A UNIFICATION-BASED NATURAL LANGUAGE INTERFACE TO A DATABASE

VOLUME II

NEIL KILBY SIMPKINS

Submitted for the degree of Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

August 1988

Contents

Volume II **Page**

Appendix

Appendix B

Grammar and Lexicons for Query Corpus

B.1 Grammar

```
                /* S A M P L E  L F G  G R A M M A R */

        /* ---- s definitions ---- */

                /* s is the distinguished symbol. */
 % s 1
s       --->    np      eqns    (up q) = controller super np sub [+wh] &
                                (up focus) = down &
                                down = controller super s1 sub np ,
                bnd s1  eqns    up = down .

                % interrogative yes/no
 % s 2
s       --->    v       eqns    (up mood) = y_n &
                                (down aux) &
                                up = down ,
                bnd np  eqns    complete (up subj) = down ,
                bnd vp  eqns    complete (up vcomp) = down .

 % s 3
s       --->    v       eqns    (up mood) = y_n &
                                (down aux) c there_be &
                                up = down ,
                bnd np  eqns    complete (up subj) = down ,
                bnd np  eqns    complete (up vcomp) = down ,
                ppadj * eqns    up = down .

        /* ---- np definitions ---- */

 % np 1
np      --->    det     eqns    up = down ,
                n       eqns    up = down ,
                pp *    eqns    (up(down pcase)) = down ,
                ppadj * eqns    up = down .

 % np 2
np      --->    det     eqns    up = down ,
                adj     eqns    (down meas) & (down meas) c + &
                                up = down .

 % np 3
np      --->    n       eqns    (down form) &
                                not (down pred) &
                                up = down ,
                ppadj * eqns    up = down .

 % np 4
np      --->    r       eqns    up = down ,
                agadj*1 eqns    (down aggregate) & (down aggregate) c + &
                                down set_val_of (up aggregates) ,
                adj *1  eqns    (down aggregate) c - &
                                down set_val_of (up adjs) ,
                n       eqns    up = down ,
```

```
                pp *    eqns    (up(down pcase)) = down,
                ppadj * eqns    up = down .

 % np 5
np      --->    np_head eqns    (up num) = (down num) &
                                complete (up head) = down &
                                down = controller super rel_s sub np ,
                rel_s   eqns    complete (up mod) = down .

 % np 6         'African countries'
np      --->    adj     eqns    down set_val_of (up adjs) ,
                n       eqns    up = down .

 % np 7
np      --->    pn      eqns    up = down &
                                not (up r) .

 % np 8         'the Baltic'
np      --->    'the' ,
                pn      eqns    (down r) = + &
                                up = down .

 % np 9
np      --->    det     eqns    (down det) & (down det) c wh &
                                up = down .

 % np 10        'two seas'
np      --->    num     eqns    up = down ,
                n       eqns    up = down .

 % np 11        'three million'
np      --->    num     eqns    up = down ,
                meas    eqns    up = down .

        /* ---- np gap definitions ---- */

 % np gap 1
npe     --->    e       eqns    up = controllee sub np .        % Gap.

        /* ---- s1 definitions ---- */

 % s' 1
s1      --->    bnd vp  eqns    (up subj) = controllee sub np &
                                up = down .

 % s' 2         % wh - where/when.
s1      --->    v       eqns    (up q-adverb) & (up q-adverb) c + &
                                up = down &
                                (down attributive) c + &
                                (down aux) & not (down aux) c have ,
                bnd np  eqns    complete (up subj) = down &
                                (up focus-subj) = down &
                                (up q-subj) = down ,
                vp1     eqns    (up vcomp) = down .

 % s' 3
s1      --->    v       eqns    up = down &
                                (down attributive) c + &
                                (down aux) & not (down aux) c have ,
                bnd np  eqns    complete (up subj) = down &
                                (up focus-subj) = down ,
                vp1     eqns    (up vcomp) = down .
```

```
 % s' 4
s1        --->     v        eqns     up = down &
                                     (down attributive) = - &
                                     (down aux) & not (down aux) c have ,
                   bnd np   eqns     complete (up subj) = down ,
                   vpl      eqns     (up vcomp) = down .

 % s' 5
s1        --->     v        eqns     up = down &
                                     (down attributive) = - &
                                     (down aux) & not (down aux) c have ,
                   bnd ap   eqns     (up subj) = controllee sub np &
                                     (up vcomp) = down .

 % s' 6
s1        --->     v        eqns     up = down &
                                     (down aux) & not (down aux) c have ,
                   bnd vpl  eqns     (up subj) = controllee sub np &
                                     (up vcomp) = down .

 % s' 7
s1        --->     v        eqns     up = down &                        % have.
                                     (down aux) & (down aux) c have &
                                     (up subj) = controllee sub np ,
                   np       eqns     (up obj) = down &
                                     (down of-obj) = (up subj) ,
                   bnd vp   eqns     (up vcomp) = down .

 % s' 8
s1        --->     v        eqns     up = down &      % 'with' prep as verb/have.
                                     (down aux) & (down aux) c have &
                                     (up subj) = controllee sub np &
                                     (up subj) = controller super rel_adj sub pp ,
                   rel_adj  eqns     (up rel_adj) = down ,
                   bnd vp   eqns     (up vcomp) = down .


          /* ---- rel_s definitions ---- */

 % rels 1                   reduced relative.
rel_s     --->     bnd vp   eqns     (down participle) &
                                     (down subj) = controllee sub np &
                                     up = down .

 % rels 2                   pied-piped relative.
rel_s     --->     pp_pied  eqns     (up pied) = down &
                                     down = controller super s1 sub np ,
                   relpn    eqns     up = down &
                                     (down rel) = + &
                                     (down wh) = + ,
                   s1       eqns     up = down .

 % rels 3                   relative with marker.
rel_s     --->     relpn    eqns     up = down ,
                   bnd vp   eqns     (down subj) = controllee sub np &
                                     up = down .

 % rels 4
rel_s     --->     relpn    eqns     up = down ,
                   v        eqns     (up aux) &
                                     (down subj) = controllee sub np &
                                     up = down ,
                   bnd vpl  eqns     (up vcomp) = down .
```

```
 % rels 5               relatives in coordination.
rel_s   --->    conj
                rel_s   eqns    down set_val_of (up conjs) ,
                'and',
                rel_s   eqns    down set_val_of (up conjs) .

 % rels 6
rel_s   --->    relpn   eqns    up = down &
                                (down case) c gen ,
                n       eqns    (down of-obj) = controllee sub np &
                                (up subj) = down ,
                bnd vp  eqns    up = down .

 % rels 7
rel_s   --->    relpn   eqns    up = down ,
                bnd np  eqns    (up subj) = down ,
                vp      eqns    up = down .

 % rels 8
rel_s   --->    bnd np  eqns    (up subj) = down ,
                p       eqns    up = down ,
                npe     eqns    (up obj) = down .

        /* ---- np_head definitions ---- */

 % rel head 1
np_head --->    r       eqns    up = down ,
                n       eqns    up = down .
 % rel head 2
np_head --->    det     eqns    up = down ,
                n       eqns    up = down .
 % rel head 3
np_head --->    np      eqns    (down r) &
                                (up num) = (down num) & up = down ,
                p       eqns    up = down .

        /* ---- rel_adj definitions (relative adjuncts) ---- */

 % rel adjunct 1
rel_adj --->    np_head eqns    (up head) = down &
                                (down of-obj) = controllee sub pp &
                                down = controller super rel_s sub np ,
                rel_s   eqns    (up mod) = down .

        /* ---- vp1 definitions ---- */

 % vp' 1
vp1     --->    npe     eqns    (down det) & (down det) c wh &
                                up = down .

 % vp' 2
vp1     --->    v       eqns    up = down ,
                bnd pp  eqns    (up(down pcase)) = down .

 % vp' 3
vp1     --->    v       eqns    up = down ,
                ppe     eqns    (up(down pcase)) = down .

 % vp' 4
vp1     --->    pn      eqns    up = down .


        /* ---- vp definitions ---- */

 % vp 1
vp      --->    v       eqns    not (down aux) & up = down ,
                np      eqns    (up obj) = down .
```

```
 % vp 2
vp       --->    v       eqns    not (down aux) & up = down ,
                 npe     eqns    (up obj) = down ,
                 bnd pp  eqns    (up(down pcase)) = down .

 % vp 3
vp       --->    v       eqns    (down aux) &
                                 up = down ,
                 neg *   eqns    up = down ,
                 vp      eqns    not (down aux) & (up vcomp) = down .

        /* ---- ppe definitions ---- */

 % pp gap 1
ppe      --->    p       eqns    up = down ,
                 npe     eqns    (up obj) = down .

        /* ---- pp definitions ---- */

 % pp 1
pp       --->    p       eqns    up = down ,
                 np      eqns    (up obj) = down .

 % pp 2
pp       --->    pmod    eqns    (down direction) &
                                 up = down ,
                 pp      eqns    (up(down pcase)) = down .

 % pp 3
pp       --->    neg     eqns    up = down ,
                 pp      eqns    up = down .

 % pp 4
pp       --->    p       eqns    up = down ,
                 n       eqns    complete (up obj) = down .

        /* ---- ppadj definitions ---- */

 % pp adjunct 1
ppadj    --->    pp      eqns    down set_val_of (up adjuncts) .
 % pp adjunct 2          coordination
ppadj    --->    conj
                 pp      eqns    down set_val_of (up adjuncts) ,
                 'and' ,
                 pp      eqns    down set_val_of (up adjuncts) .

        /* ---- pp_pied definitions ---- */

 % pp pied 1
pp_pied --->     np      eqns    (up subj) = down &
                                 (up num) = (down num) ,
                 ppe     eqns    up = down .

        /* ---- ap definitions ---- */

 % ap 1
ap       --->    adj     eqns    up = down.

        /* ---- det definitions ---- */

 % det 1
det      --->    np      eqns    (up poss) = down ,
                 '''s' .

 % det 2
det      --->    deg     eqns    up = down ,
```

```
         'than' ,
         num       eqns     up = down .

/* ************************************************************ */
```

B.2 Domain Lexicon

```
/* ************************************************************ */
/*       FILE    : lex_dom                                      */
/*       PURPOSE : lexical entries for geographical domain.     */
/* ************************************************************ */

         /* ---- Entries for category : adj ---- */

african ~
        adj      eqns     (up pred) = african & (up aggregate) = - &
                          (up quant-domain) = country &
                          (up sem) = african(quant) .
american ~
        adj      eqns     (up pred) = american & (up aggregate) = - &
                          (up quant-domain) = country &
                          (up sem) = american(quant) .
asian ~
        adj      eqns     (up pred) = asian & (up aggregate) = - &
                          (up quant-domain) = country &
                          (up sem) = asian(quant) .
european ~
        adj      eqns     (up quant-domain) = country &
                          (up pred) = european & (up aggregate) = - &
                          (up sem) = european(quant) .

         /* ---- Entries for category : n ---- */

area ~
        n        eqns     (up pred) = area >> [(up of-obj)] &
                          (up sem) = area(of-obj,quant) & (up num) = sg &
                          (up of-obj-domain) = place &
                          (up quant-domain) = area .
areas ~
        n        eqns     (up pred) = area >> [(up of-obj)] &
                          (up sem) = area(of-obj,quant) & (up num) = pl &
                          (up of-obj-domain) = place &
                          (up quant-domain) = area .
capital ~
        n        eqns     (up pred) = capital >> [(up of-obj)] &
                          (up sem) = capital(of-obj,quant) & (up num) = sg &
                          (up of-obj-domain) = country &
                          (up quant-domain) = city
                 or       (up pred) = capital >> [(up poss)] &
                          (up sem) = capital(poss, quant) &
                          not (up poss-pcase) & (up num) = sg &
                          (up poss-domain) = country &
                          (up quant-domain) = city .
capitals ~
        n        eqns     (up pred) = capital >> [(up of-obj)] &
                          (up sem) = capital(of-obj, quant) &
                          (up of-obj-domain) = country & (up num) = pl &
                          (up quant-domain) = city
                 or       (up pred) = capital >> [(up poss)] &
                          (up sem) = capital(poss, quant) & (up num) = sg &
                          not (up poss-pcase) &
                          (up poss-domain) = country &
                          (up quant-domain) = city .
cities ~
        n        eqns     (up num) = pl & (up pred) = city &
```

```
                        (up sem) = city(quant) & (up quant-domain) = city .
continent ~
        n       eqns    (up pred) = continent & (up sem) = continent(quant) &
                        (up num) = sg & (up quant-domain) = continent .
continents ~
        n       eqns    (up pred) = continent & (up sem) = continent(quant) &
                        (up num) = pl & (up quant-domain) = continent .
countries ~
        n       eqns    (up pred) = country & (up sem) = country(quant) &
                        (up num) = pl & (up quant-domain) = country .
country ~
        n       eqns    (up pred) = country & (up sem) = country(quant) &
                        (up num) = sg & (up quant-domain) = country .
ocean ~
        n       eqns    (up pred) = ocean & (up sem) = ocean(quant) &
                        (up num) = sg & (up quant-domain) = ocean .
oceans ~
        n       eqns    (up pred) = ocean & (up sem) = ocean(quant) &
                        (up num) = pl & (up quant-domain) = ocean .
population ~
        n       eqns    (up num) = sg &
                        (up pred) = population >> [(up of-obj)] &
                        (up sem) = population(of-obj,quant) &
                        (up of-obj-domain) = peopled &
                        (up quant-domain) = measure
                or      (up num) = sg &
                        (up pred) = population >> [(up poss)] &
                        (up sem) = population(poss, quant) &
                        (up case) & (up case) c gen &
                        (up quant-domain) = measure &
                        (up poss-domain) = peopled .
sea ~
        n       eqns    (up num) = sg &
                        (up pred) = sea &
                        (up sem) = sea(quant) &
                        (up quant-domain) = sea·.
seas ~
        n       eqns    (up num) = pl &
                        (up pred) = sea &
                        (up sem) = sea(quant) &
                        (up quant-domain) = sea .
river ~
        n       eqns    (up pred) = river &
                        (up sem) = river(quant) &
                        (up quant-domain) = river &
                        (up num) = sg .
rivers ~
        n       eqns    (up pred) = river &
                        (up sem) = river(quant) &
                        (up quant-domain) = river &
                        (up num) = pl .

                /* ---- Entries for category : v ---- */

border ~
        v       eqns    (up nonfinite) = + &
                        (up pred) = border >> [(up subj),(up obj)] &
                        (up sem) = borders(subj, obj) &
                        (up subj-domain) = location &
                        (up obj-domain) = location .
borders ~
        v       eqns    (up pred) = border >> [(up subj),(up obj)] &
                        (up sem) = borders(subj, obj) &
                        (up tense) = present &
                        (up subj-domain) = location &
                        (up obj-domain) = location .
bordered ~
```

```
        v       eqns    (up pred) = border >> [(up subj),(up by-obj)] &
                        (up sem) = borders(subj, by-obj) &
                        (up tense) = past &
                        (up subj-domain) = location &
                        (up by-obj-domain) = location .
bordering ~
        v       eqns    (up pred) = border >> [(up subj),(up obj)] &
                        (up sem) = borders(subj, obj) &
                        (up tense) = present &
                        (up subj-domain) = location &
                        (up participle) = ing &
                        (up obj-domain) = location .
contain ~

        v       eqns    (up pred) = contains >> [(up subj),(up obj)] &
                        (up sem) = contains(subj,obj) &
                        (up nonfinite) = + &
                        (up subj-domain) = location &
                        (up obj-domain) = location .
contains ~
        v       eqns    (up pred) = contains >> [(up subj),(up obj)] &
                        (up sem) = contains(subj,obj) &
                        (up tense) = present &
                        (up subj-domain) = location &
                        (up obj-domain) = location .
exceeds ~
        v       eqns    (up pred) = exceeds >> [(up subj),(up obj)] &
                        (up sem) = exceeds(subj,obj) &
                        (up subj-domain) = measure &
                        (up obj-domain) = measure &
                        (up tense) = present .
exceeding ~
        v       eqns    (up pred) = exceeds >> [(up subj),(up obj)] &
                        (up sem) = exceeds(subj,obj) &
                        (up subj-domain) = measure &
                        (up obj-domain) = measure &
                        (up tense) = present &
                        (up participle) = ing .
flow ~
        v       eqns    (up pred) = flow >> [(up subj),(up thru-obj)] &
                        (up sem) = flows(subj,thru-obj) &
                        (up subj-domain) = river &
                        (up thru-obj-domain) = location &
                        (up nonfinite) = + .
flows ~
        v       eqns    (up pred) = flow >>
                                    [(up subj),(up obj),(up into-obj)] &
                        (up sem) = flows(subj,obj,into-obj) &
                        (up obj-pcase) & (up obj-pcase) c from &
                        (up tense) = present &
                        (up subj-domain) = river &
                        (up obj-domain) = country &
                        (up into-obj-domain) = sea .

                /* ---- Entries for category : pn ---- */

atlantic ~
        pn      eqns    (up num) = sg & (up r) & (up r) c + &
                        (up pred) = atliantic &
                        (up pn_type) = ocean &
                        (up domain) = ocean &
                        (up sem) = atlantic .
australasia ~
        pn      eqns    (up num) = sg &
                        (up pred) = australasia &
                        (up pn_type) = continent &
                        not (up r) &            % not 'the australasia'.
```

```
                              (up domain) = continent &
                              (up sem) = australasia .
afghanistan ~
        pn      eqns          (up num) = sg &
                              (up pred) = afghanistan &
                              (up pn_type) = country &
                              not (up r) &
                              (up domain) = country &
                              (up sem) = afghanistan .
baltic ~
        pn      eqns          (up num) = sg & (up r) & (up r) c + &
                              (up pred) = baltic &
                              (up pn_type) = sea &
                              (up domain) = ocean &
                              (up sem) = baltic .
black_sea ~
        pn      eqns          (up num) = sg &
                              (up r) & (up r) c + &
                              (up pred) = black_sea &
                              (up domain) = sea &
                              (up sem) = black_sea .
china ~
        pn      eqns          (up num) = sg &
                              (up pred) = china &
                              (up pn_type) = country &
                              not (up r) &
                              (up domain) = country &
                              (up sem) = china .
danube ~
        pn      eqns          (up num) = sg & (up r) & (up r) c + &
                              (up pred) = danube &
                              (up pn_type) = river &
                              (up domain) = river &
                              (up sem) = danube .
equator ~
        pn      eqns          (up num) = sg & (up r) & (up r) c + &
                              (up domain) = latitude &
                              (up pred) = equator &
                              (up sem) = equator .
europe ~
        pn      eqns          (up num) = sg &
                              (up pred) = europe &
                              (up pn_type) = continent &
                              not (up r)  &
                              (up domain) = continent &
                              (up sem) = europe .
india ~
        pn      eqns          (up num) = sg &
                              (up pred) = india &
                              (up pn_type) = country &
                              not (up r) &
                              (up domain) = country &
                              (up sem) = india .
london ~
        pn      eqns          (up num) = sg &
                              (up pred) = london &
                              (up pn_type) = city &
                              not (up r) &
                              (up domain) = city &
                              (up sem) = london .
mediterranean ~
        pn      eqns          (up num) = sg &
                              (up pred) = mediterranean &
                              (up pn_type) = sea &
                              (up r) & (up r) c + &
                              (up domain) = sea &
                              (up sem) = mediterranean .
```

```
upper_volta ~
        pn      eqns    (up num) = sg &
                        (up pred) = upper_volta &
                        (up pn_type) = country &
                        not (up r) &
                        (up domain) = country &
                        (up sem) = upper_volta .

                /* ---- Entries for category : pmod ---- */

south ~
        pmod    eqns    (up pred) = south >> [(up subj),(up of-obj)] &
                        (up sem) = southof(subj,of-obj) &
                        (up direction) = + &
                        (up subj-domain) = location &
                        (up of-obj-domain) = location .

        /* ************************************************************   */
```

B.3 General Lexicon

```
        /* ************************************************************* */
        /*      FILE    : gen_lex                                         */
        /*      PURPOSE : domain independent lexical entries.             */
        /* ************************************************************* */

        /* S A M P L E  L F G  L E X I C O N */

                /* ---- Entries for adjectives ---- */

large ~
        adj     eqns    (up pred) = large >> [(up subj)] &
                        (up sem) = area(subj,quant) &
                        (up meas) = + .                 % 'how small'.
small ~
        adj     eqns    (up pred) = small >> [(up subj)] &
                        (up sem) = area(subj,quant) &
                        (up meas) = + .

                /* ---- Entries for aggregate adjectives ---- */

average ~
        agadj   eqns    (up aggregate) = + &            % aggregate operator.
                        (up pred) = average &
                        (up sem) = average(quant) .
largest ~
        agadj   eqns    (up aggregate) = + &
                        (up pred) = largest &
                        (up sem) = largest(quant) .
smallest ~
        agadj   eqns    (up aggregate) = + &
                        (up pred) = smallest &
                        (up sem) = smallest(quant) .
total ~
        agadj   eqns    (up aggregate) = + &
                        (up pred) = total &
                        (up sem) = total(quant) .

                /* ---- Entries for determiners ---- */

any ~
        det     eqns    (up det) = any &
                        (up quantifier) = any .
each ~
        det     eqns    (up num) = sg &
```

```
                              (up det) = each &
                              (up quantifier) = each .
how ~
        det      eqns         (up det) = wh &
                              (up quantifier) = wh &
                              up = controllee sub [+wh] .
'how many' ~
        det      eqns         (up det) = numberof &
                              (up quantifier) = numberof &      % top-level quantifier.
                              up = controllee sub [+wh] &
                              (up num) = pl &
                              (up def) = -  .
no ~
        det      eqns         (up det) = no &
                              (up quantifier) = no .
some ~
        det      eqns         (up det) = some &
                              (up quantifier) = some .
what ~
        det      eqns         (up det) = wh &
                              (up quantifier) = wh &
                              up = controllee sub [+wh] .
where ~                                   % adverb as wh-front (also 'when')
        det      eqns         (up adverb) = + & (up det) = wh &
                              (up quantifier) = wh &
                              (up pcase) = in &
                              (up pred) = in >> [(up subj)] &
                              (up sem) = in(subj, quant) &
                              up = controllee sub [+wh] .
which ~
        relpn    eqns         (up rel) = + &            % relative and determiner.
                              (up wh) = +

   and  det      eqns         (up det) = wh &
                              (up quantifier) = wh &
                              up = controllee sub [+wh] .

                 /* ---- Relative markers ---- */
whose ~
        relpn    eqns         (up rel) = + &
                              (up case) = gen .
that ~
        relpn    eqns         (up rel) = + .

                 /* ---- Entries for nouns ---- */

there ~
        n        eqns         (up form) = there .       % dummy pred.

percentage ~
        n        eqns         (up pred) = percentage >> [(up of-obj)] &
                              (up sem) = percentage(of-obj,quant) &
                              (up proportional) = + .

                 /* ---- Entries for numbers ---- */

one ~
        num      eqns         (up card) = one & (up num) = sg .
two ~
        num      eqns         (up card) = two & (up num) = pl .
1 ~
        num      eqns         (up card) = one & (up num) = sg .
2 ~
        num      eqns         (up card) = two & (up num) = pl .
10 ~
        num      eqns         (up card) = ten & (up num) = pl .
18 ~
```

```
        num     eqns    (up card) = eighteen & (up num) = pl .

                /* ---- Entries for category : v ---- */

are ~                            % existential coupled with 'there'.
        v       eqns    (up pred) = exists >> [(up vcomp)]-[(up subj)] &
                        (up attributive) = - &
                        (up subj-form) &
                        (up subj-form) c there &
                        (up vcomp-num) c pl &
                        (up tense) = present &
                        (up aux) = be &
                        (up num) = pl
                                 % attributive.
                or      (up attributive) = + &
                        (up pred) = attribute >> [(up vcomp)]-[(up subj)] &
                        not (up subj-form) & not (up vcomp-form) &
                        (up subj-num) c pl &
                        (up vcomp-subj) = (up subj) &      % functional control.
                        (up tense) = present &
                        (up aux) = be & (up num) = pl
                                 % equative.
                or      (up pred) = equate >> [(up vcomp)=(up subj)] &
                        (up attributive) = - &
                        not (up subj-form) & not (up vcomp-form) &
                        (up subj-num) c pl & (up tense) = present &
                        (up aux) = be & (up num) = pl .

is ~                             % existential coupled with 'there'.
        v       eqns    (up pred) = exists >> [(up vcomp)]-[(up subj)] &
                        (up attributive) = - & (up subj-form) &
                        (up subj-form) c there & (up vcomp-num) c sg &
                        (up person) = third & (up tense) = present &
                        (up num) = sg & (up aux) = there_be
                                 % attributive.
                or      (up attributive) = + & (up tense) = present &
                        (up aux) = be &
                        (up vcomp-subj) = (up subj) &
                        (up pred) = attribute >> [(up vcomp)]-[(up subj)] &
                        (up subj-num) c sg &
                        (up person) = third & (up num) = sg
                                 % equative.
                or      (up tense) = present & (up attributive) = - &
                        (up aux) = be &
                        (up pred) = equate >> [(up vcomp)=(up subj)] &
                        (up subj-num) c sg &
                        (up person) = third & (up num) = sg   .

does ~
        v       eqns    (up aux) = do & (up person) = third &
                        (up tense) = present &
                        (up num) = sg & (up subj-num) c sg &
                        (up pred) = do >> [(up vcomp)]-[(up subj)] &
                        (up vcomp-subj) = (up subj) &
                        (up vcomp-nonfinite) .
has ~
        v       eqns    (up aux) = have & (up num) = sg &
                        (up subj-num) c sg &
                        (up pred) = have >> [(up vcomp)]-[(up subj)] &
                        (up vcomp-subj) = (up subj) &
                        (up vcomp-mv) = have .
have ~
        v       eqns    (up aux) = have & (up num) = pl &
                        (up subj-num) c pl &
                        (up pred) = have >> [(up vcomp)]-[(up subj),(up obj)] &
                        (up vcomp-subj) = (up obj) .

with ~
```

```
        v       eqns    (up aux) = have &     % 'with' functioning as 'have'.
                        (up pred) = have >> [(up vcomp)]-[(up subj)] &
                        (up vcomp-subj) = (up subj) &
                        (up subj-num) c pl .

                /* ---- Entries for prepositions ---- */

by ~
        p       eqns    (up pcase) = by .
from ~
        p       eqns    (up pcase) = from .
in ~
        p       eqns    (up pcase) = in &
                        (up pred) = in >> [(up subj),(up obj)] &
                        (up sem) = in(subj,obj) &
                        (up subj-domain) = place &
                        (up obj-domain) = place .
into ~
        p       eqns    (up pcase) = into .
of ~
        p       eqns    (up pcase) = of .
through ~
        p       eqns    (up pcase) = thru .
to ~
        p       eqns    (up pcase) = to .

                /* ---- Entries for category : r ---- */

a ~
        r       eqns    (up r) = a & (up def) = - &
                        (up num) = sg & (up quantifier) = a .
the ~
        r       eqns    (up r) = the &           % takes number from np head.
                        (up def) = + &
                        (up quantifier) = the .

                /* ---- Entries for category : q ---- */

more ~
        deg     eqns    (up q) = more .

                /* ---- Entries for category : neg ---- */

not ~
        neg     eqns    (up neg) = + .

                /* ---- Entries for category : meas ---- */

million ~
        meas    eqns    (up meas) = million .

        /* ************************************************************   */
```

Appendix C

F-structure Production Operators

The three operators all cause modifications to a collection of entities and variable assigments *C*. First the sub-operator "substitute" is defined. This is used in the definition of all three operators The operators are described in Kaplan and Bresnan [1982, p273] :

C.1 Definition of Substitute

For two entities *old* and *new*, Substitute[*new*, *old*] replaces all occurrences of *old* in *C* with *new*, assigns *new* as the value of variables that previously had *old* as their assignment (in addition to any variables that had *new* as their value previously), and removes *old* from *C*. Applying the substitute operator makes all previous designators of *old* and *new* designators of *new*.

C.2 Definition of Locate

The "locate" operator takes a designator *D* as input. If successful, it finds a value for *D* in a possibly modified entity collection.

If D is an entity in C
Then Locate[D] is simply D
Else **IF** D is a symbol **or** semantic form character string
  Then Locate[D] is the symbol **or** semantic form with that representation
  Else **If** D is a variable,
    Then **If** D is already assigned a value in C
      Then Locate[D] is that value
      Else, a new place-holder is added to C **and**
        assigned as the value of D **and**
        Locate[D] is that new place-holder
    Else D is a function-application expression of the form (F S)
      Let F and S be the entities Locate[F] and Locate[S], respectively
      If S is **not** a symbol **or** place-holder **or**
        F is **not** an F-structure **or** place-holder
      Then the F-description has no solution
      Else **If** F is an F-structure
        Then **If** S is a symbol or place-holder with a value defined in F

Then Locate[D] is that value
Else S is a place-holder **or**
a symbol for which F has no value **and**
F is modifier to define a new place-holder as the value of S **and** Locate[D] is that place-holder
Else F is a place-holder **and**
a new F-structure F' is constructed with a single pair that assigns a new place-holder value to S **and**
Substitute[F', F] is performed **and**
Locate[D] is the new place-holder value.

C.3 Definition of Merge

The Merge operator is like Locate is defined recursively. It takes two entities E_1 and E_2 as input. Its result is an entity E, which might be newly constructed. The new entity is substituted for both E_1 and E_2 in C so that all designators of E_1 and E_2 become designators of E instead.

If E_1 and E_2 are the same entity
Then Merge[E_1 ,E_2] is that entity and C is not modified
Else If E_1 and E_2 are both symbols or both semantic forms
Then the F-description has no solution
Else **If** E_1 and E_2 are both F-structures
Then let A_1 and A_2 be the sets of attributes of E_1 and E_2 respectively **and**
a new F-structure E constructed where :

$$E = \{ \langle a, v \rangle \mid a \in A_1 \cup A_2 \text{ and } v = \text{Merge[Locate[}(E_1 a)\text{], Locate[}(E_2 a)\text{]]} \}$$

Substitute[E, E_1] and Substitute[E, E_2] are both performed **and** the result of Merge[E_1, E_2] is then E
Else **If** E_1 and E_2 are both sets
Then a new set $E = E_1 \cup E_2$ is constructed, Substitute[E, E_1] **and**
Substitute[E, E_2] are both performed and the result of Merge[E_1, E_2] is then E
Else **If** E_1 is a place-holder
Then Substitute[E_2, E_1] is performed **and**
the result of Merge[E_1, E_2] is E_2
Else **If** E_2 is a place-holder
Then Substitute[E_1, E_2] is performed **and**
the result of Merge[E_1, E_2] is E_1
Else E_1 and E_2 are entities of different types **and**

the F-description has no solution.

C.4 Definition of Include

The Include operator takes two arguments a new set with a single member *e* and another entity *s*. The operator may simple be described as :

Perform Merge[{e}, s]

Thus if *s* is a set the merge operator will be performed directly to include *e* in *s* and form a new set.

Appendix D

Query Corpus

The query corpus consists of the following twenty-three queries, taken from the Chat-80 interface system :

(Q1)	What rivers are there ?
(Q2)	Does Afghanistan border China ?
(Q3)	What is the capital of upper-volta ?
(Q4)	Where is the largest country ?
(Q5)	Which countries are European ?
(Q6)	Which country's capital is London ?
(Q7)	Which is the largest African country ?
(Q8)	How large is the smallest American country ?
(Q9)	What is the ocean that borders African countries and that borders Asian countries ?
(Q10)	What are the capitals of the countries bordering the Baltic ?
(Q11)	Which countries are bordered by two seas ?
(Q12)	How many countries does the Danube flow through ?
(Q13)	What is the total area of the countries south of the Equator and not in Australasia ?
(Q14)	What is the average area of the countries in each continent ?
(Q15)	Is there more than one country in each continent ?
(Q16)	Is there some ocean that does not border any country ?
(Q17)	What are the countries from which a river flows into the Black-Sea ?
(Q18)	What are the continents no country in which contains more than two cities whose population exceeds 1 million ?
(Q19)	Which country bordering the Mediterranean borders a country that is bordered by a country whose population exceeds the population of India ?
(Q20)	Which countries have a population exceeding 10 million ?
(Q21)	Which countries with a population exceeding 10 million border the Atlantic ?
(Q22)	What percentage of countries border each ocean ?
(Q23)	What countries are there in Europe ?

Appendix E

EBNF Description of LFG Notations

E.1 Grammar Notation

Terminals in bold type are defined as Prolog operators.

[grammar]	::=	[rule] '.' [rest_rules].
[rest_rules]	::=	[grammar] | [empty].
[rule]	::=	[lhs] '**- - ->**' [rhs].
[lhs]	::=	[category].
[rhs]	::=	[rhs_cat] [rest_rhs] |
[rest_rhs]	::=	',' [rhs] | [empty].
[rhs_cat]	::=	[word] | '**bnd**' [category_def] '**eqns**' [equations] | '**lnk**' [category_def] '**eqns**' [equations] | '**conj**' [category_def] '**eqns**' [equations] | [category_def] '**eqns**' [equations].
[category_def]	::=	[category] '*****' [number] | [category] '*****' | [category].
[equations]	::=	[equation] '**&**' [equations] | [equation].
[equation]	::=	'**up**' '**=**' '**down**' | '**down**' '**set_val_of**' '(' '**up**' [function] ')' | '(' '**up**' '(' '**down**' '**pcase**' ')' ')' '**=**' '**down**' | '**complete**' '(' '**up**' '(' '**down**' '**pcase**' ')' ')' '**=**' '**down**' | '(' '**up**' [function] ')' '**=**' '**controllee**' [sub_script] | '(' '**down**' [function] ')' '**=**' '**controllee**' [sub_script] | '(' '**down**' [function] '**-**' [function] ')' '**=**' '**controllee**' [sub_script] | '**up**' '**=**' '**controllee**' [sub_script] '(' '**up**' [function] ')' '**=**' '**controller**' [super_script] [sub_script] | '**down**' '**=**' '**controller**' [super_script] [sub_script] | '(' '**up**' [function] '**-**' [feature] ')' '**c**' [value] | '(' '**down**' [function] '**-**' [feature] ')' '**c**' [value] | '(' '**up**' '**pred**' ')' '**=**' [predicate] '**>>**' '[' [governed] ']' | '(' '**down**' '**pred**' ')' '**=**' [predicate] '**>>**' '[' [governed] ']' | '(' '**up**' '**sem**' ')' '**=**' [sem_def] | '(' '**down**' '**sem**' ')' '**=**' [sem_def] | '**not**' '(' '**up**' [feature] '**c**' [value] | '**not**' '(' '**down**' [feature] '**c**' [value] | '**not**' '(' '**up**' [function] '**-**' [feature] '**c**' [value] | '**not**' '(' '**down**' [function] '**-**' [feature] '**c**' [value] | '(' '**up**' [function] '**-**' '**domain**' ')' '**=**' [type] '(' '**down**' [function] '**-**' '**domain**' ')' '**=**' [type] | '(' '**up**' [function] ')' '**=**' '**down**' | '**complete**' '(' '**up**' [function] ')' '**=**' '**down**' | '(' '**up**' [feature] ')' '**=**' '(' '**down**' [feature] ')' |

```
                  '(' 'up' [feature] ')' '=' [value] |
                  '(' 'down' [feature] ')' '=' [value] |
                  '(' 'up' [feature] ')' 'c' [value] |
                  '(' 'down' [feature] ')' 'c' [value] |
                  'not' '(' 'up' [feature] ')' |
                  'not' '(' 'down' [feature] ')' |
                  '(' 'up' [feature] ')' |
                  '(' 'down' [feature] ')'
                  '(' 'up' [function] '-' [feature] ')' '=' 'down'
                  '(' 'up' [function] '-' [feature] ')'
                  '(' 'down' [function] '-' [feature] ')'
                  '(' 'down' [function] '-' [function] ')' '=' '(' 'up' [function] ')'.
[governed]    ::= '(' 'up' [designator] ')' [other_gov].
[other_gov]   ::= [governed] | [empty].
[predicate]   ::= 'have' | 'population' ......    /* Prolog symbol, semantic form name */
[sem_def]     ::= [sem_pred] '(' [sem_args] ')'.
[sem_args]    ::= [sem_arg] [other_sems].
[sem_arg]     ::= 'quant' | [designator].
[other_sems]  ::= ',' [sem_args] | [empty]
[sem_pred]    ::= 'area' | 'country' ......        /* Prolog symbol, database predicate name */
[super_script] ::= 'super' [category].
[sub_script]  ::= 'sub' [category].
[category]    ::= 's' | 'np' | 'vp' | 's1' | 'v' | 'n' ......   /* Prolog symbol.*/
[function]    ::= [designator] | [other_func].
[designator]  ::= 'subj' | 'obj' | 'vcomp' | ......
[other_func]  ::= 'q' | 'focus' | ......                  /* Prolog symbol */
[feature]     ::= 'num' | 'gen' | 'tense' | 'pers' ......  /* Prolog symbol */
[number]      ::= '1' | '2' | '3' | '4' ......             /* integer */
[value]       ::= 'sg' | 'past' ......                     /* any Prolog symbol */
[word]        ::= 'that' | ''s' ......   /* word which will be added to the lexicon */
```

E.2 Lexicon Notation

```
[lexicon]        ::= [lexical_entry] '.' [rest_entries].
[rest_entries]   ::= [lexicon] | [empty].
[lexical_entry]  ::= [lexical_word] '~' [descriptions].
[descriptions]   ::= [category] 'eqns' [cat_defs] [other_cats].
[other_cats]     ::= 'and' [descriptions] | [empty].
[cat_defs]       ::= [equations] [other_eqns].
[other_eqns]     ::= 'or' [cat_defs] | [empty].
[equations]      ::= [equation] [rest_eqns].
[rest_eqns]      ::= '&' [equations] | [empty].
[equation]       ::= '(' 'up' [feature] ')' '=' [value] |
                     'up' '=' 'controllee' [sub_script] |
                     '(' 'up' 'pred' ')' '=' [pred_def] |
                     '(' 'up' 'sem' ')' '=' [sem_def] |
                     '(' 'up' 'domain' ')' '=' [type] |
                     '(' 'up' 'quantifier' ')' '=' [quantifier] |
                     '(' 'up' [function] '-' [function] '-' 'domain' ')' '=' [type] |
                     '(' 'up' 'quant' '-' 'domain' ')' '=' [type] |
                     '(' 'up' [function] '-' 'domain' ')' '=' [type] |
```

```
                   '(' 'up' [function] '-' [feature] ')' 'c' [value] |
                   '(' 'up' [feature] ')' 'c' [value] |
                   '(' 'up' [function] '-' [function] ')' '=' '(' 'up' [function] ')' |
                   '(' 'up' [function] '-' [feature] ')' |
                   '(' 'up' [feature] ')' |
                   '(' 'up' [function] '-' [feature] ')' '=' [value] |
                   'not' '(' 'up' [function] '-' [feature] ')' '=' [value] |
                   'not' '(' 'up' [feature] ')' |
                   'not' '(' 'up' [function] '-' [feature] ')' 'c' [value] |
                   'not' '(' 'up' [feature] ')' 'c' [value].
[pred_def]     ::= [pred_name] [pred_args].
[pred_args]    ::= '>>' '[' [p_args] ']' | [empty].
[p_args]       ::= '(' 'up' [function] ')' [other_functs].
[other_functs] ::= [p_args] | [empty].
[sem_def]      ::= [sem_pred] '(' [sem_args] ')'.
[sem_args]     ::= [sem_arg] [other_sems].
[sem_arg]      ::= 'quant' | [designator].
[other_sems]   ::= ',' [sem_args] | [empty]
[sub_script]   ::= 'sub' [category].
[value]        ::= /* any Prolog symbol */
[function]     ::= [designator] | [other_func].
[designator]   ::= 'subj' | 'obj' | 'vcomp' | ......
[other_func]   ::= 'q' | 'focus' | ......                /* any Prolog symbol */
[category]     ::= 's' | 'np' | 'vp' | 's1' | 'v' | 'n' ......   /* any Prolog symbol. */
[type]         ::= 'country' | 'continent' ......   /* Prolog symbol, defined as domain type */
[pred_name]    ::= 'size' | 'country' ......        /* Prolog symbol, semantic form name */
[sem_pred]     ::= 'area' | 'country' ......        /* Prolog symbol, database predicate name */
```

Appendix F

Sample Interactions

The following trace from a Quintus Prolog engine and Emacs editor, running on a Sun 3/60 workstation with four megabytes of main memory under Unix Version 3.4 [Bourne, 1982], contains :

- code compilation,
- database meta-data production,
- grammar pre-processing,
- lexicon pre-processing,
- grammar compilation,
- lexicon compilation,
- tracing information from execution of the twenty-three test queries (given in Appendix D).

Each query trace shows :

- the query being executed,
- the list of complete edges added to the WI Chart each with the form : 'edge (Starting vertex number) to (Ending vertex number) of category (Grammatical category)',
- the parse time (in micro-seconds),
- an outline of the F-structure produced (an information structure where I is an index used to co-index F-structure portions, S is the typed slot list, Q the quantifier, P the semantic predicate and F the F-structure itself (features and functions),
- the time taken (in micro-seconds) to translate the F-structure into an initial logical expression and the initial logical expression itself,
- the time taken (in micro-seconds) to plan the query and the planned query itself,
- the time taken to execute the query against the Chat-80 database and the output (solution).

```
Quintus Prolog Release 2.2 (Sun-3, Unix 3.4)
Copyright (C) 1987, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700
[consulting ./users/sun/pg/simpkink/prolog.ini...]
prolog.ini executed
[prolog.ini consulted 0.100 sec 244 bytes]

| ?- compile(top_level).
[compiling /users/sun/pg/simpkink/it/top_level.pl...]
 [compiling /users/sun/pg/simpkink/it/process.pl...]
  [compiling /users/sun/pg/simpkink/it/define_lfg.pl...]
  [define_lfg.pl compiled in module define_lfg 1.650 sec 4,152 bytes]
  [compiling /users/sun/pg/simpkink/it/reader.pl...]
   [compiling /users/sun/q2.2/library/ctypes.pl...]
    [compiling /users/sun/q2.2/library/between.pl...]
     [compiling /users/sun/q2.2/library/types.pl...]
     [types.pl compiled in module types 4.934 sec 5,568 bytes]
    [between.pl compiled in module between 8.283 sec 8,324 bytes]
   [ctypes.pl compiled in module ctypes 12.283 sec 13,916 bytes]
  [reader.pl compiled in module reader 14.600 sec 16,292 bytes]
  [compiling /users/sun/pg/simpkink/it/del_mod.pl...]
  [del_mod.pl compiled in module delete_module 0.700 sec 604 bytes]
  [compiling /users/sun/pg/simpkink/it/parser.pl...]
   [compiling /users/sun/pg/simpkink/it/counters.pl...]
   [counters.pl compiled in module counters 0.717 sec 720 bytes]
   [compiling /users/sun/pg/simpkink/it/fast_basics.pl...]
   [fast_basics.pl compiled in module fast_basics 19.500 sec 7,120 bytes]
   [compiling /users/sun/pg/simpkink/it/lookup.pl...]
    [compiling /users/sun/pg/simpkink/it/spelling.pl...]
    [spelling.pl compiled in module spelling_corrector
     2.100 sec 1,848 bytes]
   [lookup.pl compiled in module lookup 3.800 sec 3,568 bytes]
   [compiling /users/sun/pg/simpkink/it/new_info.pl...]
   [new_info.pl compiled in module new_info 0.717 sec 684 bytes]
   [compiling /users/sun/pg/simpkink/it/eval_eqns.pl...]
    [compiling /users/sun/pg/simpkink/it/fs_basics.pl...]
     [compiling /users/sun/pg/simpkink/it/greater_than.pl...]
     [greater_than.pl compiled in module greater_than
      0.733 sec 472 bytes]
    [fs_basics.pl compiled in module fs_basics 3.834 sec 3,552 bytes]
    [compiling /users/sun/pg/simpkink/it/fs_functs.pl...]
     [compiling /users/sun/pg/simpkink/it/type_system.pl...]
      [compiling /users/sun/pg/simpkink/it/geo_types.pl...]
      [geo_types.pl compiled in module geo_types 0.800 sec 828 bytes]
     [type_system.pl compiled in module type_system
      1.817 sec 1,836 bytes]
     [compiling /users/sun/pg/simpkink/it/unify.pl...]
     [unify.pl compiled in module unify 21.767 sec 12,696 bytes]
    [fs_functs.pl compiled in module fs_functs 32.483 sec 21,648 bytes]
    [compiling /users/sun/pg/simpkink/it/wf_fs.pl...]
    [wf_fs.pl compiled in module wf_fs 11.550 sec 6,880 bytes]
   [eval_eqns.pl compiled in module eval_eqns 77.150 sec 49,572 bytes]
   [compiling /users/sun/pg/simpkink/it/make_links.pl...]
    [compiling /users/sun/pg/simpkink/it/gram_pp.pl...]
     [compiling /users/sun/q2.2/library/findall.pl...]
     [findall.pl compiled in module findall 1.367 sec 1,448 bytes]
     [compiling /users/sun/pg/simpkink/it/more_basics.pl...]
     [more_basics.pl compiled in module more_basics
      1.200 sec 1,072 bytes]
     [compiling /users/sun/pg/simpkink/it/gram_eqns.pl...]
      [compiling /users/sun/pg/simpkink/it/desig.pl...]
      [desig.pl compiled in module desig 0.734 sec 748 bytes]
     [gram_eqns.pl compiled in module gram_eqns 21.066 sec 12,236 bytes]
     [compiling /users/sun/pg/simpkink/it/gram_wrds.pl...]
     [gram_wrds.pl compiled in module gram_wrds 0.833 sec 656 bytes]
    [gram_pp.pl compiled in module gram_pp 46.267 sec 29,000 bytes]
```

```
 [make_links.pl compiled in module make_links 53.933 sec 33,864 bytes]
 [compiling /users/sun/pg/simpkink/it/traces.pl...]
  [compiling /users/sun/pg/simpkink/it/pretty.pl...]
  [pretty.pl compiled in module pretty 6.417 sec 4,748 bytes]
 [traces.pl compiled in module traces 8.017 sec 6,624 bytes]
 [compiling /users/sun/pg/simpkink/it/graphic.pl...]
 [graphic.pl compiled in module graphic 0.700 sec 408 bytes]
 [compiling /users/sun/pg/simpkink/it/extend.pl...]
 [extend.pl compiled in module extend 42.017 sec 19,884 bytes]
[parser.pl compiled in module parser 218.517 sec 132,780 bytes]
[compiling /users/sun/pg/simpkink/it/execute.pl...]
 [compiling /users/sun/pg/simpkink/it/database.pl...]
  [compiling /users/sun/pg/simpkink/it/db_borders.rel...]
  [db_borders.rel compiled in module 'db_borders.rel'
   22.117 sec 31,988 bytes]
  [compiling /users/sun/pg/simpkink/it/db_city.rel...]
  [db_city.rel compiled in module 'db_city.rel' 2.667 sec 3,736 bytes]
  [compiling /users/sun/pg/simpkink/it/db_contains.rel...]
  [db_contains.rel compiled in module 'db_contains.rel'
   9.016 sec 12,884 bytes]
  [compiling /users/sun/pg/simpkink/it/db_country.rel...]
  [db_country.rel compiled in module db_country_rel
   12.700 sec 18,364 bytes]
  [compiling /users/sun/pg/simpkink/it/db_river.rel...]
  [db_river.rel compiled in module 'db_river.rel'
   3.367 sec 2,448 bytes]
  [compiling /users/sun/pg/simpkink/it/db_aggregate.pl...]
   [compiling /users/sun/q2.2/library/aggregate.pl...]
   [aggregate.pl compiled in module aggregate 3.200 sec 2,144 bytes]
  [db_aggregate.pl compiled in module db_aggregate
   6.033 sec 4,292 bytes]
 [database.pl compiled in module database 63.634 sec 81,960 bytes]
 [compiling /users/sun/pg/simpkink/it/set_of1.pl...]
 [set_of1.pl compiled in module set_of1 1.184 sec 896 bytes]
[execute.pl compiled in module execute 69.050 sec 86,640 bytes]
[compiling /users/sun/pg/simpkink/it/translate.pl...]
 [compiling /users/sun/pg/simpkink/it/pre_sem.pl...]
  [compiling /users/sun/pg/simpkink/it/quants.pl...]
  [quants.pl compiled in module quants 13.900 sec 8,396 bytes]
 [pre_sem.pl compiled in module pre_sem 24.483 sec 14,960 bytes]
[translate.pl compiled in module translate 69.817 sec 40,940 bytes]
[compiling /users/sun/pg/simpkink/it/make_static.pl...]
[make_static.pl compiled in module make_static 1.567 sec 1,124 bytes]
[compiling /users/sun/pg/simpkink/it/pre_execute.pl...]
 [compiling /users/sun/pg/simpkink/it/plan_query.pl...]
  [compiling /users/sun/q2.2/library/occurs.pl...]
  [occurs.pl compiled in module occurs 3.317 sec 2,540 bytes]
  [compiling /users/sun/pg/simpkink/it/db_meta.pl...]
  [db_meta.pl compiled in module db_meta 24.916 sec 14,340 bytes]
 [plan_query.pl compiled in module plan_query 53.084 sec 32,464 bytes]
[pre_execute.pl compiled in module pre_execute 56.266 sec 34,592 bytes]
[compiling /users/sun/pg/simpkink/it/lex_pp.pl...]
 [compiling /users/sun/pg/simpkink/it/lex_eqns.pl...]
 [lex_eqns.pl compiled in module lex_eqns 7.633 sec 4,312 bytes]
[lex_pp.pl compiled in module lex_pp 21.300 sec 13,460 bytes]
[process.pl compiled in module process 463.483 sec 341,944 bytes]


    Ok ready to start session

 ---------------------------------------------------------
 SELECT MODE :
 ---------------------------------------------------------
 Enter a query          :  e      Enter a query no.     :  n
 Preprocess grammar     : pg      Preprocess lexicon    : pl
 Compile grammar        : cg      Compile lexicon       : cl
 Turn tracing on        :  t      Turn tracing off      : nt
```

```
Meta-Data on DB      : md        Quit                  :  q
-----------------------------------------------------
Mode ? > md.
------------------------
File Containing Database Spec. ? : db_spec.

[consulting /users/sun/pg/simpkink/it/db_spec.pl...]
[db_spec.pl consulted in module db_spec 1.250 sec 2,432 bytes]
  --------------------------------
  Producing Meta-Data on : database
  --------------------------------


      Processed Relation : african/1
          Predicate Size : 780
          Domain Size : 8
      Processed Relation : american/1
          Predicate Size : 640
          Domain Size : 8
      Processed Relation : area/1 Has an Integer Argument
      Processed Relation : area/2
          Predicate Size : integer
          Domain 1 : integer
          Domain 2 : 1610
      Processed Relation : asian/1
          Predicate Size : 1090
          Domain Size : 8
      Processed Relation : borders/2
          Predicate Size : 8560
          Domain 1 : 50
          Domain 2 : 50
      Processed Relation : capital/1
          Predicate Size : 1560
          Domain Size : 10
      Processed Relation : capital/2
          Predicate Size : 1560
          Domain 1 : 10
          Domain 2 : 10
      Processed Relation : circle_of_latitude/1
          Predicate Size : 50
          Domain Size : 10
      Processed Relation : city/1
          Predicate Size : 710
          Domain Size : 10
      Processed Relation : contains/2
          Predicate Size : 7960
          Domain 1 : 89
          Domain 2 : 27
      Processed Relation : continent/1
          Predicate Size : 60
          Domain Size : 10
      Processed Relation : country/1
          Predicate Size : 1560
          Domain Size : 10
      Processed Relation : drains/2
          Predicate Size : 410
          Domain 1 : 10
          Domain 2 : 40
      Processed Relation : eastof/2 Has all Integer Arguments
      Processed Relation : european/1
          Predicate Size : 720
          Domain Size : 8
      Processed Relation : exceeds/2 Has all Integer Arguments
      Processed Relation : flows/2
          Predicate Size : 880
          Domain 1 : 21
          Domain 2 : 16
```

```
Processed Relation : flows/3
    Predicate Size : 880
    Domain 1 : 41
    Domain 2 : 53
    Domain 3 : 49
Processed Relation : in/2
    Predicate Size : 7960
    Domain 1 : 27
    Domain 2 : 89
Processed Relation : latitude/1 Has an Integer Argument
Processed Relation : latitude/2
    Predicate Size : integer
    Domain 1 : integer
    Domain 2 : 1610
Processed Relation : longitude/1 Has an Integer Argument
Processed Relation : longitude/2
    Predicate Size : integer
    Domain 1 : integer
    Domain 2 : 1560
Processed Relation : northof/2 Has all Integer Arguments
Processed Relation : ocean/1
    Predicate Size : 50
    Domain Size : 10
Processed Relation : place/1
    Predicate Size : 1910
    Domain Size : 10
Processed Relation : population/1 Has an Integer Argument
Processed Relation : population/2
    Predicate Size : integer
    Domain 1 : integer
    Domain 2 : 2270
Processed Relation : region/1
    Predicate Size : 180
    Domain Size : 10
Processed Relation : rises/2
    Predicate Size : 410
    Domain 1 : 10
    Domain 2 : 17
Processed Relation : river/1
    Predicate Size : 410
    Domain Size : 10
Processed Relation : sea/1
    Predicate Size : 60
    Domain Size : 10
Processed Relation : seamass/1
    Predicate Size : 110
    Domain Size : 10
Processed Relation : southof/2 Has all Integer Arguments
Processed Relation : westof/2 Has all Integer Arguments
Processed Relation : = /2 Has all Integer Arguments
Processed Relation : < /2 Has all Integer Arguments
Processed Relation : > /2 Has all Integer Arguments

------------------------
File for meta-data storage ? : db_temp.
------------------------
[compiling /users/sun/pg/simpkink/it/db_temp...]
[db_temp compiled in module db_meta 6.650 sec 4,516 bytes]
Finished processing database

---------------------------------------------------------
SELECT MODE :
---------------------------------------------------------
Enter a query        :  e     Enter a query no.      :  n
Preprocess grammar   : pg     Preprocess lexicon     : pl
Compile grammar      : cg     Compile lexicon        : cl
Turn tracing on      :  t     Turn tracing off       : nt
```

```
Meta-Data on DB     : md      Quit                    :  q
----------------------------------------------------------
Mode ? > pg.
------------------------
grammar file (q to quit) : grammar.
------------------------

Rules Syntactically Ok : 53 rules in all.
------------------------
grammar file (q to quit) : q.
------------------------
ok quit
------------------------
Starting to pre-process grammar rules

Grammar rules pre-processed

----------------------------------------------------------
SELECT MODE :
----------------------------------------------------------
Enter a query       :  e      Enter a query no.       :  n
Preprocess grammar  : pg      Preprocess lexicon      : pl
Compile grammar     : cg      Compile lexicon         : cl
Turn tracing on     :  t      Turn tracing off        : nt
Meta-Data on DB     : md      Quit                    :  q
----------------------------------------------------------
Mode ? > pl.
------------------------
lexicon file (q to quit) : lex_dom.
------------------------
Starting to pre-process dictionary
Dictionary file :  lex_dom

processed :  african
processed :  american
processed :  asian
processed :  european
processed :  area
processed :  areas
processed :  capital
processed :  capitals
processed :  cities
processed :  continent
processed :  continents
processed :  countries
processed :  country
processed :  ocean
processed :  oceans
processed :  population
processed :  sea
processed :  seas
processed :  river
processed :  rivers
processed :  border
processed :  borders
processed :  bordered
processed :  bordering
processed :  contain
processed :  contains
processed :  exceeds
processed :  exceeding
processed :  flow
processed :  flows
processed :  atlantic
processed :  australasia
processed :  afghanistan
processed :  baltic
```

```
processed :  black_sea
processed :  china
processed :  danube
processed :  equator
processed :  europe
processed :  india
processed :  london
processed :  mediterranean
processed :  upper_volta
processed :  south

Finished pre-processing dictionary

lexicon file (q to quit) : lex_gen.
------------------------
delete existing lexical entries (y/n) : n.
------------------------
Starting to pre-process dictionary
Dictionary file :  lex_gen

processed :  large
processed :  small
processed :  average
processed :  largest
processed :  smallest
processed :  total
processed :  any
processed :  each
processed :  how
processed :  how many
processed :  no
processed :  some
processed :  what
processed :  where
processed :  which
processed :  whose
processed :  that
processed :  there
processed :  percentage
processed :  one
processed :  two
processed :  1
processed :  2
processed :  10
processed :  18
processed :  are
processed :  is
processed :  does
processed :  has
processed :  have
processed :  with
processed :  by
processed :  from
processed :  in
processed :  into
processed :  of
processed :  through
processed :  to
processed :  a
processed :  the
processed :  more
processed :  not
processed :  million

Finished pre-processing dictionary

lexicon file (q to quit) : q.
```

```
     ------------------------
     ok quit
     -----------------------------------------------------------
     SELECT MODE :
     -----------------------------------------------------------
     Enter a query        :  e       Enter a query no.      :  n
     Preprocess grammar   : pg       Preprocess lexicon     : pl
     Compile grammar      : cg       Compile lexicon        : cl
     Turn tracing on      :  t       Turn tracing off       : nt
     Meta-Data on DB      : md       Quit                   :  q
     -----------------------------------------------------------
     Mode ? > cg.
     ------------------------
     Compiler output file for rules ? (q to quit) : temp_rules.
     ------------------------------
     Deleting Grammar Pre-processor
     Module :  gram_pp  deleted.

[compiling /users/sun/pg/simpkink/it/temp_rules...]
[temp_rules compiled in module parser 25.300 sec 13,456 bytes]
     Compiler output file for links ? : temp_links.
     ------------------------
[compiling /users/sun/pg/simpkink/it/temp_links...]
[temp_links compiled in module make_links 2.766 sec 1,692 bytes]
     -----------------------------------------------------------
     SELECT MODE :
     -----------------------------------------------------------
     Enter a query        :  e       Enter a query no.      :  n
     Preprocess grammar   : pg       Preprocess lexicon     : pl
     Compile grammar      : cg       Compile lexicon        : cl
     Turn tracing on      :  t       Turn tracing off       : nt
     Meta-Data on DB      : md       Quit                   :  q
     -----------------------------------------------------------
     Mode ? > cl.
     ------------------------
     Compiler output file for lexicon ? (q to quit) : temp_lex.
     ------------------------
     Deleting Lexicon Pre-processor
     Module :  lex_pp  deleted.

[compiling /users/sun/pg/simpkink/it/temp_lex...]
[temp_lex compiled in module lookup 64.616 sec 32,596 bytes]

     -----------------------------------------------------------
     SELECT MODE :
     -----------------------------------------------------------
     Enter a query        :  e       Enter a query no.      :  n
     Preprocess grammar   : pg       Preprocess lexicon     : pl
     Compile grammar      : cg       Compile lexicon        : cl
     Turn tracing on      :  t       Turn tracing off       : nt
     Meta-Data on DB      : md       Quit                   :  q
     -----------------------------------------------------------
     Mode ? > t.
     ------------------------
     Tracing is turned on.
     -----------------------------------------------------------
     SELECT MODE :
     -----------------------------------------------------------
     Enter a query        :  e       Enter a query no.      :  n
     Preprocess grammar   : pg       Preprocess lexicon     : pl
     Compile grammar      : cg       Compile lexicon        : cl
     Turn tracing on      :  t       Turn tracing off       : nt
     Meta-Data on DB      : md       Quit                   :  q
     -----------------------------------------------------------
     Mode ? > n.
     ------------------------
         Number (1-23) ? > 1.
```

```
        ------------------------
        [what,rivers,are,there]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 1 to 3 of category np_head
edge 1 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category v
edge 3 to 4 of category v
edge 3 to 4 of category v
edge 4 to 4 of category e
edge 4 to 4 of category npe
edge 4 to 4 of category vpl
edge 4 to 5 of category n
edge 4 to 5 of category np
edge 5 to 5 of category e
edge 5 to 5 of category npe
edge 5 to 5 of category vpl
edge 3 to 5 of category sl

     Parse Time = 1700

     Created an F-structure

**********
| I : []
| S : [subj=B:C,vcomp=B:river]
| Q : q
| P : null(subj)
| F : ---------
   attributive= -
   aux= be
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : river(F)
      | F : ---------
         det= wh
         num= pl
         pred= river
         ----------
      **********
   num= pl
   pred= existential_be(vcomp,subj)
   q=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
         ----------
      **********
   subj=
      **********
      | I : []
      | S : []
      | Q : []
      | P : []
      | F : ---------
         form= there
```

```
    ----------
   **********
tense= present
vcomp=
   **********
   | I : 1
   | S : []
   | Q : wh
   | P : river(F)
   | F : ---------
      det= wh
      num= pl
      pred= river
      ----------
   **********
----------
**********

     Translation Time = 33

     Created a semantic representation

wh(A,
     river(A)
  ).



     Planning Time = 0

     Representation after Planning

wh(A,
     river(A)
  ).



     Execution Time = 200

       amazon, amu_darya, amur, brahmaputra, colorado, congo_river,
       cubango, danube, don, elbe, euphrates, ganges, hwang_ho,
       indus, irrawaddy, lena, limpopo, mackenzie, mekong, mississippi,
       murray, niger_river, nile, ob, oder, orange, orinoco,
       parana, rhine, rhone, rio_grande, salween, senegal_river,
       tagus, vistula, volga, volta, yangtze, yenisei, yukon,
       and zambesi

       ----------------------------------------------------------
       SELECT MODE :
       ----------------------------------------------------------
       Enter a query        :  e      Enter a query no.      :  n
       Preprocess grammar   : pg      Preprocess lexicon     : pl
       Compile grammar      : cg      Compile lexicon        : cl
       Turn tracing on      :  t      Turn tracing off       : nt
       Meta-Data on DB      : md      Quit                   :  q
       ----------------------------------------------------------
       Mode ? > n.
       ------------------------
           Number (1-23) ? > 2.
       ------------------------
       [does,afghanistan,border,china]

edge 1 to 2 of category v
edge 2 to 2 of category e
edge 2 to 3 of category pn
edge 2 to 3 of category np
```

```
edge 3 to 3 of category e
edge 3 to 4 of category v
edge 4 to 4 of category e
edge 4 to 4 of category npe
edge 4 to 5 of category pn
edge 4 to 5 of category np
edge 3 to 5 of category vp

     Parse Time = 767

     Created an F-structure

**********
| I : []
| S : [subj=B:country,vcomp=C:D]
| Q : y_n
| P : passto(vcomp)
| F : ---------
   aux= do
   mood= y_n
   num= sg
   person= third
   pred= do(vcomp,subj)
   subj=
      **********
      | I : []
      | S : domain=country
      | Q : []
      | P : afghanistan
      | F : ---------
         num= sg
         pn_type= country
         pred= afghanistan
         ----------
      **********
   tense= present
   vcomp=
      **********
      | I : []
      | S : [obj=I:location,subj=J:location]
      | Q : []
      | P : borders(J,I)
      | F : ---------
         nonfinite= +
         obj=
            **********
            | I : []
            | S : domain=country
            | Q : []
            | P : china
            | F : ---------
               num= sg
               pn_type= country
               pred= china
               ----------
            **********
         pred= border(subj,obj)
         subj=
            **********
            | I : []
            | S : domain=country
            | Q : []
            | P : afghanistan
            | F : ---------
               num= sg
               pn_type= country
               pred= afghanistan
```

```
                  ----------
              **********
          ----------
      **********
  ----------
**********

      Translation Time = 50

      Created a semantic representation

yn(
   borders(afghanistan,china)
  ).



      Planning Time = 0

      Representation after Planning

yn(
   borders(afghanistan,china)
  ).



      Execution Time = 0

        indeed

       -------------------------------------------------------
       SELECT MODE :
       -------------------------------------------------------
       Enter a query        :  e     Enter a query no.     :  n
       Preprocess grammar   : pg     Preprocess lexicon    : pl
       Compile grammar      : cg     Compile lexicon       : cl
       Turn tracing on      :  t     Turn tracing off      : nt
       Meta-Data on DB      : md     Quit                  :  q
       -------------------------------------------------------
       Mode ? > n.
       ------------------------
           Number (1-23) ? > 3.
       ------------------------
       [what,is,the,capital,of,upper_volta]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 3 to 3 of category e
edge 3 to 3 of category npe
edge 3 to 3 of category vpl
edge 3 to 4 of category r
edge 3 to 4 of category the
edge 4 to 4 of category e
edge 4 to 5 of category n
edge 3 to 5 of category np_head
edge 3 to 5 of category np
edge 4 to 5 of category n
edge 3 to 5 of category np_head
edge 3 to 5 of category np
edge 5 to 5 of category e
edge 5 to 6 of category p
edge 3 to 6 of category np_head
```

```
edge 3 to 6 of category np_head
edge 6 to 6 of category e
edge 6 to 7 of category pn
edge 6 to 7 of category np
edge 5 to 7 of category pp
edge 5 to 7 of category ppadj
edge 3 to 7 of category np
edge 3 to 7 of category np
edge 3 to 7 of category np
edge 7 to 7 of category e
edge 7 to 7 of category npe
edge 7 to 7 of category vp1
edge 2 to 7 of category s1

     Parse Time = 3633

     Created an F-structure

**********
| I : []
| S : [subj=B:city,vcomp=B:city]
| Q : q
| P : equate(vcomp,subj)
| F : ---------
   attributive= -
   aux= be
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
         ----------
      **********
   num= sg
   person= third
   pred= equate(vcomp,subj)
   q=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
         ----------
      **********
   subj=
      **********
      | I : []
      | S : [of-obj=F:G]
      | Q : the
      | P : capital(F,H)
      | F : ---------
         def= +
         num= sg
         of=
            **********
            | I : []
            | S : [obj=F:country]
            | Q : []
            | P : []
            | F : ---------
               obj=
                  **********
```

```
                    | I : []
                    | S : domain=country
                    | Q : []
                    | P : upper_volta
                    | F : ---------
                       num= sg
                       pn_type= country
                       pred= upper_volta
                       ----------
                    **********
                 pcase= of
                 ----------
              **********
           pred= capital(of-obj)
           r= the
           ----------
        **********
     tense= present
     vcomp=
        **********
        | I : 1
        | S : []
        | Q : wh
        | P : []
        | F : ---------
           det= wh
           ----------
        **********
     ----------
**********

     Translation Time = 34

     Created a semantic representation

wh(A,
     the_sg(A,
           capital(upper_volta,A)
           )
  ).


     Planning Time = 17

     Representation after Planning

wh(A,
     capital(upper_volta,A)
  ).


     Execution Time = 17

       found : ouagadougou

      ---------------------------------------------------------
      SELECT MODE :
      ---------------------------------------------------------
      Enter a query        :  e     Enter a query no.      :  n
      Preprocess grammar   : pg     Preprocess lexicon     : pl
      Compile grammar      : cg     Compile lexicon        : cl
      Turn tracing on      :  t     Turn tracing off       : nt
      Meta-Data on DB      : md     Quit                   :  q
      ---------------------------------------------------------
      Mode ? > n.
```

```
    ------------------------
        Number (1-23) ? > 4.
    ------------------------
    [where,is,the,largest,country]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 3 to 3 of category e
edge 3 to 3 of category npe
edge 3 to 3 of category vpl
edge 3 to 4 of category r
edge 3 to 4 of category the
edge 4 to 4 of category e
edge 4 to 5 of category agadj
edge 5 to 5 of category e
edge 5 to 6 of category n
edge 3 to 6 of category np
edge 6 to 6 of category e
edge 6 to 6 of category npe
edge 6 to 6 of category vpl
edge 2 to 6 of category s1
edge 2 to 6 of category s1

     Parse Time = 2067

     Created an F-structure
**********
| I : []
| S : [subj=B:country,vcomp=C:D]
| Q : q
| P : passto(vcomp)
| F : ---------
   attributive= +
   aux= be
   focus=
      **********
      | I : 1
      | S : [subj=G:H]
      | Q : wh
      | P : in(G,I)
      | F : ---------
         adverb= +
         det= wh
         pcase= in
         pred= in(subj)
         subj=
            **********
            | I : []
            | S : []
            | Q : the
            | P : country(J)
            | F : ---------
               aggregates=
                  **********
                  | I : []
                  | S : []
                  | Q : []
                  | P : largest(K)
                  | F : ---------
                     aggregate= +
                     pred= largest
                     ----------
                  **********
```

```
                 _______________
                 def= +
                 num= sg
                 pred= country
                 r= the
                 ----------
              **********
           ----------
        **********
     num= sg
     person= third
     pred= attribute_be(vcomp,subj)
     q=
        **********
        | I : 1
        | S : [subj=G:H]
        | Q : wh
        | P : in(G,I)
        | F : ---------
           adverb= +
           det= wh
           pcase= in
           pred= in(subj)
           subj=
              **********
              | I : []
              | S : []
              | Q : the
              | P : country(J)
              | F : ---------
                 aggregates=
                    **********
                    | I : []
                    | S : []
                    | Q : []
                    | P : largest(K)
                    | F : ---------
                       aggregate= +
                       pred= largest
                       ----------
                    **********
                    _______________
                 def= +
                 num= sg
                 pred= country
                 r= the
                 ----------
              **********
           ----------
        **********
     subj=
        **********
        | I : []
        | S : []
        | Q : the
        | P : country(J)
        | F : ---------
           aggregates=
              **********
              | I : []
              | S : []
              | Q : []
              | P : largest(K)
              | F : ---------
                 aggregate= +
                 pred= largest
                 ----------
```

```
              **********
              ______________
           def= +
           num= sg
           pred= country
           r= the
           ----------
        **********
     tense= present
     vcomp=
        **********
        | I : 1
        | S : [subj=G:H]
        | Q : wh
        | P : in(G,I)
        | F : ---------
           adverb= +
           det= wh
           pcase= in
           pred= in(subj)
           subj=
              **********
              | I : []
              | S : []
              | Q : the
              | P : country(J)
              | F : ---------
                 aggregates=
                    **********
                    | I : []
                    | S : []
                    | Q : []
                    | P : largest(K)
                    | F : ---------
                       aggregate= +
                       pred= largest
                       ----------
                    **********
                    ______________
                 def= +
                 num= sg
                 pred= country
                 r= the
                 ----------
              **********
           ----------
        **********
     ----------
  **********

     Translation Time = 67

     Created a semantic representation

  wh(A,
     setof(B,
             country(B)
           ,C) &
     aggreg(largest(D),country,C,E) &
     in(E,A)
    ).



     Planning Time = 33

     Representation after Planning
```

```
wh(A,
     setof(B,
             country(B)
           ,C) &
     aggreg(largest(D),country,C,E) &
     in(E,A)
  ).


     Execution Time = 2100

       northern_asia, and asia

      --------------------------------------------------------
      SELECT MODE :
      --------------------------------------------------------
      Enter a query       :  e      Enter a query no.     :  n
      Preprocess grammar  : pg      Preprocess lexicon    : pl
      Compile grammar     : cg      Compile lexicon       : cl
      Turn tracing on     :  t      Turn tracing off      : nt
      Meta-Data on DB     : md      Quit                  :  q
      --------------------------------------------------------
      Mode ? > n.
      ------------------------
          Number (1-23) ? > 5.
      ------------------------
      [which,countries,are,european]

edge 1 to 2 of category relpn
edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 1 to 3 of category np_head
edge 1 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category v
edge 3 to 4 of category v
edge 3 to 4 of category v
edge 4 to 4 of category e
edge 4 to 4 of category npe
edge 4 to 4 of category vpl
edge 4 to 5 of category adj
edge 4 to 5 of category ap
edge 3 to 5 of category s1
edge 3 to 5 of category s1

     Parse Time = 1617

     Created an F-structure

**********
| I : []
| S : [subj=B:country,vcomp=B:country]
| Q : q
| P : equate(vcomp,subj)
| F : ---------
   attributive= -
   aux= be
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : country(E)
```

```
        | F : ---------
            det= wh
            num= pl
            pred= country
            ----------
        **********
   num= pl
   pred= equate(vcomp,subj)
   q=
        **********
        | I : 1
        | S : []
        | Q : wh
        | P : []
        | F : ---------
            det= wh
            ----------
        **********
   subj=
        **********
        | I : 1
        | S : []
        | Q : wh
        | P : country(E)
        | F : ---------
            det= wh
            num= pl
            pred= country
            ----------
        **********
   tense= present
   vcomp=
        **********
        | I : []
        | S : []
        | Q : []
        | P : european(F)
        | F : ---------
            aggregate= -
            pred= european
            ----------
        **********
   ----------
**********

     Translation Time = 50

     Created a semantic representation

wh(A,
     european(A) &
     country(A)
  ).



     Planning Time = 33

     Representation after Planning

wh(A,
     european(A) &
     { country(A) }
  ).
```

```
Execution Time = 200

  bulgaria, czechoslovakia, east_germany, hungary, poland,
  romania, denmark, finland, norway, sweden, albania, andorra,
  cyprus, greece, italy, malta, monaco, portugal, san_marino,
  spain, yugoslavia, austria, belgium, eire, france, iceland,
  liechtenstein, luxembourg, netherlands, switzerland,
  united_kingdom, and west_germany

 -------------------------------------------------------
 SELECT MODE :
 -------------------------------------------------------
 Enter a query        :  e     Enter a query no.     :  n
 Preprocess grammar   : pg     Preprocess lexicon    : pl
 Compile grammar      : cg     Compile lexicon       : cl
 Turn tracing on      :  t     Turn tracing off      : nt
 Meta-Data on DB      : md     Quit                  :  q
 -------------------------------------------------------
 Mode ? > n.
 ------------------------
     Number (1-23) ? > 6.
 ------------------------
 [which,country,'s,capital,is,london]

edge 1 to 2 of category relpn
edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 1 to 3 of category np_head
edge 1 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category 's
edge 1 to 4 of category det
edge 4 to 4 of category e
edge 4 to 5 of category n
edge 4 to 5 of category n
edge 1 to 5 of category np_head
edge 1 to 5 of category np
edge 5 to 5 of category e
edge 5 to 6 of category v
edge 5 to 6 of category v
edge 5 to 6 of category v
edge 6 to 6 of category e
edge 6 to 6 of category npe
edge 6 to 6 of category vp1
edge 6 to 7 of category pn
edge 6 to 7 of category np
edge 6 to 7 of category vp1
edge 5 to 7 of category s1
edge 5 to 7 of category s1

     Parse Time = 2417

     Created an F-structure

**********
| I : []
| S : [subj=B:city,vcomp=B:city]
| Q : q
| P : equate(vcomp,subj)
| F : ---------
   attributive= -
   aux= be
   focus=
      **********
      | I : 3
```

```
    | S : [poss=E:country]
    | Q : []
    | P : capital(E,F)
    | F : ---------
        num= sg
        poss=
           **********
           | I : 1
           | S : []
           | Q : wh
           | P : country(G)
           | F : ---------
               det= wh
               num= sg
               pred= country
               ----------
           **********
        pred= capital(poss)
        ----------
    **********
num= sg
person= third
pred= equate(vcomp,subj)
q=
    **********
    | I : 1
    | S : []
    | Q : wh
    | P : []
    | F : ---------
        det= wh
        ----------
    **********
subj=
    **********
    | I : 3
    | S : [poss=E:country]
    | Q : []
    | P : capital(E,F)
    | F : ---------
        num= sg
        poss=
           **********
           | I : 1
           | S : []
           | Q : wh
           | P : country(G)
           | F : ---------
               det= wh
               num= sg
               pred= country
               ----------
           **********
        pred= capital(poss)
        ----------
    **********
tense= present
vcomp=
    **********
    | I : []
    | S : domain=city
    | Q : []
    | P : london
    | F : ---------
        num= sg
        pn_type= city
        pred= london
```

```
          ----------
        **********
    ----------
**********

     Translation Time = 50

     Created a semantic representation

wh(A,
     country(A) &
     capital(A,london)
  ).


     Planning Time = 17

     Representation after Planning

wh(A,
     capital(A,london) &
     { country(A) }
  ).


     Execution Time = 17

       found : united_kingdom

      --------------------------------------------------------
      SELECT MODE :
      --------------------------------------------------------
      Enter a query        :  e      Enter a query no.      :  n
      Preprocess grammar   : pg      Preprocess lexicon     : pl
      Compile grammar      : cg      Compile lexicon        : cl
      Turn tracing on      :  t      Turn tracing off       : nt
      Meta-Data on DB      : md      Quit                   :  q
      --------------------------------------------------------
      Mode ? > n.
      ------------------------
          Number (1-23) ? > 7.
      ------------------------
      [which,is,the,largest,african,country]

edge 1 to 2 of category relpn
edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 3 to 3 of category e
edge 3 to 3 of category npe
edge 3 to 3 of category vp1
edge 3 to 4 of category r
edge 3 to 4 of category the
edge 4 to 4 of category e
edge 4 to 5 of category agadj
edge 5 to 5 of category e
edge 5 to 6 of category adj
edge 6 to 6 of category e
edge 6 to 7 of category n
edge 3 to 7 of category np
edge 7 to 7 of category e
edge 7 to 7 of category npe
```

```
edge 7 to 7 of category vpl
edge 2 to 7 of category s1

     Parse Time = 2066

     Created an F-structure

**********
| I : []
| S : [subj=B:country,vcomp=B:country]
| Q : q
| P : equate(vcomp,subj)
| F : ---------
   attributive= -
   aux= be
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
         ----------
      **********
   num= sg
   person= third
   pred= equate(vcomp,subj)
   q=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
         ----------
      **********
   subj=
      **********
      | I : []
      | S : []
      | Q : the
      | P : country(F)
      | F : ---------
         adjs=
            **********
            | I : []
            | S : []
            | Q : []
            | P : african(G)
            | F : ---------
               aggregate= -
               pred= african
               ----------
            **********
            ______________
         aggregates=
            **********
            | I : []
            | S : []
            | Q : []
            | P : largest(H)
            | F : ---------
               aggregate= +
               pred= largest
               ----------
```

```
                **********
              ______________
            def= +
            num= sg
            pred= country
            r= the
            ----------
         **********
   tense= present
   vcomp=
         **********
         | I : 1
         | S : []
         | Q : wh
         | P : []
         | F : ---------
            det= wh
            ----------
         **********
   ----------
**********
```

