := l + 1;
          EEUDec[l] := Decision;
       END;
    END

    ELSE IF (Cardinality(Attrib) < 1) THEN
  (* We checked all attributes and more than one alternative is left, *)
  (*    so choose at random.                                          *)
    BEGIN
       DoRand(Decision);
       Deciding:= FALSE;
       (* For debug; external to rule:                               *)
       IF (WinsOut = TRUE) AND (l < 10) THEN
       BEGIN
          l := l + 1;
          EEUDec[l] := Decision;
       END;

(*****************************************************************)
(* Shift to EU rule if:                                          *)
(*   a) All processing on an att is finished (ie, IsEmpty(2)=T), and*)
(*   b) There is > 1 alt left (ie-last att processing didn't elim  *)
(*      down to one--if it did, that one=decision), and            *)
(*   c) There are < 4 alts left.                                   *)
(*****************************************************************)

    END ELSE IF (IsEmpty(2)) AND (Cardinality(Altern)<4) THEN
    BEGIN  (* Timed AddUtility code *)

       (* Main loop for Timed AU code:    ***************************)
       WHILE ((Deciding) AND (TimeUp=FALSE)) DO

       (* Sets out-of-time condition in Timed Riskless SIM: *)
       IF Time[Rule] >= TimeLimit THEN
```

```
      BEGIN

         Deciding := FALSE;
         TimeUp   := TRUE;
      END

         (* 1 *)
    ELSE IF AtEnd(Altern) THEN
         BEGIN
            Deciding:= FALSE;     (* Exhausted all alternatives.*)
            Decision:= STM[5].Alt;
            (* For debug; external to rule:                          *)
            IF (WinsOut=TRUE) AND (l < 10) THEN  (* Put winner in array    *)
               BEGIN                             (* and output to TraceOut *)
                  l := l + 1;                    (* file.                  *)
                  EEUDec[l] := Decision;
               END;


         (* 2 *)
     END ELSE IF AtEnd(Attrib) THEN  (* End of this Alt's Att's     *)
         BEGIN
            GetMax(4,5,5);       (* Get largest value.              *)

             Forget(4);          (* Init for new value calculation,  *)

             GetNext(Altern);    (*    and move on.                  *)
             GetNext(Attrib);    (* Sets att to 1.  Att always at the*)
                                 (* at-end value of -1 here.         *)

         (* 3 *)
     END ELSE IF (IsEmpty(4)) THEN (* First attribute case.          *)
         BEGIN
            Remember(1);
            GetWeight(2);
            GetNext(Attrib);
            Mult(1,2,4);          (* Put result directly into accum.  *)
         END

         (* 4 *)
     ELSE          (* Else add in the weighted value of this att *)
         BEGIN
            Remember(1);         (* The attribute value.            *)
            GetWeight(2);        (* The attribute weight.           *)
            GetNext(Attrib);     (* Riskless GetNext skips weight.  *)
            Mult(1,2,3);         (* The total weight.               *)
            Add(3,4,4);          (* Add this weight in.             *)

         END;    (* TimedAU Code Main WHILE Loop Ends ***************)

(******** AU Processing if Out of Time Condition Reached: ************)
        IF (TimeUp=TRUE) THEN
        BEGIN
           TimeUp := FALSE;  (* Set to F as a switch. *)
           IF STM[5].Alt > 0 THEN
              Decision := STM[5].Alt
           ELSE
              DoRand(Decision);

           EEULimit := EEULimit + 1;
```

```
        (* For Debug, external to rule: *)
        IF (WinsOut=TRUE) AND (l<10) THEN
        BEGIN
           l := l + 1;
           EEUDec[l] := Decision;
        END;

        Deciding := FALSE;
     END; (* IF TimeUp *)

(******************* END of Timed AU Code ****************************)


   END ELSE IF (AtEnd(Altern)) THEN
   (* We have eliminated all we can with this attribute.             *)
   BEGIN
      Elim(Attrib);
      SetPoint(5);                  (* Go back to start.             *)
      Forget(6);                    (* Marks that we need a new most *)
                                    (* important attribute.          *)
      Forget(5);
      Forget(2);                    (* Marks need for new cutoff.    *)

   END ELSE IF (AtEnd(Attrib) AND NOT IsEmpty(6)) THEN
   (* We have the most important attribute, so start processing:     *)
   BEGIN
      SetPoint(6);
      Remember(1);
      GetCut(1,2);
      GetMin(1,2,3);                 (* Get min of cutoff or value.  *)

   END ELSE IF NOT IsEmpty(3) AND (STM[3].Alt<>CutOffVal) THEN
   (* The current attribute did not meet the cutoff; it was the min. *)
   BEGIN
      Elim(Altern);
      Forget(3);

   END ELSE IF NOT IsEmpty(3) AND (STM[3].Alt = CutOffVal) THEN
   (* The current attribute met the cutoff; cutoff was the min.      *)
   BEGIN
      GetNext(Altern);
      Forget(3);

   END ELSE IF (IsEmpty(2) AND IsEmpty(4) AND IsEmpty(5)) THEN
   (* Initialize search for the most important attribute.            *)
   BEGIN
      GetWeight(5);
      GetNext(Attrib);              (* Riskless GetNext skips wgts.  *)
      GetMax(5,5,6);


   END ELSE IF (IsEmpty (2) AND IsEmpty(4) AND NOT IsEmpty(6) ) THEN
   (* Continue search for the most important attribute.              *)
   BEGIN
      GetWeight(4);
      GetMax(4,6,6);                (* Current winner.               *)
      Forget(4);                    (* Mark that we still need one.  *)
      GetNext(Attrib);              (* Riskless GetNext skips wgts.  *)
```

```
    END ELSE
    (* Check this attribute against the cutoff.                          *)
    BEGIN
       Remember(1);
       GetMin(1,2,3);                (* Min of cutoff and value.        *)

    END;    (* Main, Outer WHILE Loop ***********************************)

    (* Outer loop processing if time up condition entered:    ***********)
    IF (TimeUp=TRUE) AND (Deciding=FALSE) THEN  (* IE- enter to end     *)
    BEGIN                                       (* processing only if   *)
       DoRand(Decision);                        (* MCD code hasn't been*)
       EEULimit := EEULimit + 1;                (* used.                *)

       (* For debug; external to rule:                                   *)
       IF (WinsOut=TRUE) AND (1 < 10) THEN  (* Put winner in array     *)
          BEGIN                             (* and output to TraceOut *)
             1 := 1 + 1;                    (* file.                  *)
             EEUDec[1] := Decision;
          END;
    END;


    END; (* DoEBAEU *)



PROCEDURE DoEBAMCD;

VAR Deciding: BOOLEAN;


BEGIN  (* DoEBAMCD *)

 Deciding   := TRUE;
 TimeUp     := FALSE;
 Time[Rule] := 0.0; (* Reinit counter for this iteration. *)

 WHILE ((Deciding) AND (TimeUp=FALSE)) DO

 (* Sets out-of-time condition in Timed Riskless SIM: *)

 IF Time[Rule] >= TimeLimit THEN
 BEGIN

    Deciding := FALSE;
    TimeUp   := TRUE;
 END

 (* 1 *)
 (* A col of atts is processed together...the IsEmpty(2) marker *)
 (* isn't true until the whole column has been processed.  If   *)
 (* only 1 alt is left in a col, it's taken as the decision.    *)

 ELSE IF (Cardinality(Altern)=1) AND (NOT AtEnd(Altern)) THEN
 BEGIN
    Deciding := FALSE;      (* Only 1 alt left in this column.   *)
    Decision := CurrValue(Altern);
    (* For debug, external to rule: *)
```

```
    IF (WinsOut=TRUE) AND (k < 10) THEN
    BEGIN
       k := k + 1;
       ECDDec[k] := Decision;
    END;
  END

(* 2 *)
    ELSE IF (Cardinality(Attrib) < 1) THEN
  (* We checked all attributes and more than one alternative is left, *)
  (*    so choose at random.                                           *)
    BEGIN
       DoRand(Decision);
       Deciding:= FALSE;
       (* For debug; external to rule:                                *)
       IF (WinsOut = TRUE) AND (k < 10) THEN
       BEGIN
          k := k + 1;
          ECDDec[k] := Decision;
       END;

(*****************************************************************)
(* Begin MCD Rule when:                                          *)
(*  a) Processing complete on all alts of current most-likely att.*)
(*     Ie- IsEmpty(2) is true = marker that col is finished.     *)
(*  b) Three or fewer alts are left in matrix.                    *)
(* NB- If alts in matrix are eliminated down to one during evaluation*)
(* of one att, then processing ends via EBA rule, w/that last alt.*)
(*****************************************************************)

(* 3 *)
    END ELSE IF (IsEmpty(2)) AND (Cardinality(Altern)<4) THEN

    BEGIN   (* MCD Rule Processing Main Loop:          ***************)


(* MCD While Loop *)
       WHILE ((Deciding) AND (TimeUp=FALSE)) DO

       (* Sets out-of-time condition in Timed Riskless SIM: *)

       IF Time[Rule] >= TimeLimit THEN
       BEGIN


          Deciding := FALSE;    (* Directs processing out of main MCD *)
          TimeUp   := TRUE;     (* WHILE loop to Time Up paragraph.   *)
       END

(* 1 MCD *)
 ELSE IF (Cardinality(Altern) = 1) THEN     (* Only best alt left. *)
      BEGIN
         Deciding := FALSE;
         Decision := CurrValue(Altern);
       (* For debug; external to rule:                               *)
       IF (WinsOut=TRUE) AND (k < 10) THEN  (* Put winner in array &  *)
       BEGIN                                (* output to TraceOut     *)
          k := k + 1;                       (* file.                  *)
          ECDDec[k] := CurrValue(Altern);
       END;
```

```
      END

(* 2 *)
ELSE  IF (NOT AtEnd(Attrib)) AND (NOT AtEnd(Altern))
      AND (IsEmpty(3)) THEN
      BEGIN
         Remember(1);
         GetNext(Altern);
         Remember(2);
         GetMax(1,2,3);
       END

(* 3A *)
ELSE  IF (NOT IsEmpty(3)) AND (STM[1].AValue = STM[3].AValue) THEN
(* First alt wins; increment.                                          *)
      BEGIN
         Add1(4);           (* Add to STM Loc 4.                       *)
         Forget(3);         (* We can compare a new pair of alts.      *)
         GetNext(Attrib);
         GetPrev(Altern);
      END

(* 3B *)
ELSE  IF (NOT IsEmpty(3)) AND (STM[2].AValue = STM[3].AValue) THEN
(* Second alt wins; decrement.                                         *)
      BEGIN
         Subt1(4);          (* Subtract from STM Loc 4.                *)
         Forget(3);         (* We can compare a new pair of alts.      *)
         GetNext(Attrib);
         GetPrev(Altern);
      END

(* 4A *)
ELSE IF (Cardinality(Altern)>2) AND (STM[4].AValue > 0.0) THEN
     (* Winner is 1st alt considered; there are more alts left.       *)
     BEGIN
        SetPoint(2);
        Elim(Altern);       (* Mark off the loser in the list.         *)
        SetPoint(1);        (* Set to winner; att always last att.     *)
        GetNext(Attrib);    (* Sets to AtEnd mark for atts.            *)
        GetNext(Attrib);    (* Sets to first att.                      *)
        Forget(4);          (* Clear out for next compare of alts.     *)
     END

(* 4B *)
ELSE IF (Cardinality(Altern)=2) AND (STM[4].AValue > 0.0) THEN
     (* Winner is 1st alt considered; there are no alts left.         *)
     BEGIN
        SetPoint(2);
        Elim(Altern);       (* Leaves cardinality of altern at 1.      *)
        SetPoint(1);        (* Make 1st alt considered current.        *)
     END

(* 5A *)
ELSE IF (Cardinality(Altern) > 2) AND (STM[4].AValue < 0.0) THEN
     (* Winner is 2d alt considered; there are more alts left.        *)
     BEGIN
        SetPoint(1);
        Elim(Altern);
        SetPoint(2);        (* Sets to winner; att always last.        *)
```

```
        GetNext(Attrib);    (* Sets to AtEnd mark for atts.            *)
        GetNext(Attrib);    (* Sets to first att.                      *)
        Forget(4);
     END

(* 5B *)
ELSE IF (Cardinality(Altern) = 2) AND (STM[4].AValue < 0.0) THEN
     (* Winner is 2d alt considered; there are no alts left.          *)
     BEGIN
        SetPoint(1);
        Elim(Altern);
        SetPoint(2);       (* Make 2d alt considered the current alt.  *)
     END

(* 6A *)
ELSE IF (Cardinality(Altern) > 2) AND (STM[4].AValue = 0.0) THEN
     (* Tie; there are more alts left.                                *)
     BEGIN
        GetMin(1,2,3);      (* Find the loser and mark it off on list. *)
        SetPoint(3);
        Elim(Altern);
        GetMax(1,2,3);      (* Find the winner and get a challenger.   *)
        SetPoint(3);        (* Sets to winner; att always last.        *)
        GetNext(Attrib);    (* Sets to AtEnd mark for atts.            *)
        GetNext(Attrib);    (* Sets to first att.                      *)
        Forget(4);
     END

(* 6B *)
ELSE IF (Cardinality(Altern) = 2) AND (STM[4].AValue = 0.0) THEN
     (* Tie; there are no alts left.                                  *)
     BEGIN
        GetMin(1,2,3);      (* Find the loser and mark it off on list. *)
        SetPoint(3);
        Elim(Altern);
        GetMax(1,2,3);      (* In case of a tie, overall winner is the *)
        SetPoint(3);        (*    winner of compare on last att.       *)

END;    (* END WHILE. MCD code main loop. ****************************)

(************** Time up processing for MCD code:  ********************)
   IF (TimeUp=TRUE) THEN
   BEGIN
      TimeUp := FALSE; (* Switch. Outer WHILE MCD loop over, and DECI-*)
         (* DING=FALSE. Setting TimeUp to FALSE here prevents us from *)
         (* entering the last, outer IF TIMEUP=TRUE EBA paragraph at  *)
         (* end. Set false as a convenient switch only.               *)
      IF STM[2].Alt <= 2 THEN     (* At least one complete comparison *)
            DoRand(Decision)      (* required for a designated current*)
      ELSE                        (* best; ie-STM[2] alt at least 3.  *)
         Decision := STM[1].Alt;

      ECDLimit := ECDLimit + 1;

      (* For debug; external to rule:                                 *)
      IF (WinsOut=TRUE) AND (k < 10) THEN
      BEGIN
         k := k + 1;
         ECDDec[k] := Decision;
      END;
```

```
   END; (*  MCD Code Processing ********************************)

(* 4 *)

    END ELSE IF (AtEnd(Altern) ) THEN
    (* We have eliminated all we can with this attribute.            *)
    BEGIN

       Elim(Attrib);
       SetPoint(5);                    (* Go back to start.          *)
       Forget(6);                      (* Marks that we need a new most *)
                                       (* important attribute.       *)
       Forget(5);
       Forget(2);                      (* Marks need for new cutoff. *)

(* 5 *)

    END ELSE IF (AtEnd(Attrib) AND NOT IsEmpty(6)) THEN
    (* We have the most important attribute, so start processing:    *)
    BEGIN
       SetPoint(6);
       Remember(1);
       GetCut(1,2);
       GetMin(1,2,3);                   (* Get min of cutoff or value. *)

(* 6 *)
    END ELSE IF NOT IsEmpty(3) AND (STM[3].Alt<>CutOffVal) THEN
    (* The current attribute did not meet the cutoff; it was the min. *)
    BEGIN
       Elim(Altern);
       Forget(3);

(* 7 *)
    END ELSE IF NOT IsEmpty(3) AND (STM[3].Alt = CutOffVal) THEN
    (* The current attribute met the cutoff; cutoff was the min.     *)
    BEGIN
       GetNext(Altern);
       Forget(3);

(* 8 *)
    END ELSE IF (IsEmpty(2) AND IsEmpty(4) AND IsEmpty(5)) THEN
    (* Initialize search for the most important attribute.           *)
    BEGIN
       GetWeight(5);
       GetNext(Attrib);              (* Riskless GetNext skips wgts. *)
       GetMax(5,5,6);


(* 9 *)
    END ELSE IF (IsEmpty (2) AND IsEmpty(4) AND NOT IsEmpty(6) ) THEN
    (* Continue search for the most important attribute.             *)
    BEGIN

       GetWeight(4);
       GetMax(4,6,6);                (* Current winner.              *)
       Forget(4);                    (* Mark that we still need one. *)
       GetNext(Attrib);              (* Riskless GetNext skips wgts. *)

(* 10 *)
```

```
    END ELSE
    (* Check this attribute against the cutoff.                          *)
    BEGIN
       Remember(1);
       GetMin(1,2,3);                (* Min of cutoff and value.        *)

    END;  (* WHILE; main outer loop ends here. ************************)

(* Timed code if rule ends while in the EBA portion...see alternative *)
(* ending above if rule ends while in the MCD portion. If it ends in  *)
(* MCD it should never enter the code below because MCD sets TimeUp to*)
(* FALSE as a switch, which should cause exit of ALL processing.      *)
    IF (TimeUp=TRUE) AND (Deciding=FALSE) THEN
    BEGIN
       Deciding := FALSE;
       DoRand(Decision);
       ECDLimit := ECDLimit + 1;

       (* For debug; external to rule:                                  *)
       IF (WinsOut=TRUE) AND (k < 10) THEN  (* Put winner in array &  *)
       BEGIN                                (* output to TraceOut     *)
          k := k + 1;                       (* file.                  *)
          ECDDec[k] := Decision;
       END;
    END;


    END; (* DoEBAMCD *)



PROCEDURE DoAddUtility;
(* NOT TIMED... run as baseline.  See rule called TimedAU for the     *)
(* timed version of this rule.                                        *)

    VAR Deciding: BOOLEAN;

    BEGIN (*DoAddUtility*)

       Deciding   := TRUE;

       WHILE (Deciding) DO

          (* 1 *)
     IF AtEnd(Altern) THEN
          BEGIN
             Deciding:= FALSE;    (* Exhausted all alternatives.*)
             Decision:= STM[5].Alt;


          (* 2 *)
      END ELSE IF AtEnd(Attrib) THEN  (* End of this Alt's Att's     *)
          BEGIN
             GetMax(4,5,5);       (* Get largest value.              *)
              Forget(4);          (* Init for new value calculation,  *)
              GetNext(Altern);    (*    and move on.                  *)
              GetNext(Attrib);    (* Sets att to 1.  Att always at the*)
                                  (* at-end value of -1 here.         *)

          (* 3 *)
```

```
    END ELSE IF (IsEmpty(4)) THEN (* First attribute case.          *)
        BEGIN
           Remember(1);
           GetWeight(2);
           GetNext(Attrib);
           Mult(1,2,4);          (* Put result directly into accum.  *)
        END

        (* 4 *)
     ELSE          (* Else add in the weighted value of this att *)
        BEGIN
           Remember(1);          (* The attribute value.             *)
           GetWeight(2);         (* The attribute weight.            *)
           GetNext(Attrib);      (* Riskless GetNext skips weight.   *)
           Mult(1,2,3);          (* The total weight.                *)
           Add(3,4,4);           (* Add this weight in.              *)

        END;    (* WHILE *)



    END;  (* DoAddUtility *)


PROCEDURE DoTimedAU;
(* This is the TIMED version of the above rule.                      *)
(* For Timed Riskless SIM, if time limit reached and a current best  *)
(* is available, take it as the decision, else choose at random.     *)

    VAR Deciding: BOOLEAN;

    BEGIN (* DoTimedAU *)

       Deciding    := TRUE;
       TimeUp      := FALSE;
       Time[Rule] := 0.0; (* Reinit counter for this iteration. *)


       WHILE ((Deciding) AND (TimeUp=FALSE)) DO

       (* Sets out-of-time condition in Timed Riskless SIM: *)
       IF Time[Rule] >= TimeLimit THEN
       BEGIN

          Deciding := FALSE;
          TimeUp   := TRUE;
       END

          (* 1 *)
      ELSE IF AtEnd(Altern) THEN
          BEGIN
             Deciding:= FALSE;     (* Exhausted all alternatives.*)
             Decision:= STM[5].Alt;
             (* For debug; external to rule:                         *)
             IF (WinsOut=TRUE) AND (r < 10) THEN
             BEGIN
                r := r+1;
                AdUDec[r] := STM[5].Alt;
             END;
```

```
        (* 2 *)
    END ELSE IF AtEnd(Attrib) THEN  (* End of this Alt's Att's     *)
        BEGIN
           GetMax(4,5,5);       (* Get largest value.              *)
            Forget(4);          (* Init for new value calculation, *)
            GetNext(Altern);    (*    and move on.                 *)
            GetNext(Attrib);    (* Sets att to 1.  Att always at the*)
                                (* at-end value of -1 here.        *)

        (* 3 *)
    END ELSE IF (IsEmpty(4)) THEN (* First attribute case.         *)
        BEGIN
           Remember(1);
           GetWeight(2);
           GetNext(Attrib);
           Mult(1,2,4);          (* Put result directly into accum.  *)
        END

        (* 4 *)
    ELSE          (* Else add in the weighted value of this att *)
        BEGIN
           Remember(1);         (* The attribute value.            *)
           GetWeight(2);        (* The attribute weight.           *)
           GetNext(Attrib);     (* Riskless GetNext skips weight.  *)
           Mult(1,2,3);         (* The total weight.               *)
           Add(3,4,4);          (* Add this weight in.             *)

        END;    (* WHILE *)

     IF (TimeUp=TRUE) THEN
     BEGIN
        IF STM[5].Alt > 0 THEN       (* If a current best is avail,  *)
           Decision := STM[5].Alt    (* take it.                     *)
        ELSE
           DoRand(Decision);

        AdULimit := AdULimit +1;

