ივ. ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტის

გამოყენებითი მათემატიკისა და კომპიუტერულ მეცნიერებათა ფაკულტეტის

კომპიუტერების მათემატიკური უზრუნველყოფისა და ინფორმაციული

ტექნოლოგიების კათედრა

დავით მიშელაშვილი

დანართი დისერტაციისთვის

ინსტრუმენტული საშუალებები ბუნებრივი ენის ტექსტების სინტაქსური და მორფოლგიური ანალიზატორის შესადგენად

2005

constraint.h

```
#ifndef _CONSTRAINT_H_
#define _CONSTRAINT_H_

#include "feature.h"
#include "parser.h"

class BoolExpEnv {
public:
  virtual ~BoolExpEnv() { }
  virtual FeatureValuePtr getvar(const FeatureVarName& name) = 0;
  virtual FeatureValue getconst(FeatureConstId cid) = 0;
  virtual FeatureValuePtr getparam(SymNum num) = 0;
};

class BoolExp {
public:
  virtual ~BoolExp() { }
  virtual bool eval() = 0;
  virtual void free() = 0;
  virtual void setenv(BoolExpEnv& env) = 0;
};

typedef BoolExp* BoolExpPtr;

class BoolConst: public BoolExp {
public:
  BoolConst(bool value): mValue(value) { }
  virtual bool eval() { return mValue; }
  virtual void free() { delete this; }
  virtual void setenv(BoolExpEnv& env) { }
private:
  bool mValue;
};

class BoolUnaryOp: public BoolExp {
public:
  BoolUnaryOp(BoolExpPtr arg): mArg(arg) { }
  virtual void free() { mArg->free(); delete this; }
  virtual void setenv(BoolExpEnv& env) { mArg->setenv(env); }
protected:
  bool arg() { return mArg->eval(); }
private:
  BoolExpPtr mArg;
};

class BoolBinaryOp: public BoolExp {
public:
  BoolBinaryOp(BoolExpPtr arg1, BoolExpPtr arg2): mArg1(arg1), mArg2(arg2) { }
  virtual void free() { mArg1->free(); mArg2->free(); delete this; }
  virtual void setenv(BoolExpEnv& env) { mArg1->setenv(env); mArg2->setenv(env); }
protected:
  bool arg1() { return mArg1->eval(); }
  bool arg2() { return mArg2->eval(); }
private:
  BoolExpPtr mArg1;
  BoolExpPtr mArg2;
};

class BoolNegOp: public BoolUnaryOp {
public:
  BoolNegOp(BoolExpPtr arg): BoolUnaryOp(arg) { }
  virtual bool eval() { return !arg(); }
};

class BoolAndOp: public BoolBinaryOp {
public:
  BoolAndOp(BoolExpPtr arg1, BoolExpPtr arg2): BoolBinaryOp(arg1, arg2) { }
  virtual bool eval() { if (arg1()) return arg2(); return false; }
```

```
};

class BoolOrOp: public BoolBinaryOp {
public:
  BoolOrOp(BoolExpPtr arg1, BoolExpPtr arg2): BoolBinaryOp(arg1, arg2) { }
  virtual bool eval() { if (!arg1()) return arg2(); return true; }
};

typedef string OperName;
enum OperType { otNoArg = 0, otUnary = 1, otBinary = 2, otMany = 3 };
enum OperCode {
  ocUnify    = 0,
  ocUniCheck = 1,
  ocEqual    = 2,
  ocAssign   = 3,
  ocNop      = 4,
  ocMultiEqual    = 5,
  ocMultiUniCheck = 6,
  ocNone     = 256
};

bool cop_nop     ();
bool cop_unify   (FeatureValuePtr arg1, FeatureValuePtr arg2);
bool cop_unicheck(FeatureValuePtr arg1, FeatureValuePtr arg2);
bool cop_equal   (FeatureValuePtr arg1, FeatureValuePtr arg2);
bool cop_assign  (FeatureValuePtr arg1, FeatureValuePtr arg2);
bool cop_meq     (FeatureValuePtrList& args);
bool cop_muc     (FeatureValuePtrList& args);

const struct { OperCode code; const char *name; OperType type;
  union
  {
    void *ptr;
    bool (*noarg) ();
    bool (*unary) (FeatureValuePtr arg);
    bool (*binary)(FeatureValuePtr arg1, FeatureValuePtr arg2);
    bool (*many)  (FeatureValuePtrList& args);
  }; }
oper_list[] =
{
  { ocUnify,    "unify",    otBinary, { (void *) cop_unify } },
  { ocUniCheck, "unicheck", otBinary, { (void *) cop_unicheck } },
  { ocEqual,    "equal",    otBinary, { (void *) cop_equal } },
  { ocAssign,   "assign",   otBinary, { (void *) cop_assign } },
  { ocNop,      "nop",      otNoArg,  { (void *) cop_nop } },
  { ocMultiEqual, "meq",    otMany,   { (void *) cop_meq } },
  { ocMultiUniCheck, "muc", otMany,   { (void *) cop_muc } },
  { ocNone,     "",         otNoArg,  { (void *) NULL } }
};

enum OperandType { optVar, optConst, optSymbol };

class OperArg {
public:
  virtual FeatureValuePtr getarg(BoolExpEnv& env) = 0;
  virtual void free() = 0;
};

typedef OperArg* OperArgPtr;
typedef list<OperArgPtr> OperArgPtrList;

bool find_cop(const OperName& name, OperCode& code);

class ArgCountMismatch: public exception { };
class EnvNotDefined: public exception { };
class VarNotFound
    {
        public:
            FeatureVarName name;
            VarNotFound(const FeatureVarName& n): name(n) { }
    };
class SymbolNotFound: public exception { public: SymNum num; SymbolNotFound(SymNum n): num(n) { }
};

class CnstrOper: public BoolExp {
public:
  CnstrOper(OperCode code, OperArgPtrList& args, bool defvalue = true):
    mCode(code), mArgList(args), mType(oper_list[code].type), mDefValue(defvalue), mEnv(NULL) {
    if (oper_list[code].type != otMany)
      if (oper_list[code].type != (int) args.size())
```

```
        throw ArgCountMismatch();
  }
  virtual ~CnstrOper() { }
  virtual bool eval() {
    if (mEnv == NULL)
      throw EnvNotDefined();
    try
    {
      switch (mType)
        {
        case otNoArg : return oper_list[mCode].noarg();
        case otUnary : return oper_list[mCode].unary((*(mArgList.begin()))->getarg(*mEnv));
        case otBinary: return oper_list[mCode].binary((*(mArgList.begin()))->getarg(*mEnv),
(*(++mArgList.begin()))->getarg(*mEnv));
       case otMany  :
         {
           FeatureValuePtrList args;
           for (OperArgPtrList::iterator i = mArgList.begin(); i != mArgList.end(); i++)
             args.push_back((*i)->getarg(*mEnv));
           return oper_list[mCode].many(args);
         }
        }
    }
    catch (VarNotFound& e)
    {
      cerr << "CnstrOper: Variable \"" << e.name << "\" not found (returning default value)." <<
endl;
    }
    catch (SymbolNotFound& e)
    {
      cerr << "CnstrOper: Symbol \"" << e.num << "\" not found (returning default value)." <<
endl;
    }
    return mDefValue;
  }
  virtual void free() { for (OperArgPtrList::iterator i = mArgList.begin(); i != mArgList.end();
i++) (*i)->free(); delete this; }
  virtual void setenv(BoolExpEnv& env) { mEnv = &env; }
  virtual void defvalue(bool defvalue) { mDefValue = defvalue; }
private:
  OperCode mCode;
  OperArgPtrList mArgList;
  OperType mType;
  bool mDefValue;
  BoolExpEnv *mEnv;
};

class OperVarArg: public OperArg {
public:
  OperVarArg(const FeatureVarName& name, const FeaturePath& path):
    mName(name), mPath(path) { }
  virtual ~OperVarArg() { }
  virtual FeatureValuePtr getarg(BoolExpEnv& env) {
    FeatureValuePtr fv = env.getvar(mName);
    if (fv == NULL)
      throw VarNotFound(mName);
    return &(fv->dig(mPath));
  }
  virtual void free() { delete this; }
private:
  FeatureVarName mName;
  FeaturePath mPath;
};

class OperSymArg: public OperArg {
public:
  OperSymArg(SymNum num, const FeaturePath& path): mNum(num), mPath(path) { }
  virtual ~OperSymArg() { }
  virtual FeatureValuePtr getarg(BoolExpEnv& env) {
    FeatureValuePtr fv = env.getparam(mNum);
    if (fv == NULL)
      throw SymbolNotFound(mNum);
    return &(fv->dig(mPath));
  }
  virtual void free() { delete this; }
private:
  SymNum mNum;
  FeaturePath mPath;
};
```

```
class ConstFeatureArg: public OperArg {
public:
  ConstFeatureArg(const FeatureValue& feat): mFeat(feat) { }
  virtual ~ConstFeatureArg() { }
  virtual FeatureValuePtr getarg(BoolExpEnv& env) { mTemp = mFeat; return &mTemp; }
  virtual void free() { delete this; }
private:
  FeatureValue mFeat;
  FeatureValue mTemp;
};

class VarFeatureArg: public OperArg {
public:
  VarFeatureArg(FeatureValuePtr feat): mFeat(feat) { }
  virtual ~VarFeatureArg() { }
  virtual FeatureValuePtr getarg(BoolExpEnv& env) { return mFeat; }
  virtual void free() { delete this; }
private:
  FeatureValuePtr mFeat;
};


#endif
```

constraint.cpp

```
#include "constraint.h"

bool find_cop(const OperName& name, OperCode& code)
{
  int i = 0;
  while (oper_list[i].code != ocNone)
    {
      if (name == oper_list[i].name)
       {
         code = oper_list[i].code;
         return true;
       }
      i++;
    }
  return false;
}


bool cop_nop()
{
  return true;
}

bool cop_unify(FeatureValuePtr arg1, FeatureValuePtr arg2)
{
  return arg1->unify(*arg2);
}

bool cop_unicheck(FeatureValuePtr arg1, FeatureValuePtr arg2)
{
  return arg1->unify(*arg2, true);
}

bool cop_equal(FeatureValuePtr arg1, FeatureValuePtr arg2)
{
  return *arg1 == *arg2;
}
```

```
bool cop_assign(FeatureValuePtr arg1, FeatureValuePtr arg2)
{
  *arg1 = *arg2;
  return true;
}

bool cop_meq(FeatureValuePtrList& args)
{
  if (args.size() > 0)
    {
      FeatureValuePtr left = *args.begin();
      args.pop_front();
      for (FeatureValuePtrList::iterator i = args.begin(); i != args.end(); i++)
        if (*left == **i)
          return true;
    }
  return false;
}

bool cop_muc(FeatureValuePtrList& args)
{
  if (args.size() > 0)
    {
      FeatureValuePtr left = *args.begin();
      args.pop_front();
      for (FeatureValuePtrList::iterator i = args.begin(); i != args.end(); i++)
        if (left->unify(**i, true))
          return true;
    }
  return false;
}
```

dict_lex.lex

```
%{
#define YYSTYPE DICT_YYSTYPE
#include "dict.h"
#define YY_DECL int dict_yylex(YYSTYPE *lvalp, YYLTYPE *llocp, void *parm)
#define YY_USER_ACTION dict_setloc(llocp);
void dict_setloc(YYLTYPE *llocp);
string escape_string(const string& str);
%}
%option yylineno

%x STR

ID [a-zA-Z0-9\x80-\xff][a-zA-Z0-9_\x80-\xff]*

%%

#.*\n
\(\*([^\*\(\)\*])*\*\)

\' { BEGIN(STR); }

<STR>\' { BEGIN(INITIAL); yytext[yyleng - 1] = '\0'; lvalp->str = yytext; return SID; }

<STR>. { yymore(); }

<STR>\n { BEGIN(INITIAL); LEX_ENV.error(yylineno) << "unterminated string." << endl; }

"None" { return NONE; }

{ID} { lvalp->str = yytext; return SID; }

"==" { return OP_UNIFY; }

"["|"]"|"("|")"|"="|","|":"|"$"|"?" { return yytext[0]; }

<<EOF>> { return 0; }
.|\n
%%
```

```
int dict_yywrap(void)
{
  return 1;
}

void dict_setloc(YYLTYPE *llocp)
{
  llocp->first_line = yylineno;
  llocp->last_line = yylineno;
  llocp->first_column = 1;
  llocp->last_column = 1;
}
```