Translation Time = 33

Created a semantic representation

```
wh(A,
     setof(B,
             country(B) &
             african(B)
           ,C) &
     aggreg(largest(D),country,C,A)
  ).
```

Planning Time = 17

Representation after Planning

```
wh(A,
     setof(B,
             african(B) &
             { country(B) }
           ,C) &
     aggreg(largest(D),country,C,A)
  ).
```

Execution Time = 517

found : sudan

```
-------------------------------------------------------
SELECT MODE :
-------------------------------------------------------
Enter a query        : e     Enter a query no.    : n
Preprocess grammar   : pg    Preprocess lexicon   : pl
Compile grammar      : cg    Compile lexicon      : cl
Turn tracing on      : t     Turn tracing off     : nt
Meta-Data on DB      : md    Quit                 : q
-------------------------------------------------------
Mode ? > n.
------------------------
    Number (1-23) ? > 8.
------------------------
```

```
[how,large,is,the,smallest,american,country]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category adj
edge 1 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category v
edge 3 to 4 of category v
edge 3 to 4 of category v
edge 4 to 4 of category e
edge 4 to 4 of category npe
edge 4 to 4 of category vpl
edge 4 to 5 of category r
edge 4 to 5 of category the
edge 5 to 5 of category e
edge 5 to 6 of category agadj
edge 6 to 6 of category e
edge 6 to 7 of category adj
edge 7 to 7 of category e
edge 7 to 8 of category n
edge 4 to 8 of category np
edge 8 to 8 of category e
edge 8 to 8 of category npe
edge 8 to 8 of category vpl
edge 3 to 8 of category sl
edge 3 to 8 of category sl
edge 3 to 8 of category sl

     Parse Time = 2300

     Created an F-structure

**********
| I : []
| S : [subj=B:country,vcomp=C:D]
| Q : q
| P : passto(vcomp)
| F : ---------
   attributive= +
   aux= be
   focus=
      **********
      | I : 1
      | S : [subj=G:H]
      | Q : wh
      | P : area(G,I)
      | F : ---------
         det= wh
         meas= +
         pred= large(subj)
         subj=
            **********
            | I : []
            | S : []
            | Q : the
            | P : country(J)
            | F : ---------
               adjs=
                  **********
                  | I : []
                  | S : []
                  | Q : []
                  | P : american(K)
                  | F : ---------
                     aggregate= -
```

```
                    pred= american
                    ----------
                 **********

                 ______________
              aggregates=
                 **********
                 | I : []
                 | S : []
                 | Q : []
                 | P : smallest(L)
                 | F : ---------
                    aggregate= +
                    pred= smallest
                    ----------
                 **********

                 ______________
              def= +
              num= sg
              pred= country
              r= the
              ----------
           **********
        ----------
     **********
  num= sg
  person= third
  pred= attribute_be(vcomp,subj)
  q=
     **********
     | I : 1
     | S : []
     | Q : wh
     | P : []
     | F : ---------
        det= wh
        ----------
     **********
  subj=
     **********
     | I : []
     | S : []
     | Q : the
     | P : country(J)
     | F : ---------
        adjs=
           **********
           | I : []
           | S : []
           | Q : []
           | P : american(K)
           | F : ---------
              aggregate= -
              pred= american
              ----------
           **********

           ______________
        aggregates=
           **********
           | I : []
           | S : []
           | Q : []
           | P : smallest(L)
           | F : ---------
              aggregate= +
              pred= smallest
              ----------
           **********
```

```
            __________________
            def= +
            num= sg
            pred= country
            r= the
            ----------
          **********
      tense= present
      vcomp=
          **********
          | I : 1
          | S : [subj=G:H]
          | Q : wh
          | P : area(G,I)
          | F : ---------
             det= wh
             meas= +
             pred= large(subj)
             subj=
                **********
                | I : []
                | S : []
                | Q : the
                | P : country(J)
                | F : ---------
                   adjs=
                      **********
                      | I : []
                      | S : []
                      | Q : []
                      | P : american(K)
                      | F : ---------
                         aggregate= -
                         pred= american
                         ----------
                      **********
                      __________________
                   aggregates=
                      **********
                      | I : []
                      | S : []
                      | Q : []
                      | P : smallest(L)
                      | F : ---------
                         aggregate= +
                         pred= smallest
                         ----------
                      **********
                      __________________
                   def= +
                   num= sg
                   pred= country
                   r= the
                   ----------
                **********
             ----------
          **********
     ----------
**********

      Translation Time = 66

      Created a semantic representation

wh(A,
     setof(B,
            country(B) &
```

```
             american(B)
          ,C) &
      aggreg(smallest(D),country,C,E) &
      area(E,A)
   ).



      Planning Time = 50

      Representation after Planning

wh(A,
      setof(B,
              american(B) &
              { country(B) }
          ,C) &
      aggreg(smallest(D),country,C,E) &
      area(E,A)
   ).



      Execution Time = 334

        found : 0--ksqmiles

       ------------------------------------------------------
       SELECT MODE :
       ------------------------------------------------------
       Enter a query        :  e     Enter a query no.    :  n
       Preprocess grammar   : pg     Preprocess lexicon   : pl
       Compile grammar      : cg     Compile lexicon      : cl
       Turn tracing on      :  t     Turn tracing off     : nt
       Meta-Data on DB      : md     Quit         .       :  q
       ------------------------------------------------------
       Mode ? > n.    .
       -----------------------
           Number (1-23) ? > 9.
       -----------------------

[what,is,the,ocean,that,borders,african,countries,and,that,borders,american,countrie
s]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 3 to 3 of category e
edge 3 to 3 of category npe
edge 3 to 3 of category vpl
edge 3 to 4 of category r
edge 3 to 4 of category the
edge 4 to 4 of category e
edge 4 to 5 of category n
edge 3 to 5 of category np_head
edge 3 to 5 of category np
edge 5 to 5 of category e
edge 5 to 5 of category npe
edge 5 to 5 of category vpl
edge 2 to 5 of category sl
edge 2 to 5 of category sl
edge 2 to 5 of category sl
edge 5 to 6 of category relpn
edge 6 to 6 of category e
```

```
edge 6 to 7 of category v
edge 7 to 7 of category e
edge 7 to 7 of category npe
edge 7 to 7 of category vpl
edge 7 to 8 of category adj
edge 8 to 8 of category e
edge 8 to 9 of category n
edge 7 to 9 of category np
edge 6 to 9 of category vp
edge 5 to 9 of category rel_s
edge 3 to 9 of category np
edge 9 to 9 of category e
edge 9 to 9 of category npe
edge 9 to 9 of category vpl
edge 2 to 9 of category sl
edge 2 to 9 of category sl
edge 2 to 9 of category sl
edge 9 to 10 of category and
edge 10 to 10 of category e
edge 10 to 11 of category relpn
edge 11 to 11 of category e
edge 11 to 12 of category v
edge 12 to 12 of category e
edge 12 to 12 of category npe
edge 12 to 12 of category vpl
edge 12 to 13 of category adj
edge 13 to 13 of category e
edge 13 to 14 of category n
edge 12 to 14 of category np
edge 11 to 14 of category vp
edge 10 to 14 of category rel_s
edge 5 to 14 of category rel_s
edge 3 to 14 of category np
edge 14 to 14 of category e
edge 14 to 14 of category npe
edge 14 to 14 of category vpl
edge 2 to 14 of category sl

     Parse Time = 5267

     Created an F-structure

**********
| I : []
| S : [subj=B:C,vcomp=B:C]
| Q : q
| P : equate(vcomp,subj)
| F : ---------
   attributive= -
   aux= be
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
         ----------
      **********
   num= sg
   person= third
   pred= equate(vcomp,subj)
   q=
      **********
      | I : 1
      | S : []
```

```
     | Q : wh
     | P : []
     | F : ---------
        det= wh
        ----------
     **********
subj=
     **********
     | I : []
     | S : []
     | Q : []
     | P : []
     | F : ---------
        head=
           **********
           | I : 2
           | S : []
           | Q : the
           | P : ocean(H)
           | F : ---------
              def= +
              num= sg
              pred= ocean
              r= the
              ----------
           **********
        mod=
           **********
           | I : []
           | S : []
           | Q : []
           | P : []
           | F : ---------
              conjs=
                 **********
                 | I : []
                 | S : [obj=J:location,subj=K:location]
                 | Q : []
                 | P : borders(K,J)
                 | F : ---------
                    obj=
                       **********
                       | I : []
                       | S : []
                       | Q : []
                       | P : country(N)
                       | F : ---------
                          adjs=
                             **********
                             | I : []
                             | S : []
                             | Q : []
                             | P : american(O)
                             | F : ---------
                                aggregate= -
                                pred= american
                                ----------
                             **********
                             __________
                          num= pl
                          pred= country
                          ----------
                       **********
                    pred= border(subj,obj)
                    rel= +
                    subj=
                       **********
```

```
                | I : 2
                | S : []
                | Q : the
                | P : ocean(H)
                | F : ---------
                   def= +
                   num= sg
                   pred= ocean
                   r= the
                   ----------
                **********
             tense= present
             ----------
          **********
          **********
          | I : []
          | S : [obj=P:location,subj=Q:location]
          | Q : []
          | P : borders(Q,P)
          | F : ---------
             obj=
                **********
                | I : []
                | S : []
                | Q : []
                | P : country(T)
                | F : ---------
                   adjs=
                      **********
                      | I : []
                      | S : []
                      | Q : []
                      | P : african(U)
                      | F : ---------
                         aggregate= -
                         pred= african
                         ----------
                      **********
                      _____________
                   num= pl
                   pred= country
                   ----------
                **********
             pred= border(subj,obj)
             rel= +
             subj=
                **********
                | I : 2
                | S : []
                | Q : the
                | P : ocean(H)
                | F : ---------
                   def= +
                   num= sg
                   pred= ocean
                   r= the
                   ----------
                **********
             tense= present
             ----------
          **********
          _____________
       ----------
    **********
 num= sg
 ----------
**********
```

```
    tense= present
    vcomp=
       **********
       ! I : 1
       ! S : []
       | Q : wh
       | P : []
       | F : ---------
          det= wh
          ----------
       **********
    ----------
**********

      Translation Time = 117

      Created a semantic representation

wh(A,
     the_sg(A,
            ocean(A) &
            country(B) &
            american(B) &
            borders(A,B) &
            country(C) &
            african(C) &
            borders(A,C)
            )
  ).



      Planning Time = 100

      Representation after Planning

wh(A,
     ocean(A) &
     { borders(A,B) &
       { american(B) } &
       { country(B) } } &
     { borders(A,C) &
       { african(C) } &
       { country(C) } }
  ).



     Execution Time = 117

       found : atlantic

      -------------------------------------------------------
      SELECT MODE :
      -------------------------------------------------------
      Enter a query       :  e     Enter a query no.     :  n
      Preprocess grammar  : pg     Preprocess lexicon    : pl
      Compile grammar     : cg     Compile lexicon       : cl
      Turn tracing on     :  t     Turn tracing off      : nt
      Meta-Data on DB     : md     Quit                  :  q
      -------------------------------------------------------
      Mode ? > n.
      ------------------------
          Number (1-23) ? > 10.
      ------------------------
      [what,are,the,capitals,of,the,countries,bordering,the,baltic]
```

```
edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 3 to 3 of category e
edge 3 to 3 of category npe
edge 3 to 3 of category vpl
edge 3 to 4 of category r
edge 3 to 4 of category the
edge 4 to 4 of category e
edge 4 to 5 of category n
edge 3 to 5 of category np_head
edge 3 to 5 of category np
edge 4 to 5 of category n
edge 3 to 5 of category np_head
edge 3 to 5 of category np
edge 5 to 5 of category e
edge 5 to 6 of category p
edge 3 to 6 of category np_head
edge 3 to 6 of category np_head
edge 6 to 6 of category e
edge 6 to 7 of category r
edge 6 to 7 of category the
edge 7 to 7 of category e
edge 7 to 8 of category n
edge 6 to 8 of category np_head
edge 6 to 8 of category np
edge 5 to 8 of category pp
edge 5 to 8 of category ppadj
edge 3 to 8 of category np
edge 3 to 8 of category np
edge 3 to 8 of category np
edge 8 to 8 of category e
edge 8 to 8 of category npe
edge 8 to 8 of category vpl
edge 2 to 8 of category s1
edge 2 to 8 of category s1
edge 2 to 8 of category s1
edge 8 to 9 of category v
edge 9 to 9 of category e
edge 9 to 9 of category npe
edge 9 to 10 of category r
edge 9 to 10 of category the
edge 10 to 10 of category e
edge 10 to 11 of category pn
edge 9 to 11 of category np
edge 8 to 11 of category vp
edge 8 to 11 of category rel_s
edge 6 to 11 of category np
edge 5 to 11 of category pp
edge 5 to 11 of category ppadj
edge 3 to 11 of category np
edge 3 to 11 of category np
edge 3 to 11 of category np
edge 11 to 11 of category e
edge 11 to 11 of category npe
edge 11 to 11 of category vpl
edge 2 to 11 of category s1

     Parse Time = 6667

     Created an F-structure

**********
| I : []
```

```
| S : [subj=B:city,vcomp=B:city]
| Q : q
| P : equate(vcomp,subj)
| F : ---------
  attributive= -
  aux= be
  focus=
     **********
     | I : 1
     | S : []
     | Q : wh
     | P : []
     | F : ---------
        det= wh
        ----------
     **********
  num= pl
  pred= equate(vcomp,subj)
  q=
     **********
     | I : 1
     | S : []
     | Q : wh
     | P : []
     | F : ---------
        det= wh
        ----------
     **********
  subj=
     **********
     | I : []
     | S : [of-obj=F:G]
     | Q : the
     | P : capital(F,H)
     | F : ---------
        def= +
        num= pl
        of=
           **********
           | I : []
           | S : [obj=F:country]
           | Q : []
           | P : []
           | F : ---------
              obj=
                 **********
                 | I : []
                 | S : []
                 | Q : []
                 | P : []
                 | F : ---------
                    head=
                       **********
                       | I : 6
                       | S : []
                       | Q : the
                       | P : country(K)
                       | F : ---------
                          def= +
                          num= pl
                          pred= country
                          r= the
                          ----------
                       **********
                    mod=
                       **********
                       | I : []
```

```
                                 | S : [obj=L:location,subj=M:location]
                                 | Q : []
                                 | P : borders(M,L)
                                 | F : ---------
                                   obj=
                                      **********
                                      | I : []
                                      | S : domain=ocean
                                      | Q : []
                                      | P : baltic
                                      | F : ---------
                                         num= sg
                                         pn_type= sea
                                         pred= baltic
                                         r= +
                                         ----------
                                      **********
                                   participle= ing
                                   pred= border(subj,obj)
                                   subj=
                                      **********
                                      | I : 6
                                      | S : []
                                      | Q : the
                                      | P : country(K)
                                      | F : ---------
                                         def= +
                                         num= pl
                                         pred= country
                                         r= the
                                         ----------
                                      **********
                                   tense= present
                                   ----------
                                **********
                           num= pl
                           ----------
                        **********
                   pcase= of
                   ----------
                **********
           pred= capital(of-obj)
           r= the
           ----------
        **********
   tense= present
   vcomp=
        **********
        | I : 1
        | S : []
        | Q : wh
        | P : []
        | F : ---------
           det= wh
           ----------
        **********
   ----------
**********

     Translation Time = 167

     Created a semantic representation

wh(A,
     of_the(B,
           country(B) &
           the_pl(A,
```

```
                  capital(B,A) &
                  borders(B,baltic)
                 )
             )
   ).

       ------------------------------------------------------
       SELECT MODE :
       ------------------------------------------------------
       Enter a query          :  e      Enter a query no.      :  n
       Preprocess grammar     : pg      Preprocess lexicon     : pl
       Compile grammar        : cg      Compile lexicon        : cl
       Turn tracing on        :  t      Turn tracing off       : nt
       Meta-Data on DB        : md      Quit                   :  q
       ------------------------------------------------------
       Mode ? > n.
       ------------------------
           Number (1-23) ? > 11.
       ------------------------
       [which,countries,are,bordered,by,two,seas]

edge 1 to 2 of category relpn
edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 1 to 3 of category np_head
edge 1 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category v
edge 3 to 4 of category v
edge 3 to 4 of category v
edge 4 to 4 of category e
edge 4 to 4 of category npe
edge 4 to 4 of category vpl
edge 4 to 5 of category v
edge 5 to 5 of category e
edge 5 to 5 of category npe
edge 5 to 6 of category p
edge 6 to 6 of category e
edge 6 to 6 of category npe
edge 5 to 6 of category ppe
edge 4 to 6 of category vpl
edge 6 to 7 of category num
edge 7 to 7 of category e
edge 7 to 8 of category n
edge 6 to 8 of category np
edge 5 to 8 of category pp
edge 4 to 8 of category vpl
edge 3 to 8 of category sl
edge 3 to 8 of category sl

     Parse Time = 2600

     Created an F-structure

**********
| I : []
| S : [subj=B:country,vcomp=C:D]
| Q : q
| P : passto(vcomp)
| F : ---------
   attributive= +
   aux= be
   focus=
      **********
      | I : 1
```

```
    | S : []
    | Q : wh
    | P : country(G)
    | F : ---------
       det= wh
       num= pl
       pred= country
       ----------
    **********
num= pl
pred= attribute_be(vcomp,subj)
q=
    **********
    | I : 1
    | S : []
    | Q : wh
    | P : []
    | F : ---------
       det= wh
       ----------
    **********
subj=
    **********
    | I : 1
    | S : []
    | Q : wh
    | P : country(G)
    | F : ---------
       det= wh
       num= pl
       pred= country
       ----------
    **********
tense= present
vcomp=
    **********
    | I : []
    | S : [by-obj=H:I,subj=J:location]
    | Q : []
    | P : borders(J,H)
    | F : ---------
       by=
          **********
          | I : []
          | S : [obj=H:location]
          | Q : []
          | P : []
          | F : ---------
             obj=
                **********
                | I : []
                | S : []
                | Q : []
                | P : sea(M)
                | F : ---------
                   card= two
                   num= pl
                   pred= sea
                   ----------
                **********
             pcase= by
             ----------
          **********
       pred= border(subj,by-obj)
       subj=
          **********
          | I : 1
```

```
                  | S : []
                  | Q : wh
                  | P : country(G)
                  | F : ---------
                     det= wh
                     num= pl
                     pred= country
                     ----------
                  **********
               tense= past
               ----------
            **********
      ----------
   **********

        Translation Time = 83

        Created a semantic representation

wh(A,
     country(A) &
     numberof(B,
               sea(B) &
               borders(A,B)
            ,2)
  ).


        Planning Time = 33

        Representation after Planning

wh(A,
     numberof(B,
               sea(B) &
               borders(A,B)
            ,2) &
     { country(A) }
  ).


        Execution Time = 534

          egypt, iran, israel, saudi_arabia, and turkey

         --------------------------------------------------------
         SELECT MODE :
         --------------------------------------------------------
         Enter a query          : e      Enter a query no.       :  n
         Preprocess grammar     : pg     Preprocess lexicon      : pl
         Compile grammar        : cg     Compile lexicon         : cl
         Turn tracing on        :  t     Turn tracing off        : nt
         Meta-Data on DB        : md     Quit                    :  q
         --------------------------------------------------------
         Mode ? > n.
         ------------------------
             Number (1-23) ? > 12.
         ------------------------
         [how,many,countries,does,the,danube,flow,through]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 1 to 3 of category det
edge 3 to 3 of category e
```

```
edge 2 to 3 of category det
edge 3 to 3 of category e
edge 3 to 4 of category n
edge 1 to 4 of category np_head
edge 1 to 4 of category np
edge 4 to 4 of category e
edge 4 to 5 of category v
edge 5 to 5 of category e
edge 5 to 5 of category npe
edge 5 to 5 of category vpl
edge 5 to 6 of category r
edge 5 to 6 of category the
edge 6 to 6 of category e
edge 6 to 7 of category pn
edge 5 to 7 of category np
edge 7 to 7 of category e
edge 7 to 7 of category npe
edge 7 to 7 of category vpl
edge 4 to 7 of category sl
edge 4 to 7 of category sl
edge 4 to 7 of category sl
edge 7 to 8 of category v
edge 8 to 8 of category e
edge 8 to 9 of category p
edge 9 to 9 of category e
edge 9 to 9 of category npe
edge 8 to 9 of category ppe
edge 7 to 9 of category vpl
edge 4 to 9 of category sl
edge 4 to 9 of category sl
edge 4 to 9 of category sl

     Parse Time = 2817

     Created an F-structure

**********
| I : []
| S : [subj=B:river,vcomp=C:D]
| Q : q
| P : passto(vcomp)
| F : ---------
   aux= do
   focus=
      **********
      | I : 2
      | S : []
      | Q : numberof
      | P : country(G)
      | F : ---------
         def= -
         det= numberof
         num= pl
         pred= country
         ----------
      **********
   num= sg
   person= third
   pred= do(vcomp,subj)
   q=
      **********
      | I : 2
      | S : []
      | Q : numberof
      | P : []
      | F : ---------
         def= -
```

```
        det= numberof
        num= pl
        ----------
     **********
   subj=
     **********
     | I : []
     | S : domain=river
     | Q : []
     | P : danube
     | F : ---------
        num= sg
        pn_type= river
        pred= danube
        r= +
        ----------
     **********
   tense= present
   vcomp=
     **********
     | I : []
     | S : [subj=J:river,thru-obj=K:L]
     | Q : []
     | P : flows(J,K)
     | F : ---------
        nonfinite= +
        pred= flow(subj,thru-obj)
        subj=
          **********
          | I : []
          | S : domain=river
          | Q : []
          | P : danube
          | F : ---------
             num= sg
             pn_type= river
             pred= danube
             r= +
             ----------
          **********
        thru=
          **********
          | I : []
          | S : [obj=K:location]
          | Q : []
          | P : []
          | F : ---------
             obj=
               **********
               | I : 2
               | S : []
               | Q : numberof
               | P : country(G)
               | F : ---------
                  def= -
                  det= numberof
                  num= pl
                  pred= country
                  ----------
               **********
             pcase= thru
             ----------
          **********
        ----------
     **********
  ----------
**********
```

```
     Translation Time = 66

     Created a semantic representation

numberof(A,
       country(A) &
       flows(danube,A)
   )



     Planning Time = 17

     Representation after Planning

numberof(A,
       flows(danube,A) &
       { country(A) }
   )



     Execution Time = 50

       found : 6

       ----------------------------------------------------------
       SELECT MODE :
       ----------------------------------------------------------
       Enter a query        :  e      Enter a query no.       :  n
       Preprocess grammar   : pg      Preprocess lexicon      : pl
       Compile grammar      : cg      Compile lexicon         : cl
       Turn tracing on      :  t      Turn tracing off        : nt
       Meta-Data on DB      : md      Quit                    :  q
       ----------------------------------------------------------
       Mode ? > n.
       ------------------------
           Number (1-23) ? > 13.
       ------------------------

[what,is,the,total,area,of,the,countries,south,of,the,equator,and,not,in,australasia]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 3 to 3 of category e
edge 3 to 3 of category npe
edge 3 to 3 of category vpl
edge 3 to 4 of category r
edge 3 to 4 of category the
edge 4 to 4 of category e
edge 4 to 5 of category agadj
edge 5 to 5 of category e
edge 5 to 6 of category n
edge 3 to 6 of category np
edge 6 to 6 of category e
edge 6 to 7 of category p
edge 3 to 7 of category np_head
edge 7 to 7 of category e
edge 7 to 8 of category r
edge 7 to 8 of category the
edge 8 to 8 of category e
edge 8 to 9 of category n
```

```
edge 7 to 9 of category np_head
edge 7 to 9 of category np
edge 6 to 9 of category pp
edge 6 to 9 of category ppadj
edge 3 to 9 of category np
edge 3 to 9 of category np
edge 9 to 9 of category e
edge 9 to 9 of category npe
edge 9 to 9 of category vp1
edge 2 to 9 of category s1
edge 2 to 9 of category s1
edge 2 to 9 of category s1
edge 9 to 10 of category pmod
edge 10 to 10 of category e
edge 10 to 11 of category p
edge 11 to 11 of category e
edge 11 to 12 of category r
edge 11 to 12 of category the
edge 12 to 12 of category e
edge 12 to 13 of category pn
edge 11 to 13 of category np
edge 10 to 13 of category pp
edge 9 to 13 of category pp
edge 9 to 13 of category ppadj
edge 7 to 13 of category np
edge 6 to 13 of category pp
edge 6 to 13 of category ppadj
edge 3 to 13 of category np
edge 3 to 13 of category np
edge 3 to 13 of category np
edge 3 to 13 of category np
edge 13 to 13 of category e
edge 13 to 13 of category npe
edge 13 to 13 of category vp1
edge 2 to 13 of category s1
edge 2 to 13 of category s1
edge 2 to 13 of category s1
edge 2 to 13 of category s1
edge 2 to 13 of category s1
edge 2 to 13 of category s1
edge 13 to 14 of category and
edge 14 to 14 of category e
edge 14 to 15 of category neg
edge 15 to 15 of category e
edge 15 to 16 of category p
edge 16 to 16 of category e
edge 16 to 17 of category pn
edge 16 to 17 of category np
edge 15 to 17 of category pp
edge 14 to 17 of category pp
edge 6 to 17 of category ppadj
edge 3 to 17 of category np
edge 9 to 17 of category ppadj
edge 7 to 17 of category np
edge 6 to 17 of category pp
edge 6 to 17 of category ppadj
edge 3 to 17 of category np
edge 3 to 17 of category np
edge 3 to 17 of category np
edge 3 to 17 of category np
edge 17 to 17 of category e
edge 17 to 17 of category npe
edge 17 to 17 of category vp1
edge 2 to 17 of category s1

    Parse Time = 10466
```

Created an F-structure

```
**********
| I : []
| S : [subj=B:area,vcomp=B:area]
| Q : q
| P : equate(vcomp,subj)
| F : ---------
   attributive= -
   aux= be
   focus=
       **********
       | I : 1
       | S : []
       | Q : wh
       | P : []
       | F : ---------
          det= wh
          ----------
       **********
   num= sg
   person= third
   pred= equate(vcomp,subj)
   q=
       **********
       | I : 1
       | S : []
       | Q : wh
       | P : []
       | F : ---------
          det= wh
          ----------
       **********
   subj=
       **********
       | I : []
       | S : [of-obj=F:G]
       | Q : the
       | P : area(F,H)
       | F : ---------
          aggregates=
              **********
              | I : []
              | S : []
              | Q : []
              | P : total(I)
              | F : ---------
                 aggregate= +
                 pred= total
                 ----------
              **********
              _______________
          def= +
          num= sg
          of=
              **********
              | I : []
              | S : [obj=F:place]
              | Q : []
              | P : []
              | F : ---------
                 obj=
                     **********
                     | I : []
                     | S : []
                     | Q : the
                     | P : country(L)
```

```
| F : ---------
   adjuncts=
      **********
      | I : []
      | S : [obj=M:place,subj=N:place]
      | Q : []
      | P : in(N,M)
      | F : ---------
         neg= +
         obj=
            **********
            | I : []
            | S : domain=continent
            | Q : []
            | P : australasia
            | F : ---------
               num= sg
               pn_type= continent
               pred= australasia
               ----------
            **********
         pcase= in
         pred= in(subj,obj)
         ----------
      **********
      **********
      | I : []
      | S : [of-obj=S:T,subj=U:location]
      | Q : []
      | P : southof(U,S)
      | F : ---------
         direction= +
         of=
            **********
            | I : []
            | S : [obj=X:location]
            | Q : []
            | P : []
            | F : ---------
               obj=
                  **********
                  | I : []
                  | S : domain=latitude
                  | Q : []
                  | P : equator
                  | F : ---------
                     num= sg
                     pred= equator
                     r= +
                     ----------
                  **********
               pcase= of
               ----------
            **********
         pred= south(subj,of-obj)
         ----------
      **********
      _____________
   def= +
   num= pl
   pred= country
   r= the
   ----------
  **********
pcase= of
----------
**********
```

```
          pred= area(of-obj)
          r= the
          ----------
       **********
   tense= present
   vcomp=
       **********
       | I : 1
       | S : []
       | Q : wh
       | P : []
       | F : ---------
          det= wh
          ----------
       **********
   ----------
**********

     Translation Time = 100

     Created a semantic representation

wh(A,
     of_the(B,
           country(B) &
           setof(C,
                   area(B,C) &
                   not ( in(B,australasia)
                      ) &
                   southof(B,equator)
                 ,D) &
           aggreg(total(E),area,D,A)
        )
  ).


     Planning Time = 83

     Representation after Planning

wh(A,
     setof(B-C,
             country(B) &
             { not ( in(B,australasia)
                   ) } &
             { southof(B,equator) } &
             { area(B,C) }
           ,D) &
     aggreg(total(E),area,D,A)
  ).


     Execution Time = 383

       found : 10228--ksqmiles

      --------------------------------------------------------
      SELECT MODE :
      --------------------------------------------------------
      Enter a query        :  e      Enter a query no.     :  n
      Preprocess grammar   : pg      Preprocess lexicon    : pl
      Compile grammar      : cg      Compile lexicon       : cl
      Turn tracing on      :  t      Turn tracing off      : nt
      Meta-Data on DB      : md      Quit                  :  q
      --------------------------------------------------------
```

```
Mode ? > n.
------------------------
    Number (1-23) ? > 14.
------------------------
[what,is,the,average,area,of,the,countries,in,each,continent]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 3 to 3 of category e
edge 3 to 3 of category npe
edge 3 to 3 of category vp1
edge 3 to 4 of category r
edge 3 to 4 of category the
edge 4 to 4 of category e
edge 4 to 5 of category agadj
edge 5 to 5 of category e
edge 5 to 6 of category n
edge 3 to 6 of category np
edge 6 to 6 of category e
edge 6 to 7 of category p
edge 3 to 7 of category np_head
edge 7 to 7 of category e
edge 7 to 8 of category r
edge 7 to 8 of category the
edge 8 to 8 of category e
edge 8 to 9 of category n
edge 7 to 9 of category np_head
edge 7 to 9 of category np
edge 6 to 9 of category pp
edge 6 to 9 of category ppadj
edge 3 to 9 of category np
edge 3 to 9 of category np
edge 9 to 9 of category e
edge 9 to 9 of category npe
edge 9 to 9 of category vp1
edge 2 to 9 of category s1
edge 2 to 9 of category s1
edge 2 to 9 of category s1
edge 9 to 10 of category p
edge 10 to 10 of category e
edge 10 to 11 of category det
edge 11 to 11 of category e
edge 11 to 12 of category n
edge 10 to 12 of category np_head
edge 10 to 12 of category np
edge 9 to 12 of category pp
edge 9 to 12 of category ppadj
edge 7 to 12 of category np
edge 6 to 12 of category pp
edge 6 to 12 of category ppadj
edge 3 to 12 of category np
edge 3 to 12 of category np
edge 3 to 12 of category np
edge 3 to 12 of category np
edge 12 to 12 of category e
edge 12 to 12 of category npe
edge 12 to 12 of category vp1
edge 2 to 12 of category s1

Parse Time = 6050

Created an F-structure
```

```
**********
| I : []
| S : [subj=B:area,vcomp=B:area]
| Q : q
| P : equate(vcomp,subj)
| F : ---------
   attributive= -
   aux= be
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
         ----------
      **********
   num= sg
   person= third
   pred= equate(vcomp,subj)
   q=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
         ----------
      **********
   subj=
      **********
      | I : []
      | S : [of-obj=F:G]
      | Q : the
      | P : area(F,H)
      | F : ---------
         aggregates=
            **********
            | I : []
            | S : []
            | Q : []
            | P : average(I)
            | F : ---------
               aggregate= +
               pred= average
               ----------
            **********
            ____________
         def= +
         num= sg
         of=
            **********
            | I : []
            | S : [obj=F:place]
            | Q : []
            | P : []
            | F : ---------
               obj=
                  **********
                  | I : []
                  | S : []
                  | Q : the
                  | P : country(L)
                  | F : ---------
                     adjuncts=
```

```
                                  **********
                                  | I : []
                                  | S : [obj=M:place,subj=N:place]
                                  | Q : []
                                  | P : in(N,M)
                                  | F : ---------
                                     obj=
                                        **********
                                        | I : []
                                        | S : []
                                        | Q : each
                                        | P : continent(Q)
                                        | F : ---------
                                           det= each
                                           num= sg
                                           pred= continent
                                           ----------
                                        **********
                                     pcase= in
                                     pred= in(subj,obj)
                                     ----------
                                  **********
                                  ______________
                              def= +
                              num= pl
                              pred= country
                              r= the
                              ----------
                           **********
                        pcase= of
                        ----------
                     **********
                  pred= area(of-obj)
                  r= the
                  ----------
               **********
            tense= present
            vcomp=
               **********
               | I : 1
               | S : []
               | Q : wh
               | P : []
               | F : ---------
                  det= wh
                  ----------
               **********
            ----------
         **********
```

Translation Time = 50

Created a semantic representation

```
wh(A,
     each(B,
           continent(B) &
           of_the(C,
                 country(C) &
                 setof(D,
                         area(C,D) &
                         in(C,B)
                        ,E) &
                 aggreg(average(F),area,E,A)
               )
        )
  ).
```

```
Planning Time = 84

Representation after Planning

wh(A,
     setof(B-C,
             continent(C) &
             setof(D-E,
                     in(D,C) &
                     { country(D) } &
                     { area(D,E) }
                   ,F) &
             aggreg(average(G),area,F,B)
           ,A)
  ).


Execution Time = 1017

  233--ksqmiles:africa, 496--ksqmiles:america, 485--ksqmiles:asia,
  543--ksqmiles:australasia, and 58--ksqmiles:europe

 -------------------------------------------------------
 SELECT MODE :
 -------------------------------------------------------
 Enter a query         : e     Enter a query no.     :  n
 Preprocess grammar    : pg    Preprocess lexicon    : pl
 Compile grammar       : cg    Compile lexicon       : cl
 Turn tracing on       :  t    Turn tracing off      : nt
 Meta-Data on DB       : md    Quit                  :  q
 -------------------------------------------------------
 Mode ? > n.
 ------------------------
     Number (1-23) ? > 15.
 ------------------------
 [is,there,more,than,one,country,in,each,continent]

edge 1 to 2 of category v
edge 1 to 2 of category v
edge 1 to 2 of category v
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 2 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category deg
edge 4 to 4 of category e
edge 4 to 5 of category than
edge 5 to 5 of category e
edge 5 to 6 of category num
edge 3 to 6 of category det
edge 6 to 6 of category e
edge 6 to 7 of category n
edge 3 to 7 of category np_head
edge 3 to 7 of category np
edge 7 to 7 of category e
edge 7 to 8 of category p
edge 8 to 8 of category e
edge 8 to 9 of category det
edge 9 to 9 of category e
edge 9 to 10 of category n
edge 8 to 10 of category np_head
edge 8 to 10 of category np
edge 7 to 10 of category pp
```

```
edge 7 to 10 of category ppadj

     Parse Time = 1883

     Created an F-structure

**********
| I : []
| S : [subj=B:C,vcomp=B:country]
| Q : y_n
| P : null(subj)
| F : ---------
   adjuncts=
      **********
      | I : []
      | S : [obj=F:place,subj=G:place]
      | Q : []
      | P : in(G,F)
      | F : ---------
         obj=
            **********
            | I : []
            | S : []
            | Q : each
            | P : continent(J)
            | F : ---------
               det= each
               num= sg
               pred= continent
               ----------
            **********
         pcase= in
         pred= in(subj,obj)
         ----------
      **********
      ____________
   attributive= -
   aux= there_be
   mood= y_n
   num= sg
   person= third
   pred= existential_be(vcomp,subj)
   subj=

      **********
      | I : []
      | S : []
      | Q : []
      | P : []
      | F : ---------
         form= there
         ----------
      **********
   tense= present
   vcomp=
      **********
      | I : []
      | S : []
      | Q : []
      | P : country(L)
      | F : ---------
         card= one
         num= sg
         pred= country
         q= more
         ----------
      **********
```

```
   ----------
**********

     Translation Time = 66

     Created a semantic representation

yn(
   each(A,
         continent(A) &
         numberof(B,
                    country(B) &
                    in(B,A)
                 ,C) &
         C > 1
      )
  ).



     Planning Time = 50

     Representation after Planning

yn(
   not ( continent(A) &
         not ( numberof(B,
                          in(B,A) &
                          { country(B) }
                       ,C) &
               { C > 1 }
             )
       )
  ).


     Execution Time = 450

       no, not true

      ---------------------------------------------------------
      SELECT MODE :
      ---------------------------------------------------------
      Enter a query       :  e       Enter a query no.     :  n
      Preprocess grammar  : pg       Preprocess lexicon    : pl
      Compile grammar     : cg       Compile lexicon       : cl
      Turn tracing on     :  t       Turn tracing off      : nt
      Meta-Data on DB     : md       Quit                  :  q
      ---------------------------------------------------------
      Mode ? > n.
      ------------------------
          Number (1-23) ? > 16.
      ------------------------
      [is,there,some,ocean,that,does,not,border,any,country]

edge 1 to 2 of category v
edge 1 to 2 of category v
edge 1 to 2 of category v
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 2 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category det
edge 4 to 4 of category e
edge 4 to 5 of category n
edge 3 to 5 of category np_head
```

```
edge 3 to 5 of category np
edge 5 to 5 of category e
edge 5 to 6 of category relpn
edge 6 to 6 of category e
edge 6 to 7 of category v
edge 7 to 7 of category e
edge 7 to 7 of category npe
edge 7 to 7 of category vpl
edge 7 to 8 of category neg
edge 8 to 8 of category e
edge 8 to 9 of category v
edge 9 to 9 of category e
edge 9 to 9 of category npe
edge 9 to 10 of category det
edge 10 to 10 of category e
edge 10 to 11 of category n
edge 9 to 11 of category np_head
edge 9 to 11 of category np
edge 8 to 11 of category vp
edge 6 to 11 of category vp
edge 5 to 11 of category rel_s
edge 3 to 11 of category np

     Parse Time = 2483

     Created an F-structure

**********
| I : []
| S : [subj=B:C,vcomp=B:D]
| Q : y_n
| P : null(subj)
| F : ---------
   attributive= -
   aux= there_be
   mood= y_n
   num= sg
   person= third
   pred= existential_be(vcomp,subj)
   subj=
      **********
      | I : []
      | S : []
      | Q : []
      | P : []
      | F : ---------
         form= there
         ----------
      **********
   tense= present
   vcomp=
      **********
      | I : []
      | S : []
      | Q : []
      | P : []
      | F : ---------
         head=
            **********
            | I : 1
            | S : []
            | Q : some
            | P : ocean(I)
            | F : ---------
               det= some
               num= sg
               pred= ocean
```

```
          ----------
       **********
  mod=
       **********
       | I : []
       | S : [subj=J:ocean,vcomp=K:L]
       | Q : []
       | P : passto(vcomp)
       | F : ---------
          aux= do
          neg= +
          num= sg
          person= third
          pred= do(vcomp,subj)
          rel= +
          subj=
             **********
             | I : 1
             | S : []
             | Q : some
             | P : ocean(I)
             | F : ---------
                det= some
                num= sg
                pred= ocean
                ----------
             **********
          tense= present
          vcomp=
             **********
             | I : []
             | S : [obj=O:location,subj=P:location]
             | Q : []
             | P : borders(P,O)
             | F : ---------
                nonfinite= +
                obj=
                   **********
                   | I : []
                   | S : []
                   | Q : any
                   | P : country(R)
                   | F : ---------
                      det= any
                      num= sg
                      pred= country
                      ----------
                   **********
                pred= border(subj,obj)
                subj=
                   **********
                   | I : 1
                   | S : []
                   | Q : some
                   | P : ocean(I)
                   | F : ---------
                      det= some
                      num= sg
                      pred= ocean
                      ----------
                   **********
                ----------
             **********
          ----------
       **********
  num= sg
  ----------
```

```
       **********
    ----------
**********

      Translation Time = 117

      Created a semantic representation

yn(
   some(A,
         ocean(A) &
         not ( any(B,
                     country(B) &
                     borders(A,B)
                  )
              )
        )
  ).



      Planning Time = 17

      Representation after Planning

yn(
   ocean(A) &
   not ( borders(A,B) &
         { country(B) }
       )
  ).



      Execution Time = 0

        indeed

       ------------------------------------------------------
       SELECT MODE :
       ------------------------------------------------------
       Enter a query         : e    Enter a query no.     :  n
       Preprocess grammar    : pg   Preprocess lexicon    : pl
       Compile grammar       : cg   Compile lexicon       : cl
       Turn tracing on       : t    Turn tracing off      : nt
       Meta-Data on DB       : md   Quit                  :  q
       ------------------------------------------------------
       Mode ? > n.
       ------------------------
           Number (1-23) ? > 17.
       ------------------------
       [what,are,the,countries,from,which,a,river,flows,into,the,black_sea]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 3 to 3 of category e
edge 3 to 3 of category npe
edge 3 to 3 of category vpl
edge 3 to 4 of category r
edge 3 to 4 of category the
edge 4 to 4 of category e
edge 4 to 5 of category n
edge 3 to 5 of category np_head
```

```
edge 3 to 5 of category np
edge 5 to 5 of category e
edge 5 to 5 of category npe
edge 5 to 5 of category vp1
edge 2 to 5 of category s1
edge 2 to 5 of category s1
edge 2 to 5 of category s1
edge 5 to 6 of category p
edge 3 to 6 of category np_head
edge 6 to 6 of category e
edge 6 to 7 of category relpn
edge 6 to 7 of category det
edge 6 to 7 of category np
edge 5 to 7 of category pp
edge 5 to 7 of category ppadj
edge 3 to 7 of category np
edge 7 to 7 of category e
edge 7 to 8 of category r
edge 8 to 8 of category e
edge 8 to 9 of category n
edge 7 to 9 of category np_head
edge 7 to 9 of category np
edge 9 to 9 of category e
edge 9 to 10 of category v
edge 10 to 10 of category e
edge 10 to 10 of category npe
edge 10 to 11 of category p
edge 11 to 11 of category e
edge 11 to 12 of category r
edge 11 to 12 of category the
edge 12 to 12 of category e
edge 12 to 13 of category pn
edge 11 to 13 of category np
edge 10 to 13 of category pp
edge 9 to 13 of category vp
edge 6 to 13 of category rel_s
edge 3 to 13 of category np
edge 13 to 13 of category e
edge 13 to 13 of category npe
edge 13 to 13 of category vp1
edge 2 to 13 of category s1

     Parse Time = 5000

     Created an F-structure

**********
| I : []
| S : [subj=B:C,vcomp=B:C]
| Q : q
| P : equate(vcomp,subj)
| F : ---------
   attributive= -
   aux= be
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
         ----------
      **********
   num= pl
   pred= equate(vcomp,subj)
   q=
```

```
        **********
        | I : 1
        | S : []
        | Q : wh
        | P : []
        | F : ---------
           det= wh
           ----------
        **********
     subj=
        **********
        | I : []
        | S : []
        | Q : []
        | P : []
        | F : ---------
           head=
              **********
              | I : 4
              | S : []
              | Q : the
              | P : country(H)
              | F : ---------
                 def= +
                 num= pl
                 pcase= from
                 pred= country
                 r= the
                 ----------
              **********
           mod=
              **********
              | I : []
              | S : [into-obj=I:J,obj=K:country,subj=L:river]
              | Q : []
              | P : flows(L,K,I)
              | F : ---------
                 into=
                    **********
                    | I : []
                    | S : [obj=I:sea]
                    | Q : []
                    | P : []
                    | F : ---------
                       obj=
                          **********
                          | I : []
                          | S : domain=sea
                          | Q : []
                          | P : black_sea
                          | F : ---------
                             num= sg
                             pred= black_sea
                             r= +
                             ----------
                          **********
                       pcase= into
                       ----------
                    **********
                 obj=
                    **********
                    | I : 4
                    | S : []
                    | Q : the
                    | P : country(H)
                    | F : ---------
                       def= +
```

```
                        num= pl
                        pcase= from
                        pred= country
                        r= the
                        ----------
                     **********
                  pred= flow(subj,obj,into-obj)
                  rel= +
                  subj=
                     **********
                     | I : []
                     | S : []
                     | Q : a
                     | P : river(R)
                     | F : ---------
                        def= -
                        num= sg
                        pred= river
                        r= a
                        ----------
                     **********
                  tense= present
                  wh= +
                  ----------
               **********
            num= pl
            ----------
         **********
      tense= present
      vcomp=
         **********
         | I : 1
         | S : []
         | Q : wh
         | P : []
         | F : ---------
            det= wh
            ----------
         **********
      ----------
**********
```

