```
//                              -*- C++ -*-
//
//                                Enigma
//                     Finite State Transducer Library
//                                (fst)
//
//  Module:  fst.fst
//  Purpose: Representation and manipulation of finite state machines.
//  Author:  John McDonough and Emilian Stoimenov
//
// This program is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation; either version 2 of the License, or (at
// your option) any later version.
//
// This program is distributed in the hope that it will be useful, but
// WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
// General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.


#ifndef _fst_h_
#define _fst_h_

#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif

#include <list>
#include <set>
#include <map>
#include <queue>

#include "common/mlist.h"
#include "common/refcount.h"
#include "common/jexception.h"
#include "common/memoryManager.h"
#include "dictionary/distribTree.h"
#include "fsm/skhash.h"
#include "config.h"


LogDouble logAdd(LogDouble a, LogDouble b);
Weight logAdd(Weight a, Weight b);
static const LogDouble  LogZero       = 1.0E10;         // ~log(0)
static const Weight     LogZeroWeight(float(1.0E10));   // ~log(0)
static const unsigned   Primes[] = {263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 3
31, 337, 347, 349, 353, 359, 367};

static const Weight ZeroWeight(float(0.0));

class WFSAcceptor;
typedef refcountable_ptr<WFSAcceptor>   WFSAcceptorPtr;

// ----- definition for class `Lexicon' -----
//
class Lexicon : public Countable {
 public:
  Lexicon(const String& nm, const String& fileName = "");
  ~Lexicon() { }

  void clear() { _list.clear(); }
  const String& name() const { return _list.name(); }
  unsigned size() const { return _list.size(); }

  void read(const String& fileName);
  void write(const String& fileName, bool writeHeader = false, WFSAcceptorPtr wfsa = NULL) c
onst;

  unsigned index(const String& symbol, bool create = false);
  const String& symbol(unsigned idx) const { return _list[idx]; }
  bool isPresent(const String& symbol) const { return _list.isPresent(symbol); }

  class Iterator;  friend class Iterator;

 private:
  typedef List<String>                  _List;
  typedef _List::Iterator               _ListIterator;
  typedef _List::ConstIterator          _ListConstIterator;

  char                                  _commentChar;
  _List                                 _list;
};

typedef refcountable_ptr<Lexicon> LexiconPtr;


// ----- definition for class `Lexicon::Iterator' -----
//
class Lexicon::Iterator {
 public:
  Iterator(LexiconPtr& lex):
    _lexicon(lex), _itr(lex->_list) {}

  String operator->()  { return *_itr; }
  String operator*()   { return *_itr; }
  String name() const  { return *_itr; }
  void operator++(int) { _itr++; }
  bool more()          { return _itr.more(); }
  inline String next();

 private:
  LexiconPtr            _lexicon;
  _ListIterator         _itr;
};

// needed for Python iterator
String Lexicon::Iterator::next() {
  if (!more())
    throw jiterator_error("end of lexicon!");

  String st(name());
  operator++(1);
  return st;
}


// ----- definition for class `WFSAcceptor' -----
//
class WFSAcceptor : public Countable {

  friend class WFSTransducer;
```

```
  friend class ContextDependencyTransducer;
  friend class HiddenMarkovModelTransducer;
  friend class WFSTProjection;
  friend class CombinedTransducer;

  friend class MinimizeFSA;
  friend class EncodeWFST;
  friend class DecodeFSA;
public:
  typedef enum { White = 0, Gray = 1, Black = 2 } Color;

  class Node;      friend class Node;
  class Edge;      friend class Edge;

  typedef refcountable_ptr<WFSAcceptor> Ptr;
  typedef refcountable_ptr<Node>        NodePtr;
  typedef refcountable_ptr<Edge>        EdgePtr;

  WFSAcceptor(LexiconPtr& inlex);
  WFSAcceptor(LexiconPtr& statelex, LexiconPtr& inlex, const String& name = "WFSAcceptor");
  virtual ~WFSAcceptor();

  const String& name() { return _name; }

  virtual NodePtr& initial(int idx = -1) { return _initial; }
  LexiconPtr& stateLexicon() { return _stateLexicon; }
  LexiconPtr& inputLexicon() { return _inputLexicon; }

  virtual void read(const String& fileName, bool noSelfLoops = false);
  virtual void write(const String& fileName = "", bool useSymbols = false);
  virtual void printStats() const;

  void reverse(const WFSAcceptorPtr& wfsa);

  bool isEndState(const String& st) const {
    _ConstNodeMapIterator itr = _final.find(_stateLexicon->index(st));
    return itr != _final.end();
  }

  bool hasFinal(unsigned state);

  bool hasFinalState() const { return (_final.size() > 0); }

  // search for epsilon cycles
  bool epsilonCycle(NodePtr& node);

  void clear() { _clear(); }

  NodePtr find(unsigned state, bool create = false) { return _find(state, create); }

  virtual const EdgePtr& edges(NodePtr& node);

  void setColor(Color c);

protected:
  virtual void _clear();
  void _resize(unsigned state);
  bool _visit(NodePtr& node, set<unsigned>& visited);

  void _addFinal(unsigned state, Weight cost = ZeroWeight);
  NodePtr _find(unsigned state, bool create = false);

  virtual Node* _newNode(unsigned state);
  virtual Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned output, Weight
 cost = ZeroWeight);

  typedef vector<NodePtr>               _NodeVector;
  typedef _NodeVector::iterator         _NodeVectorIterator;
  typedef _NodeVector::const_iterator   _ConstNodeVectorIterator;

  typedef map<unsigned, NodePtr >       _NodeMap;
  typedef _NodeMap::iterator            _NodeMapIterator;
  typedef _NodeMap::const_iterator      _ConstNodeMapIterator;
  typedef _NodeMap::value_type          _ValueType;

  _NodeVector& _allNodes() { return _nodes; }
  _NodeMap&    _finis()    { return _final; }

  String                                _name;
  unsigned                              _totalNodes;
  unsigned                              _totalFinalNodes;
  unsigned                              _totalEdges;

  LexiconPtr                            _stateLexicon;
  LexiconPtr                            _inputLexicon;

  NodePtr                               _initial;
  _NodeVector                           _nodes;
  _NodeMap                              _final;
};

typedef WFSAcceptor::Edge               WFSAcceptorEdge;
typedef WFSAcceptor::Node               WFSAcceptorNode;

typedef WFSAcceptor::Ptr                WFSAcceptorPtr;
typedef WFSAcceptor::EdgePtr            WFSAcceptorEdgePtr;
typedef WFSAcceptor::NodePtr            WFSAcceptorNodePtr;


// ----- definition for class 'WFSAcceptor::Edge' -----
//
class WFSAcceptor::Edge : public Countable {
  friend void _addFinal(unsigned state, Weight cost);

  friend class Node;
  friend class WFSAcceptor;
  friend class WFSTransducer;
  friend class WFSTSortedInput;
  friend class WFSTSortedOutput;
  friend class WFSTComposition;
  friend class WFSTDeterminization;
  friend class WeightPusher;
  friend class DepthFirstApplyConfidences;

public:
  static const double MinimumCost = 1.0E-04;

  Edge(NodePtr& prev, NodePtr& next, unsigned symbol, Weight cost = ZeroWeight);
  Edge(NodePtr& prev, NodePtr& next, unsigned input, unsigned output, Weight cost);

  virtual ~Edge();

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return _prev;   }
```

```
        NodePtr& next()        { return _next;   }
  const NodePtr& prev()  const { return _prev;   }
  const NodePtr& next()  const { return _next;   }

  unsigned       input()  const { return _input;  }
  unsigned       output() const { return _output; }
  Weight         cost()   const { return _cost;   }

  virtual void   write(FILE* fp = stdout);
  virtual void   write(LexiconPtr& statelex, LexiconPtr& inputlex, FILE* fp = stdout);
  virtual void   write(LexiconPtr& statelex, LexiconPtr& inputlex, LexiconPtr& outputlex, FI
LE* fp = stdout);

  EdgePtr& _edges() { return _edgeList; }

  static void report() { memoryManager().report(); }

 protected:
  NodePtr               _prev;
  NodePtr               _next;
  EdgePtr               _edgeList;

 private:
  void _setInput(unsigned toX)  { unsigned* i = (unsigned*) &_input;  *i = toX; }
  void _setOutput(unsigned toX) { unsigned* i = (unsigned*) &_output; *i = toX; }
  void _setCost(Weight wgt) { Weight* c = (Weight*) &_cost; *c = wgt; }

  static MemoryManager<Edge>& memoryManager();

  const unsigned        _input;
  const unsigned        _output;
  const Weight          _cost;
};


// ----- definition for class 'WFSAcceptor::Node' -----
//
class WFSAcceptor::Node : public Countable {

  friend class WFSAcceptor;
  friend class WFSTransducer;
  friend class WFSTSortedInput;
  friend class WFSTSortedOutput;
  friend class WFSTComposition;
  friend class WFSTDeterminization;

  typedef struct {
    unsigned final:1, index:28, color:2, allowPurge:1;
  } _NodeIndex;

 public:
  Node(unsigned idx, Weight cost = ZeroWeight);
  virtual ~Node();

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

  unsigned index() const { return _index.index; }
  Weight cost() const { return _cost; }
  bool isFinal() const { return (_index.final == 1); }
```

```
  void write(FILE* fp = stdout);
  void write(LexiconPtr& statelex, FILE* fp = stdout);

  void writeArcs(FILE* fp = stdout);
  void writeArcs(LexiconPtr& statelex, LexiconPtr& arclex, FILE* fp = stdout);

  unsigned edgesN() const;

  void  setColor(Color c) { _index.color = unsigned(c); }
  Color color() const { return Color(_index.color); }

  void  enablePurge() { _index.allowPurge = 1; }
  void  disablePurge() { _index.allowPurge = 0; }
  bool  canPurge()    { return (_index.allowPurge == 1); }


  Iterator* iterator();

  void _setCost(Weight cost = ZeroWeight) {
    // cout << "Setting cost " << float(cost) << " for node " << index() << endl;
    _cost = cost; _index.final = 1;
  }

  static void report() { memoryManager().report(); }

  // protected:
  EdgePtr& _edges() { return _edgeList; }

  virtual void _addEdge(EdgePtr& newEdge);
  virtual void _addEdgeForce(EdgePtr& newEdge);
  void _clear();
  void _setIndex(unsigned idx) { _index.index = idx; }

 private:
  static MemoryManager<Node>& memoryManager();

  _NodeIndex            _index;
  Weight                _cost;

 protected:
  static const unsigned _MaximumIndex;

  EdgePtr               _edgeList;
};

// ----- definition for class 'WFSAcceptor::Node::Iterator' -----
//
class WFSAcceptor::Node::Iterator {
 public:
  Iterator(const WFSAcceptor::NodePtr& node)
    : _edgePtr(node->_edgeList) { }
  Iterator(WFSAcceptor* wfsa, WFSAcceptor::NodePtr& node)
    : _edgePtr(wfsa->edges(node)) { }
  Iterator(WFSAcceptorPtr& wfst, WFSAcceptor::NodePtr& node)
    : _edgePtr(wfst->edges(node)) { }
  ~Iterator() { }

  bool more() const { return _edgePtr.isNull() == false; }
  WFSAcceptor::EdgePtr& edge() { return _edge(); }
  void operator++(int) {
    EdgePtr edge(_edgePtr);
    if (more()) _edgePtr = edge->_edges();
  }
```

```
 protected:
  WFSAcceptor::EdgePtr& _edge() { return _edgePtr; }
  inline WFSAcceptor::EdgePtr& _next();

  WFSAcceptor::EdgePtr      _edgePtr;
};

// needed for Python iterator
//
WFSAcceptor::EdgePtr& WFSAcceptor::Node::Iterator::_next() {
  if (!more())
    throw jiterator_error("end of edges!");

  EdgePtr& ed(_edge());
  operator++(1);
  return ed;
}


// ----- definition for class 'WFSTransducer' -----
//
class WFSTransducer;
typedef Inherit<WFSTransducer, WFSAcceptorPtr>          WFSTransducerPtr;

template <class WFSType, class NodePtr, class EdgePtr> class _Decoder;

class WFSTransducer : public WFSAcceptor {
  friend class WeightPusher;
  friend class WFSTProjection;
  friend class EncodeWFST;
  friend class PurgeWFST;
 public:
  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSAcceptor::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSAcceptor::EdgePtr> EdgePtr;

  friend class _Decoder<WFSTransducer, NodePtr, EdgePtr>;

  virtual void purgeUnique(unsigned count = 10000) { }

 protected:
  typedef vector<NodePtr>               _NodeVector;
  typedef map<unsigned, NodePtr >       _NodeMap;
  /*
  typedef _NodeMap::iterator            _NodeMapIterator;
  typedef _NodeMap::const_iterator      _ConstNodeMapIterator;
  typedef _NodeMap::value_type          _ValueType;
  */

 public:
  WFSTransducer(const WFSAcceptorPtr& wfsa, bool convertWFSA = true, const String& name = "W
FSTransducer");
  WFSTransducer(LexiconPtr& statelex, LexiconPtr& inlex, LexiconPtr& outlex, const String& n
ame = "WFSTransducer");

  virtual ~WFSTransducer() { }

  virtual NodePtr& initial(int idx = -1);

  NodePtr find(unsigned state, bool create = false) { return Cast<NodePtr>(_find(state, crea
te)); }

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& node);

  virtual void read(const String& fileName, bool noSelfLoops = false);
  virtual void reverseRead(const String& fileName, bool noSelfLoops = false);
  virtual void write(const String& fileName = "", bool useSymbols = false);
  virtual void printStats() const;

  LexiconPtr& outputLexicon() { return _outputLexicon; }

  void reverse(const WFSTransducerPtr& wfst);

  void replaceSymbol(const String& fromSym, const String& toSym, bool input = true);

 protected:
  _NodeVector& _allNodes() { return Cast<_NodeVector>(_nodes); }
  _NodeMap& _finis()  { return Cast<_NodeMap>(_final); }
  void _convert(const WFSAcceptorPtr& wfsa);
  void _reindex();

  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

 private:
  void _replaceSym(NodePtr& node, unsigned fromX, unsigned toX, bool input);

  LexiconPtr                                            _outputLexicon;
};

typedef WFSTransducer::Edge                             WFSTransducerEdge;
typedef WFSTransducer::Node                             WFSTransducerNode;

typedef WFSTransducer::EdgePtr                          WFSTransducerEdgePtr;
typedef WFSTransducer::NodePtr                          WFSTransducerNodePtr;

WFSTransducerPtr reverse(const WFSTransducerPtr& wfst);


// ----- definition for class 'WFSTransducer::Edge' -----
//
class WFSTransducer::Edge : public WFSAcceptor::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next,
       unsigned input, unsigned output, Weight cost = ZeroWeight)
    : WFSAcceptor::Edge(prev, next, input, output, cost) { }
  virtual ~Edge() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  virtual void write(FILE* fp = stdout);
  virtual void write(LexiconPtr& statelex, LexiconPtr& inputlex,
                     LexiconPtr& outputlex, FILE* fp = stdout);
```

```
  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

  static void report() { memoryManager().report(); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTransducer::Node' -----
//
class WFSTransducer::Node : public WFSAcceptor::Node {
  // friend void WFSAcceptor::_addFinal(unsigned state, Weight cost);
  friend void WFSAcceptor::read(const String& fileName, bool noSelfLoops = false);
  // friend void WFSAcceptor::_clear();
  friend class WFSAcceptor;

 public:
  Node(unsigned idx, Weight cost = ZeroWeight)
    : WFSAcceptor::Node(idx, cost) { }

  virtual ~Node() { }

  void writeArcs(FILE* fp = stdout);
  void writeArcs(LexiconPtr& statelex, LexiconPtr& inlex, LexiconPtr& outlex, FILE* fp = std
out);

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

  static void report() { memoryManager().report(); }

 private:
  static MemoryManager<Node>& memoryManager();
};


// ----- definition for class 'WFSTransducer::Node::Iterator' -----
//
class WFSTransducer::Node::Iterator : public WFSAcceptor::Node::Iterator {
 public:
  Iterator(const WFSTransducer::NodePtr& node)
    : WFSAcceptor::Node::Iterator(node) { }
  Iterator(WFSTransducer* wfst, WFSTransducer::NodePtr& node)
    : WFSAcceptor::Node::Iterator(wfst, node) { }
  Iterator(WFSTransducerPtr& wfst, WFSTransducer::NodePtr& node)
    : WFSAcceptor::Node::Iterator(wfst, node) { }
  ~Iterator() { }

  WFSTransducer::EdgePtr& edge() { return Cast<WFSTransducer::EdgePtr>(_edge()); }
};


// ----- definition for class 'ConfidenceEntry' -----
//
class ConfidenceEntry {
public:
  ConfidenceEntry()
    : _word(""), _weight(0.0) { }

  ConfidenceEntry(const String& word, Weight weight)
    : _word(word), _weight(weight) { }

  /* const */ String                                  _word;
  /* const */ Weight                                  _weight;
};


// ----- definition for class 'ConfidenceList' -----
//
class ConfidenceList : public Countable, public List<ConfidenceEntry, String> {
  typedef List<ConfidenceEntry, String>::Iterator       _Iterator;
public:
  class Iterator;  friend class Iterator;

  ConfidenceList(const String& nm) : List<ConfidenceEntry, String>(nm) { }
  ~ConfidenceList() { }

  void push(ConfidenceEntry entry) {
    static char sz[100];
    sprintf(sz, "%d", size());
    add(sz, entry);
  }

  Weight weight(int depth) const;
  String word(int depth) const;

  void binarize(float threshold = 0.5);

  void write(const String& fileName = "");
};

typedef refcountable_ptr<ConfidenceList>        ConfidenceListPtr;


// ----- definition for class 'ConfidenceList::Iterator' -----
//