dict_yacc.y

```
%{
#include "feature.h"
#define YYSTYPE DICT_YYSTYPE
#define YYPARSE_PARAM DICT_YYPARSE_PARAM
#define YYLEX_PARAM DICT_YYLEX_PARAM
#include "dict.h"
%}

%pure_parser

%token SID
%token OP_UNIFY
%token NONE

%%

input: /* empty */
       | input def { }
       | input entry { }
       | input show_var { }
       | input find_word { }
       | input unify { }
;

entry:    word fsconstr { $$.feat = $2.feat; ENV.lex().addWord($1.str, $2.feat); }
       | word error { ENV.error(@1.first_line) << "syntax error in lexical entry definition" <<
endl; yyerrok; }
;

word: SID { $$.str = $1.str; }
;

pairs:
       | pairs name ':' fv { $$.feat = $1.feat; $$.feat.set($2.str, $4.fval); }
;

fv:   NONE { $$.fval.empty(); }
    | value { $$.fval.simple($1.str); }
    | fsconstr { $$.fval.complex($1.feat); }
;

value: SID { $$.str = $1.str; }
;

name: SID { $$.str = $1.str; }
;

fsconstr: '[' pairs ']' { $$.feat = $2.feat; }
       | '[' '(' init ')' pairs ']' { $$.feat = $3.feat; $$.feat.merge($5.feat); }
       | '[' error ']' { ENV.error(@1.first_line) << "syntax error in feature constructor." <<
endl; yyerrok; }
       | '[' '(' error ')' pairs ']' { ENV.error(@2.first_line) << "syntax error in feature
constructor initialization part." << endl; yyerrok; }
       | '[' '(' error ')' error ']' { ENV.error(@2.first_line) << "syntax error in feature
constructor and initialiazation part." << endl; yyerrok; }
;

init:    var { if (ENV.vars().exists($1.str))
                      if (ENV.vars().value($1.str).isComplex())
                       $$.feat = ENV.vars().value($1.str).complex();
                     else
                       ENV.warn(@1.first_line) << "feature variable \"" << $1.str
                                               << "\" is not complex (ignoring)."
                                              << endl;
                   else
                     ENV.warn(@1.first_line) << "feature variable \"" << $1.str
                                << "\" not defined."
                               << endl;
                 }
            | init ',' var {  $$.feat = $1.feat;
                                   if (ENV.vars().exists($3.str))
                                   if (ENV.vars().value($3.str).isComplex())
                                      $$.feat.merge(ENV.vars().value($3.str).complex());
                                   else
                                      ENV.warn(@3.first_line) << "feature variable \"" << $3.str
                                                             << "\" is not complex (ignoring)."
                                                            << endl;
                                  else
                                    ENV.warn(@3.first_line) << "feature variable \"" << $3.str
                                              << "\" not defined."
                                             << endl;
                              }
;

var: SID { $$.str = $1.str; }
;

def:      var '=' fv { ENV.vars().set($1.str, $3.fval); $$.fval = $3.fval; }
        | var '=' def { ENV.vars().set($1.str, $3.fval); $$.fval = $3.fval; }
       | var '=' error { ENV.error(@1.first_line) << "syntax error in variable definition." <<
endl; yyerrok; }
;

show_var: '$' var { if (ENV.vars().exists($2.str))
                      ENV.out() << ENV.vars().value($2.str) << endl;
                   else
                     ENV.out() << "feature variable \""
                               << $2.str << "\" not defined."
                              << endl;
                 }
;

find_word: '?' word {
  FeatureStructList ls = ENV.lex().findall($2.str);
  FeatureStructList::iterator i;

  if (ls.size() == 0)
    {
      ENV.out() << "lexical entry \"" << $2.str << "\" was not found." << endl;
    }
  else
    {
       for (i = ls.begin(); i != ls.end(); i++)
         ENV.out() << *i << endl << endl;
       ENV.out() << "found " << ls.size() << " entry(s)." << endl;
    }
}
;

unify: var OP_UNIFY var {
        if (ENV.vars().exists($1.str))
          {
            if (ENV.vars().exists($3.str))
              {
                 if (ENV.vars().value($1.str).unify(ENV.vars().value($3.str)))
                  ENV.out() << "unification successful." << endl;
                 else
                  ENV.out() << "unification failed." << endl;
              }
```

```
             else
              ENV.out() << "feature variable \""
                        << $3.str << "\" not defined."
                        << endl;
          }
         else
          ENV.out() << "feature variable \""
                    << $1.str << "\" not defined."
                    << endl;
       }

%%
```

ecmd.h

```
#ifndef _ECMD_H_
#define _ECMD_H_

#include <string>
#include <list>
#include "environ.h"

enum CmdCode
{
  ccHelp,
  ccQuit,
  ccLexClear,
  ccLexLoad,
  ccLexFind,
  ccLexInfo,
  ccGrmLoad,
  ccSymList,
  ccRuleList,
  ccParse,
  ccDspFeatOpt,
  ccVarList,
  ccVarClear,
  ccDebugParser,
  ccMaxSol,
  ccLargestBush,
  ccMatchCtrl,
  ccFieldCat,
  ccFieldLex,
  ccNone
};

enum DspFeatOpt { dfoNone, dfoAll, dfoRoot };

class CmdEnv;
typedef list<string> StringList;
typedef int (CmdHandler)(CmdEnv& env, const StringList& args);

struct CmdTable
{
  CmdCode cc;
  const char *name;
  int argc;
  CmdHandler *handler;
  const char *dscr;
  const char *help;
};

class CmdEnv {
public:
  virtual CmdTable* cmdtable()        = 0;
  virtual Lexicon& lex()              = 0;
  virtual FeatureVarTable& vars()     = 0;
  virtual SymbolTable& symtable()     = 0;
  virtual RuleTable& rules()          = 0;
  virtual CnstrTable& cnstr()         = 0;
  virtual FeatureVarTable& vartable() = 0;
  virtual FeatureName lex_field()     = 0;
  virtual FeatureName cat_field()     = 0;
  virtual void cat_field(const FeatureName& name) = 0;
  virtual void lex_field(const FeatureName& name) = 0;
  virtual MatchControl& match_control()     = 0;
  virtual DspFeatOpt dsp_feat_opt()         = 0;
  virtual void dsp_feat_opt(DspFeatOpt dfo) = 0;
  virtual void debug_parser(bool debug)     = 0;
  virtual bool debug_parser()         = 0;
  virtual void max_sol(unsigned int maxsol) = 0;
  virtual unsigned int max_sol()      = 0;
  virtual void largest_bush(bool lb)  = 0;
  virtual bool largest_bush()         = 0;
  virtual void match_ctrl_state(bool state) = 0;
  virtual bool match_ctrl_state()     = 0;
  virtual ostream& out()              = 0;
  virtual ostream& err()              = 0;
};

int ecmd_help    (CmdEnv& env, const StringList& args);
int ecmd_quit    (CmdEnv& env, const StringList& args);
int ecmd_lexclear(CmdEnv& env, const StringList& args);
int ecmd_lexload (CmdEnv& env, const StringList& args);
int ecmd_lexfind (CmdEnv& env, const StringList& args);
int ecmd_lexinfo (CmdEnv& env, const StringList& args);
int ecmd_grmload (CmdEnv& env, const StringList& args);
int ecmd_symlist (CmdEnv& env, const StringList& args);
int ecmd_rulelist(CmdEnv& env, const StringList& args);
int ecmd_parse   (CmdEnv& env, const StringList& args);
int ecmd_dspfeatopt(CmdEnv& env, const StringList& args);
int ecmd_varlist (CmdEnv& env, const StringList& args);
int ecmd_varclear(CmdEnv& env, const StringList& args);
int ecmd_debugparser(CmdEnv& env, const StringList& args);
int ecmd_maxsol  (CmdEnv& env, const StringList& args);
int ecmd_largestbush(CmdEnv& env, const StringList& args);
int ecmd_matchctrl(CmdEnv& env, const StringList& args);
int ecmd_fieldcat(CmdEnv& env, const StringList& args);
int ecmd_fieldlex(CmdEnv& env, const StringList& args);

#endif
```