Translation Time = 166

Created a semantic representation

```
wh(A,
     the_pl(A,
            country(A) &
            a(B,
                  river(B) &
                  flows(B,A,black_sea)
                )
          )
  ).
```

Planning Time = 17

Representation after Planning

```
wh(A,
     setof(B,
              flows(C,B,black_sea) &
              { country(B) } &
              { river(C) }
```

```
          ,A)
   ).

Execution Time = 16

 romania, and soviet_union

-------------------------------------------------------
SELECT MODE :
-------------------------------------------------------
Enter a query         :  e     Enter a query no.      :  n
Preprocess grammar    : pg     Preprocess lexicon     : pl
Compile grammar       : cg     Compile lexicon        : cl
Turn tracing on       :  t     Turn tracing off       : nt
Meta-Data on DB       : md     Quit                   :  q
-------------------------------------------------------
Mode ? > n.
------------------------
    Number (1-23) ? > 18.
------------------------

[what,are,the,continents,no,country,in,which,contains,more,than,two,cities,whose,popu
lation,exceeds,1,million]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 2 to 3 of category v
edge 3 to 3 of category e
edge 3 to 3 of category npe
edge 3 to 3 of category vpl
edge 3 to 4 of category r
edge 3 to 4 of category the
edge 4 to 4 of category e
edge 4 to 5 of category n
edge 3 to 5 of category np_head
edge 3 to 5 of category np
edge 5 to 5 of category e
edge 5 to 5 of category npe
edge 5 to 5 of category vpl
edge 2 to 5 of category sl
edge 2 to 5 of category sl
edge 2 to 5 of category sl
edge 5 to 6 of category det
edge 6 to 6 of category e
edge 6 to 7 of category n
edge 5 to 7 of category np_head
edge 5 to 7 of category np
edge 7 to 7 of category e
edge 7 to 8 of category p
edge 8 to 8 of category e
edge 8 to 8 of category npe
edge 5 to 8 of category rel_s
edge 3 to 8 of category np
edge 7 to 8 of category ppe
edge 5 to 8 of category pp_pied
edge 8 to 9 of category relpn
edge 8 to 9 of category det
edge 8 to 9 of category np
edge 7 to 9 of category pp
edge 7 to 9 of category ppadj
edge 5 to 9 of category np
edge 9 to 9 of category e
```

```
edge 9 to 10 of category v
edge 10 to 10 of category e
edge 10 to 10 of category npe
edge 10 to 11 of category deg
edge 11 to 11 of category e
edge 11 to 12 of category than
edge 12 to 12 of category e
edge 12 to 13 of category num
edge 10 to 13 of category det
edge 13 to 13 of category e
edge 13 to 14 of category n
edge 10 to 14 of category np_head
edge 10 to 14 of category np
edge 9 to 14 of category vp
edge 9 to 14 of category s1
edge 5 to 14 of category rel_s
edge 3 to 14 of category np
edge 14 to 14 of category e
edge 14 to 14 of category npe
edge 14 to 14 of category vp1
edge 2 to 14 of category s1
edge 2 to 14 of category s1
edge 2 to 14 of category s1
edge 14 to 15 of category relpn
edge 15 to 15 of category e
edge 15 to 16 of category n
edge 15 to 16 of category n
edge 16 to 16 of category e
edge 16 to 17 of category v
edge 17 to 17 of category e
edge 17 to 17 of category npe
edge 17 to 18 of category num
edge 18 to 18 of category e
edge 18 to 19 of category meas
edge 17 to 19 of category np
edge 16 to 19 of category vp
edge 14 to 19 of category rel_s
edge 10 to 19 of category np
edge 9 to 19 of category vp
edge 9 to 19 of category s1
edge 5 to 19 of category rel_s
edge 3 to 19 of category np
edge 19 to 19 of category e
edge 19 to 19 of category npe
edge 19 to 19 of category vp1
edge 2 to 19 of category s1

     Parse Time = 9117

     Created an F-structure

**********
| I : []
| S : [subj=B:C,vcomp=B:C]
| Q : q
| P : equate(vcomp,subj)
| F : ---------
   attributive= -
   aux= be
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
```

```
        ----------
     **********
num= pl
pred= equate(vcomp,subj)
q=
     **********
     | I : 1
     | S : []
     | Q : wh
     | P : []
     | F : ---------
        det= wh
        ----------
     **********
subj=
     **********
     | I : []
     | S : []
     | Q : []
     | P : []
     | F : ---------
        head=
           **********
           | I : 3
           | S : []
           | Q : the
           | P : continent(H)
           | F : ---------
              def= +
              num= pl
              pred= continent
              r= the
              ----------
           **********
        mod=
           **********
           | I : []
           | S : [obj=I:location,subj=J:location]
           | Q : []
           | P : contains(J,I)
           | F : ---------
              obj=
                 **********
                 | I : []
                 | S : []
                 | Q : []
                 | P : []
                 | F : ---------
                    head=
                       **********
                       | I : 6
                       | S : []
                       | Q : []
                       | P : city(N)
                       | F : ---------
                          card= two
                          num= pl
                          pred= city
                          q= more
                          ----------
                       **********
                    mod=
                       **********
                       | I : []
                       | S : [obj=O:measure,subj=P:measure]
                       | Q : []
                       | P : exceeds(P,O)
```

```
                | F : ---------
                   case= gen
                   obj=
                      **********
                      | I : []
                      | S : []
                      | Q : []
                      | P : []
                      | F : ---------
                         card= one
                         meas= million
                         num= sg
                         ----------
                      **********
                   pred= exceeds(subj,obj)
                   rel= +
                   subj=
                      **********
                      | I : []
                      | S : [of-obj=T:city]
                      | Q : []
                      | P : population(T,U)
                      | F : ---------
                         num= sg
                         of=
                            **********
                            | I : []
                            | S : [obj=T:peopled]
                            | Q : []
                            | P : []
                            | F : ---------
                               obj=
                                  **********
                                  | I : 6
                                  | S : []
                                  | Q : []
                                  | P : city(N)
                                  | F : ---------
                                     card= two
                                     num= pl
                                     pred= city
                                     q= more
                                     ----------
                                  **********
                               pcase= of
                               ----------
                            **********
                         pred= population(of-obj)
                         ----------
                      **********
                   tense= present
                   ----------
                **********
             num= pl
             ----------
          **********
       pied=
          **********
          | I : 5
          | S : [obj=W:place,subj=X:place]
          | Q : []
          | P : in(X,W)
          | F : ---------
             num= sg
             obj=
                **********
                | I : 3
```

```
          | S : []
          | Q : the
          | P : continent(H)
          | F : ---------
             def= +
             num= pl
             pred= continent
             r= the
             ----------
          **********
       pcase= in
       pred= in(subj,obj)
       subj=
          **********
          | I : []
          | S : []
          | Q : no
          | P : country(Z)
          | F : ---------
             det= no
             num= sg
             pred= country
             ----------
          **********
       ----------
    **********
 pred= contains(subj,obj)
 rel= +
 subj=
    **********
    | I : 5
    | S : [obj=W:place,subj=X:place]
    | Q : []
    | P : in(X,W)
    | F : ---------
       num= sg
       obj=
          **********
          | I : 3
          | S : []
          | Q : the
          | P : continent(H)
          | F : ---------
             def= +
             num= pl
             pred= continent
             r= the
             ----------
          **********
       pcase= in
       pred= in(subj,obj)
       subj=
          **********
          | I : []
          | S : []
          | Q : no
          | P : country(Z)
          | F : ---------
             det= no
             num= sg
             pred= country
             ----------
          **********
       ----------
    **********
 tense= present
 wh= +
```

```
                  ----------
               **********
            num= pl
            ----------
         **********
      tense= present
      vcomp=
         **********
         | I : 1
         | S : []
         | Q : wh
         | P : []
         | F : ---------
            det= wh
            ----------
         **********
      ----------
   **********

     Translation Time = 400

     Created a semantic representation

wh(A,
     the_pl(A,
            continent(A) &
            no(B,
                   country(B) &
                   in(B,A) &
                   numberof(C,
                              city(C) &
                              population(C,D) &
                              exceeds(D,1--million) &
                              contains(B,C)
                          ,E) &
                   E > 2
                )
         )
  ).


     Planning Time = 267

     Representation after Planning

wh(A,
     setof(B,
              continent(B) &
              not ( in(C,B) &
                     { country(C) } &
                     { numberof(D,
                                  contains(C,D) &
                                  { city(D) } &
                                  { population(D,E) &
                                    { exceeds(E,1--million) } }
                              ,F) &
                       { F > 2 } }
                  )
            ,A)
  ).


     Execution Time = 700

       africa, antarctica, and australasia
```

```
-------------------------------------------------------
SELECT MODE :
-------------------------------------------------------
Enter a query        :  e     Enter a query no.      :  n
Preprocess grammar   :  pg    Preprocess lexicon     :  pl
Compile grammar      :  cg    Compile lexicon        :  cl
Turn tracing on      :  t     Turn tracing off       :  nt
Meta-Data on DB      :  md    Quit                   :  q
-------------------------------------------------------
Mode ? > n.
------------------------
    Number (1-23) ? > 19.
------------------------
```

```
[which,country,bordering,the,mediterranean,borders,a,country,that,is,bordered,by,a,co
untry,whose,population,exceeds,the,population,of,india]

edge 1 to 2 of category relpn
edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 1 to 3 of category np_head
edge 1 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category v
edge 4 to 4 of category e
edge 4 to 4 of category npe
edge 4 to 5 of category r
edge 4 to 5 of category the
edge 5 to 5 of category e
edge 5 to 6 of category pn
edge 4 to 6 of category np
edge 3 to 6 of category vp
edge 3 to 6 of category s1
edge 3 to 6 of category rel_s
edge 1 to 6 of category np
edge 6 to 6 of category e
edge 6 to 7 of category v
edge 7 to 7 of category e
edge 7 to 7 of category npe
edge 7 to 8 of category r
edge 8 to 8 of category e
edge 8 to 9 of category n
edge 7 to 9 of category np_head
edge 7 to 9 of category np
edge 6 to 9 of category vp
edge 6 to 9 of category s1
edge 9 to 9 of category e
edge 9 to 10 of category relpn
edge 10 to 10 of category e
edge 10 to 11 of category v
edge 10 to 11 of category v
edge 10 to 11 of category v
edge 11 to 11 of category e
edge 11 to 11 of category npe
edge 11 to 11 of category vp1
edge 11 to 12 of category v
edge 12 to 12 of category e
edge 12 to 12 of category npe
edge 12 to 13 of category p
edge 13 to 13 of category e
edge 13 to 13 of category npe
edge 12 to 13 of category ppe
edge 11 to 13 of category vp1
```

```
edge 13 to 14 of category r
edge 14 to 14 of category e
edge 14 to 15 of category n
edge 13 to 15 of category np_head
edge 13 to 15 of category np
edge 12 to 15 of category pp
edge 11 to 15 of category vpl
edge 9 to 15 of category rel_s
edge 9 to 15 of category rel_s
edge 7 to 15 of category np
edge 6 to 15 of category vp
edge 6 to 15 of category sl
edge 9 to 15 of category rel_s
edge 15 to 15 of category e
edge 15 to 16 of category relpn
edge 16 to 16 of category e
edge 16 to 17 of category n
edge 16 to 17 of category n
edge 17 to 17 of category e
edge 17 to 18 of category v
edge 18 to 18 of category e
edge 18 to 18 of category npe
edge 18 to 19 of category r
edge 18 to 19 of category the
edge 19 to 19 of category e
edge 19 to 20 of category n
edge 18 to 20 of category np_head
edge 18 to 20 of category np
edge 17 to 20 of category vp
edge 15 to 20 of category rel_s
edge 19 to 20 of category n
edge 18 to 20 of category np_head
edge 18 to 20 of category np
edge 20 to 20 of category e
edge 20 to 21 of category p
edge 18 to 21 of category np_head
edge 18 to 21 of category np_head
edge 21 to 21 of category e
edge 21 to 22 of category pn
edge 21 to 22 of category np
edge 20 to 22 of category pp
edge 20 to 22 of category ppadj
edge 18 to 22 of category np
edge 17 to 22 of category vp
edge 15 to 22 of category rel_s
edge 18 to 22 of category np
edge 18 to 22 of category np
edge 17 to 22 of category vp
edge 15 to 22 of category rel_s
edge 13 to 22 of category np
edge 12 to 22 of category pp
edge 11 to 22 of category vpl
edge 9 to 22 of category rel_s
edge 9 to 22 of category rel_s
edge 7 to 22 of category np
edge 6 to 22 of category vp
edge 6 to 22 of category sl

     Parse Time = 8067

     Created an F-structure

**********
| I : []
| S : [obj=B:location,subj=C:location]
| Q : q
| P : borders(C,B)
```

```
| F : ---------
  focus=
     **********
     | I : 2
     | S : []
     | Q : []
     | P : []
     | F : ---------
        head=
           **********
           | I : 1
           | S : []
           | Q : wh
           | P : country(G)
           | F : ---------
              det= wh
              num= sg
              pred= country
              ----------
           **********
        mod=
           **********
           | I : []
           | S : [obj=H:location,subj=I:location]
           | Q : []
           | P : borders(I,H)
           | F : ---------
              obj=
                 **********
                 | I : []
                 | S : domain=sea
                 | Q : []
                 | P : mediterranean
                 | F : ---------
                    num= sg
                    pn_type= sea
                    pred= mediterranean
                    r= +
                    ----------
                 **********
              participle= ing
              pred= border(subj,obj)
              subj=
                 **********
                 | I : 1
                 | S : []
                 | Q : wh
                 | P : country(G)
                 | F : ---------
                    det= wh
                    num= sg
                    pred= country
                    ----------
                 **********
              tense= present
              ----------
           **********
        num= sg
        ----------
     **********
  obj=
     **********
     | I : []
     | S : []
     | Q : []
     | P : []
     | F : ---------
```

```
head=
    **********
    | I : 3
    | S : []
    | Q : a
    | P : country(O)
    | F : ---------
       def= -
       num= sg
       pred= country
       r= a
       ----------
    **********
mod=
    **********
    | I : []
    | S : [subj=P:country,vcomp=Q:R]
    | Q : []
    | P : passto(vcomp)
    | F : ---------
       attributive= +
       aux= be
       num= sg
       person= third
       pred= attribute_be(vcomp,subj)
       rel= +
       subj=
          **********
          | I : 3
          | S : []
          | Q : a
          | P : country(O)
          | F : ---------
             def= -
             num= sg
             pred= country
             r= a
             ----------
          **********
       tense= present
       vcomp=
          **********
          | I : []
          | S : [by-obj=U:V,subj=W:location]
          | Q : []
          | P : borders(W,U)
          | F : ---------
             by=
                **********
                | I : []
                | S : [obj=U:location]
                | Q : []
                | P : []
                | F : ---------
                   obj=
                      **********
                      | I : []
                      | S : []
                      | Q : []
                      | P : []
                      | F : ---------
                         head=
                            **********
                            | I : 4
                            | S : []
                            | Q : a
                            | P : country(A1)
```

```
| F : ---------
   def= -
   num= sg
   pred= country
   r= a
   ----------
**********
mod=
   **********
   | I : []
   | S : [obj=B1:measure,subj=C1:measure]
   | Q : []
   | P : exceeds(C1,B1)
   | F : ---------
      case= gen
      obj=
         **********
         | I : []
         | S : [of-obj=F1:G1]
         | Q : the
         | P : population(F1,H1)
         | F : ---------
            def= +
            num= sg
            of=
               **********
               | I : []
               | S : [obj=F1:peopled]
               | Q : []
               | P : []
               | F : ---------
                  obj=
                     **********
                     | I : []
                     | S : domain=country
                     | Q : []
                     | P : india
                     | F : ---------
                        num= sg
                        pn_type= country
                        pred= india
                        ----------
                     **********
                  pcase= of
                  ----------
               **********
            pred= population(of-obj)
            r= the
            ----------
         **********
      pred= exceeds(subj,obj)
      rel= +
      subj=
         **********
         | I : []
         | S : [of-obj=L1:country]
         | Q : []
         | P : population(L1,M1)
         | F : ---------
            num= sg
            of=
               **********
               | I : []
               | S : [obj=L1:peopled]
               | Q : []
               | P : []
               | F : ---------
```

```
                                                          obj=
                                                             **********
                                                             | I : 4
                                                             | S : []
                                                             | Q : a
                                                             | P : country(A1)
                                                             | F : ---------
                                                                def= -
                                                                num= sg
                                                                pred= country
                                                                r= a
                                                                ----------
                                                             **********
                                                          pcase= of
                                                          ----------
                                                       **********
                                                    pred= population(of-obj)
                                                    ----------
                                                 **********
                                              tense= present
                                              ----------
                                           **********
                                        num= sg
                                        ----------
                                     **********
                                  pcase= by
                                  ----------
                               **********
                            pred= border(subj,by-obj)
                            subj=
                               **********
                               | I : 3
                               | S : []
                               | Q : a
                               | P : country(O)
                               | F : ---------
                                  def= -
                                  num= sg
                                  pred= country
                                  r= a
                                  ----------
                               **********
                            tense= past
                            ----------
                         **********
                      ----------
                   **********
                num= sg
                ----------
             **********
          pred= border(subj,obj)
          q=
             **********
             | I : 1
             | S : []
             | Q : wh
             | P : []
             | F : ---------
                det= wh
                ----------
             **********
          subj=
             **********
             | I : 2
             | S : []
             | Q : []
             | P : []
```

```
        | F : ---------
          head=
             **********
             | I : 1
             | S : []
             | Q : wh
             | P : country(G)
             | F : ---------
                det= wh
                num= sg
                pred= country
                ----------
             **********
          mod=
             **********
             | I : []
             | S : [obj=H:location,subj=I:location]
             | Q : []
             | P : borders(I,H)
             | F : ---------
                obj=
                   **********
                   | I : []
                   | S : domain=sea
                   | Q : []
                   | P : mediterranean
                   | F : ---------
                      num= sg
                      pn_type= sea
                      pred= mediterranean
                      r= +
                      ----------
                   **********
                participle= ing
                pred= border(subj,obj)
                subj=
                   **********
                   | I : 1
                   | S : []
                   | Q : wh
                   | P : country(G)
                   | F : ---------
                      det= wh
                      num= sg
                      pred= country
                      ----------
                   **********
                tense= present
                ----------
             **********
          num= sg
          ----------
       **********
   tense= present
   ----------
**********

     Translation Time = 700

     Created a semantic representation

wh(A,
     country(A) &
     a(B,
          country(B) &
          borders(A,mediterranean) &
          a(C,
```

```
                    country(C) &
                    the_sg(D,
                         population(india,D) &
                         population(C,E) &
                         exceeds(E,D) &
                         borders(B,C) &
                         borders(A,B)
                         )
                 )
           )
  ).


     Planning Time = 317

     Representation after Planning

 wh(A,
      population(india,B) &
      borders(A,mediterranean) &
      { country(A) } &
      { borders(A,C) &
        { country(C) } &
        { borders(C,D) &
          { population(D,E) &
            { exceeds(E,B) } } &
          { country(D) } } }
  ).


     Execution Time = 500

       found : turkey

     ------------------------------------------------------------
     SELECT MODE :
     ------------------------------------------------------------
     Enter a query        : e      Enter a query no.      :  n
     Preprocess grammar   : pg     Preprocess lexicon     : pl
     Compile grammar      : cg     Compile lexicon        : cl
     Turn tracing on      :  t     Turn tracing off       : nt
     Meta-Data on DB      : md     Quit                   :  q
     ------------------------------------------------------------
     Mode ? > n.
     -------------------------
         Number (1-23) ? > 20.
     -------------------------
     [which,countries,have,a,population,exceeding,10,million]

edge 1 to 2 of category relpn
edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 1 to 3 of category np_head
edge 1 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category v
edge 4 to 4 of category e
edge 4 to 5 of category r
edge 5 to 5 of category e
edge 5 to 6 of category n
edge 4 to 6 of category np_head
edge 4 to 6 of category np
edge 5 to 6 of category n
```

```
edge 4 to 6 of category np_head
edge 4 to 6 of category np
edge 6 to 6 of category e
edge 6 to 7 of category v
edge 7 to 7 of category e
edge 7 to 7 of category npe
edge 7 to 8 of category num
edge 8 to 8 of category e
edge 8 to 9 of category meas
edge 7 to 9 of category np
edge 6 to 9 of category vp
edge 6 to 9 of category rel_s
edge 4 to 9 of category rel_adj
edge 3 to 9 of category s1

     Parse Time = 2450

     Created an F-structure

**********
| I : []
| S : [obj=B:measure,subj=C:country,vcomp=D:E]
| Q : q
| P : passto(vcomp)
| F : ---------
   aux= have
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : country(H)
      | F : ---------
         det= wh
         num= pl
         pred= country
         ----------
      **********
   num= pl
   obj=
      **********
      | I : []
      | S : [of-obj=I:J]
      | Q : a
      | P : population(I,K)
      | F : ---------
         def= -
         num= sg
         of=
            **********
            | I : []
            | S : [obj=I:peopled]
            | Q : []
            | P : []
            | F : ---------
               obj=
                  **********
                  | I : 1
                  | S : []
                  | Q : wh
                  | P : country(H)
                  | F : ---------
                     det= wh
                     num= pl
                     pred= country
                     ----------
                  **********
```

```
          ----------
         **********
       pred= population(of-obj)
       r= a
       ----------
   **********
pred= have(vcomp,subj,obj)
q=
   **********
   | I : 1
   | S : []
   | Q : wh
   | P : []
   | F : ---------
      det= wh
      ----------
   **********
subj=
   **********
   | I : 1
   | S : []
   | Q : wh
   | P : country(H)
   | F : ---------
      det= wh
      num= pl
      pred= country
      ----------
   **********
vcomp=
   **********
   | I : []
   | S : [obj=M:measure,subj=N:measure]
   | Q : []
   | P : exceeds(N,M)
   | F : ---------
      obj=
         **********
         | I : []
         | S : []
         | Q : []
         | P : []
         | F : ---------
            card= ten
            meas= million
            num= pl
            ----------
         **********
      participle= ing
      pred= exceeds(subj,obj)
      subj=
         **********
         | I : []
         | S : [of-obj=I:J]
         | Q : a
         | P : population(I,K)
         | F : ---------
            def= -
            num= sg
            of=
               **********
               | I : []
               | S : [obj=I:peopled]
               | Q : []
               | P : []
               | F : ---------
                  obj=
```

```
                                **********
                                | I : 1
                                  S : []
                                  Q : wh
                                | P : country(H)
                                | F : ---------
                                    det= wh
                                    num= pl
                                    pred= country
                                    ----------
                                **********
                              ----------
                            **********
                        pred= population(of-obj)
                        r= a
                        ----------
                    **********
                tense= present
                ----------
              **********
          ----------
        **********

      Translation Time = 67

      Created a semantic representation

wh(A,
      country(A) &
      a(B,
            population(A,B) &
            exceeds(B,10--million)
         )
   ).



      Planning Time = 67

      Representation after Planning

wh(A,
      country(A) &
      population(A,B) &
      { exceeds(B,10--million) }
   ).



      Execution Time = 334

        afghanistan, algeria, argentina, australia, bangladesh,
        brazil, burma, canada, china, colombia, czechoslovakia,
        east_germany, egypt, ethiopia, france, india, indonesia,
        iran, italy, japan, kenya, mexico, morocco, nepal, netherlands,
        nigeria, north_korea, pakistan, peru, philippines, poland,
        south_africa, south_korea, soviet_union, spain, sri_lanka,
        sudan, taiwan, tanzania, thailand, turkey, united_kingdom,
        united_states, venezuela, vietnam, west_germany, yugoslavia,
        and zaire

      -----------------------------------------------------
      SELECT MODE :
      -----------------------------------------------------
      Enter a query         : e      Enter a query no.     : n
      Preprocess grammar    : pg     Preprocess lexicon    : pl
      Compile grammar       : cg     Compile lexicon       : cl
```

```
Turn tracing on      :  t      Turn tracing off       : nt
Meta-Data on DB      : md      Quit                   :  q
-----------------------------------------------------------
Mode ? > n.
------------------------
    Number (1-23) ? > 21.
------------------------
[which,countries,with,a,population,exceeding,10,million,border,the,atlantic]

edge 1 to 2 of category relpn
edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 1 to 3 of category np_head
edge 1 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category v
edge 4 to 4 of category e
edge 4 to 5 of category r
edge 5 to 5 of category e
edge 5 to 6 of category n
edge 4 to 6 of category np_head
edge 4 to 6 of category np
edge 5 to 6 of category n
edge 4 to 6 of category np_head
edge 4 to 6 of category np
edge 6 to 6 of category e
edge 6 to 7 of category v
edge 7 to 7 of category e
edge 7 to 7 of category npe
edge 7 to 8 of category num
edge 8 to 8 of category e
edge 8 to 9 of category meas
edge 7 to 9 of category np
edge 6 to 9 of category vp
edge 6 to 9 of category rel_s
edge 4 to 9 of category rel_adj
edge 9 to 9 of category e
edge 9 to 10 of category v
edge 10 to 10 of category e
edge 10 to 10 of category npe
edge 10 to 11 of category r
edge 10 to 11 of category the
edge 11 to 11 of category e
edge 11 to 12 of category pn
edge 10 to 12 of category np
edge 9 to 12 of category vp
edge 3 to 12 of category s1

   Parse Time = 3017

   Created an F-structure

**********
| I : []
| S : [subj=B:country,vcomp=C:D]
| Q : q
| P : passto(vcomp)
| F : ---------
   aux= have
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : country(G)
```

```
    | F : ---------
       det= wh
       num= pl
       pred= country
       ----------
    **********
pred= have(vcomp,subj)
q=
    **********
    | I : 1
    | S : []
    | Q : wh
    | P : []
    | F : ---------
       det= wh
       ----------
    **********
rel_adj=
    **********
    | I : []
    | S : []
    | Q : []
    | P : []
    | F : ---------
       head=
          **********
          | I : 2
          | S : [of-obj=I:country]
          | Q : a
          | P : population(I,J)
          | F : ---------
             def= -
             num= sg
             of=
                **********
                | I : []
                | S : [obj=I:peopled]
                | Q : []
                | P : []
                | F : ---------
                   obj=
                      **********
                      | I : 1
                      | S : []
                      | Q : wh
                      | P : country(G)
                      | F : ---------
                         det= wh
                         num= pl
                         pred= country
                         ----------
                      **********
                   pcase= of
                   ----------
                **********
             pred= population(of-obj)
             r= a
             ----------
          **********
       mod=
          **********
          | I : []
          | S : [obj=L:measure,subj=M:measure]
          | Q : []
          | P : exceeds(M,L)
          | F : ---------
             obj=
```

```
                    **********
                    | I : []
                    | S : []
                    | Q : []
                    | P : []
                    | F : ---------
                       card= ten
                       meas= million
                       num= pl
                       ----------
                    **********
                 participle= ing
                 pred= exceeds(subj,obj)
                 subj=
                    **********
                    | I : 2
                    | S : [of-obj=I:country]
                    | Q : a
                    | P : population(I,J)
                    | F : ---------
                       def= -
                       num= sg
                       of=
                          **********
                          | I : []
                          | S : [obj=I:peopled]
                          | Q : []
                          | P : []
                          | F : ---------
                             obj=
                                **********
                                | I : 1
                                | S : []
                                | Q : wh
                                | P : country(G).
                                | F : ---------
                                   det= wh
                                   num= pl
                                   pred= country
                                   ----------
                                **********
                             pcase= of
                             ----------
                          **********
                       pred= population(of-obj)
                       r= a
                       ----------
                    **********
                 tense= present
                 ----------
              **********
           ----------
        **********
     subj=
        **********
        | I : 1
        | S : []
        | Q : wh
        | P : country(G)
        | F : ---------
           det= wh
           num= pl
           pred= country
           ----------
        **********
     vcomp=
        **********
```

```
        | I : []
        | S : [obj=Q:location,subj=R:location]
        | Q : []
        | P : borders(R,Q)
        | F : ---------
           nonfinite= +
           obj=
              **********
              | I : []
              | S : domain=ocean
              | Q : []
              | P : atlantic
              | F : ---------
                 num= sg
                 pn_type= ocean
                 pred= atliantic
                 r= +
                 ----------
              **********
           pred= border(subj,obj)
           subj=
              **********
              | I : 1
              | S : []
              | Q : wh
              | P : country(G)
              | F : ---------
                 det= wh
                 num= pl
                 pred= country
                 ----------
              **********
           ----------
        **********
     ----------
**********

     Translation Time = 150

     Created a semantic representation

wh(A,
     country(A) &
     a(B,
           population(A,B) &
           borders(A,atlantic) &
           exceeds(B,10--million)
        )
  ).



     Planning Time = 50

     Representation after Planning

wh(A,
     borders(A,atlantic) &
     { country(A) } &
     { population(A,B) &
       { exceeds(B,10--million) } }
  ).



     Execution Time = 234
```

```
france, netherlands, spain, west_germany, united_kingdom,
morocco, nigeria, south_africa, zaire, argentina, brazil,
canada, colombia, mexico, united_states, and venezuela

---------------------------------------------------------
SELECT MODE :
---------------------------------------------------------
Enter a query        :  e     Enter a query no.     :  n
Preprocess grammar   :  pg    Preprocess lexicon    : pl
Compile grammar      :  cg    Compile lexicon       : cl
Turn tracing on      :  t     Turn tracing off      : nt
Meta-Data on DB      :  md    Quit                  :  q
---------------------------------------------------------
Mode ? > n.
------------------------
    Number (1-23) ? > 22.
------------------------
[what,percentage,of,countries,border,each,ocean]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 1 to 3 of category np_head
edge 1 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category p
edge 4 to 4 of category e
edge 4 to 5 of category n
edge 3 to 5 of category pp
edge 3 to 5 of category ppadj
edge 1 to 5 of category np
edge 1 to 5 of category np
edge 5 to 5 of category e
edge 5 to 6 of category v
edge 6 to 6 of category e
edge 6 to 6 of category npe
edge 6 to 7 of category det
edge 7 to 7 of category e
edge 7 to 8 of category n
edge 6 to 8 of category np_head
edge 6 to 8 of category np
edge 5 to 8 of category vp
edge 5 to 8 of category sl

     Parse Time = 1833

     Created an F-structure

**********
| I : []
| S : [obj=B:location,subj=C:location]
| Q : q
| P : borders(C,B)
| F : ---------
   focus=
      **********
      | I : 1
      | S : [of-obj=F:G]
      | Q : wh
      | P : percentage(F,H)
      | F : ---------
         det= wh
         of=
            **********
            | I : []
            | S : [obj=F:country]
```

```
            | Q : []
            | P : []
            | F : ---------
               obj=
                  **********
                  | I : []
                  | S : []
                  | Q : []
                  | P : country(J)
                  | F : ---------
                     num= pl
                     pred= country
                     ----------
                  **********
               pcase= of
               ----------
            **********
         pred= percentage(of-obj)
         proportional= +
         ----------
      **********
   nonfinite= +
   obj=
      **********
      | I : []
      | S : []
      | Q : each
      | P : ocean(K)
      | F : ---------
         det= each
         num= sg
         pred= ocean
         ----------
      **********
   pred= border(subj,obj)
   q=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : []
      | F : ---------
         det= wh
         ----------
      **********
   subj=
      **********
      | I : 1
      | S : [of-obj=F:G]
      | Q : wh
      | P : percentage(F,H)
      | F : ---------
         det= wh
         of=
            **********
            | I : []
            | S : [obj=F:country]
            | Q : []
            | P : []
            | F : ---------
               obj=
                  **********
                  | I : []
                  | S : []
                  | Q : []
                  | P : country(J)
                  | F : ---------
```

```
                    num= pl
                    pred= country
                    ----------
                 **********
              pcase= of
              ----------
           **********
        pred= percentage(of-obj)
        proportional= +
        ----------
     **********
   ----------
**********

     Translation Time = 50

     Created a semantic representation

wh(A,
     each(B,
           ocean(B) &
           setof(C,
                   country(C)
                 ,D) &
           numberof(A,
                     pick(A,D) &
                     borders(A,B)
                  ,E) &
           card(F,D) &
           ratio(E,F,A)
        )
  ).


     Planning Time = 150

     Representation after Planning

wh(A,
     setof(B-C,
            setof(D,
                    country(D)
                  ,E) &
            card(F,E) &
            ocean(C) &
            numberof(B,
                      pick(B,E) &
                      { borders(B,C) }
                   ,G) &
            { ratio(G,F,B) }
          ,A)
  ).


     Execution Time = 2133

      2:arctic_ocean, 35:atlantic, 14:indian_ocean, 20:pacific,
      and 0:southern_ocean

     ------------------------------------------------------------
     SELECT MODE :
     ------------------------------------------------------------
     Enter a query        : e      Enter a query no.       : n
     Preprocess grammar   : pg     Preprocess lexicon      : pl
     Compile grammar      : cg     Compile lexicon         : cl
```

```
Turn tracing on      : t       Turn tracing off       : nt
Meta-Data on DB      : md      Quit                   :  q
----------------------------------------------------------
Mode ? > n.
------------------------
    Number (1-23) ? > 23.
------------------------
[what,countries,are,there,in,europe]

edge 1 to 2 of category det
edge 1 to 2 of category np
edge 2 to 2 of category e
edge 2 to 3 of category n
edge 1 to 3 of category np_head
edge 1 to 3 of category np
edge 3 to 3 of category e
edge 3 to 4 of category v
edge 3 to 4 of category v
edge 3 to 4 of category v
edge 4 to 4 of category e
edge 4 to 4 of category npe
edge 4 to 4 of category vpl
edge 4 to 5 of category n
edge 4 to 5 of category np
edge 5 to 5 of category e
edge 5 to 5 of category npe
edge 5 to 5 of category vpl
edge 3 to 5 of category sl
edge 5 to 6 of category p
edge 6 to 6 of category e
edge 6 to 7 of category pn
edge 6 to 7 of category np
edge 5 to 7 of category pp
edge 5 to 7 of category ppadj
edge 4 to 7 of category np
edge 7 to 7 of category e
edge 7 to 7 of category npe
edge 7 to 7 of category vpl
edge 3 to 7 of category sl

    Parse Time = 2433

    Created an F-structure

**********
| I : []
| S : [subj=B:C,vcomp=B:country]
| Q : q
| P : null(subj)
| F : ---------
   attributive= -
   aux= be
   focus=
      **********
      | I : 1
      | S : []
      | Q : wh
      | P : country(F)
      | F : ---------
         det= wh
         num= pl
         pred= country
         ----------
      **********
   num= pl
   pred= existential_be(vcomp,subj)
   q=
```

```
          **********
          | I : 1
          | S : []
          | Q : wh
          | P : []
          | F : ---------
             det= wh
             ----------
          **********
      subj=
          **********
          | I : []
          | S : []
          | Q : []
          | P : []
          | F : ---------
             adjuncts=
                **********
                | I : []
                | S : [obj=H:place,subj=I:place]
                | Q : []
                | P : in(I,H)
                | F : ---------
                   obj=
                      **********
                      | I : []
                      | S : domain=continent
                      | Q : []
                      | P : europe
                      | F : ---------
                         num= sg
                         pn_type= continent
                         pred= europe
                         ----------
                      **********
                   pcase= in
                   pred= in(subj,obj)
                   ----------
                **********

             form= there
             ----------
          **********
      tense= present
      vcomp=
          **********
          | I : 1
          | S : []
          | Q : wh
          | P : country(F)
          | F : ---------
             det= wh
             num= pl
             pred= country
             ----------
          **********
      ----------
**********

     Translation Time = 33

     Created a semantic representation

wh(A,
     in(A,europe) &
     country(A)
  ).
```

```
     Planning Time = 0

     Representation after Planning

wh(A,
     in(A,europe) &
     { country(A) }
  ).


     Execution Time = 167

       bulgaria, czechoslovakia, east_germany, hungary, poland,
       romania, denmark, finland, norway, sweden, albania, andorra,
       cyprus, greece, italy, malta, monaco, portugal, san_marino,
       spain, yugoslavia, austria, belgium, eire, france, iceland,
       liechtenstein, luxembourg, netherlands, switzerland,
       united_kingdom, and west_germany

      ----------------------------------------------------------
      SELECT MODE :
      ----------------------------------------------------------
      Enter a query        :  e     Enter a query no.      :  n
      Preprocess grammar   : pg     Preprocess lexicon     : pl
      Compile grammar      : cg     Compile lexicon        : cl
      Turn tracing on      :  t     Turn tracing off       : nt
      Meta-Data on DB      : md     Quit                   :  q
      ----------------------------------------------------------
      Mode ? > q.
      -------------------------
      Ok goodbye
      __________________________________________________________
[top_level.pl compiled 643.917 sec 401,692 bytes]
```