class ConfidenceList::Iterator : public List<ConfidenceEntry, String>::Iterator {
public:
  Iterator(ConfidenceListPtr list) : List<ConfidenceEntry, String>::Iterator(*list) { }
};


// ----- definition for class 'WFSTSortedInput' -----
//
class WFSTSortedInput : public WFSTransducer {

 public:
  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTransducer::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTransducer::EdgePtr> EdgePtr;

  WFSTSortedInput(const WFSAcceptorPtr& wfsa);
  WFSTSortedInput(const WFSTransducerPtr& A, bool dynamic = false);
  WFSTSortedInput(LexiconPtr& statelex, LexiconPtr& inlex, LexiconPtr& outlex,
                  bool dynamic = false, const String& name = "WFST Sorted Input");

  virtual ~WFSTSortedInput() { }
```

```
  void fromWords(const String& words, const String& end = "</s>", const String& filler = "",
 bool clear = true);
  ConfidenceListPtr fromConfs(const String& confs, const String& end = "</s>", const String&
 filler = "");

  virtual NodePtr& initial(int idx = -1);

  NodePtr find(unsigned state, bool create = false) { return Cast<NodePtr>(_find(state, crea
te)); }
  NodePtr find(const WFSTransducer::NodePtr& node, bool create = false);

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& node);

  virtual void purgeUnique(unsigned count = 10000);

 protected:
  static const NodePtr& whiteNode();
  static const NodePtr& grayNode();
  static const NodePtr& blackNode();

  _NodeVector& _allNodes() { return Cast<_NodeVector>(_nodes); }
  _NodeMap&    _finis()    { return Cast<_NodeMap>(_final); }

  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Node* _newNode(const WFSTransducer::NodePtr& node, Color col = White,
 Weight cost = ZeroWeight);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

  virtual void _purgeUnique(unsigned count);

  unsigned                                          _findCount;
  const bool                                        _dynamic;
  WFSTransducerPtr                                  _A;

private:
  list<String> _split(const String& words);
  ConfidenceListPtr _splitConfs(const String& words);
  unsigned _highestIndex();
};

typedef WFSTSortedInput::Edge                       WFSTSortedInputEdge;
typedef WFSTSortedInput::Node                       WFSTSortedInputNode;

typedef WFSTSortedInput::EdgePtr                    WFSTSortedInputEdgePtr;
typedef WFSTSortedInput::NodePtr                    WFSTSortedInputNodePtr;

typedef Inherit<WFSTSortedInput, WFSTransducerPtr>  WFSTSortedInputPtr;


// ----- definition for class 'WFSTSortedInput::Edge' -----
//
class WFSTSortedInput::Edge : public WFSTransducer::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next,
       unsigned input, unsigned output, Weight cost = ZeroWeight)
    : WFSTransducer::Edge(prev, next, input, output, cost) { }
  virtual ~Edge() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

       NodePtr& prev()         { return Cast<NodePtr>(_prev); }
       NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTSortedInput::Node' -----
//
class WFSTSortedInput::Node : public WFSTransducer::Node {
  friend class WFSAcceptor;
  friend class WFSTSortedInput;
  friend class WFSTEpsilonRemoval;
  friend class WFSTRemoveEndMarkers;
  friend class WFSTComposition;
  friend class WFSTDeterminization;
  friend class WFSTProjection;
  friend class CombinedTransducer;
  friend class WFSTCombinedHC;

  template <class Semiring>
  friend class WFSTComp;

  template <class Semiring>
  friend class WFSTDet;

 public:
  Node(unsigned idx, Color col = White, Weight cost = ZeroWeight)
    : WFSTransducer::Node(idx, cost),
      _expanded(false), _lastEdgesCall(0), _nodeA(NULL) { setColor(col); }
  Node(const WFSTransducer::NodePtr& nodeA, Color col = White, Weight cost = ZeroWeight)
    : WFSTransducer::Node(nodeA->index(), cost),
    _expanded(false), _lastEdgesCall(0), _nodeA(nodeA) { setColor(col); }

  virtual ~Node() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

 protected:
  virtual void _addEdge(WFSAcceptor::EdgePtr& newEdge);
  virtual void _addEdgeForce(WFSAcceptor::EdgePtr& newEdge);

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

  bool                                              _expanded;      // edges have been e
xpanded
  unsigned                                          _lastEdgesCall; // last adjacency li
st request
  WFSTransducer::NodePtr                            _nodeA;

 private:
  static MemoryManager<Node>& memoryManager();
};
```

```
// ----- definition for class 'WFSTSortedInput::Node::Iterator' -----
//
class WFSTSortedInput::Node::Iterator : public WFSTransducer::Node::Iterator {
 public:
  Iterator(WFSTSortedInput* wfst, WFSTSortedInput::NodePtr& node)
    : WFSTransducer::Node::Iterator(wfst, node) { }
  Iterator(WFSTSortedInputPtr& wfst, WFSTSortedInput::NodePtr& node)
    : WFSTransducer::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTSortedInput::EdgePtr& edge() { return Cast<WFSTSortedInput::EdgePtr>(_edge()); }
};


// ----- definition for class 'WFSTRemoveEndMarkers' -----
//
class WFSTRemoveEndMarkers : public WFSTSortedInput {
 public:
  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedInput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedInput::EdgePtr> EdgePtr;

 public:
  WFSTRemoveEndMarkers(WFSTSortedInputPtr& A, const String& end = "#", bool dynamic = false,
                       const String& name = "WFST Remove Word End Markers");

  virtual ~WFSTRemoveEndMarkers() { }

  virtual NodePtr& initial(int idx = -1);

  NodePtr find(const WFSTSortedInput::NodePtr& nd, bool create = false);

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& node);

 protected:
  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

 private:
  const String                                      _end;
  WFSTSortedInputPtr                                _A;
};

typedef WFSTRemoveEndMarkers::Edge                  WFSTRemoveEndMarkersEdge;
typedef WFSTRemoveEndMarkers::Node                  WFSTRemoveEndMarkersNode;

typedef WFSTRemoveEndMarkers::EdgePtr               WFSTRemoveEndMarkersEdgePtr;
typedef WFSTRemoveEndMarkers::NodePtr               WFSTRemoveEndMarkersNodePtr;

typedef Inherit<WFSTRemoveEndMarkers, WFSTransducerPtr> WFSTRemoveEndMarkersPtr;


// ----- definition for class 'WFSTRemoveEndMarkers::Edge' -----
//
class WFSTRemoveEndMarkers::Edge : public WFSTSortedInput::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next,
       unsigned input, unsigned output, Weight cost = ZeroWeight)
    : WFSTSortedInput::Edge(prev, next, input, output, cost) { }
  virtual ~Edge() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTRemoveEndMarkers::Node' -----
//
class WFSTRemoveEndMarkers::Node : public WFSTSortedInput::Node {
  friend class WFSAcceptor;
  friend class WFSTRemoveEndMarkers;

 public:
  Node(unsigned idx, Color col = White, Weight cost = ZeroWeight);
  Node(const WFSTSortedInput::NodePtr& nodeA, Color col = White, Weight cost = ZeroWeight);

  virtual ~Node() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();
};


// ----- definition for class 'WFSTRemoveEndMarkers::Node::Iterator' -----
//
class WFSTRemoveEndMarkers::Node::Iterator : public WFSTransducer::Node::Iterator {
 public:
  Iterator(WFSTRemoveEndMarkers* wfst, WFSTRemoveEndMarkers::NodePtr& node)
    : WFSTransducer::Node::Iterator(wfst, node) { }
  Iterator(WFSTRemoveEndMarkersPtr& wfst, WFSTRemoveEndMarkers::NodePtr& node)
    : WFSTransducer::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTRemoveEndMarkers::EdgePtr& edge() { return Cast<WFSTRemoveEndMarkers::EdgePtr>(_edge()
); }
};

WFSTransducerPtr removeEndMarkers(WFSTSortedInputPtr& A);


// ----- definition for class 'WFSTSortedOutput' -----
//
```

```
class WFSTSortedOutput : public WFSTransducer {
 public:
  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTransducer::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTransducer::EdgePtr> EdgePtr;

 public:
  WFSTSortedOutput(const WFSAcceptorPtr& wfsa, bool convertFlag = true);
  WFSTSortedOutput(const WFSTransducerPtr& A);
  WFSTSortedOutput(LexiconPtr& statelex, LexiconPtr& inlex, LexiconPtr& outlex, const String
& name = "WFST Sorted Output");

  virtual ~WFSTSortedOutput() { }

  virtual NodePtr& initial(int idx = -1);

  NodePtr find(unsigned state, bool create = false) { return Cast<NodePtr>(_find(state, crea
te)); }
  NodePtr find(const WFSTransducer::NodePtr& node, bool create = false);

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& node);
  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& node, WFSTSortedInput::NodePtr& comp);

 protected:
  _NodeMap& _finis()  { return Cast<_NodeMap>(_final); }

  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

 private:
  WFSTransducerPtr                                    _A;
};

typedef WFSTSortedOutput::Edge                        WFSTSortedOutputEdge;
typedef WFSTSortedOutput::Node                        WFSTSortedOutputNode;

typedef WFSTSortedOutput::EdgePtr                     WFSTSortedOutputEdgePtr;
typedef WFSTSortedOutput::NodePtr                     WFSTSortedOutputNodePtr;

typedef Inherit<WFSTSortedOutput, WFSTransducerPtr>     WFSTSortedOutputPtr;


// ----- definition for class 'WFSTSortedOutput::Edge' -----
//
class WFSTSortedOutput::Edge : public WFSTransducer::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next,
       unsigned input, unsigned output, Weight cost = ZeroWeight)
    : WFSTransducer::Edge(prev, next, input, output, cost) { }
  virtual ~Edge() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTSortedOutput::Node' -----
//
class WFSTSortedOutput::Node : public WFSTransducer::Node {
  friend class WFSAcceptor;
  friend class WFSTSortedOutput;

 public:
  Node(unsigned idx, Weight cost = ZeroWeight)
    : WFSTransducer::Node(idx, cost), _nodeA(NULL) { }
  Node(const WFSTransducer::NodePtr& nodeA, Weight cost = ZeroWeight)
    : WFSTransducer::Node(nodeA->index(), cost), _nodeA(nodeA) { }

  virtual ~Node() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

 protected:
  virtual void _addEdge(WFSAcceptor::EdgePtr& newEdge);
  virtual void _addEdgeForce(WFSAcceptor::EdgePtr& newEdge);

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();

  WFSTransducer::NodePtr                                _nodeA;
};


// ----- definition for class 'WFSTSortedOutput::Node::Iterator' -----
//
class WFSTSortedOutput::Node::Iterator : public WFSTransducer::Node::Iterator {
 public:
  Iterator(WFSTSortedOutput* wfst, WFSTSortedOutput::NodePtr& node)
    : WFSTransducer::Node::Iterator(wfst, node) { }
  Iterator(WFSTSortedOutputPtr& wfst, WFSTSortedOutput::NodePtr& node)
    : WFSTransducer::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTSortedOutput::EdgePtr& edge() { return Cast<WFSTSortedOutput::EdgePtr>(_edge()); }
};


// ----- definition for class 'WFSTComposition' -----
//
class WFSTComposition : public WFSTSortedInput {
 public:
  static void reportMemoryUsage();

  class Node;
  class Edge;
```

```
  class Iterator;

  typedef Inherit<Node, WFSTSortedInput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedInput::EdgePtr> EdgePtr;

  virtual ~WFSTComposition() { }

  virtual NodePtr& initial(int idx = -1);

  virtual NodePtr find(const WFSTSortedOutput::NodePtr& nodeA, const WFSTSortedInput::NodePt
r& nodeB, unsigned short filter = 0) = 0;

  virtual void purgeUnique(unsigned count = 10000);

 protected:
  // C = A o B
  WFSTComposition(WFSTSortedOutputPtr& A, WFSTSortedInputPtr& B, LexiconPtr& stateLex,
                  bool dynamic = false, const String& name = "WFST Composition");

  _NodeMap& _finis()  { return Cast<_NodeMap>(_final); }

  unsigned _findIndex(unsigned stateA, unsigned stateB, unsigned short filter);

  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

  WFSTSortedOutputPtr                                   _A;
  WFSTSortedInputPtr                                    _B;

 protected:
  // unused state id bits of left state hold bits of filter state
  class _StateHashKey;
  class _State {
  public:
    _State(StateId l, StateId f, StateId r) : _l(l | (f << StateIdBits)), _r(r) {}
    StateId l() const { return _l & StateIdMask; }
    StateId f() const { return _l >> StateIdBits; }
    StateId r() const { return _r; }
    bool operator< (const _State &s) const {
      if (_l > s._l) return false;
      if (_l < s._l) return true;
      return (_r < s._r);
    }
    bool operator==(const _State &s) const { return (_l == s._l) && (_r == s._r); }

  private:
    StateId _l;
    StateId _r;
  };

  struct _StateHashKey {
    u32 operator() (const _State& s) { return 2239 * (size_t)s.l() + (size_t)s.r() + s.f();
}
  };

  class _StateHashEqual {
  public:
    u32 operator()(const _State& s1, const _State& s2) const { return s1.l() == s2.l() && s1
.r() == s2.r() && s1.f() == s2.f(); }
  };

  typedef Hash<_State, _StateHashKey, _StateHashEqual>  _IndexMap;
  typedef _IndexMap::const_iterator                     _IndexMapConstIterator;
  typedef std::pair<_IndexMap::Cursor, bool>            _IndexMapPair;
  typedef _IndexMap::Cursor                             _IndexMapCursor;

  _IndexMap                                             _indexMap;
};

typedef WFSTComposition::Edge                           WFSTCompositionEdge;
typedef WFSTComposition::Node                           WFSTCompositionNode;

typedef WFSTComposition::EdgePtr                        WFSTCompositionEdgePtr;
typedef WFSTComposition::NodePtr                        WFSTCompositionNodePtr;

typedef Inherit<WFSTComposition, WFSTSortedInputPtr>    WFSTCompositionPtr;

void WFSTComposition_reportMemoryUsage();


// ----- definition for class 'WFSTComposition::Edge' -----
//
class WFSTComposition::Edge : public WFSTSortedInput::Edge {
  friend void reportMemoryUsage();
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next, unsigned input, unsigned output, Weight cost = ZeroWeig
ht);
  virtual ~Edge();

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

  static void report() { memoryManager().report(); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTComposition::Node' -----
//
class WFSTComposition::Node : public WFSTSortedInput::Node {
  friend void reportMemoryUsage();
  friend class WFSTComposition;

  template <class Semiring>
  friend class WFSTComp;

 public:
  Node(unsigned short filter, const WFSTSortedOutput::NodePtr& nodeA, const WFSTSortedInput:
:NodePtr& nodeB,
       unsigned idx, Color col = White, Weight cost = ZeroWeight);

  virtual ~Node();

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }
```

```
  class Iterator;  friend class Iterator;

  static void report() { memoryManager().report(); }

 protected:
  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();

  unsigned short                                    _filter;        // state of composit
ion filter
  WFSTSortedOutput::NodePtr                         _nodeA;
  WFSTSortedInput::NodePtr                          _nodeB;
};


// ----- definition for class 'WFSTComposition::Node::Iterator' -----
//
class WFSTComposition::Node::Iterator : public WFSTSortedInput::Node::Iterator {
 public:
  Iterator(WFSTComposition* wfst, WFSTComposition::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }
  Iterator(WFSTCompositionPtr& wfst, WFSTComposition::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTComposition::EdgePtr& edge() { return Cast<WFSTComposition::EdgePtr>(_edge()); }
};


// ----- definition for class 'TropicalSemiring' -----
//
class TropicalSemiring {
 public:
  static Weight otimes(Weight a, Weight b) {
    return Weight(float(a) + float(b));
  }
  static Weight oplus(Weight a, Weight b) { return Weight(float(min(float(a), float(b)))); }
  static Weight inverse(Weight a) { return Weight(-float(a)); }
};


// ----- definition for class 'LogProbSemiring' -----
//
class LogProbSemiring {
 public:
  static Weight otimes(Weight a, Weight b) {
    return Weight(float(a) + float(b));
  }
  static Weight oplus(Weight ap, Weight bp) { return logAdd(ap, bp); }
  static Weight inverse(Weight a) { return Weight(-float(a)); }
};


// ----- definition for class 'WeightPusher' -----
//
class WeightPusher {
  typedef WFSTransducer::_NodeVector::iterator          _NodeVectorIterator;
  typedef WFSTransducer::_NodeVector::const_iterator    _NodeVectorConstIterator;
  typedef WFSTransducer::_NodeMap::iterator             _NodeMapIterator;
  typedef WFSTransducer::_NodeMap::const_iterator       _NodeMapConstIterator;
  typedef WFSTransducer::NodePtr                        _NodePtr;

  class _Potential {
  public:
    _Potential(const _NodePtr& node, Weight d = ZeroWeight, Weight r = ZeroWeight)
      : _node(node), _d(d), _r(r) { }

     void print() const { printf("Node %4d : d = %g : r = %g\n", _node->index(), float(_d),
 float(_r)); }

    const _NodePtr                                      _node;
    Weight                                              _d;
    Weight                                              _r;
  };

  typedef map<unsigned, _Potential>                     _PotentialMap;
  typedef _PotentialMap::iterator                       _Iterator;