        IF (WinsOut=TRUE) AND (r < 10) THEN
        BEGIN
           r := r+1;
           AdUDec[r] := Decision;
        END;

     END;

  END;  (* DoTimedAU *)



PROCEDURE DoSatisficing;
(* Starts with the first alternative and looks at its first attri-  *)
(* bute.  If the att is >= CutOffValue, the next att in the alt is  *)
(* considered.  If all atts in the alt meet the test, that alt is   *)
(* chosen.  If one of the atts is < CutOffValue, no other atts in   *)
(* that alt are considered and the first att in the next alt is     *)
(* tested.  If no alt has atts that all meet the CutOffValue, an    *)
(* alt is chosen at random.  Production order important for rule    *)
(* function.                                                        *)
```

```
    VAR   Deciding: BOOLEAN;

    BEGIN (* DoSatisficing *)

    Deciding   :=TRUE;
    TimeUp     := FALSE;
    Time[Rule] := 0.0; (* Reinit counter for this iteration. *)


        WHILE ((Deciding) AND (TimeUp=FALSE)) Do

        (* Sets out-of-time condition for Timed Riskless SIM: *)
        IF Time[Rule] >= TimeLimit THEN
        BEGIN
           Deciding := FALSE;
           TimeUp   := TRUE;
        END

(* 1 *)
        ELSE IF AtEnd(Altern) THEN
   (* All alts have been considered; none have atts that all meet     *)
   (* CutOffValue:                                                    *)
        BEGIN
           DoRand(Decision);        (* Pick an alt choice randomly.    *)
           Deciding := FALSE;
           SatRndCnt:= SatRndCnt+1;(* Counter external to rule...     *)
           (* For debug; also external to rule:                       *)
           IF (WinsOut = TRUE) AND (j < 10) THEN
           BEGIN
              j := j+1;
              SatDec[j] := CurrValue(Altern);
           END;
        END  (* IF *)

(* 2 *)
   ELSE IF (AtEnd(Attrib)) AND (STM[2].AValue = STM[3].AValue) THEN
   (* Alt failed on last att; special last-att case.                  *)
        BEGIN
           GetNext(Altern);
           GetNext(Attrib);         (* Sets to first att in list.      *)
           Forget(1);
        END

(* 3 *)
   ELSE IF (NOT AtEnd(Attrib)) AND (NOT IsEmpty(1))
           AND (STM[2].AValue = STM[3].AValue) THEN
   (* Alt failed before last att; cycle back to AtEnd of att list.    *)
   (* After cycle ends, following production fires.                   *)
        GetPrev(Attrib)             (* Loops until AtEnd(Attrib).      *)


(* 4 *)
   ELSE IF (AtEnd(Attrib)) AND (IsEmpty(1)) THEN
        BEGIN
           Deciding := FALSE;
           Decision := CurrValue(Altern);
           SatRulCnt:= SatRulCnt+1;(* Counter external to rule...     *)
           (* For debug; also external to rule:                       *)
           IF (WinsOut = TRUE) AND (j < 10) THEN
```

```
            BEGIN                                   (* Put winner in array  *)
               j := j + 1;                          (* and output to Trace- *)
               SatDec[j] := CurrValue(Altern); (* Out file.                 *)
            END;
         END

(* 5 *)
   ELSE IF (AtEnd(Attrib)) AND (STM[1].AValue = STM[3].AValue) THEN
   (* Implied: NOT IsEmpty(1);                                              *)
   (* Alt passed on last att; special last-att case.                        *)
        Forget(1)                    (* The next production to fire will*)
                                     (*    be the decision.                *)

(* 6 *)
   ELSE IF (IsEmpty(1)) THEN
   (* Implied: NOT AtEnd                                                    *)
   (* See if an att passes the cutoff.                                      *)
        BEGIN
           Remember(1);
           GetCut(2,2);
           GetMax(1,2,3);
           GetNext(Attrib);
        END

(* 7 *)
   ELSE IF STM[1].AValue = STM[3].AValue THEN
   (* Implied: NOT IsEmpty(1) and NOT AtEnd(Attrib).                        *)
   (* Alt passed before last att; get the next to test.                     *)
        BEGIN
           Forget(1);

        END;  (* WHILE *)


    IF (TimeUp=TRUE) THEN
    BEGIN
       DoRand(Decision);
       SatLimit := SatLimit + 1;

       SatRndCnt:= SatRndCnt+1;(* Counter external to rule...     *)
       (* For debug; also external to rule:                      *)
       IF (WinsOut = TRUE) AND (j < 10) THEN
       BEGIN
          j := j+1;
          SatDec[j] := CurrValue(Altern);
       END;
    END;


    END; (* DoSatisficing *)



(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*  Run the rules.                                                         *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

PROCEDURE DoARule;

    VAR AltNum,AttNum: INTEGER;
```

```
        L: MemLoc;
        O:OperResults;
        Spread: REAL;

    BEGIN  (* DoARule *)
        (* initialize alternative list *)
{       clrmsguu(24,40,40);
        errmsguu(24,40,0,15,'Initializing Lists...');)
        InitList(Altern);

        FOR AltNum:= 1 TO NAlternatives DO BEGIN
            PutValue(Altern,Altnum);

            (* Get normative choice *)
            IF ChoiceRanking[AltNum] = 1 THEN AltAU := AltNum;

            (* Get normative choice for utility *)
            IF ChoiceURank[AltNum] = 1 THEN UAltAU := AltNum;
        END;



        CloseListAdditions(Altern);

        (* Initialize attribute list *)
        InitList(Attrib);
        FOR AttNum:= 1 TO NAttributes DO BEGIN
            PutValue(Attrib,AttNum);
            PutValue(Attrib,(AttNum + NAttributes));
        END; (* FOR *)

        CloseListAdditions(Attrib);


        (* Clear out memory. *)
        FOR L:= 1 TO NMemCell DO
        WITH STM[L] DO
        BEGIN
        OpType := Empty;
        Alt := 0;
        Att := 0;
        AValue := 0.0;
        END; {WITH}



        (* Clear out the operation counts *)
        FOR O:= Empty TO AttAbsence DO
        OperationCount[O]:= 0;

        cursetvv(24,40);
        write(output,'                                                ');
        cursetvv(24,40);
        write(output,'Performing: ',rule);



        CASE rule OF
            Rand:
                BEGIN
```

```
                DoRand(AltNum);
                AltRand := AltNum; (* Keeps rank of random choice as
                                      baseline for comparison *)
            END;

        EBA:
            DoEBA(AltNum,EBARulCnt,EBARndCnt,EBALimit);

        LexiSemi:
            DoLexiSemi(AltNum,LSORulCnt,LSORndCnt,LexLimit);

        Lexi:
            DoLexi(AltNum,LxLimit);

        EqualWeight:
            DoEqualWeight(AltNum,EqWLimit);

        MCD:
            DoMCD(AltNum,MCDTieCnt,MCDLimit);

        EBAEU:
            DoEBAEU(AltNum,EEULimit);

        EBAMCD:
            DoEBAMCD(AltNum,ECDLimit);

        AddUtility:
            DoAddUtility(AltNum);

        TimedAU:
            DoTimedAU(AltNum,AdULimit);

        Satisficing:
            DoSatisficing(AltNum,SatRulCnt,SatRndCnt,SatLimit);

    END; (* CASE *)

    cursetvv(24,40);
    write(output,'                                          ');
    cursetvv(24,40);
    write(output,'Recording Results...');

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* Check to see if this iteration of the rule picked the alternative *)
(* considered to be the best, and if so, increment a counter which   *)
(* will be used in Sim.Pas to calc the proportion of best choices.   *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
     IF AltNum = AltAu THEN
        BestChoice[Rule] := BestChoice[Rule] + 1;


    MeanData[Rule].DomValues := MeanData[Rule].DomValues+DomValues[AltNum];
    MeanData[Rule].DomExist := MeanData[Rule].DomExist + DomExist;
    MeanData[Rule].ChoiceValues := MeanData[Rule].ChoiceValues +
            ChoiceValues[AltNum];
    MeanData[Rule].ChoiceRanking := MeanData[Rule].ChoiceRanking +
            ChoiceRanking[AltNum];
    MeanData[Rule].ChoiceUtils := MeanData[Rule].ChoiceUtils +
            ChoiceUtils[AltNum];
```

```
        MeanData[Rule].ChoiceURank := MeanData[Rule].ChoiceURank +
                ChoiceURank[AltNum];

        FOR O:= Empty TO AttAbsence DO
            MeanData[Rule].OperationCount[O] :=
                MeanData[Rule].OperationCount[O]
                +OperationCount[O];


        StdData[Rule].DomValues := StdData[Rule].DomValues +
                SQR (DomValues[AltNum]);
        StdData[Rule].DomExist := StdData[Rule].DomExist +
                SQR(DomExist);
        StdData[Rule].ChoiceValues := StdData[Rule].ChoiceValues +
                SQR(ChoiceValues[AltNum]);
        StdData[Rule].ChoiceRanking := StdData[Rule].ChoiceRanking +
                SQR(ChoiceRanking[AltNum]);
        StdData[Rule].ChoiceUtils := StdData[Rule].ChoiceUtils +
                SQR(ChoiceUtils[AltNum]);
        StdData[Rule].ChoiceURank := StdData[Rule].ChoiceURank +
                SQR(ChoiceURank[AltNum]);

        FOR O:= Empty TO AttAbsence DO
            StdData[Rule].OperationCount[O] :=
            StdData[Rule].OperationCount[O] +
                SQR(OperationCount[O]);

        Used[Rule] := TRUE {Remember that we used this one}

    END; (* DoARule *)
{$list+}
END. {Unit}
```

```
{Interface for FFORMSUU, Fast Forms Utility.  These are a series of }
{general purpose routines to assist in screen handling.             }

{Version 2.0 (C)Copyright Blaise Computing, Inc.  1982              }

 interface; unit fformsuu(alarmuu,abortuu,errmsguu,clrmsguu,inkeyuu,
                          pauseuu);

 procedure alarmuu;
 procedure abortuu (const msg : lstring);
 procedure errmsguu(row,col,fore,back : byte; const msg : lstring);
 procedure clrmsguu(row,col,len : byte);
 function  inkeyuu (var ch : char) : boolean;
 procedure pauseuu (const msg : lstring);

 begin
 end;
```

```
INTERFACE;
UNIT LSTPRIMS(MostFix,AtEnd,Cardinality,CurrValue,GetNext,GetPrev,
                SetPoint,Elim,InitList,PutValue,CloseListAdditions);
USES SIM;
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* Primitive routines for handling the attribute and alternative lists     *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

FUNCTION MostFix(List: MemListType): BOOLEAN [PURE];

FUNCTION AtEnd(List: MemListType): BOOLEAN [PURE];
(* Is the List at the end marker? *)

FUNCTION Cardinality(List: MemListType): INTEGER [PURE];
(* Does this list have zero or one element (besides the header)] *)

FUNCTION CurrValue(List: MemListType):INTEGER [PURE];
(* Returns current value pointed to on List *)

PROCEDURE GetNext(List: MemListType);
(* Advances to next non-deleted element of list.  Assumes list is not empty. *)

PROCEDURE GetPrev(List: MemListType);
(* Moves back to first previous non-deleted element of list.
    Assumes list is not empty. *)


PROCEDURE SetPoint ( L: MemLoc);
(* Moves the alternative or attribute in L to the appropriate pointer *)
(* Saves the current Alt and Att in memory location *)


PROCEDURE Elim(List: MemListType);
(* Deletes current element from list (eliminates it from consideration) *)

(* * The following routines are for initializing and filling lists. * * * *)

PROCEDURE InitList(List: MemListType);
(* Initializes List to an empty list open for additions. *)


PROCEDURE PutValue(List: MemListType; Val: INTEGER);
(* Creates a new element at end of List.  Assumes AtEnd is true. *)

PROCEDURE CloseListAdditions(List: MemListType);
(* Initializes List to an empty list open for additions. *)

END; {Interface}
```

```
(*****************************************************************)
(* VERSION 1.2                                  TIMED RISKLESS SIM      *)
(* LSTPRIMS.PAS                                 11/12/85                *)
(*****************************************************************)
(* GetNext and GetPrev made generic to handle both risky and risk- *)
(* less.  SetPoint revised to handle empty (eliminated Alterns and *)
(* Attribs) cells in the Contents array.  Elim modified for risk-  *)
(* attribute handling; also, operationcounts distinguished to show *)
(* elimination of alternatives vs attributes.                      *)
(* "Time/weight" values for each operation proc read from an array *)
(* called OpWtArr, whose values are supplied by the user via an    *)
(* external file, WtsFile, in SIM.PAS.                             *)
(*****************************************************************)

{$INCLUDE:'sim.int'}
{$INCLUDE:'lstprims.int'}
{$INCLUDE:'stmprims.int'}
IMPLEMENTATION OF LSTPRIMS;
USES SIM,stmprims;
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* Primitive routines for handling the attribute and alternative lists     *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

FUNCTION MostFix;
{(List: MemListType): BOOLEAN [PURE]}
(* tests pointer for end of attributes in DoMostLikely *)

    BEGIN (* MostFix *)
        WITH MemLists[List] DO
            MostFix:= Pointer = (2 * NAttributes);
     END; (* MostFix *)


FUNCTION AtEnd;
{(List: MemListType): BOOLEAN [PURE]}
(* Is the List at the end marker? *)

    BEGIN (* AtEnd *)
        WITH MemLists[List] DO
            AtEnd:= Pointer = 0;
            TraceSTM('AtEnd',0,0,0);
    END; (* AtEnd *)


FUNCTION Cardinality;
{(List: MemListType): INTEGER [PURE]}
(* Does this list have zero or one element (besides the header)] *)

    BEGIN (* Cardinality *)
        WITH MemLists[List] DO
            Cardinality:= NContent;
            TraceSTM('Cardinality',0,0,0);
    END; (* Cardinality *)


FUNCTION CurrValue;
{(List: MemListType):INTEGER [PURE]}
(* Returns current value pointed to on List *)
```

```
    BEGIN (* CurrValue *)
        WITH MemLists[List] DO
            CurrValue:= Contents[Pointer];
    END; (* CurrValue *)


PROCEDURE GetNext;
{(List: MemListType)}
(* Advances to next non-deleted element of list.  Assumes list is not empty. *)

VAR TimeVal: REAL;

    BEGIN (* GetNext *)
        WITH MemLists[List] DO
            REPEAT
                Pointer:= (Pointer + 1) MOD (TotContent + 1);
            UNTIL (Contents[Pointer] <> 0) AND
                  ((RISKY) OR (List=Altern) OR ((List=Attrib)
                  AND ( ((Pointer MOD 2) = 1) OR (Pointer = 0))));
(* For riskless choice, move two times, unless AtEnd or passing  *)
(* by AtEnd.                                                      *)

        TraceSTM('GetNext',0,0,0);
        OperationCount[TheNext]:= OperationCount[TheNext] + 1;

        (* For Timed Sim, call Procedure in STMPRIMS to incr counter: *)
        TimeVal := OpWtArr[TheNext];
        Timer(TimeVal);

    END; (* GetNext *)


PROCEDURE GetPrev;
{(List: MemListType)}
(* Moves back to first previous non-deleted element of list.
    Assumes list is not empty. Modified for riskless version.   *)

VAR TimeVal: REAL;

    BEGIN (* GetPrev *)
        WITH MemLists[List] DO
            REPEAT
                Pointer:= (Pointer - 1 + TotContent + 1) MOD (TotContent + 1);
            UNTIL (Contents[Pointer] <> 0) AND
            ((RISKY) OR (List=Altern) OR ((List=Attrib)
            AND( ((Pointer MOD 2) = 1)
            OR (Pointer = 0))));

(* For riskless choice move 2 times unless AtEnd or passing by AtEnd. *)

        TraceSTM('GetPrev',0,0,0);
        OperationCount[ThePrev]:= OperationCount[ThePrev] + 1;

        (* For Timed Sim, call Procedure in STMPRIMS to incr counter: *)
        TimeVal := OpWtArr[ThePrev];
        Timer(TimeVal);

    END; (* GetPrev *)
```

```
PROCEDURE SetPoint;
{(L: MemLoc)}

(* Moves the alternative or attribute in L to the appropriate    *)
(* pointer.  Saves the current Alt and Att in memory location.   *)
(* If the target has been deleted, sets the pointer to the next  *)
(* non-empty cell.  If all have been deleted, sets it to the     *)
(* At-End marker.                                                *)

(* Alt Example. Cycles thru pointers until Contents' pointer equals   *)
(* STM[L]'s Alt.  However: STM[L]'s Alt could be a number for which   *)
(* there's no longer a match in the Contents array, since ELIM can    *)
(* set a Contents cell to zero.  Looping could be indefinite in search*)
(* of that eliminated value, if not for the "HOLD" processing.  Each  *)
(* increment to "HOLD" represents a complete cycle thru alts; ie-     *)
(* AtEnd(Altern) signals the start of a new cycle through.            *)

    VAR
        temp: MemCell;
        hold: INTEGER;
     TimeVal: REAL;

    BEGIN (* SetPoint *)
        temp.Alt:=CurrValue(Altern);
        temp.Att:= CurrValue(Attrib);

        WITH MemLists[Altern] DO
        BEGIN
           HOLD := 0; (* Kluge to prevent infinite loop if the Target *)
                      (* variable has been eliminated.                *)
           REPEAT
              IF (AtEnd(Altern)) THEN HOLD := HOLD+1;
              Pointer:=(Pointer + 1) MOD (TotContent + 1);
           UNTIL (Contents[Pointer] = STM[L].Alt)
           OR ((HOLD=2) AND (Contents[Pointer]>STM[L].Alt))
           OR ((HOLD=3) AND (Contents[Pointer]<STM[L].Alt))   (* change*)
           OR ((HOLD>=4) AND (AtEnd(Altern)));                 (* change*)
        END; (* With *)

        WITH MemLists[Attrib] DO
        BEGIN
           HOLD := 0;
           REPEAT
              IF (AtEnd(Attrib)) THEN HOLD := HOLD+1;
              Pointer:=(Pointer + 1) MOD (TotContent + 1);
           UNTIL (Contents[Pointer] = STM[L].Att)
           OR ((HOLD=2) AND (((Contents[Pointer]>STM[L].Att) and (RISKY))
             OR ((Contents[Pointer]>STM[L].Att) AND
               ODD(Pointer) AND (NOT RISKY))))
           OR ((HOLD>=3) AND AtEnd(Attrib));
        END; (* With *)

        STM[L].Alt:=temp.Alt;
        STM[L].Att:=temp.Att;
        TraceSTM('SetPoint',L,0,0);
        OperationCount[Finger] := OperationCount[Finger] + 1;

        (* For Timed Sim, call Procedure in STMPRIMS to incr counter: *)
        TimeVal := OpWtArr[Finger];
```

```
        Timer(TimeVal);

    END; (* SetPoint *)


PROCEDURE Elim;
{(List: MemListType)}
(* Deletes current element from list (eliminates from consideration). *)
(* Decrements cardinality.  See special code for riskless attribs.    *)
(* Note OperationCount increment: separate for alt vs. att.           *)

VAR TimeVal: REAL;

    BEGIN (* Elim *)
        IF AtEnd(List) THEN
            writeln('*** Attempt to delete end-list marker ***')
        ELSE WITH MemLists[List] DO
        BEGIN
           IF (List = Attrib) AND (RISKY = FALSE) THEN
             BEGIN
                Contents[Pointer]:=0;
                NContent:=NContent - 2;
                GetNext(List);
             END
        ELSE BEGIN
                Contents[Pointer]:= 0;
                NContent:= NContent - 1;
                GetNext(List);
             END;
         END;  (* WITH *)

        TraceSTM('Elim',0,0,0);
        IF (List=Altern) THEN
           OperationCount[AltAbsence]:=OperationCount[AltAbsence]+1;
        IF (List=Attrib) THEN
           OperationCount[AttAbsence]:=OperationCount[AltAbsence]+1;

        (* For Timed Sim, call Procedure in STMPRIMS to incr counter: *)
        IF (List=Altern) THEN
           TimeVal := OpWtArr[AltAbsence];
        IF (List=Attrib) THEN
           TimeVal := OpWtArr[AttAbsence];
        Timer(TimeVal);

    END; (* Elim *)


(* * The following routines are for initializing and filling lists. * * * *)

PROCEDURE InitList;
{(List: MemListType)}
(* Initializes List to an empty list open for additions. *)

    BEGIN (* InitList *)
        WITH MemLists[List] DO BEGIN
            NContent:= 0;
            Contents[0]:= -1; (* Sentinal for empty list spot. *)
        END;
```

```
    END; (* InitList *)


PROCEDURE PutValue;
{(List: MemListType; Val: INTEGER)}
(* Creates a new element at end of List.  Assumes AtEnd is true. *)

    BEGIN (* PutValue *)
        WITH MemLists[List] DO BEGIN
            Contents[NContent + 1]:= Val;
            NContent:= NContent + 1;
        END;
    END; (* PutValue *)


PROCEDURE CloseListAdditions;
{(List: MemListType)}
(* Initializes List to an empty list open for additions. *)

    BEGIN (* CloseListAdditions *)
        WITH MemLists[List] DO BEGIN
            TotContent:= NContent;
            Pointer:= 1;
        END;
    END; (* CloseListAdditions *)

END. {unit}
```

```
{$floatcalls+}
{$list-}
INTERFACE;
UNIT NumPrims (ISwap, IMax, IMin, RMax, RMin);

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*  Primitive number handling routines.                                      *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

PROCEDURE ISwap(VAR X1,X2: INTEGER);
FUNCTION IMax(X1,X2: INTEGER): INTEGER [PURE];
FUNCTION IMin(X1,X2: INTEGER): INTEGER [PURE];
FUNCTION RMax(X1,X2: REAL): REAL [PURE];
FUNCTION RMin(X1,X2: REAL): REAL [PURE];
END;
{$list+}
```