Appendix G

Prolog Implementation Code

All of the system code is listed here, except for the actual database files themselves. The basic structure of the database files (relations) is however outlined by the code and comments in the file which loads and provides access to the database relations 'database.pl'.

```
/* ************************************************************ */
/*      FILE    : README                                        */
/*      PURPOSE : instructions for use.                         */
/*      AUTHOR  : N. K. SIMPKINS.                               */
/*                ( Aston University, Dept. Computer Science,    */
/*                  Birmingham, B4 7ET )                        */
/* ************************************************************ */

/*
The system can be loaded by consulting or compiling the file
'top_level.pl' :

        'consult(top_level)' or '[top_level]'

when consult is used (as above) the system will be interpreted.
To compile the system type :

                'compile(top_level)'

The database used contains geographical data taken from the
Chat-80 system, this code (although modified) remains :

                Copyright 1986,
                Fernando C.N. Pereira and David H.D. Warren,
                (All Rights Reserved)

Each file which is derived from the Chat-80 database is
stated as being such in the file heading.

The other files are under :

                Copyright 1987,
                Neil K. Simpkins,
                (All Rights Reserved).

The system includes a sample grammar 'grammar' and two sample
lexicons 'lex_dom' the domain orientated lexicon and 'lex_gen'
the domain free lexicon.

After loading all the code the grammar(s) and lexicon(s) must
first be pre-preprocessed. To do this type 'hi' which starts
the system up and then follow the prompts. After pre-
processing the lexicon(s) and grammar(s) these may be compiled.
During compilation the pre-processed lexicon(s), grammar(s)
and the top-down link relation produced from the grammar are
written out to files. These file names are prompted for by
the system.

The system may be driectly instructed to process one of the
```

```
sample queries from the Chat-80 corpus by chosing the 'n'
option and then typing a query number. The numbered queries
in this corpus are listed below.

        -------------------------------------------
        List of queries in the corpus, taken from the
        CHAT-80 test file.
        -------------------------------------------

1) What rivers are there ?
2) Does afghanistan border china ?
3) What is the capital of upper_volta ?
4) Where is the largest country ?
5) Which countries are european ?
6) Which country's capital is London ?
7) Which is the largest african country ?
8) How large is the smallest american country ?
9) What is the ocean that borders african countries and
        that borders american countries ?
10) What are the capitals of the countries bordering the Baltic ?
11) Which countries are bordered by two seas ?
12) How many countries does the danube flow through ?
13) What is the total area of the countries south of the equator
        and not in Australasia ?
14) What is the average area of the countries in each continent ?
15) Is there more than one country in each continent ?
16) Is there some ocean that does not border any country ?
17) What are the countries from which a river flows into
        the Black-Sea ?
18) What are the continents no country in which contains more than
        two cities whose population exceeds 1 million ?
19) Which country bordering the Mediterranean borders a country that
        is bordered by a country whose population exceeds
        the population of India ?
20) Which countries have a population exceeding 10 million ?
21) Which countries with a population exceeding 10 million border
        the Atlantic
22) What percentage of countries border each ocean ?
23) What countries are there in Europe ?
*/

/* ************************************************************ */



/* ************************************************************ */
/*      FILE    : counters.pl                                   */
/*      PURPOSE : counters by assert/retract to the database.   */
/* ************************************************************ */

/* E X P O R T S */

:- module(counters, [
            new_counter/1,          % creates a new named counter.
            next_counter/2,         % get the next counter value.
            clear_counter/1         % deletes a counter.
            ]).

/* I M P O R T S */

% <none>.

/* P R E D I C A T E S */

/* new_counter(+Name), create a new global counter in the database
   with initial value '1'. */
```

```
new_counter(Name) :- assert(counter(Name, 1)).

        /* next_counter(+Name, -Value), get the next value of counter
           'Name' and increment the counters value by one. */

next_counter(Name, Value) :-
        retract(counter(Name, Value)), !, Next_value is Value + 1,
        assert(counter(Name, Next_value)).

        /* clear_counter(+Name), remove counter 'Name from the database. */

clear_counter(Name) :- retract(counter(Name, _)), !.

      /* ************************************************************     */



        /* ************************************************************ */
        /*      FILE    : database.pl                                    */
        /*      PURPOSE : database system top level.                     */
        /*                Adapted from code :                            */
        /*                 Copyright 1986,                               */
        /*                 Fernando C.N. Pereira and David H.D. Warren, */
        /*                 All Rights Reserved                           */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(database, [
                db/1    % db(+database_predicate).
                ]).

        /* I M P O R T S */

                /* LARGE BASIC RELATIONS */

        /* borders(?place, ?place).
           -----------------------
           either way round ie borders(a, b) & borders(b, a) are both
           defined. */

:- use_module('db_borders.rel', [ borders/2 ]).

        /* city(?city, ?country, ?population).
           -----------------------------------
           population is in thousands. */

:- use_module('db_city.rel',            [ city/3 ]).

        /* contains(?place, ?sub_place).
           ------------------------
           to any level ie contains(?Continent, +City) is defined
           via : 'contains(-Country, +City) ,
                  contains(-Continent, +Country)'. */

:- use_module('db_contains.rel',        [ contains/2 ]).

        /* country(?Country, ?Region, ?Latitude, ?Longitude,
           -------------------------------------------------
                        ?'Area/1000', ?'Area mod 1000',
                        ------------------------------
                        ?'Population/1000000',
                        ----------------------
                        ?'Population mod 1000000/1000', ?Capital, ?Currency).
                        --------------------------------------------------- */

:- use_module('db_country.rel',         [ country/10 ]).
```

```
        /* river(?Name, ?Places_list).
           --------------------------
           the list 'Places_list' is an ordered list of places through
           which the river flows (destination to source). */

:- use_module('db_river.rel',            [ river/2 ]).

        /* aggregate operators */

:- use_module(db_aggregate, [
                 aggreg/4
                 ]).

        /* P R E D I C A T E S */

db(african(C))                   :- african(C).
db(american(C))                  :- american(C).
db(area(A))                      :- area(A).     % check A is an area (units).
db(area(A, P))                   :- area(A, P).
db(asian(C))                     :- asian(C).
db(borders(P, P1))               :- borders(P, P1).
db(capital(C))                   :- capital(C).
db(capital(A, B))                :- capital(A, B).
db(circle_of_latitude(C))        :- circle_of_latitude(C).
db(city(C))                      :- city(C).
db(contains(P, Sub_p))           :- in(Sub_p, P).
db(continent(C))                 :- continent(C).
db(country(C))                   :- country(C).
db(drains(R, P))                 :- drains(R, P).
db(eastof(P, P1))                :- eastof(P, P1).
db(european(P))                  :- european(P).
db(exceeds(A, B))                :- exceeds(A, B).
db(flows(R, P))                  :- flows(R, P).
db(flows(R, P, Top))             :- flows(R, P; Top).
db(in(P, Sub_p))                 :- in(P, Sub_p).
db(latitude(L))                  :- latitude(L).
db(latitude(P, L))               :- latitude(P, L).
db(longitude(L))                 :- longitude(L).
db(longitude(P, L))              :- longitude(P, L).
db(northof(P, P1))               :- northof(P, P1).
db(ocean(O))                     :- ocean(O).
db(place(P))                     :- place(P).
db(population(P))                :- population(P).
db(population(P, Pop))           :- population(P, Pop).
db(region(R))                    :- region(R).
db(rises(R, P))                  :- rises(R, P).
db(river(R))                     :- river(R).
db(sea(S))                       :- sea(S).
db(seamass(S))                   :- seamass(S).
db(southof(P, P1))               :- southof(P, P1).
db(westof(P, P1))                :- westof(P, P1).
db(aggreg(P, T, S, Ans))         :- aggreg(P, T, S, Ans).

                /* DERIVED AND SMALL RELATIONS : */

continent(africa).      continent(america).      continent(antarctica).
continent(asia).        continent(australasia). continent(europe).

in_continent(scandinavia, europe).
in_continent(western_europe, europe).
in_continent(eastern_europe, europe).
in_continent(southern_europe, europe).
in_continent(north_america, america).
in_continent(central_america, america).
in_continent(caribbean, america).
in_continent(south_america, america).
```

```
in_continent(north_africa, africa).
in_continent(west_africa, africa).
in_continent(central_africa, africa).
in_continent(east_africa, africa).
in_continent(southern_africa, africa).
in_continent(middle_east,  asia).
in_continent(indian_subcontinent, asia).
in_continent(southeast_east, asia).
in_continent(far_east, asia).
in_continent(northern_asia, asia).

ocean(arctic_ocean).     ocean(atlantic).        ocean(indian_ocean).
ocean(pacific).          ocean(southern_ocean).

sea(baltic).             sea(black_sea).         sea(caspian).
sea(mediterranean).      sea(persian_gulf).      sea(red_sea).

exceeds(X -- U, Y -- U) :- !, X > Y.
exceeds(X1 -- U1, X2 -- U2) :-
        ratio(U1, U2, M1, M2),
        (X1 * M1) > (X2 * M2).

ratio(thousand,million,1,1000).        ratio(million,thousand,1000,1).
ratio(ksqmiles,sqmiles,1000,1).        ratio(sqmiles,ksqmiles,1,1000).

area(_x -- ksqmiles).

capital(C) :- capital(_x, C).

city(C) :- city(C, _, _).

country(C) :- country(C,_,_,_,_,_,_,_,_).

latitude(_x -- degrees).

longitude(_x -- degrees).

place(X) :- continent(X); region(X); seamass(X); country(X).

population(_x -- million).

population(_x -- thousand).

region(R) :- in_continent(R,_).

african(X) :- in(X,africa).

american(X) :- in(X,america).

asian(X) :- in(X,asia).

european(X) :- in(X,europe).

in(X,Y) :- var(X), nonvar(Y), !, contains(Y,X).
in(X,Y) :- in0(X,W), ( W=Y ; in(W,Y) ).

in0(X,Y) :- in_continent(X,Y).
in0(X,Y) :- city(X,Y,_).
in0(X,Y) :- country(X,Y,_,_,_,_,_,_,_).
in0(X,Y) :- flows(X,Y).

eastof(X1,X2) :- longitude(X1,L1), longitude(X2,L2), exceeds(L2,L1).

northof(X1,X2) :- latitude(X1,L1), latitude(X2,L2), exceeds(L1,L2).

southof(X1,X2) :- latitude(X1,L1), latitude(X2,L2), exceeds(L2,L1).
```

```
westof(X1,X2) :- longitude(X1,L1), longitude(X2,L2), exceeds(L1,L2).

circle_of_latitude(equator).
circle_of_latitude(tropic_of_cancer).
circle_of_latitude(tropic_of_capricorn).
circle_of_latitude(arctic_circle).
circle_of_latitude(antarctic_circle).

latitude(equator,0--degrees).
latitude(tropic_of_cancer,23--degrees).
latitude(tropic_of_capricorn,-23--degrees).
latitude(arctic_circle,67--degrees).
latitude(antarctic_circle,-67--degrees).
latitude(C,L -- degrees) :- country(C,_,L,_,_,_,_,_,_,_).

longitude(C,L -- degrees) :- country(C,_,_,L,_,_,_,_,_,_).

area(C,A -- ksqmiles) :- country(C,_,_,_,A,_,_,_,_,_).

area(Cont, Area -- Unit) :-
        continent(Cont),
        setof(C - A1, (in(C, Cont), country(C), area(C, A1)), C_areas),
        aggreg(total(_T), area, C_areas, Area -- Unit).

population(C, P -- thousand) :- city(C,_,P).

population(C, P -- million) :- country(C,_,_,_,_,_,P,_,_,_).

capital(Country,Capital) :- country(Country,_,_,_,_,_,_,_,Capital,_).

seamass(X) :- ocean(X).          seamass(X) :- sea(X).

river(R) :- river(R, _l).

rises(R,C) :- river(R,L), last(L,C).

drains(R,S) :- river(R,L), first(L,S).

flows(R,C) :- flows(R,C,_).

flows(R,C1,C2) :- river(R,L), links(L,C2,C1).

first([X|_],X).

last([X],X).                     last([_|L],X) :- last(L,X).

links([X1,X2|_],X1,X2).          links([_|L],X1,X2) :- links(L,X1,X2).

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : db_aggregate.pl                                */
        /*      PURPOSE : aggregate operators                            */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(db_aggregate, [
                aggreg/4
                ]).

        /* I M P O R T S */

:- use_module(library(aggregate), [
                aggregate/3
```

```
                ]).

:- use_module(library(findall), [
                findall/3
                ]).

:- use_module(fast_basics, [
                flength/2
                ]).

:- use_module(database, [
                db/1
                ]).

:- use_module(execute, [
                execute/1
                ]).

        /* P R E D I C A T E S */
%1
aggreg(largest(Entity), Type, Set, Largest) :- !,
        def(largest(Entity), Type, Set, Attribute, Preds),
        aggregate( max(Attribute, Entity),
                                execute(Preds), max(_, Largest) ), !.
%2
aggreg(smallest(Entity), Type, Set, Smallest) :- !,
        def(smallest(Entity), Type, Set, Attribute, Preds),
        aggregate( min(Attribute, Entity),
                                execute(Preds), min(_, Smallest) ), !.
%3
aggreg(average(Average), Type, Set, Average) :- !,
        def(average(Average), Type, Set, _attribute, Preds),
        execute(Preds), !.
%4
aggreg(total(Total), Type, Set, Total) :- !,
        def(total(Total), Type, Set, _attribute, Preds),
        execute(Preds), !.
%5
aggreg(numberof(_entity), _type, Set, Number) :- flength(Number, Set).


        % def(+Aggreg_pred(Ans), +Type, Set, Attribute, Preds).

        % the aggregate operator is largest, the entity type is country,
        % so the operator is taken to refer to area. Each country from
        % the set is taken in turn and the max. value of attribute
        % 'area' selected.
%1
def(largest(Entity), country, Countries, Area,
                                        pick(Entity, Countries) &
                                        area(Entity, Area -- _unit) ).
%2
def(largest(Continent), continent, Continents, Area,
                                        pick(Continent, Continents) &
                                        area(Continent, Area -- _unit) ).
%3
def(smallest(Entity), country, Countries, Area,
                                        pick(Entity, Countries) &
                                        area(Entity, Area -- _unit) ).
%4
def(smallest(Continent), continent, Continents, Area,
                                        pick(Continent, Continents) &
                                        area(Continent, Area -- _unit) ).
%5
def(average(Average--Units), area, [Country - (Area--Units)|Rest], Area,
                total(Total--Units, [Country - (Area--Units)|Rest]) &
                card(Length, [Country - (Area--Units)|Rest]) &
```

```
                Average is Total // Length).
%6
def(total(Total--Units), area, [Country - (Area--Units)|Rest], Area,
                total(Total--Units, [Country - (Area--Units)|Rest]) ).

        /* ********************************************************* */


        /* ********************************************************* */
        /*      FILE    : db_meta.pl                                 */
        /*      PURPOSE : produce meta-data about relations in a domain */
        /*                database.                                  */
        /* ********************************************************* */

        /* E X P O R T S */

:- module(db_meta, [
                produce_meta/1,         % produce meta data about a DB.
                db_solutions/3          % no. of solutions to a DB pred.
                ]).

        /* I M P O R T S */

:- use_module(fast_basics, [
                flength/2               % flength(?Length, ?List).
                ]).

:- use_module(make_static, [
                statisize/3
                ]).

:- use_module(graphic, [
                line/3
                ]).

        /* O P E R A T O R S */

:- op(  50,      fx,     [ relations ]).
:- op(  50,      fx,     [ mod ]).
:- op(  50,      fx,     [ access ]).
:- op(  40,     xfx,     [ type ]).
:- op(  40,      fx,     [ sols ]).
:- op( 100,      fx,     [ free, fixed]).

        /* D Y N A M I C S */

:- dynamic solutions/2, solved/1.

        /* solutions/2 holds the calculated cost of each predicate in the
           database given the state of each of the predicate's args
           (instantiated ie fixed or uninstantiated ie free). The different
           costs for predicate states are stored ('a1' is arg no. one etc) :

            P/1 [(a1-free), (a1-fixed)]
            P/2 [(a1-free,a2-free), (a1-fixed,a2-free),
                 (a1-free,a2-fixed), (a1-fixed,a2-fixed)]
            P/3 [(a1-free,a2-free,a3-free), (a1-fixed,a2-free,a3-free),
                 (a1-free,a2-fixed,a3-free), (a1-free,a2-free,a3-fixed),
                 (a1-fixed,a2-fixed,a3-free), (a1-fixed,a2-free,a3-fixed),
                 (a1-free,a2-fixed,a3-fixed), (a1-fixed,a2-fixed,a3-fixed)] */

        /* P R E D I C A T E S */

produce_meta(Database_spec) :-
        consult(Database_spec),         % the database specification.
        relations Rels,
```

```
        mod M,
        access P,
        line('-', 3, 36),
        format('~N~*|~w~w~n', [3, 'Producing Meta-Data on : ', M]),
        line('-', 3, 36), nl, nl,
        count_each_rel(Rels, M, P).

produce_meta :- format('~N~w~2n', ['Error in Database Specification']).


count_each_rel([], _, _) :- !,
        nl, line('-', 6, 30),
        format('~N~*|~w', [6, 'File for meta-data storage ? : ']),
        read(File), nl,
        line('-', 6, 30),
        statisize(db_meta, [db_meta:solutions(_,_)], File),
        format('~N~*|~w~2n', [6, 'Finished processing database']).

count_each_rel([Rel|_], Module, P) :- count_rel(Rel, Module, P), fail.

count_each_rel([_|Rels], Module, P) :- count_each_rel(Rels, Module, P).

 %1
count_rel(R/1 type [inf], _, _) :- !,
        assert(solutions(R/1, [sols integer, sols 1])),
        format('~N~*|~w~w~w~n', [6, 'Processed Relation : ',
                                        R/1, ' Has an Integer Argument']).
 %2
count_rel(R/2 type [inf, inf], _, _) :- !,
        assert(solutions(R/2, [sols integer, sols integer,
                                        sols integer, sols 1])),
        format('~N~*|~w~w~w~n', [6, 'Processed Relation : ',
                                        R/2, ' Has all Integer Arguments']).
 %3
count_rel(R/3 type [inf, inf, inf], _, _) :- !,
        assert(solutions(R/3, [sols integer, sols integer,
                              sols integer, sols integer, sols integer,
                                        sols integer, sols integer, sols 1])),
        format('~N~*|~w~w~w~n', [6, 'Processed Relation : ',
                                        R/3, ' Has all Integer Arguments']).
 %4
count_rel(R/1 type [symbol], Module, P) :- !,
        Relation =.. [R,A],
        Access =.. [P,Relation],
        bagof(A, Module:Access, Ans), flength(N, Ans),
        setof(A, Module:Access, Ans1), flength(N1, Ans1),
        Num is (N*10),
        N2 is (N1/N), N3 is (N2*10), Num_insta is integer(N3),
        assert(solutions(R/1, [sols Num, sols Num_insta])),
        format('~N~*|~w~w~n', [6, 'Processed Relation : ', R/1]),
        format('~*|~w~w~n', [10, 'Predicate Size : ', Num]),
        format('~*|~w~w~n', [10, 'Domain Size : ', Num_insta]).
 %5
count_rel(R/2 type [symbol, symbol], Module, P) :- !,
        Relation =.. [R,A,A1],
        Access =.. [P,Relation],
        bagof(A-A1, Module:Access, Ans2), flength(N, Ans2),
        setof(A-A1, Module:Access, Ans3), flength(Nsub, Ans3),
        setof(A, Module: A1^Access, Ansa), flength(Na, Ansa),
        setof(A1, Module:A^Access, Ansb), flength(Nb, Ansb),
        Num is (N*10),
        N1 is (N/Na),   N2 is (N1*10), Num_insta  is integer(N2),
        N3 is (N/Nb),   N4 is (N3*10), Num_instb  is integer(N4),
        N5 is (N/Nsub), N6 is (N5*10), Num_instab is integer(N6),
        assert(solutions(R/2, [sols Num, sols Num_insta,
                              sols Num_instb, sols Num_instab ])),
        format('~N~*|~w~w~n', [6, 'Processed Relation : ', R/2]),
```

```
        format('~*|~w~w~n', [10, 'Predicate Size : ', Num]),
        format('~*|~w~w~n', [10, 'Domain 1 : ', Num_insta]),
        format('~*|~w~w~n', [10, 'Domain 2 : ', Num_instb]).
%6
count_rel(R/2 type [symbol, inf], Module, P) :- !,
        Relation =.. [R,A,A1],
        Access =.. [P,Relation],
        bagof(A-A1, Module:Access, Ans), flength(N, Ans),
        setof(A-A1, Module:Access, Ans2), flength(Nsub, Ans2),
        Num is (N*10),
        N1 is (N/Nsub), N2 is (N1*10), Num_insta is integer(N2),
        assert(solutions(R/2, [sols integer, sols Num_insta,
                                              sols Num, sols 1])),
        format('~N~*|~w~w~n', [6, 'Processed Relation : ', R/2]),
        format('~*|~w~w~n', [10, 'Predicate Size : ', integer]),
        format('~*|~w~w~n', [10, 'Domain 1 : ', integer]),
        format('~*|~w~w~n', [10, 'Domain 2 : ', Num]).
%7
count_rel(R/3 type [symbol, symbol, symbol], Module, P) :- !,
        Relation =.. [R,A,A1,A2],
        Access =.. [P,Relation],
        bagof(A-A1-A2, Module:Access, Ans), flength(N, Ans),
        setof(A-A1-A2, Module:Access, Ans1), flength(Nsub, Ans1),
        setof(A, Module:A1^A2^Access, Ansa), flength(Numa, Ansa),
        setof(A1, Module:A^A2^Access, Ansb), flength(Numb, Ansb),
        setof(A2, Module:A^A1^Access, Ansc), flength(Numc, Ansc),
        Num is (N*10),
        N1 is (N/Numa), N2 is (N1*10), Num_insta is integer(N2),
        N3 is (N/Numb), N4 is (N3*10), Num_instb is integer(N4),
        N5 is (N/Numc), N6 is (N5*10), Num_instc is integer(N6),
        N7 is (N/(Numa*Numb)), N8 is (N7*10), Num_instab is integer(N8),
        N9 is (N/(Numa*Numc)), N10 is (N9*10), Num_instac is integer(N10),
        N11 is (N/(Numb*Numc)), N12 is (N11*10), Num_instbc is integer(N12),
        N13 is (Nsub/N), N14 is (N13*10), Num_instabc is integer(N14),
        assert(solutions(R/3, [sols Num, sols Num_insta, sols Num_instb,
                                        sols Num_instc, sols Num_instab,
                                        sols Num_instac, sols Num_instbc,
                                                   sols Num_instabc])),
        format('~N~*|~w~w~n', [6, 'Processed Relation : ', R/3]),
        format('~*|~w~w~n', [10, 'Predicate Size : ', Num]),
        format('~*|~w~w~n', [10, 'Domain 1 : ', Numa]),
        format('~*|~w~w~n', [10, 'Domain 2 : ', Numb]),
        format('~*|~w~w~n', [10, 'Domain 3 : ', Numc]).

%8
count_rel(R/4 type [symbol, symbol, symbol, symbol], Module, P) :- !,
        Relation =.. [R,A,A1,A2,A3],
        Access =.. [P,Relation],
        bagof(A-A1-A2, Module:Access, Ans), flength(N, Ans),
        setof(A-A1-A2, Module:Access, Ans1), flength(Nsub, Ans1),
        setof(A, Module:A1^A2^A3^Access, Ansa), flength(Numa, Ansa),
        setof(A1, Module:A^A2^A3^Access, Ansb), flength(Numb, Ansb),
        setof(A2, Module:A^A1^A3^Access, Ansc), flength(Numc, Ansc),
        setof(A3, Module:A^A1^A2^Access, Ansd), flength(Numd, Ansd),
        Num is (N*10),
        N1 is (N/Numa), N2 is (N1*10), Num_insta is integer(N2),
        N3 is (N/Numb), N4 is (N3*10), Num_instb is integer(N4),
        N5 is (N/Numc), N6 is (N5*10), Num_instc is integer(N6),
        N7 is (N/Numd), N8 is (N7*10), Num_instd is integer(N8),
        N9 is (N/(Numa*Numb)), N10 is (N9*10), Num_instab is integer(N10),
        N11 is (N/(Numa*Numc)), N12 is (N11*10), Num_instac is integer(N12),
        N13 is (N/(Numa*Numd)), N14 is (N13*10), Num_instad is integer(N14),
        N15 is (N/(Numb*Numc)), N16 is (N15*10), Num_instbc is integer(N16),
        N17 is (N/(Numb*Numd)), N18 is (N17*10), Num_instbd is integer(N18),
        N19 is (N/(Numc*Numd)), N20 is (N19*10), Num_instcd is integer(N20),
        N21 is (N/(Numa*(Numb*Numc))), N22 is (N21*10),
                                          Num_instabc is integer(N22),
```

```
        N23 is (N/(Numa*(Numc*Numd))), N24 is (N23*10),
                                              Num_instacd is integer(N24),
        N25 is (N/(Numb*(Numc*Numd))), N26 is (N25*10),
                                              Num_instbcd is integer(N26),
        N27 is (N/(Numa*(Numb*Numd))), N28 is (N27*10),
                                              Num_instabd is integer(N28),
        N29 is (N/Nsub), N30 is (N29*10), Num_instabcd is integer(N30),
        assert(solutions(R/4, [sols Num,
                               sols Num_insta, sols Num_instb,
                               sols Num_instc, sols Num_instd,
                               sols Num_instab, sols Num_instac,
                               sols Num_instad, sols Num_instbc,
                               sols Num_instbd, sols Num_instcd,
                               sols Num_instabc, sols Num_instabd,
                               sols Num_instacd, sols Num_instbcd,
                                              sols Num_instabcd])),
        format('~N~*|~w~w~n', [6, 'Processed Relation : ', R/3]),
        format('~*|~w~w~n', [10, 'Predicate Size : ', Num]),
        format('~*|~w~w~n', [10, 'Domain 1 : ', Numa]),
        format('~*|~w~w~n', [10, 'Domain 2 : ', Numb]),
        format('~*|~w~w~n', [10, 'Domain 3 : ', Numc]),
        format('~*|~w~w~n', [10, 'Domain 3 : ', Numd]).

%1
db_solutions(P/1, [fixed _], Num) :- !,          % one arg instantiated.
        solutions(P/1, [_,sols Number]),
        convert(fixed, Number, Num).
%2
db_solutions(P/1, [free _], Num) :- !,
        solutions(P/1, [sols Number,_]),
        convert(free, Number, Num).
%3      all free (2)
db_solutions(P/2, [free _, free _], Num) :- !,
        solutions(P/2, [sols Number,_,_,_]),
        convert(free, Number, Num).
%4      fixed a
db_solutions(P/2, [fixed _, free _], Num) :- !,
        solutions(P/2, [_,sols Number,_,_]),
        convert(free, Number, Num).
%5      fixed b
db_solutions(P/2, [free _, fixed _], Num) :- !,
        solutions(P/2, [_,_,sols Number,_]),
        convert(free, Number, Num).
%6      fixed a,b
db_solutions(P/2, [fixed _, fixed _], Num) :- !,
        solutions(P/2, [_,_,_,sols Number]),
        convert(fixed, Number, Num).
%7      all free (3)
db_solutions(P/3, [free _, free _, free _], Num) :- !,
        solutions(P/3, [sols Number,_,_,_,_,_,_,_]),
        convert(free, Number, Num).
%8      fixed a
db_solutions(P/3, [fixed _, free _, free _], Num) :- !,
        solutions(P/3, [_,sols Number,_,_,_,_,_,_]),
        convert(free, Number, Num).
%9      fixed b
db_solutions(P/3, [free _, fixed _, free _], Num) :- !,
        solutions(P/3, [_,_,sols Number,_,_,_,_,_]),
        convert(free, Number, Num).
%10     fixed c
db_solutions(P/3, [free _, free _, fixed _], Num) :- !,
        solutions(P/3, [_,_,_,sols Number,_,_,_,_]),
        convert(free, Number, Num).
%11     fixed a,b
db_solutions(P/3, [fixed _, fixed _, free _], Num) :- !,
        solutions(P/3, [_,_,_,_,sols Number,_,_,_]),
        convert(free, Number, Num).
```

```
 %12    fixed a,c
db_solutions(P/3, [fixed _, free _, fixed _], Num) :- !,
        solutions(P/3, [_,_,_,_,sols Number,_,_]),
        convert(free, Number, Num).
 %13    fixed b,c
db_solutions(P/3, [free _, fixed _, fixed _], Num) :- !,
        solutions(P/3, [_,_,_,_,_,sols Number,_]),
        convert(free, Number, Num).

db_solutions(P/3, [fixed _, fixed _, fixed _], Num) :- !,
        solutions(P/3, [_,_,_,_,_,_,sols Number]),
        convert(free, Number, Num).

db_solutions(P/4, [free _, free _, free _, free _], Num) :- !,
        solutions(P/4, [sols Number,_,_,_,_,_,_,_,_,_,_,_,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [free _, fixed _, free _, free _], Num) :- !,
        solutions(P/4, [_,_,sols Number,_,_,_,_,_,_,_,_,_,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [free _, free _, fixed _, free _], Num) :- !,
        solutions(P/4, [_,_,_,sols Number,_,_,_,_,_,_,_,_,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [free _, free _, free _, fixed _], Num) :- !,
        solutions(P/4, [_,_,_,_,sols Number,_,_,_,_,_,_,_,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [fixed _, fixed _, free _, free _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,sols Number,_,_,_,_,_,_,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [fixed _, free _, fixed _, free _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,_,sols Number,_,_,_,_,_,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [fixed _, free _, free _, fixed _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,_,_,sols Number,_,_,_,_,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [free _, fixed _, fixed _, free _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,_,_,_,sols Number,_,_,_,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [free _, fixed _, free _, fixed _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,_,_,_,_,sols Number,_,_,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [free _, free _, fixed _, fixed _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,_,_,_,_,_,sols Number,_,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [fixed _, fixed _, fixed _, free _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,_,_,_,_,_,_,sols Number,_,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [fixed _, fixed _, free _, fixed _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,_,_,_,_,_,_,_,sols Number,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [fixed _, free _, fixed _, fixed _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,_,_,_,_,_,_,_,_,sols Number,_,_]),
        convert(free, Number, Num).

db_solutions(P/4, [free _, fixed _, fixed _, fixed _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,sols Number,_]),
```

```
        convert(free, Number, Num).

db_solutions(P/4, [fixed _, fixed _, fixed _, fixed _], Num) :- !,
        solutions(P/4, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,sols Number]),
        convert(free, Number, Num).


convert(free, integer, 100000).        % infinity, well close.

convert(fixed, integer, 1).     convert(_, N, N).


        % pre-defined costs of predicates
 %1
solutions(aggreg/4, [sols integer, sols integer, sols integer,
                        sols 1, sols integer, sols integer, sols 1,
                        sols integer, sols 1, sols integer, sols 1,
                        sols 1, sols 1, sols 1, sols integer, sols 1]).
 %2
solutions(card/2, [sols integer, sols integer, sols 1, sols 1]).
 %3
solutions(ratio/3, [sols integer, sols integer, sols integer, sols integer,
                        sols 1, sols integer, sols integer, sols 1]).
 %4     pick/2 is a generator like setof/3.
solutions(pick/2, [sols integer, sols integer, sols gen, sols gen]).

        /* ********************************************************** */



        /* ********************************************************** */
        /*      FILE    : db_spec.pl                                  */
        /*      PURPOSE : aggregate operators                         */
        /* ********************************************************** */

        /* E X P O R T S */

:- module(db_spec, [
                relations/1,    % list of database relations.
                mod/1,          % name of the database module.
                access/1        % the predicate giving access to
                ]).             % database relations.

        /* P R E D I C A T E S */

relations [
                african/1 type [symbol],
                american/1 type [symbol],
                area/1 type [inf],
                area/2 type [symbol, inf],
                asian/1 type [symbol],
                borders/2 type [symbol, symbol],
                capital/1 type [symbol],
                capital/2 type [symbol, symbol],
                circle_of_latitude/1 type [symbol],
                city/1 type [symbol],
                contains/2 type [symbol, symbol],
                continent/1 type [symbol],
                country/1 type [symbol],
                drains/2 type [symbol, symbol],
                eastof/2 type [inf, inf],
                european/1 type [symbol],
                exceeds/2 type [inf, inf],
                flows/2 type [symbol, symbol],
                flows/3 type [symbol, symbol, symbol],
                in/2 type [symbol, symbol],
                latitude/1 type [inf],
```

```
              latitude/2 type [symbol, inf],
              longitude/1 type [inf],
              longitude/2 type [symbol, inf],
              northof/2 type [inf, inf],
              ocean/1 type [symbol],
              place/1 type [symbol],
              population/1 type [inf],
              population/2 type [symbol, inf],
              region/1 type [symbol],
              rises/2 type [symbol, symbol],
              river/1 type [symbol],
              sea/1 type [symbol],
              seamass/1 type [symbol],
              southof/2 type [inf, inf],
              westof/2 type [inf, inf],
              '='/2 type [inf, inf],
              '<'/2 type [inf, inf],
              '>'/2 type [inf, inf],
              length/2 type [inf, symbol]
              ].

mod database.     % the module containing the database.

access db.        % access to relations is via predicate db(+Relation_call).

        /* ********************************************************** */



        /* ********************************************************** */
        /*      FILE    : define_lfg.                                 */
        /*      PURPOSE : operators to define LFG syntax in Prolog.   */
        /*      NOTES   : since operators are not local to modules a  */
        /*                single file is used for the major operators. */
        /*                this also helps identify the required       */
        /*                priorities of operators relative to those at */
        /*                other levels of data structures.            */
        /* ********************************************************** */

        /* E X P O R T S */

:- module(define_lfg, [ /*none*/ ]).

        /* I M P O R T S */

        %       There is a problem with using angle brackets as LFG does
        % for 'pred' attributes. Quintus appears to have a bug so
        % that postfix operators ( eg ':-op(900, xf, [ > ])' ) cannot be
        % used in expressions such as (assuming '=' is a highier
        % precedence infix operator :
        %
        %               | ?- A = (fred > = john), display(A).
        %
        %               ** Syntax error:  **
        %               A= (fred> =
        %               ** here **
        %               john)
        %
        % instead we would have to use :
        %
        %               | ?- A = ((fred >) = john), display(A).
        %               =(>(fred),john)
        %               A = (fred>)=john
        %
        % That is to say the argument of a postfix operator MUST be put
```

```
        % in brackets.
        %
        %       Until such time as this is changed the notation will have
        % to be different to the LFG angled brackets, used to denote
        % subcategorised functions. I have used '>>' and a list) instead.
        % The operators that should be used would be something like :
        %
        %               :- op(  930,    xfx,    [ < ]).
        %               :- op(  930,     fx,    [ < ]).
        %               :- op(  980,    xfy,    [ > ]).
        %               :- op(  925,     xf,    [ > ]). POSTFIX OPERATOR.

        /* O P E R A T O R S */

        % LFG rule operator.
:- op(  1001,   xfy,    [ ---> ]).

        % operator for bounded nodes in grammar.
:- op(  982,     fx,    [ bnd ]).

        % operator for bounded nodes with linkage equation in grammar.
:- op(  982,     fx,    [ lnk ]).

        % operator for conjunction rules.
:- op(  1001,    fx,    [ conj ]).

        % sugar before LFG equations.
:- op(  981,     fy,    [ eqns ]).
:- op(  981,    xfy,    [ eqns ]).

        % sugar between LFG equations in grammar and lexicon.
:- op(  971,    xfy,    [ & ]).

        % feature, set or function values.
:- op(  960,    xfy,    [ = ]).

        % set value, these are 'attribute set_val_of value' in the input but
        % changed to 'attribute = set_value_of value' internally so that every
        % value in an f-structure is of form 'attribute = description'.
:- op(  960,    xfx,    [ set_val_of ]).

        % constraints, these are 'attribute c value' in the input but
        % changed to 'attribute = val_c value' internally so that every value
        % in an f-structure is of form 'attribute = description'.
:- op(  940,    xfx,    [ c ]).

        % 'not' is used in negative value negative existential and negative
        % constraint equations.
:- op(  941,     fx,    [ not ]).
:- op(  940,    xfx,    [ not ]).

        % designator prefixes governable functions which form the set of
        % designators.
:- op(  390,     fx,    [ designator ]).

        % '-' is used as sugar between a function and a feature and between
        % a function and sub-function eg 'subj - num', 'of - obj'.
:- op(  380,    xfy,    [ - ]).

:- op(  920,     fx,    [ complete ]).

        % 'up' is equivalent of an LFG upward pointing arrow
        % (immediate dominace variable).
:- op(  910,     fx,    [ up ]).

        % 'down' is equivalent of an LFG downward pointing arrow
:- op(  910,     fx,    [ down ]).
```

```
:- op( 900,     fx,     [ fs_is ]).

:- op( 900,     xfx,    [ fs_is ]).

        % prefix for a governable function (designator) in reformed rules
:- op( 370,     fx,     [ d ]).

        % sugar prefix to a controller superscript.
:- op( 900,     xfx,    [ super ]).

        % sugar prefix to a contro-ller/llee subscript.
:- op( 890,     xfx,    [ sub ]).

        % for the [+wh] subscript on contro-llers/llees in WH-Fronts.
:- op( 880,     fx,     [ + ]).

        % the Kleene-Star operator.
:- op( 982,     xfx,    [ * ]).

        % sugar between a word and its definitions.
:- op( 989,     xfy,    [ ~ ]).

        % conjunctions of word entries having different grammatical
        % categories.
:- op( 983,     xfy,    [ and ]).

        % disjuntions of word definitions having a single category.
:- op( 975,     xfy,    [ or ]).

        % prefix operator for pred attributes in lexicon (see not above).
:- op( 959,     xfx,    [ >> ]).

        % operators for predicates holding rules after pre-processing,
        % these are asserted into the parser module.

                % rules with first member of RHS a category.
:- op( 989,     xfx,    [ rules_which_cat ]).

                % rules with first member of RHS a word.
:- op( 989,     xfx,    [ rules_which_word ]).

        % this operator is inserted into rules labelled as conjunctions.
        % the operator prefixes the equations and causes the controller
        % and controllee lists to be unified with those from other
        % branches of the conjunction so that the controller and
        % controllee lists from a number of different conjunction branches
        % are matched with a single pair of controller and controllee lists
        % which are passed into a conjunction.
:- op( 969,     fx,     [ conj_cntrl ]).

:- op( 970,     xfx,    [ if ]).

:- op( 970,     fx,     [ if ]).

:- op( 950,     fx,     [ word_is ]).

        % all other LFG equation operators are also used
        % except for 'super' and 'sub'.
:- op( 890,     yfx,    [ ^ ]).                 % for f-structure info.

:- op( 891,     xfx,    [ glob ]).              % global pointers.

:- op( 890,     yfx,    [ ind ]).               % index in info. struct.

:- op( 942,     fx,     [ exists ]).            % exixtential constraints.
```

```
:- op(  910,    xfx,    [ : ]).                    % types in slots.

:- op(  941,     fx,    [ val_c ]).                % value constraint.

:- op(  941,     fx,    [ neg_c ]).                % neg. value constraints.

:- op(  941,     fx,    [ fs ]).                   % function value.

:- op(  940,     fx,    [ ptr ]).                  % pointer value.

:- op(  941,     fx,    [ set ]).                  % set value

:- op(  941,     fx,    [ atom ]).                 % atomic value (symbol).

:- op(  700,    xfx,    [ stype ]).                % for slots types.

:- op(  891,     fx,    [ moved ]).                % moved F-structure.

:- op(  892,    xfx,    [ temp ]).

:- op(  889,     fx,    [ var ]).

:- op(  500,    xfx,    [ root_of ]).    % for temp. controller syntax
                                         % during rule pre-processing.

:- op(  500,    xfy,    [ -- ]).         % units in the database.

:- op(  500,     fy,    [ gt ]).         % for semantics '>'.

:- op(  500,     fy,    [ lt ]).         % for semantics '<'.

:- op(   10,     fx,    [ strong, weak ]). % quantifier types.

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : delete_module.pl                              */
        /*      PURPOSE : delete the predicates in a given module to    */
        /*                reclaim space. Only the predicates can be     */
        /*                removed not the actual module declaration.    */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(delete_module, [
                del_mod/1
                ]).

        /* I M P O R T S */

 % none.

        /* P R E D I C A T E S */

        /* del_mod(+M).
           check that the name 'M' is a currently loaded module and is so
           delete all of its predicates. */
 %1
del_mod(Mod) :- current_module(Mod), delete_predicates(Mod).

        /* delete_predicates(+M).
           find the predicate name and arity of all predicates in module
           'M' and abolish these. */
 %1
delete_predicates(Mod):-
```

```
        current_predicate(Name,Mod:Pred),
        functor(Pred, Name, Arity),
        abolish(Mod:Name, Arity), fail.
 %2
delete_predicates(Mod) :-
        format('~N ~*| ~w ~w ~w ~2n', [5, 'Module : ', Mod, ' deleted.']),
        trimcore,
        garbage_collect, !.

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : desig.pl                                       */
        /*      PURPOSE : the governable designators set.                */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(desig, [
                designator/1
                ]).

        /* I M P O R T S */

 % none.

        /* P R E D I C A T E S */

designator subj .       designator obj .        designator vcomp .
designator ncomp .      designator poss .       designator of-obj .
designator of .         designator in-obj .     designator in .
designator thru-obj .   designator thru .

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : eval_eqns.pl                                   */
        /*      PURPOSE : evaluation of rule equations.                  */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(eval_eqns, [
                evaluate/4,
                evaluate/5
                ]).

        /* I M P O R T S */

:- use_module(counters, [
                next_counter/2
                ]).

:- use_module(fast_basics, [
                fappend/3,
                fdelete/3
                ]).

:- use_module(fs_basics, [
                member_fs/2,
                sort_fs/2
                ]).
```

```
:- use_module(fs_functs, [
                add_to_set/4,
                slot_funct/3,
                get_type/2
                ]).

:- use_module(unify, [
                merge/6,
                new_fs/6,
                no_constraints_ptr/2
                ]).

:- use_module(new_info, [
                empty_info/1,
                empty_info1/1
                ]).

:- use_module(wf_fs, [
                well_formed/4,
                contains/2
                ]).

        /* P R E D I C A T E S */

        /* evaluate(+Cat, +E, +Info_dwn, +Info_up, -Info_up1). */

%1      unify the controllees in a coordination.
evaluate(Cat, conj_cntrl E, Info^Clees glob Ptrs,
                        Info1^Clees glob Ptrs1, Info2^Clees2 glob Ptrs2) :- !,
        fappend(Ptrs, Ptrs1, Ptrs3),
        express(Cat, E, _, Info, Info1, Info2, Ptrs3, Ptrs2,
                                        Clees, Clees3, [], Nclees1, _),
        fappend(Nclees1, Clees3, Clees2), !.
%2      if there is a controller it must be consumed at this node.
evaluate(Cat, bound & E, Info^Clees glob Ptrs,
                Info1^Clees1 glob Ptrs1, Info2^Clees2 glob Ptrs2) :- !,
        fappend(Ptrs, Ptrs1, Ptrs3),
        express(Cat, E, _, Info, Info1, Info2, Ptrs3, Ptrs2,
                                        Clees, [], [], Nclees1, _),
        fappend(Nclees1, Clees1, Clees2).
%3
evaluate(Cat, E, Info^Clees glob Ptrs,
                Info1^Clees1 glob Ptrs1, Info2^Clees2 glob Ptrs2) :- !,
        fappend(Clees, Clees1, Clees3),
        fappend(Ptrs, Ptrs1, Ptrs3),
        express(Cat, E, _, Info, Info1, Info2, Ptrs3, Ptrs2,
                                        Clees3, Clees4, [], Nclees1, _),
        fappend(Nclees1, Clees4, Clees2).
%4
evaluate(Cat, bound & E, Info^[/*no clees*/] glob Ptrs,
                Info1^Clees1 glob Ptrs1, Info2^Clees2 glob Ptrs2) :- !,
        fappend(Ptrs, Ptrs1, Ptrs3),
        express(Cat, E, _, Info, Info1, Info2, Ptrs3, Ptrs2,
                                        [], Clees3, [], Nclees1, _),
        fappend(Clees3, Clees1, Clees4),
        fappend(Nclees1, Clees4, Clees2).
%5
evaluate(Cat, linkage & E, Info^[Clee|Clees] glob Ptrs,
                Info1^Clees1 glob Ptrs1, Info2^[Clee|Clees2] glob Ptrs2) :-
        evaluate(Cat, bound & E, Info^Clees glob Ptrs,
                        Info1^Clees1 glob Ptrs1, Info2^Clees2 glob Ptrs2).
%6
evaluate(Cat, E, Info^Clees glob Ptrs,
                Info1^Clees1 glob Ptrs1, Info2^Clees2 glob Ptrs2) :- !,
        fappend(Clees, Clees1, Clees3),
        fappend(Ptrs, Ptrs1, Ptrs3),
        express(Cat, E, _, Info, Info1, Info2, Ptrs3, Ptrs2,
```

```
                                        Clees3, Clees4, [], Ncleesl, _),
        fappend(Ncleesl, Clees4, Clees2).

        /* evaluate(+Cat, +E, +Info_dwn, -Info_up). */
 %1
evaluate(Cat, E, Info, Info2) :-
        empty_infol(Infol),
        evaluate(Cat, E, Info, Infol, Info2), !.

 %1
express(Cat, Eqn & Eqns, Next, Info, Info_up, Info_out,
                Ptrs, Ptrsl, Clees, Cleesl, Nclees, Ncleesl, Cmplt) :- !,
        eval(Eqn, Cat, Next, Info, Infol, Info_up, Info_upl,
                Ptrs, Ptrs2, Clees, Clees2, Nclees, Nclees2, Cmplt),
        express(Cat, Eqns, Next, Infol, Info_upl, Info_out,
                Ptrs2, Ptrsl, Clees2, Cleesl, Nclees2, Ncleesl, Cmplt).
 %2
express(Cat, Eqn, Next, Info, Info_up, Info_out,
                Ptrs, Ptrsl, Clees, Cleesl, Nclees, Ncleesl, Cmplt) :- !,
        eval(Eqn, Cat, Next, Info, Infol, Info_up, Info_out,
                Ptrs, Ptrs2, Clees, Cleesl, Nclees, Ncleesl, Cmplt),
        completed(Cmplt, Ptrs2, Ptrsl, Infol, Next).

 %1
completed(no, Ptrs, Ptrs, Info, Info) :- !.
 %2
completed(complt, Ptrs, Ptrsl, Info, Infol) :-
        well_formed(Info, Infol, Ptrs, Ptrsl).


        % equation evaluation.
        % eval(+Eqn, +Node_cat, -^Inferior_fsl, +Inferior_fs,
        %               -Inferior_fsl, +Superior_fs, -Superior_fsl,
        %                       +Pointers, -Pointersl,
        %                       +Controllees, -Controlleesl, ?Completed).

        % here it is assumed that pcase is present in the subordinate
 %1       F-structure that has been created so far.
eval((up(down pcase)) = down, _, Next, Dwn^Dfs, Dwnl^Dfsl,
        I ind Slots^Quant^Sem^Var^Upfs,
                I ind Slotsl^Quant^Sem^Var^Upfsl,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Nclees, _) :- !,
        member_fs(pcase = atom Val, Dfs),
        get_type(Dwn^Dfs, Type),
        slot_funct(Val:Type, Slots, Slotsl),
        replace(Val = fs Infol, Val = fs Next, Ptrs, Ptrs2, Upfs, Upfsl),
        merge_or_empty(Infol, Dwn^Dfs, Ptrs2, Ptrsl, down, Dwnl^Dfsl).
 %2
eval((down F-Fl) = (up d F2), _, _, I ind Slots^Quant^Sem^Var^Fs,
                I ind Slots^Quant^Sem^Var^Fsl,
                Iu ind Slotsu^Quantu^Semu^Varu^Fsu,
                Iu ind Slotsu^Quantu^Semu^Varu^Fsul,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Nclees, _) :- !,
        replace(F-Fl = fs Info, Fl = fs Final, Ptrs, Ptrs2, Fs, Fsl),
        replace(F2 = fs Infol, F2 = fs Final, Ptrs2, Ptrs3, Fsu, Fsul),
        merge_or_empty(Info, Infol, Ptrs3, Ptrsl, up, Final).
        % here it is assumed that Ft is present in the subordinate
 %3       F-structure that has been created so far.
eval((up Ft) = (down Ft), _, _, Dwn^Dfs, Dwn^Dfs,
                Info_up^Upfs, Info_up^Upfsl, Ptrs, Ptrsl,
                                        Clees, Clees, Nclees, Nclees, _) :- !,
        member_fs(Ft = atom Val, Dfs),
        new_fs([Ft = atom Val], Upfs, Ptrs, up, Upfsl, Ptrsl).

        % a controller can either belong to this domain, in which case
        % the controllee must have been found by now (BU), or it can
        % belong to another node (note equations are ordered so that a
```

```
        % controller for this domain comes first) in which case it must be
        % passed to that domain.
 %4
eval((up q) = controller(Cat, Script, Cl_info), Cat, _,
                Info, Info, I ind Slots^_^Sem^Var^Up_fs,
                I ind Slots^q^Sem^Var^Up_fsl, Ptrs, Ptrsl,
                                Clees, Cleesl, Nclees, Nclees, _) :- !,
        get_controllee(Cl_info, Script, Clees, Cleesl),
        new_fs([q = fs Cl_info], Up_fs, Ptrs, up, Up_fsl, Ptrsl).

        % a function prefixed by 'd' is a designator, labelled as such by
        % the grammar pre-processor.
 %5
eval((up d F) = controller(Cat, Script, Cl_info), Cat, _, Info, Info,
                I ind Slots^Quant^Sem^Var^Up_fs,
                I ind Slotsl^Quant^Sem^Var^Up_fsl, Ptrs, Ptrsl,
                                Clees, Cleesl, Nclees, Nclees, _) :- !,
        get_controllee(Cl_info, Script, Clees, Cleesl),
        get_type(Cl_info, D),
        slot_funct(F:D, Slots, Slotsl),
        new_fs([F = fs Cl_info], Up_fs, Ptrs, up, Up_fsl, Ptrsl).
 %6
eval((up F) = controller(Cat, Script, Cl_info), Cat, _,
                Info, Info, Rest_info^Up_fs, Rest_info^Up_fsl,
                        Ptrs, Ptrsl, Clees, Cleesl, Nclees, Nclees, _) :- !,
        get_controllee(Cl_info, Script, Clees, Cleesl),
        new_fs([F = fs Cl_info], Up_fs, Ptrs, up, Up_fsl, Ptrsl).

        % a controllee may be found here or may be found later ie
 %7     further up the tree, but not past a bounded node.
eval((up d F) = controllee sub Script, _, _, Info, Info,
                I ind Slots^Quant^Sem^Var^Fs,
                        I ind Slotsl^Quant^Sem^Var^Fsl,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Ncleesl, _) :- !,
        get_controller(C_info, Script, Nclees, Ncleesl),
        get_type(C_info, D),
        slot_funct(F:D, Slots, Slotsl),
        new_fs([F = fs C_info], Fs, Ptrs, up, Fsl, Ptrsl).
 %8
eval((up F) = controllee sub Script, _, _, Info, Info,
                Info_up^Fs, Info_up^Fsl, Ptrs, Ptrsl,
                                Clees, Clees, Nclees, Ncleesl, _) :- !,
        get_controller(C_info, Script, Nclees, Ncleesl),
        new_fs([F = fs C_info], Fs, Ptrs, up, Fsl, Ptrsl).
 %9
eval((down F-Fl) = controllee sub Script, _, _,
        I ind Slots^Quant^Sem^Var^Fs,
                I ind Slotsl^Quant^Sem^Var^Fsl, Info_up, Info_up,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Ncleesl, _) :- !,
        get_controller(C_info, Script, Nclees, Ncleesl),
        get_type(C_info, D),
        slot_funct(F:D, Slots, Slotsl),
        empty_info(In ind Sn stype _^Quantn^Semn^Varn^_),
        sort_fs([Fl = fs C_info, pcase = atom F], Ffl_fs),
        new_fs([F = fs In ind Sn stype [Fl = _:_]^Quantn^
                        Semn^Varn^Ffl_fs], Fs, Ptrs, up, Fsl, Ptrsl).
 %10
eval((down d F) = controllee sub Script, _, _,
        I ind Slots^Quant^Sem^Var^Fs,
                I ind Slotsl^Quant^Sem^Var^Fsl, Info_up, Info_up,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Ncleesl, _) :- !,
        get_controller(C_info, Script, Nclees, Ncleesl),
        get_type(C_info, D),
        slot_funct(F:D, Slots, Slotsl),
        new_fs([F = fs C_info], Fs, Ptrs, up, Fsl, Ptrsl).
 %11
eval((down F) = controllee sub Script, _, _, Info^Fs, Info^Fsl, Info_up,
```

```
                Info_up, Ptrs, Ptrsl, Clees, Clees, Nclees, Ncleesl, _) :- !,
        get_controller(C_info, Script, Nclees, Ncleesl),
        new_fs([F = fs C_info], Fs, Ptrs, up, Fsl, Ptrsl).
 %12
eval(up = controllee sub Script, _, _, _, _, _, C_info,
                        Ptrs, Ptrs, Clees, Clees, Nclees, Ncleesl, _) :- !,
        get_controller(C_info, Script, Nclees, Ncleesl).
 %13    controller belongs to this domain.
eval(down = controller(Cat, Script, Next), Cat, Next, _,
        I ind Slots^Quant^Sem^Var^Fs, Info_up, Info_up,
                        Ptrs, Ptrs, Clees, Cleesl, Nclees, Nclees, _) :- !,
        next_counter(index, I),
        get_controllee(I ind Slots^Quant^Sem^Var^Fs, Script, Clees, Cleesl).
 %14
eval(up = down, _, _, Info, Info_upl, Info_up, Info_upl,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Nclees, _) :- !,
        merge(Info, Info_up, up, Ptrs, Info_upl, Ptrsl).

 %15    add a set member to a moved fs.
eval(down set_val_of (up F), _, Next, Info, Info,
                moved Infoup temp Infos, moved Infoup temp Infosl,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Nclees, _) :- !,
        empty_info(In ind Slots^Quant^Sem^Var^_),
        merge(Infos, In ind Slots^Quant^Sem^Var^[F = set [Next]],
                                                _, Ptrs, Infosl, Ptrsl).
 %16
eval(down set_val_of (up F), _, Next, Info, Info,
                Rest_info^Up_fs, Rest_info^Up_fsl,
                        Ptrs, Ptrs, Clees, Clees, Nclees, Nclees, _) :- !,
        add_to_set(F, var Next, Up_fs, Up_fsl).
 %17
eval((up d F - F1) = down, _, Next, moved Val temp Infos,
                moved Val temp Infosl, In ind Slots^Quant^Sem^Var^Fs,
                In ind Slotsl^Quant^Sem^Var^Fsl,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Nclees, _) :- !,
        get_type(Infos, D),
        slot_funct(F:D, Slots, Slotsl),
        replace(F - F1 = fs Infol, F1 = fs Next, Ptrs, Ptrs2, Fs, Fsl),
        merge_or_empty(Infol, Infos, Ptrs2, Ptrsl, up, Infosl).
 %18
eval((up F - F1) = down, _, Next, moved Val temp Infos,
                moved Val temp Infosl, Info_up^Fs, Info_up^Fsl,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Nclees, _) :- !,
        replace(F - F1 = fs Infol, Ptrs, Ptrs2, F1 = fs Next, Fs, Fsl),
        merge_or_empty(Infol, Infos, Ptrs2, Ptrsl, up, Infosl).
 %19
eval((up d F - F1) = down, _, Next, Info_dwn, Info_dwnl,
        In ind Slots^Quant^Sem^Var^Fs,
                In ind Slotsl^Quant^Sem^Var^Fsl,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Nclees, _) :- !,
        get_type(Info_dwn, D),
        slot_funct(F:D, Slots, Slotsl),
        replace(F - F1 = fs Infol, F1 = fs Next, Ptrs, Ptrs2, Fs, Fsl),
        merge_or_empty(Infol, Info_dwn, Ptrs2, Ptrsl, up, Info_dwnl).
 %20
eval((up F - F1) = down, _, Next, Info_dwn, Info_dwnl,
                Info_up^Fs, Info_up^Fsl,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Nclees, _) :- !,
        replace(F - F1 = fs Infol, F1 = fs Next, Ptrs, Ptrs2, Fs, Fsl),
        merge_or_empty(Infol, Info_dwn, Ptrs2, Ptrsl, down, Info_dwnl).
 %21
eval((up d F - F1) = down, _, Next, Info_dwn, Info_dwnl,
        In ind Slots^Quant^Sem^Var^Fs,
                In ind Slotsl^Quant^Sem^Var^Fsl,
                        Ptrs, Ptrsl, Clees, Clees, Nclees, Nclees, _) :- !,
        get_type(Info_dwn, D),
        slot_funct(F:D, Slots, Slotsl),
```

```
        replace(F - F1 = fs Info2, F1 = fs Next, Ptrs, Ptrs2, Fs, Fs1),
        merge_or_empty(Info2, Info_dwn, Ptrs2, Ptrs1, down, Info_dwn1).
 %22
eval((up d F) = down, _, Next, moved Val temp Infos,
                moved Val temp Infos1, I ind Slots^Quant^Sem^Var^Fs,
                I ind Slots1^Quant^Sem^Var^Fs1,
                        Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, _) :- !,
        get_type(Infos, D),
        slot_funct(F:D, Slots, Slots1),
        replace(F = fs Info1, F = fs Next, Ptrs, Ptrs2, Fs, Fs1),
        merge_or_empty(Info1, Infos, Ptrs2, Ptrs1, up, Infos1).
 %23
eval((up F) = down, _, Next, moved Val temp Infos,
                moved Val temp Infos1, Info_up^Fs, Info_up^Fs1,
                        Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, _) :- !,
