Simple Huffman coder

```
--  This chapter contains routines to transmit and receive character
--  streams encoded by a fixed Huffman code.
--

--
--  character set description
--

VAL INT bits.in.character    IS 8 :
VAL INT number.of.characters IS 1 << bits.in.character :
VAL INT number.of.codes      IS number.of.characters :

VAL INT character.mask IS BITNOT ((BITNOT 0) << bits.in.character) :

INT FUNCTION index(VAL INT char) IS char /\ character.mask :

VAL [number.of.characters]REAL32 probability IS ... :
        --  table of expected probabilities of characters;
        --  probability[index(char)] is the probability of char

--
--  structure of the tree
--

VAL INT root IS 0 :
VAL INT size.of.tree IS (2 * number.of.codes) - 1 :

VAL INT number.of.arrays IS 4 :

VAL INT eldest.index         IS 0 :
VAL INT parent.index         IS 1 :
VAL INT character.index      IS 2 :
VAL INT representative.index IS 3 :
```

```
--
--  insert.new.node
--

PROC insert.new.node([][]INT tree,
                     INT left.limit,
                     VAL INT right.limit,
                     INT new.node,
                     []REAL32 weight,
                     VAL REAL32 weight.of.new.node )
  INT weight.limit :
  SEQ
    IF
      IF node = left.limit FOR right.limit - left.limit
        weight[node] <= weight.of.new.node
          weight.limit := node
      TRUE
        weight.limit := right.limit
    []INT eldest IS tree[eldest.index] :
    []INT character IS tree[character.index] :
    SEQ node = left.limit FOR weight.limit - left.limit
      character[node-1], eldest[node-1], weight[node-1] :=
        character[node],   eldest[node],   weight[node]
    left.limit, new.node := left.limit - 1, weight.limit - 1
    weight[new.node] := weight.of.new.node
:

--
-- construct leaf nodes of tree
--

PROC construct.leaves([][]INT tree,
                      INT left.limit, right.limit,
                      []REAL32 weight,
                      VAL []REAL32 probability    )
  VAL INT minimum.character IS -(number.of.characters / 2) :
  SEQ char = minimum.character FOR number.of.characters
    INT new.node :
    SEQ
      insert.new.node(tree, left.limit, right.limit,
                      new.node, weight,
                      probability[index(char)]      )
      []INT eldest IS tree[eldest.index] :
      []INT character IS tree[character.index] :
      eldest[new.node], character[new.node] := root, char
:
```

```
--
--  construct non-leaf nodes of the tree
--

PROC construct.other.nodes([][]INT tree,
                           INT left.limit, right.limit,
                           []REAL32 weight             )
  WHILE (right.limit - left.limit) <> 1
    INT new.node :
    REAL32 new.weight :
    SEQ
      right.limit := right.limit - 2
      new.weight := weight[right.limit] + weight[right.limit + 1]
      insert.new.node(tree, left.limit, right.limit,
                      new.node, weight, new.weight  )
      []INT eldest IS tree[eldest.index] :
      eldest[new.node] := right.limit
:

--
--  construct parent from eldest, representative from character
--

PROC invert.representation([][]INT tree)
  []INT eldest IS tree[eldest.index] :
  []INT parent IS tree[parent.index] :
  []INT character IS tree[character.index] :
  []INT representative IS
      [tree[representative.index] FROM 0 FOR number.of.characters] :
  SEQ node = root FOR size.of.tree
    IF
      eldest[node] = root
        representative[index(character[node])] := node
      eldest[node] <> root
        SEQ child = eldest[node] FOR 2
          parent[child] := node
:
```

```
--
--  construct.tree
--

PROC construct.tree([][]INT tree, VAL []REAL32 probability)

  INT left.limit, right.limit :
  [size.of.tree]REAL32 weight :

  SEQ
    left.limit   := size.of.tree
    right.limit  := size.of.tree

    -- left.limit = size.of.tree
    -- (right.limit - left.limit) = 0

    construct.leaves(tree, left.limit, right.limit,
                           weight, probability     )

    -- left.limit = number.of.codes
    -- (right.limit - left.limit) = number.of.codes

    construct.other.nodes(tree, left.limit, right.limit, weight)

    -- left.limit = root
    -- (right.limit - left.limit) = 1

    invert.representation(tree)
:
```

```
--
--  encode.character, decode.character
--

PROC encode.character(CHAN OF BIT output,
                      VAL [][]INT tree,
                      VAL INT char       )

  VAL []INT eldest IS tree[eldest.index] :
  VAL []INT parent IS tree[parent.index] :
  VAL []INT character IS tree[character.index] :
  VAL []INT representative IS