```
{$list-}
{$include:'numprims.int'}
IMPLEMENTATION OF NumPrims;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*  Primitive number handling routines.                                        *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

PROCEDURE ISwap;

    VAR Temp: INTEGER;

    BEGIN (* ISwap *)
        Temp:= X1;
        X1:= X2;
        X2:= Temp;
    END; (* ISwap *)


FUNCTION IMax;

    BEGIN (* IMax *)
        IF X1 > X2 THEN
            IMax:= X1
        ELSE IMax:= X2;
    END; (* IMax *)


FUNCTION IMin;

    BEGIN (* IMin *)
        IF X1 < X2 THEN
            IMin:= X1
        ELSE IMin:= X2;
    END; (* IMin *)


FUNCTION RMax;

    BEGIN (* RMax *)
        IF X1 > X2 THEN
            RMax:= X1
        ELSE RMax:= X2;
    END; (* RMax *)


FUNCTION RMin;

    BEGIN (* RMin *)
        IF X1 < X2 THEN
            RMin:= X1
        ELSE RMin:= X2;
    END; (* RMin *)

END.
{$list+}
```

```
(*********************************************************)
(* Program: Process                                 8/84 *)
(* Input: gamble files from Decide program. Format for   *)
(* each set must be 4 alternatives x 3 attributes (3 pay- *)
(* offs and 3 probabilities) with NRuns total sets.      *)
(* Output: same payoffs and probabilities, plus calculated*)
(* variance and expected value for each alternative in   *)
(* each set, and minimum variance and mean of the variance*)
(* for all the alternatives in each set.  Output is      *)
(* sorted by the minimum variance of each set, low to hi. *)
(*********************************************************)

PROGRAM Process (Input,Output,InData,OutData,NRuns);

   CONST
        Sp2 =  '  ';      (* Output formatting: 2 spaces. *)
        Sp3 =  '   ';     (*                    3 spaces. *)
        Sp4 =  '    ';    (*                    4 spaces. *)
     MaxSize = 200;

    TYPE
      Vector = SUPER ARRAY [1..*] of
             RECORD
                 AltXPayProb: ARRAY[1..4, 1..6] of REAL;
                 EVArr      : ARRAY[1..4]       of REAL;
                 VarArr     : ARRAY[1..4]       of REAL;
                 VBar       : REAL;
                 VMin       : REAL;
             END;

     GVector = Vector(MaxSize);      (* Handles up to 200 runs.      *)

    VAR
         InData: TEXT;
        OutData: TEXT;
          NRuns: INTEGER;
          GData: GVector;      (* Vector w/200 record-data cells.    *)

         (* Input Variables *)
         NAlternatives, NAttributes, Tran, AltNum: INTEGER;
         Att1, Att2, Att3                        : INTEGER;
         Pay1, Pay2, Pay3                        : REAL;
         Prob1, Prob2, Prob3                     : REAL;

         (* Sort Variables *)
         TmpArr                                  : ARRAY[1..4] of REAL;
         jmp, m, n                               : INTEGER;
         tmp, tindex                             : REAL;
         sorted                                  : BOOLEAN;

         (* VMinArr Matrix:                                      *)
         (*   1st col = initial position in unsorted list        *)
         (*   2nd col = VMin of a set of gambles                 *)
         VMinArr: ARRAY[1..200, 1..2] of REAL;

         (* Index to GData; sorted *)
         GIndex                                  : INTEGER;
         GRIndex                                 : REAL;

         i, j, k, l                              : INTEGER;
```

```
        InputCount : INTEGER;   (* Count of records read in *)
        OutputCount: INTEGER;   (* and written out.         *)


BEGIN  (* Process *)

    Rewrite(OutData);           (* create output file *)
    Reset(InData);              (* open input file    *)

    Writeln (Output, 'Program: Process');
    Writeln (Output, ' ');
    Writeln (Output, 'Calculating EV, Variance, VBar, VMin...');
    Writeln (Output, ' ');

    InputCount  := 0;
    OutputCount := 0;


    FOR k := 1 to NRuns DO
    BEGIN
        FOR i := 1 to 4 DO
        BEGIN

            Readln (InData,
                    NAlternatives, NAttributes, Tran, AltNum,
                    Att1, Pay1, Prob1,
                    Att2, Pay2, Prob2,
                    Att3, Pay3, Prob3);
            InputCount := InputCount + 1;


            WITH GData[k] DO        (* using GData record fields *)
            BEGIN

            (* Assign input data to record matrix: *)
                AltXPayProb[i,1] := Pay1;
                AltXPayProb[i,2] := Prob1;
                AltXPayProb[i,3] := Pay2;
                AltXPayProb[i,4] := Prob2;
                AltXPayProb[i,5] := Pay3;
                AltXPayProb[i,6] := Prob3;

            (* Compute expected value of each alternative *)
            (* in the set:                                *)
                EVArr[i] := (AltXPayProb[i,1] * AltXPayProb[i,2]) +
                            (AltXPayProb[i,3] * AltXPayProb[i,4]) +
                            (AltXPayProb[i,5] * AltXPayProb[i,6]);

            (* Compute the variance of ea alt in the set: *)
               VarArr[i] :=  1/3 * (sqr(AltXPayProb[i,2]))
                           + 1/3 * (sqr(AltXPayProb[i,4]))
                           + 1/3 * (sqr(AltXPayProb[i,6]))
                           - 2/9 *      AltXPayProb[i,2]
                           - 2/9 *      AltXPayProb[i,4]
                           - 2/9 *      AltXPayProb[i,6]
                           + 1/9;

           END; (* WITH *)
        END; (* FOR; of 4-loop *)
```

```
      (* Calculate VBar for the set: *)
      WITH GData[k] DO
      BEGIN
      VBar := (VarArr[1] + VarArr[2] + VarArr[3] + VarArr[4])/4;
      END; (* WITH *)

      (* Put the values in VarArr into a temporary array *)
      (* which will then be sorted:                      *)
      WITH GData[k] DO
      BEGIN
      For l := 1 to 4 DO
      BEGIN
          TmpArr[l] := VarArr[l];
      END;  (* FOR *)
      END;  (* WITH *)

      (* Find the minimum variance by sorting TmpArr.  Shell  *)
      (* sort from Grogono.                                   *)
      jmp := 4;                (* 4 values to sort *)
      WHILE jmp > 1 DO
      BEGIN
          jmp := jmp DIV 2;
          REPEAT
              sorted := TRUE;
              FOR m := 1 to 4 - jmp DO
              BEGIN
                  n := m + jmp;
                  IF TmpArr[m] > TmpArr[n] THEN
                  BEGIN
                            tmp := TmpArr[m];
                      TmpArr[m] := TmpArr[n];
                      TmpArr[n] := tmp;
                         sorted := FALSE;
                  END; (* IF *)
              END; (* FOR *)
          UNTIL sorted; (* REPEAT *)
      END; (* WHILE *)

      (* Minimum variance for the set is the 1st element in *)
      (* the sorted array.                                  *)
      WITH GData[k] DO
      BEGIN
          VMin := TmpArr[1];
      END; (* WITH *)

      (* Set up the pointer array, so it holds both the     *)
      (* value of k in this loop of NRuns, & the VMin value *)
      (* of this set.                                       *)
      VMinArr[k,1] := k;          (* 1st cell = index value  *)
      VMinArr[k,2] := TmpArr[1]; (* 2nd cell = min variance *)

END; (* FOR; of NRuns-loop *)

(* Sort VMinArr by VMin, keeping the associated value of k  *)
(* with it's VMin value.  The k values in the sorted array  *)
(* will then be used to output GData in sorted VMin order.  *)

jmp := NRuns;                    (* NRuns values to sort.     *)
WHILE jmp > 1 DO
```

```
BEGIN
    jmp := jmp DIV 2;
    REPEAT
        sorted := TRUE;
        FOR m := 1 to NRuns - jmp DO
        BEGIN
            n := m + jmp;
            IF VMinArr[m,2] > VMinArr[n,2] THEN
            BEGIN
                tmp := VMinArr[m,2];          tindex := VMinArr[m,1];
      VMinArr[m,2] := VMinArr[n,2];   VMinArr[m,1] := VMinArr[n,1];
      VMinArr[n,2] := tmp;            VMinArr[n,1] := tindex;
      sorted := FALSE;
            END; (* IF *)
        END; (* FOR *)
   UNTIL sorted; (* REPEAT *)
END; (* WHILE *)

Writeln (Output, 'Sort by VMin complete.');
Writeln (Output, ' ');


(* VMinArr is a matrix now in sorted order by VMin from *)
(* low to high.  Associated w/the value VMin is an in-  *)
(* teger indicating the element of the GData array that *)
(* contains the data belonging with VMin.  A loop reads *)
(* down this ordered VMinArr from the 1st element to the*)
(* last and retrieves the integer which is used to index*)
(* GData for output.                                    *)

(* Write headings: *)
Writeln (OutData,
         Sp2, 'Set', Sp4, 'Altern', Sp3,
         Sp3, '$1', Sp3, Sp4, 'P1', Sp2, Sp3,
         Sp3, '$2', Sp3, Sp4, 'P2', Sp2, Sp3,
         Sp3, '$3', Sp3, Sp4, 'P3', Sp4, Sp2,
         Sp3, 'EV', Sp3, Sp2, Sp3, 'V', Sp4,
         '  ', 'VBar', Sp4, 'VMin');
Writeln (OutData, '_____________________________________________',
                  '_____________________________________________',
                  '_____________________________');

Writeln (OutData, Sp4);

FOR i := 1 to NRuns DO
BEGIN

    GRIndex := VMinArr[i,1];       (* Get index as real var.*)
    GIndex  := round(GRIndex);     (* Put into integer var  *)
                                   (*  w/rounding function. *)
    WITH GData[GIndex] DO
    BEGIN
        FOR j := 1 to 4 DO
        BEGIN
            Writeln
            (OutData, 'Set#', i:3, ' Altern#', j:1, Sp2,

            AltXPayProb[j,1]: 8:2, Sp2,
            AltXPayProb[j,2]: 6:3, Sp3,
            AltXPayProb[j,3]: 8:2, Sp2,
```

```
                AltXPayProb[j,4]: 6:3, Sp3,
                AltXPayProb[j,5]: 8:2, Sp2,
                AltXPayProb[j,6]: 6:3, Sp4,

                EVArr[j]: 8:3, Sp2,
                VarArr[j]: 6:3, Sp2,
                VBar: 6:3, Sp2,
                VMin: 6:3);
            END; (* FOR *)

            Writeln (OutData, '    ');     (* space a line *)
            OutputCount := OutputCount + 1;
        END; (* WITH *)
    END; (* FOR *)
    Writeln (Output, '# Input Records  = ', InputCount);
    Writeln (Output, '# Output Records = ', OutputCount);
    Writeln (Output, ' '); Writeln (Output, 'End Program.');

END. (* Program *)
```

```
(*****************************************************************)
(* VERSION 1.2                                        SIM RISKLESS    *)
(* RANDOM.INT                                         6/21/85         *)
(*****************************************************************)

{$list+}
INTERFACE;
UNIT RANDOM(Ggubs,IntRand,GenProbs,vsrtr,vsrta,IStat,RStat,
        Permuter,Hostile,Dirichlet,Ggnml,Ggnms,MakeSig,
        Ggums,URand);
USES Sim;



(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*  random number generation and sorting.                                  *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)



PROCEDURE Ggubs(VAR Seed: INTEGER4; nrand: INTEGER; VAR randnum: Vector);

FUNCTION IntRand(Bot, Top: INTEGER): INTEGER;

PROCEDURE GenProbs(VAR RVs: Vector; nRV: INTEGER);

PROCEDURE vsrtr (VAR a: Vector; la: INTEGER; VAR ir: Map);

PROCEDURE vsrta (VAR A: Vector; la: INTEGER);

PROCEDURE IStat (CONST A: Map; CONST n: INTEGER; VAR Mean, StdDev: REAL);

PROCEDURE RStat (CONST A: Vector; CONST n: INTEGER; VAR Mean, StdDev: REAL);

PROCEDURE Permuter(VAR  NumRV: INTEGER; VAR RValues: Vector);

PROCEDURE Hostile(VAR RVs: Vector; nRV: INTEGER);

(* Added 10/84: *)
PROCEDURE Dirichlet (VAR RVs: Vector; nRV: INTEGER);

(* Added 6/85.  Sanjoy Ghose's translation of certain IMSL random     *)
(*              number routines:                                      *)

PROCEDURE Ggnml (VAR Seed:INTEGER4; nrand:INTEGER; VAR Arr1:VECTOR);
(* Normal deviate random number generator. *)

PROCEDURE Ggnms (VAR x:VTable; NCAlt,NCAtt:INTEGER; ArrSig:VTable);
(* Produces normal variates with a given variance-covariance matrix. *)

PROCEDURE MakeSig;
(* Generates a variance-covariance matrix for GGNMS; modified 6/85.   *)

PROCEDURE Ggums (VAR X:VTable; NCAlt,NCAtt:INTEGER; ArrSig:VTable);
(* Translates the normals into a uniform, but which probably generates *)
(* some ties in the process.                                           *)

FUNCTION URand (VAR Seed: Integer4): Real;
```

```
END;
```

```
(*****************************************************************)
(* VERSION 1.2                                        SIM RISKLESS   *)
(* RANDOM.PAS                                         11/19/85       *)
(*****************************************************************)

{$INCLUDE:'sim.int'}
{$INCLUDE:'random.int'}
IMPLEMENTATION OF RANDOM;
USES SIM;


(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*  random number generation and sorting.                                  *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)


PROCEDURE Ggubs;
(This is a PASCAL implementation of the Schrage portable  random
number generator)

    Const       a = 16807;
                b15 = 32768;
                b16 = 65536;
                p = 2147483647;
                sdiv = 4.656612875e-10;

    Var         xhi, xalo, leftlo, fhi, k : integer4;
                i: INTEGER;

    BEGIN (* Ggubs *)
        FOR i := 1 TO nrand DO
            BEGIN
                xhi := seed div b16;
                xalo := (seed - xhi * b16) * a;
                leftlo := xalo div b16;
                fhi := xhi * a + leftlo;
                k := fhi div b15;
                seed := (((xalo - leftlo * b16) - p) + (fhi - k * b15) * b16) +
                if seed < 0 then seed := seed + p;
                randnum[i] := float4(seed) * sdiv;
            END
    END; (* Ggubs *)



FUNCTION IntRand;

    VAR RealTemp: AttVec;
                                (* Holds the temp. real # between 0 and 1 *)
        Int: INTEGER;

    BEGIN (* IntRand *)

        Int:=12;
        Ggubs(DSeed,Int,RealTemp);

        IntRand:= TRUNC((Top - Bot + 1) * RealTemp[1]) + Bot;


        (* 0 <= RealTemp[1] < 1  ==> 0 <== trunc() < Top - Bot + 1 ==>
```

```
            Bot <= IntRand < Top + 1 ==> Bot <= IntRand <= Top *)

    END; (* IntRand *)


PROCEDURE GenProbs;
(* Generates nRV uniform[0,1] random variables which are normalized to sum to
   1, but are otherwise independent & uniform.  Method used by Thorngate.
   See Johnson and Payne Appendix B for it properties. *)

    VAR Sum: REAL;
        I: INTEGER;

    BEGIN (* GenProbs *)
        (* Get all variables unif(0,1) *)
        I:= 12;
        Ggubs(DSeed,I,RVs);

        (* Get the sum of the uniform variables. *)
        Sum:= 0;
        FOR I:= 1 TO nRV DO
            Sum:= Sum + RVs[I];

        (* Normalize the variables. *)
        FOR I:= 1 TO nRV DO
            RVs[I]:= RVs[I] / Sum;
    END; (* GenProbs *)



PROCEDURE vsrtr;
(* Shell sort adapted from Grogono *)
{Functions identically to the IMSL routine of the same name}


    VAR
        jmp, m, n, tindex: INTEGER;
        tmp: REAL;
        sorted: BOOLEAN;

    BEGIN (* vsrtr *)
        jmp := la;
        WHILE jmp > 1 DO
            BEGIN
                jmp := jmp DIV 2;
                REPEAT
                    sorted := TRUE;
                    FOR m := 1 TO la - jmp DO
                        BEGIN
                            n := m + jmp;
                            IF A[m] > A[n] THEN
                                BEGIN
                                    tmp := A[m];   tindex := ir[m];
                                    A[m] := A[n];  ir[m] := ir[n];
                                    A[n] := tmp;   ir[n] := tindex;
                                    sorted := FALSE
                                END
                        END (* FOR *)
                UNTIL sorted
            END (* WHILE *)
```

```
    END; (* vsrtr *)


PROCEDURE vsrta;
(* Shell sort adapted from Grogono *)
{Identical to IMSL routine of the same name}
    VAR
        jmp, m, n: INTEGER;
        tmp: REAL;
        sorted: BOOLEAN;

    BEGIN (* vsrta *)
        jmp := la;
        WHILE jmp > 1 DO
            BEGIN
                jmp := jmp DIV 2;
                REPEAT
                    sorted := TRUE;
                    FOR m := 1 TO la - jmp DO
                        BEGIN
                            n := m + jmp;
                            IF A[m] > A[n] THEN
                                BEGIN
                                    tmp := A[m];
                                    A[m] := A[n];
                                    A[n] := tmp;
                                    sorted := FALSE
                                END
                        END (* FOR *)
                UNTIL sorted
            END (* WHILE *)
    END; (* vsrta *)


PROCEDURE IStat;
{Subroutine which  calculates the means and StdDev  of an integer vector
 A, of length n}

    VAR     J: INTEGER;

    BEGIN (* IStat *)




        FOR J := 1 TO n DO
            BEGIN
                Mean := Mean + A[J];
                StdDev := StdDev + SQR (A[J])
            END;

        If (I = NRun) Then
        BEGIN
        StdDev := SQRT ((StdDev - SQR (Mean) / n) / (n - 1));
        Mean := Mean / n;
        End;

    END; (* IStat *)
```

```
PROCEDURE RStat;
{Calculates the Mean and StdDev of REAL vector A of length n}
    VAR     J: INTEGER;

    BEGIN (* RStat *)



        FOR J := 1 TO n DO
            BEGIN
                Mean := Mean + A[J];
                StdDev := StdDev + SQR (A[J])
            END;

        If (I = NRun) Then
        BEGIN
        StdDev := SQRT ((StdDev - SQR (Mean) / n) / (n - 1));
        Mean := Mean / n;
        END;

    END; (* RStat *)


PROCEDURE Permuter;
(* (VAR NumRV:INTEGER; VAR RValues:VECTOR); *)
(* Called from InitProbabilities in START.PAS.  Procedure to create an*)
(* array used for a random assignment of hostile environment probs to *)
(* the array of attributes.  More generally, a routine which simply   *)
(* permutes RValues.                                                  *)

    VAR RandArray: AttMap;
        TempReal:  AttVec;
        J        : INTEGER;

    BEGIN (* Permuter *)
        FOR J := 1 TO NumRV DO
            RandArray[J]:= J;

        (*These are "imsl" routines that generate randoms & sort arrays*)
        J:=12;
        Ggubs(DSeed,J,TempReal);
        vsrtr(TempReal,NumRV,RandArray);

        FOR J:= 1 TO NumRV DO
            TempReal[J]:=RValues[RandArray[J]];

        FOR J:= 1 TO NumRV DO
            Rvalues[J]:=TempReal[J];

    END; (* Permuter *)


PROCEDURE Hostile;
(* VAR RVs:VECTOR; nRV:INTEGER); *)
(* Called from InitProbabilities in START.PAS.                    *)
(* The high variance equivalent to GenProbs.  Returns in RVs a *)
(* vector of length nRVs, a high variance set of probabilities.*)

    VAR sum: REAL;
        J,AttNum: INTEGER;
```

```
        AttRand: AttMap;
        Transform: AttVec;

    BEGIN (* Hostile *)
        sum:= 0;
        J:=12;
        Ggubs(DSeed,J,RVs);
        FOR J:=1 TO (nRV - 1) DO BEGIN
            Transform[J]:=(1.0 - sum) * RVs[J];
            sum:=sum + Transform[J];
        END; (* FOR *)
        Transform[nRV]:=1.0 - sum;

        Permuter(nRV,Transform);

        FOR J:=1 TO nRV DO
            RVs[J]:=Transform[J];
    END; (* Hostile *)

PROCEDURE Dirichlet;
(* VAR RVs:Vector; nRV:INTEGER *)
(* Added 10/84. Called from InitProbabilities in START.PAS.    *)


VAR
    NRand:   INTEGER;                (* Number of random numbers.   *)
    CalcArr: Array[0..10] of REAL; (* Intermed. array. Zero/one   *)
                                     (*  set as lower/upper limits. *)
    FullArr: Array[0..10] of REAL; (* Contains weights after calc.*)
                                     (*  w/zero/one low/up limits.  *)
    II     : INTEGER;                (* Looping index.              *)
    RandNum: AttVec;                 (* Array passed to Ggubs.      *)
                                     (* Array[1..12] of REAL;VECTOR.*)

BEGIN  (* Dirichlet *)

    NRand := nRV - 1;                (* Establish number of rand #s. *)
    Ggubs(DSeed, NRand, RandNum);    (* Generate random numbers.     *)
    Vsrta(RandNum, NRand);           (* Sort them low to high.       *)

    (* Define lower & upper limits of an array and put sorted      *)
    (* RandNum array into this array:                              *)
    CalcArr[0]   := 0;
    CalcArr[nRV] := 1;
    FOR II := 1 to NRand DO
        CalcArr[II] := RandNum[II];