ecmd.cpp

```
#include <iomanip>
#include <stdio.h>
#include "ecmd.h"
#include "dict.h"
#include "rule.h"

const char *DEF_LEX_FILE_EXT = ".lex";
const char *DEF_GRM_FILE_EXT = ".grm";

int ecmd_help(CmdEnv& env, const StringList& args)
{
  int i = 0;
  CmdTable *ct = env.cmdtable();
  if (args.size() == 0)
    {
      env.out() << "Help:" << endl;
      while (ct[i].cc != ccNone)
        {
          string name = ct[i].name;
          name = "`" + name + "'";
          env.out() << " " << setw(8) << name.c_str() << ":"
                   << " " << ct[i].dscr << endl;
          i++;
        }
    }
  else
    {
      CmdCode cc = ccNone;
      for (i = 0; ct[i].cc != ccNone; i++)
        if (strcmp(ct[i].name, args.begin()->c_str()) == 0)
          {
            cc = ct[i].cc;
            break;
          }
      if (cc == ccNone)
        {
          env.out() << "No help available for `" << args.begin()->c_str()
                   << "'."
                   << endl;
        }
      else
        {
          env.out() << "Command " << "`" << ct[i].name
                   << "': " << ct[i].dscr << "." << endl
                   << "Usage: " << ct[i].help << endl;
        }
    }
  return 0;
}

int ecmd_quit(CmdEnv& env, const StringList& args)
{
  env.out() << "Quit." << endl;
  exit(0);
}

int ecmd_lexclear(CmdEnv& env, const StringList& args)
{
  env.lex().clear();
  env.out() << "Lexicon has been cleared" << endl;
  return 0;
}

int ecmd_lexload(CmdEnv& env, const StringList& args)
{
  FILE *f;
  f = fopen(args.begin()->c_str(), "r");
  if (f == NULL)
    f = fopen((*(args.begin()) + DEF_LEX_FILE_EXT).c_str(), "r");
  if (f == NULL)
    {
      env.err() << "Unable to open file: \"" << *(args.begin()) << "\""
               << "[" << DEF_LEX_FILE_EXT << "]"
               << endl;
      return 1;
    }
  else
    {
      env.lex().clear();
      DictYyEnv yyenv(env.vartable(), env.lex());
      dict_yylineno = 0;
      dict_yyrestart(f);
      dict_yyparse(&yyenv);
      fclose(f);
    }
  return 0;
}

int ecmd_lexfind(CmdEnv& env, const StringList& args)
{
  FeatureStructList fsl = env.lex().findall(*(args.begin()));
  if (fsl.size() == 0)
    {
      env.out() << "Lexical entry not found." << endl;
    }
  else
    {
      for (FeatureStructList::iterator i = fsl.begin(); i != fsl.end(); i++)
        env.out() << *i << endl;
      env.out() << endl << fsl.size() << " entry(ies) was(were) found."
```

```
              << endl;
    }
  return 0;
}

int ecmd_lexinfo (CmdEnv& env, const StringList& args)
{
  env.out() << "Current Lexicon Information:" << endl
           << "Word Count: " << env.lex().size() << endl;

  return 0;
}

int ecmd_grmload (CmdEnv& env, const StringList& args)
{
  FILE *f;
  f = fopen(args.begin()->c_str(), "r");
  if (f == NULL)
    f = fopen((*(args.begin()) + DEF_GRM_FILE_EXT).c_str(), "r");
  if (f == NULL)
    {
      env.err() << "Unable to open file: \"" << *(args.begin()) << "\""
              << "[" << DEF_GRM_FILE_EXT << "]"
              << endl;
      return 1;
    }
  else
    {
      env.rules().clear();
      env.cnstr().clear();
      env.symtable().clear();
      RuleYyEnv yyenv(env.symtable(), env.rules(), env.vartable(), env.lex(), env.cnstr());
      rule_yylineno = 0;
      rule_yyrestart(f);
      rule_yyparse(&yyenv);
      fclose(f);
    }
  return 0;
}

int ecmd_symlist (CmdEnv& env, const StringList& args)
{
  env.out() << env.symtable() << endl
           << env.symtable().size() << " symbol(s) total." << endl;
  return 0;
}

int ecmd_rulelist(CmdEnv& env, const StringList& args)
{
  if (env.rules().size() == 0)
    {
      env.out() << "No rules were found." << endl;
    }
  else
    {
      env.out() << env.rules();
      env.out() << env.rules().size() << " rule(s) total." << endl;
    }
  return 0;
}

unsigned int show_tree_item(CmdEnv& env, TreeItem& item, unsigned int num, bool scielent, bool
marks = false, bool line = true, bool show_lex = true)
{
  IdType name = env.symtable().namebyid(item.sym());
  unsigned int size = 5 + name.size();
  FeatureStruct feat;
  FeatureValue  fv = item.feat();
  if (fv.isComplex())
    feat = fv.complex();
  bool has_lex = show_lex && feat.exists(env.lex_field());
  FeatureType lex;
  if (feat[env.lex_field()].isSimple())
    lex = feat[env.lex_field()].simple();
  if (has_lex) size += 2 + lex.size();
  if (!scielent)
    {
      if (marks)
        {
          env.out() << "|";
```

```
         if (line)
             for (unsigned int i = 1; i < size; i++)
               env.out() << "-";
         else
             for (unsigned int i = 1; i < size; i++)
               env.out() << " ";
       }
      else
        {
          env.out().setf(ios::left);
          env.out() << name << ":" << setw(3) << num;
         if (has_lex)
           env.out() << "(" << lex << ")";
         env.out() << " ";
       }
    }
  return size;
}

unsigned int calc_parse_tree_width(CmdEnv& env, ParseTreePtr ptree)
{
  unsigned int width = 0;
  for (ParseTree::NodeList::iterator i = ptree->nodes().begin(); i != ptree->nodes().end(); i++)
  {
    unsigned int item_width = show_tree_item(env, i->item(), 0, true, false, true, !i-
>has_sub());
    if (i->has_sub())
      {
        unsigned int children_width = calc_parse_tree_width(env, i->sub());
        width += children_width > item_width ? children_width : item_width;
      }
    else
      width += item_width;
  }
  return width;
}

struct TreeOutput
{
  bool is_sep;
  union {
    ParseTreePtr tree;
    unsigned int space;
  };
};
typedef list<TreeOutput> TreeOutputList;

void show_parse_tree(CmdEnv& env, TreeOutputList& ptrees, unsigned int& num)
{
  {
    unsigned int n = num;
    unsigned int total_width = 0;
    for (TreeOutputList::iterator t = ptrees.begin(); t != ptrees.end(); t++)
      {
        if (t->is_sep)
        {
          for (unsigned int k = 0; k + total_width < t->space; k++)
            env.out() << " ";
         total_width = t->space > total_width ? t->space : total_width;
        }
        else
        {
          ParseTreePtr ptree = t->tree;
          for (ParseTree::NodeList::iterator i = ptree->nodes().begin(); i != ptree-
>nodes().end(); i++)
            {
             bool line = true;
             if (i + 1 == ptree->nodes().end())
               line = false;
              unsigned int item_width = show_tree_item(env, i->item(), n, false, true, line, !i-
>has_sub());
             unsigned int req_width = item_width;
             if (i->has_sub())
               {
                 unsigned int tmp_width = calc_parse_tree_width(env, i->sub());
                 if (tmp_width > req_width) req_width = tmp_width;
               }
             for (unsigned int k = item_width; k < req_width; k++)
               if (line)
                 env.out() << "-";
               else
                 env.out() << " ";
             total_width += req_width;
              n++;
            }
          }
        }
      env.out() << endl;
    }

  TreeOutputList to_list;
  unsigned int total_width = 0;
  for (TreeOutputList::iterator t = ptrees.begin(); t != ptrees.end(); t++)
    {
      if (t->is_sep)
      {
        for (unsigned int k = 0; k + total_width < t->space; k++)
          env.out() << " ";
       total_width = t->space > total_width ? t->space : total_width;
       to_list.push_back(*t);
      }
      else
      {
      ParseTreePtr ptree = t->tree;
      for (ParseTree::NodeList::iterator i = ptree->nodes().begin(); i != ptree->nodes().end();
i++)
        {
          unsigned int item_width = show_tree_item(env, i->item(), num, false, false, true, !i-
>has_sub());
         unsigned int req_width = item_width;
         TreeOutput to;
         if (i->has_sub())
           {
             unsigned int tmp_width = calc_parse_tree_width(env, i->sub());
             if (tmp_width > req_width) req_width = tmp_width;
             to.is_sep = false;
             to.tree = i->sub();
             to_list.push_back(to);
           }
         for (unsigned int k = item_width; k < req_width; k++)
           env.out() << " ";
         total_width += req_width;
         to.is_sep = true;
         to.space = total_width;
         to_list.push_back(to);
          num++;
        }

      }
    }
  env.out() << endl;
  TreeOutputList::iterator i;
  for (i = to_list.begin(); i != to_list.end(); i++)
    if (!i->is_sep)
      break;
  if (i != to_list.end())
    show_parse_tree(env, to_list, num);
}

int show_features(CmdEnv& env, ParseTreePtrList trees, unsigned int& num, bool recursive = true)
{
  ParseTreePtrList tree_list;
  for (ParseTreePtrList::iterator t = trees.begin(); t != trees.end(); t++)
    {
      ParseTreePtr tree = *t;
      for (ParseTree::NodeList::iterator i = tree->nodes().begin(); i != tree->nodes().end();
i++)
        {
          env.out() << num << ": " << env.symtable().namebyid(i->item().sym()) << endl
                    << i->item().feat() << endl;
          num++;
          if (i->has_sub())
             tree_list.push_back(i->sub());
       }
    }
  if (recursive && tree_list.size() > 0)
    show_features(env, tree_list, num, recursive);
  return 0;
}
```

```
int show_solution(CmdEnv& env, ParseTreePtr ptree)
{
  TreeOutputList trees;
  TreeOutput to;
  ParseTreePtrList ptpl;
  unsigned int num = 1, tree_num = 1;
  to.is_sep = false;
  to.tree = ptree;
  trees.push_back(to);
  show_parse_tree(env, trees, tree_num);
  env.out() << endl;
  ptpl.push_back(ptree);
  switch (env.dsp_feat_opt())
    {
      case dfoAll:
        show_features(env, ptpl, num);
       env.out() << endl;
        break;
      case dfoRoot:
        show_features(env, ptpl, num, false);
       env.out() << endl;
      break;
      case dfoNone:
      break;
    }
  return 0;
}

int parse_sentence(CmdEnv& env, FeatureStructList& fsl)
{
    Parser parser(env.rules().list(), env.match_control());
    if (env.debug_parser())
      parser.debug(true);
    env.out() << "Parsing: ";
    for (FeatureStructList::iterator i = fsl.begin(); i != fsl.end(); i++)
      {
        FeatureName lex_field, cat_field;
       FeatureStruct fs = *i;
       lex_field = env.lex_field();
       cat_field = env.cat_field();
       if (fs.exists(lex_field))
         {
           if (fs.exists(cat_field))
             {
               FeatureType lex, cat;
              FeatureValue val;
              val = fs.value(lex_field);
              if (val.isSimple())
                lex = val.simple();
              else
                {
                  env.err() << endl << "lex - field is not simple value."
                            << endl;
                  return 1;
                }
              val = fs.value(cat_field);
              if (val.isSimple())
                cat = val.simple();
              else
                {
                  env.err() << endl << "cat - field is not simple value."
                            << endl;
                  return 1;
                }
              env.out() << lex
                        << "(" << cat << ") ";

               Symbol s;
              if (env.symtable().symbyname(cat, s))
                {
                  parser.feed(s, *i);
                }
              else
                {
                  env.err() << endl << "Can not find category \"" << cat
                            << "\" in the current grammar." << endl;
                }
             }
           else
             {
               env.err() << "Lexical entry haven't apropriate (lex) field in it's feature
structure." << endl
                         << "Can not continue parsing of this sentence." << endl;
               return 1;
             }
         }
       else
         {
           env.err() << "Lexical entry haven't apropriate (cat) field in it's feature structure."
<< endl
                     << "Can not continue parsing of this sentence." << endl;
           return 2;
         }
      }
    env.out() << endl;

    ParseTreePtrList sol;
    sol = parser.parse(env.match_ctrl_state(), env.max_sol(), env.largest_bush());
    if (sol.size() > 0)
      {
        env.out() << sol.size() << " solution(s) was(were) found." << endl;
        unsigned int i = 1;
        for (ParseTreePtrList::iterator pt = sol.begin(); pt != sol.end(); pt++, i++)
          {
            env.out() << "Parse Tree " << i << ":" << endl;
            show_solution(env, *pt);
          }
      }
    else
      {
        env.out() << "Parsing failed." << endl;
       if (env.largest_bush() && parser.bestsol() != NULL)
         {
           env.out() << "Largest bush:" << endl;
           show_solution(env, parser.bestsol());
         }
      }
    return 0;
}

int generate_and_parse_word_comb(CmdEnv& env, FeatureStructListList::iterator pos,
FeatureStructListList::iterator end_of_list, FeatureStructList& fsl)
{
  if (pos == end_of_list)
    {
      parse_sentence(env, fsl);
    }
  else
    {
      for (FeatureStructList::const_iterator i = pos->begin(); i != pos->end(); i++)
        {
         FeatureStructListList::iterator new_pos = pos;
         fsl.push_back(*i);
         generate_and_parse_word_comb(env, ++new_pos, end_of_list, fsl);
         fsl.pop_back();
       }
    }
  return 0;
}

int ecmd_parse   (CmdEnv& env, const StringList& args)
{
  if (args.size() == 0)
    {
      env.out() << "Nothing to parse!" << endl;
    }
  else
    {
      FeatureStructListList fsll;
      int k = 1;
      for (StringList::const_iterator i = args.begin(); i != args.end(); i++, k++)
        {
         FeatureStructList fsl = env.lex().findall(*i);
         if (fsl.size() == 0)
           {
             env.err() << "Couldn't find word \"" << *i << "\" in the current lexicon."
                       << endl;
             return 1;
           }
         else
```

```
           {
             if (fsl.size() > 1)
               env.out() << "Note: word \"" << *i << "\"[" << k << "] has " << fsl.size()
                         << " entries in the current lexicon." << endl;
             fsll.push_back(fsl);
           }
       }
      FeatureStructList dummy;
      generate_and_parse_word_comb(env, fsll.begin(), fsll.end(), dummy);
    }
  return 0;
}

int ecmd_dspfeatopt(CmdEnv& env, const StringList& args)
{
  string opt = *(args.begin());
  if (opt == "none")
    env.dsp_feat_opt(dfoNone);
    else
      if (opt == "root")
        env.dsp_feat_opt(dfoRoot);
      else
        if (opt == "all")
          env.dsp_feat_opt(dfoAll);
       else
         env.out() << "unknown option: \"" << opt << "\"." << endl;
  return 0;
}

int ecmd_varlist (CmdEnv& env, const StringList& args)
{
  env.out() << env.vartable() << endl;
  return 0;
}

int ecmd_varclear(CmdEnv& env, const StringList& args)
{
  env.vartable().clear();
  env.out() << "Variable table has been cleared." << endl;
  return 0;
}

int ecmd_debugparser(CmdEnv& env, const StringList& args)
{
  string opt = *(args.begin());
  if (opt == "on")
    {
      env.debug_parser(true);
      env.out() << "parser debugging is turned on." << endl;
    }
  else
    if (opt == "off")
      {
        env.debug_parser(false);
        env.out() << "parser debugging is turned off." << endl;
      }
  return 0;
}

int ecmd_maxsol  (CmdEnv& env, const StringList& args)
{
  unsigned int maxsol = strtoul(args.begin()->c_str(), NULL, 10);
  env.max_sol(maxsol);
  if (maxsol == 0)
    env.out() << "No limit on solution number." << endl;
  else
    env.out() << "Limit of " << maxsol << " has been set on solution number." << endl;
  return 0;
}

int ecmd_largestbush(CmdEnv& env, const StringList& args)
{
  string opt = *(args.begin());
  if (opt == "on")
    {
      env.largest_bush(true);
      env.out() << "Largest bush is turned on." << endl;
    }
  else
    if (opt == "off")
```

```
      {
        env.largest_bush(false);
        env.out() << "Largest bush is turned off." << endl;
      }
  return 0;
}

int ecmd_matchctrl(CmdEnv& env, const StringList& args)
{
  string opt = *(args.begin());
  if (opt == "on")
    {
      env.match_ctrl_state(true);
      env.out() << "Match control is turned on." << endl;
    }
  else
    if (opt == "off")
      {
        env.match_ctrl_state(false);
        env.out() << "Match control is turned off." << endl;
      }
  return 0;
}

int ecmd_fieldcat(CmdEnv& env, const StringList& args)
{
    env.cat_field(*(args.begin()));
    return 0;
}

int ecmd_fieldlex(CmdEnv& env, const StringList& args)
{
    env.lex_field(*(args.begin()));
    return 0;
}
```