  typedef _PotentialMap::const_iterator                 _ConstIterator;
  typedef _PotentialMap::value_type                     _ValueType;

  typedef queue<unsigned, list<unsigned> >              _Queue;

  typedef set<unsigned>                                 _Set;
  typedef _Set::iterator                                _SetIterator;
  typedef _Set::const_iterator                          _SetConstIterator;

  typedef LogProbSemiring                               _Semiring;

 public:
  WeightPusher(WFSTransducerPtr& wfst) : _wfst(wfst) { }
  ~WeightPusher() { }

  void push();

 private:
  void _resetPotentials(const WFSTransducerPtr& reverse);
  void _calculatePotentials(const WFSTransducerPtr& reverse);
  void _printPotentials() const;
  void _reweightNodes();
  void _reweightArcs(WFSTransducer::NodePtr& node, bool initFlag = false);
  bool _isEqual(Weight l, Weight r) {
    return (fabs(float(l) - float(r)) < 1.0e-04);
  }

  WFSTransducerPtr              _wfst;
  _PotentialMap                 _potentialMap;
  _Queue                        _queue;
  _Set                          _set;
};

void pushWeights(WFSTransducerPtr& wfst);


// ----- definition for class 'WFSTEpsilonRemoval' -----
//
class WFSTEpsilonRemoval : public WFSTSortedInput {
  typedef WFSTransducer::_NodeMap::iterator             _NodeMapIterator;
  typedef WFSTransducer::_NodeMap::const_iterator       _NodeMapConstIterator;
  typedef WFSTransducer::NodePtr                        _NodePtr;

  class _Potential {
```

```
  public:
    _Potential(const _NodePtr& node, Weight d = ZeroWeight, Weight r = ZeroWeight)
      : _node(node), _d(d), _r(r) { }

     void print() const { printf("Node %4d : d = %g : r = %g\n", _node->index(), float(_d),
 float(_r)); }

    _NodePtr                                          _node;
    Weight                                            _d;
    Weight                                            _r;
  };

  typedef map<unsigned, _Potential>                   _PotentialMap;
  typedef _PotentialMap::iterator                     _Iterator;
  typedef _PotentialMap::const_iterator               _ConstIterator;
  typedef _PotentialMap::value_type                   _ValueType;

  typedef queue<unsigned, list<unsigned> >            _Queue;

  typedef set<unsigned>                               _Set;
  typedef _Set::iterator                              _SetIterator;
  typedef _Set::const_iterator                        _SetConstIterator;

  typedef TropicalSemiring                            _Semiring;

 public:
  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedInput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedInput::EdgePtr> EdgePtr;

  WFSTEpsilonRemoval(WFSTransducerPtr& wfst) : WFSTSortedInput(wfst) { }
  ~WFSTEpsilonRemoval() { }

  virtual NodePtr& initial(int idx = -1);

  NodePtr find(unsigned state, bool create = false) { return Cast<NodePtr>(_find(state, crea
te)); }
  NodePtr find(const WFSTransducer::NodePtr& node, bool create) { return Cast<NodePtr>(WFSTS
ortedInput::find(node, create)); }

  const EdgePtr& edges(WFSAcceptor::NodePtr& nd);

 protected:
  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Node* _newNode(const WFSTransducer::NodePtr& node, Color col = White,
 Weight cost = ZeroWeight);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

 private:
  void _discoverClosure(const _NodePtr& node);
  void _calculateClosure(const _NodePtr& node);

  bool _isEqual(Weight l, Weight r) {
    return (fabs(float(l) - float(r)) < 1.0e-04);
  }

  _PotentialMap               _potentialMap;
  _Queue                      _queue;
  _Set                        _set;
};

typedef WFSTEpsilonRemoval::Edge                      WFSTEpsilonRemovalEdge;
typedef WFSTEpsilonRemoval::Node                      WFSTEpsilonRemovalNode;

typedef WFSTEpsilonRemoval::EdgePtr                   WFSTEpsilonRemovalEdgePtr;
typedef WFSTEpsilonRemoval::NodePtr                   WFSTEpsilonRemovalNodePtr;

typedef Inherit<WFSTEpsilonRemoval, WFSTSortedInputPtr> WFSTEpsilonRemovalPtr;


// ----- definition for class 'WFSTEpsilonRemoval::Edge' -----
//
class WFSTEpsilonRemoval::Edge : public WFSTSortedInput::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next,
       unsigned input, unsigned output, Weight cost = ZeroWeight)
    : WFSTSortedInput::Edge(prev, next, input, output, cost) { }
  virtual ~Edge() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTEpsilonRemoval::Node' -----
//
class WFSTEpsilonRemoval::Node : public WFSTSortedInput::Node {
  friend class WFSTEpsilonRemoval;

 public:
  Node(unsigned idx, Color col = White, Weight cost = ZeroWeight)
  : WFSTSortedInput::Node(idx, col, cost) { }
  Node(const WFSTransducer::NodePtr& nodeA, Color col = White, Weight cost = ZeroWeight)
    : WFSTSortedInput::Node(nodeA, col, cost) { }

  virtual ~Node() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

 protected:
  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();
};


// ----- definition for class 'WFSTEpsilonRemoval::Node::Iterator' -----
```

```
//
class WFSTEpsilonRemoval::Node::Iterator : public WFSTSortedInput::Node::Iterator {
 public:
  Iterator(WFSTEpsilonRemoval* wfst, WFSTEpsilonRemoval::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }
  Iterator(WFSTEpsilonRemovalPtr& wfst, WFSTEpsilonRemoval::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTEpsilonRemoval::EdgePtr& edge() { return Cast<WFSTEpsilonRemoval::EdgePtr>(_edge()); }
};

WFSTEpsilonRemovalPtr removeEpsilon(WFSTransducerPtr& wfst);


// ----- definition for class 'WFSTProjection' -----
//
class WFSTProjection : public WFSTSortedInput {
 public:
  typedef enum { Input = 0, Output } Side;

  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedInput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedInput::EdgePtr> EdgePtr;

  WFSTProjection(WFSTransducerPtr& wfst, Side side = Input);
  ~WFSTProjection() { }

  virtual NodePtr& initial(int idx = -1);

  NodePtr find(unsigned state, bool create = false) { return Cast<NodePtr>(_find(state, crea
te)); }
  NodePtr find(const WFSTransducer::NodePtr& node, bool create) { return Cast<NodePtr>(WFSTS
ortedInput::find(node, create)); }

  const EdgePtr& edges(WFSAcceptor::NodePtr& nd);

protected:
  /*
  virtual WFSAcceptor::Node* _newNode(unsigned state);
  */
  virtual WFSAcceptor::Node* _newNode(const WFSTransducer::NodePtr& node, Color col = White,
 Weight cost = ZeroWeight);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

 private:
  Side                                      _side;
};

typedef WFSTProjection::Edge                WFSTProjectionEdge;
typedef WFSTProjection::Node                WFSTProjectionNode;

typedef WFSTProjection::EdgePtr             WFSTProjectionEdgePtr;
typedef WFSTProjection::NodePtr             WFSTProjectionNodePtr;

typedef Inherit<WFSTProjection, WFSTSortedInputPtr>     WFSTProjectionPtr;
```

```
// ----- definition for class 'WFSTProjection::Edge' -----
//
class WFSTProjection::Edge : public WFSTSortedInput::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next,
       unsigned input, unsigned output, Weight cost = ZeroWeight)
    : WFSTSortedInput::Edge(prev, next, input, output, cost) { }
  virtual ~Edge() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTProjection::Node' -----
//
class WFSTProjection::Node : public WFSTSortedInput::Node {
  friend class WFSTProjection;

 public:
  Node(const WFSTransducer::NodePtr& nodeA, Color col = White, Weight cost = ZeroWeight)
  : WFSTSortedInput::Node(nodeA, col, cost) { }
  virtual ~Node() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

 protected:
  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();
};


// ----- definition for class 'WFSTProjection::Node::Iterator' -----
//
class WFSTProjection::Node::Iterator : public WFSTSortedInput::Node::Iterator {
 public:
  Iterator(WFSTProjection* wfst, WFSTEpsilonRemoval::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }
  Iterator(WFSTProjectionPtr& wfst, WFSTEpsilonRemoval::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTProjection::EdgePtr& edge() { return Cast<WFSTProjection::EdgePtr>(_edge()); }
};

WFSTProjectionPtr project(WFSTransducerPtr& wfst, const String& side = "Output");
```

```
// ----- definition for class template 'WFSTComp' -----
//
template <class Semiring>
class WFSTComp : public WFSTComposition {
 public:
  WFSTComp(WFSTSortedOutputPtr& A, WFSTSortedInputPtr& B, LexiconPtr& stateLex,
           bool dynamic = false, const String& name = "WFST Composition")
    : WFSTComposition(A, B, stateLex, dynamic, name) { }

  virtual ~WFSTComp() { }

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& nd);

  NodePtr find(unsigned state) { return Cast<NodePtr>(_find(state)); }
  NodePtr find(const WFSTSortedOutput::NodePtr& nodeA, const WFSTSortedInput::NodePtr& nodeB
, unsigned short filter = 0);
};

void barf();

template <class Semiring>
const WFSTComposition::EdgePtr& WFSTComp<Semiring>::edges(WFSAcceptor::NodePtr& nd)
{
  NodePtr& node(Cast<NodePtr>(nd));

  /*
  if (node->_nodeA.isNull() == false && node->_nodeB.isNull() == false)
    printf("Composing C = %d : A = %d : B = %d : filter = %d\n",
           node->index(), node->_nodeA->index(), node->_nodeB->index(), node->_filter);
  */

  // are edges already expanded?
  if (node->_expanded) return node->_edges();

  WFSTSortedOutput::EdgePtr edgeListA(_A->edges(node->_nodeA, node->_nodeB));
  WFSTSortedInput::EdgePtr  edgeListB(_B->edges(node->_nodeB));

  WFSTSortedOutput::EdgePtr elA;
  WFSTSortedInput::EdgePtr elB;

  WFSTSortedOutput::EdgePtr tmpOutput;
  WFSTSortedInput::EdgePtr tmpInput;

  // move ahead only on side 'A'
  if (node->_filter != 1) {
    elA = edgeListA;
    while (elA.isNull() == false && elA->output() == 0) {

      const WFSTSortedOutput::NodePtr& nodeA(elA->next());
      const WFSTSortedInput::NodePtr&  nodeB(node->_nodeB);
      NodePtr nextNode(find(nodeA, nodeB, /* filter= */ 2));

      EdgePtr edgePtr(new Edge(node, nextNode, elA->input(), elA->output(), elA->cost()));
      node->_addEdgeForce(edgePtr);

      /*
      printf("Advancing only on 'A' side:\n");
      edgePtr->write(stateLexicon(), inputLexicon(), outputLexicon());
      printf("New Node A = %d : New Node B = %d\n", nodeA->index(), nodeB->index());  fflush
(stdout);
      */

      tmpOutput = elA->_edges();  elA = tmpOutput;
    }
  }

  // move ahead only on side 'B'
  if (node->_filter != 2) {
    elB = edgeListB;
    while (elB.isNull() == false && elB->input() == 0) {

      const WFSTSortedOutput::NodePtr& nodeA(node->_nodeA);
      const WFSTSortedInput::NodePtr&  nodeB(elB->next());
      NodePtr nextNode(find(nodeA, nodeB, /* filter= */ 1));

      EdgePtr edgePtr(new Edge(node, nextNode, elB->input(), elB->output(), elB->cost()));
      node->_addEdgeForce(edgePtr);

      /*
      printf("Advancing only on 'B' side:\n");
      edgePtr->write(stateLexicon(), inputLexicon(), outputLexicon());
      printf("New Node A = %d : New Node B = %d\n", nodeA->index(), nodeB->index());  fflush
(stdout);
      */

      tmpInput = elB->_edges();  elB = tmpInput;
    }
  }

  // now move ahead on both sides simultaneously
  while (edgeListA.isNull() == false && edgeListB.isNull() == false) {

    // look for a match
    if (edgeListA->output() != edgeListB->input()) {
      if (edgeListA->output() < edgeListB->input()) {
        tmpOutput = edgeListA->_edges();  edgeListA = tmpOutput;
      } else {
        tmpInput = edgeListB->_edges();   edgeListB = tmpInput;
      }
      continue;
    }

    // can only pair null transitions in state 0
    unsigned match = edgeListA->output();
    if (match == 0 && node->_filter != 0) {
      while (edgeListA.isNull() == false && edgeListA->output() == match) {
        tmpOutput = edgeListA->_edges();  edgeListA = tmpOutput;
      }
      while (edgeListB.isNull() == false && edgeListB->input() == match) {
        tmpInput = edgeListB->_edges();    edgeListB = tmpInput;
      }
      continue;
    }

    // found a match; move forward on sides 'A' and 'B'
    elA = edgeListA;
    while (elA.isNull() == false && elA->output() == match) {
      elB = edgeListB;
      while (elB.isNull() == false && elB->input() == match) {
        Weight c = Semiring::otimes(elA->cost(), elB->cost());

        const WFSTSortedOutput::NodePtr& nodeA(elA->next());
        const WFSTSortedInput::NodePtr&  nodeB(elB->next());
        NodePtr nextNode(find(nodeA, nodeB, /* filter= */ 0));
```

```
        EdgePtr edgePtr(new Edge(node, nextNode, elA->input(), elB->output(), c));
        node->_addEdgeForce(edgePtr);

        /*
        printf("Advancing on both sides:\n");
        edgePtr->write(stateLexicon(), inputLexicon(), outputLexicon());
        printf("Node A = %d : Node B = %d\n", nodeA->index(), nodeB->index());  fflush(stdou
t);
        */

        tmpInput = elB->_edges();  elB = tmpInput;
      }
      tmpOutput = elA->_edges();  elA = tmpOutput;
    }

    // 'elA' and 'elB' now point to first elements beyond current match
    edgeListA = elA;
    edgeListB = elB;
  }

  static EdgePtr ptr;
  ptr = node->_edges();

  node->_expanded = true;  node->_lastEdgesCall = _findCount;
  if (_dynamic == false) {
    node->_nodeA = NULL;  node->_nodeB = NULL;
  }

  return ptr;
}


template <class Semiring>
WFSTComposition::NodePtr WFSTComp<Semiring>::
find(const WFSTSortedOutput::NodePtr& nodeA, const WFSTSortedInput::NodePtr& nodeB, unsigned
 short filter)
{
  ++_findCount;

  unsigned stateA   = nodeA->index();
  unsigned stateB   = nodeB->index();
  unsigned newIndex = _findIndex(stateA, stateB, filter);

  /*
  printf("WFSTComposition::find : stateA = %d : stateB = %d : filter = %d\n", stateA, stateB
, filter);
  */

  if (initial()->index() == newIndex) return initial();

  _NodeMapIterator itr = _final.find(newIndex);

  if (itr != _final.end()) {
    assert((*itr).second->isFinal() == true);
    return Cast<NodePtr>((*itr).second);
  }

  if (newIndex < _nodes.size() && _nodes[newIndex].isNull() == false) {
    NodePtr& node(Cast<NodePtr>(_nodes[newIndex]));
    if (node == whiteNode())
      node = new Node(filter, nodeA, nodeB, newIndex, White);
    else if (node == grayNode())
      node = new Node(filter, nodeA, nodeB, newIndex, Gray);
    else if (node == blackNode())
      node = new Node(filter, nodeA, nodeB, newIndex, Black);
    return node;
  }

  // Create a new state
  Weight c = Semiring::otimes(nodeA->cost(), nodeB->cost());
  NodePtr newNode(new Node(filter, nodeA, nodeB, newIndex));
  if (nodeA->isFinal() && nodeB->isFinal()) {
    newNode->_setCost(c);
    _final.insert(WFSAcceptor::_ValueType(newIndex, newNode));
    itr = _final.find(newIndex);
    return Cast<NodePtr>((*itr).second);
  } else {
    if (newIndex >= _nodes.size()) _resize(newIndex);
    _nodes[newIndex] = newNode;
    return Cast<NodePtr>(_nodes[newIndex]);
  }
}

WFSTSortedInputPtr compose(WFSTSortedOutputPtr& sortedA, WFSTSortedInputPtr& sortedB,
                           const String& semiring = "Tropical", const String& name = "Compos
ition");


// ----- definition for class 'BreadthFirstSearch' -----
//
class BreadthFirstSearch : public Countable {
 public:
  BreadthFirstSearch(unsigned purgeCount = 10000);
  virtual ~BreadthFirstSearch();

  void search(WFSTransducerPtr& A);

 protected:
  typedef WFSTransducer::EdgePtr               EdgePtr;
  typedef WFSTransducer::NodePtr               NodePtr;
  typedef WFSTransducer::Node::Iterator        Iterator;
  typedef list<NodePtr>                        _NodeQueue;
  typedef _NodeQueue::iterator                 _NodeQueueIterator;

  void _clear();
  virtual void _expandNode(WFSTransducerPtr& A, NodePtr& node);

  _NodeQueue                                   _nodeQueue;
  unsigned                                     _purgeCount;
};

typedef refcountable_ptr<BreadthFirstSearch> BreadthFirstSearchPtr;

void breadthFirstSearch(WFSTransducerPtr& A, unsigned purgeCount = 10000);


// ----- definition for class 'BreadthFirstWrite' -----
//
class BreadthFirstWrite : private BreadthFirstSearch {
 public:
  BreadthFirstWrite(const String& fileName = "", bool useSymbols = true, unsigned purgeCount
 = 10000);
  virtual ~BreadthFirstWrite();

  void write(WFSTransducerPtr& A) { search(A); }
```

```
 protected:
  virtual void _expandNode(WFSTransducerPtr& A, NodePtr& node);

 private:
  const String                                  _fileName;
  FILE*                                         _fp;
  bool                                          _useSymbols;
};

typedef refcountable_ptr<BreadthFirstWrite> BreadthFirstWritePtr;

void breadthFirstWrite(WFSTransducerPtr& A, const String& fileName, bool useSymbols, unsigne
d purgeCount = 10000);


// ----- definition for class 'DepthFirstSearch' -----
//
class DepthFirstSearch : public Countable {
 public:
  DepthFirstSearch() : _depth(0) { }
  ~DepthFirstSearch() { }

  void search(WFSTransducerPtr& A);

protected:
  typedef WFSTransducer::EdgePtr                EdgePtr;
  typedef WFSTransducer::NodePtr                NodePtr;
  typedef WFSTransducer::Node::Iterator         Iterator;

  virtual void _expandNode(WFSTransducerPtr& A, NodePtr& node);