    (* Calculate weights: set up array FullArr w/zero and one as   *)
    (* lower and upper limits.  Then calculate weights using ran-  *)
    (* dom numbers in CalcArr and put results into FullArr.        *)
    FullArr[0]   := 0;
    FullArr[nRV] := 1;
    FOR II := 1 to nRV DO
        FullArr[II] := CalcArr[II] - CalcArr[II-1];

    (* Transfer nRV number of weights (NAttributes) in FullArr     *)
    (* to the array RVs (ie, TempValues, for later transfer to     *)
    (* array Values).                                              *)
    FOR II := 1 to nRV DO
        RVs[II] := FullArr[II];
```

```
END; (* Dirichlet *)


Procedure GGNML;
{ (var seed:integer4;nrand:integer;VAR ARR1:AttVec)   }

VAR
      Temp1,Temp2,Temp3,Temp4 : Real;
      Us:AttVec;
      I:INTEGER;

BEGIN (*GGNML*)
   I:=1;

   WHILE I<=NRAND DO
     BEGIN


                BEGIN
                Temp1:=urand(dseed) ;
                Temp2 := sqrt(-2.0 *
                ln(Temp1));
                Temp3:=urand(dseed);
                temp4 := 6.2831853 * Temp3;
                ARR1[I] := temp2 * cos(temp4);
                END;


        IF (I+1)<=NRAND THEN
            BEGIN
            ARR1[I+1] := temp2 * sin(temp4);
            END;
        I:=I+2;
     END
END; (*GGNML*)



PROCEDURE GGNMS;
{(VAR X:Vtable;NCAlt,NCAtt:INTEGER;VAR ARRSIG:Vtable);}

VAR

        k1,k,j,i,k2,j1,j2,J3:INTEGER;
        ck,sum,sum1:REAL;
        sum2,arr1:AttVec;
        arrc:Vtable;


BEGIN(*GGNMS*)

   FOR j3:=1 to NCAlt DO
   FOR J1:=1 TO NCAtt DO
    BEGIN(*FOR*)
      x[j3,J1]:=0.0
    END;(*FOR*)
      FOR j1:=1 to NCAtt DO
      FOR j2:=1 to NCAtt DO
```

```
      BEGIN(*FOR*)
         arrc[j1,j2]:=0.0
      END;(*FOR*)


  FOR i:=1 to NCAtt DO
   BEGIN(*FOR*)
       arrc[i,1]:=arrsig[i,1]/SQRT(arrsig[1,1])

   END;(*FOR*)


  FOR i:=2 to NCAtt DO(*LINE 10, FORTRAN*)
    BEGIN(*FOR*)
      sum:=0.0;
      k1:=i-1;
      FOR k:=1 to k1 DO(*LINE 13, FORTRAN*)
        BEGIN(*FOR*)
          sum:=sum+arrc[i,k]*arrc[i,k]
        END;(*FOR,LINE 13, FORTRAN*)
      ck:=arrsig[i,i]-sum;
      IF ck<0.0
         THEN WRITE(output,'STOP CALCULATIONS')
         ELSE arrc[i,i]:=SQRT(ck);


      IF i<NCAtt
        THEN
        BEGIN(*THEN*)
           k1:=i;
           FOR j:=2 to k1 DO (*LINE21,FORTRAN*)
             BEGIN(*FOR*)
               sum1:=0.0;
               k2:=j-1;
               FOR k:=1 to k2 DO (*LINE 24,FORTRAN*)
                 BEGIN(*FOR*)
                   sum1:=sum1+arrc[i+1,k] * arrc[j,k]
                 END;(*FOR*)
                 arrc[i+1,j]:=(arrsig[i+1,j]-sum1)/arrc[j,j]


             END(*FOR,LINE 21,FORTRAN*)
        END; (*THEN*)
    END;(*FOR,LINE 10*)

      FOR J3 := 1 to NCAlt DO
              BEGIN   {FOR}
                      FOR i:= 1 to NCAtt DO
                                    Sum2[i] := 0;
                      GGNML(Dseed,NCAtt,arr1);

                      FOR i:= 1 TO NCAtt DO
                              FOR  j := 1 to NCAtt DO
                                      sum2[i] := sum2[i] + arrc[i,j] * arr1[j]

                      For i := 1 to NCAtt DO
                              x[j3,i] := sum2 [i];
   END;     (*FOR J3*)

END;(*GGNMS*)
```

```
PROCEDURE MakeSig;
VAR     i,j     :Integer;
         direction: real;
         Equal: Boolean;


BEGIN  (* MakeSig.  Mod. 6/85 to answer "Are covariances equal?" and *)
       (* "If yes, give correlation; if no, name input file for      *)
       (* values" -- ie, SigFile -- via input in program header.     *)

    IF CVEq = TRUE THEN         (* CVEq = Are CoVariances equal? *)
       Direction := Corr;       (* Corr = correlation.           *)
    IF CVEq = FALSE THEN
       Reset (SigFile);         (* Open input file of sigma values.*)

    i := 1; j := 1;
    WHILE i <= NAttributes DO
    BEGIN

      j := 1;
      WHILE j <= NAttributes DO
      BEGIN
         IF CVEq THEN
         BEGIN                   (* Covariances are equal. *)
            IF i<>j
            THEN
               sigma[i,j]:=Direction
            ELSE
               sigma[i,j]:= 1.0;
         j:=j+1;

         END
       ELSE
         BEGIN                   (* Covariances are unequal.        *)
  (* Fills upper diagonal half of sigma matrix from external file. *)
         IF (i=j) THEN
            sigma[i,j]:=1.0;
         IF (i>j) THEN
            REPEAT        (* until *)
               IF (i=j) THEN
                  sigma[i,j]:=1.0;
               j:=j+1;
            UNTIL (j>i);  (* repeat loop *)
         IF (i<>j) AND (j <= NAttributes) THEN
         BEGIN
            Read(SigFile, sigvalue);
            sigma[i,j]:=sigvalue;
         END;
         j:=j+1;

         END; (* ELSE *)

      END; (* While j DO *)

       i:=i+1;
   END; (* While i DO *)
```

```
   IF CVEq = FALSE THEN
   BEGIN
      Close (SigFile);
  (* Fill in lower half of sigma w/matching values: *)
      For j:=1 to (NAttributes - 1) DO
         For i:= 2 to NAttributes DO
            sigma[i,j]:=sigma[j,i];
   END;

 (* Test code: eliminate after testing...prints out matrix. *)
  FOR i:=1 to NAttributes DO
    FOR j:=1 to NAttributes DO
    BEGIN
       Writeln(output, 'Sigma ',i,j,' = ',sigma[i,j]);
    END;
 (*************************************************************)

END;    {MakeSig}



PROCEDURE GGUMS;
{(VAR X:VTable;NCAlt,NCAtt:INTEGER;VAR ARRSIG:VTable);}

{Segments the normal variates produced by a call to GGNMS to produce
a series of 0,1 uniforms with the same correlation.  It does this by
taking the normals and sorting them into one of a set of bins, and then doing
a (hopefully) appropriate division that would make sure the normals fall
in the bin.}

LABEL 1;

VAR I,J :INTEGER;
Sign    :BOOLEAN;
y       :Vtable;

BEGIN   {ggums}

GGNMS(X,NCAlt,NCAtt,ARRSIG);

FOR i:= 1 to NCAlt DO
FOR j:= 1 to NCAtt DO
   BEGIN(*FOR*)
        IF (X[I,J] < 0.0) THEN
                BEGIN
                x[i,j] := x[i,j] * (-1.0);
                Sign := true;
                END
        ELSE Sign := FALSE;

        IF (x[i,j] < 0.1) THEN
BEGIN
 y[i,j] := 0.5;
        GOTO 1;
END;
        IF (x[i,j] <  0.2) THEN
BEGIN
 y[i,j] := 0.5398 ;
        GOTO 1;
END;
```

```
        IF (x[i,j] <  0.3) THEN
BEGIN
 y[i,j] := 0.5793 ;
        GOTO 1;
END;
        IF (x[i,j] <  0.4) THEN
BEGIN
 y[i,j] := 0.6179 ;
        GOTO 1;
END;
        IF (x[i,j] <  0.5) THEN
BEGIN
 y[i,j] := 0.6554 ;
        GOTO 1;
END;
        IF (x[i,j] <  0.6) THEN
BEGIN
 y[i,j] := 0.6915 ;
        GOTO 1;
END;
        IF (x[i,j] <  0.7) THEN
BEGIN
 y[i,j] := 0.7257 ;
        GOTO 1;
END;
        IF (x[i,j] <  0.8) THEN
BEGIN
 y[i,j] := 0.7580 ;
        GOTO 1;
END;
        IF (x[i,j] <  0.9) THEN
BEGIN
 y[i,j] := 0.7881 ;
        GOTO 1;
END;
        IF (x[i,j] <    1.0) THEN
BEGIN
 y[i,j] := 0.8159 ;
        GOTO 1;
END;
        IF (x[i,j] <    1.1) THEN
BEGIN
 y[i,j] := 0.8413 ;
        GOTO 1;
END;
        IF (x[i,j] <    1.2) THEN
BEGIN
 y[i,j] := 0.8643 ;
        GOTO 1;
END;
        IF (x[i,j] <    1.3) THEN
BEGIN
 y[i,j] := 0.8849 ;
        GOTO 1;
END;
        IF (x[i,j] <    1.4) THEN
BEGIN
 y[i,j] := 0.9032 ;
        GOTO 1;
END;
```

```
        IF (x[i,j] <    1.5) THEN
BEGIN
 y[i,j] := 0.9192 ;
        GOTO 1;
END;
        IF (x[i,j] <    1.6) THEN
BEGIN
 y[i,j] := 0.9332 ;
        GOTO 1;
END;
        IF (x[i,j] <    1.7) THEN
BEGIN
 y[i,j] := 0.9452 ;
        GOTO 1;
END;
        IF (x[i,j] <    1.8) THEN
BEGIN
 y[i,j] := 0.9554 ;
        GOTO 1;
END;
        IF (x[i,j] <    1.9) THEN
BEGIN
 y[i,j] := 0.9641 ;
        GOTO 1;
END;
        IF (x[i,j] <    2.0) THEN
BEGIN
 y[i,j] := 0.9713 ;
        GOTO 1;
END;
        IF (x[i,j] <    2.1) THEN
BEGIN
 y[i,j] := 0.9772 ;
        GOTO 1;
END;
        IF (x[i,j] <    2.2) THEN
BEGIN
 y[i,j] := 0.9821 ;
        GOTO 1;
END;
        IF (x[i,j] <    2.3) THEN
BEGIN
 y[i,j] := 0.9861 ;
        GOTO 1;
END;
        IF (x[i,j] <    2.4) THEN
BEGIN
 y[i,j] := 0.9893 ;
        GOTO 1;
END;
        IF (x[i,j] <    2.5) THEN
BEGIN
 y[i,j] := 0.9918 ;
        GOTO 1;
END;
        IF (x[i,j] <    2.6) THEN
BEGIN
 y[i,j] := 0.9938 ;
        GOTO 1;
END;
```

```
        IF (x[i,j] <    2.7) THEN
BEGIN
 y[i,j] := 0.9953 ;
        GOTO 1;
END;
        IF (x[i,j] <    2.8) THEN
BEGIN
 y[i,j] := 0.9965 ;
        GOTO 1;
END;
        IF (x[i,j] <    2.9) THEN
BEGIN
 y[i,j] := 0.9974 ;
        GOTO 1;
END;
        IF (x[i,j] <    3.0) THEN
BEGIN
 y[i,j] := 0.9981 ;
        GOTO 1;
END;
        IF (x[i,j] <    3.1) THEN
BEGIN
 y[i,j] := 0.9987 ;
        GOTO 1;
END;
        IF (x[i,j] <    3.2) THEN
BEGIN
 y[i,j] := 0.9987 ;
        GOTO 1;
END;
        IF (x[i,j] <    3.3) THEN
BEGIN
 y[i,j] := 0.9987 ;
        GOTO 1;
END;
        IF (x[i,j] <    3.4) THEN
BEGIN
 y[i,j] := 0.9988 ;
        GOTO 1;
END;
        IF (x[i,j] <    3.5) THEN
BEGIN
 y[i,j] := 0.9988 ;
        GOTO 1;
END;
    IF (x[i,j] >=   3.5) THEN
 y[i,j] := 0.9989;

1:      IF sign THEN  x[i,j] := 0.5 - (y[i,j] - 0.5)
                ELSE x[i,j] := y[i,j]           {reverses for neg. numbers}

   END;(*FOR*)

END;   {ggums}


FUNCTION URand;
(* VAR Seed:Integer4) *)

VAR
```

```
    Rands: AttVec;

BEGIN
   Ggubs(Seed,1,Rands);
   URand:=Rands[1];
END;  (* URand *)


END. {Unit}
```

```
{Interface for SCREENVV, Screen handler routines for TOOLS            }
{Version 2.0 (C)Copyright Blaise Computing, Inc.  1982                }

interface; unit screenvv(adaptvv,resetvv,modevv,actpagvv,pagsetvv,
                         pagposvv,scrollvv,clsvv,locvv,cursetvv,
                         curposvv,attribvv,cursorvv,readvv,
                         writevv,scnwrtvv);

  var
    adaptvv : boolean;           ! True if color/graphics adapter used.

  procedure resetvv (mode : byte);
  procedure modevv  (var mode,columns,cur_apage : byte);
  procedure actpagvv(apage : byte);
  procedure pagsetvv(cpage : byte);
  procedure pagposvv(var cpage : byte);
  procedure scrollvv(no_lines,attrib,upper_row,upper_col,
                          lower_row,lower_col,code : byte);
  procedure clsvv;
  procedure locvv   (var row,col : byte; code : byte);
  procedure cursetvv(row,col : byte);
  procedure curposvv(var row,col : byte);
  procedure attribvv(foreground,background : byte; ch : char;
                                                   no_chars : word);
  procedure cursorvv(on_off,high,low : byte);
  procedure readvv  (var ch : char; var foreat,backat : byte);
  procedure writevv (ch : char; no_chars : word);
  procedure scnwrtvv(ch : char; fore    : byte);

  begin
  end;
```

```
{$floatcalls+}
{$linesize:130}

(*****************************************************************)
(* VERSION 1.2                                  TIMED RISKLESS SIM  *)
(* SIM.INT                                      11/12/85            *)
(*****************************************************************)
(*         - 2 vars added to pgm header: CutOffValue and         *)
(* DIST, to specify cutoff for gamble value (was 500)            *)
(* and distribution (was 1000). Added also to SIM.PAS            *)
(* and changes made to START's Initialize and InitValues.        *)
(* 6/6/85 - SIM divided into risky & riskless versions.          *)
(*  Risky no longer input by user, but var assigned in           *)
(*  Decide. Special set of rules for Riskless version.           *)
(*  Pgm header FmtPrt added to edit or condense output.          *)
(*  JPIndex variables added for calc in Sim.Pas.                 *)
(* XLimit - Problem ...some kind of strange overflow?            *)
(*  Dummy var added to attempt to circumvent.                    *)
(* 9/27/85 - Untimed AU rule performed automatically to          *)
(*  provide baseline data for JPIndex calc.  Timed AU rule       *)
(*  renamed TimedAU; added to rules. No output generated         *)
(*  for untimed AU rule; values simply used for JPIndex.         *)
(*****************************************************************)

INTERFACE;
UNIT Sim (NMemCell,MaxAttributes,TwoXMaxAttributes,MaxAlternatives,MaxList,
        Combined,CutOffVal,OperResults,
        Empty,Minimum,Maximum,Difference,Sum,Product,
        Quotient,Momento,Blood,Squint,Oversight,TheNext,ThePrev,
        Finger,AltAbsence,AttAbsence,MemLoc,MemCell,
        MemListType,Altern,Attrib,
        MemListRecord,
        Vector,Map,AttVec,AttMap,
        TwoXAttVec,Vtable,AltVec,AltMap,CombVec,

        Risky,

        Rules,
        Rand,AddUtility,TimedAU,EBA,LexiSemi,
        Lexi,EqualWeight,MCD,EBAEU,
        EBAMCD,Satisficing,
        Rule,Used,MeanData,StdData,

        O,Mean,StdDev,STM,Values,CutOffs,DomValues,
        ChoiceUtils,ChoiceValues,ChoiceUProp,ChoiceProp,ChoicePro2,
        ChoiceIndex,ChoiceRanking,ChoiceURank,MemLists,AltAU,UAltAU,AltRand,
        NRep,Noise,

        Time,TimeUp,OpWtArr,

        AAADummy,AdULimit,EBALimit,LexLimit,
        LxLimit,EqWLimit,MCDLimit,EEULimit,
        ECDLimit,SatLimit,

        EBARulCnt,EBARndCnt,
        SatRulCnt,SatRndCnt,LSORulCnt,LSORndCnt,MCDTieCnt,
        JPIdx,MCVArr,

        Sigma,SigmaX,SigValue,
```

```
        BestChoice,Proportion,

        WinsOut,DecChoiceArr,
        AdUDec,EBADec,LexDec,LxDec,EqWDec,
        MCDDec,EEUDec,ECDDec,SatDec,

        RunSim,Debug,NRun,NAlternatives,NAttributes,
        FmtPrt,TraceOut,Gamble,Sigfile,WtsFile,Tran,
        PG,TimeLimit,NumDomAltern,CVEq,Corr,JND,
        DSeed,Dist,CutOffValue,RulesToRun,

        DomExist,OperationCount,

        i, j, k, l, m, n, p, q, r, s,
        Initialize);


{$list-}
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * **)
(*             HUMAN DECISION MODELING PROGRAM.                            *)
(*  Written by Ethan Bradford on 6/20/81 for Eric Johnson.                 *)
(*  Modified for IBM PC by Matthew J. Saltzman 9/83                        *)
(*                Carnegie-Mellon University.                              *)
(*  Modified by Paula Ecklund for John Payne,                              *)
(*                Duke University.                                         *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * **)
(*  THIS PROGRAM PROVIDES AN INTERPRETER FOR (BUILT IN) DECISION RULES.     *)
(*  THESE DECISION RULES ARE ABLE TO ACCESS (R/W) A SHORT TERM MEMORY (STM)*)
(*  WITH A FIXED AMOUNT OF STORAGE, AND TWO LISTS OF NUMBERS (MEMLISTS)    *)
(*  USING A SET OF LIMITED INTERFACING FUNCTIONS.  THE RULE MUST CHOOSE    *)
(*  FROM AMONG SEVERAL ALTERNATIVES, EACH OF WHICH HAS A LIST OF           *)
(*  ATTRIBUTES WHICH HAVE A PAYOFF AND AN EXPECTED PROBABILITY OF          *)
(*  OCCURENCE.                                                             *)
(*  THE PURPOSE OF THE PROGRAM IS TO TEST THE ACCURACY AND EFFICIENCY OF   *)
(*  THE RULES.                                                             *)
(*                                                                         *)
(*  For a more complete description of the rules themselves, and the       *)
(*  formalisms involved, see the paper by Johnson and Payne.               *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)


CONST

    NMemCell = 8;  (* Number of separate items in STM. *)

    MaxAttributes = 12;  (* Max # of attributes of objects are rated on. *)
    TwoXMaxAttributes = 24;  (* 2 * MaxAttributes *)

    MaxAlternatives = 12;  (* Max # of alternatives considered. *)
    MaxList = 24;  (* max(MaxAlternatives,TwoXMaxAttributes) *)
    Combined = 144 ; (* The total number OF items IN the matrix *)

    CutOffVal = -1;  (* Result of GetCut *)

TYPE

(* * * * * * * * * * Types of the Short Term Memory. * * * * * * * * * * *)

        (*These are the markers identifying the operation
        which created this memory cell.  "Obscure" names
```

```
        include Momento for Remembers, Blood for GetCut,
        Finger for SetPoint, and Oversight for Forget*)

    OperResults = (Empty,Minimum,Maximum,Difference,Sum,Product,
                       Quotient,Momento,Blood,Squint,Oversight,TheNext,ThePrev,
                       Finger,AltAbsence,AttAbsence);

    MemLoc = 0..NMemCell;  {0 is a special 'non-existent' memory cell
                                used by TraceSTM}

    MemCell = RECORD  (* One "element" of decision maker's memory. *)

        (* Operation which created this cell *)
        OpType: OperResults;

        (* Attribute under consideration *)
        Att: -1..TwoXMaxAttributes;

        (* The alternative or cutoff we are remembering *)
        Alt: CutOffVal..MaxAlternatives;

        (* The value of this alternative in this att *)
        AValue: REAL;
    END;

(* * * * * * * * * * Types of the memory number lists  * * * * * * * * * * *)

    MemListType = (Altern,Attrib); (* Kinds of lists *)

    MemListRecord = RECORD

        Contents: ARRAY[0..MaxList]  OF INTEGER;
        TotContent,NContent,Pointer: INTEGER;

    END;

(* * * * * * * * * Types for keeping and passing program information * * * *)

    Vector = SUPER ARRAY[1..*] OF REAL;
    Map = SUPER ARRAY[1..*] OF INTEGER;

    AttVec = Vector (MaxAttributes);
    AttMap = Map (MaxAttributes);

    TwoXAttVec = Vector (TwoXMaxAttributes);

    VTable = ARRAY [1..MaxAlternatives] of TwoXAttVec;

    AltVec = Vector (MaxAlternatives);

    AltMap = Map (MaxAlternatives);

    CombVec = Vector (Combined);

    Rules = (Rand,AddUtility,TimedAU,EBA,LexiSemi,Lexi,
             EqualWeight,MCD,EBAEU,EBAMCD,Satisficing);

    DecChoiceArr = ARRAY[1..10] of INTEGER;  (* Type to hold first    *)
    (* 10 alts chosen for ea. rule.  For debugging, w/hdr var WinsOut.*)
```