        replace(F = fs Info1, F = fs Next, Ptrs, Ptrs2, Fs, Fs1),
        merge_or_empty(Info1, Infos, Ptrs2, Ptrs1, up, Infos1).
 %24
eval((up d F) = down, _, Next, Info, Info1,
        I ind Slots^Quant^Sem^Var^Upfs,
                I ind Slots1^Quant^Sem^Var^Upfs1,
                         Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, _) :- !,
        get_type(Info, D),
        slot_funct(F:D, Slots, Slots1),
        replace(F = fs Info2, F = fs Next, Ptrs, Ptrs2, Upfs, Upfs1),
        merge_or_empty(Info2, Info, Ptrs2, Ptrs1, down, Info1).
 %25
eval((up F) = down, _, Next, Info, Info1, Info_up^Fs, Info_up^Fs1,
                        Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, _) :- !,
        replace(F = fs Info2, F = fs Next, Ptrs, Ptrs2, Fs, Fs1),
        merge_or_empty(Info2, Info, Ptrs2, Ptrs1, down, Info1).
 %26
eval(down fs_is Next, _, Next, [] ind Slots^Quant^Sem^Var^Fs,
                I ind Slots^Quant^Sem^Var^Fs, Info_up, Info_up,
                        Ptrs, Ptrs, Clees, Clees, Nclees, Nclees, _) :- !,
        next_counter(index, I).
 %27
eval(down fs_is Next, _, Next, Info_dwn, Info_dwn, Info_up, Info_up,
                        Ptrs, Ptrs, Clees, Clees, Nclees, Nclees, _) :- !.
 %28
eval((up d F) fs_is moved V temp Infon, _, _, Info, Info,
        I ind Slots^Quant^Sem^Var^Fs,
                I ind Slots1^Quant^Sem^Var^Fs1,
                         Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, _) :- !,
        slot_funct(F:_, Slots, Slots1),
        replace(F = fs Info1, F = fs moved V temp Infon, Ptrs, Ptrs1, Fs, Fs1),
        merge_or_empty(Info1, moved V temp Infon).
 %29
eval(up Eqn_info, _, _, Info, Info, moved Infom temp Infos,
                moved Infom temp Infos1,
                         Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, _) :- !,
        merge(Infos, Eqn_info, up, Ptrs, Infos1, Ptrs1).
 %30
eval(down Eqn_info, _, _, Info, Info, moved Infom temp Infos,
                moved Infom temp Infos1,
                        Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, _) :- !,
        merge(Infos, Eqn_info, down, Ptrs, Infos1, Ptrs1).
 %31
eval(up Eqn_info, _, _, Info, Info, Up_info, Up_info1,
                        Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, _) :- !,
        merge(Eqn_info, Up_info, up, Ptrs, Up_info1, Ptrs1).
 %32
eval(down Eqn_info, _, _, Info, Info1, Up_info, Up_info,
                        Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, _) :- !,
        merge(Eqn_info, Info, down, Ptrs, Info1, Ptrs1).
 %33
eval((up F - F1) = down, _, Next, Info_dwn, Info_dwn1,
```

```
                Info_up^Fs, Info_up^Fs1,
                        Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, _) :- !,
        replace(F - F1 = fs Info2, F1 = fs Next, Ptrs, Ptrs2, Fs, Fs1),
        merge_or_empty(Info2, Info_dwn, Ptrs2, Ptrs1, down, Info_dwn1).
%34
eval(complete (up d F) = down, _, Next, moved Val temp Infos,
                moved Val temp Infos1, Index ind Slots^Quant^Sem^Var^Fs,
                Index ind Slots1^Quant^Sem^Var^Fs1,
                Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, complt) :- !,
        get_type(Infos, D),
        slot_funct(F:D, Slots, Slots1),
        replace(F = fs Info1, F = fs Next, Ptrs, Ptrs2, Fs, Fs1),
        merge_or_empty(Info1, Infos, Ptrs2, Ptrs1, down, Infos1).
%35
eval(complete (up F) = down, _, Next, moved Val temp Infos,
                moved Val temp Infos1, Info_up^Fs, Info_up^Fs1,
                Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, complt) :- !,
        replace(F = fs Info1,  F = fs Next, Ptrs, Ptrs2, Fs, Fs1),
        merge_or_empty(Info1, Infos, Ptrs2, Ptrs1, down, Infos1).
%36
eval(complete (up d F) = down, _, Next, Info_dwn, Info_dwn1,
        Index ind Slots^Quant^Sem^Var^Fs,
                Index ind Slots1^Quant^Sem^Var^Fs1,
                Ptrs, Ptrs1, Clees, Clees, Nclees, Nclees, complt) :- !,
        get_type(Info_dwn, D),
        slot_funct(F:D, Slots, Slots1),
        replace(F = fs Info1,  F = fs Next, Ptrs, Ptrs2, Fs, Fs1),
        merge_or_empty(Info1, Info_dwn, Ptrs2, Ptrs1, down, Info_dwn1).
%37
eval(complete (up F) = down, _, Next, Info_dwn, Info_dwn1,
                Info_up^Fs, Info_up^Fs1, Ptrs, Ptrs1,
                                Clees, Clees, Nclees, Nclees, complt) :- !,
        replace(F = fs Info1,  F = fs Next, Ptrs, Ptrs2, Fs, Fs1),
        merge_or_empty(Info1, Info_dwn, Ptrs2, Ptrs1, down, Info_dwn1).
%38
eval(controller(Sc, Cl_info), _, _, Info_dwn, Info_dwn, Info_up,
                Info_up, Ptrs, Ptrs, Clees, Clees1, Nclees, Nclees, _) :- !,
        get_controllee(Cl_info, Sc, Clees, Clees1).


%1
merge_or_empty(empty, _) :- !.
%2
merge_or_empty(moved Val temp Info, moved Val temp Info) :- !.
%3
merge_or_empty(Info, moved _ temp Info) :- !.
%3
merge_or_empty(Info, Info).


%1
merge_or_empty(empty, empty, Ptrs, Ptrs, _, _) :- !.
%2
merge_or_empty(empty, Info^Fs, Ptrs, Ptrs, down, Info^Fs) :- !,
        no_constraints_ptr(Fs, Ptrs).
%3
merge_or_empty(empty, Info, Ptrs, Ptrs, up, Info) :- !.
%4
merge_or_empty(Info^Fs, empty, Ptrs, Ptrs, down, Info^Fs) :- !,
        no_constraints_ptr(Fs, Ptrs).
%5
merge_or_empty(Info, empty, Ptrs, Ptrs, up, Info) :- !.
%6
merge_or_empty(Info, Info1, Ptrs, Ptrs1, Arrow, Info2) :-
        merge(Info, Info1, Arrow, Ptrs, Info2, Ptrs1).
```

```
        /* replace(+Old_pair, +New_pair, +Ptrs, -Ptrs, +Fs, -Fs1).
           replace the value of Old_pair with that of New_pair in F-Structure
           'Fs' to produce 'Fs1', if the pair does not exist then
           return a value of 'empty' for the attribute in 'Old_pair'. */
 %1
replace(F = fs empty, F = fs Final,
                        Ptrs, Ptrs, [/*fs*/], [F = fs Final]) :- !.
 %2
replace(F = fs Current, F = fs Final, Ptrs, Ptrs1,
                        [F = fs ptr N|Rest], [F = fs ptr N|Rest]) :- !,
        replace_pointer_val(N, Ptrs, Current, Final, Ptrs1).
 %3
replace(F = fs Current, F = fs Final, Ptrs, Ptrs,
                        [F = fs Current|Rest], [F = fs Final|Rest]) :- !.
 %4
replace(F = Val, F = fs Final, Ptrs, Ptrs1,
                                [F1 = Val1|Rest], [F1 = Val1|Rest1]) :-
        F @> F1, !,
        replace(F = Val, F = fs Final, Ptrs, Ptrs1, Rest, Rest1).
 %5
replace(F = fs empty, F = fs Final, Ptrs, Ptrs, Fs, [F = fs Final|Fs]) :- !.
 %6
replace(F - d F1 = fs empty, d F1 = fs Final, Ptrs, Ptrs, [/*fs*/],
                [F = fs [] ind part stype [F1 = _:_]^
                                        Quant^Sem^Var^[F1 = fs Final]]) :- !,
        empty_info(_^Quant^Sem^Var^_).
 %7
replace(F - F1 = fs empty, F1 = fs Final, Ptrs, Ptrs, [/*fs*/],
                [F = fs [] ind part stype [F1 = _:_]^
                                        Quant^Sem^Var^[F1 = fs Final]]) :- !,
        empty_info(_^Quant^Sem^Var^_).
 %8
replace(F - d F1 = fs Current, d F1 = fs Final, Ptrs, Ptrs1,
                [F = fs Ind ind Slots^Quant^Sem^Var^Fs|Rest],
                [F = fs Ind ind Slots1^Quant^Sem^Var^Fs1|Rest]) :- !,
        slot_funct(F1:_, Slots, Slots1),
        replace(F1 = fs Current, F1 = fs Final, Ptrs, Ptrs1, Fs, Fs1).
 %9
replace(F - F1 = fs Current, F1 = fs Final, Ptrs, Ptrs1,
                [F = fs Slots^Quant^Sem^Var^Fs|Rest],
                        [F = fs Slots^Quant^Sem^Var^Fs1|Rest]) :- !,
        replace(F1 = fs Current, F1 = fs Final, Ptrs, Ptrs1, Fs, Fs1).
 %10
replace(F - F1 = fs Current, F1 = fs Final, Ptrs, Ptrs1,
                                        [F2 = Val|Rest], [F2 = Val|Rest1]) :-
        F @> F2, !,
        replace(F - F1 = fs Current, F1 = fs Final, Ptrs, Ptrs1, Rest, Rest1).
 %10
replace(F - d F1 = fs empty, d F1 = fs Final, Ptrs, Ptrs, Fs,
                [F = fs [] ind full stype [F1 = _:_]^
                                Quant^Sem^Var^[F1 = fs Final]|Fs]) :- !,
        empty_info(_slots^Quant^Sem^Var^_fs).
 %11
replace(F - F1 = fs empty, F1 = fs Final, Ptrs, Ptrs, Fs,
                [F = fs [] ind full stype Slots^
                                Quant^Sem^Var^[F1 = fs Final]|Fs]) :- !,
        empty_info(_i ind _ stype Slots^Quant^Sem^Var^_fs).

 %1
replace_pointer_val(N, Ptrs, Current, Final, [N = fs Final|Ptrs1]) :-
        fdelete(N = fs Current, Ptrs, Ptrs1).

        % when a controller is found (in its own domain) this procedure is
        % called to get the corresponding controllee moved information. The
        % controllee must have been found at this point (BU).
 %1
get_controllee(Info1, Script, [controllee(Script,
```

```
                                moved Info2 temp Infos)|Rest], Rest):- !,
        merge(Infos, Info1, down, [], Info2, _).
 %2
get_controllee(Info_full, Script,
                        [controllee(Script,Info)|Clees], Clees) :- !,
        full(Info, Info_full).

        % when a controllee is found this procedure is called to match
        % the controllee with a controller. The controller may not yet have
        % been reached in with case
        % controllees must be nil, add controller to list.
 %1     controllers must be nil, add controllee to list.
get_controller(moved Info temp O1, Script, Clees,
                [controllee(Script, moved Info temp O1)|Clees]) :- !.

 %1
full(I ind St stype Slots^Quant^[]^Var^Fs,
                        I ind St stype Slots^Quant^[]^Var^Fs) :- !.
 %2
full(I ind _ stype Slots^Quant^Sem^Var^Fs,
                        I ind full stype Slots^Quant^Sem^Var^Fs) :- !.
 %3
full(moved Info temp Infos, moved Info temp Infos3) :- !,
        full(Infos, Info2), merge(Infos, Info2, up, [], Infos3, _).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : execute.pl                                    */
        /*      PURPOSE : execute predicate logic queries against the   */
        /*                Prolog database.                              */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(execute, [
                execute/1
                ]).

        /* I M P O R T S */

:- use_module(database, [
                db/1            % database relations.
                ]).

:- use_module(fast_basics, [
                flength/2
                ]).

:- use_module(set_of1, [
                set_of1/3       % will suceed if no solutions.
                ]).

:- use_module(traces, [
                trace_on/0
                ]).

        /* P R E D I C A T E S */
 %1
execute( wh(Set, setof(Var, Preds, Set) ) ) :- !,
        set_of1(Var, execute(Preds), Set), answer(Set).
 %2
execute( wh(Var, Preds) ) :- !,
        set_of1(Var, execute(Preds), Ans),      % may have free variables.
        answer(Ans).
```

```
 %3
execute( y_n(Condition) ) :-
         ( execute(Condition), !, answer(y) | answer(n), ! ).
 %4
execute( setof(V, Preds, L) ) :-  !, set_of1(V, execute(Preds), L).
 %5
execute( length(Set, Int) ) :- !, flength(Int, Set).
 %6
execute( numberof(Var, Preds) ) :- !,
        set_of1(Var, execute(Preds), Set),
        flength(Num, Set), answer([Num]).
 %7
execute( numberof(Var, Preds, Num) ) :- !,
         ( var(Num), !, set_of1(Var, execute(Preds), Set)
        |
           setof(Var, execute(Preds), Set) ), flength(Num, Set).
 %8
execute( Pred & Preds ) :- !, execute(Pred), execute(Preds).
 %9
execute( \+ { Preds } ) :- !, \+ execute(Preds).
 %10
execute( \+ Pred ) :- !, \+ execute(Pred).
 %11
execute( pick(_, []) ):- !, fail.
 %12
execute( pick(Element, [Element|_]) ).
 %13
execute( pick(Element, [_|Rest]) ) :- !, execute( pick(Element, Rest) ).
 %14
execute( total(0 -- _, []) ).
 %15
execute( total(T -- Unit, [_ - (Val -- Unit)|R]) ) :- !,
        execute( total(Tr -- Unit, R) ), T is Tr + Val.
 %16
execute( card(L, List) ) :- !, flength(L, List).
 %17
execute(ratio(N, N1, Ans) ) :- !, Ans is (N * 100) // N1.
 %18
execute( A is B // C ) :- !, A is B // C.
 %19
execute( A > B ) :- !, A > B.
 %20
execute( A < B ) :- !, A < B.
 %21
execute([ Preds ]) :- !, execute(Preds), !.
 %22
execute( Pred ) :- !, db(Pred).

 %1
answer(n) :- report_time, format('~N~*|~w~2n', [7, 'no, not true']).
 %2
answer(y) :- report_time, format('~N~*|~w~2n', [7, 'indeed']).
 %3
answer([A - B]) :-
        report_time, format('~N~*|~w~w~w~w~2n', [7, 'found : ', A, ':', B]).
 %4
answer([A]) :-
        report_time, format('~N~*|~w~w~2n', [7, 'found : ', A]).
 %5
answer([H - B|R]) :-
        report_time, format('~N~*|~w~w~w~w', [7, H, ':', B, ', ']),
        answer_rest(R).
 %6
answer([H|R]) :-
        report_time, format('~N~*|~w~w', [7, H, ', ']), answer_rest(R).

 %1
```

```
answer_rest([Last - B]) :-
        line_position(user, Chars),
        ( Chars < 55 | format('~n~*|', 7) ),
        format('~w~w~w~w~2n', ['and ', Last, ':', B]).
 %2
answer_rest([Last]) :-
        line_position(user, Chars),
        ( Chars < 60 | format('~n~*|', 7) ),
        format('~w~w~2n', ['and ',Last]).
 %3
answer_rest([H - B|R]) :-
        line_position(user, Chars),
        ( Chars < 55 | format('~n~*|', 7) ),
        format('~w~w~w~w', [H, ':', B, ', ']), answer_rest(R).
 %4
answer_rest([H|R]) :-
        line_position(user, Chars),
        ( Chars < 60 | format('~n~*|', 7) ), format('~w~w', [H, ', ']),
        answer_rest(R).

 %1
report_time :-
        trace_on, !, statistics(runtime, [_, T]),
        format('~N~n~*|~w~w~2n', [5, 'Execution Time = ', T]).
 %2
report_time :- !.

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : extend.pl                                     */
        /*      PURPOSE : extends active edge by complete edge.         */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(extend, [
                extend/4,
                end_or_new/4,
                new/1
                ]).

        /* I M P O R T S */

:- use_module(eval_eqns, [
                evaluate/4,
                evaluate/5
                ]).

:- use_module(parser, [
                reached_target/3
                ]).

:- use_module(more_basics, [
                clause_or_assert/1
                ]).

        /* P R E D I C A T E S */

        /* extend(+Active_edge, +Complete_edge, +Target_edge, -Target_flag).
           extend the base completed edge upwards :

           Take the complete edge from the base (most recent first) and :

           a)   combine it with the active edges which end where this edge
```

```
                starts, this may produce new active and/or complete edges.
           b)   see if it either invokes (completes or is the first part
                of the right-hand-side of a grammar rule) new rules which
                can be hypothesised as starting where this edge starts.

           the base's (complete) edge matches the first part of the
           remainder of an active edge, a new active edge is created which
           spans both edges and has the active edge's remainder minus the
           first part completed by the base's edge. */

        % extend Kleene-Star rules (edges) with added numbers for max.
        % number of repetitions note that only two consecutive Kleene
        % stars are currently allowed in rules.

 %1       extend(A* N A* N1 Rem) by(A) ->
        %                 active(A* N-1 A* N1 Rem) <& active(A* N1 Rem)> or
        %                 active(A* N1-1 Rem) <& active(Rem)>.
extend([Sv1,Sv,Head, (Cata * N if Ea, Cata * N1 if Eb, Rem),Info1],
                                   [Sv,Ev,Cata,Info], _t, _tf) :- !,
         ( evaluate(Cata, Ea, Info, Info1, Info2) ->
           end_kl(N, [Sv1,Ev,Head,
                         (Cata * N if Ea, Cata * N1 if Eb, Rem),Info2])
        |
           evaluate(Cata, Eb, Info, Info1, Info2), !,
           end_kl(N1, [Sv1,Ev,Head,(Cata * N1 if Eb, Rem),Info2]) ).

 %2       extend(A* N A* Rem) by(A) ->
        %                 active(A* N-1 A* Rem) <& active(A* Rem)> or
        %                 active(A* Rem) <& active(Rem)>.
extend([Sv1,Sv,Head, (Cata * N if Ea, Cata * if Eb, Rem),Info1],
                                   [Sv,Ev,Cata,Info], _t, _tf) :- !,
         ( evaluate(Cata, Ea, Info, Info1, Info2) ->
           end_kl(N, [Sv1,Ev,Head,(Cata * N if Ea, Cata * if Eb, Rem),Info2])
        |
           evaluate(Cata, Eb, Info, Info1, Info2), !,
           new([Sv1,Ev,Head,(Cata * if Eb ,Rem),Info2]) ).

 %3       extend(A* A* N1 Rem) by(A) ->
        %                 active(A* A* N1 Rem) <& active(A* N1 Rem)> or
        %                 active(A* N1-1 Rem) <& active(Rem)>.
extend([Sv1,Sv,Head, (Cata * if Ea, Cata * N1 if Eb, Rem),Info1],
                                   [Sv,Ev,Cata,Info], _t, _tf) :- !,
         ( evaluate(Cata, Ea, Info, Info1, Info2) ->
           new([Sv1,Ev,Head,(Cata * if Ea, Cata * N1 if Eb, Rem),Info2])
        |
           evaluate(Cata, Eb, Info, Info1, Info2), !,
           end_kl(N1, [Sv1,Ev,Head,(Cata * N1 if Eb ,Rem),Info2]) ).

 %4       extend(A* N B* N1 Rem) by(A) ->
        %                 active(A* N-1 B* N1 Rem) <& active(B* N1 Rem)>.
extend([Sv1,Sv,Head, (Cata * N if Ea, Catb * N1 if Eb, Rem),Info1],
                                   [Sv,Ev,Cata,Info], _t, _tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
        end_kl(N, [Sv1,Ev,Head,(Cata * N if Ea, Catb * N1 if Eb, Rem),Info2]).

 %5       extend(A* N B* Rem) by(A) ->
        %                 active(A* N-1 B* Rem) <& active(B* Rem)>.
extend([Sv1,Sv,Head, (Cata * N if Ea, Catb * if Eb, Rem),Info1],
                                   [Sv,Ev,Cata,Info], _t, _tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
        end_kl(N, [Sv1,Ev,Head,(Cata * N if Ea, Catb * if Eb , Rem),Info2]).

 %6       extend(A* B* N1 Rem) by(A) ->
        %                 active(A* B* N1 Rem) <& active(B* N1 Rem)>.
extend([Sv1,Sv,Head, (Cata * if Ea, Catb * N1 if Eb, Rem),Info1],
                                   [Sv,Ev,Cata,Info], _t, _tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
```

```
        new([Svl,Ev,Head,(Cata * if Ea, Catb * N1 if Eb , Rem),Info2]).

%7        extend(A* N B* N1 Rem) by(B) -> active(B* N1 Rem) <& active(Rem)>.
extend([Svl,Sv,Head, (_ * _ if _, Catb * N1 if Eb, Rem),Info1],
                                    [Sv,Ev,Catb,Info], _t, _tf) :- !,
        evaluate(Catb, Eb, Info, Info1, Info2), !,
        new([Svl,Ev,Head,(Catb * N1 if Eb, Rem),Info2]).

%8        extend(A* N B* Rem) by(B) -> active(B* Rem) <& active(Rem)>.
extend([Svl,Sv,Head, (_ * _ if _, Catb * if Eb, Rem),Info1],
                                    [Sv,Ev,Catb,Info], _t, _tf) :- !,
        evaluate(Catb, Eb, Info, Info1, Info2),
        new([Svl,Ev,Head,(Catb * if Eb, Rem),Info2]).

%9        extend(A* B* N1 Rem) by(B) -> active(B* N1 Rem) <& active(Rem)>.
extend([Svl,Sv,Head, (_ * _ if _, Catb * N1 if Eb, Rem),Info1],
                                    [Sv,Ev,Catb,Info], _t, _tf) :- !,
        evaluate(Catb, Eb, Info, Info1, Info2),
        new([Svl,Ev,Head,(Catb * N1 if Eb, Rem),Info2]).

%10       extend(A* N B* N1 Rem) by(C) -> extend(Rem) by(C).
extend([Svl,Sv,Head,(_ * _ if _, _ * _ if _, Rem),Info1],
                                    [Sv,Ev,Catc,Info], T, Tf) :- !,
        extend([Svl,Sv,Head,Rem,Info1], [Sv,Ev,Catc,Info], T, Tf).

%11       extend(A* N B* Rem) by(C) -> extend(Rem) by(C).
extend([Svl,Sv,Head,(_ * _ if _, _ * if _, Rem),Info1],
                                    [Sv,Ev,Catc,Info], T, Tf) :- !,
        extend([Svl,Sv,Head,Rem,Info1], [Sv,Ev,Catc,Info], T, Tf).

%12       extend(A* B* N1 Rem) by(C) -> extend(Rem) by(C).
extend([Svl,Sv,Head,(_ * if _, _ * _ if _, Rem),Info1],
                                    [Sv,Ev,Catc,Info], T, Tf) :- !,
        extend([Svl,Sv,Head,Rem,Info1], [Sv,Ev,Catc,Info], T, Tf).

%13       extend(A* N A* N1) by(A) -> active(A* N-1 A* N1) & complete_edge or
        %                      -> active(A* N1) & complete_edge.
        % Either A must be found (atleast once) as the complete edge
        % (with neither A) has all ready been generated.
extend([Svl,Sv,Head,(Cata * N if Ea, Cata * N1 if Eb),Info1],
                                    [Sv,Ev,Cata,Info], T, Tf) :- !,
          ( evaluate(Cata, Ea, Info, Info1, Info2) ->
             ( reached_target([Svl,Ev,Head,Info2], T, Tf), !
             |
              end_kl(N, [Svl,Ev,Head,(Cata * N if Ea, Cata * N1 if Eb),Info2])
             )
          |
            evaluate(Cata, Eb, Info, Info1, Info2), !,
             ( reached_target([Svl,Ev,Head,Info2], T, Tf), !
             |
               end_kl(N1, [Svl,Ev,Head,(Cata * N1 if Eb),Info2])
             )
          ).

%14       extend(A* N A*) by(A) -> active(A* N-1 A*) & complete_edge or
        %                      -> active(A*) & complete_edge.
        % Either A must be found (atleast once) as the complete edge
        % (with neither A) has all ready been generated.
extend([Svl,Sv,Head,(Cata * N if Ea, Cata * if Eb),Info1],
                                    [Sv,Ev,Cata,Info], T, Tf) :- !,
          ( evaluate(Cata, Ea, Info, Info1, Info2) ->
             ( reached_target([Svl,Ev,Head,Info2], T, Tf), !
             |
              end_kl(N, [Svl,Ev,Head,(Cata * N if Ea, Cata * if Eb),Info2])
             )
          |
            evaluate(Cata, Eb, Info, Info1, Info2), !,
```

```
            ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
            |
              new([Sv1,Ev,Head,(Cata * if Eb),Info2])
            )
          ).

%15       extend(A* A* N1) by(A) -> active(A* A* N1) & complete_edge or
        %                         -> active(A* N1-1) & complete_edge.
        % Either A must be found (atleast once) as the complete edge
        % (with neither A) has all ready been generated.
extend([Sv1,Sv,Head,(Cata * if Ea, Cata * N1 if Eb),Info1],
                                      [Sv,Ev,Cata,Info], T, Tf) :- !,
          ( evaluate(Cata, Ea, Info, Info1, Info2) ->
            ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
            |
              new([Sv1,Ev,Head,(Cata * if Ea, Cata * N1 if Eb),Info2])
            )
          |
            evaluate(Cata, Eb, Info, Info1, Info2), !,
            ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
            |
              end_kl(N1, [Sv1,Ev,Head,(Cata * N1 if Eb),Info2])
            )
          ).

%16       extend(A* N B* N1) by(A) -> active(A* N-1 B* N1) & complete_edge.
        % either A or B must be found (atleast once) as the complete edge
        % (with neither A or B) has all ready been generated.
extend([Sv1,Sv,Head,(Cata * N if Ea, Catb * N1 if Eb),Info1],
                                      [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
         ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
        |
           Nn is N - 1, !, Nn > 0,
           new([Sv1,Ev,Head,(Cata * Nn if Ea,.Catb * N1 if Eb),Info2]) ).

%17       extend(A* N B*) by(A) -> active(A* N-1 B*) & complete_edge.
        % either A or B must be found (atleast once) as the complete edge
        % (with neither A or B) has all ready been generated.
extend([Sv1,Sv,Head,(Cata * N if Ea, Catb * if Eb),Info1],
                                      [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
         ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
        |
           end_kl(N, [Sv1,Ev,Head,(Cata * N if Ea, Catb * if Eb),Info2]) ).

%18       extend(A* B* N1) by(A) -> active(A* B* N1) & complete_edge.
        % either A or B must be found (atleast once) as the complete edge
        % (with neither A or B) has all ready been generated.
extend([Sv1,Sv,Head,(Cata * if Ea, Catb * N1 if Eb),Info1],
                                      [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
         ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
        |
           new([Sv1,Ev,Head,(Cata * if Ea, Catb * N1 if Eb),Info2]) ).

%19       extend(A* N B* N1) by(B) -> active(B* N1-1) & complete_edge.
        % either A or B must be found atleast once as the complete edge
        % (with neither A or B) has all ready been generated.
extend([Sv1,Sv,Head,(Cata * _ if Ea, Catb * N1 if Eb),Info1],
                                      [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
         ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
        |
           end_kl(N1, [Sv1,Ev,Head,(Catb * N1 if Eb),Info2]) ).

%20       extend(A* N B*) by(B) -> active(B*) & complete_edge.
```

```
        % either A or B must be found atleast once as the complete edge
        % (with neither A or B) has all ready been generated.
extend([Sv1,Sv,Head,(Cata * _ if Ea, Catb * if Eb),Info1],
                                        [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
         ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
        |
           new([Sv1,Ev,Head,(Catb * if Eb),Info2]) ).

 %21      extend(A* B* N1) by(B) -> active(B* N1-1) & complete_edge.
        % either A or B must be found atleast once as the complete edge
        % (with neither A or B) has all ready been generated.
extend([Sv1,Sv,Head,(Cata * if Ea, Catb * N1 if Eb),Info1],
                                        [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
         ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
        |
           end_kl(N1, [Sv1,Ev,Head,(Catb * N1 if Eb),Info2]) ).

 %22      extend(A* N Rem) by(A) -> active(A* N-1 Rem) <& active(Rem)>.
extend([Sv1,Sv,Head,(Cata * N if Ea, Rem),Info1],
                                        [Sv,Ev,Cata,Info], _t, _tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
        end_kl(N, [Sv1,Ev,Head,(Cata * N if Ea, Rem),Info2]).

 %23      extend(A* N Rem) by(B) -> extend(Rem) by (B).
extend([Sv1,Sv,Head,(_ * _ if _, Rem),Info1], [Sv,Ev,Catb,Info], T, Tf) :- !,
        extend([Sv1,Sv,Head,Rem,Info1], [Sv,Ev,Catb,Info], T, Tf).

 %24      extend(A* N) by(A) -> active(A* N-1) & complete_edge.
extend([Sv1,Sv,Head,(Cata * N if Ea),Info1], [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
         ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
        |
           end_kl(N, [Sv1,Ev,Head,(Cata * N if Ea),Info2]) ).

 %25      extend(A B* N C* N1) by(A) -> complete_edge & active(B* N C* N1).
        % note that the new Kleene sequence must be used, ie atleast one
        % B or C must be found.
extend([Sv1,Sv,Head, (Cata if Ea, Catb * N if Eb, Catc * N1 if Ec),Info1],
                                        [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
         ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
        |
           new([Sv1,Ev,Head,(Catb * N if Eb, Catc * N1 if Ec),Info2]) ).

 %26      extend(A B* N C*) by(A) -> complete_edge & active(B* N C*).
        % note that the new Kleene sequence must be used, ie atleast one
        % B or C must be found.
extend([Sv1,Sv,Head, (Cata if Ea, Catb * N if Eb, Catc * if Ec),Info1],
                                        [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
         ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
        |
           new([Sv1,Ev,Head,(Catb * N if Eb, Catc * if Ec),Info2]) ).

 %27      extend(A B* C* N1) by(A) -> complete_edge & active(B* C* N1).
        % note that the new Kleene sequence must be used, ie atleast one
        % B or C must be found.
extend([Sv1,Sv,Head, (Cata if Ea, Catb * if Eb, Catc * N1 if Ec),Info1],
                                        [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2), !,
         ( reached_target([Sv1,Ev,Head,Info2], T, Tf), !
        |
           new([Sv1,Ev,Head,(Catb * if Eb, Catc * N1 if Ec),Info2]) ).

 %28      extend(A B* N) by(A) -> complete_edge & active(B* N).
```

```
        % note that the new Kleene sequence must be used, ie atleast one
        % B must be found.
extend([Svl,Sv,Head,(Cata if Ea, Catb * N if Eb),Infol],
                                        [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Infol, Info2), !,
         ( reached_target([Svl,Ev,Head,Info2], T, Tf), !
        |
           new([Svl,Ev,Head,(Catb * N if Eb),Info2]) ).

 %30       extend(A* A* Rem) by(A) -> active(A* A* Rem) <& active(A* Rem)> or
        %                          -> active(A* Rem) <& active(Rem)>.
extend([Svl,Sv,Head, (Cata * if Ea, Cata * if Eb, Rem),Infol],
                                        [Sv,Ev,Cata,Info], _t, _tf) :- !,
         ( evaluate(Cata, Ea, Info, Infol, Info2) ->
           new([Svl,Ev,Head,(Cata * if Ea, Cata * if Eb , Rem),Info2])
        |
           evaluate(Cata, Eb, Info, Infol, Info2), !,
           new([Svl,Ev,Head,(Cata * if Eb , Rem),Info2]) ).

 %31       extend(A* B* Rem) by(A) -> active(A* B* Rem) <& active(B* Rem)>.
extend([Svl,Sv,Head, (Cata * if Ea, Catb * if Eb, Rem),Infol],
                                        [Sv,Ev,Cata,Info], _, _) :- !,
        evaluate(Cata, Ea, Info, Infol, Info2),
        new([Svl,Ev,Head,(Cata * if Ea, Catb * if Eb , Rem),Info2]).

 %32       extend(A* B* Rem) by(B) -> active(B* Rem) <& active(Rem)>.
extend([Svl,Sv,Head, (_ * if _, Catb * if Eb, Rem),Infol],
                                        [Sv,Ev,Catb,Info], _, _) :- !,
        evaluate(Catb, Eb, Info, Infol, Info2),
        new([Svl,Ev,Head,(Catb * if Eb, Rem),Info2]).

 %33       extend(A* B* Rem) by(C) -> extend(Rem) by(C).
extend([Svl,Sv,Head,(_ * if _, _ * if _, Rem),Infol],
                                        [Sv,Ev,Catc,Info], T, Tf) :- !,
        extend([Svl,Sv,Head,Rem,Infol], [Sv,Ev,Catc,Info], T, Tf).

 %34       extend(A* A*) by(A) -> active(A* A*) & complete_edge or
        %                     -> active(A*) & complete_edge.
        % Either A or B must be found (atleast once) as the complete edge
        % (with neither A or B) has all ready been generated.
extend([Svl,Sv,Head,(Cata * if Ea, Cata * if Eb),Infol],
                                        [Sv,Ev,Cata,Info], T, Tf) :- !,
         ( evaluate(Cata, Ea, Info, Infol, Info2) ->
           end_or_new([Svl,Ev,Head,Info2], T, Tf,
                        [Svl,Ev,Head,(Cata * if Ea, Cata * if Eb),Info2])
        |
           evaluate(Cata, Eb, Info, Infol, Info2), !,
           end_or_new([Svl,Ev,Head,Info2], T, Tf,
                        [Svl,Ev,Head,(Cata * if Eb),Info2]) ).

 %35       extend(A* B*) by(A) -> active(A* B*) & complete_edge. Only
        % either A or B must be found (atleast once) as the complete edge
        % (with neither A or B) has all ready been generated.
extend([Svl,Sv,Head,(Cata * if Ea, Catb * if Eb),Infol],
                                        [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Infol, Info2),
        end_or_new([Svl,Ev,Head,Info2], T, Tf,
                        [Svl,Ev,Head,(Cata * if Ea, Catb * if Eb),Info2]).

 %36       extend(A* B*) by(B) -> active(B*) & complete_edge.
        % either A or B must be found atleast once as the complete edge
        % (with neither A or B) has all ready been generated.
extend([Svl,Sv,Head,(Cata * if Ea, Catb * if Eb),Infol],
                                        [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Infol, Info2),
        end_or_new([Svl,Ev,Head,Info2], T, Tf,
                        [Svl,Ev,Head,(Cata * if Ea, Catb * if Eb),Info2]).
```

```
 %37      extend(A* Rem) by(A) -> active(A* Rem) <& active(Rem)>.
extend([Sv1,Sv,Head,(Cata * if Ea, Rem),Info1],
                                       [Sv,Ev,Cata,Info], _t, _tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2),
        new([Sv1,Ev,Head,(Cata * if Ea, Rem),Info2]).

 %38      extend(A* Rem) by(B) -> extend(Rem) by (B).
extend([Sv1,Sv,Head,(_ * if _, Rem),Info1], [Sv,Ev,Catb,Info], T, Tf) :- !,
        extend([Sv1,Sv,Head,Rem,Info1], [Sv,Ev,Catb,Info], T, Tf).

 %39      extend(A*) by(A) -> active(A*) & complete_edge.
        % either A or B must be found atleast once as the complete edge
        % (with neither A or B) has all ready been generated.
extend([Sv1,Sv,Head,(Cata * if Ea),Info1], [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2),
        end_or_new([Sv1,Ev,Head,Info2], T, Tf,
                                [Sv1,Ev,Head,(Cata * if Ea),Info2]).

 %40      extend(A B* C*) by(A) -> complete_edge & active(B* C*).
        % note that the new Kleene sequence must be used, ie atleast one
        % B or C must be found.
extend([Sv1,Sv,Head, (Cata if Ea, Catb * if Eb, Catc * if Ec),Info1],
                                       [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2),
        end_or_new([Sv1,Ev,Head,Info2], T, Tf,
                [Sv1,Ev,Head,(Catb * if Eb, Catc * if Ec),Info2]).

 %41      extend(A B*) by(A) -> complete_edge & active(B*).
        % note that the new Kleene sequence must be used, ie atleast one
        % B must be found.
extend([Sv1,Sv,Head,(Cata if Ea, Catb * if Eb),Info1],
                                       [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2),
        end_or_new([Sv1,Ev,Head,Info2], T, Tf,
                                [Sv1,Ev,Head,(Catb * if Eb),Info2]).

 %42      extend(A Rem) by(A) -> active(Rem).
extend([Sv1,Sv,Head,(Cata if Ea, Rem),Info1],
                                       [Sv,Ev,Cata,Info], _t, _tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2),
        new([Sv1,Ev,Head,Rem,Info2]).

 %43      extend(word_is W Rem) by(W) -> active(Rem).
extend([Sv1,Sv,Head,(word_is Word , Rem),Info1], [Sv,Ev,Word], _t, _tf) :- !,
        new([Sv1,Ev,Head,Rem,Info1]).

        /* the base's edge completes an active edge, finish if this is the
           target edge, otherwise add the newly completed edge to the front
           of the base so that it will be used next (LIFO). */

 %44      extend(A) by(A) -> complete_edge.
extend([Sv1,Sv,Head,Cata if Ea,Info1], [Sv,Ev,Cata,Info], T, Tf) :- !,
        evaluate(Cata, Ea, Info, Info1, Info2),
        end_or_new([Sv1,Ev,Head,Info2], T, Tf, []).

 %45     extend(word_is W) by(W) -> complete_edge.
extend([Sv1,Sv,Head,word_is Word,Info1], [Sv,Ev,Word], T, Tf) :- !,
         end_or_new([Sv1,Ev,Head,Info1], T, Tf, []).

        % either produced the target edge or produce a new active edge.
 %1
end_or_new(Complete_edge, Target, Target_flag, _) :-
        reached_target(Complete_edge, Target, Target_flag), !.

end_or_new(Complete_edge, _, _, New_active) :- !,
        new(New_active),
```

```
        clause_or_assert(parser:base_edge(Complete_edge)).

        % see if a limited Kleene-star repetition is finished.
 %1     only 1 left so finished repetitions carry on with rule.
end_kl(1, [Sv, Ev, H, (_ * _ if _, Rem), Info]) :- !,
        new([Sv, Ev, H, (Rem), Info]).
 %2     no more of rule left.
end_kl(1, [_, _, _, (_ * _ if _), _]) :- !.
 %3     more repetitions left.
end_kl(N, [Sv, Ev, H, (C * _ if E, Rem), Info]) :- !,
        N1 is N - 1, new([Sv, Ev, H, (C * N1 if E, Rem), Info]).
 %4
end_kl(N, [Sv, Ev, H, (C * _ if E), Info]) :- !,
        N1 is N - 1, new([Sv, Ev, H, (C * N1 if E), Info]).


 %1     assert new edge into the parser, extracting next categories.
new([/*new_active*/]) :- !.
 %2
new([Sv,Ev,Head,(Cata * N if Ea, Catb * N1 if Eb,
                                        Catc if Ec, Rem),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb, Catc],
                [Sv,Ev,Head,(Cata * N if Ea, Catb * N1 if Eb,
                                        Catc if Ec, Rem),Info])), !.
 %3
new([Sv,Ev,Head,(Cata * N if Ea, Catb * if Eb, Catc if Ec, Rem),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb, Catc],
                [Sv,Ev,Head,(Cata * N if Ea, Catb * if Eb,
                                        Catc if Ec, Rem),Info])), !.
 %4
new([Sv,Ev,Head,(Cata * if Ea, Catb * N1 if Eb, Catc if Ec, Rem),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb, Catc],
                [Sv,Ev,Head,(Cata * if Ea, Catb * N1 if Eb,
                                        Catc if Ec, Rem),Info])), !.
 %5
new([Sv,Ev,Head,(Cata * if Ea, Catb * if Eb, Catc if Ec, Rem),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb, Catc],
                [Sv,Ev,Head,(Cata * if Ea, Catb * if Eb,
                                        Catc if Ec, Rem),Info])), !.
 %6
new([Sv,Ev,Head,(Cata * N if Ea, Catb * N1 if Eb, Catc if Ec),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb, Catc],
                [Sv,Ev,Head,(Cata * N if Ea, Catb * N1 if Eb,
                                        Catc if Ec),Info])), !.
 %7
new([Sv,Ev,Head,(Cata * N if Ea, Catb * if Eb, Catc if Ec),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb, Catc],
                [Sv,Ev,Head,(Cata * N if Ea, Catb * if Eb,
                                                Catc if Ec),Info])), !.
 %8
new([Sv,Ev,Head,(Cata * if Ea, Catb * N1 if Eb, Catc if Ec),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb, Catc],
                [Sv,Ev,Head,(Cata * if Ea, Catb * N1 if Eb,
                                                Catc if Ec),Info])), !.
 %9
new([Sv,Ev,Head,(Cata * if Ea, Catb * if Eb, Catc if Ec),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb, Catc],
                [Sv,Ev,Head,(Cata * if Ea, Catb * if Eb,
                                                Catc if Ec),Info])), !.
 %10
new([Sv,Ev,Head,(Cata * N if Ea, Catb * N1 if Eb),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb],
                [Sv,Ev,Head,(Cata * N if Ea, Catb * N1 if Eb),Info])), !.
 %11
new([Sv,Ev,Head,(Cata * N if Ea, Catb * if Eb),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb],
                [Sv,Ev,Head,(Cata * N if Ea, Catb * if Eb),Info])), !.
```

```
 %12
new([Sv,Ev,Head,(Cata * if Ea, Catb * N1 if Eb),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb],
                [Sv,Ev,Head,(Cata * if Ea, Catb * N1 if Eb),Info])), !.
 %13
new([Sv,Ev,Head,(Cata * if Ea, Catb * if Eb),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb],
                        [Sv,Ev,Head,(Cata * if Ea, Catb * if Eb),Info])), !.
 %14
new([Sv,Ev,Head,(Cata * N if Ea, Catb if Eb, Rem),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb],
                [Sv,Ev,Head,(Cata * N if Ea, Catb if Eb, Rem),Info])), !.
 %15
new([Sv,Ev,Head,(Cata * if Ea, Catb if Eb, Rem),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb],
                        [Sv,Ev,Head,(Cata * if Ea, Catb if Eb, Rem),Info])), !.
 %16
new([Sv,Ev,Head,(Cata * N if Ea, Catb if Eb),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb],
                        [Sv,Ev,Head,(Cata * N if Ea, Catb if Eb),Info])), !.
 %17
new([Sv,Ev,Head,(Cata * if Ea, Catb if Eb),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata, Catb],
                        [Sv,Ev,Head,(Cata * if Ea, Catb if Eb),Info])), !.
 %18
new([Sv,Ev,Head,(Cata * N if Ea),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata],
                                [Sv,Ev,Head,(Cata * N if Ea),Info])), !.
 %19
new([Sv,Ev,Head,(Cata * if Ea),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cata],
                                [Sv,Ev,Head,(Cata * if Ea),Info])), !.
 %20
new([Sv,Ev,Head,(Cat if E, Rem),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cat],
                                [Sv,Ev,Head,(Cat if E, Rem),Info])), !.
 %21
new([Sv,Ev,Head,(Cat if E),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [Cat],
                                [Sv,Ev,Head,(Cat if E),Info])), !.
 %22
new([Sv,Ev,Head,(word_is W, Rem),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [W],
                                [Sv,Ev,Head,(word_is W, Rem),Info])), !.
 %23
new([Sv,Ev,Head,(word_is W),Info]) :- !,
        clause_or_assert(parser:active_edge(Ev, [W],
                                [Sv,Ev,Head,(word_is W),Info])), !.

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : fast_basics.pl                                 */
        /*      PURPOSE : defines semi-declarative versions of basic     */
        /*                  list procedures (deterministic).             */
        /*      NOTES   : declarative versions of procedures apply only */
        /*                 to lists with 11 or less members.             */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(fast_basics, [
                fappend/3,              % fappend(+List, +List1, -List2).
                fmember/2,              % fmember(+Element, +List).
                flength/2,              % flength(?Length, ?List).
```

```
                fdelete/3,              % fdelete(+Element, +List, -List1).
                freverse/2,             % freverse(+List, -List1).
                fcommon_member/2        % fcommon_member(+List, +List).
                ]).

        /* P R E D I C A T E S */

fappend(List, [], List) :- !.
fappend([], List, List) :- !.
fappend([A,B,C,D,E,F,G,H,I,J,K|Rest], List,
                                [A,B,C,D,E,F,G,H,I,J,K|Rest1]) :- !,
        fappend(Rest, List, Rest1).
fappend([A,B,C,D,E,F,G,H,I,J|Rest], List,
                                [A,B,C,D,E,F,G,H,I,J|Rest1]) :- !,
        fappend(Rest, List, Rest1).
fappend([A,B,C,D,E,F,G,H,I|Rest], List,
                                [A,B,C,D,E,F,G,H,I|Rest1]) :- !,
        fappend(Rest, List, Rest1).
fappend([A,B,C,D,E,F,G,H|Rest], List, [A,B,C,D,E,F,G,H|Rest1]) :- !,
        fappend(Rest, List, Rest1).
fappend([A,B,C,D,E,F,G|Rest], List, [A,B,C,D,E,F,G|Rest1]) :- !,
        fappend(Rest, List, Rest1).
fappend([A,B,C,D,E,F|Rest], List, [A,B,C,D,E,F|Rest1]) :- !,
        fappend(Rest, List, Rest1).
fappend([A,B,C,D,E|Rest], List, [A,B,C,D,E|Rest1]) :- !,
        fappend(Rest, List, Rest1).
fappend([A,B,C,D|Rest], List, [A,B,C,D|Rest1]) :- !,
        fappend(Rest, List, Rest1).
fappend([A,B,C|Rest], List, [A,B,C|Rest1]) :- !,
        fappend(Rest, List, Rest1).
fappend([A,B|Rest], List, [A,B|Rest1]) :- !,
        fappend(Rest, List, Rest1).
fappend([A|Rest], List, [A|Rest1]) :- !,
        fappend(Rest, List, Rest1).


fmember(El, [El|_]).
fmember(El, [_,El|_]).
fmember(El, [_,_,El|_]).
fmember(El, [_,_,_,El|_]).
fmember(El, [_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,El|_]).
fmember(El, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_|R]) :- !,
        fmember(El, R).


flength( 0, []) :- !.
flength( 1, [_]) :- !.
flength( 2, [_,_]) :- !.
flength( 3, [_,_,_]) :- !.
flength( 4, [_,_,_,_]) :- !.
flength( 5, [_,_,_,_,_]) :- !.
flength( 6, [_,_,_,_,_,_]) :- !.
flength( 7, [_,_,_,_,_,_,_]) :- !.
flength( 8, [_,_,_,_,_,_,_,_]) :- !.
```

```
flength( 9, [_,_,_,_,_,_,_,_,_]) :- !.
flength(10, [_,_,_,_,_,_,_,_,_,_]) :- !.
flength(11, [_,_,_,_,_,_,_,_,_,_,_]) :- !.
flength(12, [_,_,_,_,_,_,_,_,_,_,_,_]) :- !.
flength(13, [_,_,_,_,_,_,_,_,_,_,_,_,_]) :- !.
flength(14, [_,_,_,_,_,_,_,_,_,_,_,_,_,_]) :- !.
flength(15, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]) :- !.
flength(16, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]) :- !.
flength(17, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]) :- !.
flength(18, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]) :- !.
flength(19, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]) :- !.
flength(20, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]) :- !.
flength( N, [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_|Rest]) :- !,
        flength(N1, Rest),
        N is N1 + 21.


fdelete(El, [El|R], R) :- !.
fdelete(El, [A,El|R], [A|R]) :- !.
fdelete(El, [A,B,El|R], [A,B|R]) :- !.
fdelete(El, [A,B,C,El|R], [A,B,C|R]) :- !.
fdelete(El, [A,B,C,D,El|R], [A,B,C,D|R]) :- !.
fdelete(El, [A,B,C,D,E,El|R], [A,B,C,D,E|R]) :- !.
fdelete(El, [A,B,C,D,E,F,El|R], [A,B,C,D,E,F|R]) :- !.
fdelete(El, [A,B,C,D,E,F,G,El|R], [A,B,C,D,E,F,G|R]) :- !.
fdelete(El, [A,B,C,D,E,F,G,H,El|R], [A,B,C,D,E,F,G,H|R]) :- !.
fdelete(El, [A,B,C,D,E,F,G,H,I,El|R], [A,B,C,D,E,F,G,H,I|R]) :- !.
fdelete(El, [A,B,C,D,E,F,G,H,I,J,El|R], [A,B,C,D,E,F,G,H,I,J|R]) :- !.
fdelete(El, [A,B,C,D,E,F,G,H,I,J|R], [A,B,C,D,E,F,G,H,I,J|R1]) :- !,
        fdelete(El, R, R1).


freverse([], []) :- !.
freverse([A], [A]) :- !.
freverse([A,B], [B,A]) :- !.
freverse([A,B,C], [C,B,A]) :- !.
freverse([A,B,C,D], [D,C,B,A]) :- !.
freverse([A,B,C,D,E], [E,D,C,B,A]) :- !.
freverse([A,B,C,D,E,F], [F,E,D,C,B,A]) :- !.
freverse([A,B,C,D,E,F,G], [G,F,E,D,C,B,A]) :- !.
freverse([A,B,C,D,E,F,G,H], [H,G,F,E,D,C,B,A]) :- !.
freverse([A,B,C,D,E,F,G,H,I], [I,H,G,F,E,D,C,B,A]) :- !.
freverse([A,B,C,D,E,F,G,H,I,J], [J,I,H,G,F,E,D,C,B,A]) :- !.
freverse([A,B,C,D,E,F,G,H,I,J,K], [K,J,I,H,G,F,E,D,C,B,A]) :- !.
freverse([A,B,C,D,E,F,G,H,I,J,K,L], [L,K,J,I,H,G,F,E,D,C,B,A]) :- !.
freverse([A,B,C,D,E,F,G,H,I,J,K,L|R1], New) :- !,
        freverse(R1, R2),
        fappend(R2, [L,K,J,I,H,G,F,E,D,C,B,A], New).


fcommon_member([], _) :- !, fail.
fcommon_member(_, []) :- !, fail.
fcommon_member([A|R], Cats) :-
         ( fmember(A, Cats), ! | fcommon_member(R, Cats)).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : fs_basics.pl                                  */
        /*      PURPOSE : simple functions on F-structure lists.        */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(fs_basics, [
```

```
                sort_fs/2,
                sort_slots/2,
                insert_fs/3,
                member_fs/2,
                delete_fs/3
                ]).

        /* I M P O R T S */

:- use_module(greater_than, [
                greater_than/3
                ]).

        /* C O N S T A N T S */

        /* ordering relation for F-structure members. */
%1
constant(fs_rel( gt(F=_, F1=_, F @> F1) )).
%2
constant(_-_=_, _-_=_, slot_rel( gt(F-_=_, F1-_=_, F @> F1) )) :- !.
%3
constant(_-_=_, _=_, slot_rel( gt(F-_=_, F1=_,  F @> F1) )) :- !.
%4
constant(_=_, _-_=_, slot_rel( gt(F=_, F1-_=_,  F @> F1) )) :- !.
%5
constant(_=_, _=_, slot_rel( gt(F=_, F1=_, F @> F1) )) :- !.


        /* P R E D I C A T E S */

        /* sort_fs(+Fs, -Fs1).
           get the defined (constant) relationship for ordering members
           in an F-structure list and sort the F-structure list
           (insertion sort), using the defined relationship. */
%1
sort_fs(Fs, Fs1) :- constant(fs_rel(R)), sort(R, Fs, Fs1).


        /* sort(+L, -L1).
           carry out a straight insertion sort on list 'L' to produce
           the ordered list 'L1'. */
%1
sort(_, [], []) :- !.
%2
sort(R, [First|Rest], Sorted) :-
        sort(R, Rest, Sorted_rest),
        insert(R, First, Sorted_rest, Sorted).


        /* insert(+R, +E, +L, -L1).
           insert the element 'E', into its proper (sorted) place using
           the relationship 'R' to determine the ordering of two members,
           into the ordered list 'L' to produce a new ordered list 'L1'. */
%1
insert(R, Elem, [Elem1|Sorted], [Elem1|Sorted1]) :-
        greater_than(Elem, Elem1, R), !,
        insert(R, Elem, Sorted, Sorted1).
%2
insert(_, Elem, Sorted, [Elem|Sorted]) :- !.


        /* insert_fs(+P, +Fs, -Fs1).
           insert the 'attribute = value' pair 'P' into the F-structure
           ordered list 'Fs' to produce a new ordered F-structure list
           'Fs1'. */
%1
insert_fs(Val, Fs, Fs1) :- constant(fs_rel(R)), insert(R, Val, Fs, Fs1).
```

```
        /* member_fs(+P, +Fs).
           get the relationship used to determine order in an F-structure
           and determine whether the 'attribute = value' pair 'P' is a
           member of the ordered F-structure list 'Fs', if it is not
           then fail. */
 %1
member_fs(Pair, Fs) :- constant(fs_rel(R)), member(R, Pair, Fs).

 %1
member(_, Pair, [Pair|_]) :- !.
 %2
member(R, Pair, [First|Rest]) :-
        greater_than(Pair, First, R),
        member(R, Pair, Rest).


        /* sort_slots(+Slots, -Slots1). */
 %1
sort_slots([], []) :- !.
 %2
sort_slots([One], [One]) :- !.
 %3
sort_slots([First, Second|Rest], Slots1) :-
        insert_slot(First, [Second], Slots2), !,
        sort_rest_slots(Rest, Slots2, Slots1).

 %1
insert_slot(S, [S1|Rest], [S1|Rest1]) :-
        constant(S, S1, slot_rel(R)),
        greater_than(S, S1, R), !,
        insert_slot(S, Rest, Rest1).
 %2
insert_slot(S, Slots, [S|Slots]).

 %1
sort_rest_slots([], Slots, Slots) :- !.
 %2
sort_rest_slots([S|Rest], Slots1, Slots2) :-
        insert_slot(S, Slots1, Slots3),
        sort_rest_slots(Rest, Slots3, Slots2).

 %1
delete_fs(Pair, [Pair|Rest], Rest) :- !.
 %2
delete_fs(Pair, [First|Rest], [First|Rest1]) :- delete_fs(Pair, Rest, Rest1).

        /* ************************************************************* */



        /* ************************************************************* */
        /*      FILE    : fs_functs.pl                                   */
        /*      PURPOSE : simple operations on information structures.   */
        /* ************************************************************* */

        /* E X P O R T S */

:- module(fs_functs, [
                add_to_set/4,
                slot_funct/3,
                delete_val/5,    % (+F, +Fs, -Fs1, +Ptrs, -Val).
                add_to_funct/6, % (+F, +Feat=Val, +Fs, -Fs1, +Ptrs, -Ptrs1).
                add_constraint/3,
                def_equal/6,     % (+F-Subf, +Subf1, +Fs, -Fs1, +Ptrs, -Ptrs1).
                new_val/6,       % (+F, +Val, +Fs, -Fs1, +Ptrs, -Ptrs1).
```

```
                get_type/2
                ]).

        /* I M P O R T S */

:- use_module(fs_basics, [
                sort_fs/2,
                insert_fs/3,
                delete_fs/3,
                member_fs/2
                ]).

:- use_module(type_system, [
                compat/3
                ]).

:- use_module(new_info, [
                empty_info/1
                ]).

:- use_module(counters, [
                next_counter/2
                ]).

:- use_module(unify, [
                merge/6
                ]).

:- use_module(fast_basics, [
                fdelete/3,
                fappend/3
                ]).

        /* P R E D I C A T E S */

        /* add_to_set(+F, ^Val, +Fs, -Fs1).
           add the meta-variable 'Val' as a information structure to the
           set of information structures 'F' in information structure 'Fs'
           to produce 'Fs1', if 'F' does not exist in 'Fs' then insert it. */
 %1
add_to_set(F, var Val, [], [F = set [Val]]) :- !.
 %2
add_to_set(F, var Val, [F = set Vals|Rest], [F = set [Val|Vals]|Rest]) :- !.
 %3
add_to_set(F, var Val, [F1 = Val1|Rest], [F1 = Val1|Rest1]) :-
        F @> F1, !,
        add_to_set(F, var Val, Rest, Rest1).
 %4
add_to_set(F, var Val, Fs, [F = set [Val]|Fs]).


        /* slot_funct(+F, +S, -S1).
           the function 'F' must be present in the slots 'S' if these
           are of type 'full' in which case return the slots unaltered
           as 'S1' but if the slots 'S' are of type 'part' then the
           function 'F' may be a member of the slots 'S' or it may have
           to be added to 'S' to produce 'S1'. */

 %1     full slots, check that function 'F' is present.
slot_funct(F, full stype Slots, full stype Slots1) :-
        slot(F, Slots, Slots1).

 %2     partial slots the function may be there or may be added.
slot_funct(F, part stype Slots, part stype Slots1) :-
        slot_or_new(F, Slots, Slots1).

 %1
```

```
slot(F:D, [F - F1 = V:D1|R], [F - F1 = V:D2|R]) :- !,  compat(D, D1, D2).
 %2
slot(F:D, [F = V:D1|R], [F = V:D2|R]) :- !, compat(D, D1, D2).
 %3
slot(F:D, [F1 - F2 = V:D1|Rest], [F1 - F2 = V:D1|Rest1]) :- !,
        F @> F1,
        slot(F:D, Rest, Rest1).
 %4
slot(F:D, [F1 = V:D1|Rest], [F1 = V:D1|Rest1]) :-
        F @> F1, !,
        slot(F:D, Rest, Rest1).

 %1
slot_or_new(F:D, [], [F = _:D]) :- !.
 %2
slot_or_new(F:D, [F = V:D1|Rest], [F = V:D2|Rest]) :- !, compat(D, D1, D2).
 %3
slot_or_new(F:D, [F1 - F2 = V:T|Rest], [F1 - F2 = V:T|Rest1]) :- !,
        F @> F1, !,
        slot_or_new(F:D, Rest, Rest1).
 %4
slot_or_new(F:D, [F1 = V:T|Rest], [F1 = V:T|Rest1]) :-
        F @> F1, !,
        slot_or_new(F:D, Rest, Rest1).
 %5
slot_or_new(F:D, Slots, [F = _:D|Slots]).


        % delete_val/4, % (+F, +Fs, -Fs1, +Ptrs, -Val).
 %1
delete_val(_, [], [], _, Info):- !, empty_info(Info).
 %2
delete_val(F, [F = fs ptr N|R], R, Ptrs, Val) :- !, fmember(N = fs Val, Ptrs).
 %3
delete_val(F, [F = fs Val|R], R, _, Val) :- !.
 %4
delete_val(F, [F1|Rest], [F1|Rest1], Ptrs, Val) :-
        delete_val(F, Rest, Rest1, Ptrs, Val).

        % new_val(+F, +Val, +Fs, -Fs1, +Ptrs, -Ptrs1).
 %1
new_val(F, Val, Fs, Fs, Ptrs, [N = fs Val|Ptrs1]) :-
        member_fs(F = fs ptr N, Fs), !,
        fdelete(N = fs _, Ptrs, Ptrs1).
 %2
new_val(F, Val, Fs, Fs1, Ptrs, Ptrs) :- insert_fs(F = fs Val, Fs, Fs1).

 %1
add_to_funct(F, Eqn, Fs, Fs1, Ptrs, Ptrs1) :-
        ( delete_fs(F = fs Val, Fs, Fs2) | empty_info(Val), Fs2 = Fs ),
        add_to(Val, F, Eqn, Fs2, Fs1, Ptrs, Ptrs1).

 %1
add_to(ptr N, _, Eqn, Fs, Fs, Ptrs, [N = fs Val1|Ptrs2]) :-
        fdelete(N = fs Val, Ptrs, Ptrs2),
        insert_val(Eqn, Val, Val1).
 %2
add_to(Info^Val, F, Eqn, Fs, Fs1, Ptrs, Ptrs) :-
        insert_val(Eqn, Val, Val1),
        insert_fs(F = fs Info^Val1, Fs, Fs1).

 %1
insert_val(Feat = atom V, Fs, Fs1) :- insert_fs(Feat = atom V, Fs, Fs1).
 %2
insert_val(Constraint, Fs, Fs1) :- add_constraint(Constraint, Fs, Fs1).

 %1
```

```
add_constraint(F = Con, Fs, Fs1) :-
        ( delete_fs(F = Con1, Fs, Fs2) | Con1 = [], Fs2 = Fs ),
        new_constraint(Con, Con1, Con2),
        insert_fs(F = Con2, Fs2, Fs1).

 %1
new_constraint(Con, [], Con) :- !.
 %2
new_constraint(Con, Con, Con) :- !.
 %3
new_constraint(exists, val_c Val, exists val_c Val) :- !.
 %4
new_constraint(val_c Val, exists, exists val_c Val) :- !.
 %5
new_constraint(exists, neg_c Vals, exists neg_c Vals) :- !.
 %6
new_constraint(neg_c Vals, exists, exists neg_c Vals) :- !.
 %7
new_constraint(exists neg_c Vals, exists neg_c Vals1, exists neg_c Vals2) :- !,
        fappend(Vals, Vals1, Vals2).
 %8
new_constraint(neg_c Vals, exists neg_c Vals1, exists neg_c Vals2) :- !,
        fappend(Vals, Vals1, Vals2).
 %9
new_constraint(exists neg_c Vals, neg_c Vals1, exists neg_c Vals2) :- !,
        fappend(Vals, Vals1, Vals2).
 %10
new_constraint(Con, Con1, _) :-
        format('~N~w~w~w~n', ['Error in combining constraints', Con, Con1]).

 %1
def_equal(F - Subf, Subf1, Fs, Fs1, Ptrs, Ptrs1) :-
        delete_fs(Subf1 = fs ptr N, Fs, Fs2),
        delete_fs(F = fs I ind part stype
                                []^Quant^Sem^Var^Subfs, Fs2, Fs3), !,
        delete_fs(Subf = fs ptr N1, Subfs, Subfs1),
        unify_pointers(N, N1, Ptrs, Ptrs1, N2),
        insert_fs(Subf = fs ptr N2, Subfs1, Subfs2),
        insert_fs(F = fs I ind part stype
                                [Subf = _:_]^Quant^Sem^Var^Subfs2, Fs3, Fs4),
        insert_fs(Subf1 = fs ptr N2, Fs4, Fs1).
 %2
def_equal(F - Subf, Subf1, Fs, Fs1, Ptrs, [N = fs Info3|Ptrs1]) :-
        delete_val(Subf1, Fs, Fs2, Ptrs, Info1),
        delete_val(F, Fs2, Fs3, Ptrs, I ind part stype []^Quant^Sem^Var^Subfs),
        delete_val(Subf, Subfs, Subfs1, Ptrs, Info2),
        next_counter(ptr, N),
        merge(Info1, Info2, up, Ptrs, Info3, Ptrs1),
        insert_fs(Subf = fs ptr N, Subfs1, Subfs2),
        insert_fs(F = fs I ind part stype
                                [Subf=_:_]^Quant^Sem^Var^Subfs2, Fs3, Fs4),
        insert_fs(Subf1 = fs ptr N, Fs4, Fs1).

 %1
unify_pointers(N, N1, Ptrs, [N2 = fs Val2|Ptrs4], N2) :-
        fdelete(N = fs Val, Ptrs, Ptrs2),
        fdelete(N1 = fs Val1, Ptrs2, Ptrs3),
        merge(Val, Val1, up, Ptrs3, Val2, Ptrs4),
        next_counter(ptr, N2).

 %1
get_type(Info, D) :- var(Info), !, Info = _ ind _ stype _^_^_^(_:D)^_.
 %2
get_type(moved _Var temp Info, D) :- get_type(Info, D).
 %3
get_type(_ ind _ stype (domain=D)^_^_^_^_, D) :- !.
 %4
```

```
get_type(_ ind _ stype _^_^_^(_:D)^_, D).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : geo_types.pl                                   */
        /*      PURPOSE : defines the types for geographical database.   */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(geo_types, [
                sub_types/2
                ]).

        /* P R E D I C A T E S */

        /* type hierarchy for geographical database */

sub_types(d,            [       location, % locations on the earth.
                                measure,  % things that have quantity.
                                place,    % locations with areas.
                                peopled   % places with populations.
                        ]).

sub_types(location,     [       sea,
                                ocean,
                                country,
                                city,
                                river,
                                capital,
                                latitude,
                                continent   .
                        ]).

sub_types(measure,      [       area,
                                population
                        ]).

sub_types(place,        [       country,
                                continent
                        ]).

sub_types(peopled,      [       country,
                                city
                        ]).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : gram_eqns.pl                                   */
        /*      PURPOSE : converts LFG simple grammar equations into     */
        /*                information structures.                        */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(gram_eqns,
                [
                change/2
                ]).
```

```
        /* I M P O R T S */

:- use_module(fs_basics,
                [
                sort_fs/2,
                sort_slots/2
                ]).

:- use_module(new_info,
                [
                empty_info/1
                ]).

:- use_module(desig,
                [
                (designator)/1
                ]).

        /* P R E D I C A T E S */

        % change(+equation, -equation1).
        % make equations closer to Prolog, check features are valid
        % symbols and mark governed functions (prefixed 'd').
%1
change(up = down, up = down) :- !.
%2
change(down set_val_of up F, down set_val_of (up F)) :-  atom(F), !.
%3
change(complete (up(down pcase)) = down,