eparser.cpp

```
#include <iostream>
#include <string>
#include <list>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <readline/readline.h>
#include <readline/history.h>
#include "rule.h"
#include "ecmd.h"

const char *release = "Enchanced Parser 0.1";
char *prompt = "&> ";

CmdTable cmd_table[] = {
  { ccHelp,      "\\h", -1, ecmd_help,     "Display Help", "\\h [cmd]" },
  { ccQuit,      "\\q",  0, ecmd_quit,     "Quit this program", "\\q" },
  { ccLexClear, "\\lc",  0, ecmd_lexclear, "Clear current lexicon", "\\lc" },
  { ccLexLoad,  "\\ll",  1, ecmd_lexload,  "Load lexicon from file", "\\ll file_name" },
  { ccLexFind,  "\\lf",  1, ecmd_lexfind,  "Find lexical entry", "\\lf searching_word" },
  { ccLexInfo,  "\\li",  0, ecmd_lexinfo,  "Information about lexicon", "\\li" },
  { ccGrmLoad,  "\\gl",  1, ecmd_grmload,  "Load grammar from file", "\\gl file_name" },
  { ccSymList,  "\\sl",  0, ecmd_symlist,  "List symbols from current grammar", "\\sl" },
  { ccRuleList, "\\rl",  0, ecmd_rulelist, "List rules from current grammar", "\\rl" },
  { ccParse,    "parse",-1, ecmd_parse,    "Parse sentence", "parse word1 [word2, ... , wordN]"
},
  { ccDspFeatOpt, "\\df",1, ecmd_dspfeatopt,"Display feature option", "\\dfo none | root | all"
},
  { ccVarList,  "\\vl",  0, ecmd_varlist,  "Show variable list", "\\vl" },
  { ccVarClear, "\\vc",  0, ecmd_varclear, "Clear variable list", "\\vc"},
  { ccDebugParser, "\\dp",1, ecmd_debugparser, "Parser debugging", "\\dp on | off"},
  { ccMaxSol,   "\\ms",  1, ecmd_maxsol,   "Set maximum number of displayed parse trees", "\\ms
num (0 - no limit)" },
  { ccLargestBush, "\\lb",1,ecmd_largestbush, "Display the largest parse bush if parsing failed",
"\\lb on | off" },
  { ccMatchCtrl, "\\mc", 1, ecmd_matchctrl, "Set match control", "\\mc on | off" },
  { ccFieldCat, "\\fc",  1, ecmd_fieldcat, "Set default 'cat' field", "\\fc cat_field_name" },
  { ccFieldLex, "\\fl",  1, ecmd_fieldlex, "Set default 'lex' field", "\\fl lex_field_name" },
  { ccNone, "", 0, NULL, "", "" }
};

class EMatchControl: public MatchControl, public BoolExpEnv {
public:
  EMatchControl(CnstrTable& cnstrtable, FeatureVarTable& vartable):
    mCnstrTable(cnstrtable), mVarTable(vartable) { }
  virtual ~EMatchControl() { }
  virtual FeatureValuePtr getvar(const FeatureVarName& name) { return &mVarTable.value(name); }
  virtual FeatureValue getconst(FeatureConstId cid) { return None; }
  virtual FeatureValuePtr getparam(SymNum num) {
    if (mLhs != NULL && mRhs != NULL)
      {
        if (mLhs->sym().num() == num)
         return &(mLhs->feat());
       for (TreeItemList::iterator i = mRhs->begin(); i != mRhs->end(); i++)
         if (i->sym().num() == num)
           return &(i->feat());
      }
    return NULL;
  }
  virtual bool match(TreeItemPtr lhs, TreeItemList& rhs) {
    BoolExpPtr bexp = mCnstrTable.cnstr(lhs->rid());
    if (bexp != NULL)
      {
        mLhs = lhs;
       mRhs = &rhs;
       bexp->setenv((BoolExpEnv&)*this);
       return bexp->eval();
      }
    return true;
  }
private:
  CnstrTable& mCnstrTable;
  FeatureVarTable& mVarTable;
  TreeItemPtr mLhs;
  TreeItemList *mRhs;
};

class ECmdEnv: public CmdEnv {
public:
  ECmdEnv() { mDspFeatOpt = dfoRoot;
              mMatchControl = new EMatchControl(mCnstrTable, mVarTable);
             mDebugParser = false;
             mMaxSol = 0;
             mLargestBush = false;
             mMatchCtrlState = true;
             mCatField = defCatField;
             mLexField = defLexField; }
  virtual ~ECmdEnv() { delete mMatchControl; }
  virtual CmdTable* cmdtable() { return cmd_table; }
  virtual Lexicon& lex() { return mLex; }
  virtual FeatureVarTable& vars() { return mVarTable; }
  virtual SymbolTable& symtable() { return mSymTable; };
  virtual RuleTable& rules() { return mRuleTable; }
  virtual CnstrTable& cnstr() { return mCnstrTable; }
  virtual FeatureVarTable& vartable() { return mVarTable; }
  virtual FeatureName lex_field() { return mLexField; }
  virtual FeatureName cat_field() { return mCatField; }
  virtual void cat_field(const FeatureName& name) { mCatField = name; };
  virtual void lex_field(const FeatureName& name) { mLexField = name; mLex.lex_field(name); };
  virtual MatchControl& match_control() { return *mMatchControl; }
  virtual DspFeatOpt dsp_feat_opt() { return mDspFeatOpt; }
  virtual void dsp_feat_opt(DspFeatOpt dfo) { mDspFeatOpt = dfo; }
  virtual void debug_parser(bool debug) { mDebugParser = debug; }
  virtual bool debug_parser() { return mDebugParser; }
  virtual void max_sol(unsigned int maxsol) { mMaxSol = maxsol; }
  virtual unsigned int max_sol() { return mMaxSol; }
  virtual void largest_bush(bool lb) { mLargestBush = lb; }
  virtual bool largest_bush() { return mLargestBush; }
  virtual void match_ctrl_state(bool state) { mMatchCtrlState = state; }
  virtual bool match_ctrl_state() { return mMatchCtrlState; }
  virtual ostream& out() { return cout; }
  virtual ostream& err() { return cerr; }
private:
  Lexicon mLex;
  FeatureVarTable mVarTable;
  SymbolTable mSymTable;
  RuleTable mRuleTable;
  CnstrTable mCnstrTable;
  EMatchControl *mMatchControl;
  DspFeatOpt mDspFeatOpt;
  bool mDebugParser;
  unsigned int mMaxSol;
  bool mLargestBush;
  bool mMatchCtrlState;
  FeatureName mCatField;
  FeatureName mLexField;
};

void show_welcome_message()
{
  cout << endl
       << " Welcome to " << release << endl
       << " Copyright (C) 2002 By VIAM (Vekua Institute of Applied Mathematics)" << endl
       << "   Author: David Mishelashvili <david@posta.ge>" << endl
       << "   Supervisor: Jemal Antidze <ja@viam.hepi.edu.ge>" << endl
       << endl
       << "   For help type: \\h" << endl
       << endl;
}

void show_help(CmdCode cc)
{
  cout << "Help: " << endl;
}

int main(int argc, char *argv[])
{
  char *cmd;
  ECmdEnv cmdenv;

  show_welcome_message();
  loop:;
  while ((cmd = readline(prompt)) != NULL)
    {
      if (strlen(cmd) > 0)
       {
         const char *cmd_delim = " \t";
         char *token;
         StringList args;
         add_history(cmd);
         token = strtok(cmd, cmd_delim);
         if (token == NULL)
           {
             cerr << "No command given!" << endl;
           }
         else
           {
             int i = 0;
             while (cmd_table[i].cc != ccNone && strcmp(cmd_table[i].name,
                                                   token) != 0) i++;
             if (cmd_table[i].cc == ccNone)
              {
                cerr << "No such command (Try `\\h' for help)." << endl;
              }
             else
              {
                while ((token = strtok(NULL, cmd_delim)) != NULL)
                  args.push_back(token);
                if (cmd_table[i].argc != -1 &&
                    cmd_table[i].argc != (int) args.size())
                  {
                    cerr << "Command parameter count mismatch!" << endl;
                  }
                else
                  {
                    int rc;
                    if ((rc = cmd_table[i].handler(cmdenv, args)) != 0)
                     {
                       cerr << "Command execution failed (" << rc << ")!"
                            << endl;
```

```
                      }
                  }
               }
           }
       }
#if !LINE_FREE_BUG
      free(cmd);
#endif
    }
    cout << "EOF: Use `\\q' to exit." << endl;
}
```

feature.h

```
#ifndef _FEATURE_H_
#define _FEATURE_H_

#include <iostream>
#include <string>
#include <list>
#include <map>

using namespace std;

typedef string FeatureName;
typedef string FeatureType;
typedef string FeatureVarName;
typedef FeatureVarName* FeatureVarNamePtr;
typedef unsigned int FeatureConstId;
typedef list<FeatureName> FeaturePath;
typedef FeaturePath* FeaturePathPtr;

class FeatureStruct;
class FeatureValue;
typedef FeatureStruct* FeatureStructPtr;
typedef FeatureValue* FeatureValuePtr;

typedef list<FeatureName> FeatureNameList;
typedef map<FeatureName, FeatureValue> FeatureNameMap;

class FeatureIsNotSimple { };

class FeatureValue {
public:
  enum ValueKind { vkNone, vkSimple, vkComplex };
  FeatureValue(): mKind(vkNone) { }
  FeatureValue(const FeatureValue& value);
  FeatureValue(const FeatureType& simpleValue):
    mKind(vkSimple), mSimple(simpleValue) { }
  FeatureValue(const FeatureStruct& featureStruct);
  ~FeatureValue() { empty(); }
  void empty(void);
  ValueKind kind(void) const { return mKind; }
  void kind(ValueKind vk) { mKind = vk; }
  bool isNone(void) const { return mKind == vkNone; }
  bool isSimple(void) const { return mKind == vkSimple; }
  bool isComplex(void) const { return mKind == vkComplex; }
  FeatureType simple(void) const { return mSimple; }
  FeatureStruct& complex(void) const { return *mComplex; }
  void simple(const FeatureType& value) { empty(); kind(vkSimple); mSimple = value; }
  void complex(const FeatureStruct& value);
  void assign(const FeatureValue& source);
  FeatureValue& dig(FeaturePath path);
  ostream& dump(ostream& os, unsigned int pos = 0) const;
  bool unify(FeatureValue& other, bool safe = false);
  FeatureValue& operator=(const FeatureValue& source) { assign(source); return *this; }
  FeatureValue& operator=(const FeatureType& source) { simple(source); return *this; }
  bool operator==(const FeatureValue& other) const;
  bool operator!=(const FeatureValue& other) const { return !(*this == other); }
  //  FeatureType (type) () { if (isSimple()) return simple(); else throw FeatureIsNotSimple(); }
private:
  ValueKind mKind;
  FeatureType mSimple;
  FeatureStructPtr mComplex;
};

const FeatureValue None = FeatureValue();

class FeatureStruct {
public:
  FeatureStruct() { };
  FeatureStruct(const FeatureStruct& featureStruct) { *this = featureStruct; }
  virtual ~FeatureStruct() { };
  virtual FeatureStructPtr dup(void) const;
  virtual void free(void) { delete this; }
  void set(const FeatureName& name, const FeatureValue& value = None) { mMap[name] = value; }
  void set(const FeatureName& name, const FeatureType& simple) { mMap[name] =
FeatureValue(simple); }
  void drop(const FeatureName& name) { mMap.erase(name); }
  bool exists(const FeatureName& name) { return mMap.find(name) != mMap.end(); }
  FeatureValue value(const FeatureName& name) { return mMap[name]; }
  FeatureValue& raw_value(const FeatureName& name) { return mMap[name]; }
  FeatureNameList names(void);
  bool unify(FeatureStruct& other, bool safe = false);
  void merge(FeatureStruct& other, bool overwrite = true);
  FeatureStruct& dig(FeaturePath path);
  ostream& dump(ostream& os, unsigned int pos = 0) const;
  FeatureStruct& operator=(const FeatureStruct& source) { mMap = source.mMap; return *this; }
  FeatureValue& operator[](const FeatureName& name) { return mMap[name]; }
  bool operator==(const FeatureStruct& other) const;
  bool operator!=(const FeatureStruct& other) const { return !(*this == other); }
private:
  FeatureNameMap mMap;
};

typedef list<FeatureStruct> FeatureStructList;
typedef list<FeatureStructList> FeatureStructListList;
typedef list<FeatureStructPtr> FeatureStructPtrList;
typedef list<FeatureValue> FeatureValueList;
typedef list<FeatureValuePtr> FeatureValuePtrList;


ostream& operator<<(ostream& os, const FeatureStruct& featureStruct);
ostream& operator<<(ostream& os, const FeatureValue& featureValue);

#endif
```