```
VAR

    Risky: BOOLEAN; (* 1 = True, Risky.  2 = False, Not Risky.          *)
           (* Value not risky assigned in Decide for this version of  *)
           (* SIM.  In risky choice, each attribute has an indepen-   *)
           (* dent probability; for riskless choice, probabilities    *)
           (* are equivalent to importance weights and all the al-    *)
           (* ternatives have the same weight for a given attribute.  *)
    JPIdx:   REAL; (*  Calculated for any rule other than Rand.        *)
           (* In timed sim, timed AU rule is renamed TimedAU; original*)
           (* AU rule run automatically to provide value for calc.    *)
    MCVArr: ARRAY[1..3] of REAL; (*  Used for JPIndex calculation.     *)

    Sigma: VTable; (* Covariance array for correlated attributes.      *)
    SigmaX:VTable; (* Another covar array for correlated attributes.   *)
    Sigvalue:REAL; (* Input value read in from Sigfile for MakeSig     *)
                   (* in Random.Pas.  Used to implement correlated     *)
                   (* attribute value generation.                      *)

   (* For Timed Sim: *)

    (* Array to hold count to compare against TimeLimit for each run  *)
    (* of each rule.  Initialized before each run in Procedure DoARule*)
    (* in file Decrules.                                               *)
    Time: Array[Rules] of REAL;
    TimeUp: BOOLEAN; (* Throws rule into random choice at time limit. *)
    OpWtArr: Array[OperResults] of REAL; (* Holds weight values for   *)
    (* the operations used by the rules.  Values read into array in   *)
    (* SIM.PAS from WtsFile, a file supplied by user.                  *)

    (* Vars contain number of times a rule reaches the time limit;    *)
    (* values incremented in Decrules and passed to SIM.PAS to output:*)
    AAADummy: INTEGER;  (* Attempt to circumvent overflow problem.(?) *)
    EqWLimit,MCDLimit,EEULimit,ECDLimit: INTEGER;
    AdULimit,EBALimit,LexLimit,SatLimit:  INTEGER;
    LxLimit: INTEGER;

    BestChoice: ARRAY[Rules] of INTEGER; (* Holds a counter of # best *)
           (* choices made by a rule.  Incremented in Decrules;       *)
           (* initialized in Sim.Pas.                                 *)
    Proportion: REAL; (* A cell of BestChoice divided by the number   *)
           (* of runs, and output in Sim.Pas to TraceOut.             *)

    (* Vars to hold counters for certain Rules. Ex:keep track of      *)
    (* # times a decision was reached by rule vs. at random.          *)
    EBARulCnt,EBARndCnt: INTEGER;    (* EBA                           *)
    SatRulCnt,SatRndCnt: INTEGER;    (* Satisficing                   *)
    LSORulCnt,LSORndCnt: INTEGER;    (* LexiSemi                      *)
    MCDTieCnt:           INTEGER;    (* Majority Confirming Dimen's   *)

    AdUDec,EBADec,LexDec,LxDec : DecChoiceArr; (* One array for ea    *)
    EqWDec,MCDDec,EEUDec,ECDDec: DecChoiceArr; (* rule, to hold 1st   *)
    SatDec:                      DecChoiceArr; (* 10 alts chosen. Used*)
           (* with pgm header "WinsOut", for debugging.               *)

    Rule: Rules;
    O: OperResults;
    Used: ARRAY[Rules] of BOOLEAN; (* remember which rules used. *)
```

```
(* Raw data collection arrays *)

StdData: ARRAY[Rules] of
   RECORD
        DomValues:      Real;
        DomExist:       Real;
        ChoiceValues:   Real;
        ChoiceRanking:  Real;
        ChoiceUtils:    Real;
        ChoiceURank:    Real;
        OperationCount: ARRAY[OperResults] of Real;
   END;
MeanData: ARRAY[Rules] of
   RECORD
        DomValues:      Real;
        DomExist:       Real;
        ChoiceValues:   Real;
        ChoiceRanking:  Real;
        ChoiceUtils:    Real;
        ChoiceURank:    Real;
        OperationCount: ARRAY[OperResults] of Real;
   END;

Mean, StdDev: REAL;

STM: ARRAY[1..NMemCell] of MemCell; (* STM = simulated Short Term Memory *)

(* For each alternative, the attribute values on the left, and    *)
(* the corresponding attribute probabilities on the right.        *)
Values: VTable;

CutOffs: TwoXAttVec; (* Lowest att. value allowed, for some rules *)
DomValues:  AltVec;  (* 0 if not dominated; The value of the
                        dominating alternative otherwise     *)
ChoiceUtils:  AltVec;   (* the utilities of each alternative  *)
ChoiceValues: AltVec; (* The Additive Utility value for an altern. *)
ChoiceUProp:  AltVec;   (* the utilities of each alternative in comparison
                           to Au  *)
ChoiceProp:   Altvec; (* The Proportion of EV in comparision to AU *)
ChoicePro2:   AltVec; (* The Proportion of EV in comparison to AU
                           using a squared loss function. *)
ChoiceIndex:  AltVec; (* The Proportion of the difference of the
                           AU rule and the random choice. *)
ChoiceRanking: AltMap; (* The ordering of the choices. *)
ChoiceURank:   AltMap; (* The ordering using utility. *)

MemLists: ARRAY[MemListType] of MemListRecord;



AltAU: INTEGER;    (* Index of the AU rank.            *)
UAltAU: INTEGER;   (* Index of the AU rank.            *)
AltRand: INTEGER; (* Index of the Random choice rank.*)


i, j, k, l, m: INTEGER;       (* Indices used w/WinsOut pgm hdr.   *)
n, p, q, r, s: INTEGER;       (* O defined elsewhere; used w/opcts.*)

NRep:    INTEGER;             (* Number of replications.  Unused.  *)
```

```
    Noise:   REAL;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * **)
(* Input variables in the header of the program call.  Annotated.      *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * **)

(* A switch and a variable added; modification.                          *)
   RunSim: INTEGER; (* 1 = Generate gambles and run thru rules.         *)
                    (* 2 = Generate gambles and output without running*)
                    (*     rules or calculating statistics.           *)
   Debug: INTEGER; (* 0 = no writes to screen; fast.                    *)
                   (* 1 = display of all operators; slow.              *)
                   (* 2 = single step; press enter to continue.        *)
   NRun,NAlternatives,NAttributes: INTEGER;
     (* Number of runs, alternatives, & events presented to rules.     *)
   FmtPrt: BOOLEAN; (* 1 = T. Format output; incl. JPIndex.           *)
                    (* 0 = F. Condense output.                        *)
   WinsOut:BOOLEAN; (* 1 = T. When FmtPrt also T, outputs 10 alt wins.*)
                    (* 0 = F. No output.                              *)
   Traceout,Gamble,SigFile,WtsFile: TEXT;
    (* Traceout: output of cumulated statistics generated by         *)
    (*           all runs.  Can be formatted or condensed.           *)
    (* Gamble:   output of gambles presented to the rules.           *)
    (* SigFile:  Values between 1.0 and -1.0 as used in generation   *)
    (*           of correlated attributes when covariance is unequal.*)
    (* Values must be supplied in the following order:               *)
    (*    [Row1,Col2], [Row1,Col3], [Row1,Col4]...to last attribute  *)
    (*        .         [Row2,Col3], [Row2,Col4]...to last attribute  *)
    (*        .                      [Row3,Col4]...to last attribute  *)
    (*        . and so on for the number of rows.                     *)
    (* This fills the top 1/2 of the matrix diagonally, omitting the *)
    (* diagonal row of 1.0 values, which are supplied by the code.   *)
    (* If covariance is not selected, supply NUL as sigfile name.    *)
    (* WtsFile:  supplies weights for operators for TRLSIM.  External*)
    (*           file supplied by user.  Supply values in the order  *)
    (*           of the OperResults list.  Values read into OpWtArr in*)
    (*           SIM.PAS.  Array then accessed by each opr procedure *)
    (*           to get current wt value.  These values increment    *)
    (*           the "time used" by each rule in timed SIM.          *)

   Tran: INTEGER;  (* variable specifies type of random number      *)
                   (* generator: 1 = Thorngate                      *)
                   (*            2 = Uniform Hyperplane             *)
                   (*            3 = Dirichlet                      *)
                   (* Indication of Dominance or CoVar along with   *)
                   (* Tran determines how the Values matrix values  *)
                   (* are generated.  See START.PAS.                *)

   PG: INTEGER;    (* 3 ways to generate payoffs:                   *)
                   (* 1=No Dom. StochValues uses InitValues, but    *)
                   (* screens out dom; for risky, proc Dominance    *)
                   (* screens out dom in payoff/prob pair sets.     *)
                   (* 2=Dom. InitValues inits payoffs randomly.     *)
                   (* 3=Correlated Attributes.                      *)
   TimeLimit: REAL; (* For Timed Sim, sets "time" maximum for rules. *)
                   (* Rule operations increment a counter until this *)
                   (* maximum is reached, and then regular rule pro- *)
                   (* cessing halts and a random choice is made.    *)
   NumDomAltern: INTEGER;
                  (* Valid only when Dominance = TRUE.              *)
```

```
                    (* a. Enter 0 to disable the option and allow a     *)
                    (*    matrix of up to 8 x 8.                        *)
                    (* b. Enter an integer > 0 to specify the number of *)
                    (*    dominated alternatives to be in each set of   *)
                    (*    gambles generated.  Requires a 4 x 3 matrix.  *)

    CVEq: BOOLEAN;   (* 0=FALSE; 1=TRUE.  If PG indicates Correlated    *)
                     (* Attributes are to be used to generate payoffs,  *)
                     (* CVEq indicates equality or nonequality of CV.   *)
    Corr: REAL;      (* If CVEq is TRUE, give the correlation desired.  *)
    JND:  REAL;      (* Just Noticeable Difference value.  Used in      *)
                     (* riskless LexiSemi rule.                         *)

    DSeed: INTEGER4;(* RV seed *)
    Dist: REAL;      (* Entering a value of zero will give DIST the   *)
                     (* default value of 1000.0. See CutOffValue also.*)
                     (* See Start's Proc InitValues.                  *)
    CutOffValue: REAL;   (* The cutoff for gamble values in certain  *)
                (* rules, such as Satisficing. Entering a value of    *)
                (* zero will allow CutOffVal to default to the mean   *)
                (* of DIST.  See START's Proc Initialize.             *)
     RulesToRun: String(15);
                (* A string of characters where 'Y' means a rule is   *)
                (* to be run, corresponding to the rules in their     *)
                (* order as defined in the list of rules in this      *)
                (* section.                                           *)

     DomExist: REAL; (* if the set contains a dominated alternative *)
                (* Not used in current version.                      *)

     (* Count of times each oper is executed. *)
     OperationCount: ARRAY[OperResults] OF INTEGER;
     (* Not a program header in current version...*)

{$list-}

PROCEDURE Initialize;
(* Initialize all our values and derivatives of values for a particular *)
(* random setup. *)

BEGIN
END;
```

```
(*****************************************************************)
(* VERSION 1.2                                    TIMED RISKLESS SIM   *)
(* SIM.PAS                                        11/12/85             *)
(*****************************************************************)
(* Var "NumDomAltern" controls # dominated alterns/set.              *)
(* 6/6/85 - SIM divided into risky & riskless verisons;              *)
(*  Risky assigned at start of Decide instead of pgm hdr.            *)
(*  FmtPrt pgm hdr allows edited or consensed output.                *)
(* 9/27/85 - Method of deriving best choice value for JP Index calc  *)
(*  changed, since AU Rule under time pressure can't provide.        *)
(*****************************************************************)

{$LIST+}
{$include:'fformsuu.int'}              (* INCLUDES changed back to   *)
{$include:'screenvv.int'}              (* order and content of old   *)
{$include:'sim.int'}                   (* working version.           *)
{$include:'start.int'}
{$include:'decrules.int'}
{$include:'random.int'}

PROGRAM Decide(Input,Output,RunSim,DeBug,NRun,NAlternatives,NAttributes,
               FmtPrt,WinsOut,TraceOut,Gamble,SigFile,WtsFile,Tran,PG,
               TimeLimit,NumDomAltern,CVEq,Corr,JND,DSeed,Dist,CutOffValue,
               RulesToRun);

USES Sim,Random,decrules,fformsuu,screenvv;



(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* Main Program                                                          *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)



PROCEDURE InitData;

BEGIN

    rewrite (TraceOut);
    rewrite (Gamble);
    reset   (SigFile);
    reset   (WtsFile);

    (* Read "time/weights" values from WtsFile into array OpWtArr,    *)
    (* for later access by each operator procedure in LSTPRIMS and    *)
    (* STMPRIMS modules.  16 operators in all.  O=type OperResults.   *)
    FOR O := Empty to AttAbsence DO
        Read(WtsFile, OpWtArr[O]);


    (* Initialize array used to hold number of times a rule selects *)
    (* the best choice:                                             *)
    FOR Rule := Rand to Satisficing DO
       BestChoice[Rule] := 0;


    (* Initialize variables used to hold the reachedlimit values, and *)
```

```
    (* to pass them from the DECRULES module here for output:          *)
    EBALimit := 0;       (* EBA                        *)
    LexLimit := 0;       (* LexiSemi                   *)
    LxLimit  := 0;       (* Lexi                       *)
    EqWLimit := 0;       (* EqualWeight                *)
    MCDLimit := 0;       (* MCD                        *)
    EEULimit := 0;       (* EBAEU                      *)
    ECDLimit := 0;       (* EBAMCD                     *)
    SatLimit := 0;       (* Satisficing                *)
    AdULimit := 0;       (* AddUtility                 *)

    (* Counters output from selected rules:     *)
    EBARulCnt := 0; EBARndCnt :=0;     (* Elimination by Attributes *)
    SatRulCnt := 0; SatRndCnt :=0;     (* Satisficing               *)
    LSORulCnt := 0; LSORndCnt :=0;     (* LexicographicSemiOrder    *)
    MCDTieCnt := 0;                    (* Maj Confirming Dimensions *)

    (* Indices used to output 1st 10 altern choices, w/WinsOut:     *)
    j := 0;  k := 0; l := 0; m := 0; n := 0; p := 0; q := 0; r := 0;
    s := 0;

FOR Rule := Rand to Satisficing DO BEGIN
   MeanData[Rule].DomValues := 0;
   MeanData[Rule].DomExist := 0;
   MeanData[Rule].ChoiceValues := 0;
   MeanData[Rule].ChoiceRanking := 0;
   MeanData[Rule].ChoiceUtils := 0;
   MeanData[Rule].ChoiceURank := 0;

   FOR O:= Empty TO AttAbsence DO
   MeanData[Rule].OperationCount[O] := 0;


   StdData[Rule].DomValues := 0;
   StdData[Rule].DomExist := 0;
   StdData[Rule].ChoiceValues := 0;
   StdData[Rule].ChoiceRanking := 0;
   StdData[Rule].ChoiceUtils :=0;
   StdData[Rule].ChoiceURank := 0;

   FOR O:= Empty TO AttAbsence DO
   StdData[Rule].OperationCount[O] := 0;

   Used[Rule] := FALSE {Remember that we used this one}

 END; {FOR}

        END;   {InitData}


(*************************************************************)
(*  Decide                                                   *)
(*************************************************************)

BEGIN (* Decide *)


(* Assign risk mode for riskless:                             *)
Risky := FALSE;
```

```
{Code for initialization of display}

clsvv;                                  {Clear the screen at startup}

cursetvv(1,26);
Write(output,'Timed Riskless DecisionLab V2.0');
Cursetvv(4,7);
Write('Number of Alternatives: ',NAlternatives:3);
Cursetvv(4,42);
write('Number of Attributes:   ',NAttributes:3);
Cursetvv(5,7);
write('Number of Trials:       ',NRun:3);
Cursetvv(5,42);
write('Time Limit:               ',TimeLimit:3:0);
Cursetvv(6,7);
write('Distribution:             ',Dist:4:0);
Cursetvv(6,42);
write('Cut Off Value:            ',CutOffValue:4:0);
Cursetvv(7,7);
write('Tran: ');
Cursetvv(7,13);
If Tran=1 then write('Thorngate');
If Tran=2 then write('Uniform Hyperplane');
If Tran=3 then write('Dirichlet');
Cursetvv(7,42);
write('PG: ');
Cursetvv(7,46);
If PG=1   then write('No Dominance');
If PG=2   then write('Dominance');
If PG=3   then write('Correlated Attributes');
Cursetvv(8,34);
write('JND: ',JND:4:3);



cursetvv(10,0);
for j := 1 to 80 do
Write(chr(205));
Cursetvv(24,0);
attribvv(0,7,' ',80);
    (* Set up raw data arrays *)

cursetvv(24,0);
write(output,'Starting...');

InitData;    (* Perform the initializing procedure above. *)

    IF FmtPrt = TRUE THEN
       Writeln (Gamble, 'Tran = ', Tran:2, '     ', 'PG = ', PG:2);

    FOR I:= 1 TO NRun DO BEGIN
        cursetvv(24,0);
        write(output,'
        cursetvv(24,0);
        write(output,'Trial Number: ',i:3);
        cursetvv(24,40);
        write('Generating Randoms');


        Initialize;
```

```
        (**** Condition: run rules or just generate gambles:  ****)
        IF RunSim = 1 THEN
        (**** Riskless rules selected by user via pgm header: ****)
        BEGIN
            IF RulesToRun[1]  = 'Y' THEN DoARule(Rand);
(* Run AddUtility -- no time pressure -- for baseline: *)
            DoARule(AddUtility);
            IF RulesToRun[2]  = 'Y' THEN DoARule(TimedAU);
            IF RulesToRun[3]  = 'Y' THEN DoARule(EBA);
            IF RulesToRun[4]  = 'Y' THEN DoARule(LexiSemi);
            IF RulesToRun[5]  = 'Y' THEN DoARule(Lexi);
            IF RulesToRun[6]  = 'Y' THEN DoARule(EqualWeight);
            IF RulesToRun[7]  = 'Y' THEN DoARule(MCD);
            IF RulesToRun[8]  = 'Y' THEN DoARule(EBAEU);
            IF RulesToRun[9]  = 'Y' THEN DoARule(EBAMCD);
            IF RulesToRun[10] = 'Y' THEN DoARule(Satisficing);
        END; (* IF *)
    END; (* FOR *)

    (* Write out the data array *)
    IF RunSim = 1 THEN
    BEGIN

    cursetvv(24,40);
    WRITE  (Output,'Calculating Statistics...');
    IF FmtPrt = TRUE THEN
    BEGIN
        WRITELN(TraceOut,'  ');
        WRITELN(TraceOut,'  ');
        WRITELN(TraceOut,'TIMED RISKLESS TRACEOUT FILE:');
        WRITELN(TraceOut,'  ');
        WRITELN(TraceOut,'-----------------------------------',
        '-----------------------------------');
    END;

    FOR Rule := Rand TO Satisficing DO
      IF Used[Rule] THEN
      BEGIN
        IF FmtPrt = TRUE THEN  (* for formatted output *)
(* Notice that rule 'numbers' have changed...with the addition of *)
(* the timed AU rule, in addition to regular AU rule.             *)
        BEGIN
        Writeln(TraceOut, '  ');
        CASE Rule OF
          Rand:        Write (TraceOut, '0 RAND                ');
          AddUtility:  Write (TraceOut, '1 ADDUTILITY          ');
          TimedAu:     Write (TraceOut, '2 TIMED ADDUTILITY    ');
          EBA:         Write (TraceOut, '3 EBA                 ');
          LexiSemi:    Write (TraceOut, '4 LEXISEMI            ');
          Lexi:        Write (TraceOut, '5 LEXI                ');
          EqualWeight: Write (TraceOut, '6 EQUALWEIGHT         ');
          MCD:         Write (TraceOut, '7 MCD                 ');
          EBAEU:       Write (TraceOut, '8 EBAEU               ');
          EBAMCD:      Write (TraceOut, '9 EBAMCD              ');
          Satisficing: Write (TraceOut, '10 SATISFICING        ');
        END; (* Case *)
        END; (* IF FmtPrt *)

       WITH MeanData[Rule] DO
          BEGIN
```

```
    IF FmtPrt = TRUE THEN  (* for formatted output *)
    BEGIN
      WRITELN(TraceOut,'Runs: ', NRun:2);
      WRITE  (TraceOut, 'Tran: ');
      IF (Tran=1) THEN
         WRITE(TraceOut, 'THORN           ') ELSE
      IF (Tran=2) THEN
         WRITE(TraceOut, 'UH              ') ELSE
      (* (Tran=3) *)
         WRITE(TraceOut, 'DIRICH          ');
      WRITE  (TraceOut, 'PG: ');
      IF (PG=1) THEN
         WRITELN(TraceOut, 'No Dominance') ELSE
      IF (PG=2) THEN
         WRITELN(TraceOut, 'Dominance');
      IF (PG=3) THEN
          IF CVEq = FALSE THEN
             WRITELN(TraceOut,
             'Correlated Attributes (unequal)')
          ELSE
             WRITELN(TraceOut,'Correlated Attributes (equal)',
                '  Corr = ', Corr:2:2);
      WRITELN(TraceOut,'Alt X Att: ',NAlternatives:2,
              ' X ',NAttributes:2, '    ',
              'JND: ',JND:4:4);
      WRITELN(TraceOut,'Dist = ',Dist:4:0, '          ',
              'CutOffValue = ',CutOffValue:4:0);
      WRITELN(TraceOut,'Time Limit = ',TimeLimit:5:0);
      WRITELN(TraceOut,'-----------------------------------',
            '-----------------------------------');
      WRITELN(TraceOut,'                          ','MEAN: ',
            '         ','STDDEV:');
    END  (* IF FmtPrt *)

    ELSE   (* for condensed output *)
      Write(TraceOut, Ord(Rule):2, NAlternatives:2,
            NAttributes:2, NRun:4, Tran:3,
            PG:3,' ', JND:6:4, Dist:6:0, CutOffValue:5:0,
            TimeLimit:5:0);