                                complete (up(down pcase)) = down) :- !.
%4
change((up(down pcase)) = down, (up(down pcase)) = down) :- !.
%5
change((down F-F1) = controllee sub Cat,
                                (down F-F1) = controllee sub Cat) :-
        atom(F), atom(F1), atom(Cat), !.
%6
change((up F) = controllee sub Cat, (up d F) = controllee sub Cat) :-
        atom(F), atom(Cat), designator F, !.
%7
change((up F) = controllee sub Cat, (up F) = controllee sub Cat) :-
        atom(F), !.
%8
change(down = controller super Dom sub Cat, down = controller(Dom, Cat, _)) :-
        atom(Dom), !.
%9
change((down F) = controllee sub Cat, (down d F) = controllee sub Cat) :-
        atom(F), designator F, !.
%10
change((down F) = controllee sub Cat, (down F) = controllee sub Cat) :-
        atom(F), !.
%11
change(up = controllee sub Cat, up = controllee sub Cat) :- atom(Cat), !.
%12
change((up F) = controller super Dom sub Cat,
                                (up d F) = controller(Dom, Cat, _)) :-
        atom(F), atom(Dom), designator F, !.
%13
change((up F) = controller super Dom sub Cat,
                                (up F) = controller(Dom, Cat, _)) :-
        atom(F), atom(Dom), !.
%14
change((up F - F1) = down, (up d F - d F1) = down) :-
        designator(F), designator(F1), atom(F), atom(F1), !.
%15
change((up F - F1) = down, (up d F - F1) = down) :-
        designator(F), \+ designator(F1), atom(F), atom(F1), !.
```

```
%16
change((up F - F1) = down, (up F - d F1) = down) :-
        \+designator(F), designator(F1), atom(F), atom(F1), !.
%17
change((up F - F1) = down, (up F - F1) = down) :-
        \+ designator(F), \+ designator(F1), atom(F), atom(F1), !.
%18
change((up F - Ft) c Value, up I ind part stype Slots^
                Quant^Sem^Var^[F = fs Fi ind Fslots^Fquant^Fsem^
                                                Fvar^[Ft = val_c Value]]) :-
        atom(F), atom(Ft), !,
        empty_info(I ind _slots^Quant^Sem^Var^_fs),
        empty_info(Fi ind Fslots^Fquant^Fsem^Fvar^_ffs),
        ( designator(F), !, Slots = [F=_:_] | Slots = [] ).
%19
change((down F - Ft) c Value, down I ind part stype Slots^
                Quant^Sem^Var^[F = fs Fi ind Fslots^Fquant^Fsem^
                                                Fvar^[Ft = val_c Value]]) :-
        atom(F), atom(Ft), !,
        empty_info(I ind _^Quant^Sem^Var^_fs),
        empty_info(Fi ind Fslots^Fquant^Fsem^Fvar^_ffs),
        ( designator(F), !, Slots = [F=_:_] | Slots = [] ).
%20
change((up sem) = Sem, up I ind part stype Slots^Quant^Sem1^Var^Fs) :-
        Sem =.. [S|Args],
        make_sem_slots(Args, Vars, Var, Slots),
        Sem1 =.. [S|Vars], !,
        empty_info(I ind _^Quant^_^_^Fs).
%21
change((down sem) = Sem, down I ind part stype Slots^Quant^Sem1^Var^Fs) :-
        Sem =.. [S|Args],
        make_sem_slots(Args, Vars, Var, Slots),
        Sem1 =.. [S|Vars], !,
        empty_info(I ind _^Quant^_^_^Fs).
%22
change((up pred) = P >> [(up F)],
                up I ind full stype [F=_:_]^Quant^Sem^Var^[pred=atom P]) :-
        atom(P), !, empty_info(I ind _slots^Quant^Sem^Var^_).
%23
change((down pred) = P >> [(up F)],
                down I ind full stype [F=_:_]^Quant^Sem^Var^[pred=atom P]) :-
        atom(P), !, empty_info(I ind _slots^Quant^Sem^Var^_).
%24
change((up pred) = P >> [(up F), (up F1)],
                up I ind full stype Slots^Quant^Sem^Var^[pred=atom P]):-
        atom(P), !,
        sort_slots([F = _:_, F1 = _:_], Slots),
        empty_info(I ind _^Quant^Sem^Var^_).
%25
change((down pred) = P >> [(up F), (up F1)],
                down I ind full stype Slots^Quant^Sem^Var^[pred = atom P]):-
        atom(P), !,
        sort_slots([F = _:_, F1 = _:_], Slots),
        empty_info(I ind _^Quant^Sem^Var^_).
%26
change((up pred) = P >> [(up F), (up F1), (up F2)],
                up I ind full stype Slots^Quant^Sem^Var^[pred = atom P]):-
        atom(P), !,
        sort_slots([F = _:_, F1 = _:_, F2 = _:_], Slots),
        empty_info(I ind _^Quant^Sem^Var^_).
%27
change((down pred) = P >> [(up F), (up F1), (up F2)],
                down I ind full stype Slots^Quant^Sem^Var^[pred = atom P]):-
        atom(P), !,
        sort_slots([F = _:_, F1 = _:_, F2 = _:_], Slots),
        empty_info(I ind _^Quant^Sem^Var^_).
%28
```

```
change((up pred) = P >> [(up F), (up F1), (up F2), (up F3)],
                up I ind full stype Slots^Quant^Sem^Var^[pred = atom P]):-
        atom(P), !,
        sort_slots([F = _:_, F1 = _:_, F2 = _:_, F3 = _:_], Slots),
        empty_info(I ind _^Quant^Sem^Var^_).
 %29
change((down pred) = P >> [(up F), (up F1), (down F2), (up F3)],
                down I ind full stype Slots^Quant^Sem^Var^[pred = atom P]):-
        atom(P), !,
        sort_slots([F = _:_, F1 = _:_, F2 = _:_, F3 = _:_], Slots),
        empty_info(I ind _^Quant^Sem^Var^_).
 %30
change((up F - domain) = Type,
                up I ind part stype [F = _:Type]^Quant^Sem^Var^Fs) :-
        atom(F), !, empty_info(I ind _^Quant^Sem^Var^Fs).
 %31
change((down F - domain) = Type,
                down I ind part stype [F = _:Type]^Quant^Sem^Var^Fs) :-
        atom(F), !, empty_info(I ind _^Quant^Sem^Var^Fs).
 %32
change((up F) = down, (up d F) = down) :- atom(F), designator F.
 %33
change((up F) = down, (up F) = down) :- atom(F), !.
 %34
change(complete (up F) = down, complete (up d F) = down) :-
        atom(F), designator F, !.
 %35
change(complete (up F) = down, complete (up F) = down) :- atom(F), !.
 %36
change((up mood) = Value,
                up I ind Slots^Value^Sem^Var^[mood = atom Value]) :- !,
        empty_info(I ind Slots^_^Sem^Var^_).
 %37
change((up Ft) = (down Ft1), (up Ft) = (down Ft1)) :- atom(Ft), atom(Ft1), !.
 %38
change((up Ft) = Value, up I ind Slots^Quant^Sem^Var^[Ft = atom Value]) :-
        atom(Ft), !,
        empty_info(I ind Slots^Quant^Sem^Var^_).
 %39
change((down Ft) = Value, down I ind Slots^Quant^Sem^Var^[Ft = atom Value]) :-
        atom(Ft), !, empty_info(I ind Slots^Quant^Sem^Var^_).
 %40
change((up Ft) c Value, up I ind Slots^Quant^Sem^Var^[Ft = val_c Value]) :-
        atom(Ft), !, empty_info(I ind Slots^Quant^Sem^Var^_).
 %41
change((down Ft) c Value, down I ind
                                Slots^Quant^Sem^Var^[Ft = val_c Value]) :-
        atom(Ft), !,
        empty_info(I ind Slots^Quant^Sem^Var^_).
 %42
change(not (up F - Ft) c Val, up I ind part stype [F=_:_]^Quant^
                Sem^Var^[F = fs Fi ind
                        Fslots^Fquant^Fsem^Fvar^[Ft = neg_c [Val]]]) :-
        atom(F), atom(Ft), !,
        empty_info(I ind _slots^Quant^Sem^Var^_),
        empty_info(Fi ind Fslots^Fquant^Fsem^Fvar^_).
 %43
change(not (down F - Ft) c Val, down I ind part stype
                [F = _:_]^Quant^Sem^Var^[F = fs Fi ind
                        Fslots^Fquant^Fsem^Fvar^[Ft = neg_c [Val]]]) :-
        atom(F), atom(Ft), !,
        empty_info(I ind _^Quant^Sem^Var^_fs),
        empty_info(Fi ind Fslots^Fquant^Fsem^Fvar^_).
 %44
change(not (up Ft) c Val, up I ind Slots^Quant^Sem^Var^[Ft = neg_c [Val]]) :-
        atom(Ft), !, empty_info(I ind Slots^Quant^Sem^Var^_).
 %45
```

```
change(not (down Ft) c Val,
                        down I ind Slots^Quant^Sem^Var^[Ft = neg_c [Val]]) :-
        atom(Ft), !, empty_info(I ind Slots^Quant^Sem^Var^_).
 %46
change(not (up Ft), up I ind Slots^Quant^Sem^Var^[Ft = none]) :-
        atom(Ft), !, empty_info(I ind Slots^Quant^Sem^Var^_).
 %47
change(not (down Ft), down I ind Slots^Quant^Sem^Var^[Ft = none]) :-
        atom(Ft), !, empty_info(I ind Slots^Quant^Sem^Var^_).
 %48
change((up Ft), up I ind Slots^Quant^Sem^Var^[Ft = exists]) :-
        atom(Ft), !, empty_info(I ind Slots^Quant^Sem^Var^_).
 %49
change((down Ft), down I ind Slots^Quant^Sem^Var^[Ft = exists]) :-
        atom(Ft), !, empty_info(I ind Slots^Quant^Sem^Var^_).
 %50
change(down fs_is Fs, down fs_is Fs).
 %51
change((up F) fs_is Fs, (up d F) fs_is Fs) :- designator F.
 %52
change(controller(Cat, Fs), controller(Cat, Fs)).
 %53
change((down F - F1) = (up F2), (down F - F1) = (up d F2)) :-
        designator F2.
 %54
change((up F - Ft), up I ind part stype Slots^Quant^
                Sem^Var^[F = fs Fi ind
                        Fslots^Fquant^Fsem^Fvar^[Ft = exists]]) :-
        atom(F), atom(Ft), !,
        empty_info(I ind _^Quant^Sem^Var^_),
        empty_info(Fi ind Fslots^Fquant^Fsem^Fvar^_),
        ( designator F, Slots = [F = _:_] | Slots = [] ).
 %55
change((down F - Ft), down I ind part stype
                Slots^Quant^Sem^Var^[F = fs Fi ind
                        Fslots^Fquant^Fsem^Fvar^[Ft = exists]]) :-
        atom(F), atom(Ft), !,
        empty_info(I ind _^Quant^Sem^Var^_fs),
        empty_info(Fi ind Fslots^Fquant^Fsem^Fvar^_),
        ( designator F, Slots = [F = _:_] | Slots = [] ).
 %56
change(Eqn, _) :-
        format('~N ~*| ~w ~n ~*| ~w ~w ~2n',
                        [5, '** Error in grammar **', 5,
                                'Unrecognised equation :', Eqn]), !, fail.

 %1
make_sem_slots(Args, Vars, Quant_var, Slots) :-
        make_slots(Args, Vars, Quant_var, Slots1),
        sort_slots(Slots1, Slots).

 %1
make_slots([/*args*/], [/*vars*/], _, [/*slots*/]) :- !.
 %2
make_slots([quant|Rest], [Var|Rest1], Var, Sem_slots) :-
        make_slots(Rest, Rest1, _, Sem_slots).
 %3
make_slots([F|Rest], [Var|Rest1], Quant_var, [F = Var:_|Rest2]) :-
        make_slots(Rest, Rest1, Quant_var, Rest2).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : gram_pp.pl                                    */
```

```
        /*      PURPOSE : pre-processes LFG grammar rules.             */
        /*      NOTES   : the LFG grammar rules are not allowed to     */
        /*                contain disjunctions. This pre-processor     */
        /*                collects all the rules with the same category */
        /*                or word as first daughter and asserts these  */
        /*                together in the database for fast access by  */
        /*                the parser. Also controller equations are    */
        /*                reformed so that the f-structure source for a */
        /*                controller and actual controller f-structure, */
        /*                which may occur at different places in a rule */
        /*                are represented by a single variable.        */
        /* ********************************************************** */

        /* E X P O R T S */

:- module(gram_pp, [
                create_grammar/0,
                '--->'/2
                ]).

:- dynamic
        '--->'/2.                       % retracted as processed.

        /* I M P O R T S */

:- ensure_loaded(define_lfg).

:- use_module(library(findall), [
                findall/3
                ]).

:- use_module(more_basics, [
                clause_or_assert/1
                ]).

:- use_module(fast_basics, [
                fappend/3
                ]).

:- use_module(gram_eqns, [
                change/2
                ]).

:- use_module(gram_wrds, [
                add_grammar_words/0
                ]).

:- use_module(desig, [
                designator/1
                ]).

:- use_module(make_links, [
                make_links/0,
                break_links/0
                ]).

        /* P R E D I C A T E S */
%1
create_grammar :-
        format('~*| ~w ~2n', [5, 'Starting to pre-process grammar rules']),
        break_links,
        make_links,                             % produce top-down links.
        pre_process_cat_starts, !,
        pre_process_word_starts, !,
        add_grammar_words,
        format('~*| ~w ~2n', [5, 'Grammar rules pre-processed']), !.
%2
```

```
create_grammar :-
        format('~*| ~w ~2n', [5, 'Error during grammar pre-processing']).

 %1     not all rule formats are in place just likely ones.
pre_process_cat_starts :-
        more_cat_rules(Cat), !,
        findall([Head, (Cat if Eqns1, Rem1)], (retract((Head ---> Cat
                        eqns Eqns, Rem)), reform((Cat eqns Eqns, Rem),
                                (Cat if Eqns1, Rem1)) ), Start1),
        findall([Head, (Cat if Eqns1)],
                      ( retract((Head ---> Cat eqns Eqns)),
                        reform(Cat eqns Eqns, Cat if Eqns1) ), Complete1),
        findall([Head, (Cat if Eqns1, Rem1)],
                      ( retract( (Head ---> bnd Cat eqns Eqns , Rem) ),
                        reform( (bnd Cat eqns Eqns , Rem),
                                          (Cat if Eqns1, Rem1) ) ), Start2),
        findall([Head, (Cat if Eqns1)],
                      ( retract((Head ---> bnd Cat eqns Eqns)),
                        reform(bnd Cat eqns Eqns, Cat if Eqns1) ), Complete2),
        findall([Head, (Cat if Eqns1, Rem1)],
                      ( retract( (Head ---> lnk Cat eqns Eqns , Rem) ),
                        reform( (lnk Cat eqns Eqns , Rem),
                                          (Cat if Eqns1, Rem1) ) ),  Start3),
        findall([Head, (Cat if Eqns1)],
                      ( retract((Head ---> lnk Cat eqns Eqns)),
                        reform(lnk Cat eqns Eqns, Cat if Eqns1) ), Complete3),
        findall([Head, (_ * _ if _ , _ * _ if _, Cat if Eqns1, Rem1)],
                      ( retract((Head ---> Cat * N eqns Eqns , Rem)),
                        reform((Cat eqns Eqns , Rem), (Cat if Eqns1, Rem1))
                      ),  Start4),
        findall([Head, (_ * _ if _ , _ * _ if _, Cat if Eqns1)],
                      ( retract((Head ---> Cat * N eqns Eqns)),
                        reform((Cat eqns Eqns), (Cat if Eqns1)) ), Complete4),
        findall([Head, (_ * if _ , _ * _ if _, Cat if Eqns1)],
                      ( retract((Head ---> Cat * N eqns Eqns)),
                        reform((Cat eqns Eqns), (Cat if Eqns1)) ), Complete5),
        findall([Head, (_ * _ if _ , _ * if _, Cat if Eqns1)],
                      ( retract((Head ---> Cat * N eqns Eqns)),
                        reform((Cat eqns Eqns), (Cat if Eqns1)) ), Complete6),
        findall([Head, (_ * if _ , _ * if _, Cat if Eqns1)],
                      ( retract((Head ---> Cat * N eqns Eqns)),
                        reform((Cat eqns Eqns), (Cat if Eqns1)) ), Complete7),
        findall([Head, (_ * _ if _ , Cat * N if Eqns1, Rem1)],
                      ( retract((Head ---> Cat * N eqns Eqns , Rem)),
                        reform((Cat eqns Eqns , Rem), (Cat if Eqns1, Rem1))
                      ),  Start5),
        findall([Head, (_ * _ if _ , Cat * N if Eqns1)],
                      ( retract((Head ---> Cat * N eqns Eqns)),
                        reform((Cat eqns Eqns), (Cat if Eqns1))
                          ), Start_complete1),
        findall([Head, (_ * if _ , Cat * N if Eqns1)],
                      ( retract((Head ---> Cat * N eqns Eqns)),
                        reform((Cat eqns Eqns), (Cat if Eqns1))
                      ),  Start_complete2),
        findall([Head, (_ * _ if _ , Cat * if Eqns1)],
                      ( retract((Head ---> Cat * N eqns Eqns)),
                        reform((Cat eqns Eqns), (Cat if Eqns1))
                      ),  Start_complete3),
        findall([Head, (Cat * N if Eqns1 , Rem1)],
                      ( retract((Head ---> Cat * N eqns Eqns , Rem)),
                        reform((Cat eqns Eqns , Rem), (Cat if Eqns1, Rem1))
                      ),  Start6),
        findall([Head, (Cat * if Eqns1 , Rem1)],
                      ( retract((Head ---> Cat * eqns Eqns , Rem)),
                        reform((Cat eqns Eqns , Rem), (Cat if Eqns1, Rem1))
                      ),  Start7),
        findall([Head, (Cat * N if Eqns1)],
```

```
                    ( retract((Head ---> Cat * N eqns Eqns)),
                      reform(Cat eqns Eqns, Cat if Eqns1)
                    ),  Start_complete4),
        findall([Head, (Cat * if Eqns1)],
                    ( retract((Head ---> Cat * eqns Eqns)),
                      reform(Cat eqns Eqns, Cat if Eqns1)
                    ),  Start_complete5),
        findall([Head, Body1],
                    ( retract((Head ---> conj Cat eqns E, Rem)),
                      reform((conj Cat eqns E, Rem), Body1) ), Start_conj),
        findall([Head, Body1],
                    ( retract((Head ---> bnd Cat eqns E, Rem)),
                      reform((bnd Cat eqns E, Rem), Body1) ), Start_bnd),
        findall([Head, Body1],
                    ( retract((Head ---> bnd Cat eqns E)),
                      reform((bnd Cat eqns E), Body1) ), Complete_bnd),
        findall([Head, Body1],
                    ( retract((Head ---> lnk Cat eqns E, Rem)),
                      reform((lnk Cat eqns E, Rem), Body1) ), Start_lnk),
        findall([Head, Body1],
                    ( retract((Head ---> bnd Cat eqns E)),
                      reform((lnk Cat eqns E), Body1) ), Complete_lnk),
        fappend(Start1, Start2, Starta), fappend(Start3, Starta, Startb),
        fappend(Start4, Startb, Startc), fappend(Start5, Startc, Startd),
        fappend(Start_complete1, Startd, Starte),
        fappend(Start_complete2, Starte, Startf),
        fappend(Start_complete3, Startf, Startg),
        fappend(Start6, Startg, Starth), fappend(Start7, Starth, Starti),
        fappend(Start_complete4, Starti, Startj),
        fappend(Start_complete5, Startj, Startk),
        fappend(Start_conj, Startk, Startl),
        fappend(Start_lnk, Startl, Startm),
        fappend(Start_bnd, Startm, All_start),
        fappend(Complete1, Complete2, Compa),
        fappend(Complete3, Compa, Compb), fappend(Complete4, Compb, Compc),
        fappend(Complete5, Compc, Compd), fappend(Complete6, Compd, Compe),
        fappend(Complete7, Compe, Compf),
        fappend(Start_complete1, Compf, Compg),
        fappend(Start_complete2, Compg, Comph),
        fappend(Start_complete3, Comph, Compi),
        fappend(Start_complete4, Compi, Compj),
        fappend(Start_complete5, Compj, Compk),
        fappend(Complete_lnk, Compk, Compl),
        fappend(Complete_bnd, Compl, All_complete),
        asserta(parser:(Cat rules_which_cat(All_start, All_complete))),
        pre_process_cat_starts.
 %2
pre_process_cat_starts :- !.

 %1
pre_process_word_starts :-
        more_word_rules(Word),
        findall([word_is Word, Head, Rem1],
                        ( retract((Head ---> Word , Rem)),
                          reform((Word , Rem), (word_is Word , Rem1))
                        ),  Start),
        findall([word_is Word, Head],
                        ( retract((Head ---> Word)), atom(Word) ), Complete),
        asserta(parser:(Word rules_which_word(Start, Complete))),
        pre_process_word_starts.
 %2
pre_process_word_starts :- !.

 %1
more_cat_rules(Cat) :- (_ ---> conj Cat eqns _, _).
 %2
more_cat_rules(Cat) :- (_ ---> Cat eqns _, _).
```

```
 %3
more_cat_rules(Cat) :- (_ ---> Cat eqns _).
 %4
more_cat_rules(Cat) :- (_ ---> bnd Cat eqns _, _).
 %5
more_cat_rules(Cat) :- (_ ---> bnd Cat eqns _).
 %6
more_cat_rules(Cat) :- (_ ---> lnk Cat eqns _, _).
 %7
more_cat_rules(Cat) :- (_ ---> lnk Cat eqns _).
 %8
more_cat_rules(Cat) :- (_ ---> Cat * eqns _, _).
 %9
more_cat_rules(Cat) :- (_ ---> Cat * eqns _).
 %10
more_cat_rules(Cat) :- (_ ---> Cat * _ eqns _, _).
 %11
more_cat_rules(Cat) :- (_ ---> Cat * _ eqns _).

 %1
more_word_rules(Word) :- (_ ---> Word , _), atom(Word).
 %2
more_word_rules(Word) :- (_ ---> Word), atom(Word).

 %1
reform((conj Body), Body1) :- !,
        domains(Body, Body2),    % move controllers to their domain roots.
        process(Body2, Body3),
        add_conj_cntrl(Body3, Body1), !.
 %2
reform(Body, New_body) :-
        domains(Body, Body1),
        process(Body1, New_body), !.

 %1
domains(Body, Body1) :-
        ctrlers_in(Body, [], Ctrls1, Body2),
        add_ctrls(Ctrls1, Body2, Ctrls2, Body1),
        all_used(Ctrls2).

all_used([]).   % no controllers left.

 %1
ctrlers_in((bnd Cat eqns E, Rest), Ctrls, Ctrls1,
                                    (bnd Cat eqns E1, Rest1)) :- !,
        ctrl_eqn(E, Cat, Ctrls, Ctrls2, E1),
        ctrlers_in(Rest, Ctrls2, Ctrls1, Rest1).
 %2
ctrlers_in((bnd Cat eqns E), Ctrls, Ctrls1, (bnd Cat eqns E1)) :- !,
        ctrl_eqn(E, Cat, Ctrls, Ctrls1, E1).
 %3
ctrlers_in((lnk Cat eqns E, Rest), Ctrls, Ctrls1,
                                    (lnk Cat eqns E1, Rest1)) :- !,
        ctrl_eqn(E, Cat, Ctrls, Ctrls2, E1),
        ctrlers_in(Rest, Ctrls2, Ctrls1, Rest1).
 %4
ctrlers_in((lnk Cat eqns E), Ctrls, Ctrls1, (lnk Cat eqns E1)) :- !,
        ctrl_eqn(E, Cat, Ctrls, Ctrls1, E1).
 %5
ctrlers_in((Cat * N eqns E, Rest), Ctrls, Ctrls1,
                                    (Cat * N eqns E1, Rest1)) :- !,
        ctrl_eqn(E, Cat, Ctrls, Ctrls2, E1),
        ctrlers_in(Rest, Ctrls2, Ctrls1, Rest1).
 %6
ctrlers_in((Cat * N eqns E), Ctrls, Ctrls1, (Cat * N eqns E1)) :- !,
        ctrl_eqn(E, Cat, Ctrls, Ctrls1, E1).
 %7
```

```
ctrlers_in((Cat * eqns E, Rest), Ctrls, Ctrls1, (Cat * eqns E1, Rest1)) :- !,
        ctrl_eqn(E, Cat, Ctrls, Ctrls2, E1),
        ctrlers_in(Rest, Ctrls2, Ctrls1, Rest1).
 %8
ctrlers_in((Cat * eqns E), Ctrls, Ctrls1, (Cat * eqns E1)) :- !,
        ctrl_eqn(E, Cat, Ctrls, Ctrls1, E1).
 %9
ctrlers_in((Cat eqns E, Rest), Ctrls, Ctrls1, (Cat eqns E1, Rest1)) :- !,
        ctrl_eqn(E, Cat, Ctrls, Ctrls2, E1),
        ctrlers_in(Rest, Ctrls2, Ctrls1, Rest1).
 %10
ctrlers_in((Word, Rest), Ctrls, Ctrls1, (Word, Rest1)) :- !,
        ctrlers_in(Rest, Ctrls, Ctrls1, Rest1).
 %11
ctrlers_in((Cat eqns E), Ctrls, Ctrls1, (Cat eqns E1)) :- !,
        ctrl_eqn(E, Cat, Ctrls, Ctrls1, E1).
 %12
ctrlers_in((Word), Ctrls, Ctrls, (Word)) :- !.

 %1     controller belongs to this node.
ctrl_eqn(down = controller super Cat sub Sc & Eqns, Cat, Ctrls,
                Ctrls1, down = controller super Cat sub Sc & Eqns1) :- !,
        ctrl_eqn(Eqns, Cat, Ctrls, Ctrls1, Eqns1).
 %2     controller belongs to another node.
ctrl_eqn(down = controller super Cat sub Sc & Eqns, Cat1, Ctrls,
        [Cat : controller(Sc, Fs)|Ctrls1], down fs_is Fs & Eqns1) :- !,
        ctrl_eqn(Eqns, Cat1, Ctrls, Ctrls1, Eqns1).
 %3     controller belongs to this node.
ctrl_eqn(down = controller super Cat sub Sc, Cat, Ctrls,
                        Ctrls, down = controller super Cat sub Sc) :- !.
 %4     controller belongs to another node.
ctrl_eqn(down = controller super Cat sub Sc, _, Ctrls,
                        [Cat : controller(Sc, Fs)|Ctrls], down fs_is Fs) :- !.
 %5
ctrl_eqn((up F) = controller super Cat sub Sc & Eqns, Cat, Ctrls,
                Ctrls1, (up F) = controller super Cat sub Sc & Eqns1) :- !,
        ctrl_eqn(Eqns, Cat, Ctrls, Ctrls1, Eqns1).
 %6
ctrl_eqn((up F) = controller super Cat sub Sc, Cat, Ctrls,
                        Ctrls, (up F) = controller super Cat sub Sc) :- !.
 %7
ctrl_eqn((up F) = controller super Cat sub Sc & Eqns, Cat1,
                Ctrls, [Cat : controller(Sc, Fs)|Ctrls1],
                                        (up F) fs_is Fs & Eqns1) :- !,
        ctrl_eqn(Eqns, Cat1, Ctrls, Ctrls1, Eqns1).
 %8
ctrl_eqn((up F) = controller super Cat sub Sc, _, Ctrls,
                [Cat : controller(Sc, Fs)|Ctrls], (up F) fs_is Fs) :- !.
 %9
ctrl_eqn(Eqn & Rest, Cat, Ctrls, Ctrls1, Eqn & Rest1) :- !,
        ctrl_eqn(Rest, Cat, Ctrls, Ctrls1, Rest1).
 %10
ctrl_eqn(Eqn, _, Ctrls, Ctrls, Eqn).


 %1
add_ctrls([], Body, [], Body) :- !.
 %2
add_ctrls([Root : Ctrler|R], Body, Ctrlers1, Body1) :-
        find_root(Root, Ctrler, Body, Body2),
        add_ctrls(R, Body2, Ctrlers1, Body1).
 %3
add_ctrls([Ctrl|R], Body, [Ctrl|R1], Body1) :-
        add_ctrls(R, Body, R1, Body1).

 %1
find_root(Cat, Ctrler, (bnd Cat eqns E, Rest), (bnd Cat eqns E1, Rest)) :-
```

```
        join_first(E, Ctrler, E1).
 %2
find_root(Cat, Ctrler, (bnd Cat eqns E), (bnd Cat eqns E1)) :-
        join_first(E, Ctrler, E1).
 %3
find_root(Cat, Ctrler, (lnk Cat eqns E, Rest), (lnk Cat eqns E1, Rest)) :-
        join_first(E, Ctrler, E1).
 %4
find_root(Cat, Ctrler, (lnk Cat eqns E), (lnk Cat eqns E1)) :-
        join_first(E, Ctrler, E1).
 %5
find_root(Cat, Ctrler, (Cat * N eqns E, Rest), (Cat * N eqns E1, Rest)) :-
        join_first(E, Ctrler, E1).
 %6
find_root(Cat, Ctrler, (Cat * N eqns E), (Cat * N eqns E1)) :-
        join_first(E, Ctrler, E1).
 %7
find_root(Cat, Ctrler, (Cat * eqns E, Rest), (Cat * eqns E1, Rest)) :-
        join_first(E, Ctrler, E1).
 %8
find_root(Cat, Ctrler, (Cat * eqns E), (Cat * eqns E1)) :-
        join_first(E, Ctrler, E1).
 %9
find_root(Cat, Ctrler, (Cat eqns E, Rest), (Cat eqns E1, Rest)) :-
        join_first(E, Ctrler, E1).
 %10
find_root(Cat, Ctrler, (Cat eqns E), (Cat eqns E1)) :-
        join_first(E, Ctrler, E1).
 %11
find_root(Cat, Ctrler, (F , R), (F , R1)) :-
        find_root(Cat, Ctrler, R, R1).

 %1
add_conj_cntrl((Cat if E, Rem), (Cat if E, Rem1)) :- !, add_conj(Rem, Rem1).
 %2
add_conj((Cat if E, Rem), (Cat if conj_cntrl E, Rem1)) :- !,
        add_conj(Rem, Rem1).
 %3
add_conj((Cat if E), (Cat if conj_cntrl E)) :- !.
 %4
add_conj((word_is W, Rem), (word_is W, Rem1)) :- !, add_conj(Rem, Rem1).
 %5
add_conj((word_is W), (word_is W)) :- !.

 %1
process((bnd Cat eqns E, Rest), (Cat if E1, Rest1)) :- !,
        convert(Cat if E, Cat if E2),
        join_first(E2, bound, E1),
        process(Rest, Rest1).
 %2
process((bnd Cat eqns E), (Cat if E1)) :- !,
        convert(Cat if E, Cat if E2),
        join_first(E2, bound, E1).
 %3
process((lnk Cat eqns E, Rest), (Cat if E1, Rest1)) :- !,
        convert(Cat if E, Cat if E2),
        join_first(E2, linkage, E1),
        process(Rest, Rest1).
 %4
process((lnk Cat eqns E), (Cat if E1)) :- !,
        convert(Cat if E, Cat if E2),
        join_first(E2, linkage, E1).
 %5
process((Cat * N eqns E , Rest), (Cat * N if E1 , Rest1)) :- !,
        convert(Cat if E, Cat if E1),
        process(Rest, Rest1).
 %6
```

```
process((Cat * eqns E , Rest), (Cat * if E1 , Rest1)) :- !,
        convert(Cat if E, Cat if E1),
        process(Rest, Rest1).
 %7
process(Cat * N eqns E, Cat * N if E1) :- !, convert(Cat if E, Cat if E1).
 %8
process(Cat * eqns E, Cat * if E1) :- !, convert(Cat if E, Cat if E1).
 %9
process((Cat eqns E , Rest), (Cat if E1 , Rest1)) :- !,
        convert(Cat if E, Cat if E1),
        process(Rest, Rest1).
 %10
process(Cat eqns E, Cat if E1) :- !, convert(Cat if E, Cat if E1).
 %11
process((W, Rest), (word_is W , Rest1)) :- !,
        clause_or_assert(temp:gram_word(W)),
        process(Rest, Rest1).
 %12
process(W, word_is W) :- !, clause_or_assert(temp:gram_word(W)).

 %1
convert(Cat if E, Cat if E1) :- alter_gram(E, E1).

 %1
alter_gram(Eqn & Eqns, New_eqn & New_eqns) :- !,
        change(Eqn, New_eqn), !,                        % import.
        alter_gram(Eqns, New_eqns).
 %2
alter_gram(Eqn, New_eqn) :- change(Eqn, New_eqn).       % import.

 %1     join an equation to a sequence.
join_first(Eqns, Eqn, Eqn & Eqns).

        /* *********************************************************** */



        /* *********************************************************** */
        /*      FILE    : gram_wrds.pl                                 */
        /*      PURPOSE : adds literals in grammar rules to the lexicon */
        /* *********************************************************** */

        /* E X P O R T S */

:- module(gram_wrds, [
                add_grammar_words/0
                ]).

        /* P R E D I C A T E S */
 %1
add_grammar_words :-
        retract(temp:gram_word(W)),
        add_wrd(W),
        add_grammar_words.
 %2
add_grammar_words :- !.

 %1
add_wrd(W) :- lookup:dict(grammar, W, _, _).
 %2
add_wrd(W) :-
        retract(lookup:dict(not_in_grammar, W, Chrs, Entries)),
        asserta(lookup:dict(grammar, W, Chrs, Entries)).
 %3
add_wrd(W) :-
        name(W, Chrs),
```

```
        asserta(lookup:dict(grammar, W, Chrs, [])).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : graphic.pl                                     */
        /*      PURPOSE : simple line drawing.                           */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(graphic, [
                line/3
                ]).

        /* I M P O R T S */

        % <none>

        /* P R E D I C A T E S */

        % line(+Char, +N, +N1).
        % draws a line of +Char's from column +N to +N1, very simple but
        % very popular with others.
 %1
line(Char, N, N1) :-
        N2 is N1 - N,                      % line length.
        name(Char, [Ascii]), format('~N~*+~*t~*+~n', [N, Ascii, N2]) -> fail.
 %2     fails to unwind and give up space & trail.
line(_, _, _) :- !.

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : greater_than.pl                                */
        /*      PURPOSE : defines the 'greater than' operator used on    */
        /*                trees, lists etc.                              */
        /* ************************************************************ */

        /*  E X P O R T S */

:- module(greater_than, [
                greater_than/3
                ]).

        /* P R E D I C A T E S */

        /* greater_than(+Elem, +Elem1, +Rel).
           test to see if tree element 'Elem' is 'greater than' tree
           element 'Elem1' according to the relationship 'Rel' which
           is described in 'insert_avl/4' above. Note that we must
           not instantiate the node holders in 'Rel' with the values
           of 'Elem' and 'Elem1' as this would render 'Rel' useless
           for further tests. */
 %1
greater_than(Elem, Elem, _) :- !, fail. % elements must not be the same.
 %2
greater_than(Elem, Elem1, gt(Pat, Pat1, Rel)) :-
        \+ is_gt(Elem, Elem1, gt(Pat1, Pat, Rel)).
```

```
        /* is_gt(+Elem, +Elem1, +Rel).
           instantiate the variables in 'Rel' which are taken as the tree
           nodes in the condition, which is also in 'Rel', and then
           execute the condition to find if 'Elem' is 'greater than'
           'Elem1'. */
 %1
is_gt(Elem, Elem1, gt(Elem1, Elem, Condition)) :- !, \+ call(Condition).

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : lex_eqns.pl                                   */
        /*      PURPOSE : produce Prolog versions of lexical equations  */
        /*                which become actual F-structures themselves.  */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(lex_eqns, [
                alter/15
                ]).

        /* I M P O R T S */

:- use_module(fs_basics, [
                insert_fs/3,
                delete_fs/3
                ]).

:- use_module(desig, [
                designator/1
                ]).

:-use_module(fs_functs, [
                delete_val/5,   % (+F, +Fs, -Fs1, +Ptrs, -Val).
                add_to_funct/6, % (+F, +Feat=Val, +Fs, -Fs1, +Ptrs, -Ptrs1).
                add_constraint/3,
                def_equal/6,    % (+F-Subf, +Subf1, +Fs, -Fs1, +Ptrs, -Ptrs1).
                new_val/6       % (+F, +Val, +Fs, -Fs1, +Ptrs, -Ptrs1).
                ]).

:- use_module(type_system, [
                type/1
                ]).

        /* P R E D I C A T E S */

        % alter(+Equation, +F_structure, -F_structure1, -Controllee,
        %               -Controllee_f_structure, -Pred, -Sem, +Types,
        %                       -Types1, -Quantifier, -Quant_var,
 %1                             +Ptrs, -Ptrs1, +Functions, -Functions1).
alter((up form) = F, Fs, Fs1, _, _, [], _, Types, Types,
                        _, _, Ptrs, Ptrs, Functs, Functs) :- !,
        insert_fs(form = atom F, Fs, Fs1).
 %2
alter(up = controllee sub S, Fs, Fs, [controllee(S, Clee_fs)],
        Clee_fs, _, _, Types, Types, _, _, Ptrs, Ptrs, Functs, Functs) :- !.
 %3
alter((up pred) = P, Fs, Fs, _, _, P, _,
                        Types, Types, _, _, Ptrs, Ptrs, Functs, Functs) :- !.
 %4
alter((up sem) = S, Fs, Fs, _, _, _, S,
                        Types, Types, _, _, Ptrs, Ptrs, Functs, Functs) :- !.
 %5
alter((up domain) = D, Fs, Fs, _, _, _, _, _,
                        domain = D, _, _, Ptrs, Ptrs, Functs, Functs) :- !,
```

```
        ( type(D), ! |
                format('~w~w~w~n', ['domain type ', D, ' unknown']), fail ).
 %6
alter((up quantifier) = Q, Fs, Fs, _, _, _, _,
                        Types, Types, Q, _, Ptrs, Ptrs, Functs, Functs) :- !.
 %7
alter((up F - Subf - domain = D), Fs, Fs1, _, _, _, _,
                        Types, Types, _, _, Ptrs, Ptrs1, Functs, Functs) :- !,
        ( type(D), ! |
                format('~w~w~w~n', ['domain type ', D, ' unknown']), fail),
        delete_val(F, Fs, Fs2, Ptrs, I ind Slots^Quant^Sem^Qvar^Subfs),
        new_slot(Subf:D, Slots, Slots1),
        new_val(F, I ind Slots1^Quant^Sem^Qvar^Subfs, Fs2, Fs1, Ptrs, Ptrs1).
 %8
alter((up quant-domain) = D, Fs, Fs, _, _, _, _,
                Types, Types, _, _:D, Ptrs, Ptrs, Functs, Functs) :- !,
        ( type(D), ! |
                format('~w~w~w~n', ['domain type ', D, ' unknown']), fail ).
 %9
alter((up F - domain) = D, Fs, Fs, _, _, _, _, Types,
                        [F:D|Types], _, _, Ptrs, Ptrs, Functs, Functs1) :- !,
        ( type(D), ! |
                format('~w~w~w~n', ['domain type ', D, ' unknown']), fail ),
        new_slot_funct(F:D, Functs, Functs1).
 %10
alter((up Funct - Feat) c Value, Fs, Fs1, _, _, _, _, Types,
                        Types, _, _, Ptrs, Ptrs1, Functs, Functs1) :- !,
        add_to_funct(Funct, Feat = val_c Value, Fs, Fs1, Ptrs, Ptrs1),
        new_slot_funct(Funct:_, Functs, Functs1).
 %11
alter((up Feat) c Value, Fs, Fs1, _, _, _, _,
                        Types, Types, _, _, Ptrs, Ptrs, Functs, Functs) :- !,
        add_constraint(Feat = val_c Value, Fs, Fs1).
 %12
alter((up F - Subf) = (up Subf1), Fs, Fs1, _, _, _, _, Types,
                        Types, _, _, Ptrs, Ptrs1, Functs, Functs1) :- !,
        def_equal(F - Subf, Subf1, Fs, Fs1, Ptrs, Ptrs1),
        new_slot_funct(F:D, Functs, Functs2),
        new_slot_funct(Subf1:D, Functs2, Functs1).
 %13
alter((up Feat) = Val, Fs, Fs1, _, _, _, _,
                        Types, Types, _, _, Ptrs, Ptrs, Functs, Functs) :- !,
        insert_fs(Feat = atom Val, Fs, Fs1).
 %14
alter((up F - Feat), Fs, Fs1, _, _, _, _,
                Types, Types, _, _, Ptrs, Ptrs1, Functs, Functs1) :- !,
        add_to_funct(F, Feat = exists, Fs, Fs1, Ptrs, Ptrs1),
        new_slot_funct(F:_, Functs, Functs1).
 %15
alter((up Feat), Fs, Fs1, _, _, _, _,
                        Types, Types, _, _, Ptrs, Ptrs, Functs, Functs) :- !,
        add_constraint(Feat = exists, Fs, Fs1).
 %16
alter((up Funct - Feat) = Val, Fs, Fs1, _, _, _, _,
                Types, Types, _, _, Ptrs, Ptrs1, Functs, Functs1) :- !,
        add_to_funct(Funct, Feat = atom Val, Fs, Fs1, Ptrs, Ptrs1),
        new_slot_funct(Funct:_, Functs, Functs1).
 %17
alter(not (up F - Feat), Fs, Fs1, _, _, _,
                _, Types, Types, _, _, Ptrs, Ptrs1, Functs, Functs) :- !,
        delete_val(F, Fs, Fs2, Ptrs,
                        Slots^Quant^Sem^Quant_var^Sub_fs),
        add_constraint(Feat = none, Sub_fs, Sub_fs1),
        new_val(F, Slots^Quant^Sem^Quant_var^Sub_fs1, Fs2, Fs1, Ptrs, Ptrs1).
 %18
alter(not (up Feat), Fs, Fs1, _, _, _, _,
                        Types, Types, _, _, Ptrs, Ptrs, Functs, Functs) :- !,
```

```
        add_constraint(Feat = none, Fs, Fs1).
 %19
alter(not (up F - Feat) c Val, Fs, Fs1, _, _, _, _,
                Types, Types, _, _, Ptrs, Ptrs1, Functs, Functs1) :- !,
        add_to_funct(F, Feat = neg_c [Val], Fs, Fs1, Ptrs, Ptrs1),
        new_slot_funct(F:_, Functs, Functs1).
 %20
alter(not (up Feat) c Val, Fs, Fs1, _, _, _, _,
                        Types, Types, _, _, Ptrs, Ptrs, Functs, Functs) :- !,
        add_constraint(Feat = neg_c [Val], Fs, Fs1).


 %1
new_slot_funct(F:_type, Slots, Slots) :- \+ designator F.
 %2
new_slot_funct(F:Type, [/*slots*/], [F = _var:Type]) :- !.
 %3
new_slot_funct(F:Type, [F = Var:Type|Rest], [F = Var:Type|Rest]) :- !.
 %4
new_slot_funct(F:Type, [F1 = Var:Type|Rest], [F1 = Var:Type|Rest1]) :-
        F @> F1, !,
        new_slot_funct(F:Type, Rest, Rest1).
 %5
new_slot_funct(F:Type, Functs, [F = _:Type|Functs]).

 %1
new_slot(F:Type, S_type stype Slots, S_type stype Slots1) :-
        new_slot_funct(F:Type, Slots, Slots1).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : lex_pp.pl                                     */
        /*      PURPOSE : pre-processes a lexicon.                      */
        /*      NOTES   : slots = [ F = Var:Type, ...] or just          */
        /*                domain = Type.                                */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(lex_pp, [
                build_dictionary/2      % (+File, +Dictionary_file_stream).
                ]).

        /* I M P O R T S */

:- use_module(fast_basics, [
                fappend/3
                ]).

:- use_module(fs_basics, [
                sort_fs/2,
                sort_slots/2,
                insert_fs/3,
                delete_fs/3
                ]).

:- use_module(desig, [
                designator/1
                ]).

:- use_module(new_info, [
                empty_info/1
                ]).
```

```
:- use_module(lex_eqns, [
                alter/15
                ]).

:- use_module(counters, [
                new_counter/1,
                clear_counter/1
                ]).

        /* P R E D I C A T E S */

        /* build_dictionary(+file, +stream). */
%1
build_dictionary(File, Stream) :-
        format('~*| ~w ~n ~*| ~w ~w ~2n',
                [5, 'Starting to pre-process dictionary', 5,
                                        'Dictionary file : ', File]),
        repeat,
                read(Stream, Entry), garbage_collect,
                build(Entry),
        !.
%2
build_dictionary(File, _stream) :-
        format('~*| ~w ~w ~2n',
                        [5, 'Unknown error in dictionary file : ', File]).

%1
build(end_of_file) :-
        format('~n ~*| ~w ~2n', [5, 'Finished pre-processing dictionary']).
%2
build(Word ~ Descriptions) :-
         ( new_counter(ptr),
           produce(Descriptions, Word_entry), !,
           name(Word, Chrs),
           clear_counter(ptr),
           prolog_flag(unknown, _, fail),
           save_entry(Word, Chrs, Word_entry),
           prolog_flag(unknown, _, trace),
           format('~*| ~w ~w ~n', [5, 'processed : ', Word])
        |
           format('~*| ~w ~w ~n', [5, 'unable to process : ', Word]) ), !,
        fail.

%1
save_entry(W, _, _) :-
        lookup:dict(not_in_grammar, W, _, _), !,
        format('~*| ~w ~w ~2n',
                        [5, 'ERROR : duplicate lexical entry of', W]),
        fail.
%2
save_entry(W, _, Entries) :-
        lookup:dict(grammar, W, _, Entries),
        Entries \== [], !,              % already entries for the word.
        format('~*| ~w ~w ~2n',
                        [5, 'ERROR : duplicate lexical entry of', W]),
        fail.
%3
save_entry(W, Chrs, Entries) :-
        retract(lookup:dict(grammar, W, _, [])), !,
        asserta(lookup:dict(grammar, W, Chrs, Entries)).
%4
save_entry(Word, Chrs, Entries) :-
        asserta(lookup:dict(not_in_grammar, Word, Chrs, Entries)).

%1
produce(Cat eqns E and Defs, New_defs) :- !,
```

```
        reform(Cat, E, Defs1), !,
        produce(Defs, Defs2),
        fappend(Defs1, Defs2, New_defs).
 %2
produce(Cat eqns E, New_defs) :- reform(Cat, E, New_defs).

 %1
reform(Cat, Eqns or Eqns1, [New_defs|Rest]) :- !,
        make(Cat, Eqns, New_defs),
        reform(Cat, Eqns1, Rest), !.
 %2
reform(Cat, Eqns, [New_def]) :- make(Cat, Eqns, New_def).


 %1     functs is slots list for non-pred/non-sem f-structures.
make(Cat, Eqns, [Cat, I ind
                        Slots^Quant^Sem^(Quant_var:T)^Fs^Clee glob Ptrs1]) :-
        order(Eqns, Eqns1),
        create(Eqns1, [/*fs*/], Fs1, Clee, Clee_info, Pred, Sem1,
                [/*types*/], Types, Quant, (Quant_var:T), [], Ptrs,
                                        [/*functs*/], Functs1),
        fill(Functs1, Slots, Sem1, Sem, Pred, Types, Fs1, Fs2, (Quant_var:T)),
        remove_nums(Ptrs, Fs2, Fs, Ptrs1),
        Clee_info = I ind Slots^Quant^Sem^(Quant_var:T)^Fs. % clee value.


 %1
remove_nums([], Fs, Fs, []) :- !.
 %2
remove_nums([Num = fs Val|R], Fs, Fs1, [Var = fs Val|R1]) :-
        remove_all(Num, Var, Fs, Fs2),
        remove_nums(R, Fs2, Fs1, R1).

 %1
remove_all(_, _, [], []) :- !.
 %2
remove_all(Num, Var, [F = fs ptr Num1|R], [F = fs ptr Var|R1]) :-
        Num1 == Num, !,
        remove_all(Num, Var, R, R1).
 %3
remove_all(Num, Var, [F = fs Info^Fs|R], [F = fs Info^Fs1|R1]) :-
        remove_all(Num, Var, Fs, Fs1),
        remove_all(Num, Var, R, R1).
 %4
remove_all(Num, Var, [F = Val|R], [F = Val|R1]) :-
        remove_all(Num, Var, R, R1).


        % put functional control equations last so that the
        % function to be passed is fully built when the equation is
 %1       executed.
order((up F - Subf) = (up Subf) & Eqns, Eqns1) :- !,
                make_last((up F - Subf) = (up Subf), Eqns, Eqns1).
 %2
order(Eqn & Eqns, Eqn & Eqns1) :- !, order(Eqns, Eqns1).
 %3
order(Eqn, Eqn).

 %1
make_last(Eqn, Eqn1 & Eqns, Eqn1 & Eqns1) :- !,
        make_last(Eqn, Eqns, Eqns1).
 %2
make_last(Eqn, Eqn1, Eqn1 & Eqn).

 %1
create(Eqn & Eqns, Fs, Fs1, Clee, Clee_var, Pred, Sem, Types, Types1,
                        Quant, Var, Ptrs, Ptrs1, Functs, Functs1) :- !,
```

```
        alter(Eqn, Fs, Fs2, Clee, Clee_var, Pred, Sem, Types, Types2,
                                Quant, Var, Ptrs, Ptrs2, Functs, Functs2),
        create(Eqns, Fs2, Fs1, Clee, Clee_var, Pred, Sem, Types2, Types1,
                                Quant, Var, Ptrs2, Ptrs1, Functs2, Functs1).
%2
create(Eqn, Fs, Fs1, Clee, Clee_var, Pred, Sem, Types, Types1,
                                Quant, Var, Ptrs, Ptrs1, Functs, Functs1) :-
        alter(Eqn, Fs, Fs1, Clee, Clee_var, Pred, Sem, Types, Types1,
                                Quant, Var, Ptrs, Ptrs1, Functs, Functs1),
        defaults(Clee, Pred, Sem, Quant).


%1      install default values on anything which may not have a value.
defaults(Clee, Pred, Sem, Quant) :-
         ( Clee = [/*null*/] | nonvar(Clee) ),
         ( Pred = [/*null*/] | nonvar(Pred) ),
         ( Sem = [/*null*/] | nonvar(Sem) ),
         ( Quant = [/*null*/] | nonvar(Quant), ! ).


%1      no semantic component.
fill(Functs, part stype Functs, [], [], [], _, Fs, Fs, _) :- !.
%2
fill(Functs, part stype Slots, Sem, Sem1, [], _, Fs, Fs, Quant_var) :-
        \+ Sem == [],
        make_part(Sem, Sem1, Functs, Slots, Quant_var).

%3      semantic component implicit in pred.
fill(_, full stype Slots, [/*sem*/], Sem1, Pred, Types, Fs, Fs1, _) :- !,
        get_sem(Pred, Slots1, Fs, Fs1, Sem1),
        type(Slots1, Slots, Types).
%4
fill(_, full stype Slots, Sem, Sem1, Pred, Types, Fs, Fs1, Quant_var) :-
        get_slots(Pred, Slots1, Fs, Fs1),
        make_full(Sem, Sem1, Slots1, Slots2, Quant_var),
        type(Slots2, Slots, Types).

%1      semantic in sem .
get_slots(P >> [(up F),(up F1),(up F2)], Slots, Fs, Fs1) :- !,
        sort_slots([F = _:_, F1 = _:_, F2 = _:_], Slots), % sort the slots.
        Prd =.. [P,F,F1,F2],
        insert_fs(pred = atom Prd, Fs, Fs1).
%2
get_slots(P >> [(up F),(up F1)], Slots, Fs, Fs1) :- !,
        sort_slots([F = _:_, F1 = _:_], Slots),
        Prd =.. [P,F,F1],
        insert_fs(pred = atom Prd, Fs, Fs1).
%3
get_slots(P >> [(up F)], [F = _:_], Fs, Fs1) :- !,
        Prd =.. [P,F],
        insert_fs(pred = atom Prd, Fs, Fs1).
%4
get_slots(P, [], Fs, Fs1) :- insert_fs(pred = atom P, Fs, Fs1).


        % semantic in pred ('be', 'do', 'have', 'with').
%1       'be' (three cases)
get_sem(P >> [(up F)=(up F1)], Slots, Fs, Fs1, equate(F, F1)) :-
        sort_slots([F = V:T, F1 = V:T], Slots),
        Prd =.. [P,F,F1],
        insert_fs(pred = atom Prd, Fs, Fs1).
%2
get_sem(attribute >> [up F]-[(up F1)], Slots, Fs, Fs1, passto(F)) :-
        sort_slots([F = _:_, F1 = _:_], Slots),
        Prd =..[attribute_be,F,F1],
        insert_fs(pred = atom Prd, Fs, Fs1).
%3
```

```
get_sem(attribute >> [up F]-[(up F1),(up F2)], Slots, Fs, Fs1, passto(F)) :-
        sort_slots([F = _:_, F1 = _:_, F2 = _:_], Slots),
        Prd =..[attribute_be,F,F1,F2],
        insert_fs(pred = atom Prd, Fs, Fs1).
 %4
get_sem(exists >> [up F]-[(up F1)], Slots, Fs, Fs1, null(F1)) :-
        sort_slots([F = V:_, F1 = V:_], Slots),
        Prd =..[existential_be,F,F1],
        insert_fs(pred = atom Prd, Fs, Fs1).

 %5       other aux verbs 'have', 'do' & 'with'.
get_sem(P >> [up F]-[(up F1)], Slots, Fs, Fs1, passto(F)) :-
        sort_slots([F = _:_, F1 = _:_], Slots),
        Prd =..[P,F,F1],
        insert_fs(pred = atom Prd, Fs, Fs1).
 %6
get_sem(P >> [up F]-[(up F1),(up F2)], Slots, Fs, Fs1, passto(F)) :-
        sort_slots([F = _:_, F1 = _:_, F2 = _:_], Slots),
        Prd =..[P,F,F1,F2],
        insert_fs(pred = atom Prd, Fs, Fs1).

 %1
make_part(Sem, Sem1, Slots, Slots1, Quant_var) :-
        remove(Sem, Vars, Sem1),
        equate_part(Vars, Slots, Slots1, Quant_var).
 %2
make_full(Sem, Sem1, Slots, Slots1, Quant_var) :-
        remove(Sem, Vars, Sem1),
        equate_all(Vars, Slots, Slots1, Quant_var).

 %1
remove(Sem, Vars, New_sem) :-
        Sem =.. [P|Args],
        change(Args, Vars1, New_args),
        sort_slots(Vars1, Vars),
        New_sem =.. [P|New_args].

 %1
change([F, F1, F2], [F=Var, F1=Var1, F2=Var2], [Var, Var1, Var2]).
 %2
change([F, F1], [F=Var, F1=Var1], [Var, Var1]).
 %3
change([F], [F=Var], [Var]).
 %4
change([], [], []).

 %1
equate_part([], [], [], _) :- !.
 %2
equate_part([], Slots, Slots, _quant_var) :- !.
 %3
equate_part(Rest, [], Slots, Quant_var) :- !,
        equate_part_rest(Rest, Slots, Quant_var).
 %4
equate_part([quant = Var|Rest], Functs, Slots, Var) :- !,
        equate_part(Rest, Functs, Slots, _var).
 %5
equate_part([F = Var|Rest], [F = Var:T|Rest1], [F = Var:T|Rest2], Quant) :- !,
        equate_part(Rest, Rest1, Rest2, Quant).
 %6
equate_part([F = Var|Rest], [F1 = V1:T1|Rest1],
                                        [F = Var:_|Rest2], Q_var) :- !,
        F1 @> F, !,
        equate_part(Rest, [F1 = V1:T1|Rest1], Rest2, Q_var).
 %7
equate_part([F = V|Rest], [F1 = V1:T1|Rest1], [F1 = V1:T1|Rest2], Q_var) :- !,
        equate_part([F = V|Rest], Rest1, Rest2, Q_var).
```

```
 %1
equate_part_rest([], [], _q_var) :- !.
 %2
equate_part_rest([quant = Q_var|Rest], Slots, Q_var:_) :- !,
        equate_part_rest(Rest, Slots, _).
 %3
equate_part_rest([F = Var|Rest], [F = Var:_type|Rest1], Q_var) :- !,
        equate_part_rest(Rest, Rest1, Q_var).

 %1
equate_all([], [], [], _) :- !.
 %2
equate_all([quant=Var|Rest], Slots, Slots1, Var:_) :- !,
        equate_all(Rest, Slots, Slots1, _).
 %3
equate_all([F=Var|Rest], [F=Var:_|Rest1], [F=Var:_|Rest2], Quant) :-
        equate_all(Rest, Rest1, Rest2, Quant).

 %1
type([], domain = D, domain = D) :- !. % no functions just a type.
 %2
type([], [], _) :- !.
 %3
type([F = Var:Type|Rest], [F = Var:Type|Rest1], Types) :-
        find_type(F, Type, Types),
        type(Rest,Rest1, Types).

 %1
find_type(_, _, []).    % no type declared.
 %2
find_type(F, T, [F:T|_]):- !.
 %3
find_type(F, T, [_|Rest]) :- find_type(F, T, Rest).

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : lookup.pl                                     */
        /*      PURPOSE : defines predicate for dictionary access.      */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(lookup, [
                lookup/1,
                lookup/3
                ]).

        /* D Y N A M I C S */

:- dynamic dict/4.      % put into this module by pre_process_dict.

        /* I M P O R T S */

:- use_module(spelling, [
                spell/2
                ]).

:- use_module(fast_basics, [
                fappend/3
                ]).

        /* P R E D I C A T E S */
```

```
        /* lookup(+Chrs).
           test to see if there is an entry for a word represented by
           the list of characters 'Chrs' in the dictionary. */
 %1
lookup(Word_chrs) :- dict(_type, _word, Word_chrs, _word_entry), !.


        /* lookup(+single_multiple, +Word, -Entries).
           lookup the entries for word 'Word' in the dictionary and
           retrieve the information 'Entries' for each of these. */
 %1
lookup(single_word, Word, [[Type, Word, Defs]]) :-
        dict(Type, Word, _chrs, Defs).
 %2
lookup(joined_word, Word, [[Type, Word, Defs]]) :-
        dict(Type, Word, _chrs, Defs).
 %3
lookup(single_word, Word, Entries) :-
        spell(Word, New_words), !,
        now_lookup(New_words, Word, Entries).
 %4
lookup(multiple_words, [Word, Word1], Entries) :-
        name(Word, Chrs), name(Word1, Chrs1),
        fappend(Chrs, [32|Chrs1], Chrs2),
        name(Word2, Chrs2),
        ( lookup(joined_word, Word2, Entries) | Entries = [] ).

 %1
now_lookup([], W, _) :-
        format('~*| ~w ~w ~n ~*| ~w ~2n', [5, 'Sorry the word : ', W, 5,
                    'is either mis-spelt or not in the system dictionary']).
 %2
now_lookup(New_words, _, Entries) :- find_words(New_words, Entries).

 %1
find_words([], []) :- !.
 %2
find_words([Word|Rest], [[Type, Word, Defs]|Rest1]) :-
        dict(Type, Word, _chrs, Defs),
        find_words(Rest, Rest1).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : make_links.pl                                  */
        /*      PURPOSE : construct top-down predictive link relation    */
        /*                for use by the parser.                         */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(make_links, [
                make_links/0,    % produce the link relation from a grammar.
                break_links/0,   % remove the link relation.
                links/2          % test the link relation.
                ]).

        /* I M P O R T S */

:- use_module(gram_pp, [
                '--->'/2
                ]).

:- use_module(library(findall), [
                findall/3
```

```
                ]).

:- use_module(fast_basics, [
                fmember/2,
                fappend/3
                ]).

        /*  D Y N A M I C S */

:- dynamic
        immed_links/2,  % immediate right-hand children.
        links/2.        % all right-hand children

        /*  P R E D I C A T E S */
 %1
break_links :- retractall(links(_, _)).

 %1
make_links :-
        linkage_cats(C),
        immediate_links(C),
        extended_links,
        assertz(links(Cat, [Cat])).

 %1
linkage_cats(Cats) :-
        findall(Cat,
                (  (_ ---> Cat eqns _, _) |
                   (_ ---> Cat eqns _) |
                   (_ ---> bnd Cat eqns _, _) |
                   (_ ---> bnd Cat eqns _) |
                   (_ ---> lnk Cat eqns _, _) |
                   (_ ---> lnk Cat eqns _) |
                   (_ ---> _ * _ eqns _, _ * _ eqns _, Cat eqns _, _) |
                   (_ ---> _ * _ eqns _, _ * _ eqns _, Cat eqns _) |
                   (_ ---> _ * _ eqns _, Cat * _ eqns _, _) |
                   (_ ---> _ * _ eqns _, Cat * _ eqns _) |
                   (_ ---> Cat * _ eqns _, _) |
                   (_ ---> Cat * _ eqns _) |
                   (_ ---> _ * eqns _, _ * eqns _, Cat eqns _, _) |
                   (_ ---> _ * eqns _, _ * eqns _, Cat eqns _) |
                   (_ ---> _ * eqns _, Cat * eqns _, _) |
                   (_ ---> _ * eqns _, Cat * eqns _) |
                   (_ ---> Cat * eqns _, _) |
                   (_ ---> Cat * eqns _) |
                   (_ ---> conj Cat eqns _) |
                   (_ ---> conj Cat eqns _, _) |
                   (_ ---> Cat, _), atom(Cat) |
                   (_ ---> Cat), atom(Cat)
                ),
                                        Cats1),
        remove_dups(Cats1, [], Cats).

 %1
remove_dups([], L, L) :- !.
 %2
remove_dups([First|Rest], List, List1) :-
        fmember(First, List) ->
                remove_dups(Rest, List, List1)
                |
                remove_dups(Rest, [First|List], List1).

 %1
immediate_links([]) :- !.
 %2
immediate_links([Cat|Rest]) :-
        findall(Head,
```

```
                        (  (Head ---> Cat eqns _, _) |
                           (Head ---> Cat eqns _) |
                           (Head ---> bnd Cat eqns _, _) |
                           (Head ---> bnd Cat eqns _) |
                           (Head ---> lnk Cat eqns _, _) |
                           (Head ---> lnk Cat eqns _) |
                           (Head ---> _ * _ eqns _, _ * _ eqns _, Cat eqns _, _) |
                           (Head ---> _ * _ eqns _, _ * _ eqns _, Cat eqns _) |
                           (Head ---> _ * _ eqns _, Cat * _ eqns _, _) |
                           (Head ---> _ * _ eqns _, Cat * _ eqns _) |
                           (Head ---> Cat * _ eqns _, _) |
                           (Head ---> Cat * _ eqns _) |
                           (Head ---> _ * eqns _, _ * eqns _, Cat eqns _, _) |
                           (Head ---> _ * eqns _, _ * eqns _, Cat eqns _) |
                           (Head ---> _ * eqns _, Cat * eqns _, _) |
                           (Head ---> _ * eqns _, Cat * eqns _) |
                           (Head ---> Cat * eqns _, _) |
                           (Head ---> Cat * eqns _) |
                           (Head ---> conj Cat eqns _) |
                           (Head ---> conj Cat eqns _, _) |
                           (Head ---> Cat, _), atom(Cat) |
                           (Head ---> Cat), atom(Cat)
                        ),
                                                    Heads1),
        remove_dups([Cat|Heads1], [], Heads),
        assert(immed_links(Cat, Heads)),
        immediate_links(Rest).


extended_links :-
        immed_links(C, Cats),
        \+ links(C, _),                 % not extended links for this cat.
        other_links(Cats, Cats, Cats1),
        asserta((links(C, Cats1) :- !)), fail.

extended_links :- retractall(immed_links(_, _)).


other_links(Links, [], Links) :- !.

other_links(Links, [C|Rest], Links1) :-
        immed_links(C, Links2), !,
        new_members(Links, Links2, New),
        fappend(New, Links, Links_new),
        other_links(Links_new, Rest, Links3),
        other_links(Links3, New, Links1).

other_links(Links, [_|Rest], Links1) :- other_links(Links, Rest, Links1).


new_members(_, [], []) :- !.

new_members(List, [First|Rest], New) :-
        fmember(First, List) ->
                 new_members(List, Rest, New)
                 |
                 ( New = [First|Rest1], new_members(List, Rest, Rest1) ).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : make_static.pl                                */
        /*      PURPOSE : defines the top_level predicate of UNLID.     */
        /* ************************************************************ */
```

```
        /* E X P O R T S */

:- module(make_static, [
                statisize/3
                ]).

        /* I M P O R T S */

        % none.

        /* P R E D I C A T E S */

statisize(Mod, Preds, File) :-
        open(File, write, Comp_stream),
        write_out(Comp_stream, Preds),
        close(Comp_stream),
        compile(Mod:File).


write_out(_, []) :- !.

write_out(Stream, [M:P|Preds]) :-
        functor(P, Name, Arity),
        nl(Stream), nl(Stream),
        write(Stream, '/*        Predicate : '), nl(Stream),
        write(Stream, Name),
        write(Stream, '/'),
        write(Stream, Arity), nl(Stream),
        write(Stream, '** Begin :        */'), nl(Stream),
        write_pred(Stream, M:P),
        abolish_pred(M:P),
        write_out(Stream, Preds).


write_pred(Stream, M:P) :-
        clause(M:P, Body), Body \== true,
        write_canonical(Stream, (P :- Body)),
        write(Stream, ' .'), nl(Stream),
        fail.


find_members(Description, [First|Rest], List1, [First|Rest1]) :-
        fits(Description, First), !,
        find_members(Description, Rest, List1, Rest1).
 %3
find_members(Description, [First|Rest], [First|Rest1], Membs) :-
        find_members(Description, Rest, Rest1, Membs).

 %1
fits_description(D, Thing) :- copy_term(D, Thing).

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : new_info.pl                                   */
        /*      PURPOSE : produces new information data structures for  */
        /*                f-structures.                                 */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(new_info, [
                empty_info/1,
                empty_info1/1
                ]).
```

```
        /* P R E D I C A T E S */

        /* empty_info(-Info).
           return a new structure to represent an initial information
           set in a parser edge (slots, quantifier, semantic predicate,
           quantifier variable and ordered f-structure list without
            controller and controllee lists. */

empty_info(     [/*index*/] ind
                part stype [/*slots*/]^
                [/*Quant*/]^
                [/*Sem*/]^
                _Quant_var^
                [/*Fs*/]).

empty_info1(    [/*index*/] ind
                part stype [/*slots*/]^
                [/*quantifier*/]^
                [/*semantic component*/]^
                _quantifier_variable^
                [/*f-structure*/]^
                [/*controllees*/] glob [/*pointers*/]).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : parser.pl                                     */
        /*      PURPOSE : defines bottom-up word incorporation parser   */
        /*                with top-down linking relation.               */
        /*      NOTES   : the bottom-up parser algorithm is very simple */
        /*                but somewhat complicated by the demands of    */
        /*                LFG itself. It has very low overheads. This   */
        /*                is quite important when control is built on   */
        /*                over the top of Prolog's top-down back-       */
        /*                tracking.                                     */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(parser, [