feature.cpp

```
#include "feature.h"

FeatureValue::FeatureValue(const FeatureValue& value): mKind(vkNone)
{
  *this = value;
}

FeatureValue::FeatureValue(const FeatureStruct& featureStruct):
  mKind(vkComplex)
{
  mComplex = featureStruct.dup();
}

void
FeatureValue::empty(void)
{
  if (isComplex())
    mComplex->free();
  kind(vkNone);
}

void
FeatureValue::complex(const FeatureStruct& value)
{
  empty();
  kind(vkComplex);
  mComplex = value.dup();
}

void
FeatureValue::assign(const FeatureValue& source)
{
  empty();
  kind(source.kind());
  if (source.isSimple()) mSimple = source.simple();
  if (source.isComplex()) mComplex = source.complex().dup();
}

FeatureValue&
FeatureValue::dig(FeaturePath path)
{
  if (path.size() > 0)
    {
      FeatureName name = *path.begin();
      path.pop_front();
      if (isComplex())
        if (complex().exists(name))
          return complex().raw_value(name).dig(path);
        else
          complex().set(name, None);
      else
        {
          FeatureStruct fs;
          fs.set(name, None);
          complex(fs);
        }
      return complex().raw_value(name).dig(path);
    }
  return *this;
}

ostream&
FeatureValue::dump(ostream& os, unsigned int pos) const
{
  if (isComplex())
    {
      complex().dump(os, pos);
      return os;
    }
  for (unsigned int i= 0; i < pos; i++)
    os << " ";
  if (isNone())
    os << "(None)";
  if (isSimple())
    os << simple();
  return os;
}

bool
FeatureValue::unify(FeatureValue& other, bool safe)
{
  if (isComplex() && other.isComplex())
    return complex().unify(other.complex(), safe);
  else
    if (isSimple() && other.isSimple())
      return simple() == other.simple();
    else
      if (isNone())
        {
          if (!safe)
            *this = other;
          return true;
        }
      if (other.isNone())
        {
          if (!safe)
```

```
           other = *this;
          return true;
       }
  return false;
}


bool
FeatureValue::operator==(const FeatureValue& other) const
{
  if (kind() == other.kind())
    {
      if (isNone()) return true;
      if (isSimple()) return simple() == other.simple();
      if (isComplex()) return complex() == other.complex();
    }
  return false;
}


FeatureNameList
FeatureStruct::names(void)
{
  FeatureNameList l;
  FeatureNameMap::const_iterator i;

  for (i = mMap.begin(); i != mMap.end(); i++)
    l.push_back(i->first);

  return l;
}

FeatureStructPtr
FeatureStruct::dup(void) const
{
  FeatureStructPtr ptr = new FeatureStruct;
  *ptr = *this;
  return ptr;
}

bool
FeatureStruct::unify(FeatureStruct& other, bool safe)
{
  FeatureNameMap::iterator i, j;

  for (i = other.mMap.begin(); i != other.mMap.end(); i++)
    {
      j = mMap.find(i->first);

      if (j == mMap.end())
       {
         if (!safe)
           mMap[i->first] = i->second;
         continue;
       }

      FeatureValue &a = i->second, &b = j->second;
      return a.unify(b, safe);
    }

  return true;
}

void
FeatureStruct::merge(FeatureStruct& other, bool overwrite)
{
  FeatureNameMap::iterator i, j;

  for (i = other.mMap.begin(); i != other.mMap.end(); i++)
    {
      j = mMap.find(i->first);

      if (j == mMap.end())
       {
         mMap[i->first] = i->second;
         continue;
       }

      FeatureValue &a = i->second, &b = j->second;
      if (a.isComplex() && b.isComplex())
        b.complex().merge(a.complex(), overwrite);
      else
        if (overwrite)
          b = a;
    }
}

FeatureStruct&
FeatureStruct::dig(FeaturePath path)
{
  if (path.size() == 0)
    {
      return *this;
    }
  else
    {
      FeatureName name = *path.begin();
      path.pop_front();
      FeatureStruct empty;
      FeatureValue fv;
      if (exists(name))
        {
          fv = value(name);
          if (!fv.isComplex())
            {
              fv.complex(empty);
              set(name, fv);
            }
        }
      else
        {
          fv.complex(empty);
          set(name, fv);
        }
      return raw_value(name).complex().dig(path);
    }
}


ostream&
FeatureStruct::dump(ostream& os, unsigned int pos) const
{
  FeatureNameMap::const_iterator i;

  if (mMap.size() == 0) os << "[ ]";

  for (i = mMap.begin(); i != mMap.end(); i++)
    {
      const FeatureValue& value = i->second;

      if (i != mMap.begin())
        for (unsigned int k = 0; k < pos + 1; k++)
          cout << ' ';
      else os << "[";

      os << (string) i->first << ": ";

      int new_pos = pos + ((string) i->first).length() + 3;

      if (value.isNone()) os << "(None)";
      if (value.isSimple()) os << value.simple();
      if (value.isComplex()) value.complex().dump(os, new_pos);

      if (i != --mMap.end()) os << endl; else os << "]";
    }

  return os;
}

bool
FeatureStruct::operator==(const FeatureStruct& other) const
{
  FeatureNameMap::const_iterator i, j;
  if (mMap.size() == other.mMap.size())
    {
      for (i = mMap.begin(); i != mMap.end(); i++)
        {
          j = other.mMap.find(i->first);
          if (j == other.mMap.end()) return false;
```

```
         if (i->second != j->second) return false;
       }

      return true;
    }

  return false;
}


ostream& operator<<(ostream& os, const FeatureStruct& featureStruct)
{
  featureStruct.dump(os);
  return os;
}

ostream& operator<<(ostream& os, const FeatureValue& featureValue)
{
  featureValue.dump(os);
  return os;
}
```

parser.h (სინტაქსური ანალიზატორისთვის)

```
#ifndef _PARSER_H_
#define _PARSER_H_

#include <iostream>
#include <iomanip>
#include <vector>
#include <list>
#include <set>

#include "feature.h"

using namespace std;

// Symbol - Terminal or nonterminal compund of the rule,
//          0 is used for identifying start symbol. Macro
//          IS_START(s) is used to check whether or not the
//          the symbol is srtarting.

typedef int SymNum;

class Symbol {
public:
  Symbol(int sym = 0, SymNum num = 0): mSym(sym), mNum(num) { }
  Symbol(const Symbol& s) { mSym = s.mSym; mNum = s.mNum; }
  int sym() const { return mSym; }
  SymNum num() const { return mNum; }
  bool operator==(Symbol other) const { return mSym == other.mSym; }
  bool operator==(int sym) const { return mSym == sym; }
  bool operator!=(Symbol other) const { return mSym != other.mSym; }
  bool operator!=(int sym) const { return mSym != sym; }
  int& (type) () { return mSym; }
private:
  int mSym;
  SymNum mNum;
};

ostream& operator<<(ostream& os, Symbol sym);

//typedef unsigned int Symbol;
typedef vector<Symbol> SymbolList;
typedef list<SymbolList> SymbolListList;
```

```
#define IS_START(s) (s == 0)

SymbolListList concat(SymbolListList& sll1, SymbolListList& sll2);
ostream& dump_symbol_list(ostream& os, const SymbolList& sl);
ostream& dump_symbol_list_list(ostream& os, const SymbolListList& sll);

typedef unsigned int RuleId;
typedef list<RuleId> RuleIdList;

class Rule {
public:
  Rule() { }
  Rule(RuleId id, Symbol left, const SymbolList& right): mRid(id), mLhs(left), mRhs(right) { }
  RuleId rid(void) const { return mRid; }
  Symbol lhs(void) const { return mLhs; }
  const SymbolList& rhs(void) const { return mRhs; }
  bool equal(const Rule& r) const {
    SymbolList::const_iterator i, j;
    if (lhs() != r.lhs() || rhs().size() != r.rhs().size())
      return false;
    for (i = rhs().begin(), j = r.rhs().begin();
        i != rhs().end() && j != r.rhs().end(); i++, j++)
      if (*i != *j) return false;
    return true;
  }
  bool operator==(const Rule& r) const { return equal(r); }
  bool operator!=(const Rule& r) const { return !equal(r); }
  ostream& dump(ostream& os) const {
    SymbolList::const_iterator i;
    os << "Rule (" << rid() << ") " << lhs() << " ->";
    for (i = rhs().begin(); i != rhs().end(); i++)
      os << " " << *i;
    return os;
  }
private:
  RuleId mRid;
  Symbol mLhs;
  SymbolList mRhs;
};

ostream& operator<<(ostream& os, const Rule& r);

typedef list<Rule> RuleList;

class TreeItem;
class ParseTree;
typedef TreeItem* TreeItemPtr;
typedef ParseTree* ParseTreePtr;
typedef list<TreeItem> TreeItemList;
typedef list<TreeItemPtr> TreeItemPtrList;

class TreeItem {
private:
  RuleId mRid;
  Symbol mSym;
  FeatureValue mFeat;
public:
  TreeItem() { }
  TreeItem(Symbol s): mRid(0), mSym(s) { }
  TreeItem(Symbol s, RuleId r): mRid(r), mSym(s) { }
  TreeItem(Symbol s, RuleId r, const FeatureValue& f):
    mRid(r), mSym(s), mFeat(f) { }
  RuleId rid(void) const { return mRid; }
  Symbol sym(void) const { return mSym; }
  FeatureValue& feat(void) { return mFeat; }
};

class ParseTree {
public:
  class Node {
  public:
    Node(const TreeItem& it): mHasSub(false), mItem(it) { }
    Node(const TreeItem& it, ParseTreePtr pt):
      mHasSub(true), mItem(it), mSub(pt) { }
    bool has_sub() { return mHasSub; }
    TreeItem& item() { return mItem; }
    ParseTreePtr sub() { return mSub; }
  private:
    bool mHasSub;
    TreeItem mItem;
    ParseTreePtr mSub;
  };
  typedef vector<Node> NodeList;


  ParseTree(unsigned int level = 0): mLevel(0) {  }
  ~ParseTree() { for (NodeList::iterator i = nodes().begin(); i != nodes().end(); i++) if (i-
>has_sub()) delete i->sub(); }
  ParseTreePtr copy();
  unsigned int level() { return mLevel; }
  unsigned int size() { return nodes().size(); }
  NodeList& nodes() { return mNodeList; }

  void push_back(const TreeItem& it) { mNodeList.push_back(Node(it)); }
  void push_back(const TreeItem& it, ParseTreePtr tp) { mNodeList.push_back(Node(it, tp)); }
  void push_back(const Node& node) { mNodeList.push_back(node); }

  void reduce(NodeList::iterator first, NodeList::iterator last, const TreeItem& it);
  void reduce(unsigned int pos, unsigned int count, const TreeItem& it);
  void ParseTree::expand(NodeList::iterator pos);
  void expand(unsigned int pos);
  ostream& dump(ostream& os);
private:
  unsigned int mLevel;
  NodeList mNodeList;
};

ostream& operator<<(ostream& os, ParseTree& t);
ostream& operator<<(ostream& os, ParseTreePtr pt);

typedef list<ParseTree> ParseTreeList;
typedef list<ParseTreePtr> ParseTreePtrList;

class MatchControl {
public:
  virtual bool match(TreeItemPtr lhs, TreeItemList& rhs) { return true; }
};

class Parser {
public:
  Parser(const RuleList& rules, MatchControl& mc):
    mRuleList(rules), mMatchCtrl(mc) { mParseTree = new ParseTree(); mDebug = false; mBestSol =
NULL; mMaxSol = 0; }
  ~Parser() { delete mParseTree; }
  void debug(bool enable = true) { mDebug = enable; }
  ostream& dbgout() { return cerr; }
  RuleList& rules() { return mRuleList; }
  ParseTreePtr bestsol() { return mBestSol; }
  void refresh() { delete mParseTree; mParseTree = new ParseTree(); }
  void feed(Symbol sym, const FeatureValue& feat);
  int find_match(const Rule& r, int pos);
  RuleList::iterator find_rule(unsigned int sp, RuleList::iterator pos);
  unsigned int simple_parse(ParseTreePtrList& sol);
  unsigned int bottom_up_parse(unsigned int sp, ParseTreePtrList& sol);
  ParseTreePtrList parse(bool match_control = true, unsigned int max_solutions = 0, bool
find_largest = false);
 private:
  RuleList mRuleList;
  MatchControl& mMatchCtrl;
  ParseTree *mParseTree;
  ParseTree *mBestSol;
  unsigned int mMaxSol;
  bool mMatchCtrlOn;
  bool mDebug;
};



#endif
```