StdDev:=(SQRT(StdData[Rule].DomValues -(SQR(DomValues) / NRun))
    / (NRun - 1));
    Mean := DomValues/NRun;
    IF FmtPrt = TRUE THEN   (* for formatted output *)
    BEGIN
      WRITELN(TraceOut,'DOMVALUES:        ',Mean: 8:3,
              '       ', StdDev: 8:3);
    END
    ELSE  (* for condensed output *)
      Write(TraceOut, Mean: 10:3, StdDev: 10:3);

  StdDev:=(SQRT(StdData[Rule].DomExist -(SQR(DomExist) / NRun))
   / (NRun - 1));
  Mean := DomExist/NRun;
  If FmtPrt = TRUE THEN  (* for formatted output *)
  BEGIN
    WRITELN(TraceOut,'DOMEXIST:         ', Mean: 8:3,
            '       ', StdDev: 8:3);
  END
```

```
          ELSE  (* for condensed output *)
            Write(TraceOut, Mean: 10:3, StdDev: 10:3);

   StdDev:=(SQRT(StdData[Rule].ChoiceValues -(SQR(ChoiceValues) / NRun))
           / (NRun - 1));
           Mean := ChoiceValues/NRun;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* Put mean ChoiceValues in array MCVArr to use in calculating the   *)
(* JPIndex. The AddUtility Rule is run automatically, and its value  *)
(* used here as baseline; timed AU rule run separately as TimedAU,   *)
(* since its values are skewed under time pressure.  JPIndex calc'd  *)
(* here only if Random rule is also run; both random and au needed.  *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

           IF Rule = RAND       THEN
             MCVArr[1] := Mean
           ELSE
             IF Rule = AddUtility THEN
                MCVArr[2] := Mean
           ELSE
             MCVArr[3] := Mean; (* New value for ea iteration. *)

           IF FmtPrt = TRUE THEN  (* for formatted output *)
           BEGIN
             WRITELN(TraceOut,'CHOICEVALUES:     ', Mean: 8:3,
                     '       ', StdDev: 8:3);
           END
           ELSE  (* for condensed output *)
             Write(TraceOut, Mean: 10:3, StdDev: 10:3);

 StdDev:=(SQRT(StdData[Rule].ChoiceRanking -(SQR(ChoiceRanking) / NRun))
           / (NRun - 1));
           Mean := ChoiceRanking/NRun;
           IF FmtPrt = TRUE THEN  (* for formatted output *)
           BEGIN
             WRITELN(TraceOut,'CHOICERANKING:    ', Mean: 8:3,
                     '       ', StdDev: 8:3);
           END
           ELSE  (* for condensed output *)
             WRITE(TraceOut, Mean: 10:3, StdDev: 10:3);

     StdDev:=(SQRT(StdData[Rule].ChoiceUtils -(SQR(ChoiceUtils) / NRun))
           / (NRun - 1));
           Mean := ChoiceUtils/NRun;
           IF FmtPrt = TRUE THEN  (* for formatted output *)
           BEGIN
             WRITELN(TraceOut,'CHOICEUTILS:      ', Mean: 8:3,
                     '       ', StdDev: 8:3);
           END
           ELSE  (* for condensed output *)
             WRITE(TraceOut, Mean: 10:3, StdDev: 10:3);

     StdDev:=(SQRT(StdData[Rule].ChoiceURank -(SQR(ChoiceURank) / NRun))
           / (NRun - 1));
           Mean := ChoiceURank/NRun;
           IF FmtPrt = TRUE THEN  (* for formatted output *)
           BEGIN
             WRITELN(TraceOut,'CHOICEURANK:      ', Mean: 8:3,
                     '       ', StdDev: 8:3);
```

```
            WRITELN(TraceOut, 'Operation:');
          END
          ELSE  (* for condensed output *)
            WRITE(TraceOut, Mean: 10:3, StdDev: 10:3);

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*            Output the OperationCount Totals:                          *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
          FOR O := Empty TO AttAbsence DO
          BEGIN
             IF FmtPrt = TRUE THEN
             BEGIN
                CASE O OF
                   Empty:      Write(TraceOut,'Empty              ');
                   Minimum:    Write(TraceOut,'Minimum            ');
                   Maximum:    Write(TraceOut,'Maximum            ');
                   Difference: Write(TraceOut,'Difference         ');
                   Sum:        Write(TraceOut,'Sum                ');
                   Product:    Write(TraceOut,'Product            ');
                   Quotient:   Write(TraceOut,'Quotient           ');
                   Momento:    Write(TraceOut,'Momento (Read)     ');
                   Blood:      Write(TraceOut,'Blood (GetCutOff)  ');
                   Squint:     Write(TraceOut,'Squint (GetJND)    ');
                   Oversight:  Write(TraceOut,'Oversight (Forget)');
                   TheNext:    Write(TraceOut,'TheNext            ');
                   ThePrev:    Write(TraceOut,'ThePrev            ');
                   Finger:     Write(TraceOut,'Finger (SetPoint)  ');
                   AltAbsence: Write(TraceOut,'Absence (Alt Elim)');
                   AttAbsence: Write(TraceOut,'Absence (Att Elim)');
                END; (* Case *)
             END; (* IF FmtPrt *)

              StdDev:=(SQRT(ABS(StdData[Rule].OperationCount[O] -
              (SQR(OperationCount[O]) / NRun))) / (NRun - 1));
              Mean := OperationCount[O] /NRun;
              IF FmtPrt = TRUE THEN    (* for formatted output  *)
              BEGIN
                WRITELN(TraceOut, Mean: 8:3,
                        '       ', StdDev: 8:3);
              END

              ELSE  (* for condensed output *)
                WRITE (TraceOut, Mean: 10:3, StdDev: 10:3);


          END;  (* FOR O *)
              IF FmtPrt = TRUE THEN    (* for formatted output *)
              BEGIN
                WRITELN(TraceOut, '-----------------------------------',
                        '----------------------------------------');
              END;

       END; (* WITH *)

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* Calculate and output the JPIndex for any rule if Random has been  *)
(* run. The AU (ie, best choice) value is derived from the AU rule,  *)
(* run WITHOUT time pressure, (separate from AU under time pressure, *)
(* which is called Rule "TimedAU").  A period not a value is written *)
(* for Rand (SAS missing data marker) to facilitate SAS analysis of  *)
```

```
(* the condensed output data.                                          *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
        IF (MCVArr[1]>0.0) AND (MCVArr[2]>0.0)
           AND (MCVArr[3]>0.0) AND
           ((MCVArr[2]-MCVArr[1])>0) THEN
              JPIdx:=((MCVArr[3]-MCVArr[1])/(MCVArr[2]-MCVArr[1]));
        IF (FmtPrt = TRUE) AND (Rule <> Rand)
                           AND (Rule <> AddUtility) THEN
        BEGIN
           IF ((MCVArr[2]-MCVArr[1]) <= 0) THEN
              Writeln(TraceOut,
                      'Divide by zero precludes JP Index calculation.')
           ELSE
              Writeln(TraceOut, 'JP Index = ', JPIdx:7:2);
        END;


        IF (FmtPrt = FALSE) THEN
        BEGIN
           IF (Rule = RAND)
              OR (Rule = AddUtility) THEN
              Write(TraceOut, '    .    ');
           IF (Rule <> Rand) AND (Rule <> AddUtility) THEN
           BEGIN
              IF ((MCVArr[2]-MCVArr[1]) > 0) THEN
                 Write(Traceout, JPIdx:7:2)
              ELSE
                 Write(Traceout, '    .    ');
           END;
        END;

 (* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
 (* Calculate the proportion of times a rule picks the best alterna-  *)
 (* tive.  A counter has been kept in the array BestChoice for each   *)
 (* rule, incremented in DecRules.                                    *)
 (* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

        Proportion := BestChoice[Rule];

        Proportion := Proportion / NRun;

        IF (FmtPrt = TRUE) THEN
           Writeln(TraceOut, 'Proportion of correct choices = ',
                    Proportion:4:2);
        IF (FmtPrt = FALSE) THEN
        BEGIN
           Write  (TraceOut, '   ');
           Write  (TraceOut, Proportion:4:2);
        END;


(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* Output certain debugging & other information for rules:           *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

  IF ((FmtPrt=TRUE) AND (Rule=Rand)) THEN
     Writeln(TraceOut, ' ');
  IF ((FmtPrt=FALSE) AND (Rule=Rand)) THEN
  BEGIN
     Write  (TraceOut,'     ');
```

```
   Writeln(TraceOut,'   .   ');
END;

IF ((FmtPrt=TRUE) AND (Rule=AddUtility)) THEN
BEGIN

    Write  (TraceOut,'*** Run without a time limit as a baseline.');
    Writeln(TraceOut, ' ');
    WRITELN(TraceOut, '-----------------------------------',
            '-----------------------------------');
    Writeln(TraceOut,' ');
 END;

 IF (FmtPrt=FALSE) AND (Rule=AddUtility) THEN
    Writeln(TraceOut,'    .    ');



 IF ((FmtPrt=TRUE) AND (Rule=TimedAU)) THEN
 BEGIN

    Write  (TraceOut,'Number of trials reaching time limit: ');
    Writeln(TraceOut, AdULimit:6);
    IF (WinsOut=TRUE) THEN
    BEGIN
       Write (TraceOut, 'Choices for Timed AU: ');
       FOR j := 1 to 10 DO
           Write (TraceOut, AdUDec[j]:2);
    END;
    Writeln(TraceOut, ' ');
    WRITELN(TraceOut, '-----------------------------------',
            '-----------------------------------');
    Writeln(TraceOut,' ');
 END;

 IF (FmtPrt=FALSE) AND (Rule=TimedAU) THEN
 BEGIN
    Write  (TraceOut,'    ');
    Writeln(TraceOut, AdULimit:6);
 END;



 IF ((FmtPrt=TRUE) AND (Rule = EBA)) THEN
 BEGIN
    Write  (Traceout,'Number of Picks by Rule = ',EBARulCnt:5,
            '     ');
    Writeln(Traceout,'Number of Random Picks = ',EBARndCnt:5);
    Write  (TraceOut, 'Number of trials reaching time limit = ');
    Writeln(TraceOut, EBALimit:6);
    IF WinsOut=TRUE THEN
    BEGIN
       Write (TraceOut,'Choices for EBA: ');
       FOR j := 1 to 10 DO
          WRITE (TraceOut, EBADec[j]:2);
    END;
    Writeln(TraceOut,' ');
    WRITELN(TraceOut, '-----------------------------------',
            '-----------------------------------');
    Writeln(Traceout,' ');
```

```
END;

IF (FmtPrt=FALSE) AND (Rule=EBA) THEN
BEGIN
   Write  (TraceOut, '    ');
   Writeln(TraceOut, EBALimit:6);
END;



IF ((FmtPrt=TRUE) AND (Rule = LexiSemi)) THEN
BEGIN
   Write  (Traceout,'Number of Picks by Rule = ',LSORulCnt:5,
           '     ');
   Writeln(Traceout,'Number of Random Picks = ',LSORndCnt:5);
   Write  (TraceOut, 'Number of trials reaching time limit = ');
   Writeln(TraceOut, LexLimit:6);
   IF WinsOut=TRUE THEN
   BEGIN
      Write (TraceOut,'Choices for LexiSemi: ');
      FOR j := 1 to 10 DO
         Write (TraceOut, LexDec[j]:2);
   END;
   Writeln(TraceOut,' ');
   WRITELN(TraceOut, '----------------------------------',
           '----------------------------------------');
   Writeln(Traceout,' ');
END;
IF (FmtPrt=FALSE) AND (Rule=LexiSemi) THEN
BEGIN
   Write  (TraceOut, '    ');
   Writeln(TraceOut, LexLimit:6);
END;



IF ((FmtPrt=TRUE) AND (Rule = Lexi)) THEN
BEGIN
   Write  (TraceOut, 'Number of trials reaching time limit = ');
   Writeln(TraceOut, LxLimit:6);
   IF WinsOut=TRUE THEN
   BEGIN
      Write (TraceOut,'Choices for Lexi: ');
      FOR s := 1 to 10 DO
         Write (TraceOut, LxDec[s]:2);
   END;
   Writeln(TraceOut,' ');
   WRITELN(TraceOut, '----------------------------------',
           '----------------------------------------');
   Writeln(Traceout,' ');
END;
IF (FmtPrt=FALSE) AND (Rule=Lexi) THEN
BEGIN
   Write  (TraceOut, '   ');
   Writeln(TraceOut, LxLimit:6);
END;



IF ((FmtPrt=TRUE) AND (Rule=EqualWeight)) THEN
```

```
BEGIN
   Write  (TraceOut, 'Number of trials reaching time limit = ');
   Writeln(TraceOut, EqWLimit:6);
   IF (WinsOut=TRUE) THEN
   BEGIN
      Write (TraceOut, 'Choices for EqualWeight: ');
      FOR j := 1 to 10 DO
          Write (TraceOut, EqWDec[j]:2);
   END;
   Writeln(TraceOut,' ');
   WRITELN(TraceOut, '-----------------------------------',
           '-----------------------------------');
   Writeln(Traceout,' ');
END;

IF (FmtPrt=FALSE) AND (Rule=EqualWeight) THEN
BEGIN
   Write  (TraceOut, '   ');
   Writeln(TraceOut, EqWLimit:6);
END;



IF ((FmtPrt=TRUE) AND (Rule = MCD)) THEN
BEGIN
   Writeln(TraceOut, 'Number of Ties between Alternatives in ',
          NRun:3, ' Runs = ',MCDTieCnt:3);
   Write  (TraceOut, 'Number of trials reaching time limit = ');
   Writeln(TraceOut, MCDLimit:6);
   IF WinsOut=TRUE THEN
   BEGIN
      Write (TraceOut, 'Choices for MCD: ');
      FOR j := 1 to 10 DO
         Write (TraceOut, MCDDec[j]:2);
   END;
   Writeln(Traceout,' ');
   WRITELN(TraceOut, '-----------------------------------',
           '-----------------------------------');
   Writeln(Traceout,' ');
END;

IF (FmtPrt=FALSE) AND (Rule=MCD) THEN
BEGIN
   Write  (TraceOut, '   ');
   Writeln(TraceOut, MCDLimit:6);
END;



IF ((FmtPrt=TRUE) AND (Rule=EBAEU)) THEN
BEGIN
   Write  (TraceOut, 'Number of trials reaching time limit = ');
   Writeln(TraceOut, EEULimit:6);
   IF (WinsOut=TRUE) THEN
   BEGIN
      Writeln(TraceOut,'When number of alternatives reaches < 3,');
      Writeln(TraceOut,'AddUtility rule entered; choice = 0.');
      Write (TraceOut, 'Choices for EBAEU: ');
      FOR j := 1 to 10 DO
          Write (TraceOut, EEUDec[j]:2);
```

```
   END;
   Writeln(TraceOut,' ');
   WRITELN(TraceOut, '-----------------------------------',
           '-----------------------------------');
   Writeln(Traceout,' ');
END;

IF (FmtPrt=FALSE) and (Rule=EBAEU) THEN
BEGIN
   Write  (TraceOut, '  ');
   Writeln(TraceOut, EEULimit:6);
END;



IF ((FmtPrt=TRUE) AND (Rule=EBAMCD)) THEN
BEGIN
   Write  (TraceOut, 'Number of trials reaching time limit = ');
   Writeln(TraceOut, ECDLimit:6);
   IF (WinsOut=TRUE) THEN
   BEGIN
      Writeln(TraceOut,'When number of alternatives reaches 3,');
      Writeln(TraceOut,'MCD rule entered; choice = 0.');
      Write (TraceOut, 'Choices for EBAMCD: ');
      FOR j := 1 to 10 DO
          Write (TraceOut, ECDDec[j]:2);
   END;
   Writeln(TraceOut,' ');
   WRITELN(TraceOut, '-----------------------------------',
           '-----------------------------------');
   Writeln(Traceout,' ');
END;

IF (FmtPrt=FALSE) AND (Rule=EBAMCD) THEN
BEGIN
   Write  (TraceOut, '  ');
   Writeln(TraceOut, ECDLimit:6);
END;



IF ((FmtPrt=TRUE) AND (Rule = Satisficing)) THEN
BEGIN
   Write  (Traceout,'Number of Picks by Rule = ',SatRulCnt:5,
           '     ');
   Writeln(Traceout,'Number of Random Picks = ',SatRndCnt:5);
   Write  (TraceOut, 'Number of trials reaching time limit = ');
   Writeln(TraceOut, SatLimit:6);
   IF WinsOut=TRUE THEN
   BEGIN
      Write (TraceOut,'Choices for Satisficing: ');
      For j := 1 to 10 DO
         Write (Traceout, SatDec[j]:2);
   END;
   Writeln(Traceout,' ');
   WRITELN(TraceOut, '-----------------------------------',
           '-----------------------------------');
   Writeln(Traceout,' ');
END;
```

```
    IF (FmtPrt=FALSE) AND (Rule=Satisficing) THEN
    BEGIN
       Write   (TraceOut, '    ');
       Writeln(TraceOut, SatLimit:6);
    END;



    END; (* If Used[Rule] *)



END; (* IF RunSim... *)

END. (* Decide *)
```

```
{$linesize:130}
INTERFACE;
UNIT Start(Initialize);
{$list-}

PROCEDURE Initialize;
(* Initialize all our values and derivatives of values for a particular *)
(* random setup. *)


END;
```

```
(*****************************************************************)
(* VERSION 1.2                                        SIM RISKLESS    *)
(* START.PAS                                          2/11/86         *)
(*****************************************************************)

{$list+}
{$include:'sim.int'}
{$include:'numprims.int'}
{$include:'StmPrims.int'}
{$include:'Random.int'}
{$include:'fformsuu.int'}
{$include:'screenvv.int'}


IMPLEMENTATION OF Sim;
USES NumPrims,StmPrims,Random,fformsuu,screenvv;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* Modifications:    7/84 - Add SetUpDominance routines.              *)
(*                  8/10/84 - Add code to allow Tran0,1 without a    *)
(*      specific number of dominated alternatives.                   *)
(*                  8/20/84 - Debug & simplify initialize code to    *)
(*      produce good gamble data for PROCESS1 program.               *)
(*                 10/30/84 - In Proc InitProbabilities, add another *)
(*      option (Tran=99) to allow generation of weights by calling   *)
(*      a new procedure in Random: Dirichlet.                        *)
(* 6/21/85 - Streamlined code; added Corr. Att.  Tran now 1, 2, or   *)
(* 3 for prob generation.  PG (PayOffGenerator) designates dom, no   *)
(* dom, or corr. att. to generate payoffs.                           *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*  START.PAS. (Initialization)     Procedure SetUpDominance added.*)
(*  Generates a specific # of dom alterns; requires 3 attributes x *)
(*  4 alterns.  Vars RunSim & NumDomAltern added to hdr/globals in *)
(*  SIM.  If NumDomAltern=0 then for Tran=0 or 1, dom occurs ran-  *)
(*  domly, and the # of attributes & alternatives can be up to 8x8.*)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

PROCEDURE Initialize;

    VAR AttNum,AltNum: INTEGER;
        InvMap,InvUMap: AltMap;
        OperIndx :  OperResults;
        SetHasDominance: BOOLEAN;          (* added *)

(*****************************************************************)
(* PROCEDURE SetUpDominance: generates a specific # of dom*)
(* alterns/set of generated probabilities & payoffs for   *)
(* tran=0 or 1. NAttributes must = 3.                              *)
(*****************************************************************)

PROCEDURE SetUpDominance;

CONST
    Lo = 1;        (* indexes to array PayOffMatrix and to *)
   Mid = 2;        (* corresponding ProbMatrix array cells.*)
    Hi = 3;

TYPE
    PairMatrix = Array[1..66,1..2] of INTEGER;
    (* Holds results of FindPairs Procedure.  Maximum #   *)
```

```
    (* of pairs w/12 alternatives is 66.                    *)
    PayProbMatrix = Array[1..12,1..3] of REAL;
    RealArr = Array[1..3] of REAL;
    SixCellArr = Vector(6);
    (* TwoSetsPayOffs of type SixCellArr is passed to      *)
    (* Vsrta sort, which expects a super array param.      *)
    (* This is super arr (real) Vector w/upper bound=6.    *)
    DomArr = Map(MaxAlternatives);
    (* DomArr must be able to hold up to MaxAltern. # of   *)
    (* cells.  Map is a super array of integer.            *)

VAR

    AltNum, AttNum: INTEGER;   (* indexes in FOR stmts.    *)
    Pairs: PairMatrix;
    (* Holds all unique pairs of alternatives.            *)
    PayOffMatrix, ProbMatrix: PayProbMatrix;
    (* Arrays which receive values from "Values" matrix.  *)
    TempPayArr, TempProbArr: RealArr;
    (* Temporary holding arrays, passed to sort routine.  *)
    DomSwitch: DomArr;
    (* Holds count of # altern in a set which show dom.   *)
    NbrPrs: INTEGER;
    (* Number of unique prs found in procedure FindPairs. *)
    AltA, AltB: INTEGER;
    (* Alternative numbers of 1 pair of alternatives.     *)
    PairSet: INTEGER;
    (* Index to Pairs array.                              *)
    TwoSetsPayOffs: SixCellArr;
    (* Payoffs from a 3-attribute pair of gambles.        *)
    CumAlt, CumAltA, CumAltB: SixCellArr;
    (* Holding arrays for comparisons.                    *)
    Alt, Sw, i, j: INTEGER;
    DomAlternCounter: INTEGER;
    (* Counts # switches set in DomSwitch Array; ie, the  *)
    (* # dominated alternatives in a set.                 *)


(***************** Nested Procedure ***********************)
PROCEDURE ModVsrtr(VAR A:RealArr; la:INTEGER; VAR ir:RealArr);
(* Modification of Vsrtr sort. Allows contents of mapped  *)
(* "ir" array to be real.  Here, ir holds probabilities.  *)