  unsigned                                      _depth;
};

typedef refcountable_ptr<DepthFirstSearch> DepthFirstSearchPtr;

void depthFirstSearch(WFSTransducerPtr& A);


// ----- definition for class 'WFSTDeterminization' -----
//
class WFSTDeterminization : public WFSTSortedInput {
 public:
  static void reportMemoryUsage();

  static const unsigned MaxResidualSymbols = 2;

 protected:
  class Substate {
  public:
    Substate() {}
    Substate(u32 state, Weight weight, u32 output) : _state(state), _weight(weight), _output
(output) {}

    bool operator< (const Substate &s) const { return _state < s._state; }

    u32                 _state;
    Weight              _weight;
    u32                 _output;
  };

  class StateEntryListBase;
```

```
  class Substates {
  public:
    typedef u32 Cursor;

    Substates() : _bins(13, UINT_MAX), _substatesN(0) {}

    Cursor size(u8 partition) const;
    u32 hash(Cursor pos) const;
    bool equal(Cursor pos1, Cursor pos2) const;
    void appendSsubstates(const StateEntryListBase* state);
    Cursor append(const StateEntryListBase* state, bool create = false);
    size_t size() const { return _substates.size(); }
    size_t getMemoryUsed() const { return _bins.getMemoryUsed() + _substates.getMemoryUsed()
; }

  private:
    // partition byte format (AABBCCC-):
    // AA    # of bits for state: 00 = 8, 01 = 16, 10 = 24, 11 = 32
    // BB    weight: 00 = default (one()), 01 = individual, 10 = weight equal to previous su
bstate
    // CCC   # of bits for output string: 000 = 0, 001 = 8, 010 = 16, 011 = 24, 100 = 32
    // 0xff  end of substate sequence
    Vector<Cursor> _bins;
    Vector<u8> _substates;
    u32 _substatesN;
  };

  class StateEntryKey;

  class StateEntry {
    friend class StateEntryKey;
    friend void reportMemoryUsage();
  public:
#if 1
    typedef unsigned Index;
#else
    typedef unsigned short Index;
#endif

    StateEntry();
    StateEntry(const vector<Index>& residualString);
    StateEntry(const WFSTSortedInput::NodePtr& node, Weight residualWeight = ZeroWeight);
    StateEntry(const WFSTSortedInput::NodePtr& node, Weight residualWeight, const vector<Ind
ex>& residualString);

    virtual ~StateEntry() { }

          WFSTSortedInput::NodePtr& state()       { return _state; }
    const WFSTSortedInput::NodePtr& state() const { return _state; }

    Weight   residualWeight()               const { return _residualWeight; }
    unsigned residualStringSize()           const { return _residualStringSize; }
    Index    residualString(unsigned i = 0) const;
    void     setWeight(const Weight& wgt) { _residualWeight = wgt; }

    bool isFinal() const { return (_state.isNull() ? true : _state->isFinal()); }

    const String& name(const LexiconPtr& outlex) const;
    int index() const { return (_state.isNull() ? -1 : int(_state->index())); }
    Weight stateCost() const { return Weight(float((_state.isNull() ? 0.0 : _state->cost()))
); }
```

```
  void write(const LexiconPtr& outlex, FILE* fp = stdout) const;

  bool operator<(const StateEntry& st)  const {
    if (index() < st.index()) return true;
    for (unsigned i = 0; i < MaxResidualSymbols; i++) {
      Index lh = (i <    _residualStringSize) ?    _residualString[i] : 0;
      Index rh = (i < st._residualStringSize) ? st._residualString[i] : 0;
      if (lh >= rh) return false;
    }
    return true;
  }

  bool operator==(const StateEntry& st) const {
    if (_state->index() != st.index()) return false;
    if (_residualStringSize != st._residualStringSize) return false;
    for (unsigned i = 0; i < _residualStringSize; i++)
      if (_residualString[i] != st._residualString[i]) return false;
    return true;
  }

private:
  StateEntry& operator=(const StateEntry& st);

  WFSTSortedInput::NodePtr                    _state;
  Weight                                      _residualWeight;
  unsigned short                              _residualStringSize;
  Index                                       _residualString[MaxResidualSymbols];
};

class StateEntryKey {
public:
  StateEntryKey(const StateEntry& stateEntry)
    : _state(stateEntry.index()), _residualStringSize(stateEntry.residualStringSize())
  {
    for (unsigned i = 0; i < MaxResidualSymbols; i++)
      _residualString[i] = 0;

    for (unsigned i = 0; i < _residualStringSize; i++)
      _residualString[i] = stateEntry._residualString[i];
  }

  bool operator<(const StateEntryKey& k)  const {
    if (_state < k._state) return true;
    if (_state > k._state) return false;

    for (unsigned i = 0; i < MaxResidualSymbols; i++) {
      if (_residualString[i] < k._residualString[i]) return true;
      if (_residualString[i] > k._residualString[i]) return false;
    }

    return false;
  }
  bool operator==(const StateEntryKey& k) const {
    if (_state != k._state) return false;
    if (_residualStringSize != k._residualStringSize) return false;

    for (unsigned i = 0; i < _residualStringSize; i++)
      if (_residualString[i] != k._residualString[i])
        return false;

    return true;
  }
```

```
private:
  const int                                   _state;
  const unsigned short                        _residualStringSize;
  unsigned                                    _residualString[MaxResidualSymbols];
};

class StateEntryListBase {
  friend void reportMemoryUsage();

protected:
  typedef set<StateEntry>                     _StateEntryList;
  typedef _StateEntryList::const_iterator     _StateEntryListConstIterator;
  typedef _StateEntryList::value_type         _StateEntryListValueType;

public:
  virtual ~StateEntryListBase();

  virtual void add(const StateEntry& stateEntry) = 0;
  const String& name(const LexiconPtr& outlex) const;

  bool isFinal() const;
  unsigned residualStringSize(vector<unsigned>* rstr = NULL) const;
  vector<unsigned> residualString() const;
  void write(const LexiconPtr& outlex, FILE* fp = stdout) const;

  class Iterator {
  public:
    Iterator(const StateEntryListBase* list)
      : _list(list->_stateEntryList), _iter(_list.begin()) { }
    ~Iterator() { }

    bool more() const { return _iter != _list.end(); }
    const StateEntry& entry() const { return *_iter; }
    void operator++(int) { _iter++; }

  private:
    const _StateEntryList&                    _list;
    _StateEntryListConstIterator              _iter;
  };

  friend class Iterator;

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  static void report() { memoryManager().report(); }

protected:
  StateEntryListBase() { }

  static MemoryManager<StateEntryListBase>& memoryManager();

  _StateEntryList                             _stateEntryList;
};

class StateEntryWeight {
public:
  StateEntryWeight(const StateEntry& entry, Weight weight)
    : _entry(entry), _weight(weight) { }

  const StateEntry& entry() const { return _entry; }
  const Weight weight() const { return _weight; }
```

```
  void setWeight(Weight wgt) { _weight = wgt; }

private:
  const StateEntry&           _entry;
  Weight                      _weight;
};

typedef map<StateEntryKey, StateEntryWeight>  _WeightMap;
typedef _WeightMap::iterator                  _WeightMapIterator;
typedef _WeightMap::value_type                _WeightMapValueType;

class ArcEntry {
  friend void reportMemoryUsage();
public:
  ArcEntry(const StateEntry& state, const WFSTSortedInput::EdgePtr& edge)
    : _state(state), _edge(edge) { }
  ~ArcEntry() { }

  unsigned symbol() const { return _edge->input(); }
  const StateEntry& state() const { return _state; }
  const WFSTSortedInput::EdgePtr& edge() const { return _edge; }
  unsigned firstOutputSymbol() const;

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  static void report() { memoryManager().report(); }

private:
  static MemoryManager<ArcEntry>& memoryManager() {
    static MemoryManager<ArcEntry> _MemoryManager("WFSTDeterminization::ArcEntry");
    return _MemoryManager;
  }

  const StateEntry&                           _state;
  const WFSTSortedInput::EdgePtr              _edge;
};

class ArcEntryList {
  friend void reportMemoryUsage();

  typedef list<const ArcEntry*>               _ArcEntryList;
  typedef _ArcEntryList::iterator             _ArcEntryListIterator;
  typedef _ArcEntryList::const_iterator       _ArcEntryListConstIterator;
  typedef _ArcEntryList::value_type           _ArcEntryListValueType;

public:
  ArcEntryList(unsigned s) : _symbol(s) { }
  ~ArcEntryList();

  unsigned symbol() const { return _symbol; }
  void add(const ArcEntry* entry) { _arcEntryList.push_back(entry); }

  u32 hash() const { return _symbol; }

  bool hashEqual(const ArcEntryList& rhs) const { return _symbol == rhs._symbol; }

  class Iterator {
  public:
    Iterator(const ArcEntryList* list)
      : _list(list->_arcEntryList), _iter(_list.begin()) { }
    ~Iterator() { }

    bool more() const { return _iter != _list.end(); }
    const ArcEntry* entry() const { return *_iter; }
    void operator++(int) { _iter++; }

  private:
    const _ArcEntryList&                      _list;
    _ArcEntryListConstIterator                _iter;
  };
  friend class Iterator;

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  static void report() { memoryManager().report(); }

private:
  static MemoryManager<ArcEntryList>& memoryManager() {
    static MemoryManager<ArcEntryList> _MemoryManager("WFSTDeterminization::ArcEntryList")
;
    return _MemoryManager;
  }

  const unsigned                              _symbol;
  _ArcEntryList                               _arcEntryList;
};

class HashTableKey {
public:
  u32 operator()(const ArcEntryList* l) const { return l->hash(); }
};

class HashTableEqual {
public:
  u32 operator()(const ArcEntryList* l, const ArcEntryList* r) const { return l->hashEqual
(*r); }
};

class ArcEntryMap {
  friend void reportMemoryUsage();

  typedef Hash<ArcEntryList*, HashTableKey, HashTableEqual>   _ArcEntryMap;
  typedef _ArcEntryMap::const_iterator                        _ArcEntryMapConstIterator;
  typedef std::pair<_ArcEntryMap::Cursor, bool>               _HashPair;
  typedef _ArcEntryMap::Cursor                                _ArcEntryMapCursor;

public:
  ArcEntryMap() { }
  ~ArcEntryMap();

  void add(const ArcEntry* arcEntry);

  class Iterator {
  public:
    Iterator(const ArcEntryMap* arcEntryMap)
      : _arcEntryMap(arcEntryMap->_arcEntryMap), _iter(_arcEntryMap.begin()) { }
    ~Iterator() { }

    bool more() const { return _iter != _arcEntryMap.end(); }
    const ArcEntryList* list() const { return *_iter; }
    void operator++(int) { _iter++; }

  private:
    const _ArcEntryMap&                       _arcEntryMap;
```

```
      _ArcEntryMapConstIterator                _iter;
    };
    friend class Iterator;

    void* operator new(size_t sz) { return memoryManager().newElem(); }
    void  operator delete(void* e) { memoryManager().deleteElem(e); }

    static void report() { memoryManager().report(); }

  private:
    static MemoryManager<ArcEntryMap>& memoryManager() {
      static MemoryManager<ArcEntryMap> _MemoryManager("WFSTDeterminization::ArcEntryMap");
      return _MemoryManager;
    }

    _ArcEntryMap                             _arcEntryMap;
  };

 public:
  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedInput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedInput::EdgePtr> EdgePtr;

  virtual ~WFSTDeterminization() { }

  virtual NodePtr& initial(int idx = -1);

  NodePtr find(unsigned state, bool create = false) { return Cast<NodePtr>(_find(state, crea
te)); }
  NodePtr find(const StateEntryListBase* state, bool create = false);

  virtual Weight cost(const StateEntryListBase* selist) const = 0;

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& node);

  virtual void purgeUnique(unsigned count = 10000);

 protected:
  unsigned _arcSymbol(const StateEntryListBase* list) const;
  unsigned _arcSymbol(const ArcEntryList* list) const;

  WFSTDeterminization(WFSTSortedInputPtr& A, LexiconPtr& statelex, bool dynamic = false);

  _NodeMap& _finis()  { return Cast<_NodeMap>(_final); }

 private:
  void _purgeUnique(unsigned count);
  virtual void _addArc(NodePtr& node, const ArcEntryList* arcEntryList) = 0;
  virtual void _addArcToEnd(NodePtr& node) = 0;
  virtual const StateEntryListBase* _initialStateEntryList(WFSTSortedInputPtr& A) = 0;

  WFSTSortedInputPtr                         _A;
  Substates                                  _substates;
};

typedef WFSTDeterminization::Edge            WFSTDeterminizationEdge;
typedef WFSTDeterminization::Node            WFSTDeterminizationNode;

typedef WFSTDeterminization::EdgePtr         WFSTDeterminizationEdgePtr;
typedef WFSTDeterminization::NodePtr         WFSTDeterminizationNodePtr;

typedef Inherit<WFSTDeterminization, WFSTSortedInputPtr> WFSTDeterminizationPtr;

void WFSTDeterminization_reportMemoryUsage();


// ----- definition for class 'WFSTDeterminization::Edge' -----
//
class WFSTDeterminization::Edge : public WFSTSortedInput::Edge {
  friend void reportMemoryUsage();
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next,
       unsigned input, unsigned output, Weight cost = ZeroWeight)
    : WFSTSortedInput::Edge(prev, next, input, output, cost) { }
  virtual ~Edge() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

  static void report() { memoryManager().report(); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTDeterminization::Node' -----
//
class WFSTDeterminization::Node : public WFSTSortedInput::Node {
  friend void reportMemoryUsage();
  friend class WFSTDeterminization;

 public:
  Node(const StateEntryListBase* stateEntryList, unsigned idx, Color col = White, Weight cos
t = ZeroWeight)
    : WFSTSortedInput::Node(idx, col, cost), _stateEntryList(stateEntryList) { }

  virtual ~Node() { delete _stateEntryList; }

  ArcEntryMap* arcEntryMap(WFSTSortedInputPtr& wfst);

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;       friend class Iterator;

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

  const StateEntryListBase* stateEntryList() const { return _stateEntryList; }

  static void report() { memoryManager().report(); }

 private:
  static MemoryManager<Node>& memoryManager();
```

```
  const StateEntryListBase*                              _stateEntryList;
};


// ----- definition for class 'WFSTDeterminization::Node::Iterator' -----
//
class WFSTDeterminization::Node::Iterator : public WFSTSortedInput::Node::Iterator {
 public:
  Iterator(WFSTDeterminization* wfst, WFSTDeterminization::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }
  Iterator(WFSTDeterminizationPtr& wfst, WFSTDeterminization::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTDeterminization::EdgePtr& edge() { return Cast<WFSTDeterminization::EdgePtr>(_edge()); }
 };
};


// ----- definition for class template 'WFSTDet' -----
//
template <class Semiring>
class WFSTDet : public WFSTDeterminization {
  class StateEntryList : public StateEntryListBase {
   public:
    StateEntryList() { }
    virtual ~StateEntryList() { }

    virtual void add(const StateEntry& stateEntry);
  };

 public:
  WFSTDet(WFSTSortedInputPtr& A, LexiconPtr& statelex, bool dynamic = false)
    : WFSTDeterminization(A, statelex, dynamic) { }

  virtual ~WFSTDet() { }

  const StateEntryListBase* _initialStateEntryList(WFSTSortedInputPtr& A);

  virtual Weight cost(const StateEntryListBase* selist) const;

 private:
  Weight _arcWeight(const ArcEntryList* list) const;
  Weight _arcWeight(const StateEntryListBase* list) const;
  virtual void _addArc(NodePtr& node, const ArcEntryList* arcEntryList);
  virtual void _addArcToEnd(NodePtr& node);
};


// ----- methods for helper class 'WFSTDet::StateEntryList' -----
//
template<class Semiring>
void WFSTDet<Semiring>::StateEntryList::add(const StateEntry& stateEntry)
{
  _StateEntryListConstIterator itr = _stateEntryList.find(stateEntry);

  if (itr == _stateEntryList.end()) {
    _stateEntryList.insert(stateEntry);
  } else {
    StateEntry& entry(Cast<StateEntry>(*itr));
    entry.setWeight(Semiring::oplus(entry.residualWeight(), stateEntry.residualWeight()));
  }
}

template<class Semiring>
void WFSTDet<Semiring>::_addArcToEnd(NodePtr& node)
{
  Weight           arcWeight(cost(node->stateEntryList()));
  vector<unsigned> residualString(node->stateEntryList()->residualString());

  assert(residualString.size() > 0);

  bool                create             = true;
  unsigned            arcSymbol          = residualString[0];
  StateEntryListBase* nextStateEntryList = new StateEntryList();
  int                 residualSize       = int(residualString.size()) - 1;

  if (residualSize < 0)
    throw jconsistency_error("Size of residual string (%d) < 0.", residualSize);

  if (residualSize > MaxResidualSymbols)
    throw jconsistency_error("Size of residual string (%d) > %d.", residualSize, MaxResidual
Symbols);

  if (residualSize > 0) {
    vector<StateEntry::Index> residual(residualSize);
    for (int i = 0; i < residualSize; i++)
      residual[i] = residualString[i + 1];

    StateEntry stateEntry(residual);
    nextStateEntryList->add(stateEntry);
  } else {
    StateEntry stateEntry;