parser.cpp (სინტაქსური ანალიზატორისთვის)

```
#include "parser.h"

ostream& operator<<(ostream& os, Symbol sym)
{
  os << sym.sym();
  if (sym.num() != 0) os << "<" << sym.num() << ">";
  return os;
}

ostream& dump_symbol_list(ostream& os, const SymbolList& sl)
{
  for (SymbolList::const_iterator i = sl.begin(); i != sl.end(); i++)
    os << *i << " ";
  return os;
}

ostream& dump_symbol_list_list(ostream& os, const SymbolListList& sll)
{
  for (SymbolListList::const_iterator i = sll.begin(); i != sll.end(); i++)
    dump_symbol_list(os, *i) << endl;
  return os;
}

SymbolListList concat(SymbolListList& sll1, SymbolListList& sll2)
{
  SymbolListList res;
  for (SymbolListList::iterator i = sll1.begin();
       i != sll1.end(); i++)
    for (SymbolListList::iterator j = sll2.begin();
         j != sll2.end(); j++)
      {
       SymbolList sl = *i;
       for (SymbolList::iterator k = j->begin(); k != j->end(); k++)
         sl.push_back(*k);
       res.push_back(sl);
      }
  return res;
}

ParseTreePtr ParseTree::copy()
{
  ParseTreePtr pt = new ParseTree();
  for (NodeList::iterator i = nodes().begin(); i != nodes().end(); i++)
    {
      if (i->has_sub())
       pt->push_back(Node(i->item(), i->sub()->copy()));
      else
       pt->push_back(Node(i->item()));
    }
  return pt;
}


void ParseTree::reduce(unsigned int pos, unsigned int count,
                      const TreeItem& it)
{
  if (count > 0 && pos + count <= mNodeList.size())
    {
      reduce(nodes().begin() + pos, nodes().begin() + pos + count, it);
    }
}

void ParseTree::reduce(NodeList::iterator first, NodeList::iterator last,
                      const TreeItem& it)
{
  NodeList::iterator cur;
  ParseTreePtr pt = new ParseTree();
  pt->nodes().insert(pt->nodes().end(), first, last);

  cur = nodes().erase(first, last);
  nodes().insert(cur, Node(it, pt));
}


void ParseTree::expand(NodeList::iterator pos)
{
  if (pos->has_sub())
    {
      ParseTreePtr pt = pos->sub();
      NodeList::iterator cur = mNodeList.erase(pos);
      mNodeList.insert(cur, pt->mNodeList.begin(), pt->mNodeList.end());
      pt->mNodeList.clear();
      delete pt;
    }
}

void ParseTree::expand(unsigned int pos)
{
  expand(mNodeList.begin() + pos);
}

ostream& ParseTree::dump(ostream& os)
{
  os << "(";
  for (NodeList::iterator i = nodes().begin(); i != nodes().end(); i++)
    {
      os << setw(3) << i->item().sym();
      if (i->has_sub()) i->sub()->dump(os);
    }
  os << ")";
  return os;
};

ostream& operator<<(ostream& os, const Rule& r)
{
  return r.dump(os);
}

ostream& operator<<(ostream& os, ParseTree& t)
{
  return t.dump(os);
}

ostream& operator<<(ostream& os, ParseTreePtr pt)
{
  return pt->dump(os);
}

void Parser::feed(Symbol sym, const FeatureValue& feat)
{
  mParseTree->push_back(TreeItem(sym, 0, feat));
}

RuleList::iterator Parser::find_rule(unsigned int sp, RuleList::iterator pos)
{
  RuleList::iterator r;
  if (pos == mRuleList.end())
    pos = mRuleList.begin();
  else
    pos++;
  for (r = pos; r != mRuleList.end(); r++)
    {
      ParseTree::NodeList::reverse_iterator i =
       mParseTree->nodes().rend() - sp - 1;
      SymbolList::const_reverse_iterator j = r->rhs().rbegin();

      while (i != mParseTree->nodes().rend() && j != r->rhs().rend())
       {
         if (i->item().sym() != *j)
           break;
         i++; j++;
       }
      if (j == r->rhs().rend())
        return r;
    }
  return mRuleList.end();
}

int Parser::find_match(const Rule& r, int pos)
{
  ParseTree::NodeList::iterator n;
  for (n = mParseTree->nodes().begin() + (pos + 1);
       n != mParseTree->nodes().end(); n++)
    {
      ParseTree::NodeList::iterator   i = n;
      SymbolList::const_iterator j = r.rhs().begin();
      pos++;
      while (i != mParseTree->nodes().end() && j != r.rhs().end())
        {
          if (i->item().sym() != *j)
            break;
          i++; j++;
        }
      if (j == r.rhs().end())
        return pos;
    }
  return -1;
}

unsigned int
Parser::simple_parse(ParseTreePtrList& sol)
{
  RuleList::iterator r;
  if (mParseTree->size() == 1)
    {
      if (IS_START(mParseTree->nodes()[0].item().sym()))
        {
          sol.push_back(mParseTree->copy());
          return 1;
        }
      else
        {
          return 0;
        }
    }
  unsigned int count = 0;
  for (r = mRuleList.begin(); r != mRuleList.end(); r++)
    {
      int pos = -1;
      while ((pos = find_match(*r, pos)) != -1)
        {
          FeatureValue lhsfv;
          mParseTree->reduce(pos, r->rhs().size(), TreeItem(r->lhs(),
                                                        r->rid(), lhsfv));
          count += simple_parse(sol);
          mParseTree->expand(pos);
        }
    }
  return count;
}

unsigned int
Parser::bottom_up_parse(unsigned int sp, ParseTreePtrList& sol)
{
  RuleList::iterator r;
  if (mMaxSol != 0 && sol.size() >= mMaxSol)
    return 0;
  if (mBestSol != NULL)
    if (mParseTree->size() < mBestSol->size())
      {
        delete mBestSol;
        mBestSol = mParseTree->copy();
      }
  if (mParseTree->size() == 1)
    {
      if (IS_START(mParseTree->nodes()[0].item().sym()))
        {
          if (mDebug) dbgout() << "new solution found." << endl;
          sol.push_back(mParseTree->copy());
          return 1;
        }
    }
  unsigned int count = 0;
  r = mRuleList.end();
  while (sp < mParseTree->nodes().size())
    {
      while ((r = find_rule(sp, r)) != mRuleList.end())
       {
         TreeItem lhs = TreeItem(r->lhs(), r->rid());
         TreeItemList rhs;
         unsigned int pos = sp - r->rhs().size() + 1;
         for (unsigned int i = pos; i < pos + r->rhs().size(); i++)
         {
           TreeItem item(r->rhs()[i - pos],
                         mParseTree->nodes()[i].item().rid(),
                         mParseTree->nodes()[i].item().feat());
           rhs.push_back(item);
         }
         if (mMatchCtrlOn)
           if (!mMatchCtrl.match(&lhs, rhs))
             {
              if (mDebug) dbgout() << "rule " << r->rid()
                                   << " should work but"
                                   << " matching failed." << endl;
              continue;
             }

         if (mDebug) dbgout() << "reduce using rule "
                              << r->rid() << "." << endl;
         mParseTree->reduce(pos, r->rhs().size(), lhs);
         count += bottom_up_parse(pos, sol);
         if (mDebug) dbgout() << "return back." << endl;
         mParseTree->expand(pos);
       }
      if (mDebug) dbgout() << "shifting "
                         << mParseTree->nodes()[sp].item().sym()
                         << "." << endl;
      sp++;
    }

  return count;
}

ParseTreePtrList
Parser::parse(bool match_control, unsigned int max_solutions,
              bool find_largest)
{
  ParseTreePtrList ptl;
  if (find_largest)
    {
      if (mBestSol != NULL) delete mBestSol;
      mBestSol = mParseTree->copy();
    }
  mMaxSol = max_solutions;
  mMatchCtrlOn = match_control;
  bottom_up_parse(0, ptl);
  return ptl;
}
```

rule_lex.lex

```
%{
#define LEX_ENV (*((RuleYyEnv* )RULE_YYLEX_PARAM))
#include "rule.h"
#include "rule_yacc.h"
#define YY_DECL int rule_yylex(YYSTYPE *lvalp, YYLTYPE *llocp, void *parm)
#define YY_USER_ACTION rule_setloc(llocp);
void rule_setloc(YYLTYPE *llocp);
string escape_string(const string& str);
%}
```

```
%option yylineno

%x STR

ID [a-zA-Z\x80-\xff][a-zA-Z0-9_\x80-\xff]*
DIG [0-9]

%%

#.*\n
\(\*([^\*\(\)\*])*\*\)

\' { BEGIN(STR); }

<STR>\' { BEGIN(INITIAL); yytext[yyleng - 1] = '\0'; lvalp->name = new string(yytext); return ID;
}

<STR>. { yymore(); }

<STR>\n { BEGIN(INITIAL); LEX_ENV.error(yylineno) << "unterminated string." << endl; }

"None" { return NONE; }

{ID} { lvalp->name = new string(yytext); return ID; }

{DIG}+ { lvalp->uival = strtoul(yytext, NULL, 10); return UNSINT; }

"->" { return LARROW; }
":=" { return ASSIGN; }
"==" { return UNICHECK; }
"<==" { return UNIFY; }


"{"|"}"|"["|"]"|"("|")"|"="|","|"."|";"|":"|"$"|"?"|"|"|"<"|">"|"-"|"&"|"~"|"+"|"_" { return
yytext[0]; }

<<EOF>> { return 0; }
.|\n
%%

int rule_yywrap(void)
{
  return 1;
}

void rule_setloc(YYLTYPE *llocp)
{
  llocp->first_line = yylineno;
  llocp->last_line = yylineno;
  llocp->first_column = 1;
  llocp->last_column = 1;
}
```

morph.h

```
#ifndef MORPH_H
#define MORPH_H

#include <string>
#include <vector>
#include "feature.h"

using namespace std;

typedef string MorphemText;

class MorphemItem
```

```
    {
    public:
        MorphemItem() { }
        MorphemItem(const MorphemText& aText, const FeatureValue& aValue)
            : mText(aText), mValue(aValue) { }
        MorphemText& text() { return mText; }
        FeatureValue& value() { return mValue; }
    private:
        MorphemText mText;
        FeatureValue mValue;
    };

typedef vector<MorphemItem> MorphemItemList;

typedef string MorphemName;
typedef int MorphemId;

class Morphem
    {
    public:
        Morphem() { mId = -1; mNullable = false; }
        Morphem(MorphemId aId, const MorphemName& aName)
            { mNullable = false; mId = aId; mName = aName; }
        int count() const;
        MorphemItem& at(int aIndex);
        void append(const MorphemItem& aItem, const string& aLexField = "");
        void clear();
        bool nullable() const;
        void setNullable(bool aNullable);
        const MorphemName& name() const;
        void setName(const MorphemName& aName);
        MorphemId id() const;
        void setId(MorphemId aId);
    private:
        MorphemName mName;
        MorphemItemList mList;
        bool mNullable;
        MorphemId mId;
    };

ostream& operator<<(ostream& os, MorphemItem& item);
ostream& operator<<(ostream& os, Morphem& morphem);

#endif

// End of File
```

morph.cpp

```
#include "morph.h"

int Morphem::count() const
    {
    return mList.size();
    }

MorphemItem& Morphem::at(int aIndex)
    {
    return mList[aIndex];
    }

void Morphem::append(const MorphemItem& aItem, const string& aLexField)
    {
    if (aLexField == "")
        mList.push_back(aItem);
    else
        {
        MorphemItem item = aItem;
        FeatureValue &fv = item.value();
        if (fv.isComplex())
            fv.complex()[aLexField] = item.text();
        mList.push_back(item);
        }
    }

void Morphem::clear()
    {
    mList.clear();
    }

bool Morphem::nullable() const
    {
    return mNullable;
    }

void Morphem::setNullable(bool aNullable)
    {
    mNullable = aNullable;
    }

const MorphemName& Morphem::name() const
    {
    return mName;
    }

void Morphem::setName(const MorphemName& aName)
    {
    mName = aName;
    }

MorphemId Morphem::id() const
    {
    return mId;
    }

void Morphem::setId(MorphemId aId)
    {
    mId = aId;
    }

ostream& operator<<(ostream& os, MorphemItem& item)
    {
    os << item.text() << " " << item.value();
    return os;
    }

ostream& operator<<(ostream& os, Morphem& morphem)
    {
    bool addComma = false;
    int i;

    os << "@" << morphem.name() << " { ";
    for (i = 0; i < morphem.count(); i++)
        {
        if (addComma) os << ", ";
        os << morphem.at(i);
        addComma = true;
        }
    if (morphem.nullable())
        {
        if (addComma) os << ", ";
        os << "NIL";
        }
    os << " }";

    return os;
    }

// End of File
```