VAR
    jmp, m, n: INTEGER;
  tmp, tindex: REAL;
       sorted: BOOLEAN;

BEGIN (* ModVsrtr *)
    jmp := la;
    WHILE jmp > 1 DO
    BEGIN
        jmp := jmp DIV 2;
        REPEAT
            sorted := TRUE;
            FOR m := 1 to la - jmp DO
            BEGIN
                n := m + jmp;
                IF A[m] > A[n] THEN
                BEGIN
```

```
                    tmp := A[m];  tindex := ir[m];
                   A[m] := A[n];   ir[m] := ir[n];
                   A[n] := tmp;    ir[n] := tindex;
                   sorted := FALSE;
                END; (* IF *)
            END; (* FOR *)
        UNTIL sorted;
    END; (* WHILE *)
END; (* ModVsrtr *)


(***************** Nested Procedure ***********************)
PROCEDURE FindPairs;
(* Finds all unique pairs of alternatives and puts them   *)
(* in Pairs matrix.  NbrPrs holds the total # unique prs. *)

VAR
     Sw, x, y, i, j: INTEGER;

BEGIN (* FindPairs *)
    y  := 2;
    i  := 0;
    j  := 1;
    Sw := 0;
NbrPrs := 0;      (* initialize counter *)

    FOR x := 1 to (NAlternatives - 1) DO
    BEGIN
        IF Sw = 1
            THEN y := (x + 1);
        WHILE y <= NAlternatives DO
        BEGIN
            i := i + 1;
            Pairs[i,j] := x;
            j := j + 1;
            Pairs[i,j] := y;
            NbrPrs := NbrPrs + 1;    (* increment counter *)
            j := 1;
            y := y + 1;
        END; (* WHILE *)
        Sw := 1;
    END; (* FOR *)
END; (* FindPairs *)


(*************************************************************)
(*  SetUpDominance                                           *)
(*************************************************************)

BEGIN  (* SetUpDominance *)

    For AltNum := 1 to NAlternatives DO
    BEGIN
        DomSwitch[AltNum] := 0;    (* init switch array.    *)
    (* MATRIX ROW TO ARRAY: Put a row from Value matrix    *)
    (* into 2 temp arrays, to send to sort procedure.      *)
    For AttNum := 1 to NAttributes DO
    BEGIN
        TempPayArr[AttNum] := Values[AltNum,AttNum];
        TempProbArr[AttNum] := Values[AltNum,AttNum+NAttributes];
```

```
END; (* FOR *)

(* SORT ARRAYS: Sort payoffs lo to hi, keeping          *)
(* corresponding probability array in payoff order.     *)
ModVsrtr(TempPayArr,NAttributes,TempProbArr);

(* ARRAYS TO MATRIX ROWS: Put sorted temp arrays into   *)
(* the current AltNum row of payoff/prob matrices.      *)
For AttNum := 1 to NAttributes DO
BEGIN
    PayOffMatrix[AltNum,AttNum] := TempPayArr[AttNum];
    ProbMatrix[AltNum,AttNum]   := TempProbArr[AttNum];
END;
END; (* FOR *)

(* Find all the possible unique pairs of alternatives.*)
FindPairs;

(* Second big loop of SetUpDominance begins here. Ea. *)
(* pair in the set is tested for dominance.           *)
FOR PairSet := 1 to NbrPrs DO
BEGIN
    AltA := Pairs[PairSet,1];
    AltB := Pairs[PairSet,2];

(* Combine the 3 payoffs of ea. altern into 1 array. *)
    j := 1;
    FOR i := 1 to 3 DO
    BEGIN
        TwoSetsPayOffs[i] := PayOffMatrix[AltA,j];
        j := j + 1;
    END; (* FOR *)
    j := 1;
    FOR i := 4 to 6 DO
    BEGIN
        TwoSetsPayOffs[i] := PayOffMatrix[AltB,j];
        j := j + 1;
    END; (* FOR *)
    (* End "Combine the 3..." *)

(* Sort the 6-cell array of payoffs from low to high. *)
Vsrta(TwoSetsPayOffs,6);

(* Run through the CUM rules, comparing each value in *)
(* the 6-cell ordered array vs. Lo, Mid, and Hi values*)
(* in the PayOffMatrix for both alternatives of the   *)
(* pair being considered.  Based on comparisons, enter*)
(* values in 2 new 6-cell arrays: CumAltA & CumAltB.  *)
Sw := 0;
For Alt := AltA to AltB do
BEGIN
    FOR i := 1 to 6 DO
    BEGIN
        (************ CUM RULES *****************)
        IF TwoSetsPayOffs[i] < PayOffMatrix[Alt,Lo]
           THEN CumAlt[i] := 0;
        IF TwoSetsPayOffs[i] >= PayOffMatrix[Alt,Lo] THEN
           IF TwoSetsPayOffs[i] < PayOffMatrix[Alt,Mid]
              THEN CumAlt[i] := ProbMatrix[Alt,Lo];
        IF TwoSetsPayOffs[i] >= PayOffMatrix[Alt,Mid] THEN
```

```
            IF TwoSetsPayOffs[i] < PayOffMatrix[Alt,Hi]
               THEN CumAlt[i] := (ProbMatrix[Alt,Lo] +
                                  ProbMatrix[Alt,Mid]);
         IF TwoSetsPayOffs[i] >= PayOffMatrix[Alt,Hi]
            THEN CumAlt[i] := 1;
      END; (* FOR *)

      IF Sw = 0 THEN              (* 1st pass: put values derived *)
         CumAltA := CumAlt        (* from AltA into array CumAltA.*)
      ELSE
         CumAltB := CumAlt;       (* 2nd pass: put values derived *)
      Sw := 1;                    (* from AltB into array CumAltB.*)
   END; (* FOR *)

   (* Compare the contents of the 6-cell arrays of cumu- *)
   (* lative probabilities to each other; set switches.  *)

   IF CumAltA[1] >= CumAltB[1] THEN
      IF CumAltA[2] >= CumAltB[2] THEN
         IF CumAltA[3] >= CumAltB[3] THEN
            IF CumAltA[4] >= CumAltB[4] THEN
               IF CumAltA[5] >= CumAltB[5] THEN
                  IF CumAltA[6] >= CumAltB[6] THEN
                     DomSwitch[AltA] := 1;
   IF CumAltA[1] <= CumAltB[1] THEN
      IF CumAltA[2] <= CumAltB[2] THEN
         IF CumAltA[3] <= CumAltB[3] THEN
            IF CumAltA[4] <= CumAltB[4] THEN
               IF CumAltA[5] <= CumAltB[5] THEN
                  IF CumAltA[6] <= CumAltB[6] THEN
                     DomSwitch[AltB] := 1;
   END; (* PairSet Loop. Loops until all pairs of alter-    *)
        (* natives in a set of gambles has been considered. *)

   (* Check to see if NumDomAltern # of switches are on, by *)
   (* adding contents of DomSwitch array.                   *)
   DomAlternCounter := 0;       (* initialize *)
   For i := 1 to NAlternatives DO
   BEGIN
       DomAlternCounter := (DomAlternCounter + DomSwitch[i]);
   END;

   (* Set the Boolean condition used to determine whether *)
   (* to keep or discard this generated gamble set.       *)
   IF DomAlternCounter = NumDomAltern
      THEN SetHasDominance := TRUE;

END; (* Procedure SetUpDominance *)


{$list-}

   PROCEDURE InitValues(VAR AttValues: TwoXAttVec);
   (* Initialize attribute VALUES *)
   (* 4/22/85 - Distribution now entered as user input, or if *)
   (* entered as a zero (no choice) defaults to 1000.0.       *)

        VAR AttNum,Tnrand: INTEGER;
            TempValues: AttVec;
```

```
      BEGIN (* InitValues *)
          Tnrand := 12;
          Ggubs(DSeed,Tnrand,TempValues);

(* Multiply payoffs to $1000 or to distribution entered by user. *)
          IF Dist = 0 THEN            (* Zero allows default.   *)
             FOR AttNum := 1 to NAttributes DO
             AttValues[AttNum]:= TempValues[AttNum]*1000.0;
          IF Dist > 0 THEN            (* Value chosen by user.  *)
             FOR AttNum := 1 to NAttributes DO
             AttValues[AttNum]:= TempValues[AttNum]*Dist;

      END; (* InitValues *)


  PROCEDURE InitProbabilities(VAR AttValues: TwoXAttVec);
  (* Initialize attribute PROBABILITIES *)
  (* Tran = 1 = Thorngate               *)
  (* Tran = 2 = Uniform Hyperplane      *)
  (* Tran = 3 = Dirichlet               *)

      VAR AttNum: INTEGER;
          TempValues: AttVec;

      BEGIN (* InitProbabilities *)
          IF (Tran = 1) THEN
             GenProbs(TempValues,NAttributes);
          IF (Tran = 2) THEN
             Hostile(TempValues,NAttributes);
          IF (Tran = 3) THEN
              Dirichlet(TempValues,NAttributes);
          FOR AttNum:= 1 TO NAttributes DO
              AttValues[AttNum + NAttributes]:= TempValues[AttNum];


      END; (* InitProbabilities *)


  FUNCTION CalcValue(CONST AttValues: TwoXAttVec): REAL [PURE];
  (* Calculate the total Value for this choice *)



      VAR AttNum: INTEGER;

      BEGIN (* CalcValue *)
          CalcValue := 0;
          FOR AttNum:= 1 TO NAttributes DO
              CalcValue := RESULT (CalcValue) +
                     AttValues[AttNum] * AttValues[AttNum + NAttributes];

      END; (* CalcValue *)


  FUNCTION CalcUtils(CONST AttUtils:TwoXAttVec):REAL [PURE];
      {Calculate Total Utility for this choice}
      VAR AttNum: INTEGER;

      BEGIN (* CalcUtils *)
          CalcUtils := 0;
```

```
        FOR AttNum:= 1 TO NAttributes DO
            (* the exponential, et al. is equivalent to x^2/3   *)
            CalcUtils := RESULT (CalcUtils) +
                         exp(ln(AttUtils[AttNum]) * (2.0 / 3.0)) *
                         AttUtils[AttNum + NAttributes];

    END; (* CalcUtils *)


PROCEDURE PutInSortedPosition(VAR Posits: AltMap; PositValues: AltVec;
                              LastPosit: INTEGER);
(* Put LastPosit in proper position in Posits, s.t. Posits is *)
(* sorted according to the values of Posits in PositValues. *)

    VAR SortInd: INTEGER;

    BEGIN (* PutInSortedPosition *)
        Posits[LastPosit]:= LastPosit;
        SortInd:= LastPosit;
        IF LastPosit > 1 THEN
            WHILE PositValues[Posits[SortInd]] <
                  PositValues[Posits[SortInd-1]] DO BEGIN

                ISwap(Posits[SortInd],Posits[SortInd-1]);

                IF SortInd > 2 THEN
                    SortInd:= SortInd-1;
                (* else loop terminates because swap made them be in order.
             END; (* WHILE *)
    END; (* PutInSortedPosition *)


PROCEDURE StochValues;

    VAR
        Index,TempOrder : AltMap;
        Temp : ARRAY [1..2] OF INTEGER;
        Order : ARRAY [1..MaxAlternatives,1..MaxAttributes] OF INTEGER;
              (* Temporary matrix in which orders are constructed.   *)
        TempRand : AttVec;
        TXTempRand: TwoXAttvec;
        Pair : ARRAY[1..MaxAlternatives,1..MaxAttributes,1..2 ]
                   OF INTEGER;
        Inval : ARRAY[1..MaxAlternatives,1..MaxAttributes] OF REAL;
        Row,Col,K,L,TempPoint : INTEGER;
        TempVec : CombVec;

    BEGIN (* StochValues *)
    (Echo that stochvalues has been called)

    cursetvv(24,40);
    write(output,'                                          ');
    cursetvv(24,40);
    write(output,'Calling Stochvalues');


        FOR Row := 2 TO NAlternatives DO
            FOR Col := 1 TO Row - 1 DO
            BEGIN
```

```
            Pair[Row,Col,1] := 0;
            Pair[Row,Col,2] := 0;
            WHILE (Pair[Row,Col,1] = Pair[Row,Col,2]) DO
                BEGIN
                    Temp[1]:=IntRand(1,NAttributes);
                    Temp[2]:=IntRand(1,NAttributes);

                    IF Temp[1] < Temp[2] THEN
                        BEGIN
                            Pair[Row,Col,1]:=Temp[1];
                            Pair[Row,Col,2]:=Temp[2];
                        END (* IF *)
                    ELSE
                        BEGIN
                            Pair[Row,Col,2]:=Temp[1];
                            Pair[Row,Col,1]:=Temp[2];
                        END; (* ELSE *)
                END; (* WHILE *)
        END; ( FOR )

    (* Initialize the Values Array *)
    FOR Row:= 1 TO NAlternatives DO
        BEGIN
            InitValues(TXTempRand);
            FOR Col := 1 TO NAttributes DO
                InVal[Row,Col]:=TXTempRand[Col];
        END; (* FOR *)

    (* Sort it in ascending order so that attributes now are *)
    (* exclusive ranges *)
    FOR Col := 1 TO NAttributes DO
        FOR  Row := 1 TO NAlternatives DO
            BEGIN
                TempPoint:= Row * NAttributes - (NAttributes - Col);
                TempVec[TempPoint]:=InVal[Row,Col];
            END; (* FOR *)

    TempPoint:=NAttributes*NAlternatives;
    Vsrta(TempVec,TempPoint);

    FOR Col := 1 TO NAttributes DO
        FOR Row := 1 TO NAlternatives DO
            BEGIN
                TempPoint := (Col * NAlternatives) -
                             (NAlternatives - Row);
                InVal[Row,Col]:=TempVec[TempPoint];
            END; (* FOR *)

    (* Initialize temporary order to NAlternatives *)
    FOR Col := 1 TO NAttributes DO
        FOR Row := 1 TO NAlternatives DO
            Order[Row,Col]:=NAlternatives;

    FOR Row:= 2 TO NAlternatives DO
        FOR Col := 1 TO Row-1 DO
            BEGIN
                Order[Row,Pair[Row,Col,1]] :=
                    Order[Row,Pair[Row,Col,1]] + 1;
                Order[Row,Pair[Row,Col,2]] :=
                    Order[Row,Pair[Row,Col,2]] - 1;
```

```
            END;

        FOR Col := 1 TO NAttributes DO
            BEGIN
                FOR Row := 1 TO NAlternatives DO
                    BEGIN
                        Index[Row]:=Row;
                        TempRand[Row]:= Order[Row,Col];
                    END; (* FOR *)

                Vsrtr(TempRand,NAlternatives,Index); (* Sort the values *)

                FOR Row := 1 TO NAlternatives DO (* Now place the sorted
                                                    values in the output array *)
                    Values[Index[Row],Col] := Inval [Row,Col];
            END; (* FOR *)

        (* Now Permute the attributes for each alternative. *)
        FOR Row := 1 TO NAlternatives DO
            BEGIN
                FOR Col := 1 TO NAttributes DO
                    TempRand[Col] := Values[Row,Col];

                Permuter(NAttributes,TempRand);

                FOR Col := 1 TO NAttributes DO
                    Values[Row,Col] := TempRand[Col];
            END; (* FOR *)

    END; (* StochValues *)


FUNCTION Dominance (InValue: VTable): BOOLEAN;

(* This function determines if their are any dominated alternatives in
the array InArray, using first order stochastic dominance.  InArray is
dimensioned by the globals Nalternatives and Nattributes.  The  value
of dominance is 0 if there are no dominated alternatives, otherwise it
returns the ranking of the dominated  alternative (lowest  cumulative
payoff = 1). It also writes the ranks of the dominated and  dominating
alternative to OUTPUT.  *)


    VAR Dominant,TDominant: BOOLEAN ;
        Row,Col,A1,A2,K,L :INTEGER; (* Index variables *)
        Tvalue,Pvalue : ARRAY [1..MaxAlternatives]OF AttVec;
        (* These hold the cumulative frequencies  *)
        Rndtmp: Altmap; (* Order of the cumulative payoffs *)
        TempSort : AltVec;

    BEGIN (* Dominance *)
    (* First Assign the value array to two variables containing the
       cumulative sums of the probabilities after sorting payoffs*)

        FOR Row := 1 TO NAlternatives DO
            BEGIN
                FOR Col := 1 TO NAttributes DO
                    BEGIN
                        Tempsort[Col]:=InValue[Row,Col];
                        Rndtmp[Col]:=Col;
```

```
            END; (* FOR *)

        Vsrtr(Tempsort,NAttributes,Rndtmp);

        FOR Col := 1 TO NAttributes  DO
            BEGIN
                Tvalue[Row,Col]:=Tempsort[Col];
                IF Col = 1  THEN
                    PValue[Row,Col]:= InValue[Row,Rndtmp[Col]+NAttri
                ELSE PValue[Row,Col]:= PValue[Row,Col-1] +
                                       InValue[Row,Rndtmp[Col]+NAttr
            END; (* FOR *)

        DomValues[Row]:=0.0;
    END; (* FOR *)

(* This section creates an index in Rndtmp which orders the
   alternatives by cumulative payoffs.*)
FOR Row := 1 TO NAlternatives DO
    BEGIN
        Tempsort[Row] := Tvalue[Row,NAttributes];
        Rndtmp[Row] := Row;
    END; (* FOR *)

Vsrtr (Tempsort,NAlternatives,Rndtmp);

(* Now we conduct the test. *)
A1:=NAlternatives;

REPEAT
    A2 := A1 - 1 ;
    REPEAT
        K:=NAttributes;
        Dominant := TRUE;
        WHILE Dominant AND (K >= 1) AND (A1 <> A2) DO
            BEGIN
                L:=NAttributes;
                WHILE Dominant AND (L >= 2) DO
                    IF (Tvalue[Rndtmp[A1],K] < Tvalue[Rndtmp[A2],L])
                       (Pvalue[Rndtmp[A1],K] > Pvalue[Rndtmp[A2],L-1
                        Dominant := FALSE
                    ELSE L := L -  1;
                K:= K - 1;
            END; (* WHILE *)

        (* Last check for the final point. *)
        IF (Tvalue[Rndtmp[A1],1] < Tvalue[Rndtmp[A2],1]) THEN
            Dominant:=FALSE;

        IF (Dominant) THEN
            BEGIN
                DomValues[Rndtmp[A2]]:=1.0;
                TDominant:=TRUE;
                Dominant:=FALSE;
            END;

        IF NOT Dominant THEN A2:= A2 -  1;
    UNTIL (Dominant OR (A2 = 0 ));

    IF NOT Dominant THEN A1:= A1 - 1;
```

```
            UNTIL (Dominant OR (A1 = 1 ));

            IF Dominant THEN
                BEGIN
                    writeln('Dominant Alternative: ',
                            A2   , ' dominates ', A1    ,', Trail: ',I);
                    Dominance := True;
                END
            ELSE Dominance:= TDominant;

        END; (* Dominance *)

{$list+)
(*********************************************************)
(*  INITIALIZE                                           *)
(*********************************************************)

    BEGIN (* Initialize *)

    (* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
    (* Three PayOffGenerator values control the generation of pay- *)
    (* offs in the values matrix:                                  *)
    (* PG = 1 = No Dominance.  Dominance screened out of the set.  *)
    (*          StochValues/Dominance procs in START.              *)
    (* PG = 2 = Dominance.  Dominance occurs randomly in the set.  *)
    (*          InitValues proc in START.                          *)
    (* PG = 3 = Correlated Attributes.  Can be declared equal with *)
    (*          the desired correlation supplied, or unequal.      *)
    (*          If unequal, values are read in from an external    *)
    (*          file called SigFile, named when pgm invoked.       *)
    (*          MakeSig/Ggnms procs in RANDOM.                     *)
    (*                                                             *)
    (* Three Tran values control the generation of probabilities   *)
    (* (or weights) in the values matrix via the InitProbabilities *)
    (* proc in START:                                              *)
    (* Tran = 1 = Thorngate.           (GenProbs  in RANDOM).      *)
    (* Tran = 2 = Uniform Hyperplane.  (Hostile   in RANDOM).      *)
    (* Tran = 3 = Dirichlet.           (Dirichlet in RANDOM).      *)
    (* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)



    (* * *  No Dominance  * * *)

     IF PG = 1 THEN        (* PayOffGenerator - No Dominance *)
     BEGIN
                REPEAT
                    StochValues;
                    FOR AltNum:= 1 TO NAlternatives DO
                    IF ((Risky) OR (AltNum = 1))
                       THEN  InitProbabilities(Values[AltNum])
                    ELSE FOR AttNum := 1 to  NAttributes DO
                         Values[AltNum][NAttributes+AttNum] :=
                                Values[1][NAttributes+AttNum];
                UNTIL ((NOT Dominance(Values)) OR (NOT RISKY)); (* REPEAT *)

        END;  (* If PG = 1 *)


    (* * *  Dominance Occurs Randomly  * * *)
```

```
IF PG = 2 THEN

     (* Modification: when input value of NumDomAltern > 0, *)
     (* selects only those gambles generated w/NumDomAltern *)
     (* number of dominated alternatives per set.           *)
     (* ** Restriction: the set must be 4 alt x 3 att.      *)

    IF NumDomAltern > 0 THEN       (* Specific # dominated alterns *)
    BEGIN

        IF NAlternatives <> 4 THEN     (* error trapping     *)
        BEGIN
            cursetvv(12,0);
        write (output, '***       Required for this run:       ***');
            cursetvv(13,0);
        write (output, '*** NAlternatives = 4  NAttributes = 3 ***');
            cursetvv(14,0);
        write (output, '*** Halt execution & reset parameters. ***');
        END;
        IF NAttributes <> 3 THEN           (* error trapping      *)
        BEGIN
            cursetvv(12,0);
        write (output, '*** Halt execution & reset paramters:  ***');
            cursetvv(13,0);
        write (output, '*** NAlternatives = 4  NAttributes = 3 ***');
        END;