                bottom_up_parse/2,
                reached_target/3
                ]).

        /* I M P O R T S */

:- use_module(counters, [
                new_counter/1,
                next_counter/2,
                clear_counter/1
                ]).

:- use_module(fast_basics, [
                fappend/3,
                fmember/2,
                fcommon_member/2
                ]).

:- use_module(lookup, [
                lookup/3
                ]).

:- use_module(new_info, [
                empty_info1/1
```

```
                ]).

:- use_module(eval_eqns, [
                evaluate/4,
                evaluate/5
                ]).

:- use_module(wf_fs, [
                well_formed/4
                ]).

:- use_module(make_links, [
                links/2
                ]).

:- use_module(traces, [
                trace_info/2
                ]).

:- use_module(graphic, [
                line/3
                ]).

:- use_module(extend, [
                extend/4,
                end_or_new/4,
                new/1
                ]).

        /* D Y N A M I C S */

:- dynamic
        rules_which_cat/2,              % put into this module by gram_pp.
        rules_which_word/2,             % put into this module by lex_pp.
        base_edge/1,                    % the current being incorporated.
        active_edge/3.                  % active edges ending at a vertex.

        /* C O N S T A N T S */

constant(distinguished(s)).             % the grammar's distinguished symbol.

        /* P R E D I C A T E S */

        /* bottom_up_parse(+Words, -F_structure).
           parse the list of words 'Words' in a bottom up fashion and
           produce the information (including LFG f-structure) 'Info'. */

bottom_up_parse(Words, Info) :-
        new_counter(ptr), new_counter(index),
        initial(Words, 1, Num),
        constant(distinguished(Cat)),
        retract(base_edge(E)), !, elevate(E, [1,Num,Cat,Info]),  clean.


        % clean the database of dynamic predicates asserted during parsing
        % (complete and active edges and counters).
 %1
clean :- retractall(parser:base_edge(_)),
         retractall(parser:active_edge(_, _, _)),
         clear_counter(index), clear_counter(ptr).


        /* initial(+Words, -Base, +V, -V1).
           build the initial list of complete edges, which will form the
           tree base 'Base' from vertex 'V' to 'V1', from the lexical entries
           for the words in list 'Words'. */
 %1
```

```
initial([], Ev, Ev) :- !.
 %2
initial([Word,Word1|Rest], Sv, Ev1) :-
        lookup(single_word, Word, Entries), !,
        lookup(multiple_words, [Word, Word1], Entries1), !,
        Ev is Sv + 1, Ev2 is Ev + 1,
        base_entries(Entries, Sv, Ev),
        base_entries(Entries1, Sv, Ev2),
        initial([Word1|Rest], Ev, Ev1).
 %3
initial([Word|Rest], Sv, Ev1) :-
        lookup(single_word, Word, Entries), !,
        Ev is Sv + 1,
        base_entries(Entries, Sv, Ev),
        initial(Rest, Ev, Ev1).

 %1
base_entries([], _, _) :- !.
 %2
base_entries([[Type, Word, Defs]|Rest], Sv, Ev) :-
        make(Type, Word, Defs, Sv, Ev),
        base_entries(Rest, Sv, Ev).


        /* make(-Base, +Entry, +V, +V1, -Rest_base).
           produce the list of base edges 'Base' from the dictionary entry
           'Entry' for some word from vertex 'V' to vertex 'V1' and return
           the 'open end' of this list 'Rest_base' so that other base
           entries can be added to the end of the list. */

make(not_in_grammar, _word, [], _, Ev) :- !,
        empty_info1(Info),
        assertz(base_edge([Ev,Ev,e,Info])).

make(grammar, Word, [], Sv, Ev) :- !,
        assertz(base_edge([Sv,Ev,Word])),
        empty_info1(Info),
        assertz(base_edge([Ev,Ev,e,Info])).

make(Type, Word,[[Cat, [] ind St stype Slots^Quant^Sem^
                (Quant_var:T)^Fs^[/*clee*/] glob Ptrs]|Rest], Sv, Ev) :- !,
        number_ptrs(Ptrs),
        assertz(base_edge([Sv,Ev,Cat,[] ind St stype
                                Slots^Quant^Sem^(Quant_var:T)^
                                                Fs^[/*clee*/] glob Ptrs])),
        make(Type, Word, Rest, Sv, Ev).

make(Type, Word, [[Cat, Ind ind St stype Slots^Quant^Sem^
                (Quant_var:T)^Fs^CleeInfo glob Ptrs]|Rest], Sv, Ev) :-
        number_ptrs(Ptrs),
        next_counter(index,Ind),
        assertz(base_edge([Sv,Ev,Cat,Ind ind St stype Slots^Quant^
                                Sem^(Quant_var:T)^Fs^CleeInfo glob Ptrs])),
        make(Type, Word, Rest, Sv, Ev).

 %1
number_ptrs([]) :- !.
 %2
number_ptrs([N = fs _|R]) :- next_counter(ptr, N), number_ptrs(R).


        /* elevate(+Base, +Act, +Targ).
           elevate (move bottom-up) from the base of the parse tree 'Base'
           (a list of complete edges) to the required top of the tree (the
           target edge) 'Targ'. */

elevate([Sv,Ev,Cat|Rest_edge], Targ) :-
```

```
        trace_info(edge, [Sv,Ev,Cat]),
        incorporate([Sv,Ev,Cat|Rest_edge], Targ, Tf), !,
         ( finished(Tf), !
        |
            ( get_next(base_edge(Next)), !, elevate(Next, Targ)
            |
              format('~*|~w~n~*|~w~2n', [5,
                       'Parsing failure all base edges used !', 5,
                                            'Returning to top_level']),
              line('-', 10, 38), print_active, clean
            )
         ).

        % edges with head equal to the distinguished category are not
        % elevated.
get_next(Base_edge) :-
        retract(base_edge([Sv,Ev,Cat|Rest_edge])),
         ( constant(distinguished(Cat)), !, get_next(Base_edge)
        |
           Base_edge = base_edge([Sv,Ev,Cat|Rest_edge]) ).


print_active :-
        retract(active_edge(Sv,Cats,_)),
        format('~N~*|~w~w   ~w~w~n',
              [10, 'Starting vertex ', Sv, 'Next category ', Cats]),
        fail.

print_active :- line('_', 10, 38).

        % finished parsing if the target flag = yes.
finished(no) :- !, fail.                finished(yes) :- !.


incorporate([Sv,Ev,Cat|Rest], T, Tf) :-
        bagof(Edge, cat_active_edge(Sv, Cat, Edge), Actives), !,
        extend_by([Sv,Ev,Cat|Rest], Actives, T, Tf).

        % no active edges match this base edge, just invoke new rules.
incorporate(Edge, T, Tf) :- invokes(Edge, T, Tf).


cat_active_edge(Sv, Cat, Edge) :-
        active_edge(Sv, Start_cats, Edge), fmember(Cat, Start_cats).


extend_by(Edge, [/*active*/], T, Tf) :- !, invokes(Edge, T, Tf).

extend_by(Edge, [A|_], T, Tf) :- extend(A, Edge, T, Tf) -> finished(Tf), !.

extend_by(Edge, [_|R], T, Tf) :- !, extend_by(Edge, R, T, Tf).


        /* invoke any grammar rules which start with the newly completed
           edge. */

invokes([Sv,Ev,Word], T, Tf) :-
        Word rules_which_word(Start, Complete), !,
        empty_infol(Info),
        invoke_active([Sv,Ev,word_is Word,Info], Start, T, Tf), !,
         ( finished(Tf)
        |
           invoke_base([Sv,Ev,word_is Word,Info], Complete, T, Tf), ! ).

invokes([Sv,Ev,Cat,Info], T, Tf) :-
        Cat rules_which_cat(Start, Complete), !,
        invoke_active([Sv,Ev,Cat,Info], Start, T, Tf), !,
```

```
          ( finished(Tf)
          |
            invoke_base([Sv,Ev,Cat,Info], Complete, T, Tf), ! ).

invokes(_, _, _) :- !.


invoke_active(_, [], _, _) :- !.

invoke_active(Edge, [Started|_], T, Tf) :-
        now_active(Edge, Started, T, Tf) -> finished(Tf), !.

invoke_active(Edge, [_|Rest], T, Tf) :- !, invoke_active(Edge, Rest, T, Tf).


invoke_base(_, [], _, _) :- !.

invoke_base(Edge, [Completed|_], T, Tf) :-
        now_base(Edge, Completed, T, Tf), finished(Tf) -> !.

invoke_base(Edge, [_|Rest], T, Tf) :- !, invoke_base(Edge, Rest, T, Tf).


        /* add partially completed newly invoked rules to the array
           of active edges. */

now_active([Sv,Ev,word_is W,Info], [word_is W,Head,Rem], _, _) :- !,
        links(Head, Cats), !,                % all the cats head could form.
        an_active_edge_has(Cats, Sv), !,     % atleast one active edge wants it.
        new([Sv,Ev,Head,Rem,Info]), !.       % construct the new edge.

now_active([Sv,Ev,Cat,Info], [Head,(Cat * if E, Rem)], _, _) :- !,
        links(Head, Cats), !,
        an_active_edge_has(Cats, Sv), !,
        evaluate(Cat, E, Info, Info1),
        new([Sv,Ev,Head,(Cat * if E, Rem),Info1]), !.

now_active([Sv,Ev,Cat,Info], [Head,(Cat * if E)], _, _) :- !,
        links(Head, Cats), !,
        an_active_edge_has(Cats, Sv), !,
        evaluate(Cat, E, Info, Info1),
        new([Sv,Ev,Head,(Cat * if E),Info1]), !.

now_active([Sv,Ev,Cata,Info], [Head,(Cata if Ea, Catb * if Eb)], T, Tf) :- !,
        links(Head, Cats), !,
        an_active_edge_has(Cats, Sv), !,
        evaluate(Cata, Ea, Info, Info1),
        end_or_new([Sv,Ev,Head,Info1], T, Tf,
                                     [Sv,Ev,Head,(Catb * if Eb),Info1]), !.

now_active([Sv,Ev,Cat,Info], [Head,(Cat if E, Rem)], _, _) :- !,
        links(Head, Cats), !,
        an_active_edge_has(Cats, Sv), !,
        evaluate(Cat, E, Info, Info1),
        new([Sv,Ev,Head,Rem,Info1]), !.


        /* add completed newly invoked rules to the front of the
           completed (base) list of edges. */

now_base([Sv,Ev,word_is W,Info], [word_is W, Head], T, Tf) :- !,
        links(Head, Cats), !,
        an_active_edge_has(Cats, Sv), !,
         ( reached_target([Sv,Ev,Head,Info], T, Tf), !
        |
           asserta(base_edge([Sv,Ev,Head,Info]))).
```

```
now_base([Sv,Ev,Cat,Info], [Head,(Cat * if E)], T, Tf) :- !,
        links(Head, Cats), !,
        an_active_edge_has(Cats, Sv), !,
        evaluate(Cat, E, Info, Info1),
         ( reached_target([Sv,Ev,Head,Info1], T, Tf), !
        |
           asserta(base_edge([Sv,Ev,Head,Info1]))).

now_base([Sv,Ev,Cat,Info], [Head,(Cat if E)], T, Tf) :- !,
        links(Head, Cats), !,
        an_active_edge_has(Cats, Sv), !,
        evaluate(Cat, E, Info, Info1),
         ( reached_target([Sv,Ev,Head,Info1], T, Tf), !
        |
           asserta(base_edge([Sv,Ev,Head,Info1]))).


reached_target([V,V1,Cat,Info glob Ptrs], [V,V1,Cat,Info1], yes) :-
        well_formed(Info, Info1, Ptrs, _), !.

reached_target(_, _, _) :- fail.


an_active_edge_has(Cats, 1) :- !,
        constant(distinguished(Cat)),           % virtual edge at vertex 1.
        fmember(Cat, Cats).

an_active_edge_has(Cats, Ev) :-
        active_edge(Ev, Cats1 , _),
        fcommon_member(Cats, Cats1), !.

an_active_edge_has(_, _) :- !, fail.

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : plan_query.pl                                 */
        /*      PURPOSE : plan queries to the database.                 */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(plan_query, [
                plan_query/2
                ]).

        /* I M P O R T S */

:- use_module(library(occurs), [
                contains_var/2
                ]).

:- use_module(db_meta, [
                db_solutions/3
                ]).

:- use_module(fast_basics, [
                fappend/3,
                fmember/2,
                flength/2
                ]).

        /* P R E D I C A T E S */
%1
plan_query(Preds, Preds1) :- plan(Preds, Preds1, _, _).
```

```
 %1
plan(wh(Ans, Preds), wh(Ans, Preds1), Inf, Min) :-
        plan(Preds, Preds1, Inf, Min).
 %2
plan(y_n(Preds), y_n(Preds1), Inf, Min) :-
        plan(Preds, Preds1, Inf, Min).
 %3
plan(numberof(Ans, Preds), numberof(Ans, Preds1), Inf, Min) :-
        plan(Preds, Preds1, Inf, Min).
 %4
plan(Pred & Preds, Query1, Inf, Min) :-
        var_list(Pred & Preds, [Vars|Vars_rest]),
         ( partition([Vars|Vars_rest], Pred & Preds, Inf, Min, Query1), !
        |
           cost(Pred, Vars, Pred1, Cost),
           cost_rest(Cost, Min, 2, 1, Num, Preds, New_preds,
                         [], Costs, Vars_rest, Vars, Pred_vars1),
           plan_rest(Pred1 & New_preds, Pred & Preds, [Vars|Vars_rest],
                         Num, [Cost|Costs], Pred_vars1, Inf, Min, Query1)
         ).
 %5
plan(setof(Var, Preds, Set), setof(Var, Preds1, Set),Inf, Min) :-
        plan(Preds, Preds1, Inf, Min).
 %6
plan(\+ { Preds }, \+ { Preds1 }, Inf, Min) :- plan(Preds, Preds1, Inf, Min).
 %7
plan(Pred, Pred1, Inf, Min) :- !,
        var_list(Pred, [Vars]), cost(Pred, Vars, Pred1, Min),
        infinite(Min, Inf).


 %1
plan_sub(numberof(Ans, Preds), [_|Vars], numberof(Ans, Preds1), Inf, Min) :-
        plan_sub(Preds, Vars, Preds1, Inf, Min).
 %2
plan_sub(Pred & Preds, [Vars|Vars_rest], Query1, Inf, Min) :-
        cost(Pred, Vars, Pred1, Cost),
        cost_rest(Cost, Min, 2, 1, Num, Preds, New_preds,
                        [], Costs, Vars_rest, Vars, Pred_vars1),
        plan_rest(Pred1 & New_preds, Pred & Preds, [Vars|Vars_rest],
                        Num, [Cost|Costs], Pred_vars1, Inf, Min, Query1).
 %3
plan_sub(setof(Var, Preds, Set), [_|Vars],
                                setof(Var, Preds1, Set), Inf, Min) :-
        plan_sub(Preds, Vars, Preds1, Inf, Min).
 %4
plan_sub(\+ { Preds }, Vars, \+ { Preds1 }, Inf, Min) :-
        plan_sub(Preds, Vars, Preds1, Inf, Min).
 %5
plan_sub(Pred, [Vars], Pred1, Inf, Min) :- !,
        cost(Pred, Vars, Pred1, Min), infinite(Min, Inf).


        % plan_rest(+Query, +Old_query, +Var_lists, +Lowest_pred_num,
        %                          +Costs, +Pred_vars1, -Inf, +Min, -Query1).
 %1
plan_rest(setof(V,Preds,Set), _, _, _, [Min], _pred_vars,
                                        Inf, Min, setof(V,Preds,Set)) :- !,
        infinite(Min, Inf).
 %2
plan_rest(numberof(V,Preds,N), _, _, _, [Min], _pred_vars,
                                        Inf, Min, numberof(V,Preds,N)) :- !,
        infinite(Min, Inf).
 %3
plan_rest(\+{Preds}, _, _, _, [Min], _pred_vars, Inf, Min, \+{Preds}) :- !,
        infinite(Min, Inf).
```

```
%4
plan_rest(Pred & Query, Old_query, Vars, Num, Costs, Pred_vars,
                              Inf, Min, Pred_low & Query_plan1) :- !,
        infinite(Min, Inf),
        ( Num = gen:Num1 | Num1 = Num),
        get_lowest(Num1, Costs, Costs1, Vars, Vars_old, Pred_vars, Vars_new,
                              Pred & Query, Old_query, Pred_low, Query2),
         ( partition(Vars_new, Query2, Inf, _Min, Query_plan1), !
         |
           new_costs(Vars_old, Vars_new, Costs1, Costs2,
                              Query2, Query3, Num2, Min2, Pred_vars2),
           plan_rest(Query3, Query2, Vars_new, Num2, Costs2,
                                     Pred_vars2, Inf, Min2, Query_plan1)
         ).
%5      only one pred.
plan_rest(Pred, _, _, _, [Min], Pred_vars, Inf, Min, Pred1) :- !,
         ( all_fixed(Pred_vars), !, Pred1 = [Pred] | Pred1 = Pred),
        infinite(Min, Inf).


        % new_costs(+Old_varlists, +New_varlists, +Costs, -Costs1,
        %                 +Preds, -Preds1, -Num, -Inf, -Min, -Next_inst_vars).
%1
new_costs([_|Old_rest], [Pvars|New_rest], [_|Cr], [C1|Cr1],
                setof(V, P, St) & Ps, setof(V, P1, St) & Ps1,
                                                   Num, Min, Vars1) :- !,
        cost(setof(V, P, St), Pvars, setof(V, P1, St), C1),
        rest_new_costs(Old_rest, New_rest, Cr, Cr1, Ps, Ps1,
                                     C1, Min, 2, 1, Num, Pvars, Vars1).
%2
new_costs([_|Old_rest], [Pvars|New_rest], [_|Cr], [C1|Cr1],
                numberof(V, P, St) & Ps, numberof(V, P1, St) & Ps1,
                                                   Num, Min, Vars1) :- !,
        cost(numberof(V, P, St), Pvars, numberof(V, P1, St), C1),
        rest_new_costs(Old_rest, New_rest, Cr, Cr1, Ps, Ps1,
                                     C1, Min, 2, 1, Num, Pvars, Vars1).
%3      predicate variable state is the same.
new_costs([Pvars|Old_rest], [Pvars|New_rest], [C|Cr], [C|Cr1],
                              P & Ps, P & Ps1, Num, Min, Vars1) :- !,
        rest_new_costs(Old_rest, New_rest, Cr, Cr1, Ps, Ps1,
                                     C, Min, 2, 1, Num, Pvars, Vars1).
%4      variables have changed state so re-cost predicate.
new_costs([_|Old_rest], [Pvars1|New_rest], [_|Cr], [C1|Cr1],
                              P & Ps, P1 & Ps1, Num, Min, Vars1) :- !,
        cost(P, Pvars1, P1, C1),
        rest_new_costs(Old_rest, New_rest, Cr, Cr1, Ps, Ps1,
                                     C1, Min, 2, 1, Num, Pvars1, Vars1).
%5
new_costs([Pvars], [Pvars], [C], [C], P, P, 1, C, Pvars) :- !.
%6
new_costs([_], [Pvars], [_], [C1],
                     setof(V, P, St), setof(V, P1, St), 1, C1, Pvars) :- !,
        cost(setof(V, P, St), Pvars, setof(V, P1, St), C1).
%7
new_costs([_], [Pvars], [_], [C1], numberof(V, P, St),
                              numberof(V, P1, St), 1, C1, Pvars) :- !,
        cost(numberof(V, P, St), Pvars, numberof(V, P1, St), C1).
%8
new_costs([_], [Pvars1], [_], [C1], P, P1, 1, C1, Pvars1) :- !,
        cost(P, Pvars1, P1, C1).


        % rest_new_costs(+Old_vars, +New_vars, +Costs, -Costs1,
        %                         +Preds, -Preds1, -Inf, +C_lowest_sofar,
        %                         -Min, +Num_next_pred, +Num_lowest_sofar,
        %                         -Num_final_lowest, Vars2, Vars1).
%1
```

```
rest_new_costs([_|Old], [Pvars1|New], [_|Cr], [C1|Cr1],
                setof(V, P, St) & Ps, setof(V, P1, St) & Ps1,
                              Min, Min1, N, Num, Num1, Vars, Vars1) :- !,
        cost(setof(V, P, St), Pvars1, setof(V, P1, St), C1),
        lowest_cost(Min, C1, N, Pvars1, Min2, Num, Num2, Vars, Vars2),
        N1 is N + 1,
        rest_new_costs(Old, New, Cr, Cr1, Ps, Ps1, Min2, Min1,
                                      N1, Num2, Num1, Vars2, Vars1).
%2
rest_new_costs([_|Old], [Pvars1|New], [_|Cr], [C1|Cr1],
                numberof(V, P, St) & Ps, numberof(V, P1, St) & Ps1,
                              Min, Min1, N, Num, Num1, Vars, Vars1) :- !,
        cost(numberof(V, P, St), Pvars1, numberof(V, P1, St), C1),
        lowest_cost(Min, C1, N, Pvars1, Min2, Num, Num2, Vars, Vars2),
        N1 is N + 1,
        rest_new_costs(Old, New, Cr, Cr1, Ps, Ps1, Min2, Min1,
                                      N1, Num2, Num1, Vars2, Vars1).
%3
rest_new_costs([Pvars|Old], [Pvars|New], [C|Cr], [C|Cr1], P & Ps,
                        P & Ps1, Min, Min1, N, Num, Num1, Vars, Vars1) :- !,
        lowest_cost(Min, C, N, Pvars, Min2, Num, Num2, Vars, Vars2),
        N1 is N + 1,
        rest_new_costs(Old, New, Cr, Cr1, Ps, Ps1, Min2, Min1,
                                      N1, Num2, Num1, Vars2, Vars1).
%4
rest_new_costs([_|Old], [Pvars1|New], [_|Cr], [C1|Cr1], P & Ps,
                        P1 & Ps1, Min, Min1, N, Num, Num1, Vars, Vars1) :- !,
        cost(P, Pvars1, P1, C1),
        lowest_cost(Min, C1, N, Pvars1, Min2, Num, Num2, Vars, Vars2),
        N1 is N + 1,
        rest_new_costs(Old, New, Cr, Cr1, Ps, Ps1, Min2, Min1,
                                      N1, Num2, Num1, Vars2, Vars1).
%5
rest_new_costs(_, [Pvars1], _, [C1], setof(V, P, St),
                setof(V, P1, St), Min, Min1, N, Num, Num1, Vars, Vars1) :- !,
        cost(setof(V, P, St), Pvars1, setof(V, P1, St), C1),
        lowest_cost(Min, C1, N, Pvars1, Min1, Num, Num1, Vars, Vars1).
%6
rest_new_costs(_, [Pvars1], _, [C1],
                        numberof(V, P, St), numberof(V, P1, St),
                              Min, Min1, N, Num, Num1, Vars, Vars1) :- !,
        cost(numberof(V, P, St), Pvars1, numberof(V, P1, St), C1),
        lowest_cost(Min, C1, N, Pvars1, Min1, Num, Num1, Vars, Vars1).
%7
rest_new_costs([Pvars], [Pvars], [C], [C], P, P,
                              Min, Min1, N, Num, Num1, Vars, Vars1) :- !,
        lowest_cost(Min, C, N, Pvars, Min1, Num, Num1, Vars, Vars1).
%8
rest_new_costs(_, [Pvars1], _, [C1], P, P1,
                              Min, Min1, N, Num, Num1, Vars, Vars1) :- !,
        cost(P, Pvars1, P1, C1),
        lowest_cost(Min, C1, N, Pvars1, Min1, Num, Num1, Vars, Vars1).


%1     previously found generator
lowest_cost(Min, P_cost, _num, _vars, Min1,
                              gen:Num, gen:Num, Vars, Vars) :- !,
        ( Min = gen, !, Min1 = P_cost
        |
          P_cost = gen, !, Min1 = Min
        |
          P_cost =< Min, !, Min1 = P_cost
        |
          Min1 = Min ).

%1     determine new lowest cost, no generator so far
lowest_cost(Min, P_cost, P_num, P_vars, Min1, Num, Num1, Vars, Vars1) :- !,
```

```
        ( Min = gen, !, Num1 = gen:Num, Min1 = P_cost, Vars1 = Vars
        |
          P_cost = gen, !,       % a generator.
               Min1 = Min, Num1 = gen:P_num, Vars1 = P_vars
        |
          P_cost =< Min, !,               % pred has lower cost.
               Min1 = P_cost, Num1 = P_num, Vars1 = P_vars
        |
          Min1 = Min, Num1 = Num, Vars1 = Vars ).


        % instantiate variables of the lowest cost predicate in the
        % variable lists of other predicates.
 %1
instantiate(_, [], []) :- !.
 %2
instantiate(Pred_vars, [Vars|Lists], [Vars1|Lists1]) :-
        new_var_state(Pred_vars, Vars, Vars1), !,
        instantiate(Pred_vars, Lists, Lists1).


 %1     no more variables in least cost predicate.
new_var_state([], Vars, Vars) :- !.
 %2
new_var_state([free V|R], Vars, Vars1) :- !,
         ( contains_var(V, Vars), !, change_state(Vars, V, Vars2),
           new_var_state(R, Vars2, Vars1)
        |
           new_var_state(R, Vars, Vars1) ).
 %3
new_var_state([fixed _|R], Vars, Vars1) :- !, new_var_state(R, Vars, Vars1).
 %4
new_var_state([Fvars|Svars], Vars, Vars1) :- !,
        new_var_state(Fvars, Vars, Vars2), new_var_state(Svars, Vars2, Vars1).
 %5
new_var_state([Fvars|Svars], Vars, Vars1) :- !,
        new_var_state(Fvars, Vars, Vars2), new_var_state(Svars, Vars2, Vars1).


 %1
change_state([], _, []) :- !.
 %2     change a variable's state to fixed.
change_state([fixed V1|R], V, [fixed V1|R1]) :- !,
         ( V == V1, !, R1 = R  |  change_state(R, V, R1) ).
 %3
change_state([free V1|R], V, [New|R1]) :- !,
         ( V == V1, !, New = fixed V, R1 = R
        |
          New = free V1, change_state(R, V, R1) ).
 %4
change_state([Fvars|Vars], V, [Fvars1|Vars1]) :-
        change_state(Fvars, V, Fvars1), change_state(Vars, V, Vars1).


 %1     see if a predicates variables have changed states.
state_changed(Vars, Vars) :- !, fail.
 %2
state_changed(_, _).

 %1
infinite(Cost, Inf_flag) :- infinity(Cost), !, Inf_flag = yes.
 %2
infinite(_, _).


        % cost_rest(+Cost_of_lowest_sofar, -Final_lowest_cost,
        %                +Current_pred_num, +Pred_num_of_lowest_sofar,
```

```
        %                  -Final_lowest_pred_num, +Rest_preds, -New_Preds,
        %                  +Costs_of_preds, -Costs_of_preds1, +Pred_vars,
        %                          +Vars_of_lowest_sofar, -Vars_of_final_lowest).
 %1
cost_rest(C, C1, N, Num, Num1, Pred & Query, Pred1 & Query1, Costs,
                [C1|Costs1], [Pred_varsn|Vars], Pred_vars, Pred_vars1) :- !,
        cost(Pred, Pred_varsn, Pred1, C1),
        lowest_cost(C, C1, N, Pred_varsn, C2, Num, Num2,
                                                    Pred_vars, Pred_vars2),
        N1 is N + 1,
        cost_rest(C2, C1, N1, Num2, Num1, Query, Query1,
                        Costs, Costs1, Vars, Pred_vars2, Pred_vars1).
 %2
cost_rest(C, C1, N, Num, Num1, Pred, Pred1, Costs, [C1|Costs],
                                 [Pred_varsn], Pred_vars, Pred_vars1) :-
        cost(Pred, Pred_varsn, Pred1, C1),
        lowest_cost(C, C1, N, Pred_varsn, C1, Num, Num1,
                                                    Pred_vars, Pred_vars1).

        % cost(+Pred, +Vars, -Pred1, -Cost).
 %1
cost(setof(V, Preds, L), [[_]|Vars], setof(V, Preds1, L), Cost) :- !,
         ( partition(Vars, Preds, _inf, Cost1, Preds1), !,
           instantiates_none(V, Vars, Cost1, Cost)
        |
           plan_sub(Preds, Vars, Preds1, Inf, Cost1),
           ( var(Inf), !, Cost2 = Cost1 | infinity(Cost2)),
           instantiates_none(V, Vars, Cost2, Cost)
         ).
 %2
cost(numberof(V, Preds, N), [[_]|Vars], numberof(V, Preds1, N), Cost) :- !,
         ( partition(Vars, Preds, _inf, Cost, Preds1), !
        |
           plan_sub(Preds, Vars, Preds1, Inf, Cost1),
           ( var(Inf), !, Cost = Cost1 | infinity(Cost)) ).
 %3
cost(\+ { Pred & Preds }, [Vars|Vars_rest], \+ { Query1 }, Cost) :- !,
         ( partition([Vars|Vars_rest], Pred & Preds, _, Cost1, Query1), !
        |
           cost(Pred, Vars, Pred1, Cost1),
           cost_rest(Cost1, Min, 2, 1, Num, Preds, New_preds,
                        [Cost1], Costs, Vars_rest, Vars, Pred_vars1),
           plan_rest(Pred1 & New_preds, Pred & Preds, [Vars|Vars_rest],
                             Num, Costs, Pred_vars1, _, Min, Query1) ),
         ( all_fixed([Vars|Vars_rest]), !, Cost = 1 | infinity(Cost) ).
 %4
cost(\+ { Pred }, Vars, \+ { Pred }, Cost) :- !,
         ( all_fixed(Vars), !, Cost = 1 | infinity(Cost) ).
 %5
cost(Pred, Vars, Pred, Cost) :-
        functor(Pred, P, Arity),
        db_solutions(P/Arity, Vars, Cost).      % get cost from meta-data.

 %1
instantiates_none(Setvars, Vars, Cost, Cost1) :-
        delete_vars(Setvars, Vars, Vars1),
        ( all_fixed(Vars1), !, Cost1 = 0 ; Cost1=Cost ).

 %1
delete_vars(Var, Vars, Vars1) :-
        var(Var), !, delete_var_list(Var, Vars, Vars1).
 %2
delete_vars(Var-Rest, Vars, Vars1) :-
        delete_var_list(Var, Vars, Vars2), delete_vars(Rest, Vars2, Vars1).

 %1
delete_var_list(_Var, [], []) :- !.
```

```
 %2
delete_var_list(Var, [First|Rest], [First1|Rest1]) :-
        delete_var(Var, First, First1), delete_var_list(Var, Rest, Rest1).


 %1
delete_var(_, [], []) :- !.
 %2
delete_var(Var, [free Var1|Rest], Vars1) :-
         ( Var == Var1, !, Vars1 = Rest
         |
           Vars1 = [free Var1|Rest1], delete_var(Var, Rest, Rest1) ).
 %3
delete_var(Var, [fixed Var1|Rest], Vars1) :-
         ( Var == Var1, !, Vars1 = Rest
         |
           Vars1 = [free Var1|Rest1], delete_var(Var, Rest, Rest1) ).
 %4
delete_var(Var, [First|Rest], [F1|Rest1]) :-
        delete_var(Var, First, F1),
        delete_var(Var, Rest, Rest1).


        %get_lowest(+Num, +Costs, -Costs1, +Vars, -Vars_old,
        %                           +Pred_vars, -Vars_new, +Preds_new,
        %                                         +Preds_old, -Pred_low, -Preds_rest).
        % return the lowest costing predicate, its variable list and cost,
        % and instantiate its variables in other predicates.
 %1
get_lowest(1, [_|Cs], Cs, [Vars|Vs], Vs, Inst_vars, Vs_new,
                \+(Preds) & _, _ & Old_rest, \+(Preds1), Old_rest) :- !,
        instantiate(Inst_vars, Vs, Vs_new),
        plan_sub(Preds, Vars, Preds1, _, _).
 %2
get_lowest(1, [_|Cs], Cs, [_|Vs], Vs, Inst_vars, Vs_new,
                                 Pred & _, _ & Old_rest, Pred, Old_rest) :- !,
        instantiate(Inst_vars, Vs, Vs_new).
 %3
get_lowest(2, [Fc,_|Cs], [Fc|Cs], [Fv, Vars|Vs], Inst_vars, Vars_new,
                 [Fv|Vs], _ & \+(Preds) & _, Old_p & _ & Old_rest,
                                             \+(Preds1), Old_p & Old_rest) :- !,
        instantiate(Inst_vars, [Fv|Vs], Vars_new),
        plan_sub(Preds, Vars, Preds1, _, _).
 %4
get_lowest(2, [Fc,_|Cs], [Fc|Cs], [Fv,_|Vs], [Fv|Vs], Inst_vars,
                 Vars_new, _ & Pred1 & _, Old_p & _ & Old_rest,
                                                Pred1, Old_p & Old_rest) :- !,
        instantiate(Inst_vars, [Fv|Vs], Vars_new).
 %5
get_lowest(2, [Fc,_|Cs], [Fc|Cs], [Fv,Vars|Vs], [Fv|Vs], Inst_vars,
                  Vars_new,_ & \+(Preds), Old_p & _, \+(Preds1), Old_p) :- !,
        instantiate(Inst_vars, [Fv|Vs], Vars_new),
        plan_sub(Preds, Vars, Preds1, _, _).
 %6
get_lowest(2, [Fc,_|Cs], [Fc|Cs], [Fv,_|Vs], [Fv|Vs], Inst_vars,
                         Vars_new,_ & Pred1, Old_p & _, Pred1, Old_p) :- !,
        instantiate(Inst_vars, [Fv|Vs], Vars_new).
 %7
get_lowest(N, [Fc|Cs], [Fc|Cs1], [Fv|Vs], [Fv|Vs1],
                 Inst_vars, [V_new|Vs_new], _ & Preds,
                             Old_p & Old_rest, Pred_lo, Old_p & Preds1) :- !,
        instantiate(Inst_vars, [Fv], [V_new]),
        N1 is N - 1,
        get_lowest(N1, Cs, Cs1, Vs, Vs1, Inst_vars, Vs_new,
                                     Preds, Old_rest, Pred_lo, Preds1).

 %1
```

```
all_fixed([]) :- !.
 %2
all_fixed([free _|_]) :- !, fail.
 %3
all_fixed([fixed _|Rest]) :- !, all_fixed(Rest).
 %4
all_fixed([List|Rest]) :- all_fixed(List), all_fixed(Rest).

        % produce a list of lists where each list contains the variables
 %1       of a predicate pre-fixed with its state.
var_list(P & P1, [Vars|Vars1]) :- !, vlist(P, Vars), var_list(P1, Vars1).
 %2
var_list(P, [Vars]) :- vlist(P, Vars).

 %1
vlist(aggreg(A, B, V, Var1), [free A, free B, free V, free Var1]).
 %2
vlist(setof(_, Preds, Set), [[free Set]|List]) :- !, var_list(Preds, List).
 %3
vlist(numberof(_, Preds, N), List) :- !,
        var_list(Preds, List1),
        ( var(N), !, List = [[free N]|List1] | List = [[fixed N]|List1] ).
 %4
vlist(\+ { Preds }, List1) :- !, var_list(Preds, List1).
 %5
vlist(Pred & Preds, [List1|List2]) :- !,
        vlist(Pred, List1), var_list(Preds, List2).
 %6
vlist(Pred, List) :- Pred =.. [_|Args], vlist_states(Args, List).


        % build a sublist with each predicate variable prefixed by
 %1       its current state.
vlist_states([], []) :- !.
 %2
vlist_states([Var|Rest_vars], [Var_state|Rest]) :-
         ( var(Var), !, Var_state = free Var | Var_state = fixed Var ),
        vlist_states(Rest_vars, Rest).

 %1
find_var(Num, Num1, Var, [fixed Var1|Rest]) :-
         ( Var == Var1, !, Num1 = Num
        |
           N1 is Num + 1, find_var(N1, Num1, Var, Rest) ).
 %2
find_var(Num, Num1, Var, [free Var1|Rest]) :-
         ( Var == Var1, !, Num1 = Num
        |
           N1 is Num + 1, find_var(N1, Num1, Var, Rest) ).

        % partition the remaining predicates in to independent portions,
 %1       fail if this cannot be done.
partition(Vars, Preds, Inf, Min, Preds1)  :-
        dependency(1, Vars, Lists1),
         ( one_list(Lists1), !, fail
        |
           divide_preds(Lists1, Vars, Var_lists, Preds, Pred_lists),
           plan_all(Pred_lists, Var_lists, Inf, Min, Preds1) ).

 %1
one_list([_]) :- !.

 %1
produce_lists(A & B, Vars, [Args|Rest]) :-
        all_vars_in(A, Vars, Args), produce_lists(B, Vars, Rest).
 %2
produce_lists(A, Vars, [Args]) :- all_vars_in(A, Vars, Args).
```

```
 %1
dependency(N, Lists, Lists1) :-
        dep_lists(N, Lists, Lists, Lists2), combine(Lists2, Lists1).

 %1
dep_lists(_, _, [], []) :- !.
 %2
dep_lists(N, Lists, [List|R], [List1|R1]) :-
        dependency_list(List, N, 1, Lists, List1),
        N1 is N + 1,
        dep_lists(N1, Lists, R, R1).


        % dependency_list(+List_of_vars, +List_number_in_lists,
        %                         +Number_of_next_list_in_lists,
        %                                 +Remaining_lists, -Dependency_list).
 %1     list must be dependent on itself, put it's own number in.
dependency_list(_, N, _, [], [N]) :- !.
 %2     list is the next from remainder of other lists.
dependency_list(List, N, N, [_|R], R1) :- !,
        N1 is N + 1, dependency_list(List, N, N1, R, R1).
 %3     look for common variables with another list.
dependency_list(List, N, N1, [F|R], List1) :-
        N2 is N1 + 1,
         ( common_var(List, F), !, List1 = [N1|R1],
           dependency_list(List, N, N2, R, R1)
        |
           dependency_list(List, N, N2, R, List1) ).


common_var([], _) :- !, fail.

common_var([[[free F]|Vlist]|R], List) :- !,
         ( free_var_member(F, List), !
        |
           ( common_var(Vlist, List), ! | common_var(R, List) ) ).

common_var([[[fixed _]|Vlist]|R], List) :- !,
        ( common_var(Vlist, List), ! | common_var(R, List) ).

common_var([fixed _|R], List) :- !, common_var(R, List).

common_var([free F|R], List) :- !,
        ( free_var_member(F, List), ! | common_var(R, List) ).


free_var_member(_, []) :- !, fail.

free_var_member(V, [fixed _|Rest]) :- !, free_var_member(V, Rest).

free_var_member(V, [free V1|Rest]) :-
        ( V == V1, ! | free_var_member(V, Rest) ).

free_var_member(V, [Vars|Rest]) :- !,
        ( free_var_member(V, Vars), ! | free_var_member(V, Rest) ).

free_var_member(V, [Vars|Rest]) :- !,
        ( free_var_member(V, Vars), ! | free_var_member(V, Rest) ).


combine([], []) :- !.

combine([First|Rest], [First1|R1]) :-
        collect(First, First, Rest, Rest1, First1), combine(Rest1, R1).
```

```
collect([], _, Lists, Lists, []) :- !.

collect([Num|Rest], List, Lists, Lists1, [Num|New]) :-
        lists_with(Num, List, Lists, Lists2, Nums),
        fappend(Nums, List, New_list),
        fappend(Nums, Rest, New_rest),
        collect(New_rest, New_list, Lists2, Lists1, New).


lists_with(_, _, [], [], []) :- !.

lists_with(N, List, [F|Rest], Lists2, New_nums) :-
         ( fmember(N, F), !,
           find_new(F, List, New),
           fappend(New, List, List1),
           lists_with(N, List1, Rest, Lists2, Rnew),
           fappend(New, Rnew, New_nums)
         |
           Lists2 = [F|Lists3], lists_with(N, List, Rest, Lists3, New_nums) ).


find_new([], _, []) :- !.

find_new([F|R], List, New) :-
         ( fmember(F, List), !, find_new(R, List, New)
         |
           New = [F|R1], find_new(R, [F|List], R1) ).


        % divide_preds(+Num_lists, +Vars, -Vars_Lists, +Preds, -Pred_lists).
divide_preds([], _, [], _, []) :- !.

divide_preds([Nums|Rest], Vars, [Var_list|Rest2], Preds, [Pred_list|Rest1]) :-
        get(Nums, Preds, Vars, Var_list, Pred_list),
        divide_preds(Rest, Vars, Rest2, Preds, Rest1).


get([N], Preds, Vars, [Var], Pred) :- !,
        nth_pred(N, Preds, Pred),
        nth_var(N, Vars, Var).

get([N|R], Preds, Vars, [V|Rest], Pred & Rest1) :-
        nth_pred(N, Preds, Pred),
        nth_var(N, Vars, V),
        get(R, Preds, Vars, Rest, Rest1).


nth_pred(1, P & _, P) :- !.

nth_pred(1, P, P) :- !.

nth_pred(N, _ & Preds, Pred) :- N1 is N - 1, nth_pred(N1, Preds, Pred).


nth_var(1, [V|_], V) :- !.

nth_var(N, [_|R], V) :- N1 is N - 1, nth_var(N1, R, V).


 %1      plan_all(+Pred_lists, +Var_lists, -Inf, -Min, -Preds1)
plan_all(Pred_lists, Vars, Inf, Min1, Preds1) :-
        plan_cut(Pred_lists, Vars, Inf, [[Min,Preds]|R]),
        order_subqueries(R, Inf, Min, Min1, [Min], [Preds], Preds1).

 %1
plan_cut([], _vars, _, []) :- !.
 %2
```

```
plan_cut([First|Rest], [Vars|Rest1], Inf, [[Cost,Preds]|Rest2]) :-
        plan_sub(First, Vars, Preds, Inf, Cost),
        plan_cut(Rest, Rest1, Inf, Rest2).

 %1
order_subqueries([], _, Min, Min, _, Preds, Preds) :- !.
 %2
order_subqueries([[Minval,Pred]|Rest], Inf, Min, Min1,
                                                  Costs, Preds, Preds1) :-
        infinite(Minval, Inf),
         ( Minval =< Min, var(Inf1), !, Min2 = Minval
        |
           Min2 = Min, Inf = Inf1 ),
        insert_pred(Pred, Minval, Costs, Costs1, Preds, Preds2),
        order_subqueries(Rest, Inf, Min2, Min1, Costs1, Preds2, Preds1).

 %1
insert_pred(P, Cost, [Cost1|Rest], Costs1, P1 & P2, P3) :- !,
         ( Cost < Cost1, !, Costs1 = [Cost,Cost1|Rest], P3 = ([P] & P1 & P2)
        |
           Costs1 = [Cost1|Rest1], P3 = (P1 & Pr),
           insert_pred(P, Cost, Rest, Rest1, P2, Pr) ).
 %2
insert_pred(P, Cost, [Cost1], Costs1, P1, P2) :-
         ( Cost < Cost1, !, Costs1 = [Cost,Cost1], P2 = ([P] & P1)
        |
           Costs1 = [Cost1,Cost], P2 = (P1 & [P]) ).

 %1
infinity(100000).                                 % well close !.

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : pretty.pl                                     */
        /*      PURPOSE : F-structure & semantic representation         */
        /*                pretty printer.                               */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(pretty, [
                pretty_info/1,
                pretty_sem/1
                ]).

        /* I M P O R T S */

        % <none>.

        /* P R E D I C A T E S */
 %1     pretty-print an information structure.
pretty_info(Info) :-
        copy_term(Info, Info1),         % make sure no changes made to info.
        numbervars(Info1, 1, _),
        ppi(Info1, 0).

 %1
ppi(I ind _ stype Slots^Quantifier^Semantic^_^Fs, N) :-
        format('~*|~w~n', [N, '**********']),
        format('~*|~w~w~n', [N, '| I : ', I]),
        format('~*|~w~w~n', [N, '| S : ', Slots]),
        format('~*|~w~w~n', [N, '| Q : ', Quantifier]),
        format('~*|~w~w~n', [N, '| P : ', Semantic]),
        format('~*|~w~n', [N, '| F : ---------']),
```

```
        N1 is N + 3,
        pp_fs(Fs, N1),
        format('~*|~w~n', [N, '**********']).

%1
pp_fs([], N) :- format('~*|~w~n', [N, '----------']).
%2
pp_fs([F = set S|R], N) :-
        N1 is N + 3, format('~*|~w~w~n', [N, F, '= ']),
        pp_set(S, N1), pp_fs(R, N).
%3
pp_fs([F = fs Val|R], N) :-
        N1 is N + 3, format('~*|~w~w~n', [N, F, '= ']),
        ppi(Val, N1), pp_fs(R, N).
%4
pp_fs([Ft = atom Val|R], N) :-
        format('~*|~w~w~w~n', [N, Ft, '= ', Val]), pp_fs(R, N).

%1
pp_set([], N) :- format('~*|~w~n', [N, '______________']).
%2
pp_set([Val|R], N) :- ppi(Val, N), pp_set(R, N).

%1      pretty-print a semantic translation.
pretty_sem(Sem) :-
        copy_term(Sem, Sem1),           % make sure no changes made to Sem.
        numbervars(Sem1, 0, _),
        pps(Sem1, 0).

%1
pps( wh(V, Ps), I) :- !,
        format('~N~*|~w~w~w~n', [I, 'wh(', V, ',']),
        I1 is I + 5, pps(Ps, I1), I2 is I + 2,
        format('~N~*|~w~3n', [I2, ').']).
%2
pps( y_n(Ps), I) :- !,
        format('~N~*|~w~n', [I, 'yn(']),
        I1 is I + 3, pps(Ps, I1), I2 is I + 2,
        format('~N~*|~w~3n', [I2, ').']).
%3
pps( numberof(V, Ps), I) :- !,
        format('~N~*|~w~w~w~n', [I, 'numberof(', V, ',']),
        I1 is I + 6, pps(Ps, I1), I2 is I + 3,
        format('~N~*|~w~3n', [I2, ')']).
%4
pps( setof(S, Ps, Set), I) :- !,
        format('~*|~w~w~w~n', [I, 'setof(', S, ',']),
        I1 is I + 8, pps(Ps, I1), I2 is I + 6,
        format('~N~*|~w~w~w', [I2, ',', Set, ')']).
%5
pps( the_sg(V, Ps), I) :- !,
        format('~*|~w~w~w~n', [I, 'the_sg(', V, ',']),
        I1 is I + 6, pps(Ps, I1), I2 is I + 6, nl,
        format('~*|~w', [I2, ')']).
%6
pps( numberof(V, Ps, N), I) :- !,
        format('~*|~w~w~w~n', [I, 'numberof(', V, ',']),
        I1 is I + 10, pps(Ps, I1), I2 is I + 7,
        format('~N~*|~w~w~w', [I2, ',', N, ')']).
%7
pps( the_pl(V, Ps), I) :- !,
        format('~*|~w~w~w~n', [I, 'the_pl(', V, ',']),
        I1 is I + 6, pps(Ps, I1), I2 is I + 3,
        format('~N~*|~w', [I2, ')']).
%8
pps( each(V, Ps), I) :- !,
        format('~*|~w~w~w~n', [I, 'each(', V, ',']),
```

```
        I1 is I + 6, pps(Ps, I1), I2 is I + 3,
        format('~N~*|~w', [I2, ')']).
%9
pps( some(V, Ps), I) :- !,
        format('~*|~w~w~w~n', [I, 'some(', V, ',']),
        I1 is I + 6, pps(Ps, I1), I2 is I + 3,
        format('~N~*|~w', [I2, ')']).
%10
pps( any(V, Ps), I) :- !,
        format('~*|~w~w~w~n', [I, 'any(', V, ',']),
        I1 is I + 6, pps(Ps, I1), I2 is I + 3,
        format('~N~*|~w', [I2, ')']).
%11
pps( of_the(V, Ps), I) :- !,
        format('~*|~w~w~w~n', [I, 'of_the(', V, ',']),
        I1 is I + 6, pps(Ps, I1), I2 is I + 3,
        format('~N~*|~w', [I2, ')']).
%12
pps( a(V, Ps), I) :- !,
        format('~*|~w~w~w~n', [I, 'a(', V, ',']),
        I1 is I + 6, pps(Ps, I1), I2 is I + 3,
        format('~N~*|~w', [I2, ')']).
%13
pps( no(V, Ps), I) :- !,
        format('~*|~w~w~w~n', [I, 'no(', V, ',']),
        I1 is I + 6, pps(Ps, I1), I2 is I + 3,
        format('~N~*|~w', [I2, ')']).
%14
pps( aggreg(Ag, Ps, Var), I) :- !,
        format('~*|~w~w~w~n', [I, 'aggreg( ', Ag, ',']),
        I1 is I + 10, pps(Ps, I1), I2 is I + 7,
        format('~N~*|~w~w', [I2, Var, ')']).
%15
pps( \+ { Ps }, I) :- !,
        format('~*|~w', [I, 'not (']),
        I1 is I + 6, pps(Ps, I1), I2 is I + 4,
        format('~N~*|~w', [I2, ')']).
%16
pps( [ Ps ], I) :- !,
        format('~*|~w', [I, '( ']), I1 is I + 2, pps(Ps, I1),
        format('~w', [' }']).
%17
pps(P & Ps, I) :- !,
        pps(P, I), format('~w~n', [' &']), pps(Ps, I).
%18
pps(A < B, I) :- !, format('~*|~w~w~w', [I, A, ' < ', B]).
%19
pps(A = B, I) :- !, format('~*|~w~w~w', [I, A, ' = ', B]).
%20
pps(A > B, I) :- !, format('~*|~w~w~w', [I, A, ' > ', B]).
%21
pps(P, I) :- format('~*|~w', [I, P]).

        /* ************************************************************ */



        /* ************************************************************ */
        /*      FILE    : pre_execute.pl                                */
        /*      PURPOSE : simplify and plan queries.                    */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(pre_execute, [
                pre_execute/2
                ]).
```

```
        /* I M P O R T S */

:- use_module(plan_query, [
                plan_query/2
                ]).

        /* P R E D I C A T E S */
 %1
pre_execute(Query, Query1) :-
        simplify(Query, Query2),
        plan_query(Query2, Query1).

 %1
simplify(wh(Var,each(Var1, Preds)),
                        wh(Set,setof(Var-Var1, Preds1, Set))) :- !,
        simplify(Preds, Preds1).
 %2
simplify(wh(Var, Preds),wh(Var, Preds1) ) :- !, simplify(Preds, Preds1).
 %3
simplify(numberof(Var, Preds), numberof(Var, Preds1) ) :- !,
         ( var(Preds), !, Preds1 = Preds | simplify(Preds, Preds1) ).
 %4
simplify(y_n(Condition), y_n(Condition1)) :- simplify(Condition, Condition1).
 %5
simplify(of_the(Var, Preds), setof(Var-Var2, Preds2, Var1)) :-
        pl_preds(Preds, Var1, Preds1),
        copy_term((Preds1,Var,Var1), (Preds1,Var,Var2)),
        simplify(Preds1, Preds2).
 %6
simplify(of_the(Var, P & setof(Var1, Preds, Set) & Preds1),
                        setof(Var-Var2, P & Preds2, Set) & Preds1) :- !,
        copy_term((Preds,Var,Var1,Preds1), (Preds,Var,Var2,Preds1)),
        simplify(Preds, Preds2).
 %9
simplify(the_sg(_var, Preds), Preds1) :- !, simplify(Preds, Preds1).
 %10
simplify(a(_, Preds), Preds1) :- !, simplify(Preds, Preds1).
 %11
simplify(some(_, Preds), Preds1) :- !, simplify(Preds, Preds1).
 %12
simplify(the_pl(Var, Preds), setof(Var1, Preds2, Var)) :- !,
        simplify(Preds, Preds1),
        copy_term((Var,Preds1), (Var1,Preds2)).
 %13
simplify(setof(Var, Preds, List), setof(Var, Preds1, List)) :- !,
        simplify(Preds, Preds1).
 %14
simplify(aggreg(Pred, Preds, Ans), aggreg(Pred, Preds1, Ans)) :- !,
        simplify(Preds, Preds1).
 %15
simplify(no(_var, Preds), \+ { Preds1 }) :- !, simplify(Preds, Preds1).
 %16
simplify(each(_var, Pred & Preds), \+ { Pred1 & \+ { Preds1 } }) :- !,
        simplify(Pred, Pred1), simplify(Preds, Preds1).
 %17
simplify(any(_var, Preds), Preds1) :- !, simplify(Preds, Preds1).
 %15
simplify(\+ { Preds }, \+ { Preds1 }) :- !, simplify(Preds, Preds1).
 %16
simplify(Pred & Preds, Pred1 & Preds1) :- !,
        simplify(Pred, Pred1), simplify(Preds, Preds1).
 %17
simplify(\+ Pred, \+ Pred1) :- simplify(Pred, Pred1).
 %18
simplify(numberof(Var, Preds, Num), numberof(Var, Preds1, Num)) :-
        simplify(Preds, Preds1).
```

```
 %19
simplify(Pred, Pred) :- !.

 %1
pl_preds(the_pl(Var1, Preds), Var1, Preds) :- !.
 %2
pl_preds(Pred & Preds, Var, Pred & Preds1) :- pl_preds(Preds, Var, Preds1).

        /* ************************************************************* */


        /* ************************************************************* */
        /*      FILE    : pre_sem.pl                                     */
        /*      PURPOSE : quantification and attachment.                 */
        /* ************************************************************* */

        /* E X P O R T S */

:- module(pre_sem, [
                quant_attach/3
                ]).

        /* I M P O R T S */

:- use_module(fs_basics, [
                member_fs/2,
                delete_fs/3
                ]).

:- use_module(fast_basics, [
                fmember/2
                ]).

:- use_module(quants, [
                quant_order/9
                ]).

:- use_module(type_system, [
                compat/3
                ]).

        /* P R E D I C A T E S */

        /* extract all semantic useful info. from the F-structure
           and attach free adjuncts to some selected function. */
 %1
quant_attach(Slots_up, I ind _ stype Slots^Quant^Sem^Var^Fs,
                       I ind _ stype Slots1^Quant^Sem1^Var^Fs2^
                                    [Pro1, Num, Number, Adj, Aggs]) :-
        next_slots(Slots_up, Slots, Next),
        features(Next, Fs, Aggs, Adj, Adjuncts, Conjs, Number,
                           Neg, Other_slots, Others, Pro, Num, Fs1),
        quant_order(Fs1, Slots, Neg, Adjuncts, Conjs,
                    Other_slots, Others, Slots1, Fs2),
        sem_or_pro(Pro, Sem, Sem1, Pro1).

 %1
sem_or_pro([], Sem, Sem, []) :- !.
 %2
sem_or_pro(+, Pro, [], Pro).


        % features(+Slots, +Fs, -Aggs, -Adj, -Adjuncts, -Num, -Conjs,
        %                            -Neg, -Other_slots, -Others, -Fs1).
 %1
features(_, [/*fs*/], [/*aggs*/], [/*adj*/],
                [/*adjuncts*/], [/*conjs*/], [/*number*/],
```

```
                    [/*neg*/], [/*other_slots*/], [/*others*/],
                                [/*pro*/], [/*num*/], [/*fs1*/]).
 %2
features(Slots, [neg = atom +|R_fs], Aggs, Adj, Adjuncts, Conjs,
                        Number, neg, Other_slots, Others, Pro, Num, Fs1) :- !,
        features(Slots, R_fs, Aggs, Adj, Adjuncts, Conjs, Number,
                                _, Other_slots, Others, Pro, Num, Fs1).
 %3
features(Slots, [card = atom Number|R_fs], Aggs, Adj, Adjuncts,
                Conjs, Int1, Neg, Other_slots, Others, Pro, Num, Fs1) :- !,
        number(Number, Int),
         ( fmember(q = atom Q, R_fs), !, new_number(Q, Int, Int1)
        |
           ( fmember(meas = atom M, R_fs), !, new_number(M, Int, Int1)
          |
             Int1 = Int
           )
         ),
        features(Slots, R_fs, Aggs, Adj, Adjuncts, Conjs, _,
                                Neg, Other_slots, Others, Pro, Num, Fs1).
 %4
features(Slots, [aggregates = set Vals|R_fs], Vals, Adj, Adjuncts,
                Conjs, Number, Neg, Other_slots, Others, Pro, Num, Fs1) :- !,
        features(Slots, R_fs, _, Adj, Adjuncts, Conjs, Number,
                                Neg, Other_slots, Others, Pro, Num, Fs1).
 %5
features(Slots, [adjs = set Vals|R_fs], Aggs, Vals, Adjuncts,
                Conjs, Number, Neg, Other_slots, Others, Pro, Num, Fs1) :- !,
        features(Slots, R_fs, Aggs, _, Adjuncts, Conjs, Number,
                                Neg, Other_slots, Others, Pro, Num, Fs1).
 %6
features(Slots, [adjuncts = set Vals|R_fs], Aggs, Adj,
                Adj_vals, Conjs, Number, Neg, Other_slots, Others,
                                Pro, Num, [adjuncts = set Adj_vals|Fs1]) :- !,
        adjuncts(Slots, Vals, Adj_vals),
        features(Slots, R_fs, Aggs, Adj, _, Conjs, Number,
                                Neg, Other_slots, Others, Pro, Num, Fs1).
 %7
features(Slots, [head = fs Hindex ind _ stype
                  Hslots^Hquant^Hsem^(Hvar:Htype)^Hfs|R_fs],
                Aggs, Adj, Adjuncts, Conjs, Number, Neg,
                        [head = Hvar:Htype, mod = Mvar:Mtype],
                [head = fs New_head, mod = fs New_mod], Pro, Num, Fs1) :- !,
        member_fs(mod = fs Mindex ind _ stype
                        Mslots^Mquant^Msem^(Mvar:Mtype)^Mfs, R_fs),
        quant_attach(Slots, Hindex ind _ stype
                        Hslots^Hquant^Hsem^(Hvar:Htype)^Hfs, New_head),
        quant_attach(Slots, Mindex ind _ stype
                        Mslots^Mquant^Msem^(Mvar:Mtype)^Mfs, New_mod),
        features(Slots, R_fs, Aggs, Adj, Adjuncts, Conjs,
                                        Number, Neg, _, _, Pro, Num, Fs1).
 %8
features(Slots, [pied = fs Pindex ind _ stype
                Pslots^Pquant^Psem^(Pvar:Ptype)^Pfs|R_fs], Aggs,
                Adj, Adjuncts, Conjs, Number, Neg, [pied = Pvar:Ptype],
                                _, Pro, Num, [pied = fs New_pied|Fs1]) :- !,
        quant_attach(Slots, Pindex ind _ stype
                        Pslots^Pquant^Psem^(Pvar:Ptype)^Pfs, New_pied),
        features(Slots, R_fs, Aggs, Adj, Adjuncts, Conjs,
                                        Number, Neg, _, _, Pro, Num, Fs1).
 %9
features(Slots, [rel_adj = fs Rindex ind _ stype
                Rslots^Rquant^Rsem^(Rvar:Rtype)^Rfs|R_fs], Aggs,
                Adj, Adjuncts, Conjs, Number, Neg, [rel_adj],
                        _, Pro, Num, [rel_adj = fs New_rel_adj|Fs1]) :- !,
        quant_attach(Slots, Rindex ind _ stype
                        Rslots^Rquant^Rsem^(Rvar:Rtype)^Rfs, New_rel_adj),
```

```
        features(Slots, R_fs, Aggs, Adj, Adjuncts, Conjs,
                                      Number, Neg, _, _, Pro, Num, Fs1).
 %9
 features(Slots, [conjs = set Vals|R_fs], Aggs, Adj,
                Adjuncts, Vals1, Number, Neg, Other_slots, Others,
                                Pro, Num, [conjs = set Vals1|Fs1]) :- !,
        conjs(Slots, Vals, Vals1),
        features(Slots, R_fs, Aggs, Adj, Adjuncts, _, Number,
                         Neg, Other_slots, Others, Pro, Num, Fs1).
 %10
 features(Slots_up, [F = fs I ind _ stype
                Slots^Quant^Sem^Var^Fs|R_fs],
                Aggs, Adj, Adjuncts, Conjs, Number, Neg,
                Other_slots, Others, Pro, Num, [F = fs Val1|Fs1]) :- !,
        next_slots(Slots_up, Slots, Next),
        quant_attach(Next, I ind _ stype
                                Slots^Quant^Sem^Var^Fs, Val1),
        features(Slots_up, R_fs, Aggs, Adj, Adjuncts, Conjs, Number,
                                Neg, Other_slots, Others, Pro, Num, Fs1).
 %11
 features(Slots, [num = atom Num|R_fs], Aggs, Adj, Adjuncts, Conjs,
                         Number, Neg, Other_slots, Others, Pro, Num, Fs1) :- !,
        features(Slots, R_fs, Aggs, Adj, Adjuncts, Conjs, Number,
                                Neg, Other_slots, Others, Pro, _, Fs1).
 %12
 features(Slots, [proportional = atom +|R_fs], Aggs, Adj, Adjuncts,
                Conjs, Number, Neg, Other_slots, Others, +, Num, Fs1) :- !,
        features(Slots, R_fs, Aggs, Adj, Adjuncts, Conjs, Number,
                                Neg, Other_slots, Others, _, Num, Fs1).
 %13
 features(Slots, [_ = atom _|R_fs], Aggs, Adj, Adjuncts, Conjs,
                         Number, Neg, Other_slots, Others, Pro, Num, Fs1) :- !,
        features(Slots, R_fs, Aggs, Adj, Adjuncts, Conjs, Number,
                                Neg, Other_slots, Others, Pro, Num, Fs1).

 %1
 next_slots(Slots_up, [], Slots_up).
 %2
 next_slots(_, Slots, Slots).


        /* adjuncts are first attached to a function in the enclosing
           f-structure and then translated as functions in f-structures. */
 %1
 adjuncts(_, [], []) :- !.
 %2
 adjuncts(Functions, [I ind _ stype
                Slots^Quant^Sem^Var^Fs|Adjuncts], [Adjunct1|New]) :-
        attach(Functions, Slots, Slots1, Fs),
        quant_attach(Slots, I ind _ stype Slots1^Quant^Sem^Var^Fs, Adjunct1),
        adjuncts(Functions, Adjuncts, New).

 %1
 conjs(_, [], []) :- !.
 %2
 conjs(Slots, [Conj|Rest_conjs], [New_conj|Rest_new]) :-
        quant_attach(Slots, Conj, New_conj),
        conjs(Slots, Rest_conjs, Rest_new).

 %1
 attach(Functions, [F - F1 = V:T|R], [F - F1 = V:T|Slots1], Fs) :-
         ( member_fs(F = fs _, Fs), !, attach(Functions, R, Slots1, Fs)
        |
           select_attach(Functions, F = V:_, fail), Slots1 = R
        ).
 %2
 attach(Functions, [F = V:T|R], [F = V:T|Slots1], Fs) :-
```

```
          ( member_fs(F = fs _val, Fs), !, attach(Functions, R, Slots1, Fs)
          |
            Slots1 = R, select_attach(Functions, F = V:_t, fail)
          ).

 %1
select_attach([], _, fail) :- !,
        write('attachment failure can not find a function to attach to'),
        nl, fail.
 %2
select_attach([], _ = V:T, _ = V:T1) :- compat(T, T1, _).
 %3
select_attach([F = V:T|R], Funct, Funct1) :-
          ( prefer(F = V:T, Funct1), !, select_attach(R, Funct, F = V:T)
          |
            select_attach(R, Funct, Funct1)
          ).


        /* pick the function to attach a free adjunct to from two alternative
           functions (the initial function assigned is the symbol 'fail' .*/

prefer(_, fail) :- !.                       % any function to none at all !.

prefer(obj = _:_, subj = _:_) :- !.         % attach to obj rather than subj.

prefer(vcomp = _:_, obj = _:_) :- !.

prefer(vcomp = _:_, subj = _:_) :- !.


new_number(more, Int, gt Int).

new_number(less, Int, lt Int).

new_number(Unit, Int, Int -- Unit) :- unit(Unit).


unit(million).  unit(thousand).


number(one, 1).          number(two, 2).          number(three, 3).
number(four, 4).         number(five, 5).         number(six, 6).
number(seven, 7).        number(eight, 8).        number(nine, 9).
number(ten, 10).         number(eleven, 11).      number(twelve, 12).
number(thirteen, 13).

        /* ************************************************************* */


        /* ************************************************************* */
        /*       FILE     : process.pl                                   */
        /*       PURPOSE  : defines the top_level predicate of the       */
        /*                  system.                                      */
        /* ************************************************************* */

        /* E X P O R T S */

:- module(process, [     % commands for the top-level menu.
                e/0,
                n/0,
                pg/0,
                pl/0,
                cg/0,
                cl/0,
                q/0,
                t/0,
```

```
                nt/0,
                md/0
                ]).

        /* I M P O R T S */

:- use_module(define_lfg).              % defines LFG operators.

:- use_module(reader, [                 % reads user queries.
                read_input/1            % (-Word_list).
                ]).

:- use_module(del_mod, [                % deletes predicates in a module.
                del_mod/1               % (+module_name).
                ]).

:- use_module(parser, [                 % parses a list of words.
                bottom_up_parse/2       % (+Words, -F_structure).
                ]).

:- use_module(execute,  [
                execute/1
                ]).

:- use_module(translate, [              % produce a semantic representation.
                translate/2
                ]).

:- use_module(make_static, [
                statisize/3
                ]).

:- use_module(pre_execute, [            % simplify & plan query.
                pre_execute/2
                ]).

:- use_module(traces, [
                trace_point/1,
                trace_info/2
                ]).

:- use_module(gram_pp, [                % pre-process a grammar.
                create_grammar/0
                ]).

:- use_module(lex_pp, [                 % pre-process a lexicon.
                build_dictionary/2
                ]).

:- use_module(graphic, [
                line/3
                ]).

:- use_module(db_meta, [
                produce_meta/1
                ]).

        /* P R E D I C A T E S */
%1
md :-   format('~N~*|~w', [6, 'File Containing Database Spec. ? : ']),
        read(File), nl, produce_meta(File).

%1
t :-    traces:trace_on, !,
        format('~N~*|~w~2n', [6, 'Tracing is already on !']).
%2
t :-    asserta(traces:trace_on), !,
```

```
        format('~N~*|~w~n', [6, 'Tracing is turned on.']).

 %1
nt :-   retract(traces:trace_on), !,
        format('~N~*|~w~n', [6, 'Tracing is turned off.']).
 %2
nt :-   format('~N~*|~w~n', [6, 'Tracing is not switched on !']).

 %1
e :-    read_input(Words), line('-', 6, 30), do_query(Words).

 %1
n :-    format('~*|~w', [10, 'Number (1-23) ? > ']),
        prompt(Old, '       Number (1-23) ? > '),
        read(N), line('-', 6, 30), prompt(_, Old),
         ( query(N, Words), !, format('~*|~w~2n', [6, Words]), do_query(Words)
        |
           format('~N~*|~w~n', [10, 'Unknown query number !']) ).

 %1
do_query(Words) :-
        prompt(_, ' Error (type CNTRL-C and then "a") > '),
        trace_point(pre_parse),
        bottom_up_parse(Words, Functional_structure),
        trace_info(fs, Functional_structure),
         ( trace_point(pre_trans),
           translate(Functional_structure, Semantic_rep),
           trace_info(s_rep, Semantic_rep),
            ( trace_point(pre_plan),
              pre_execute(Semantic_rep, Semantic_rep1),
              trace_info(p_rep, Semantic_rep1),
              trace_point(pre_exec),
              execute(Semantic_rep1)
           |
              format('~*|~w~2n', [6, 'Execution against database failed']),
              line('-', 6, 30)
            )
        |
           format('~*|~w~2n', [10, 'Semantic translation failed']),
           line('-', 6, 30)
         ).

 %1
pg :-   load_grammar(none, Num_grams), !,
         ( Num_grams = none, ! | create_grammar, ! ).

 %1
pl :-   format('~*|~w', [6, 'lexicon file (q to quit) : ']),
        read(L_file), line('-', 6, 30),
        L_file \== q,
        may_delete_old_lexicon,
        open(L_file, read, L_stream),
        build_dictionary(L_file, L_stream), !,
        close(L_stream),
        pl.
 %2
pl :- format('~*|~w~n', [6, 'ok quit']), !.

 %1
load_grammar(_, Grams1) :-
        format('~*|~w', [6, 'grammar file (q to quit) : ']),
        read(G_file), line('-', 6, 30),
        G_file \== q,
        may_delete_old_grammar,
        open(G_file, read, G_stream),
        read(G_stream, Rule),
        read_rest_rules(G_stream, Rule, 0),
```

```
        close(G_stream),
        load_grammar(one, Grams1).
 %2
load_grammar(Grams, Grams) :-
        format('~*|~w~n', [6, 'ok quit']), line('-', 6, 30), !.

 %1
read_rest_rules(_stream, end_of_file, Num) :- !,