mp.atg

```
COMPILER MP

#include <iostream>
#include "feature.h"
#include "constraint.h"
#include "parser.h"
#include "mp_env.h"

using namespace std;

#define MAX_STR_BUF 1024

string StripQuotes(const string& s)
    {
    if (s.size() > 1)
        return s.substr(1, s.size() - 2);
    else
       return s;
    }

CHARACTERS
  high_char = CHR(192) .. CHR(255) .
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" + high_char .
  digit  = "0123456789" .
  tab    = CHR(9) .
  eol    = CHR(13) .
  lf     = CHR(10) .
  strings_double  = CHR(32) .. CHR(255) - '"' .
  strings_single  = CHR(32) .. CHR(255) - "'" .

COMMENTS FROM "/*" TO "*/"
COMMENTS FROM "#" TO lf
COMMENTS FROM "#" TO eol

IGNORE
  tab + eol + lf

TOKENS
  number = digit { digit } .
  identifier = ( "_" | letter) { "_" | letter | digit } .
  string     = '"' { strings_double } '"' | "'" { strings_single } "'" .

PRODUCTIONS
  MP = { RuleOrConstDef | VariableDef | MorphemDef |
                ShowInfo | IncludeFile } "." .

  /*
  Header = (. string s; .) "ANALYZER" Ident<s> (. EnvAnalyzerName() = s; .) .
  Footer = (. string s; .) "END" Ident<s> "."
    (. if (EnvAnalyzerName() != s)
           { EnvHandleError("Analyzer name mismatch: '" + s + "'"); } .) .
  */

  ShowInfo = "?" ShowInfoArgs [ ";" (. EnvShowInfoEndl(""); .) ] .

  ShowInfoArgs =
    (. string msg; FeatureValue fv; FeatureValuePtr fvp; .)
    ( Str<msg> (. EnvShowInfo(msg); .) |
      ConstRef<fv> (. EnvShowInfo(fv) .) |
      VariableRef<fvp> (. EnvShowInfo(*fvp) .) )
    .

  IncludeFile =
    (. string s; .)
    "<" Str<s> ">"
    (.
    string prepFileName = create_preprocessed_file(s);
    FILE *file = fopen(prepFileName.c_str(), "r");
    mpScanner mps(fileno(file), 1);
    mpError mpe("MPF", &mps);
    mpParser mpp(&mps, &mpe, mEnv);
    mpp.Parse();
    if (!mpp.Successful())
        {
        cerr <<"file: " << s << " ERROR: Parsing failed" << endl;
        mpe.SetOutput(stderr);
        mpe.PrintListing(&mps);
        //mpe.SummarizeErrors();
        }
    fclose(file);
    .) .

  RuleOrConstDef =
    (. string s; .)
    Ident<s> (RuleDef<s> | ConstDef<s>) .

  RuleDef<const string& name> =
    (. Rule rule; rule.lhsName() = name; .)
    "->" RuleRhs<rule> ";"
    (. EnvAddRule(rule); .) .

  RuleRhs<Rule& rule> = RhsItemSeq<rule> .
  RhsItemSeq<Rule& rule> = RhsItem<rule> { RhsItem<rule> } .
  RhsItem<Rule& rule> =
    (. Symbol symbol; BoolExpPtr bep = NULL; .)
    RhsSymbol<rule, symbol, true> (. rule.createSymbol(symbol); .)
    [ RhsSemAction<rule, bep> ]
    (. rule.addNewSymbol(symbol, bep); .) .
  RhsSemAction<Rule& rule, BoolExpPtr& bep> = "{" RhsSemExp<rule, bep> "}" .
  RhsSemExp<Rule& rule, BoolExpPtr& bep> =
    (. BoolExpPtr bep1, bep2; .)
    RhsSemTerm<rule, bep1> (. bep = bep1; .)
    { "|" RhsSemTerm<rule, bep2> (. bep = new BoolOrOp(bep, bep2); .) } .
  RhsSemTerm<Rule& rule, BoolExpPtr& bep> =
    (. BoolExpPtr bep1, bep2; .)
    RhsSemFact<rule, bep1> (. bep = bep1; .)
    { "&" RhsSemFact<rule, bep2> (. bep = new BoolAndOp(bep, bep2); .) } .
  RhsSemFact<Rule& rule, BoolExpPtr& bep> =
    (. bool neg = false; bool hasDefVal = false;
       bool defVal = false; CnstrOperPtr cop; int n; .)
    [ "~" (. neg = true; .) ]
    (
     Num<n> (. bep = new BoolConst(n); .) |
     [ "+" (. hasDefVal = true; defVal = true; .) |
       "-" (. hasDefVal = true; defVal = false; .) ]
    RhsSemCop<rule, cop> (. if (hasDefVal) cop->defvalue(defVal);
                            bep = cop; .) |
    "(" RhsSemExp<rule, bep> ")" )
    (. if (neg) bep = new BoolNegOp(bep); .) .
  RhsSemCop<Rule& rule, CnstrOperPtr& cop> =
    RhsSemOp<rule, cop> | RhsSemFCall<rule, cop> .
  RhsSemOp<Rule& rule, CnstrOperPtr& cop> =
    (. OperArgPtr arg1, arg2; OperArgPtrList args; OperCode c; .)
    RhsSemArg<rule, arg1> (. args.push_back(arg1); .)
    ( ":=" (. c = ocAssign; .) |
      "==" (. c = ocUniCheck; .) |
      "<==" (. c = ocUnify; .) |
      "=" (. c = ocEqual; .) )
    ( RhsSemArg<rule, arg2> (. args.push_back(arg2);
                               cop = new CnstrOper(c, args) .) |
      RhsSemArgs<rule, args>
      (. if (c == ocEqual)
           cop = new CnstrOper(ocMultiEqual, args);
         if (c == ocUniCheck)
           cop = new CnstrOper(ocMultiUniCheck, args);
      .) ) .
  RhsSemArgs<Rule& rule, OperArgPtrList& args> =
    (. OperArgPtr arg; .)
    "(" { RhsSemArg<rule, arg> (. args.push_back(arg); .) } ")" .
  RhsSemArg<Rule& rule, OperArgPtr& arg> =
    (. FeatureValuePtr fvp; FeatureStruct fs; string s; .)
    VariableRef<fvp> (. arg = new VarFeatureArg(fvp); .) |
    Fs<fs> (. arg = new ConstFeatureArg(fs); .) |
    Str<s> (. arg = new ConstFeatureArg(s); .) |
    Path<arg, rule> .
  RhsSemFCall<Rule& rule, CnstrOperPtr& cop> =
    (. string s; OperArgPtrList args; .)
    Ident<s> RhsSemArgs<rule, args>
    (. OperCode code;
       if (find_cop(s, code))
         try {
            cop = new CnstrOper(code, args);
         }
         catch (ArgCountMismatch& e) {
           EnvHandleError("Argument count mismatch");
         }
       else
         EnvHandleError("Unknown function '" + s + "'");
    .) .
  Path<OperArgPtr& arg, Rule& rule> =
    (. string s; FeaturePath path; SymNum symNum; Symbol symbol; .)
    "<"
      ( RhsSymbol<rule, symbol, false> PathCompSeq<path>
        (. if (symbol.morphemId() == 0)
             arg = new OperSymArg(0, path);
           else
             if (rule.findSymbol(symbol, symNum))
               arg = new OperSymArg(symNum, path);
             else
               EnvHandleError("Symbol not found") .) |
        "$" Ident<s> PathCompSeq<path> (. arg = new OperVarArg(s, path); .) ) .
  PathCompSeq<FeaturePath& path> =
    (. string s; .)
    { Ident<s> (. path.push_back(s); .) } ">" .

  RhsSymbol<Rule& rule, Symbol& symbol, bool decl> =
    (. string s; int n = 1; .)
    Ident<s> [ "." Num<n> ]
    (. if (s == rule.lhsName())
         {
            if (decl)
              {
                EnvHandleError("Morphem name should not match the rule name: '"
                               + s + "'");
              }
            else
              {
                if (n > 1) { EnvHandleError("Symbol has an incorrect index: '"
                                            + s + "'"); }
                else
                  {
                  symbol.morphemId() = 0; symbol.ordNum() = 1;
                  }
              }
         }
       else
         if (!EnvMorphemExists(s))
           { EnvHandleError("Couldn't find morphem: '@" + s + "'"); }
         else
           { symbol.morphemId() = EnvMorphem(s).id(); symbol.ordNum() = n; }
    .) .

  MorphemDef =
    (. string s; Morphem &m = EnvNewMorphem(); .)
    "@" Ident<s> "=" "{" MorphItemSeq<m> [ ";" "NIL" (. m.setNullable(true) .) ] "}"
    (. if (EnvMorphemExists(s))
            { EnvHandleError("Morphem already defined: '@" + s + "'"); }
        else
            { m.setName(s); EnvMorphem(s) = m; } .) .

  MorphItemSeq<Morphem& m> =
    (. MorphemItem i1, i2; .)
    MorphItem<i1> (. m.append(i1, EnvMorphemLexField()) .)
    { "," MorphItem<i2> (. m.append(i2, EnvMorphemLexField()) .) } .

  MorphItem<MorphemItem& item> =
    (. string s; FeatureValue fv; .)
    Str<s> FsFeatureValue<fv>
    (. item.text() = s; item.value() = fv; .) .

  VariableDef =
    (. string s; FeatureValue fv; .)
    "$" Ident<s> "=" FsFeatureValue<fv>
    (. EnvVariable(s) = fv;
       // System variables
       if (!strcasecmp(s.c_str(), "lex") && fv.isSimple())
         EnvMorphemLexField() = fv.simple();
    .) .

  ConstRef<FeatureValue& fv> =
    (. string s; .)
    Ident<s>
    (. if (EnvConstantExists(s))
           { fv = EnvConstant(s); }
       else
           { EnvHandleError("Constant not found: '$" + s + "'"); }.) .

  VariableRef<FeatureValuePtr& fv> =
    (. string s; .)
    "$" Ident<s>
    (. if (EnvVariableExists(s))
           { fv = &(EnvVariable(s)); }
       else
           { EnvHandleError("Variable not found: '$" + s + "'"); }.) .

  ConstDef<const string& name> =
    (. FeatureValue fv; .)
    "=" FsFeatureValue<fv>
    (. if (!EnvConstantExists(name))
           { EnvConstant(name) = fv; }
       else
           { EnvHandleError("Constant already defined: '" + name + "'"); } .) .

  Ident<string& s> =
    (. char str[MAX_STR_BUF]; .)
    identifier
    (. LexString(str, MAX_STR_BUF - 1); s = str; .) .

  Num<int& n> =
    (. char str[MAX_STR_BUF]; .)
    number
    (. LexString(str, MAX_STR_BUF - 1); n = atoi(str); .) .

  Str<string& s> =
    (. char str[MAX_STR_BUF]; .)
    string
    (. LexString(str, MAX_STR_BUF - 1); s = StripQuotes(str); .) .

  Fs<FeatureStruct& fs> =
    (. FeatureStruct fs1, fs2; .)
    "[" [ FsInitPart<fs1> ] [ FsPairSeq<fs2> ] "]"
    (. fs.merge(fs1); fs.merge(fs2); .).

  FsInitPart<FeatureStruct& fs> = "(" FsInitSeq<fs> ")" .
  FsInitSeq<FeatureStruct& fs> =
    (. FeatureStruct fs1, fs2; .)
  FsVariable<fs1> (. fs.merge(fs1); .)
  { FsVariable<fs2> (. fs.merge(fs2); .) } .

  FsVariable<FeatureStruct& fs> =
    (. FeatureValue fv; FeatureValuePtr fvp; .)
    ConstRef<fv>
    (.
       if (fv.isComplex())
           { fs = fv.complex(); }
       else
           { EnvHandleError("Constant is not a feature structure"); } .)
    |
    VariableRef<fvp>
    (. if (fvp->isComplex())
           { fs = fvp->complex(); }
       else
           { EnvHandleError("Variable is not a feature structure"); } .) .

  FsPairSeq<FeatureStruct& fs> =
    (. FeatureStruct fs1, fs2; .)
    FsPair<fs1> { FsPair<fs2> }
    (. fs.merge(fs1); fs.merge(fs2); .) .

  FsPair<FeatureStruct& fs> =
    (. FeatureName name; FeatureValue value; .)
    FsFeatureName<name> ":" FsFeatureValue<value>
    (. fs[name] = value; .) .
  FsFeatureName<FeatureName& name> =
    (. char str[MAX_STR_BUF]; .)
    identifier (. LexString(str, MAX_STR_BUF - 1); name = str; .) .

  FsFeatureValue<FeatureValue& value> =
    (. char str[MAX_STR_BUF]; FeatureStruct fs; .)
    "+"        (. LexString(str, MAX_STR_BUF - 1); value.simple(str); .) |
    "-"        (. LexString(str, MAX_STR_BUF - 1); value.simple(str); .) |
    number     (. LexString(str, MAX_STR_BUF - 1); value.simple(str); .) |
    identifier (. LexString(str, MAX_STR_BUF - 1); value.simple(str); .) |
    string     (. LexString(str, MAX_STR_BUF - 1);
                      value.simple(StripQuotes(str)); .) |
    Fs<fs>     (. value.complex(fs); .) .

END MP.
```