    SetHasDominance := FALSE;
    REPEAT
    FOR AltNum:= 1 to NAlternatives do
       BEGIN
           InitValues(Values[AltNum]);
           IF ((Risky) OR (AltNum = 1))
              THEN  InitProbabilities(Values[AltNum])
           ELSE
              FOR AttNum := 1 to NAttributes DO
                  Values[AltNum][NAttributes+AttNum] :=
                  Values[1][NAttributes+AttNum];
       END; (* FOR AltNum... *)

    SetUpDominance;     (* Checks # dom alterns in gamble set.  *)
    UNTIL SetHasDominance;  (* Repeat until specified # found. *)
END                                 (**** IF NumDomAltern > 0 ****)


ELSE (* For Dominance allowed, but NumDomAltern disabled with  *)
     (* a zero value.  Dominance is random.                    *)

BEGIN
    For AltNum := 1 to NAlternatives do
    BEGIN
        InitValues(Values[AltNum]);
        IF ((Risky) OR (AltNum = 1))
           THEN InitProbabilities(Values[AltNum])
        ELSE
           FOR AttNum := 1 to NAttributes DO
               Values[AltNum][NAttributes+AttNum] :=
               Values[1][NAttributes+AttNum];
    END; (* FOR AltNum... *)
```

```
     (* Check for a dominanted alternative.  *)
            IF Dominance(Values) THEN
                DomExist:= 1
            ELSE DomExist:=0;
END;      (* End IF PG = 2 *)


(* * *  Correlated Attributes  * * *)

IF PG = 3 THEN
BEGIN
   MakeSig;     (* Proc in RANDOM which fills Sigma Array. *)
   Ggnms(SigmaX,NAlternatives,NAttributes,Sigma);
                (* Fills SigmaX array with attributes.     *)
   For AltNum := 1 to NAlternatives DO
   BEGIN
      FOR AttNum := 1 to NAttributes DO
         Values[AltNum,AttNum] := (SigmaX[AltNum,AttNum]*1000.00);
      IF ((RISKY) OR (AltNum = 1))
         THEN InitProbabilities(Values[AltNum])
      ELSE
         For AttNum := 1 to NAttributes DO
            Values[AltNum][NAttributes + AttNum] :=
            Values[1][NAttributes + AttNum];
   END; (* For AltNum *)
END; (* IF PG = 3 *)


    (* Common to all three PayOffGenerators: *)

    FOR AltNum := 1 to NAlternatives DO
    BEGIN

                ChoiceValues[AltNum]:= CalcValue(Values[AltNum]);
                ChoiceUtils[AltNum]:=  CalcUtils(Values[AltNum]);

                (* Do one pass of a sort here. *)
                PutInSortedPosition(InvMap,ChoiceValues,AltNum);
                PutInSortedPosition(InvUMap,ChoiceUtils,AltNum);
     END; (* FOR *)


    (* Convert InvMap from array over rankings to array of rankings. *)
    FOR AltNum:= 1 TO NAlternatives DO
        BEGIN
            ChoiceRanking[InvMap[AltNum]]:=NAlternatives - AltNum + 1;
            ChoiceURank[InvUMap[AltNum]]:=NAlternatives - AltNum + 1;
        END;


    (* CUTOFFS:                                                  *)
    (* 4/22/85 - Use user-specified cutoff or if user allows *)
    (* default (zero), use mean of distribution as cutoff.   *)

    IF CutOffValue = 0 THEN     (* Default to mean of dist.  *)
    BEGIN
```

```
            FOR AttNum := 1 TO NAttributes DO
               CutOffs[AttNum]:=Dist / 2;
               CutOffValue:=Dist / 2;
         END;
         IF CutOffValue > 0 THEN      (* User-specified cutoff.    *)
            FOR AttNum := 1 to NAttributes DO
               CutOffs[AttNum]:=CutOffValue;

         (* Cut offs for prob. are mean of that dist. *)
         FOR AttNum:= NAttributes + 1 TO 2 * NAttributes DO
             CutOffs[AttNum]:= 1 / NAttributes;


(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*      Write out the gambles presented to the rules:                    *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

      IF (FmtPrt=TRUE) THEN   (* Formatted Gamble Output *)
      BEGIN
         Writeln(Gamble, ' ');
         FOR AltNum:= 1 TO NAlternatives DO
         BEGIN
           WRITE(Gamble,AltNum:2);
           FOR AttNum:=1 to NAttributes DO
           BEGIN
             WRITE(Gamble,AttNum:2,Values[AltNum,AttNum]:8:2,
                  Values[AltNum,NAttributes+AttNum]:6:3);
           END;  (* For AttNum *)
           WRITELN(Gamble,'  ');
         END; (* FOR AltNum *)
         Writeln (Gamble, ' ');
       END  (* If FmtPrt *)

    ELSE BEGIN      (* Condensed Gamble Output *)
         FOR AltNum := 1 TO NAlternatives DO
         BEGIN
            WRITE(Gamble,NAlternatives:2,NAttributes:2,Tran:3,AltNum:2);
            FOR AttNum:=1 to NAttributes DO
            BEGIN
               WRITE(Gamble,AttNum:2,Values[AltNum,AttNum]:8:2,
                    Values[AltNum,NAttributes+AttNum]:6:3);
            END; (* For AttNum *)
            WRITELN(Gamble,' ');
         END; (* For AltNum *)
         END; (* ELSE *)

     END; (* Initialize *)

{$list-}
END. {Unit}
```

```
(*****************************************************************)
(* VERSION 1                                    TIMED RISKLESS SIM    *)
(* STMPRIMS.INT                                 08/20/85              *)
(*****************************************************************)



INTERFACE;
UNIT STMPRIMS(TraceSTM,Timer,Remember,GetCut,GetJND,Forget,IsEmpty,
 Mult,Divi,Add,Add1,Subt,Subt1,GetMax,MaxProb,GetMin,GetWeight);
USES Sim;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*  Primitive routines to access the short term memory.                   *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
PROCEDURE TraceSTM(CONST Op:string; l1,l2,l3:MemLoc);

PROCEDURE Timer(TimeVal:REAL);

PROCEDURE Remember(L: MemLoc);
(* Brings current alternative and attribute into short term memory. *)

PROCEDURE GetCut(M1,Result: MemLoc);
(* Brings current alternative and attribute into short term memory. *)

PROCEDURE GetJND(Result: MemLoc);
(* Brings Just Noticeable Difference value into short term memory.   *)
(* Used by riskless LexiSemi rule.                                   *)

PROCEDURE Forget(L: MemLoc);
(* Throw away info in location L and say this spot is open. *)

FUNCTION IsEmpty(L: MemLoc): BOOLEAN;
(* True iff L is a free memory spot. *)


PROCEDURE Mult(M1,M2,Result: MemLoc);
(* Remembers the result of multiplying the Values of M1 and M2 in Result *)
(* The alternative of the result is the greater of the two input alts. *)


PROCEDURE Divi( M1,M2,Result: MemLoc);
(* Remembers the result of dividing the Value's of M1 and M2 in Result *)
(* Division by zero is trapped and the return is equal to zero *)
(* The alternative of the result is the greater of the input alts. *)


PROCEDURE Add(M1,M2,Result: MemLoc);
(* Remembers the result of adding the Values of M1 and M2 in Result *)
(* The alternative of the result is the greater of the two input alts. *)


PROCEDURE Add1(Result: MemLoc);
(* Adds 1 to the value in Result. *)
(* Used to increment, can be changed to use a constant if necessary. *)


PROCEDURE Subt(M1,M2,Result: MemLoc);
(* Remembers the result of subtracting the Values of M2 from M1 in Result *)
(* The alternative of the result is the greater of the two input alts. *)
```

```
PROCEDURE Subt1(Result: MemLoc);
(* Subtracts 1 from the value in Result. *)
(* Used to decrement; can be changed to use a constant if necessary. *)

PROCEDURE GetMax(M1,M2,Result: MemLoc);
(* Remembers the maximum of the values of M1 and M2 in Result *)
(* The alternative of the result is the greater of the two input alts. *)


PROCEDURE MaxProb(M1,M2,Result1,Result2: MemLoc);
(* Evaluates most likely payoff for a given alternative under
   consideration in DoMostLikely. *)


PROCEDURE GetMin(M1,M2,Result: MemLoc);
(* Remembers the minimum of the values of M1 and M2 in Result.    *)
(* The alternative of the result is the LESSER of the input alts. *)

PROCEDURE GetWeight(M1: MemLoc);
(* Brings current attribute weight into short term memory. *)
(* Used by riskless AddUtility and EBA.                    *)

END;
```

```
(*****************************************************************)
(* VERSION 1                                        TIMED RISKLESS SIM   *)
(* STMPRIMS.PAS                                     11/12/85             *)
(* Reads weights for operators from OpWtArr array, which contains     *)
(* values input from an external file supplied by the user.           *)
(*****************************************************************)


{$include:'screenvv.int'}
{$include:'fformsuu.int'}
{$INCLUDE:'sim.int'}
{$INCLUDE:'stmprims.int'}
{$include:'numprims.int'}
{$include:'lstprims.int'}


IMPLEMENTATION OF STMPRIMS;
USES sim,lstprims,numprims,fformsuu,screenvv;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(*  Primitive routines to access the short term memory.                  *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Procedure TraceSTM;
{(CONST Op:string; l1,l2,l3:MemLoc)}

CONST
        xop = 1;
        yop = 12;
        xmem=30;
        ymem=12;
        xatt=1;
        yatt=15;

VAR
        I : Byte;
        ch: char;

BEGIN
If Debug > 0 then
        BEGIN
        Cursetvv(yop,xop);
        Write(output,'                                  ');
        Cursetvv(yop,xop);
        Write(output,'Operator: ',op);
        IF(l1 <> 0) THEN WRITE(l1:2);
        IF(l2 <> 0) THEN WRITE(l2:2);
        IF(l3 <> 0) THEN WRITE(l3:3:2);

        cursetvv(yatt,xatt);
        write(output,'Alternative: ',CurrValue(Altern):3);
        cursetvv(yatt+1,xatt);
        write(output,'Attribute: ',CurrValue(Attrib):3);
END; {IF}
If Debug > 1 then
        BEGIN
        cursetvv(ymem,xmem);
        Write('Memory Contents:');
    FOR I := 1 to NMemCell DO
       WITH STM[I] DO
```

```
                BEGIN
                cursetvv(ymem+I,xmem);
                write(I:3,optype:3,Att:3,Alt:3,AValue:8:2);
                END; {with}
        cursetvv(23,24);
        write('Press return to continue');
        read(ch);
END; {if}

END; {TraceSTM}



PROCEDURE Timer;
(* TimeVal:REAL; *)
(* Specific to Timed Sim.  This procedure is called by each operator, *)
(* such as Remember, Elim, etc.   The operator's TimeVal weight is    *)
(* passed and used to increment the total "time" used for the rule    *)
(* being run.  A condition statement in the rule then sets TimeUp     *)
(* to TRUE if the TimeLimit has been reached, and normal processing   *)
(* in the rule stops.                                                 *)

VAR TimeUp: BOOLEAN;

BEGIN   (* Timer *)
   TimeUp := FALSE;
   Time[Rule] := Time[Rule] + TimeVal;
   IF Time[Rule] >= TimeLimit THEN
      TimeUp := TRUE;


END; (* Timer *)



PROCEDURE Remember;
(* Brings current alternative and attribute into short term memory. *)

 VAR TimeVal: REAL;

    BEGIN (* Remember *)
        WITH STM[L] DO BEGIN
           OpType:= Momento;
           Alt:= CurrValue(Altern);
           Att:= CurrValue(Attrib);
           AValue:= Values[Alt][Att];


        END; (* WITH *)

        TraceSTM('Remember',L,0,0);
        OperationCount[Momento]:= OperationCount[Momento] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Momento];
        Timer(TimeVal);

    END; (* Remember *)
```

```
PROCEDURE GetCut;
(* Brings current alternative and attribute into short term memory. *)

 VAR TimeVal: REAL;

    BEGIN (* GetCut *)
        WITH STM[Result] DO BEGIN
            Att:= STM[M1].Att;
            IF Att = 0 THEN Att:=1;
            Alt:= CutOffVal;
            AValue:= CutOffs[Att];
            OpType:= Blood;
        END; (* WITH *)

        TraceSTM('GetCut',M1,Result,0);
        OperationCount[Blood]:= OperationCount[Blood] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Blood];
        Timer(TimeVal);

    END; (* GetCut *)


PROCEDURE GetJND;
(* Brings Just Noticeable Difference value into STM. *)
(* Result:MemLoc *)

 VAR TimeVal: REAL;


BEGIN  (* GetJND *)
   WITH STM[Result] DO
   BEGIN
      AValue := JND;
      OpType := Squint;
   END; (* With *)

   TraceSTM('GetJND',Result,0,0);
   OperationCount[Squint]:=OperationCount[Squint]+1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Squint];
        Timer(TimeVal);

END; (* GetJnd *)


PROCEDURE Forget;
(* Throw away info in location L and say this spot is open. *)

 VAR TimeVal: REAL;

    BEGIN (* Forget *)
        WITH STM[L] DO BEGIN
            OpType:= Empty;
            Alt:= 0;
            Att:= 0;
            AValue:= 0.0;
        END; (* WITH *)
```

```
        TraceSTM('Forget',L,0,0);
        OperationCount[Oversight]:= OperationCount[Oversight] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[OverSight];
        Timer(TimeVal);

    END; (* Forget *)


FUNCTION IsEmpty;
(* True iff L is a free memory spot. *)

 VAR TimeVal: REAL;

    BEGIN (* IsEmpty *)
        WITH STM[L] DO
            IsEmpty:= OpType = Empty;

        TraceSTM('IsEmpty',L,0,0);
        OperationCount[Empty] := OperationCount[Empty] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Empty];
        Timer(TimeVal);

    END; (* IsEmpty *)


PROCEDURE Mult;
(* Remembers the result of multiplying the Values of M1 and M2 in Result *)
(* The alternative of the result is the greater of the two input alts. *)

 VAR TimeVal: REAL;

    BEGIN (* Mult *)
        WITH STM[Result] DO BEGIN
            AValue:= STM[M1].AValue * STM[M2].AValue;

            OpType:= Product;

            Alt:= STM[M1].Alt;
            Att:= STM[M1].Att;
        END; (* WITH *)

        TraceSTM('Mult',M1,M2,Result);
        OperationCount[Product]:= OperationCount[Product] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Product];
        Timer(TimeVal);

    END; (* Mult *)


PROCEDURE Divi;
(* Remembers the result of dividing the Value's of M1 and M2 in Result *)
(* Division by zero is trapped and the return is equal to zero *)
(* The alternative of the result is the greater of the input alts. *)
```

```
 VAR TimeVal: REAL;

    BEGIN (* Div *)
        WITH STM[Result] DO BEGIN
            IF (STM[M2].AValue = 0) THEN STM[Result].AValue := 0
            ELSE AValue:=STM[M1].AValue / STM[M2].AValue;

            OpType:= Quotient;

            Alt:= STM[M1].Alt;
            Att:= STM[M2].Att;
        END; (* WITH *)

        TraceSTM('Divi',M1,M2,Result);
        OperationCount[Quotient]:= OperationCount[Quotient] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Quotient];
        Timer(TimeVal);

    END; (* Divi *)


PROCEDURE Add;
(* Remembers the result of adding the Values of M1 and M2 in Result *)
(* The alternative of the result is the greater of the two input alts. *)

 VAR TimeVal: REAL;

    BEGIN (* Add *)
        WITH STM[Result] DO BEGIN
            AValue:= STM[M1].AValue + STM[M2].AValue;

            OpType:= Sum;

            Alt:= STM[M1].Alt;
            Att:= STM[M1].Att;
        END; (* WITH *)

        TraceSTM('Add',M1,M2,Result);
        OperationCount[Sum]:= OperationCount[Sum] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Sum];
        Timer(TimeVal);

    END; (* Add *)


PROCEDURE Add1;
(* Adds 1 to the value in Result *)
(* Used to increment, can be changed to use a constant if necessary. *)

 VAR TimeVal: REAL;

    BEGIN (* Add *)
        WITH STM[Result] DO BEGIN
            AValue:= STM[Result].AValue + 1.0;
```

```
            OpType:= Sum;

            Alt:= STM[Result].Alt;
            Att:= STM[Result].Att;
        END; (* WITH *)

        TraceSTM('Add1',Result,0,0);
        OperationCount[Sum]:= OperationCount[Sum] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Sum];
        Timer(TimeVal);

    END; (* Add1 *)


PROCEDURE Subt;
(* Remembers the result of subtracting the Values of M2 from M1 in Result *)
(* The alternative of the result is the greater of the two input alts. *)

 VAR TimeVal: REAL;

    BEGIN (* Subt *)
        WITH STM[Result] DO BEGIN
            AValue:= STM[M1].AValue - STM[M2].AValue;

            OpType:= Difference;

            Alt:= STM[M1].Alt;
            Att:= STM[M1].Att;
        END; (* WITH *)

        TraceSTM('Subt',M1,M2,Result);
        OperationCount[Difference]:= OperationCount[Difference] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Difference];
        Timer(TimeVal);

    END; (* Subt *)


PROCEDURE Subt1;
(* Subtracts 1 from the value in Result. *)
(* Used to decrement, can be changed to use a constant if necessary. *)

 VAR TimeVal: REAL;

    BEGIN (* Add *)
        WITH STM[Result] DO BEGIN
            AValue:= STM[Result].AValue - 1.0;

            OpType:= Difference;

            Alt:= STM[Result].Alt;
            Att:= STM[Result].Att;
        END; (* WITH *)

        TraceSTM('Subt1',Result,0,0);
        OperationCount[Difference]:= OperationCount[Difference] + 1;
```

```
        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Difference];
        Timer(TimeVal);

    END; (* Subt1 *)


PROCEDURE GetMax;
(* Remembers the maximum of the values of M1 and M2 in Result *)
(* The alternative of the result is the greater of the two input alts. *)

 VAR TimeVal: REAL;

    BEGIN (* GetMax *)
        WITH STM[Result] DO BEGIN
            IF STM[M1].AValue > STM[M2].AValue THEN BEGIN
                AValue:= STM[M1].AValue;
                Alt:= STM[M1].Alt;
                Att:= STM[M1].Att;
            END ELSE BEGIN
                AValue:= STM[M2].AValue;
                Alt:= STM[M2].Alt;
                Att:= STM[M2].Att;
            END;

            OpType:= Maximum;
        END; (* WITH *)

        TraceSTM('GetMax',M1,M2,Result);
        OperationCount[Maximum]:= OperationCount[Maximum] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Maximum];
        Timer(TimeVal);

    END; (* GetMax *)


PROCEDURE MaxProb;
(* Evaluates most likely payoff for a given alternative under
   consideration in DoMostLikely. *)

 VAR TimeVal: REAL;

    BEGIN (* MaxProb *)
        WITH STM[Result1] DO BEGIN
            IF STM[M1].AValue>STM[M2].AValue THEN BEGIN
                AValue:= STM[M1].AValue;
                Att:= STM[M1].Att;
                Alt:= STM[M1].Alt;
                OpType:= STM[M1].OpType;
            END;
        END;

        WITH STM[Result2] DO BEGIN
            IF STM[M1].AValue>STM[M2].AValue THEN BEGIN
                AValue:= STM[3].AValue;
                Att:= STM[3].Att;
                Alt:= STM[3].Alt;
```

```
            OpType:= STM[3].OpType;
        END;
    END; (* WITH *)

    (* For Timed Sim, call proc to increment counter: *)
    TimeVal := 1.0;
    Timer(TimeVal);

END; (* MaxProb *)


PROCEDURE GetMin;
(* Remembers the minimum of the values of M1 and M2 in Result *)
(* The alternative of the result is the LESSER of the input alts. *)

VAR TimeVal: REAL;

    BEGIN (* GetMin *)
        WITH STM[Result] DO BEGIN
            IF STM[M2].AValue = 0 THEN BEGIN
            (* always puts empty cells as the lesser *)
                AValue:= STM[M1].AValue;
                Att:= STM[M1].Att;
                Alt:= STM[M1].Alt;
            END ELSE IF STM[M1].AValue < STM[M2].AValue THEN BEGIN
                AValue:= STM[M1].AValue;
                Att:= STM[M1].Att;
                Alt:= STM[M1].Alt;
            END ELSE BEGIN
                AValue:= STM[M2].AValue;
                Alt:= STM[M2].Alt;
                Att:= STM[M2].Att;
            END;

            OpType:= Minimum;
        END; (* WITH *)

        TraceSTM('GetMin',M1,M2,Result);
        OperationCount[Minimum]:= OperationCount[Minimum] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Minimum];
        Timer(TimeVal);

    END; (* GetMin *)

PROCEDURE GetWeight;
{M1 :MemLoc}
(* Brings current  attribute weight into STM into short term memory. *)

VAR TimeVal: REAL;

    BEGIN (* GetWeight *)
        WITH STM[M1] DO BEGIN
            Att:= CurrValue(Attrib);
            Alt:= CurrValue(Altern);
            AValue:= Values[1][NAttributes+Att];
            OpType:= Momento;
        END; (* WITH *)
```

```
        TraceSTM('GetWeight',M1,0,0);
        IF Risky THEN Write(Output,'Should not be used in Risky Choice');
        OperationCount[Momento]:= OperationCount[Momento] + 1;
        OperationCount[TheNext]:= OperationCount[TheNext] + 1;

        (* For Timed Sim, call proc to increment counter: *)
        TimeVal := OpWtArr[Momento];
        Timer(TimeVal);

    END; (* GetWeight *)

END.
{$list+}
```