       [tree[representative.index] FROM 0 FOR number.of.characters] :

  VAL INT size.of.encoding IS number.of.codes - 1 :
  [size.of.encoding]BOOL encoding :
  INT length, node :
  SEQ
    length := 0
    node   := representative[index(char)]
    WHILE node <> root
      SEQ
        encoding[length] := node <> eldest[parent[node]]
        length := length + 1
        node   := parent[node]
    SEQ i = 1 FOR length
      output ! encoding[length - i]
:

PROC decode.character(CHAN OF BIT input,
                      VAL [][]INT tree,
                      INT char          )

  VAL []INT eldest IS tree[eldest.index] :
  VAL []INT character IS tree[character.index] :

  INT node :
  SEQ
    node := root
    WHILE eldest[node] <> root
      BOOL go.right :
      SEQ
        input ? go.right
        IF
          go.right
            node := eldest[node] + 1
          NOT go.right
            node := eldest[node]
    char := character[node]

:
```

```
--
--  copy.encoding, copy.decoding
--

VAL INT end.of.message IS -1 :

PROC copy.encoding(CHAN OF INT source,
                   CHAN OF SIGNAL end.of.source,
                   CHAN OF BIT sink             )
  --
  -- read characters from source, sending their encodings along sink,
  -- until a signal is received along end.of.source
  --
  BOOL more.characters.expected :
  [number.of.arrays][size.of.tree]INT tree :
  SEQ
    construct.tree(tree, probability)
    more.characters.expected := TRUE
    WHILE more.characters.expected
      ALT
        INT char :
        source ? char
          encode.character(sink, tree, char)
        end.of.source ? CASE signal
          more.characters.expected := FALSE
    encode.character(sink, tree, end.of.message)

:

PROC copy.decoding(CHAN OF BIT source, CHAN OF INT sink)
  --
  --  read a bit stream from source, decoding it into characters and
  --  send these along sink until end.of.message is decoded
  --
  BOOL more.characters.expected :
  [number.of.arrays][size.of.tree]INT tree :
  SEQ
    construct.tree(tree, probability)
    more.characters.expected := TRUE
    WHILE more.characters.expected
      INT char :
      SEQ
        decode.character(source, tree, char)
        IF
          char <> end.of.message
            sink ! char
          char = end.of.message
            more.characters.expected := FALSE
:
```

```
--
--  encode.into.blocks, decode.from.blocks
--

--
--  this code uses the packing routines from an earlier chapter
--
--    PROC pack.bits.into.blocks(...)
--    PROC unpack.bits.from.blocks(...)
--

PROC encode.into.blocks(CHAN OF INT source,
                        CHAN OF SIGNAL end.of.source,
                        CHAN OF BLOCK sink           )
  CHAN OF BIT bit.stream :
  CHAN OF SIGNAL end.of.bit.stream :
  PAR
    SEQ
      copy.encoding(source, end.of.source, bit.stream)
      end.of.bit.stream ! signal
    pack.bits.into.blocks(bit.stream, end.of.bit.stream, sink)
:

PROC discard(CHAN OF BIT source, CHAN OF SIGNAL end.of.source)
  BOOL more.expected :
  SEQ
    more.expected := TRUE
    WHILE more.expected
      ALT
        BOOL bit :
        source ? bit
          SKIP
        end.of.source ? CASE signal
          more.expected := FALSE
:

PROC decode.from.blocks(CHAN OF BLOCK block.source, CHAN OF INT sink)
  CHAN OF SIGNAL end.of.block.source, end.of.bit.stream :
  CHAN OF BIT bit.stream :
  PAR
    SEQ
      unpack.bits.from.blocks(block.source,
                              end.of.block.source, bit.stream )
      end.of.bit.stream ! signal                    -- 'feed-forward'
    SEQ
      copy.decoding(bit.stream, sink)
      PAR
        end.of.block.source ! signal                -- 'feed-back'
        discard(bit.stream, end.of.bit.stream)
:
```

```
--
--  copy.over.serial.medium, copy.over.blocked.medium
--

PROC copy.over.serial.medium(CHAN OF INT source,
                             CHAN OF SIGNAL end.of.source,
                             CHAN OF INT sink            )
  --
  --  copy characters from source to sink until a signal is received
  --  from end.of.source
  --
  CHAN OF BIT serial.medium :
  PAR
    copy.encoding(source, end.of.source, serial.medium)
    copy.decoding(serial.medium, sink)
:

PROC copy.over.blocked.medium(CHAN OF INT source,
                              CHAN OF SIGNAL end.of.source,
                              CHAN OF INT sink            )
  --
  --  copy characters from source to sink until a signal is received
  --  from end.of.source
  --
  CHAN OF BLOCK blocked.medium :
  PAR
    encode.into.blocks(source, end.of.source, blocked.medium)
    decode.from.blocks(blocked.medium, sink)
:
```