    nextStateEntryList->add(stateEntry);
  }

  NodePtr nextNode(find(nextStateEntryList, create));
  EdgePtr newEdge(new Edge(node, nextNode, 0, arcSymbol, arcWeight));

  node->_addEdgeForce(newEdge);
}


// ----- methods for class 'WFSTDet' -----
//
template<class Semiring>
const WFSTDeterminization::StateEntryListBase*
WFSTDet<Semiring>::_initialStateEntryList(WFSTSortedInputPtr& A)
{
  WFSTDeterminization::StateEntryListBase* stateEntryList = new StateEntryList();

  StateEntry stateEntry(A->initial());
  stateEntryList->add(stateEntry);

  return stateEntryList;
}

template<class Semiring>
Weight WFSTDet<Semiring>::cost(const StateEntryListBase* selist) const
{
  Weight c(LogZeroWeight);
  bool  foundFinal = false;
  for (StateEntryListBase::Iterator itr(selist); itr.more(); itr++) {
    if (itr.entry().isFinal() == false) continue;
    const WFSTSortedInput::NodePtr& node(itr.entry().state());
```

```
    foundFinal = true;
    Weight stateWeight = itr.entry().residualWeight();
    if (node.isNull() == false)
      stateWeight = Semiring::otimes(stateWeight, node->cost());
    c = Semiring::oplus(c, stateWeight);
  }

  return foundFinal ? c : ZeroWeight;
}

template <class Semiring>
void WFSTDet<Semiring>::_addArc(NodePtr& node, const ArcEntryList* arcEntryList)
{
  Weight   arcWeight  = _arcWeight(arcEntryList);
  unsigned arcSymbol  = _arcSymbol(arcEntryList);
  unsigned arcSymbolN = (arcSymbol == 0) ? 0 : 1;

  // create a new state corresponding to a 'StateEntryListBase'
  bool create = true;
  StateEntryListBase* stateEntryList = new StateEntryList();
  for (ArcEntryList::Iterator itr(arcEntryList); itr.more(); itr++) {
    const ArcEntry* entry(itr.entry());

    Weight rw             = Semiring::otimes(entry->state().residualWeight(), entry->edge()-
>cost());
    Weight iaw            = Semiring::inverse(arcWeight);
    Weight residualWeight = Semiring::otimes(rw, iaw);
    int    residualSize   = entry->state().residualStringSize() - arcSymbolN;
    int    allocSize      = residualSize + ((entry->edge()->output() != 0) ? 1 : 0);

    if (allocSize < 0)
      throw jconsistency_error("Size of residual string (%d) < 0.", allocSize);

    if (allocSize > MaxResidualSymbols)
      throw jconsistency_error("Size of residual string (%d) > %d.", allocSize, MaxResidualS
ymbols);

    if (allocSize > 0) {
      vector<StateEntry::Index> residualString(allocSize);

      for (int i = 0; i < residualSize; i++)
        residualString[i] = entry->state().residualString(i + arcSymbolN);

      if (entry->edge()->output() != 0)
        residualString[residualSize] = entry->edge()->output();

      StateEntry stateEntry(entry->edge()->next(), residualWeight, residualString);
      stateEntryList->add(stateEntry);
    } else {
      StateEntry stateEntry(entry->edge()->next(), residualWeight);
      stateEntryList->add(stateEntry);
    }

  }

  NodePtr nextNode(find(stateEntryList, create));
  EdgePtr newEdge(new Edge(node, nextNode, arcEntryList->symbol(), arcSymbol, arcWeight));
  node->_addEdgeForce(newEdge);

  /*
  printf("New State Entry List:\n");
  printf("%s\n", stateEntryList->name(outputLexicon()).c_str());
  printf("New Edge:\n");
  newEdge->write(stateLexicon(), inputLexicon(), outputLexicon());
  fflush(stdout);
  */
}

template <class Semiring>
Weight WFSTDet<Semiring>::_arcWeight(const ArcEntryList* aelist) const
{
  _WeightMap weightMap;
  for (ArcEntryList::Iterator itr(aelist); itr.more(); itr++) {
    const ArcEntry* entry(itr.entry());
    /*
    unsigned input  = entry->edge()->input();
    unsigned output = entry->edge()->output();

    const StateEntry& stateEntry(entry->state());
    unsigned sindex = stateEntry.state()->index();
    */

    StateEntryKey key(entry->state());
    _WeightMapIterator itr = weightMap.find(key);

    if (itr == weightMap.end()) {
      StateEntryWeight stateEntryWeight(entry->state(), entry->edge()->cost());
      weightMap.insert(_WeightMapValueType(key, stateEntryWeight));
    } else {
      StateEntryWeight& stateEntryWeight((*itr).second);
      Weight weight(Semiring::oplus(stateEntryWeight.weight(), entry->edge()->cost()));
      stateEntryWeight.setWeight(weight);
    }
  }

  Weight arcWgt(LogZeroWeight);
  for (_WeightMapIterator itr = weightMap.begin(); itr != weightMap.end(); itr++) {
    StateEntryWeight& stateEntryWeight((*itr).second);
    arcWgt = Semiring::oplus(arcWgt, Semiring::otimes(stateEntryWeight.entry().residualWeigh
t(), stateEntryWeight.weight()));
  }

  return arcWgt;
}

WFSTDeterminizationPtr determinize(WFSTransducerPtr& A, const String& semiring = "Tropical",
 unsigned count = 10000);

WFSTDeterminizationPtr determinize(WFSTSortedInputPtr& sortedA, const String& semiring = "Tr
opical", unsigned count = 10000);


// ----- definition for class 'WFSTIndexed' -----
//
class WFSTIndexed : public WFSTSortedOutput {

  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedOutput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedOutput::EdgePtr> EdgePtr;

  typedef list<EdgePtr>                          _EdgeList;
  typedef _EdgeList::iterator                    _EdgeListIterator;
```

```
  typedef map<unsigned, _EdgeList>              _EdgeMap;
  typedef _EdgeMap::iterator                    _EdgeMapIterator;
  typedef _EdgeMap::value_type                  _EdgeMapValueType;

  typedef vector<NodePtr>                       _NodeVector;
  typedef _NodeVector::iterator                 _NodeVectorIterator;
  typedef _NodeVector::const_iterator           _ConstNodeVectorIterator;

  typedef map<unsigned, NodePtr >               _NodeMap;
  typedef _NodeMap::iterator                    _NodeMapIterator;
  typedef _NodeMap::const_iterator              _ConstNodeMapIterator;
  typedef _NodeMap::value_type                  _ValueType;

 public:
  WFSTIndexed(const WFSAcceptorPtr& wfsa, bool convertWFSA = true);

  WFSTIndexed(LexiconPtr& statelex, LexiconPtr& inlex,
              const String& grammarFile = "", const String& name = "WFST Grammar");

  WFSTIndexed(LexiconPtr& statelex, LexiconPtr& inlex, LexiconPtr& outlex, const String& nam
e = "WFST Grammar");
  virtual ~WFSTIndexed() { }

  virtual void read(const String& fileName, bool noSelfLoops = false);

  void setLattice(WFSTSortedInputPtr& lattice) { _lattice = lattice; }

  virtual NodePtr& initial(int idx = -1);

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& nd);
  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& nd, WFSTSortedInput::NodePtr& comp);

 protected:
  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

 private:
  void _convert(const WFSAcceptorPtr& wfsa);
  void _indexEdges();
  void _expandNode(NodePtr& node, WFSTSortedInput::NodePtr& latticeNode);
  _NodeMap&    _finis()    { return Cast<_NodeMap>(_final); }
  _NodeVector& _allNodes() { return Cast<_NodeVector>(_nodes); }

  WFSTSortedInputPtr                            _lattice;
};

typedef Inherit<WFSTIndexed, WFSTSortedOutputPtr> WFSTIndexedPtr;


// ----- definition for class 'WFSTIndexed::Edge' -----
//
class WFSTIndexed::Edge : public WFSTSortedOutput::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next, unsigned input, unsigned output, Weight cost = ZeroWeig
ht);
  virtual ~Edge();

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTIndexed::Node' -----
//
class WFSTIndexed::Node : public WFSTSortedOutput::Node {
  friend class WFSTIndexed;
 public:
  Node(unsigned idx, Weight cost = ZeroWeight)
    : WFSTSortedOutput::Node(idx, cost) { }
  virtual ~Node() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

 protected:
  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();

  void _indexEdges();

  _EdgeMap                                      _edgeMap;
  WFSTSortedInput::NodePtr                      _latticeNode;
};


// ----- definition for class 'WFSTIndexed::Node::Iterator' -----
//
class WFSTIndexed::Node::Iterator : public WFSTSortedOutput::Node::Iterator {
 public:
  Iterator(WFSTIndexed* wfst, WFSTIndexed::NodePtr& node)
    : WFSTSortedOutput::Node::Iterator(wfst, node) { }
  Iterator(WFSTIndexedPtr& wfst, WFSTIndexed::NodePtr& node)
    : WFSTSortedOutput::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTIndexed::EdgePtr& edge() { return Cast<WFSTIndexed::EdgePtr>(_edge()); }
};


// ----- definition for class 'WFSTLexicon' -----
//
class WFSTLexicon : public WFSTSortedOutput {

  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedOutput::NodePtr> NodePtr;
```

```
  typedef Inherit<Edge, WFSTSortedOutput::EdgePtr> EdgePtr;

  typedef map<String, unsigned>                 _PronunciationList;
  typedef _PronunciationList::iterator          _PronunciationListIterator;
  typedef _PronunciationList::value_type        _PronunciationListValueType;

  typedef list<EdgePtr>                         _EdgeList;
  typedef _EdgeList::iterator                   _EdgeListIterator;
  typedef _EdgeList::const_iterator             _EdgeListConstIterator;

 public:
  WFSTLexicon(LexiconPtr& statelex, LexiconPtr& inlex, LexiconPtr& outlex,
              const String& sil = "SIL", const String& breath = "{+BREATH+:WB}", const Strin
g& eos = "</s>", const String& end = "#",
              unsigned maxWordN = 65535, const String& lexiconFile = "", bool epsilonToBranc
h = false,
              const String& name = "WFST Lexicon");
  virtual ~WFSTLexicon();

  // read the lexicon
  virtual void read(const String& fileName, bool tclFormat = false);

  // set the grammar
  void setGrammar(WFSTSortedInputPtr& grammar) { _grammar = grammar; }

  virtual NodePtr& initial(int idx = -1);

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& nd);
  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& nd, WFSTSortedInput::NodePtr& grammarNo
de);

 private:
  void          _clear();
  void          _addWord(const String& line);
  list<String>  _parsePronunciation(const String& pronunciation);
  unsigned      _determineEndSymbol(const String& pronunciation);
  void          _expandNode(NodePtr& node, WFSTSortedInput::NodePtr& grammarNode);

  WFSTSortedInputPtr                            _grammar;
  const unsigned                                _sil;
  const unsigned                                _breath;
  const unsigned                                _eosOutput;
  const String                                  _end;
  const unsigned                                _maxWordN;
        unsigned                                _maxEndX;
  const bool                                    _epsilonToBranch;
  _PronunciationList                            _pronunciationList;
  _EdgeList*                                    _branchList;
  unsigned                                      _nodesN;
  NodePtr                                       _branch;
  NodePtr                                       _endButOne;
  NodePtr                                       _endNode;
};

typedef Inherit<WFSTLexicon, WFSTSortedOutputPtr> WFSTLexiconPtr;


// ----- definition for class 'WFSTLexicon::Edge' -----
//
class WFSTLexicon::Edge : public WFSTSortedOutput::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next, unsigned input, unsigned output, Weight cost = ZeroWeig
ht);
  virtual ~Edge();

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTLexicon::Node' -----
//
class WFSTLexicon::Node : public WFSTSortedOutput::Node {
  friend class WFSTLexicon;
 public:
  Node(unsigned idx, bool isBranch = false, Weight cost = ZeroWeight)
    : WFSTSortedOutput::Node(idx, cost), _isBranch(isBranch) { }
  virtual ~Node() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

 protected:
  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();

  const bool                                    _isBranch;
  WFSTSortedInput::NodePtr                      _grammarNode;
};


// ----- definition for class 'WFSTLexicon::Node::Iterator' -----
//
class WFSTLexicon::Node::Iterator : public WFSTSortedOutput::Node::Iterator {
 public:
  Iterator(WFSTLexicon* wfst, WFSTLexicon::NodePtr& node)
    : WFSTSortedOutput::Node::Iterator(wfst, node) { }
  Iterator(WFSTLexiconPtr& wfst, WFSTLexicon::NodePtr& node)
    : WFSTSortedOutput::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTLexicon::EdgePtr& edge() { return Cast<WFSTLexicon::EdgePtr>(_edge()); }
};


// ----- definition for class 'WFSTContextDependency' -----
//
class WFSTContextDependency : public WFSTSortedOutput {
  class StateName : public Countable {
  public:
```

```
    StateName(WFSTSortedInputPtr& dict, unsigned contextLen);
    StateName(const StateName& src);
    ~StateName();

    void* operator new(size_t sz) { return memoryManager().newElem(); }
    void  operator delete(void* e) { memoryManager().deleteElem(e); }

    inline StateName& operator=(const StateName& s);

    inline StateName operator+(unsigned shift) const;
    inline StateName operator+(const String& shift) const;

    unsigned phone() const { return _names[_len]; }
    unsigned left(unsigned l) const;
    unsigned right(unsigned l) const;
    unsigned rightMost() const { return _names[2*_len-1]; }

    String name(unsigned rc) const;
    String name(const String& rc = "") const;
    operator String() const { return name(); }

    unsigned index() const;

    static void report() { memoryManager().report(); }

  private:
    static const unsigned                       MaxContextLength;

    static MemoryManager<StateName>& memoryManager() {
      static MemoryManager<StateName> _MemoryManager("StateName");
      return _MemoryManager;
    }

    WFSTSortedInputPtr                          _dict;
    unsigned short                              _len;
    unsigned short                              _names[6];
  };

  friend class StateName;

  typedef refcountable_ptr<StateName>           StateNamePtr;

 public:
  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedOutput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedOutput::EdgePtr> EdgePtr;

  WFSTContextDependency(LexiconPtr& statelex, LexiconPtr& inlex, WFSTSortedInputPtr& dict,
                        unsigned contextLen = 1, const String& sil = "SIL", const String& ep
s = "eps", const String& eos = "</s>",
                        const String& end = "#", const String& wb = "WB", const String& name
 = "WFST Context Dependency");

  virtual ~WFSTContextDependency() { }

  virtual NodePtr& initial(int idx = -1);

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& nd);
  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& node, WFSTSortedInput::NodePtr& comp);

 protected:
  _NodeMap& _finis()  { return Cast<_NodeMap>(_final); }

  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

  WFSTSortedInputPtr                            _dict;

 private:
  void _addSelfLoops(NodePtr& node, WFSTSortedInput::NodePtr& dictNode);
  void _expandNode(NodePtr& node, WFSTSortedInput::NodePtr& dictNode);

  const unsigned                                _contextLen;
  const unsigned                                _silInput;
  const unsigned                                _silOutput;
  const unsigned                                _eosOutput;
  const String                                  _eps;
  const String                                  _end;
  const String                                  _wb;
};

typedef WFSTContextDependency::Edge                             WFSTContextDependencyEdge;
typedef WFSTContextDependency::Node                             WFSTContextDependencyNode;

typedef WFSTContextDependency::EdgePtr                          WFSTContextDependencyEdgePtr
;
typedef WFSTContextDependency::NodePtr                          WFSTContextDependencyNodePtr
;

typedef Inherit<WFSTContextDependency, WFSTSortedOutputPtr>     WFSTContextDependencyPtr;


// ----- definition for class 'WFSTContextDependency::Edge' -----
//
class WFSTContextDependency::Edge : public WFSTSortedOutput::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next, unsigned input, unsigned output, Weight cost = ZeroWeig
ht);
  virtual ~Edge();

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTContextDependency::Node' -----
//
class WFSTContextDependency::Node : public WFSTSortedOutput::Node {
  friend class WFSTContextDependency;
 public:
  Node(const StateName& stateName, Weight cost = ZeroWeight)
```

```
    : WFSTSortedOutput::Node(stateName.index(), cost), _dictNode(NULL), _stateName(stateName
) { }

  virtual ~Node() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

 protected:
  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();

  WFSTSortedInput::NodePtr                      _dictNode;
  StateName                                     _stateName;
};


// ----- definition for class 'WFSTContextDependency::Node::Iterator' -----
//
class WFSTContextDependency::Node::Iterator : public WFSTSortedOutput::Node::Iterator {
 public:
  Iterator(WFSTContextDependency* wfst, WFSTContextDependency::NodePtr& node)
    : WFSTSortedOutput::Node::Iterator(wfst, node) { }
  Iterator(WFSTContextDependencyPtr& wfst, WFSTContextDependency::NodePtr& node)
    : WFSTSortedOutput::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTContextDependency::EdgePtr& edge() { return Cast<WFSTContextDependency::EdgePtr>(_edge
()); }