parser.h (მორფოლოგიური ანალიზატორისთვის)

```
#ifndef PARSER_H
#define PARSER_H

#include <iostream>
#include <string>
#include <vector>
#include <map>
#include "morph.h"
#include "constraint.h"
#include "feature.h"

using namespace std;

class RhsItem
    {
    public:
        RhsItem() { mBoolExp = NULL; }
        RhsItem(MorphemId aMorphemId, SymNum aSymNum, BoolExpPtr aBoolExp)
            : mMorphemId(aMorphemId), mSymNum(aSymNum), mBoolExp(aBoolExp) { }
        MorphemId& morphemId() { return mMorphemId; }
        SymNum& symNum() { return mSymNum; }
        BoolExpPtr& boolExp() { return mBoolExp; }
    private:
        MorphemId mMorphemId;
        SymNum mSymNum;
        BoolExpPtr mBoolExp;
    };

typedef vector<RhsItem> RhsItemList;

class Symbol
    {
    public:
        MorphemId& morphemId() { return mMorphemId; }
        int& ordNum() { return mOrdNum; }
        bool operator<(const Symbol& aSymbol) const;
        bool operator==(const Symbol& aSymbol) const;
        bool operator!=(const Symbol& aSymbol) const;
    private:
        MorphemId mMorphemId;
        int mOrdNum;
    };

typedef map<Symbol, SymNum> SymbolMap;

class Rule
    {
```

```
    public:
        Rule() { mSymNum = 0; }
        string& lhsName() { return mLhsName; }
        int rhsCount() const { return mRhsItemList.size(); }
        RhsItem& rhsAt(int aIndex) { return mRhsItemList[aIndex]; }
        void createSymbol(Symbol aSymbol);
        void addNewSymbol(Symbol aSymbol, BoolExpPtr aBoolExp);
        void free() { for (int i = 0; i < rhsCount(); i++)
                        rhsAt(i).boolExp()->free(); }
        bool findSymbol(Symbol aSymbol, SymNum& aSymNum) const;
    private:
        string mLhsName;
        int mSymNum;
        RhsItemList mRhsItemList;
        SymbolMap mSymbolMap;
    };

typedef vector<Rule> RuleVector;

class RuleList
    {
    public:
        int count() const { return mRuleVector.size(); }
        Rule& at(int aIndex) { return mRuleVector[aIndex]; }
        void append(Rule& aRule) { mRuleVector.push_back(aRule); }
        void free() { for (int i = 0; i < count(); i++) at(i).free(); }
    private:
        RuleVector mRuleVector;
    };

typedef string VariableName;
typedef map<VariableName, FeatureValue> VariableStoreMap;

class VariableStore
    {
    public:
        FeatureValue& value(const VariableName& name);
        bool exists(const VariableName& name);
        void dump(ostream& os);
    private:
        VariableStoreMap mMap;
    };

typedef map<MorphemId, Morphem> MorphemStoreMap;

class MorphemStore
    {
    public:
        MorphemStore() { mIdGen = 0; }
        Morphem& newMorphem()
            { Morphem& m = mMap[++mIdGen]; m.setId(mIdGen); return m; }
        Morphem& value(MorphemId aId);
        bool exists(MorphemId aId);
        Morphem& value(const MorphemName& name);
        bool exists(const MorphemName& name);
        void dump(ostream& os);
    private:
        MorphemStoreMap mMap;
        MorphemId mIdGen;
    };

class Solution
    {
    public:
        Solution() { }
        Solution(int aRuleIndex, const FeatureValue& aFeatureValue)
            : mRuleIndex(aRuleIndex), mFeatureValue(aFeatureValue) { }
        int ruleIndex() const { return mRuleIndex; }
        FeatureValue& featureValue() { return mFeatureValue; }
    private:
        int mRuleIndex;
        FeatureValue mFeatureValue;
    };

typedef vector<Solution> SolutionList;

typedef map<SymNum, FeatureValue> SymbolValueMap;

class SymbolValueStore
    {
```

```
    public:
        bool exists(SymNum symNum) { return mMap.find(symNum) != mMap.end(); }
        FeatureValue& value(SymNum symNum) { return mMap[symNum]; }
        void erase(SymNum symNum) { mMap.erase(symNum); }
        void clear() { mMap.clear(); }
        SymbolValueStore& operator=(const SymbolValueStore& store);
    private:
        SymbolValueMap mMap;
    };

class MorphAnalyzer: public BoolExpEnv
    {
    public:
        MorphAnalyzer(RuleList& aRuleList, MorphemStore& aMorphemStore,
                      VariableStore& aVariableStore)
            : mRuleList(aRuleList), mMorphemStore(aMorphemStore),
              mVariableStore(aVariableStore) { }
        ~MorphAnalyzer() { }
        void analyze(const string& aWord, int aRuleIndex,
                int aRhsIndex, SolutionList& aSolList);
        void analyze(const string& aWord, int aRuleIndex, SolutionList& aSolList);
        void analyze(const string& aWord, SolutionList& aSolList);
    public: // functions from base classes
        virtual FeatureValuePtr getvar(const FeatureVarName& name)
            { return &(mVariableStore.value(name)); }
        virtual FeatureValuePtr getparam(SymNum num)
            { return &(mSymbolValueStore.value(num)); }
    private:
        RuleList& mRuleList;
        MorphemStore& mMorphemStore;
        VariableStore& mVariableStore;
        SymbolValueStore mSymbolValueStore;
    };

ostream& operator<<(ostream& os, Symbol& symbol);
ostream& operator<<(ostream& os, Rule& rule);
ostream& operator<<(ostream& os, RuleList& rl);

#endif

// End of File
```

parser.cpp (მორფოლოგიური ანალიზატორისთვის)

```
#include "parser.h"

bool Symbol::operator<(const Symbol& aSymbol) const
    {
    return
        mMorphemId < aSymbol.mMorphemId ||
        mMorphemId == aSymbol.mMorphemId &&
        mOrdNum < aSymbol.mOrdNum;
    }

bool Symbol::operator==(const Symbol& aSymbol) const
    {
    return
        mMorphemId == aSymbol.mMorphemId &&
        mOrdNum == aSymbol.mOrdNum;
    }

bool Symbol::operator!=(const Symbol& aSymbol) const
    {
    return !(*this == aSymbol);
    }
```

```
void Rule::createSymbol(Symbol aSymbol)
    {
    SymNum symNum;
    if (!findSymbol(aSymbol, symNum))
        mSymbolMap[aSymbol] = ++mSymNum;
    }

void Rule::addNewSymbol(Symbol aSymbol, BoolExpPtr aBoolExp)
    {
    SymNum symNum;
    if (!findSymbol(aSymbol, symNum))
        {
        mSymbolMap[aSymbol] = ++mSymNum;
        symNum = mSymNum;
        }
    mRhsItemList.push_back(RhsItem(aSymbol.morphemeId(), symNum, aBoolExp));
    }

bool Rule::findSymbol(Symbol aSymbol, SymNum& aSymNum) const
    {
    SymbolMap::const_iterator i = mSymbolMap.find(aSymbol);

    if (i == mSymbolMap.end())
        return false;

    aSymNum = i->second;

    return true;
    }

FeatureValue& VariableStore::value(const VariableName& name)
    {
    return mMap[name];
    }

bool VariableStore::exists(const VariableName& name)
    {
    return mMap.find(name) != mMap.end();
    }

void VariableStore::dump(ostream& os)
    {
    for (VariableStoreMap::iterator i = mMap.begin(); i != mMap.end(); i++)
        os << i->first << " = " << i->second << endl;
    }

Morphem& MorphemStore::value(const MorphemName& name)
    {
    for (MorphemStoreMap::iterator i = mMap.begin(); i != mMap.end(); i++)
        if (i->second.name() == name)
            return i->second;
    Morphem &m = newMorphem();
    m.setName(name);
    return m;
    }

bool MorphemStore::exists(const MorphemName& name)
    {
    for (MorphemStoreMap::iterator i = mMap.begin(); i != mMap.end(); i++)
        if (i->second.name() == name)
            return true;
    return false;
    }

Morphem& MorphemStore::value(MorphemId aId)
    {
    return mMap[aId];
    }

bool MorphemStore::exists(MorphemId aId)
    {
    return mMap.find(aId) != mMap.end();
    }

void MorphemStore::dump(ostream& os)
    {
    for (MorphemStoreMap::iterator i = mMap.begin(); i != mMap.end(); i++)
        os << i->first << " = " << i->second << endl;
    }
```

```
ostream& operator<<(ostream& os, Symbol& symbol)
    {
    os << "Symbol " << symbol.morphemId() << ":" << symbol.ordNum();
    return os;
    }

ostream& operator<<(ostream& os, Rule& rule)
    {
    int i;
    os << "Rule " << rule.lhsName() << " -> ";
    for (i = 0; i < rule.rhsCount(); i++)
        os << rule.rhsAt(i).morphemId() << " ";
    return os;
    }

ostream& operator<<(ostream& os, RuleList& rl)
    {
    int i;
    for (i = 0; i < rl.count(); i++)
        cout << rl.at(i) << endl;
    return os;
    }

SymbolValueStore& SymbolValueStore::operator=(const SymbolValueStore& store)
    {
    mMap = store.mMap;
    return *this;
    }

void MorphAnalyzer::analyze(const string& aWord, int aRuleIndex,
        int aRhsIndex, SolutionList& aSolList)
    {
    int i;
    Rule& rule = mRuleList.at(aRuleIndex);
    if (aRhsIndex >= rule.rhsCount())
        {
        if (aWord == "")
            aSolList.push_back(Solution(aRuleIndex,
                    mSymbolValueStore.value(0)));
        return;
        }
    SymbolValueStore tempStore;
    RhsItem& rhsItem = rule.rhsAt(aRhsIndex);
    MorphemId morphemId = rhsItem.morphemId();
    SymNum symNum = rhsItem.symNum();
    BoolExpPtr bep = rhsItem.boolExp();
    if (!mMorphemStore.exists(morphemId))
        {
        cerr << "MorphAnalyzer::analyze: Morphem ID not found!" << endl;
        return;
        }
    Morphem& morphem = mMorphemStore.value(morphemId);

    for (i = 0; i < morphem.count(); i++)
        {
        string text = morphem.at(i).text();
        int k = aWord.find(text);
        if (k == 0)
            {
            tempStore = mSymbolValueStore;
            mSymbolValueStore.value(symNum) = morphem.at(i).value();

            bool accept = true;

            if (bep)
                {
                bep->setenv(*this);
                accept = bep->eval();
                }

            if (accept)
                analyze(aWord.substr(text.size(),
                            aWord.size() - text.size()),
                        aRuleIndex,
                        aRhsIndex + 1, aSolList);

            mSymbolValueStore = tempStore;
            }
        }
    if (morphem.nullable())
        {
        tempStore = mSymbolValueStore;
        analyze(aWord, aRuleIndex, aRhsIndex + 1, aSolList);
        mSymbolValueStore = tempStore;
        }
    }

void MorphAnalyzer::analyze(const string& aWord, int aRuleIndex,
        SolutionList& aSolList)
    {
    mSymbolValueStore.clear();
    analyze(aWord, aRuleIndex, 0, aSolList);
    }

void MorphAnalyzer::analyze(const string& aWord, SolutionList& aSolList)
    {
    int i;
    for (i = 0; i < mRuleList.count(); i++)
        analyze(aWord, i, aSolList);
    }

// End of File
```