};


// ----- definition for class 'WFSTHiddenMarkovModel' -----
//
class WFSTHiddenMarkovModel : public WFSTSortedOutput {

  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedOutput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedOutput::EdgePtr> EdgePtr;

  typedef map<unsigned, EdgePtr>                _EdgeMap;
  typedef _EdgeMap::iterator                    _EdgeMapIterator;
  typedef _EdgeMap::value_type                  _EdgeMapValueType;

 public:
  WFSTHiddenMarkovModel(LexiconPtr& statelex, LexiconPtr& inlex, LexiconPtr& outlex, const D
istribTreePtr& distribTree,
                        bool caching = false, const String& sil = "SIL", const String& eps =
 "eps", const String& end = "#",
                        const String& name = "WFST hidden Markov model");
  ~WFSTHiddenMarkovModel();

  void setContext(WFSTSortedInputPtr& context) { _context = context; }

  virtual NodePtr& initial(int idx = -1);

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& nd);

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& nd, WFSTSortedInput::NodePtr& comp);

 private:
  const String& _transSymbol(const NodePtr& node);
  void _addSelfLoops(NodePtr& startNode, WFSTSortedInput::NodePtr& contextNode);
  void _expandPhone(NodePtr& node);
  void _expandSilence(NodePtr& node, unsigned nStates = 4);
  void _expandNode(NodePtr& node, WFSTSortedInput::NodePtr& contextNode);

  const DistribTreePtr                          _distribTree;
  WFSTSortedInputPtr                            _context;
  const bool                                    _caching;
  const unsigned                                _silInput;
  const unsigned                                _silOutput;
  const String                                  _eps;
  const String                                  _end;
  _EdgeMap                                      _edgeMap;
};

typedef Inherit<WFSTHiddenMarkovModel, WFSTSortedOutputPtr> WFSTHiddenMarkovModelPtr;


// ----- definition for class 'WFSTHiddenMarkovModel::Edge' -----
//
class WFSTHiddenMarkovModel::Edge : public WFSTSortedOutput::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next, unsigned input, unsigned output, Weight cost = ZeroWeig
ht);
  virtual ~Edge();

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTHiddenMarkovModel::Node' -----
//
class WFSTHiddenMarkovModel::Node : public WFSTSortedOutput::Node {
  friend class WFSTHiddenMarkovModel;
 public:
  Node(unsigned output = 0, bool isSilence = false, unsigned short idx = 0);

  virtual ~Node() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;
```

```
 protected:
  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();

  static unsigned _convertIndex(unsigned polyphoneX, unsigned short stateX);

  const bool                                          _isSilence;
  const unsigned                                      _output;
  const unsigned short                                _state;
  WFSTSortedInput::NodePtr                            _context;
};


// ----- definition for class 'WFSTHiddenMarkovModel::Node::Iterator' -----
//
class WFSTHiddenMarkovModel::Node::Iterator : public WFSTSortedOutput::Node::Iterator {
 public:
  Iterator(WFSTHiddenMarkovModel* wfst, WFSTHiddenMarkovModel::NodePtr& node)
    : WFSTSortedOutput::Node::Iterator(wfst, node) { }
  Iterator(WFSTHiddenMarkovModelPtr& wfst, WFSTHiddenMarkovModel::NodePtr& node)
    : WFSTSortedOutput::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTHiddenMarkovModel::EdgePtr& edge() { return Cast<WFSTHiddenMarkovModel::EdgePtr>(_edge
()); }
};


// ----- definition for class 'WFSTCompare' -----
//
class WFSTCompare : public WFSTSortedInput {
  static const double CostTolerance;
 public:
  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedInput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedInput::EdgePtr> EdgePtr;

  WFSTCompare(LexiconPtr& statelex, LexiconPtr& inlex, LexiconPtr& outlex, WFSTSortedInputPt
r& comp)
    : WFSTSortedInput(statelex, inlex, outlex), _comp(comp) { }

  virtual ~WFSTCompare() { }

  virtual NodePtr& initial(int idx = -1);

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& nd);

 private:
  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

  WFSTSortedInputPtr                                  _comp;
};

typedef WFSTCompare::Edge                             WFSTCompareEdge;
typedef WFSTCompare::Node                             WFSTCompareNode;
typedef WFSTCompare::EdgePtr                          WFSTCompareEdgePtr;
typedef WFSTCompare::NodePtr                          WFSTCompareNodePtr;

typedef Inherit<WFSTCompare, WFSTSortedInputPtr>      WFSTComparePtr;


// ----- definition for class 'WFSTCompare::Edge' -----
//
class WFSTCompare::Edge : public WFSTSortedInput::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next,
       unsigned input, unsigned output, Weight cost = ZeroWeight)
    : WFSTSortedInput::Edge(prev, next, input, output, cost) { }
  virtual ~Edge() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTCompare::Node' -----
//
class WFSTCompare::Node : public WFSTSortedInput::Node {
  friend class WFSTCompare;
 public:
  Node(unsigned idx, Color col = White, Weight cost = ZeroWeight)
    : WFSTSortedInput::Node(idx, col, cost), _compNode(NULL) { }

  virtual ~Node() { }

  void setComp(WFSTSortedInput::NodePtr& cmp) { _compNode = cmp; }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

 protected:
  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();

  WFSTSortedInput::NodePtr                        _compNode;
};


// ----- definition for class 'WFSTCompare::Node::Iterator' -----
//
class WFSTCompare::Node::Iterator : public WFSTSortedInput::Node::Iterator {
 public:
```

```
  Iterator(WFSTCompare* wfst, WFSTCompare::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }
  Iterator(WFSTComparePtr& wfst, WFSTCompare::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }

  ~Iterator() { }
};


//
// Note: The following class template would be more readable if
//       the nested class definitions were moved outside the main
//       class definition. Unfortunately, this causes g++ to
//       seg fault. Perhaps this bug will be fixed in a later
//       release. (jmcd, May 15, 2004)
//
// ----- definition for class template `WFST' -----
//
template <class NodeType, class ArcType>
class WFST : public WFSTransducer {
 public:
  class Node;
  class Edge;

  typedef Inherit<Node, WFSTransducer::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTransducer::EdgePtr> EdgePtr;
  /* class Iterator; */

 protected:
  typedef vector<NodePtr>                       _NodeVector;
  typedef typename _NodeVector::iterator        _NodeVectorIterator;
  typedef typename _NodeVector::const_iterator  _ConstNodeVectorIterator;

  typedef map<unsigned, NodePtr >               _NodeMap;
  typedef typename _NodeMap::iterator           _NodeMapIterator;
  typedef typename _NodeMap::const_iterator     _ConstNodeMapIterator;
  typedef typename _NodeMap::value_type         _ValueType;

 public:


  // ----- definition for class template `WFST::Node' -----
  //
  class Node : public WFSTransducer::Node {
  public:
    Node(unsigned idx)
      : WFSTransducer::Node(idx, ZeroWeight) { }

    Node(unsigned idx, const NodeType& d, Weight cost = ZeroWeight)
      : WFSTransducer::Node(idx, cost), _data(d) { }

    virtual ~Node() { }

    void* operator new(size_t sz) { return memoryManager().newElem(); }
    void  operator delete(void* e) { memoryManager().deleteElem(e); }

    virtual void write(FILE* fp = stdout, bool writeData = false);
    virtual void write(LexiconPtr& stateLex, FILE* fp = stdout, bool writeData = false);

    virtual void addEdge(EdgePtr& ed)      { WFSAcceptor::Node::_addEdge(ed);      }
    virtual void addEdgeForce(EdgePtr& ed) { WFSAcceptor::Node::_addEdgeForce(ed); }

          NodeType& data()       { return _data; }
    const NodeType& data() const { return _data; }

    // void _addFinal(unsigned state, Weight cost) { WFSAcceptor::_addFinal(state, cost); }


    // ----- definition for class template `WFST::Node::Iterator' -----
    //
    class Iterator : public WFSTransducer::Node::Iterator {
    public:
      Iterator(const NodePtr& node)
        : WFSTransducer::Node::Iterator(node) { }

      ~Iterator() { }

      EdgePtr& edge() { return Cast<EdgePtr>(_edge()); }
      EdgePtr next() {
        if (!more())
          throw jiterator_error("end of edges!");

        EdgePtr ed(edge());
        operator++(1);
        return ed;
      }
    };

    static void report() { memoryManager().report(); }

  protected:
    EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

  private:
    static MemoryManager<Node>& memoryManager();

    NodeType    _data;
  };


  // ----- definition for class template `WFST::Edge' -----
  //
  class Edge : public WFSTransducer::Edge {
  public:
    Edge(NodePtr& prev, NodePtr& next,
         unsigned input, unsigned output, const ArcType& d, Weight cost = ZeroWeight)
      : WFSTransducer::Edge(prev, next, input, output, cost), _data(d) { }
    virtual ~Edge() { }

    void* operator new(size_t sz) { return memoryManager().newElem(); }
    void  operator delete(void* e) { memoryManager().deleteElem(e); }

          NodePtr& prev()       { return Cast<NodePtr>(_prev); }
          NodePtr& next()       { return Cast<NodePtr>(_next); }
    const NodePtr& prev() const { return Cast<NodePtr>(_prev); }
    const NodePtr& next() const { return Cast<NodePtr>(_next); }

    virtual void write(FILE* fp = stdout, bool writeData = false);
    virtual void write(LexiconPtr& stateLex, LexiconPtr& inputLex,
                       LexiconPtr& outputLex, FILE* fp = stdout, bool writeData = false);

          ArcType& data()       { return _data; }
    const ArcType& data() const { return _data; }

    EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }
```

```
    static void report() { memoryManager().report(); }

  private:
    static MemoryManager<Edge>& memoryManager();

    ArcType _data;
  };

  WFST(LexiconPtr& statelex, LexiconPtr& inlex, LexiconPtr& outlex)
    : WFSTransducer(statelex, inlex, outlex) { }
  virtual ~WFST() { }

  NodePtr& initial(int idx = -1);

  NodePtr find(unsigned state, bool create = false) { return Cast<NodePtr>(_find(state, crea
te)); }

  void addFinal(unsigned state, Weight cost) {
    WFSAcceptor::_addFinal(state, cost);
  }

  // read parameterized transducer
  virtual void read(const String& fileName, bool noSelfLoops = false, bool readData = false)
;

  // write parameterized transducer
  virtual void write(const String& fileName = "", bool useSymbols = false, bool writeData =
false);

  /*
    friend class Iterator;d
  */

 protected:
  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

  _NodeVector& _allNodes() { return Cast<_NodeVector>(_nodes); }
  _NodeMap&    _finis()    { return Cast<_NodeMap>(_final); }
};


// ----- methods for class template 'WFST' -----
//
template <class NodeType, class ArcType>
typename WFST<NodeType, ArcType>::NodePtr& WFST<NodeType, ArcType>::initial(int idx)
{
  if (_initial.isNull()) _initial = ((idx >= 0) ? new Node(idx) : new Node(0));

  return Cast<NodePtr>(_initial);
}

template <class NodeType, class ArcType>
WFSAcceptor::Node* WFST<NodeType, ArcType>::_newNode(unsigned state)
{
  return new Node(state, NodeType());
}

template <class NodeType, class ArcType>
WFSAcceptor::Edge* WFST<NodeType, ArcType>::
_newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned output, Weight cost)
{
  return new Edge(from, to, input, output, ArcType(), cost);
}

template <class NodeType, class ArcType>
void WFST<NodeType, ArcType>::read(const String& fileName, bool noSelfLoops, bool readData)
{
  if (fileName == "")
    jio_error("File name is null.");

  _clear();

  printf("\nReading WFST from file %s\n", fileName.c_str());

  FILE* fp = fileOpen(fileName, "r");
  size_t  n;
  static char* buffer = NULL;

  while(getline(&buffer, &n, fp) > 0) {
    static char* token[6];
    token[0] = strtok(buffer, " \t\n");
    unsigned s1;
    sscanf(token[0], "%u", &s1);
    unsigned i = 0;
    while((i < 5) && ((token[++i] = strtok(NULL, " \t\n")) != NULL) );

    if (i == 1) {                       // add a final state with zero cost

      _addFinal(s1);

      if (readData) {
        NodePtr nd(find(s1));
        nd->data().read(fp);
      }

      // printf("Added final node %d.\n", s1);  fflush(stdout);

    } else if (i == 2) {                // add a final state with non-zero cost

      float cost;
      sscanf(token[1], "%f", &cost);

      _addFinal(s1, Weight(cost));

      if (readData) {
        NodePtr nd(find(s1));
        nd->data().read(fp);
      }

      // printf("Added final node %d with cost %g.\n", s1, cost);  fflush(stdout);

    } else if (i == 4 || i == 5) {      // add an arc

      bool create = true;
      unsigned s2;
      sscanf(token[1], "%u", &s2);

      if (s1 == s2 && noSelfLoops) continue;

      NodePtr from(find(s1, create));
      NodePtr to(find(s2, create));

      char* p = NULL;
      unsigned input = strtoul(token[2], &p, 0);
```

```
      if (p == token[2])
        input = _inputLexicon->index(token[2]);

      p = NULL;
      unsigned output = strtoul(token[3], &p, 0);
      if (p == token[3])
        output = outputLexicon()->index(token[3]);

      if (s1 == s2 && input == 0) continue;

      float cost = 0.0;
      if (i == 5)
        sscanf(token[4], "%f", &cost);

      EdgePtr edgePtr((Edge*) _newEdge(from, to, input, output, Weight(cost)));
      from->addEdgeForce(edgePtr);  _totalEdges++;

      // printf("Added arc from %d to %d with cost %g.\n", s1, s2, cost);  fflush(stdout);

      if (readData)
        edgePtr->data().read(fp);

    } else
      throw jio_error("Transducer file %s is inconsistent.", fileName.chars());
  }

  fileClose( fileName, fp);
}

template <class NodeType, class ArcType>
void WFST<NodeType, ArcType>::write(const String& fileName, bool useSymbols, bool writeData)
{
  FILE* fp = stdout;
  if (fileName != "")
    fp = fileOpen(fileName, "w");

  // write edges leaving from initial state
  for (typename Node::Iterator itr(initial()); itr.more(); itr++)
    if (useSymbols)
      itr.edge()->write(stateLexicon(), inputLexicon(), outputLexicon(), fp, writeData);
    else
      itr.edge()->write(fp, writeData);

  // write edges leaving from intermediate states
  for (_ConstNodeVectorIterator itr = _allNodes().begin(); itr != _allNodes().end(); itr++)
{
    const NodePtr& nd(*itr);
    if (nd.isNull()) continue;

    for (typename Node::Iterator itr(nd); itr.more(); itr++)
      if (useSymbols)
        itr.edge()->write(stateLexicon(), inputLexicon(), outputLexicon(), fp, writeData);
      else
        itr.edge()->write(fp, writeData);
  }

  // write final states
  for (_NodeMapIterator itr=_finis().begin(); itr != _finis().end(); itr++) {
    NodePtr& nd((*itr).second);

    // write edges
    for (typename Node::Iterator itr(nd); itr.more(); itr++)
      if (useSymbols)
        itr.edge()->write(stateLexicon(), inputLexicon(), outputLexicon(), fp, writeData);
      else
        itr.edge()->write(fp, writeData);

    // write nodes
    if (useSymbols)
      nd->write(stateLexicon(), fp, writeData);
    else
      nd->write(fp, writeData);
  }

  if (fp != stdout)
    fileClose( fileName, fp);
}


// ----- methods for class template 'WFST::Node' -----
//
template <class NodeType, class ArcType>
void WFST<NodeType, ArcType>::Node::write(FILE* fp, bool writeData)
{
  WFSTransducer::Node::write(fp);

  if (writeData)
    _data.write(fp);
}

template <class NodeType, class ArcType>
void WFST<NodeType, ArcType>::Node::write(LexiconPtr& stateLex, FILE* fp,
                                          bool writeData)
{
  WFSTransducer::Node::write(stateLex, fp);

  if (writeData)
    _data.write(fp);
}


// ----- methods for class template 'WFST::Edge' -----
//
template <class NodeType, class ArcType>
void WFST<NodeType, ArcType>::Edge::write(FILE* fp, bool writeData)
{
  WFSTransducer::Edge::write(fp);

  if (writeData)
    _data.write(fp);
}

template <class NodeType, class ArcType>
void WFST<NodeType, ArcType>::Edge::write(LexiconPtr& stateLex, LexiconPtr& inlex, LexiconPt
r& outlex, FILE* fp,
                                          bool writeData)
{
  WFSTransducer::Edge::write(stateLex, inlex, outlex, fp);

  if (writeData)
    _data.write(fp);
}


// ----- definition for class 'ContextDependencyTransducer' -----
//
```

```
class ContextDependencyTransducer {
  static LexiconPtr                         _phoneLexicon;
  static WFSTransducerPtr                   _dict;
  class StateNameList;
  class StateName {
    friend class StateNameList;
  public:
    StateName(unsigned contextLen, const String& beg = "#");
    StateName(const StateName& src);
    ~StateName();

    void* operator new(size_t sz) { return memoryManager().newElem(); }
    void  operator delete(void* e) { memoryManager().deleteElem(e); }

    inline StateName& operator=(const StateName& s);

    inline StateName operator+(unsigned shift) const;
    inline StateName operator+(const String& shift) const;

    const String& phone() const { return _phoneLexicon->symbol(_names[_len]); }
    inline const String& left(unsigned l) const;
    inline const String& right(unsigned l) const;

    String name(unsigned rc) const;
    String name(const String& rc = "") const;
    operator String() const { return name(); }
    bool rightContextContains(const String& sym) const;
    bool rightMostContextContains(const String& sym) const;

    unsigned depth() const { return _depth; }
    void incDepth() { _depth++; }

  private:
    static const unsigned                   MaxContextLength;

    static MemoryManager<StateName>& memoryManager() {
      static MemoryManager<StateName> _MemoryManager("StateName");
      return _MemoryManager;
    }

    unsigned short                          _len;
    unsigned short                          _names[6];
    unsigned                                _depth;
    StateName*                              _next;
  };

  friend class StateName;

  class StateNameList {

    typedef set<unsigned>                   _IndexSet;
    typedef _IndexSet::iterator             _IndexSetIterator;
    typedef _IndexSet::const_iterator       _IndexSetConstIterator;

  public:
    StateNameList();
    ~StateNameList();

    void push(const StateName& stateName, unsigned index);
    StateName* pop();
    bool isPresent(unsigned index) const;
    void clear();
    unsigned size() const { return _indexSet.size(); }

  private:
    StateName*                              _stateName;
    _IndexSet                               _indexSet;
  };

  typedef WFSTransducer::Edge               Edge;
  typedef WFSTransducer::EdgePtr            EdgePtr;
  typedef WFSTransducer::Node               Node;
  typedef WFSTransducer::NodePtr            NodePtr;

 public:
  ContextDependencyTransducer(unsigned contextLen = 1,
                              const String& sil = "SIL", const String& eps = "eps",
                              const String& end = "#", const String& wb = "WB", const String
& eos = "</s>");
  ~ContextDependencyTransducer() { }

  WFSTransducerPtr build(LexiconPtr& lexicon, WFSTransducerPtr& dict, const String& name = "
C");

 private:
  static unsigned                           _cnt;

  void _addSelfLoops(WFSTransducerPtr& wfst, NodePtr& oldNode, const NodePtr& dictNode) cons
t;
  void _expandToEnd(WFSTransducerPtr& wfst, StateName stateName);
  void _expandNode(WFSTransducerPtr& wfst, const StateName& stateName, const NodePtr& dictNo
de);

  const unsigned                            _contextLen;
  const String                              _sil;
  const String                              _eps;
  const String                              _end;
  const String                              _wb;
  const String                              _eos;
  NodePtr                                   _branch;

  StateNameList                             _stateNameList;
};

// build context-dependency transducer
WFSTransducerPtr
buildContextDependencyTransducer(LexiconPtr& inputLexicon, WFSTransducerPtr& dict, unsigned
contextLen = 1,
                                 const String& sil = "SIL", const String& eps = "eps", const
 String& end = "#",
                                 const String& wb = "WB", const String& eos = "</s>");


// ----- definition for class 'HiddenMarkovModelTransducer' -----
//
class HiddenMarkovModelTransducer {

  typedef WFSTransducer::Edge               Edge;
  typedef WFSTransducer::EdgePtr            EdgePtr;
  typedef WFSTransducer::Node               Node;
  typedef WFSTransducer::NodePtr            NodePtr;

 public:
  HiddenMarkovModelTransducer(const String& sil = "SIL", const String& eps = "eps", const St
ring& end = "#")
    : _sil(sil), _eps(eps), _end(end) { }
```

```
  ~HiddenMarkovModelTransducer() { }

  WFSTransducerPtr build(LexiconPtr& inputLexicon, LexiconPtr& outputLexicon,
                         const DistribTreePtr& distribTree, unsigned noEnd = 1, const String
& name = "H") const;

 private:
  void _addSelfLoops(WFSTransducerPtr& wfst, NodePtr& startNode, unsigned noEnd) const;
  void _expandSilence(WFSTransducerPtr& wfst, const DistribTreePtr& dt,
                      NodePtr& startNode, NodePtr& finalNode, unsigned nStates = 4) const;
  void _expandPhone(WFSTransducerPtr& wfst, const String& outSymbol, const DistribTreePtr& d
t,
                    NodePtr& startNode, NodePtr& finalNode) const;

  const String                                  _sil;
  const String                                  _eps;
  const String                                  _end;
};

WFSTransducerPtr buildHiddenMarkovModelTransducer(LexiconPtr& inputLexicon, LexiconPtr& outp
utLexicon,
                                                  const DistribTreePtr& distribTree, unsigne
d noEnd = 1,
                                                  const String& sil = "SIL", const String& e
ps = "eps",
                                                  const String& end = "#");

// ----- definition for class 'CombinedTransducer' -----
//
class CombinedTransducer {

  friend class WFSTCombinedHC;

  typedef WFSTSortedInput::Edge                         Edge;
  typedef WFSTSortedInput::EdgePtr                      EdgePtr;
  typedef WFSTSortedInput::Node                         Node;
  typedef WFSTSortedInput::NodePtr                      NodePtr;

public:
  typedef DistribTree::_BitMatrix                       _BitMatrix;
  typedef DistribTree::_BitMatrixList                   _BitMatrixList;
  typedef DistribTree::_BitMatrixListIterator           _BitMatrixListIterator;
  typedef DistribTree::_BitMatrixListConstIterator      _BitMatrixListConstIterator;

public:
  typedef list<String>                          ds;             // distribution sequence

  class seqrec {
    public:
      seqrec(const String& ds1, const String& ds2, const _BitMatrixList& bmatlist)
        : bmlist(bmatlist)
    {
      seq.push_back(ds1);  seq.push_back(ds2);
    }

    ds                                          seq;
    _BitMatrixList                              bmlist;
  };

  typedef list<seqrec>                          ldsb;           // list of distribution sequ
ences with bitmap lists
  typedef map<String, ldsb>                     ldsbp;          // list of distribution sequ
ences with bitmap lists keyed by a phone

  typedef ds::iterator                          dsIterator;
  typedef ldsb::iterator                        ldsbIterator;
  typedef ldsbp::iterator                       ldsbpIterator;

  typedef ldsbp::value_type                     ldsbpValueType;

  typedef list<unsigned>                        _SymbolList;
  typedef _SymbolList::iterator                 _SymbolListIterator;
  typedef _SymbolList::const_iterator           _SymbolListConstIterator;

  class _Metastate;
  typedef refcountable_ptr<_Metastate>          _MetastatePtr;

  typedef list<_MetastatePtr>                   _MetastateList;
  typedef _MetastateList::iterator              _MetastateListIterator;
  typedef _MetastateList::const_iterator        _MetastateListConstIterator;
  typedef _MetastateList::value_type            _MetastateListValueType;

  class _Metastate : public Countable {
  public:
    // default constructor
    _Metastate() {}
    _Metastate(const NodePtr& bNode, const NodePtr& eNode, unsigned output, const _SymbolLis
t& symbolList, const _BitMatrixList& bmlist)
      : _beginNode(bNode), _endNode(eNode), _outputSymbol(output), _symbolList(symbolList),
_bitMatrixList(bmlist) {}

    // accessor methods
    unsigned outputSymbol() const               { return _outputSymbol; }
    const _BitMatrixList& bitMatrixList() const { return _bitMatrixList; }
    const _SymbolList& symbolList() const       { return _symbolList; }
    const unsigned beginSymbol() const          { return _symbolList.front(); }
    const unsigned endSymbol() const            { return _symbolList.back(); }
    NodePtr& beginNode()                        { return _beginNode; }
    NodePtr& endNode()                          { return _endNode; }

    _MetastateList& metastateList()             { return _metastateList; }
    const _BitMatrix& bitMask() const           { return _mask; }
    void setBitMask(const _BitMatrix& mask)     { _mask = mask; }
    vector<unsigned> hash() const               { return _hash; }

    String symbols(unsigned phoneX = 0) const;

  private:
    unsigned                                    _outputSymbol;
    _BitMatrixList                              _bitMatrixList;
    _SymbolList                                 _symbolList;
    NodePtr                                     _beginNode;
    NodePtr                                     _endNode;
    _BitMatrix                                  _mask;
    _MetastateList                              _metastateList;
    vector<unsigned>                            _hash;
  };

  typedef multimap<String, _MetastatePtr>       _MetastateMap;
  typedef _MetastateMap::iterator               _MetastateMapIterator;
  typedef _MetastateMap::const_iterator         _MetastateMapConstIterator;
  typedef _MetastateMap::value_type             _MetastateMapValueType;

  typedef map<String, _MetastatePtr>            _MetastateSingleMap;
  typedef _MetastateMap::iterator               _MetastateSingleMapIterator;
```

```
  typedef _MetastateMap::const_iterator         _MetastateSingleMapConstIterator;
  typedef _MetastateMap::value_type             _MetastateSingleMapValueType;

  typedef multimap<unsigned, _MetastatePtr>     _MetastateSet;
  typedef _MetastateSet::iterator               _MetastateSetIterator;
  typedef _MetastateSet::const_iterator         _MetastateSetConstIterator;
  typedef _MetastateSet::value_type             _MetastateSetValueType;

  class _StateSequenceEntry { // the "Metastate" class
  public:
    _StateSequenceEntry(unsigned begInput, unsigned begOutput, const _BitMatrixList& bmlist,
 NodePtr& bnode, NodePtr& enode)
      : _begInput(begInput), _begOutput(begOutput), _bitMatrixList(bmlist), _begNode(bnode),
 _endNode(enode){ }
    _StateSequenceEntry(){_begInput = 0; _begOutput = 0; _begNode = 0; _endNode = 0;}

    unsigned input()  const { return _begInput;  }
    unsigned output() const { return _begOutput; }
    const _BitMatrixList& bitMatrixList() const { return _bitMatrixList; }
    NodePtr& begNode() { return _begNode; }
    NodePtr& endNode() { return _endNode; }

  private:
    unsigned                                    _begInput;
    unsigned                                    _begOutput;
    _BitMatrixList                              _bitMatrixList;
    NodePtr                                     _begNode;
    NodePtr                                     _endNode;
  };

  typedef list<_StateSequenceEntry>             _StateSequenceList;
  typedef _StateSequenceList::iterator          _StateSequenceIterator;

public:
  CombinedTransducer(unsigned contextLen = 2, const String& sil = "SIL", const String& eps =
 "eps", const String& end = "#", const String& eos = "</s>")
    : _contextLen(contextLen), _sil(sil), _eps(eps), _end(end), _eos(eos) { }

  ~CombinedTransducer() { }

  WFSTSortedInputPtr build(LexiconPtr& distribLexicon, DistribTreePtr& distribTree, LexiconP
tr& phoneLexicon,
                           unsigned endN = 1, bool correct = true, const String& name = "HC"
);

private:
  void _constructBitmaps();
  void _checkBitmaps();
  void _enumStateSequences(WFSTSortedInputPtr& wfst);
  void _enumSilenceStates(WFSTSortedInputPtr& wfst, unsigned statesN = 4);
  void _connectStateSequences(WFSTSortedInputPtr& wfst);
  void _connectStateSequencesEx(WFSTSortedInputPtr& wfst);
  _MetastatePtr _findMetastate(_MetastateMap& listT, const _MetastatePtr& s, const _BitMatri
xList& listL);
  _MetastatePtr _createMetastate(WFSTSortedInputPtr& wfst, const _MetastatePtr& s, const _Bi
tMatrixList& listL);

  void _addSelfLoops(WFSTSortedInputPtr& wfst, NodePtr& startNode) const;

  void _firstPass();
  void _secondPass(WFSTSortedInputPtr& wfst);
  void _secondPassEx(WFSTSortedInputPtr& wfst);
  static _BitMatrixList _reduce(const _BitMatrixList& src, const _BitMatrix& mask);
  void _calcBitMasks();
  void _iterateBitMasks();
  bool _checkBitMatrix(const _BitMatrix& bm, const String& phone);
  bool _checkBitMatrixList(const _BitMatrixList& bmlist, const String& phone);

  LexiconPtr _extractPhoneLex(LexiconPtr phoneLex);

  unsigned                                      _contextLen;
  DistribTree::_BitmapList                      _leafBitmaps;
  ldsbp                                         _validStateSequences; // phone − list of sta
te ID sequences (i.e.AH:  AH(|)−b(223)−−>AH(|)−m(54)−−>AH(|)−e(65))
  const String                                  _sil;
  const String                                  _eps;
  const String                                  _end;
  const String                                  _eos;

  _StateSequenceList                            _stateSequenceList;
  unsigned                                      _nodeCount;
  _MetastateSet                                 _listS;
  _MetastateSingleMap                           _mapS;
  _MetastateMap                                 _listT;

  LexiconPtr                                    _distribLexicon;
  DistribTreePtr                                _distribTree;
  LexiconPtr                                    _phoneLexicon;
  LexiconPtr                                    _phoneLexiconOutput;
  unsigned                                      _endN;
  String                                        _name;
  bool                                          _correct;

};

WFSTSortedInputPtr buildCombinedTransducer(LexiconPtr& distribLexicon, DistribTreePtr& distr
ibTree, LexiconPtr& phoneLexicon,
                                           unsigned endN = 1, const String& sil = "SIL", con
st String& eps = "eps", const String& end = "#",
                                           const String& eos = "</s>", bool correct = true);


// ----- definition for class 'WFSTCombinedHC' -----
//
class WFSTCombinedHC : public WFSTSortedInput {
 public:
  static void reportMemoryUsage();

  typedef enum{Begin = 0, Middle = 1, End = 2, Unknown = 3} NodeType;

  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedInput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedInput::EdgePtr> EdgePtr;

  typedef DistribTree::_BitMatrix                       _BitMatrix;
  class _BitMatrixList : public Countable, public DistribTree::_BitMatrixList {
  public:
    _BitMatrixList() {}
    _BitMatrixList(int count, const _BitMatrix& bm) : DistribTree::_BitMatrixList(count, bm)
 { }
    _BitMatrixList(const DistribTree::_BitMatrixList& other) : DistribTree::_BitMatrixList(o
ther) { }
  };
```

```
  typedef refcountable_ptr<_BitMatrixList>      _BitMatrixListPtr;

  typedef _BitMatrixList::iterator              _BitMatrixListIterator;
  typedef _BitMatrixList::const_iterator        _BitMatrixListConstIterator;

  typedef list<unsigned>                        _SymbolList;
  typedef _SymbolList::iterator                 _SymbolListIterator;
  typedef _SymbolList::const_iterator           _SymbolListConstIterator;

  class _Metastate;
  typedef refcountable_ptr<_Metastate>          _MetastatePtr;

  typedef list<_MetastatePtr>                   _MetastateList;
  typedef _MetastateList::iterator              _MetastateListIterator;
  typedef _MetastateList::const_iterator        _MetastateListConstIterator;
  typedef _MetastateList::value_type            _MetastateListValueType;

  // own metastate with a pointer to a bit matrix list
  class _Metastate : public Countable {
    friend class WFSTCombinedHC;
  public:
    // default constructor
    _Metastate() {}
    _Metastate(const NodePtr& bNode, const NodePtr& eNode, unsigned output, const _SymbolLis
t& symbolList, const _BitMatrixListPtr& bmlist)
      : _beginNode(bNode), _endNode(eNode), _outputSymbol(output), _symbolList(symbolList),
_bitMatrixList(bmlist) {}

    // accessor methods
    unsigned outputSymbol() const                 { return _outputSymbol; }
          _BitMatrixListPtr& bitMatrixList()      { return _bitMatrixList; }
    const _BitMatrixListPtr& bitMatrixList() const { return _bitMatrixList; }
    const _SymbolList& symbolList() const         { return _symbolList; }
    const unsigned beginSymbol() const            { return _symbolList.front(); }
    const unsigned endSymbol() const              { return _symbolList.back(); }
    NodePtr& beginNode()                          { return _beginNode; }
    NodePtr& endNode()                            { return _endNode; }

    _MetastateList& metastateList()             { return _metastateList; }
    const _BitMatrix& bitMask() const           { return _mask; }
    void setBitMask(const _BitMatrix& mask)     { _mask = mask; }
    vector<unsigned> hash() const               { return _hash; }

    String symbols(unsigned phoneX = 0) const;

  private:
    unsigned                                    _outputSymbol;
    _BitMatrixListPtr                           _bitMatrixList;
    _SymbolList                                 _symbolList;
    NodePtr                                     _beginNode;
    NodePtr                                     _endNode;
    _BitMatrix                                  _mask;
    _MetastateList                              _metastateList;
    vector<unsigned>                            _hash;
  };

  typedef multimap<String, _MetastatePtr>       _MetastateMap;
  typedef _MetastateMap::iterator               _MetastateMapIterator;
  typedef _MetastateMap::const_iterator         _MetastateMapConstIterator;
  typedef _MetastateMap::value_type             _MetastateMapValueType;

  typedef map<String, _MetastatePtr>            _MetastateSingleMap;
  typedef _MetastateMap::iterator               _MetastateSingleMapIterator;
  typedef _MetastateMap::const_iterator         _MetastateSingleMapConstIterator;
  typedef _MetastateMap::value_type             _MetastateSingleMapValueType;

  typedef multimap<unsigned, _MetastatePtr>     _MetastateSet;
  typedef _MetastateSet::iterator               _MetastateSetIterator;
  typedef _MetastateSet::const_iterator         _MetastateSetConstIterator;
  typedef _MetastateSet::value_type             _MetastateSetValueType;


  WFSTCombinedHC(LexiconPtr& distribLexicon, DistribTreePtr& distribTree,
                 LexiconPtr& stateLexicon, LexiconPtr& phoneLexicon,
                 unsigned endN = 1, const String& sil = "SIL", /*const unsigned silStates =
4, */
                 const String& eps = "eps", const String& end = "#",
                 const String& eos = "</s>", bool correct = true, unsigned hashKeys = 1,
                 bool approximateMatch = false, bool dynamic = false);

  virtual ~WFSTCombinedHC() { }

  virtual NodePtr& initial(int idx = -1);

  NodePtr find(unsigned state, bool create = false) { return Cast<NodePtr>(_find(state, crea
te)); }

  const EdgePtr& edges(WFSAcceptor::NodePtr& nd);

 protected:
  virtual void _purgeUnique(unsigned count = 10000);

 private:
  virtual WFSAcceptor::Node* _newNode(const unsigned state,
                                      const NodeType& type = Unknown,
                                      const _MetastatePtr&  metastate = NULL);
  // stub needed for find()
  virtual WFSAcceptor::Node* _newNode(const unsigned state) {return _newNode(state, Unknown,
 NULL); };

  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

  LexiconPtr  _extractPhoneLex(LexiconPtr phoneLex);
  void        _expandNode(NodePtr& node);
  void        _connectToFinal(NodePtr& node);

  void _constructBitmaps();
  void _enumStateSequences();
  void _enumSilenceStates(unsigned statesN = 4);

  _MetastatePtr _findMetastate(const _MetastatePtr& s, const _BitMatrixListPtr& listL);
  _MetastatePtr _createMetastate(const _MetastatePtr& s, const _BitMatrixListPtr& listL);

  void _addSelfLoops(NodePtr& startNode);

  void _firstPass();
  void _secondPassEx();

  _BitMatrixListPtr _reduce(const _BitMatrixListPtr& src, const _BitMatrix& mask);
  void _calcBitMasks();

  bool _checkBitMatrix(const _BitMatrix& bm, const String& phone);
  bool _checkBitMatrixList(const _BitMatrixListPtr& bmlist, const String& phone);
```

```
  vector<unsigned> _hash(const _BitMatrixListPtr& listL);
  vector<unsigned> _calcRunLengths(const _BitMatrixListPtr& listL);

  unsigned                      _contextLen;
  String                        _silInput;

  unsigned                      _nodeCount;       // keep track of the current node count wh
en creating new nodes
  NodePtr                       _finalNode;       // there's only one final node in the comb
ined HC

  const String                  _sil;
  const String                  _eps;
  const String                  _end;
  const String                  _eos;
  LexiconPtr                    _distribLexicon;
  LexiconPtr                    _phoneLexicon;
  LexiconPtr                    _phoneLexiconOutput;
  unsigned                      _endN;
  bool                          _correct;
  unsigned                      _hashKeys;

  DistribTree::_BitmapList                  _leafBitmaps;
  CombinedTransducer::ldsbp                 _validStateSequences;
  CombinedTransducer::_StateSequenceList    _stateSequenceList;
  _MetastateSet                             _listS;
  _MetastateSingleMap                       _mapS;

  _MetastatePtr                             _silCopy;

  _MetastateMap                             _listT;
  bool                                      _approximateMatch;
};

typedef Inherit<WFSTCombinedHC, WFSTSortedInputPtr> WFSTCombinedHCPtr;


// ----- definition for class 'WFSTCombinedHC::Edge' -----
//
class WFSTCombinedHC::Edge : public WFSTSortedInput::Edge {
  // friend void reportMemoryUsage();
  // friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next, unsigned input, unsigned output, Weight cost = ZeroWeig
ht)
    : WFSTSortedInput::Edge(prev, next, input, output, cost) { }
  virtual ~Edge() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

  static void report() { memoryManager().report(); }

 private:
  static MemoryManager<Edge>& memoryManager();
};
```

```
// ----- definition for class 'WFSTCombinedHC::Node' -----
//
class WFSTCombinedHC::Node : public WFSTSortedInput::Node {
  friend void reportMemoryUsage();
  friend class WFSTCombinedHC;

 public:
  Node(unsigned idx, const NodeType& type = Unknown, const _MetastatePtr& metastate = NULL)
    : WFSTSortedInput::Node(idx), _type(type), _metastate(metastate){ }

  virtual ~Node() {}

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

  static void report() { memoryManager().report(); }

 protected:
  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();

  NodeType   _type;                             // node type in the metastate
  _MetastatePtr _metastate;                     // the metastate to which this node belongs
};

// ----- definition for class 'WFSTCombinedHC::Node::Iterator' -----
//
class WFSTCombinedHC::Node::Iterator : public WFSTSortedInput::Node::Iterator {
 public:
  Iterator(WFSTCombinedHC* wfst, WFSTCombinedHC::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }
  Iterator(WFSTCombinedHCPtr& wfst, WFSTCombinedHC::NodePtr& node)
    : WFSTSortedInput::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTCombinedHC::EdgePtr& edge() { return Cast<WFSTCombinedHC::EdgePtr>(_edge()); }
};

WFSTDeterminizationPtr
buildDeterminizedHC(LexiconPtr& distribLexicon, DistribTreePtr& distribTree, LexiconPtr& pho
neLexicon,
                    unsigned endN = 1, const String& sil = "SIL", const String& eps = "eps",
 const String& end = "#",
                    const String& eos = "</s>", bool correct = true, unsigned hashKeys = 1,
bool approximateMatch = false, bool dynamic = false, unsigned count = 10000);

WFSTransducerPtr buildHC(LexiconPtr& distribLexicon, DistribTreePtr& distribTree, LexiconPtr
& phoneLexicon,
                         unsigned endN = 1, const String& sil = "SIL", const String& eps = "
eps", const String& end = "#",
                         const String& eos = "</s>", bool correct = true, unsigned hashKeys
= 1, bool approximateMatch = false, bool dynamic = false);


// ----- definition for class 'WFSTAddSelfLoops' -----
//
```

```
class WFSTAddSelfLoops : public WFSTSortedInput {
 public:
  class Node;
  class Edge;
  class Iterator;

  typedef Inherit<Node, WFSTSortedInput::NodePtr> NodePtr;
  typedef Inherit<Edge, WFSTSortedInput::EdgePtr> EdgePtr;

 public:
  WFSTAddSelfLoops(WFSTSortedInputPtr& A, const String& end = "#", unsigned endN = 1, bool d
ynamic = false,
                       const String& name = "WFST Add Self Loops");

  virtual ~WFSTAddSelfLoops() { }

  virtual NodePtr& initial(int idx = -1);

  NodePtr find(const WFSTSortedInput::NodePtr& nd, bool create = false);

  virtual const EdgePtr& edges(WFSAcceptor::NodePtr& node);

  void _addSelfLoops(NodePtr& startNode);

 protected:
  virtual WFSAcceptor::Node* _newNode(unsigned state);
  virtual WFSAcceptor::Edge* _newEdge(NodePtr& from, NodePtr& to, unsigned input, unsigned o
utput, Weight cost = ZeroWeight);

 private:
  const String                                        _end;
  const unsigned                                      _endN;
  WFSTSortedInputPtr                                  _A;
};

typedef WFSTAddSelfLoops::Edge                        WFSTAddSelfLoopsEdge;
typedef WFSTAddSelfLoops::Node                        WFSTAddSelfLoopsNode;

typedef WFSTAddSelfLoops::EdgePtr                     WFSTAddSelfLoopsEdgePtr;
typedef WFSTAddSelfLoops::NodePtr                     WFSTAddSelfLoopsNodePtr;

typedef Inherit<WFSTAddSelfLoops, WFSTransducerPtr>   WFSTAddSelfLoopsPtr;


// ----- definition for class 'WFSTAddSelfLoops::Edge' -----
//
class WFSTAddSelfLoops::Edge : public WFSTSortedInput::Edge {
  friend void _addFinal(unsigned state, Weight cost);
 public:
  Edge(NodePtr& prev, NodePtr& next,
       unsigned input, unsigned output, Weight cost = ZeroWeight)
    : WFSTSortedInput::Edge(prev, next, input, output, cost) { }
  virtual ~Edge() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

        NodePtr& prev()         { return Cast<NodePtr>(_prev); }
        NodePtr& next()         { return Cast<NodePtr>(_next); }
  const NodePtr& prev()   const { return Cast<NodePtr>(_prev); }
  const NodePtr& next()   const { return Cast<NodePtr>(_next); }

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Edge>& memoryManager();
};


// ----- definition for class 'WFSTAddSelfLoops::Node' -----
//
class WFSTAddSelfLoops::Node : public WFSTSortedInput::Node {
  friend class WFSAcceptor;
  friend class WFSTAddSelfLoops;

 public:
  Node(unsigned idx, Color col = White, Weight cost = ZeroWeight);
  Node(const WFSTSortedInput::NodePtr& nodeA, Color col = White, Weight cost = ZeroWeight);

  virtual ~Node() { }

  void* operator new(size_t sz) { return memoryManager().newElem(); }
  void  operator delete(void* e) { memoryManager().deleteElem(e); }

  class Iterator;  friend class Iterator;

  EdgePtr& _edges() { return Cast<EdgePtr>(_edgeList); }

 private:
  static MemoryManager<Node>& memoryManager();
};


// ----- definition for class 'WFSTAddSelfLoops::Node::Iterator' -----
//
class WFSTAddSelfLoops::Node::Iterator : public WFSTransducer::Node::Iterator {
 public:
  Iterator(WFSTAddSelfLoops* wfst, WFSTAddSelfLoops::NodePtr& node)
    : WFSTransducer::Node::Iterator(wfst, node) { }
  Iterator(WFSTAddSelfLoopsPtr& wfst, WFSTAddSelfLoops::NodePtr& node)
    : WFSTransducer::Node::Iterator(wfst, node) { }

  ~Iterator() { }

  WFSTAddSelfLoops::EdgePtr& edge() { return Cast<WFSTAddSelfLoops::EdgePtr>(_edge()); }
};


// ----- definition for class 'MinimizeFSA' -----
//
class MinimizeFSA {
  typedef WFSAcceptor::NodePtr                NodePtr;
  typedef WFSAcceptor::Edge                   Edge;
  typedef WFSAcceptor::EdgePtr                EdgePtr;
  typedef WFSAcceptor::Node::Iterator         NodeIterator;

  typedef set<unsigned>                       _Block;
  typedef _Block::iterator                    _BlockIterator;
  typedef _Block::const_iterator              _BlockConstIterator;

  typedef map<unsigned, _Block>               _BlockList;
  typedef _BlockList::iterator                _BlockListIterator;
  typedef _BlockList::value_type              _BlockListValueType;

  typedef map<unsigned, unsigned>             _InBlock;
  typedef _InBlock::iterator                  _InBlockIterator;
```

```
  typedef _InBlock::value_type                  _InBlockValueType;

  class _WaitKey {
  public:
    _WaitKey(unsigned blk, unsigned sym)
      : _block(blk), _symbol(sym) { }

    unsigned block()  const { return _block;  }
    unsigned symbol() const { return _symbol; }

    bool operator<(const _WaitKey& key)  const {
      if (_block > key._block) return false;
      if (_block < key._block) return true;
      return _symbol <  key._symbol;
    }
    bool operator==(const _WaitKey& key) const { return (_block == key._block && _symbol ==
key._symbol); }

  private:
    unsigned                                    _block;
    unsigned                                    _symbol;
  };

  class _Waiting {
    typedef list<_WaitKey>                      _WaitList;
    typedef _WaitList::iterator                 _WaitListIterator;

    typedef set<_WaitKey>                       _WaitSet;
    typedef _WaitSet::iterator                  _WaitSetIterator;
  public:
    _Waiting() { }
    ~_Waiting() { }

    _WaitKey pop() {
      _WaitListIterator itr = _waitlist.begin();
      if (itr == _waitlist.end())
        throw jkey_error("'_WaitList' is empty; cannot pop.");

      _WaitKey key(*itr);
      _waitlist.pop_front();

      _WaitSetIterator wsitr = _waitset.find(key);
      if (wsitr == _waitset.end())
        throw jconsistency_error("Could not find block %d symbol %d",
                                 key.block(), key.symbol());
      _waitset.erase(wsitr);

      return key;
    }

    void push(const _WaitKey& key) {
      if (_waitset.find(key) != _waitset.end()) return;

      _waitlist.push_back(key); _waitset.insert(key);
    }

    bool more() const { return _waitlist.size() != 0; }

    bool contains(const _WaitKey& key) const { return _waitset.find(key) != _waitset.end();
}

  private:
    _WaitList                                   _waitlist;
    _WaitSet                                    _waitset;
  };

 public:
  MinimizeFSA(const WFSAcceptorPtr& fsa);

  WFSAcceptorPtr minimize();

 private:
  _Block _inverse(const _WaitKey& key);
  _Block _jlist(const _Block& inverse);

  void _updateWaiting(unsigned blockX, const _Block& block);
  void _updateWaiting(unsigned j, const _Block& blockj, unsigned q, const _Block& blockq);

  void _initialize();
  void _partition();
  void _printBlocks();
  WFSAcceptorPtr _connect();

  WFSAcceptorPtr                                _fsa;
  WFSAcceptorPtr                                _reverse;
  _Waiting                                      _waiting;
  _InBlock                                      _inblock;
  _BlockList                                    _blocklist;
};

WFSAcceptorPtr minimizeFSA(const WFSAcceptorPtr& fsa);


// ----- definition for class 'EncodeWFST' -----
//
class EncodeWFST {
 public:
  EncodeWFST(WFSTransducerPtr& wfst)
    : _wfst(wfst) { }

  WFSAcceptorPtr encode();

 private:
  unsigned _maxNodeIndex();
  void _encode(WFSAcceptorPtr& fsa, WFSTransducer::NodePtr& node, unsigned& totalNodes);

  char _buffer[1000];
  WFSTransducerPtr                              _wfst;
};

WFSAcceptorPtr encodeWFST(WFSTransducerPtr& wfst);


// ----- definition for class 'DecodeFSA' -----
//
class DecodeFSA {
  class _SymbolKey {
  public:
    _SymbolKey(unsigned i, unsigned o, Weight c)
      : _input(i), _output(o), _cost(c) { }

    unsigned input()  const { return _input;  }
    unsigned output() const { return _output; }
    Weight cost() const { return _cost; }

  private:
```

```
   unsigned                                  _input;
   unsigned                                  _output;
   Weight                                    _cost;
  };

  typedef map<unsigned, _SymbolKey>          _SymbolMap;
  typedef _SymbolMap::iterator               _SymbolMapIterator;
  typedef _SymbolMap::value_type             _SymbolMapValueType;

 public:
  DecodeFSA(LexiconPtr& inputLexicon, LexiconPtr& outputLexicon, WFSAcceptorPtr& fsa)
    : _inputLexicon(inputLexicon), _outputLexicon(outputLexicon), _fsa(fsa) { }

  WFSTSortedInputPtr decode();

 private:
  void _initialize();
  void _decode(WFSTransducerPtr& wfst, WFSAcceptor::NodePtr& node);
  void _crackSymbol(const String& symbol, String& input, String& output, float& weight);

  _SymbolMap                                 _symbolMap;

  LexiconPtr                                 _inputLexicon;
  LexiconPtr                                 _outputLexicon;
  WFSAcceptorPtr                             _fsa;
};

WFSTSortedInputPtr decodeFSA(LexiconPtr& inputLexicon, LexiconPtr& outputLexicon, WFSAccepto
rPtr& fsa);


// ----- definition for class 'PurgeWFST' -----
//
class PurgeWFST {

  typedef WFSTransducer::_NodeVector         _NodeVector;
  typedef _NodeVector::iterator              _NodeVectorIterator;
  typedef _NodeVector::const_iterator        _NodeVectorConstIterator;

  typedef WFSTransducer::_NodeMap            _NodeMap;
  typedef _NodeMap::iterator                 _NodeMapIterator;
  typedef _NodeMap::const_iterator           _NodeMapConstIterator;

  typedef set<unsigned>                      _NodeSet;
  typedef _NodeSet::iterator                 _NodeSetIterator;
  typedef _NodeSet::const_iterator           _NodeSetConstIterator;

  typedef WFSTransducer::Node                _Node;
  typedef WFSTransducer::NodePtr             _NodePtr;
  typedef WFSTransducer::Edge                _Edge;
  typedef WFSTransducer::EdgePtr             _EdgePtr;

public:
  PurgeWFST() { }
  ~PurgeWFST()  { }

  WFSTSortedInputPtr purge(const WFSTransducerPtr& wfst);

private:
  void _whichNodes(const WFSTransducerPtr& backward);
  void _connect(const _NodePtr& node, WFSTSortedInputPtr& forward);

  _NodeSet                                   _keep;
};

WFSTSortedInputPtr purgeWFST(const WFSTransducerPtr& wfst);

#endif
```