        format('~n~*|~w~w~w~n',
                [6, 'Rules Syntactically Ok : ', Num, ' rules in all.']),
        line('-', 6, 30).
 %2
read_rest_rules(Stream, Rule, Num) :-
        asserta(gram_pp:Rule),
        Num1 is Num + 1,
        read(Stream, Next_rule),
        read_rest_rules(Stream, Next_rule, Num1).

 %1
may_delete_old_grammar :-
        prolog_flag(unknown, _, fail),
        rules_exist,
        format('~*|~w', [6, 'delete existing rules (y/n) : ']),
        read(Ans), line('-', 6, 30),
        Ans == y,
        retractall(parser:rules_which_cat(_, _)),
        retractall(parser:rules_which_word(_, _)),
        retractall(lookup:dict(grammar, _, _, [])),
        delete_grammar_words,
        prolog_flag(unknown, _, trace).
 %2
may_delete_old_grammar :- prolog_flag(unknown, _, trace).

 %1
rules_exist :- parser:rules_which_cat(_, _), !.

 %1
delete_grammar_words :-
        retract(lookup:dict(grammar, W, Chrs, Entries)),
        asserta(lookup:dict(not_in_grammar, W, Chrs, Entries)),
        delete_grammar_words.
 %2
delete_grammar_words :- !.

 %1
may_delete_old_lexicon :-
        prolog_flag(unknown, _, fail),
        lexical_entries_exist,
        format('~*|~w', [6, 'delete existing lexical entries (y/n) : ']),
        read(Ans), line('-', 6, 30),
        Ans == y,
        delete_lexical_words,
        prolog_flag(unknown, _, trace).
 %2
may_delete_old_lexicon :- prolog_flag(unknown, _, trace).

 %1
delete_lexical_words :-
        lookup:dict(grammar, W, Chrs, Entries),
        Entries \== [],
        retract(lookup:dict(grammar, W, Chrs, Entries)),
        asserta(lookup:dict(grammar, W, Chrs, [])),
        delete_lexical_words.
 %2
delete_lexical_words :- retractall(lookup:dict(not_in_grammar, _, _, _)).

 %1     existing lexical entries.
```

```
lexical_entries_exist :- lookup:dict(not_in_grammar, _, _, _), !.

 %1
cg :-   format('~*|~w',
                [6, 'Compiler output file for rules ? (q to quit) : ']),
        read(R_file), format('~N~6+~45t~30+~n', []),
        R_file \== q, !,
        format('~*|~w~n', [6, 'Deleting Grammar Pre-processor']),
        del_mod(gram_pp),
        statisize(parser, [parser:rules_which_cat(_, _)], R_file), !,
        format('~*|~w', [6, 'Compiler output file for links ? : ']),
        read(L_file), line('-', 6, 30),
        statisize(make_links, [make_links:links(_,_)], L_file).
 %2
cg :- !, garbage_collect.

 %1
cl :-   format('~*|~w',
                [6, 'Compiler output file for lexicon ? (q to quit) : ']),
        read(C_file),
        line('-', 6, 30),
        C_file \== q, !,
        format('~*|~w~n', [6, 'Deleting Lexicon Pre-processor']),
        del_mod(lex_pp),
        statisize(lookup, [lookup:dict(_, _, _, _)], C_file).
 %2
cl :- !, garbage_collect.

 %1
q :-    format('~*|~w~n', [6, 'Ok goodbye']), line('_', 6, 62).

        /* ************************************************************* */

        % The twenty-three test queries taken from Chat-80.

query( 1, [what,rivers,are,there]).
query( 2, [does,afghanistan,border,china]).
query( 3, [what,is,the,capital,of,'upper_volta']).
query( 4, [where,is,the,largest,country]).
query( 5, [which,countries,are,european]).
query( 6, [which,country,'''s',capital,is,london]).
query( 7, [which,is,the,largest,african,country]).
query( 8, [how,large,is,the,smallest,american,country]).
query( 9, [what,is,the,ocean,that,borders,african,countries,and,
                that,borders,american,countries]).
query(10, [what,are,the,capitals,of,the,countries,bordering,the,baltic]).
query(11, [which,countries,are,bordered,by,two,seas]).
query(12, [how,many,countries,does,the,danube,flow,through]).
query(13, [what,is,the,total,area,of,the,countries,south,of,the,equator,
                and,not,in,australasia]).
query(14, [what,is,the,average,area,of,the,countries,in,each,continent]).
query(15, [is,there,more,than,one,country,in,each,continent]).
query(16, [is,there,some,ocean,that,does,not,border,any,country]).
query(17, [what,are,the,countries,from,which,a,river,flows,
                into,the,'black_sea']).
query(18, [what,are,the,continents,no,country,in,which,contains,more,than,
                two,cities,whose,population,exceeds,1,million]).
query(19, [which,country,bordering,the,mediterranean,borders,a,country,that,
                is,bordered,by,a,country,whose,population,exceeds,
                the,population,of,india]).
query(20, [which,countries,have,a,population,exceeding,10,million]).
query(21, [which,countries,with,a,population,exceeding,10,million,border,
                the,atlantic]).
query(22, [what,percentage,of,countries,border,each,ocean]).
query(23, [what,countries,are,there,in,europe]).

        /* ************************************************************* */
```

```
        /* ********************************************************** */
        /*      FILE    : quants.pl                                   */
        /*      PURPOSE : functional quantifier scoping.              */
        /* ********************************************************** */

        /* E X P O R T S */

:- module(quants, [
                quant_order/9
                ]).

        /* I M P O R T S */

:- use_module(fs_basics, [
                member_fs/2,
                insert_fs/3,
                delete_fs/3
                ]).

        /* P R E D I C A T E S */

 %1
quant_order(Fs, Slots, Neg, Adjuncts, Conjs, [/*oslots*/], _, Slots1, Fs1) :-
        order_slots(Fs, Slots, Fs1, Slots2),
        neg(Neg, Slots2, Slots3),
        sets(Adjuncts, Conjs, Slots3, Slots1).
 %2
quant_order(Fs, Slots, Neg, Adjuncts, Conjs,
                                [pied = V:T], _, [pied = V:T|Slots1], Fs1) :-
        order_slots(Fs, Slots, Fs1, Slots2),
        neg(Neg, Slots2, Slots3),
        sets(Adjuncts, Conjs, Slots3, Slots1).
 %3
quant_order(Fs, Slots, Neg, Adjuncts, Conjs,
                                [rel_adj], _, [rel_adj=_:_|Slots1], Fs1) :-
        order_slots(Fs, Slots, Fs1, Slots2),
        neg(Neg, Slots2, Slots3),
        sets(Adjuncts, Conjs, Slots3, Slots1).

 %4     relative (head+mod) or pied-piped (pied).
quant_order(_fs, [/*slots*/], _neg, _adjuncts, _conjs,
                        Other_slots, Other_fs, Other_slots, Other_fs).

 %1
order_slots(Fs, A = B, Fs, A = B).
 %2
order_slots(Fs, Slots, Fs1, Slots1) :-
        find_strong(Fs, Slots, Fs1, Desc),
        order([f|Slots], [weak|Desc], Slots1).  % top level quantifier.

        % strong quantifiers.
strong each.    strong any.

        % weak quantifiers (including no quantifier).
weak the.       weak wh.        weak a.         weak no.
weak numberof.  weak some.      weak _.         weak [/*none*/].


 %1     find_strong(+Fs, +Slots, -Fs1, -Desc).
find_strong(Fs, _ = _, Fs, []) :- !.
 %2
find_strong(Fs, [], Fs, []) :- !.
 %3
find_strong(Fs, [F - F1 = _:_|R], Fs1, Desc) :-
        delete_fs(F = fs If ind Stf stype
                                Slotsf^Quantf^Semf^Varf^Fsf^Ftf, Fs, Fs2),
        delete_fs(F1 = fs If1 ind Stf1 stype
```

```
                        Slotsfl^Quantfl^Semfl^Varfl^Fsfl^Ftfl, Fsf, Fsf2),
        order_sub(Fsfl, Quantfl, Slotsfl, Fsf2, Slotsf2, Det),
        ( Det == strong, !, Desc = [strong|Descl] | Desc = [weak|Descl] ),
        insert_fs(Fl = fs Ifl ind Stfl stype
                        Slotsf2^Quantfl^Semfl^Varfl^Fsf2^Ftfl, Fsf2, Fsf3),
        insert_fs(F = fs If ind Stf stype
                                Slotsf^Quantf^Semf^Varf^Fsf3^Ftf, Fs2, Fs3),
        find_strong(Fs3, R, Fsl, Descl).
 %4
find_strong(Fs, [F = _:_|R], Fsl, Desc) :-
        delete_fs(F = fs If ind Fst stype
                                Slotsf^Quantf^Semf^Varf^Fsf^Ftf, Fs, Fs2),
        order_sub(Fsf, Quantf, Slotsf, Fsfl, Slotsfl, Det),
        ( Det == strong, !, Desc = [strong|Descl] | Desc = [weak|Descl] ),
        insert_fs(F = fs If ind Fst stype
                                Slotsfl^Quantf^Semf^Varf^Fsfl^Ftf, Fs2, Fs3),
        find_strong(Fs3, R, Fsl, Descl).

 %5     in the case of adjuncts the function may not be present.
find_strong(Fs, [_ = _:_|R], Fsl, [weak|Desc]) :-
        find_strong(Fs, R, Fsl, Desc).


 %1     order_sub(+Fs, +Quant, +Slots, -Fsl, -Slotsl, -Det).
order_sub(Fs, _, A = B, Fs, A = B, weak).
 %2
order_sub(Fs, Quant, [f], Fs, [f], Type) :-
        ( strong Quant, !, Type = strong | Type = weak ).
 %3
order_sub(Fs, Quant, Slots, Fsl, Slotsl, Det) :-
        find_strong(Fs, Slots, Fsl, Desc),
        ( strong Quant, !, D = strong | D = weak),
        order([f|Slots], [D|Desc], Slotsl),
        ( Desc = [strong|_], !, Det = strong | Det = weak).


        % quantifier scoping according to function.
 %1
order(A = B, _, A = B) :- !.
 %2
order([], _, []) :- !.
 %3
order([S], _, [S]) :- !.
 %4
order([F = V:T, Fl = Vl:Tl], [weak, strong], [Fl = Vl:Tl, F = V:T]) :- !.
 %5
order([f, F = V:T, Fl = Vl:Tl], [weak, strong, weak],
                                        [F = V:T, Fl = Vl:Tl, f]) :- !.
 %6
order(Slots, [strong, weak], Slots) :- !.
 %7
order([F = V:T, Fl = Vl:Tl, F2 = V2:T2], [weak, weak, strong],
                                                [F2 = V2:T2|Slotsl]) :- !,
        default_order([F = V:T, Fl = Vl:Tl], Slotsl).
 %8
order([F = V:T, Fl = Vl:Tl, F2 = V2:T2], [weak, strong, weak],
                                                [Fl = Vl:Tl|Slotsl]) :- !,
        default_order([F = V:T, F2 = V2:T2], Slotsl).
 %9
order([F = V:T, Fl = Vl:Tl, F2 = V2:T2], [strong, weak, weak],
                                                [F = V:T|Slotsl]) :- !,
        default_order([Fl = Vl:Tl, F2 = V2:T2], Slotsl).
 %10
order(Slots, [weak, weak], Slotsl) :- !,
        default_order(Slots, Slotsl).
 %11
order(Slots, [weak, weak, weak], Slotsl) :- !,
```

```
        default_order(Slots, Slots1).
 %12
order(Slots, [weak, weak, weak, weak], Slots1) :- !,
        default_order(Slots, Slots1).

        % default scope orders (when all are weak).
 %1
default_order([subj = V:T, vcomp = V1:T1], [subj = V:T, vcomp = V1:T1]) :- !.
 %2
default_order([obj = V:T, subj = V1:T1], [subj = V1:T1, obj = V:T]) :- !.
 %3
default_order([by-obj = V:T, subj = V1:T1],
                                [subj = V1:T1, by-obj = V:T]) :- !.
 %4
default_order([of-obj = V:T, subj = V1:T1],
                                [subj = V1:T1, of-obj = V:T]) :- !.
 %5
default_order([into-obj = V:T, subj = V1:T1],
                                        [subj = V1:T1, into-obj = V:T]) :-!.
 %6
default_order([subj = V:T, thru-obj = V1:T1],
                                        [subj= V:T, thru-obj = V1:T1]) :-!.
 %7
default_order([head = V:T, mod = V1:T1], [mod = V1:T1, head = V:T]) :- !.
 %8
default_order([into-obj = V:T, obj = V1:T1, subj = V2:T2],
                        [subj = V2:T2, into-obj = V:T, obj = V1:T1]) :- !.
 %9
default_order([obj = V:T, subj = V1:T1, vcomp = V2:T2],
                        [subj = V1:T1, obj = V:T, vcomp = V2:T2]) :- !.
 %10
default_order([f, F1], [F1, f]) :- !.
 %11
default_order([f, subj = V:T, vcomp = V1:T1],
                                        [subj = V:T, vcomp = V1:T1, f]) :- !.
 %12
default_order([f, obj = V:T, subj = V1:T1],
                                        [subj = V1:T1, obj = V:T, f]) :- !.
 %13
default_order([f, by-obj = V:T, subj = V1:T1],
                                        [subj = V1:T1, by-obj = V:T, f]) :- !.
 %14
default_order([f, subj = V:T, thru-obj = V1:T1],
                                        [subj= V:T, thru-obj = V1:T1, f]) :-!.
 %15
default_order([f, of-obj = V:T, subj = V1:T1],
                                        [subj = V1:T1, of-obj = V:T, f]) :- !.
 %16
default_order([f, into-obj = V:T, obj = V1:T1, subj = V2:T2],
                        [subj = V2:T2, into-obj = V:T, obj = V1:T1, f]) :- !.
 %17
default_order([f, obj = V:T, subj = V1:T1, vcomp = V2:T2],
                        [subj = V1:T1, obj = V:T, vcomp = V2:T2, f]) :- !.


        % scope of negation.
 %1
neg([], Slots, Slots).
 %2
neg(neg, [F = V:T], [neg, F = V:T]).
 %3
neg(neg, [subj = V:T, vcomp = V1:T1], [subj = V:T, neg, vcomp = V1:T1]).
 %4
neg(neg, [subj = V:T, obj = V1:T1], [subj = V:T, neg, obj = V1:T1]).
 %5
neg(neg, [F = V:T, f], [neg, F = V:T, f]).
 %6
```

```
neg(neg, [subj = V:T, vcomp = V1:T1, f], [subj = V:T, neg, vcomp = V1:T1, f]).
 %7
neg(neg, [subj = V:T, obj = V1:T1, f], [subj = V:T, neg, obj = V1:T1, f]).


        % add set values to slots.
 %1
sets([/*adjuncts*/], [/*conjs*/], Slots, Slots) :- !.
 %2
sets([], _, Slots, [conjs|Slots]) :- !.
 %3
sets(_, [], Slots, [adjuncts|Slots]) :- !.
 %4
sets(_, _, Slots, [adjuncts, conjs|Slots]).

        /* ************************************************************** */


        /* ************************************************************  */
        /*      FILE    : reader.pl                                       */
        /*      PURPOSE : reads user input from the terminal and          */
        /*                produces a list of words.                       */
        /* ************************************************************* */

        /* E X P O R T S */

:- module(reader, [
                read_input/1
                ]).

        /* I M P O R T S */

:- use_module(library(ctypes), [
                is_alpha/1,             % upper or lower case letter.
                is_csym/1,              % letter, digit or underscore.
                is_digit/1,             % decimal digit.
                is_period/1,            % full-stop.
                is_quote/1,             % single or double quote.
                to_lower/2              % convert upper to lower case.
                ]).

        /* P R E D I C A T E S */

        /* read_input(-Words).
           read user input (command or query) and produce a list of
           words 'Words'. The library definition of read_sent/1 could
           have been used but this does not quite perform as required. */
 %1
read_input(Words) :-
        prompt(_, '         query > '), get(Chr),
        read_rest_command(Chr, Words),
        get0(_), !.                             % throw away <return>.

 %1     given the first character read the rest of a command
read_rest_command(Chr, Words):-
           (    ( is_alpha(Chr) -> read_word(Chr, Word, Chr1)
                |
                  is_digit(Chr) -> read_num(Chr, Word, Chr1)
               ) ->
               Words = [Word|Rest_words], !, read_rest_words(Chr1, Rest_words)
          |
             nl, prompt(_, ' continue > '), read_input(Words) ).

 %1     given the first character read a word
read_word(Chr, Word, Chr2) :-
        get0(Chr1), read_rest_word(Chr1, Rest_chrs, Chr2), !,
         ( word_char(Chr, New_chr) ->
```

```
                name(Word, [New_chr|Rest_chrs])
                |
                name(Word, [Chr|Rest_chrs]) ).

 %1     given the first digit read a number
read_num(Num, Word, Num2) :-
        is_digit(Num), !,
        get0(Num1), read_rest_num(Num1, Rest_num, Num2),
        name(Word,[Num|Rest_num]).

 %1     given the start of a word read the rest
read_rest_word(Chr, Chrs, Chr2) :-
         word_char(Chr, New_chr) ->
                get0(Chr1), read_rest_word(Chr1, Rest_chrs, Chr2), !,
                Chrs = [New_chr|Rest_chrs]
                |
                (Chrs, Chr2) = ([], Chr).

 %1     given the start of a number read the rest
read_rest_num(Num, Num_digits, Num2) :-
         is_digit(Num) ->
                get0(Num1), read_rest_num(Num1, Rest_num, Num2), !,
                Num_digits = [Num|Rest_num]
                |
                (Num_digits, Num2) = ([], Num).

 %1     given the first command word read the rest of the words
read_rest_words(Chr, Words) :-
         is_period(Chr) -> Words = []                    % '.', '?' or '!'.
         |
         is_quote(Chr) ->
                read_word(Chr, Word, Chr1),
                read_rest_words(Chr1, Rest_words), !,
                Words = [Word|Rest_words]
         |
         word_char(Chr, Chr1) ->
                read_word(Chr1, Word, Chr2),
                read_rest_words(Chr2, Rest_words), !,
                Words = [Word|Rest_words]
         |
         is_digit(Chr) ->
                read_num(Chr, Num, Chr1),
                read_rest_words(Chr1, Rest_words), !,
                Words = [Num|Rest_words]
         |
         get(Chr1),             % read to the next significant character.
         read_rest_words(Chr1, Words).


        % test a character to see if it is part of this token and
 %1       convert upper case to lower case.
word_char(Input_chr, New_chr) :-
         is_csym(Input_chr) ->                 % letter, digit or underscore.
                to_lower(Input_chr, New_chr)
                |
                Input_chr == 45 ->             % convert '-' to '_'.
                New_chr = 95.

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : set_of1.pl                                     */
        /*      PURPOSE : collect sets of query solutions.               */
        /*                Prolog database.                               */
        /* ************************************************************ */
```

```
        /* E X P O R T S */

:- module(set_of1, [
                set_of1/3
                ]).

        /* I M P O R T S */

:- use_module(library(findall), [
                findall/3
                ]).

:- use_module(execute, [
                execute/1
                ]).

:- use_module(fast_basics, [
                fmember/2,
                freverse/2
                ]).

        /* P R E D I C A T E S */

set_of1(Var, execute(Preds), Set) :-
        findall(Var, execute:execute(Preds), List), remove_dups(List, Set).


remove_dups([], []) :- !.

remove_dups(List, Set) :- remove_dups(List, [], Set).


remove_dups([], Set, Set1) :- freverse(Set, Set1), !.

remove_dups([First|Rest], Set, Set1) :-
        fmember(First, Set) ->
                 remove_dups(Rest, Set, Set1)
                |
                 remove_dups(Rest, [First|Set], Set1).

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : spelling.pl                                    */
        /*      PURPOSE : attempt to correct the spelling of a word.     */
        /*      NOTES   : spelling correction is attempted by one        */
        /*                character insertion, deletion, substitution,   */
        /*                and word character permutation. Permutations   */
        /*                are only tried on words of 'n' characters      */
        /*                (see 'constant(max_permutation_length(n))').   */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(spelling_corrector, [
                spell/2
                ]).

        /* I M P O R T S */

:- use_module(lookup, [
                lookup/1
                ]).

        /* C O N S T A N T S */
```

```
constant(max_permutation_length(6)).  % restricted length of list permutated.

        /* P R E D I C A T E S */

        /* spell(+W, -Cor).
           produce a single spelling correction 'Cor', of the word, 'W'
           selected by the user from the list of possibly correct words
           produced. */
 %1
spell(Word, New_words) :- name(Word, Word_chrs), convert(Word_chrs, New_words).

        /* correct(+Chrs, -W_list).
           find all the possible words 'W_list' in the dictionary that
           match the list of characters 'Chrs' with a one character
           alteration or character permutation. */
 %1
convert(Word_chrs, Poss_correct_words) :-
        setof(New_words,
                        possible_corrections(Word_chrs, New_words),
                                                        Poss_correct_words).
 %1
possible_corrections(Word_chrs, New_words) :-
        length(Word_chrs, Num_chrs),
        constant(max_permutation_length(Num)),
        Num_chrs < Num ,
        permutations_of(Word_chrs, New_words).
 %2
possible_corrections(Word_chrs, New_words) :-
        add_char_to(Word_chrs, New_words).
 %3
possible_corrections(Word_chrs, New_words) :-
        delete_char_from(Word_chrs, New_words).
 %4
possible_corrections(Word_chrs, New_words) :-
        replace_char_in(Word_chrs, New_words).

 %1     generate character permutation and search lexicon.
permutations_of(Word_chrs, New_word) :-
        perm(Word_chrs, New_chrs),
        lookup(New_chrs), name(New_word, New_chrs).

 %1     add one character to the word and search lexicon.
add_char_to(Word_chrs, New_word) :-
        add_char(Word_chrs, New_chrs),
        lookup(New_chrs), name(New_word, New_chrs).

 %1     delete one character from the word and search lexicon.
delete_char_from(Word_chrs, New_word) :-
        delete_char(Word_chrs, New_chrs),
        lookup(New_chrs), name(New_word, New_chrs).

 %1     replace one character in the word and search lexicon.
replace_char_in(Word_chrs, New_word) :-
        replace_char(Word_chrs, New_chrs),
        lookup(New_chrs), name(New_word, New_chrs).
 %1
add_char(W, [_|W]).                     % add a variable to list.
 %2
add_char([H|T], [H|R]) :- add_char(T, R).

 %1
delete_char([_|T], T).                  % delete a member of a list.
 %2
delete_char([H|T], [H|R]) :- delete_char(T, R).

 %1
replace_char([_|T], [_|T]).             % replace member with variable.
```

```
 %2
replace_char([H|T], [H|R]) :- replace_char(T, R).

        % perm(List, Perm).
        % is true when List and Perm are permutations of each other.
        % The main use of perm/2 is to generate permutations.
 %1
perm([], []).
 %2
perm(List, [First|Perm]) :- select(First, List, Rest), perm(Rest, Perm).

        % select(?Element, ?Set, ?Residue)
        % is true when Set is a list, Element occurs in Set, and Residue is
        % everything in Set except Element (things stay in the same order).
 %1
select(Element, [Element|Rest], Rest).
 %2
select(Element, [Head|Tail], [Head|Rest]) :- select(Element, Tail, Rest).

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : top_level.pl                                  */
        /*      PURPOSE : defines the top_level predicate (loads code). */
        /* ************************************************************ */

        /* E X P O R T S */

        % non-module so as to be visible in user.

        /* I M P O R T S */

:- use_module(process, [            % menu commands :
                e/0,                % execute a query.
                n/0,                % execute a numbered query.
                pg/0,               % pre-process a grammar (files).
                pl/0,               % pre-process a lexicon (files).
                cg/0,               % compile a complete grammar.
                cl/0,               % compile a complete lexicon.
                q/0,                % quit.
                t/0,                % turn tracing on.
                nt/0,               % turn tracing off.
                md/0                % produce meta-data from a DB specification.
                ]).

:- use_module(graphic, [
                line/3              % draws a line of a given character.
                ]).

        /* P R E D I C A T E S */
 %1     the top-level menu.
hi :-   line('-', 6, 62),
        format('~*|~w~n', [6, 'SELECT MODE :']), line('-', 6, 62),
        format('~*|~w~*|~w', [6, 'Enter a query', 26, ':  e']),
        format('~*|~w~*|~w~n', [36, 'Enter a query no.', 58, ':  n']),
        format('~*|~w~*|~w', [6, 'Preprocess grammar', 26, ': pg']),
        format('~*|~w~*|~w~n', [36, 'Preprocess lexicon', 58, ': pl']),
        format('~*|~w~*|~w', [6, 'Compile grammar', 26, ': cg']),
        format('~*|~w~*|~w~n', [36, 'Compile lexicon', 58, ': cl']),
        format('~*|~w~*|~w', [6, 'Turn tracing on', 26, ':  t']),
        format('~*|~w~*|~w~n', [36, 'Turn tracing off', 58, ': nt']),
        format('~*|~w~*|~w', [6, 'Meta-Data on DB', 26, ': md']),
        format('~*|~w~*|~w~n', [36, 'Quit', 58, ':  q']),
        line('-', 6, 62),
        prompt(Old, '        Mode ? > '),
        format('~*|~w', [6, 'Mode ? > ']),
```

```
        read(Mode),
        line('-', 6, 30),
        prompt(_, Old),
          ( legal(Mode),                        % legal command.
            call(Mode), !                       % do the command.
        ;
             ( \+ legal(Mode), !,               % illegal command.
               format('~N~*|~w', [6, 'Please type one of :']),
               format('~N~*|~w~n', [6, 'e, n, pg, pl, cg, cl, t, nt or q !']),
               line('-', 6, 30)
             ;
               true                             % some error, but keep going.
             )
          ), !,
          ( Mode == q                           % stop.
        ;
            garbage_collect, hi                 % go round again.
          ).


        % legal command letters.

legal(e).        legal(n).        legal(pg).       legal(pl).
legal(cg).       legal(cl).       legal(q).        legal(t).
legal(nt).       legal(md).

        % Start-up message.

:- format('~2n~*|~w~2n', [10, 'Ok ready to start session']), hi.

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : traces.pl                                     */
        /*      PURPOSE : produces output tracing information.          *
        /* ************************************************************ */

        /* E X P O R T S */

:- module(traces, [
                trace_point/1,
                trace_info/2,
                trace_on/0
                ]).

        /* I M P O R T S */

:- use_module(pretty, [
                pretty_info/1,
                pretty_sem/1
                ]).

        /* D Y N A M I C S */

:- dynamic trace_on/0.                  % tracing on/off flag.

        /* PREDICATES : */
 %1
trace_point(_) :-                       % each point is labelled but treated
        trace_on, !,                    % the same at this time :
        statistics(runtime, _).         % reset the timer.
 %2
trace_point(_) :- ! .

 %1
trace_info(fs, Fs) :-
```

```
        trace_on, !, statistics(runtime, [_, T]),
        format('~N~n~*|~w~w~2n', [5, 'Parse Time = ', T]),
        format('~N~*|~w~2n', [5, 'Created an F-structure']),
        pretty_info(Fs).
 %2
trace_info(s_rep, Sr) :-
        trace_on, !, statistics(runtime, [_, T]),
        format('~N~n~*|~w~w~2n', [5, 'Translation Time = ', T]),
        format('~N~*|~w~2n', [5, 'Created a semantic representation']),
        pretty_sem(Sr).
 %3
trace_info(p_rep, Sr) :-
        trace_on, !, statistics(runtime, [_, T]),
        format('~N~n~*|~w~w~2n', [5, 'Planning Time = ', T]),
        format('~N~*|~w~2n', [5, 'Representation after Planning']),
        pretty_sem(Sr).
 %4
trace_info(edge, [Sv, Ev, Cat]) :-
        trace_on, !,
        format('~N~*t~w~w~w~w~w~w~n',
                        [5, 'edge ', Sv, ' to ', Ev, ' of category ', Cat]).
 %5
trace_info(_, _) :- !.

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : translate.pl                                   */
        /*      PURPOSE : translates an f-structure into a semantic     */
        /*                representation (query).                       */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(translate, [
                translate/2     % (+F_structure, +Indexes, -Pred_expr).
                ]).

        /* I M P O R T S */

:- use_module(fs_basics, [
                member_fs/2,
                delete_fs/3
                ]).

:- use_module(fast_basics, [
                fmember/2,
                fdelete/3
                ]).

:- use_module(pre_sem, [
                quant_attach/3
                ]).

        % indexed functions are translated eg :
        % 'q' quantifier, 'head' all, 'focus' none (omitted).

:- op(100, xfx, [all, quant, omit]).


        % the list of indexes is biult up during translation of an
        % f-structure where each indexed f-structure adds an element
        % of the form 'Function:Index = Var:Type' to the list.

        /* P R E D I C A T E S */
 %1
```

```
translate(Fs, Rep) :-
        quant_attach([], Fs, New_rep), !,
        preds(New_rep, [], [/*indexes*/], Rep).

        % yes/no question with translation of single sub-function.
 %1       here the question has adjuncts/negation which must be translated
        % before passing to the sub-function.
preds([] ind _ stype Slots^y_n^passto(F)^(Fvar:_)^Fs^_, Vars, I, y_n(R)) :- !,
        passto(Slots, Slots1),
        slots(more, fs, Slots1, Fvar:_, Fs, Vars, Vars1,
                                [/*null*/], Preds1, R, Rest1, I, I1),
        member_fs(F = Fvar:_, Slots),
        member_fs(F = fs Findex ind _ stype
                        Fslots^Fquant^Fsem^(Fvar:_)^Ffs^Fffs, Fs),
        slot(last, F, Fvar:_, Findex ind _ stype Fslots^Fquant^Fsem^
                                (Fvar:_)^Ffs^Fffs, Vars1, _,
                                Preds1, Preds2, Rest1, Preds2, I1, _).
 %2
preds([] ind _ stype Slots^y_n^passto(F)^(Fvar:_)^Fs^_, Vars, I, y_n(R)) :- !,
        passto(Slots, Slots1),
        slots(more, fs, Slots1, Fvar:_, Fs, Vars, Vars1,
                                [/*null*/], Preds1, R, Rest1, I, I1),
        join_rest(Preds1, Rest1, Rest2),
        member_fs(F = fs Val, Fs),
        preds(Val, Vars1, I1, Rest2).
 %3
preds([] ind _ stype Slots^y_n^null(F)^(Var:T)^Fs^Feats,
                                                Vars, I, y_n(R)) :- !,
        member_fs(F = fs Val, Fs),
        null(Val, Adjuncts),
        adjuncts(more, Adjuncts, Var:T, Vars, Vars1,
                                [/*null*/], Preds2, R, Resta, I, I2),
        fdelete(F = _:_, Slots, Slots1),
        sub_slots(last, [[],[]|Feats], fs, Slots1, Var:T, Fs,
                        Vars1, _, Preds2, Preds3, Resta, Preds3, I2, _).

 %4       yes/no question with semantic component at top level and
        % a number of sub-functions.
preds([] ind _ stype Slots^y_n^Sem^(Var:T)^Fs^Feats, Vars, I, y_n(R)) :- !,
        sub_slots(last, [[],Sem|Feats], fs, Slots, Var:T, Fs,
                                        Vars, _, [], _, R, _, I, _).

 %5       wh-fronted question with translation of single sub-function.
preds([] ind _ stype Slots^q^null(F)^(Fvar:Type)^Fs^Feats,
                                                        Vars, I, Repo) :- !,
        member_fs(q = fs Qi ind _ stype _^Qq^_^(Qvar:Qtype)^_^_, Fs),
        member_fs(focus = fs Fi ind _ stype _^_^_^(Fvar:Ftype)^_^_, Fs),
        member_fs(F = fs Val, Fs),
        null(Val, Adjuncts),
        adjuncts(more, Adjuncts, Fvar:_, Vars, Vars1,
                                [/*null*/], Preds2, Repo, Rest2, I, Ia),
        fdelete(F = _:_, Slots, Slots1),
        sub_slots(last, [Qq,[]|Feats], fs, Slots1, Fvar:Type, Fs, Vars1, _,
                        Preds2, Preds3, Rest2, Preds3, [Qi = q quant
                        Qvar:Qtype, Fi = focus omit Fvar:Ftype|Ia], _).

 %6       wh-fronted question with translation of single sub-function.
preds([] ind _ stype Slots^q^passto(F)^(Fvar:Type)^Fs^
                        [[], Num, Int, Adj, Aggs], Vars, I, Repo) :- !,
        member_fs(q = fs Qi ind _ stype _^Qq^_^(Qvar:Qtype)^_^_, Fs),
        member_fs(focus = fs Fi ind _ stype _^_^_^(Fvar:Ftype)^_^_, Fs),
        passto(Slots, Slots1),
        slots(more, fs, Slots1, Fvar:_, Fs, Vars, Vars1,
                        [/*null*/], Preds2, Rep1, Rest2,
                [Qi = q quant Qvar:Qtype, Fi = focus omit Fvar:Ftype|I], I1),
        member_fs(F = fs Val, Fs),
        preds(Val, Vars1, I1, Rep2),
```

```
        join(Rep2, Preds2, Rest2),
        trans(last, fs, Fs, Qq, Qvar:Type, [/*sem*/], Num,
                        Int, Adj, Aggs, Vars1, Rep1, Preds3, Repo, Preds3).

%7      wh_fronted question with equative 'be' note that the function's
        % types are the same.
preds([] ind _ stype [F = V:T, F1 = V:T,f]^q^equate(_comp,subj)^_^Fs^
                        [[], Num, Int, Adj, Aggs], Vars, I, Repo) :- !,
        member_fs(q = fs Qi ind _ stype _^Qq^_^(Qvar:Qtype)^_^_, Fs),
        member_fs(focus = fs Fi ind _ stype _^_^_^(Fvar:Ftype)^_^_, Fs),

        member_fs(F = fs Valf, Fs), member_fs(F1 = fs Valf1, Fs),
        slot(last, F, V:T, Valf, Vars, Vars1, [], Preds1, Rep1, Preds1,
                [Qi = q quant Qvar:Qtype, Fi = focus omit Fvar:Ftype|I], I1),
        slot(last, F1, V:T, Valf1, Vars1, _, [], Preds2, Rep2, Preds2,I1, _),
        join(Rep2, Rep1, Rep),
        trans(last, fs, Fs, Qq, Qvar:Qtype, [/*null*/],
                        Num, Int, Adj, Aggs, Vars1, Rep, _, Repo, _).

%8      wh-fronted with sub-functions.
preds([] ind _ stype Slots^q^Sem^Var^Fs^
                        [[], Num, Int, Adj, Aggs], Vars, I, Repo) :- !,
        member_fs(q = fs Qi ind _ stype _^Qq^_^(Qvar:Qtype)^_^_, Fs),
        member_fs(focus = fs Fi ind _ stype _^_^_^(Fvar:Ftype)^_^_, Fs),
        slots(more, fs, Slots, Var, Fs, Vars, Vars1, [/*null*/],
                Preds1, Rep, Rest1, [Qi = q quant Qvar:Qtype,
                                Fi = focus omit Fvar:Ftype|I], _),
        join(Preds1, Sem, Rest1),
        trans(last, fs, Fs, Qq, Qvar:Qtype, [/*null*/],
                        Num, Int, Adj, Aggs, Vars1, Rep, _, Repo, _).

%9      function with a number of sub-functions, usually this will be
        % a sub-function itself which is to be the only function translated
        % after a 'passto function' directive.
preds([] ind _ stype Slots^Quant^Sem^(Var:T)^Fs^Feats, Vars, I, Rep) :- !,
        sub_slots(last, [Quant,Sem|Feats], fs, Slots, Var:T, Fs,
                        Vars, _, [/*null*/], Preds3, Rep, Preds3, I, _).

%10     an indexed structure the index of which has been found already,
        % this must a function some of which has been translated :
        % 'head' : all, 'q' : quantifier.
preds(Index ind _ stype [/*slots*/]^Quant^Sem^Var^Fs^Feats,
                                                Vars, I, Rep) :- !,
        index_means(fs, Index, []^Quant^Sem^Var^Fs^Feats,
                        []^Quanta^Sema^Vara^Fsa^[Num,Int,Adj,Aggs], I, _),
        trans(last, fs, Fsa, Quanta, Vara, Sema,
                        Num, Int, Adj, Aggs, Vars, [/*null*/], _, Rep, _).

%11     as above but with sub-functions.
preds(Index ind _ stype Slots^Quant^Sem^Var^Fs^Feats, Vars, I, Rep) :- !,
        index_means(fs, Index, Slots^Quant^Sem^Var^Fs^Feats,
                                Slotsa^Quanta^Sema^Vara^Fsa^Featsa, I, I1),
        sub_slots(last, [Quanta,Sema|Featsa], fs, Slotsa, Vara, Fsa,
                        Vars, _, [/*null*/], Preds1, Rep, Preds1, I1, _).


%1      remove semantic content of f-structure with 'form' feature
        % but not attached adjuncts.
null(_Index ind _ stype _Slots^_Quant^_Sem^_Var^Fs^_Feats, Vals) :-
        ( member_fs(adjuncts = set Vals, Fs), ! | Vals = []).

%1      change the content of an f-structure to reflect the translation
        % state of its index.
index_means(F, Num, Info, Info1, I, I1) :-
        fdelete(Num = Desc, I, I2), !,
        indexed(Desc, Info, Info1),
        new_index(F, Num = Desc, I2, I1).
```

```
 %2       a new index.
index_means(F, Num, Slots^Quant^Sem^(Var:Type)^Fs^Feats, Slots^Quant^
                         Sem^(Var:Type)^Fs^Feats, I, [Num = F all Var:Type|I]).

 %1
indexed(_ quant Var:Type, Slots^_^Sem^_^Fs^Feats,
                         Slots^[/*quant*/]^Sem^(Var:Type)^Fs^Feats) :- !.
 %2
indexed(_ all Var:Type, _^_^_^_^Fs^_,
                         [/*slots*/]^[/*quant*/]^[/*sem*/]^
                                        (Var:Type)^Fs^[[],[],[],[],[]]) :- !.
 %3
indexed(_ omit Var:Type, Slots^Quant^Sem^(Var:Type)^Fs^Feats,
                              Slots^Quant^Sem^(Var:Type)^Fs^Feats) :- !.

 %1
new_index(F, Num = _ quant Var:Type, I, [Num = F all Var:Type|I]) :- !.
 %2
new_index(F, Num = _ omit Var:Type, I, [Num = F all Var:Type|I]) :- !.
 %3
new_index(F, Num = _ all Var:Type, I, [Num = F all Var:Type|I]) :- !.


        % passto(+Slots, -Slots1).
        % before passing to a subfunction the set functions and negation
        % are attended to, here the slots are adjusted to contain only
        % these.
 %1
passto([], []) :- !.
 %2
passto([adjuncts|R], [adjuncts|R1]) :- passto(R, R1).
 %3
passto([con|R], [con|R1]) :- passto(R, R1).
 %4
passto([neg|R], [neg|R1]) :- passto(R, R1)..
 %5
passto([rel_adj=_:_|R], [rel_adj=_:_|R1]) :- passto(R, R1).
 %6
passto([_ = _:_|R], R1) :- passto(R, R1).
 %7
passto([f|R], R1) :- passto(R, R1).

        % translate the sub-functions listed in 'slots'.
        % slots(+more_last, +Slots, ^+Var, +Fs,
        %                         +Preds, -Preds1, Rest, -Rest1, +I, -I1).
 %1
slots(_ml, _, [/*slots*/], _, _, Vars, Vars,
                          Preds, Preds, Rest, Rest, I, I) :- !.
 %2
slots(_ml, _, _ = _, _, _, Vars, Vars, Preds, Preds, Rest, Rest, I, I) :- !.
 %3
slots(last, _, [neg], _, _, Vars, Vars, Preds, [/*null*/],
                                         \+ { Preds }, _, I, I) :- !.
 %4
slots(more, _, [neg], _, _, Vars, Vars, Preds, [/*null*/],
                                  \+ { Rest }, Rest1, I, I) :- !,
        join_rest(Preds, Rest, Rest1).
 %5
slots(last, F, [neg|Rest_slots], Var, Fs, Vars, Vars1,
                          Preds, [/*null*/], \+ { Rest }, _, I, I1) :-
        slots(last, F, Rest_slots, Var, Fs, Vars, Vars1,
                          Preds, Preds1, Rest, Preds1, I, I1).
 %6
slots(more, F, [neg|Rest_slots], Var, Fs, Vars, Vars1,
                          Preds, \+ { Preds2 }, Rest, Rest1, I, I1) :-
        slots(more, F, Rest_slots, Var, Fs, Vars, Vars1,
                          [], Preds1, Rest, Rest1, I, I1),
```

```
        join(Preds, Preds1, Preds2).
 %7
slots(M1, _, [adjuncts], Var, Fs, Vars, Vars1,
                                        Preds, Preds1, Rest, Rest1, I, I1) :-
        member_fs(adjuncts = set Vals, Fs),
        adjuncts(M1, Vals, Var, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1).
 %8
slots(M1, F, [adjuncts|Rest_slots], Var, Fs, Vars, Vars1,
                                        Preds, Preds1, Rest, Rest1, I, I1) :-
        member_fs(adjuncts = set Vals, Fs),
        adjuncts(more, Vals, Var, Vars, Vars2,
                                Preds, Preds2, Rest, Rest2, I, I2),
        slots(M1, F, Rest_slots, Var, Fs, Vars2, Vars1,
                                Preds2, Preds1, Rest2, Rest1, I2, I1).
 %9
slots(M1, _f, [conjs], Var, Fs, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1) :-
        member_fs(conjs = set Vals, Fs),
        conjs(M1, Vals, Var, Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1).
 %10
slots(M1, F, [conjs|Rest_slots], Var, Fs, Vars, Vars1,
                                        Preds, Preds1, Rest, Rest1, I, I1) :-
        member_fs(conjs = set Vals, Fs),
        conjs(more, Vals, Var, Vars, Vars2, Preds, Preds2, Rest, Rest2, I, I2),
        slots(M1, F, Rest_slots, Var, Fs, Vars2, Vars1,
                                Preds2, Preds1, Rest2, Rest1, I2, I1).

        % in the case of an adjunct a single function may be missing
        % from the f-structure.
 %11
slots(M1, adjunc, [F1-Subf = Var1:T1], _var:_type, Fs,
                Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :- !,
         ( member_fs(F1 = fs [] ind _ stype _^_^_^Sub_fs^_, Fs), !,
           member_fs(Subf = fs Val, Sub_fs),
           slot(M1, F1-Subf, Var1:T1, Val, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1)
         |
           [Vars,Preds,Rest,I] = [Vars1,Preds1,Rest1,I1]  % function missing.
         ).
 %12
slots(M1, _f, [F1-Subf = Var1:T1], _var:_type, Fs, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1) :- !,
        member_fs(F1 = fs [] ind _ stype _^_^_^Sub_fs^_, Fs),
        member_fs(Subf = fs Val, Sub_fs),
        slot(M1, F1-Subf, Var1:T1, Val, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1).
 %13
slots(M1, adjunc, [F1-Subf = Var1:T1|Rest_slots], Var:T, Fs,
                        Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :- !,
         ( delete_fs(F1 = fs [] ind _ stype _^_^_^Sub_fs^_, Fs, Fs2), !,
           member_fs(Subf = fs Val, Sub_fs),
           slot(more, F1-Subf, Var1:T1, Val, Vars, Vars2,
                                        Preds, Preds2, Rest, Rest2, I, I2),
           slots(M1, adjunc, Rest_slots, Var:T, Fs2, Vars2, Vars1,
                                        Preds2, Preds1, Rest2, Rest1, I2, I1)
         |
           slots(M1, adjunc, Rest_slots, Var:T, Fs2, Vars, Vars1,
                                        Preds, Preds1, Rest, Rest1, I, I1)
         ).
 %14
slots(M1, F, [F1-Subf = Var1:T1|Rest_slots], Var:T, Fs,
                        Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :- !,
        delete_fs(F1 = fs [] ind _ stype _^_^_^Sub_fs^_, Fs, Fs2),
        member_fs(Subf = fs Val, Sub_fs),
        slot(more, F1-Subf, Var1:T1, Val, Vars, Vars2,
                                Preds, Preds2, Rest, Rest2, I, I2),
```

```
        slots(M1, F, Rest_slots, Var:T, Fs2, Vars2, Vars1,
                        Preds2, Preds1, Rest2, Rest1, I2, I1).
 %15
slots(last, adjunc, [F1 = Var1:T1], _var, Fs, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1) :- !,
         ( member_fs(F1 = fs Val, Fs), !,
           slot(last, adjunc, Var1:T1, Val, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1)
        |
           [Vars,Preds,Rest,I] = [Vars1,Preds1,Rest1,I1]
         ).
 %16
slots(last, F, [F1 = Var1:T1], _var, Fs, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1) :- !,
        member_fs(F1 = fs Val, Fs),
        slot(last, F, Var1:T1, Val, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1).
 %17
slots(more, adjunc, [F1 = Var1:T1], _var, Fs, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1) :- !,
         ( member_fs(F1 = fs Val, Fs), !,
           slot(more, F1, Var1:T1, Val, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1)
        |
           [Vars,Preds,Rest,I] = [Vars1,Preds1,Rest1,I1]
         ).
 %18
slots(more, _, [F1 = Var1:T1], _var, Fs, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1) :- !,
        member_fs(F1 = fs Val, Fs),
        slot(more, F1, Var1:T1, Val, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1).

 %19      makes a function's variable equal to the head of relative.
        % also note that the superior function name is passed down
        % as this function's name & a head although indexed is always
        % translated in its entirety with one exception : if the head's
        % quantifier is that of the wh-front function 'q'.

slots(M1, F, [head = Var:T1|Rest_slots], Var:T, Fs, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1) :- !,
        member_fs(head = fs Hi ind _ stype
                        Hslots^Hquant^Hsem^Hvar^Hfs^Hfeats, Fs),
        index_means(head, Hi, Hslots^Hquant^Hsem^Hvar^Hfs^Hfeats,
                        Hslotsa^Hquanta^Hsema^Hvara^Hfsa^Hfeatsa, I, I2),
        slot(more, F, Var:T1, [/*index*/] ind _ stype
                        Hslotsa^Hquanta^Hsema^Hvara^Hfsa^Hfeatsa,
                        Vars, Vars2, Preds, Preds2, Rest, Rest2, I2, I3),
        join_rest(Preds2, Rest2, Rest3),
        slots(M1, F, Rest_slots, Var:T, Fs, Vars2, Vars1,
                        [/*null*/], Preds1, Rest3, Rest1, I3, I1).

 %20
slots(M1, F, [pied = Var:T1|Rest_slots], Var:_T, Fs, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1) :- !,
        member_fs(pied = fs Pi ind _ stype
                        Pslots^Pquant^Psem^(Var:Tp)^Pfs^Pfeats, Fs),
        ( Pslots == [] | Pslots = [_f = Var:_t|_r] ),
        slot(more, F, Var:T1, [] ind _ stype
                Pslots^Pquant^Psem^(Var:Tp)^Pfs^Pfeats, Vars, Vars2,
                Preds, Preds2, Rest, Rest2, [Pi = pied all Var:Tp|I], I2),
        join_rest(Preds2, Rest2, Rest3),
        slots(M1, F, Rest_slots, Var:T1, Fs, Vars2, Vars1,
                        [], Preds1, Rest3, Rest1, I2, I1).
 %21
slots(M1, adjunc, [F1 = Var1:T1|Rest_slots], Var:T, Fs,
                        Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :- !,
```

```
          ( member_fs(F1 = fs Val, Fs), !,
            slot(more, F1, Var1:T1, Val, Vars, Vars2,
                                  Preds, Preds2, Rest, Rest2, I, I2),
            slots(M1, adjunc, Rest_slots, Var:T, Fs,
                         Vars2, Vars1, Preds2, Preds1, Rest2, Rest1, I2, I1)
          |
            slots(M1, adjunc, Rest_slots, Var:T, Fs,
                         Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1)
          ).
 %22
slots(M1, F, [F1 = Var1:T1,f], Var:T, Fs, Vars, Vars1,
                                  Preds, Preds1, Rest, Rest1, I, I1) :- !,
        member_fs(F1 = fs Val, Fs),
        slot(M1, F1, Var1:T1, Val, Vars, Vars2,
                                  Preds, Preds2, Rest, Rest2, I, I2),
        slots(M1, F, [f], Var:T, Fs, Vars2, Vars1,
                         Preds2, Preds1, Rest2, Rest1, I2, I1).
 %22
slots(M1, F, [F1 = Var1:T1|Rest_slots], Var:T, Fs, Vars, Vars1,
                                  Preds, Preds1, Rest, Rest1, I, I1) :- !,
        member_fs(F1 = fs Val, Fs),
        slot(more, F1, Var1:T1, Val, Vars, Vars2,
                                  Preds, Preds2, Rest, Rest2, I, I2),
        slots(M1, F, Rest_slots, Var:T, Fs, Vars2, Vars1,
                         Preds2, Preds1, Rest2, Rest1, I2, I1).
 %23
slots(_, _, [f], _, _, Vars, Vars, Preds, Preds, Rest, Rest, I, I).


        % slot(+last_more, +F, +Var:Type, +Val, +Vars, -Vars1,
        %                         +Preds, -Preds1, +^Rest, -^Rest1, +I, -I1).
 %1
slot(Last_more, _, Var:_, [] ind _ stype Slots^_^passto(F)^(Var:Type)^Fs^
                [_Pro,_num,_int,_adj,_aggs], Vars, [Var|Vars3],
                                         Preds, Preds1, Rest, Rest1, I, I1) :-
        passto(Slots, Slots1),
        slots(more, fs, Slots1, Var:_, Fs, Vars, Vars2,
                                         Preds, Preds2, Rest, Rest2, I, I2),
        member_fs(F = fs Val, Fs),
        slot(Last_more, F, Var:Type, Val, Vars2, Vars3,
                                  Preds2, Preds1, Rest2, Rest1, I2, I1).

        % sub-function which is simple proper-noun, which forms an
        % argument value.
 %2     last & simple constant (from proper noun).
slot(last, _, Sem:_t,
                [] ind _ stype (domain = D)^[]^Sem^(Sem:D)^_^_,
                                  Vars, Vars, Preds, [], Preds, _, I, I) :- !.
 %3     more & simple constant (from proper noun).
slot(more, _, Sem:_type,
                [] ind _ stype (domain = D)^[]^Sem^(Sem:D)^_^_,
                         Vars, Vars, Preds, Preds, Rest, Rest, I, I) :- !.
 %4
slot(last, F, Var:_, [/*index*/] ind _ stype
                Slots^Quant^Sem^(Var:Type)^Fs^Feats,
                         Vars, [Var|Vars2], Preds, [], Rest, _, I, I1) :- !,
        sub_slots(last, [Quant,Sem|Feats], F, Slots, Var:Type, Fs,
                                  Vars, Vars2, Preds, _, Rest, _, I, I1).
 %5
slot(more, F, Var:_, [/*index*/] ind _ stype
                Slots^Quant^Sem^(Var:Type)^Fs^Feats, Vars,
                         [Var|Vars2], Preds, Preds1, Rest, Rest1, I, I1) :- !,
        sub_slots(more, [Quant,Sem|Feats], F, Slots, Var:Type, Fs,
                         Vars, Vars2, Preds, Preds1, Rest, Rest1, I, I1).
 %6     last & indexed sub-function.
slot(last, F, Var:_, Index ind _ stype
                Slots^Quant^Sem^(Var:Type)^Fs^Feats,
```

```
                        Vars, [Var|Vars2], Preds, [], Rest, _, I, I1) :-
        index_means(F, Index, Slots^Quant^Sem^(Var:Type)^Fs^Feats,
                                Slotsa^Quanta^Sema^Vara^Fsa^Featsa, I, I2),
        sub_slots(last, [Quanta,Sema|Featsa], F, Slotsa, Vara, Fsa,
                        Vars, Vars2, Preds, _, Rest, _, I2, I1).
%7      more & indexed sub-function.
slot(more, F, Var:_t1, Index ind _ stype
                Slots^Quant^Sem^(Var:Type)^Fs^Feats, Vars,
                        [Var|Vars1], Preds, Preds1, Rest, Rest1, I, I1) :-
        index_means(F, Index, Slots^Quant^Sem^(Var:Type)^Fs^Feats,
                                Slotsa^Quanta^Sema^Vara^Fsa^Featsa, I, I2),
        join_rest(Sema, Rest, Rest2),
        sub_slots(more, [Quanta,[]|Featsa], F, Slotsa, Vara, Fsa,
                        Vars, Vars1, Preds, Preds1, Rest2, Rest1, I2, I1).


        % sub_slots(+last_more, Function_feats, F, Slots, Var:Type, Fs,
%1                      Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1).
sub_slots(_, [[],[],[],[],[],[],[]], _, [f], _, _, Vars, Vars,
                                        Preds, Preds, Rest, Rest, I, I).
%2
sub_slots(M1, [Quant,[],[],Num,Int,Adj,Aggs], F, [f], Var:T, Fs,
                        Vars, Vars, Preds, Preds1,  Rest, Rest1, I, I) :- !,
        trans(M1, F, Fs, Quant, Var:T, [], Num, Int, Adj, Aggs, Vars,
                                                Preds, Preds1, Rest, Rest1).
%3
sub_slots(M1, [Quant,[],Pro,Num,Int,Adj,Aggs], F, [f], Var:T, Fs,
                        Vars, Vars, Preds, Preds1, Rest, Rest1, I, I) :- !,
        proportional(Pro, Var:T, Preds, Preds2, Rest, Rest2),
        trans(M1, F, Fs, Quant, Var:T, [], Num, Int, Adj, Aggs, Vars,
                                                Preds2, Preds1, Rest2, Rest1).
%4
sub_slots(M1, [Quant,Sem,[],Num,Int,Adj,Aggs], F, [f], Var:T, Fs,
                        Vars, Vars, Preds, Preds1,  Rest, Rest1, I, I) :- !,
        trans(M1, F, Fs, Quant, Var:T, Sem, Num, Int, Adj, Aggs, Vars,
                                                Preds, Preds1, Rest, Rest1).
%5
sub_slots(M1, [Quant,[],[],Num,Int,Adj,Aggs], F, [f|R], Var:T, Fs,
                        Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :- !,
        trans(more, F, Fs, Quant, Var:T, [], Num, Int, Adj, Aggs, Vars,
                                                Preds, Preds3, Rest, Rest3),
        slots(M1, F, R, Var:T, Fs,
                        Vars, Vars1, Preds3, Preds1, Rest3, Rest1, I, I1).
%6
sub_slots(M1, [Quant,[],Pro,Num,Int,Adj,Aggs], F, [f|R], Var:T, Fs,
                        Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :- !,
        proportional(Pro, Var:T, Preds, Preds2, Rest, Rest2),
        trans(more, F, Fs, Quant, Var:T, [], Num, Int, Adj, Aggs, Vars,
                                                Preds2, Preds3, Rest2, Rest3),
        slots(M1, F, R, Var:T, Fs,
                        Vars, Vars1, Preds3, Preds1, Rest3, Rest1, I, I1).
%7
sub_slots(M1, [Quant,Sem,[],Num,Int,Adj,Aggs], F, [f|R], Var:T, Fs,
                        Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :- !,
        trans(more, F, Fs, Quant, Var:T, Sem, Num, Int, Adj, Aggs, Vars,
                                                Preds, Preds2, Rest, Rest2),
        slots(M1, F, R, Var:T, Fs,
                        Vars, Vars1, Preds2, Preds1, Rest2, Rest1, I, I1).
%8
sub_slots(M1, _, F, [head = V:T1,mod = V1:T2], Var:T, Fs,
                        Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :- !,
        slots(M1, F, [head = V:T1,mod = V1:T2], Var:T, Fs,
                        Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1).
%9
sub_slots(M1, Feats, F, [pied = V:T1|R], Var:T, Fs,
                        Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :- !,
        slots(more, F, [pied = V:T1|R], Var:T, Fs,
```

```
                        Vars, Vars2, Preds, Preds2, Rest, Rest2, I, I2),
        sub_slots(M1, Feats, F, [f], Var:T, Fs,
                        Vars2, Vars1, Preds2, Preds1, Rest2, Rest1, I2, I1).
 %10
sub_slots(last, [Quant,[],[],Num,Int,[],[]], F, [Slot,f], Var:T, Fs,
                        Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :- !,
        slots(last, F, [Slot], Var:T, Fs, Vars, Vars1,
                                Preds, Preds2, Rep, Preds2, I, I1),
        trans(last, F, Fs, Quant, Var:T, [], Num, Int, [], [],
                                        Vars1, Rep, Preds1, Rest, Rest1).
 %11
sub_slots(last, [Quant,Sem,[],Num,Int,Adj,Aggs], F, [Slot,f], Var:T,
                        Fs, Vars, Vars1, Preds, Preds1, Rest, _, I, I1) :- !,
        slots(more, F, [Slot], Var:T, Fs, Vars, Vars1,
                                Preds, Preds2, Rest, Rest2, I, I1),
        trans(last, F, Fs, Quant, Var:T, Sem, Num, Int, Adj, Aggs, Vars1,
                                        Preds2, Preds1, Rest2, Preds1).
 %12
sub_slots(M1, Feats, F, [Slot|R], Var:T, Fs, Vars, Vars1,
                                Preds, Preds1, Rest, Rest1, I, I1) :- !,
        slots(more, F, [Slot], Var:T, Fs, Vars, Vars2,
                                Preds, Preds2, Rest, Rest2, I, I2),
        sub_slots(M1, Feats, F, R, Var:T, Fs, Vars2, Vars1,
                                        Preds2, Preds1, Rest2, Rest1, I2, I1).
 %13
sub_slots(M1, [Quant,Sem,[],Num,Int,Adj,Aggs], F, [], Var:T, Fs,
                        Vars, Vars, Preds, Preds1, Rest, Rest1, I, I) :- !,
        trans(M1, F, Fs, Quant, Var:T, Sem, Num, Int, Adj, Aggs, Vars,
                                                Preds, Preds1, Rest, Rest1).
 %14
sub_slots(M1, [Quant,[],[],Num,Int,Adj,Aggs], F, [], Var:T, Fs,
                        Vars, Vars, Preds, Preds1, Rest, Rest1, I, I) :- !,
        trans(M1, F, Fs, Quant, Var:T, [], Num, Int, Adj, Aggs, Vars,
                                                Preds, Preds1, Rest, Rest1).
 %15
sub_slots(M1, [Quant,[],Pro,Num,Int,Adj,Aggs], F, [], Var:T, Fs,
                        Vars, Vars, Preds, Preds1, Rest, Rest1, I, I) :- !,
        proportional(Pro, Var:T, Preds, Preds2, Rest, Rest2),
        trans(M1, F, Fs, Quant, Var:T, [], Num, Int, Adj, Aggs, Vars,
                                        Preds2, Preds1, Rest2, Rest1).


        % trans(+last_more, +F, +Fs, 3*+Quant, +^Var:+Type,
        %               5*+Sem, 6*+Num, 7*+Int, 8*+Adj, 9*+Aggs, +Vars,
        %                       +Preds, -Preds1, +Rest, -Rest1, +I, -I1).

 %1     last, no quantifier, no aggregates & no integer.
trans(last, _, _, [/*quant*/], Var:_, Sem, _, [/*int*/], Adjs,
                                [/*aggs*/], _, Preds, [], Rest, _) :- !,
        join_adjs(Var, Sem, Adjs, Sem1),
        join(Preds, Sem1, Rest).

 %2     more, no quantifier, no aggregates & no integer.
trans(more, _f, _fs, [/*quant*/], Var:_, Sem, _, [/*int*/],
                        Adjs, [/*aggs*/], _, Preds, Preds1, Rest, Rest) :- !,
        join_adjs(Var, Sem, Adjs, Sem1),
        join(Preds, Sem1, Preds1).

 %3     more, quantifier, no aggregates & no integer.
trans(more, F, Fs, Quant, Var:_,  Sem, Num, [/*int*/], Adjs,
                        [/*aggs*/], _, Preds, Preds, Rest, Rest1) :- !,
        select_quant(F, Fs, Num, Quant, Var, Rest, Rest2),
        join_adjs(Var, Sem, Adjs, Sem1),
        join_rest(Sem1, Rest2, Rest1).

 %4     last, quantifier, no aggregates & no integer.
trans(last, F, Fs, Quant, Var:_, Sem, Num, [/*int*/], Adjs,
```

```
                                [/*aggs*/], _, Preds, _, Rest, _) :- !,
        select_quant(F, Fs, Num, Quant, Var, Rest, Rest2),
        join_adjs(Var, Sem, Adjs, Sem1),
        join(Sem1, Preds, Rest2).

 %5     last, quantifier, aggregates & no integer.
trans(last, F, Fs, Quant, Var:Type, Sem, _, [/*int*/],
                                Adjs, Aggs, Vars, Preds, [], Rest, _) :- !,
        join_adjs(Var, Preds, Adjs, Preds1),
        aggs(last, F, Fs, Aggs, Quant, Var:Type,
                                Sem, Vars, Preds1, _, Rest, _).

 %6     more, quantifier, aggregates & no integer.
trans(more, F, Fs, Quant, Var:Type, Sem, _, [/*int*/],
                        Adjs, Aggs, Vars, Preds, Preds1, Rest, Rest1) :- !,
        join_adjs(Var, Preds, Adjs, Preds2),
        aggs(more, F, Fs, Aggs, Quant, Var:Type,
                                Sem, Vars, Preds2, Preds1, Rest, Rest1).

 %7     more, no quantifier, no aggregates &  integer.
trans(more, _f, _fs, [/*quant*/], Var:T, Sem, _, Int,
                        Adjs, [/*aggs*/], _, Preds, Preds1, Rest, Rest1) :- !,
        join_adjs(Var, Sem, Adjs, Sem1),
        integer(more, Var:T, Int, Sem1, Preds, Preds1, Rest, Rest1).
 %8
trans(last, _f, _fs, [/*quant*/], Var:T, Sem, _, Int,
                                Adjs, [/*aggs*/], _, Preds, [], Rest, _) :-
        join_adjs(Var, Sem, Adjs, Sem1),
        integer(last, Var:T, Int, Sem1, Preds, _, Rest, _).

 %1
integer(more, N--Unit:_, N--Unit, Sem, Preds, Preds1, Rest, Rest) :-
        join(Sem, Preds, Preds1).
 %2
integer(more, Var:_, gt N, Sem, Preds, Preds2,
                        numberof(Var, Rest1, N1) & N1 > N, Rest1) :- !,
        join(Sem, Preds, Preds2).
 %3
integer(more, Var:_, lt N, Sem, Preds, Preds2,
                        numberof(Var, Rest1, N1) & N1 < N, Rest1) :- !,
        join(Sem, Preds, Preds2).
 %4
integer(last, Var:_, gt N, Sem, Preds, _,
                                numberof(Var, Preds2, N1) & N1 > N, _) :- !,
        join(Sem, Preds, Preds2).
 %5
integer(last, Var:_, lt N, Sem, Preds, _,
                                numberof(Var, Preds2, N1) & N1 < N, _) :- !,
        join(Sem, Preds, Preds2).
 %6
integer(more, Var:_, N, Sem, Preds, _,
                                numberof(Var, Preds2 & Rest, N), Rest) :- !,
        join(Sem, Preds, Preds2).
 %7
integer(last, Var:_, N, Sem, Preds, _, numberof(Var, Preds2, N), _) :- !,
        join(Sem, Preds, Preds2).


 %1     select_quant(+F, +Fs, +Num, +Quant, +Var, +^Rest, -^Rest2),
select_quant(of-obj, _fs, _, the, Var, of_the(Var, Rest1), Rest1) :- !.
 %2
select_quant(_, _, pl, the, Var, the_pl(Var, Rest1), Rest1) :- !.
 %3
select_quant(_, _, _, the, Var, the_sg(Var, Rest1), Rest1) :- !.
 %4
select_quant(_, _fs, _, each, Var, each(Var, Rest1), Rest1) :- !.
 %5
```

```
select_quant(_, _fs, _, no, Var, no(Var, Rest1), Rest1) :- !.
 %6
select_quant(_, _fs, _, any, Var, any(Var, Rest1), Rest1) :- !.
 %7
select_quant(_, _fs, _, some, Var, some(Var, Rest1), Rest1) :- !.
 %8
select_quant(_, _fs, sg, a, Var, a(Var, Rest1), Rest1) :- !.
 %9
select_quant(_, _fs, _, numberof, Var, numberof(Var, Rest1), Rest1) :- !.
 %10
select_quant(_, _fs, _, wh, Var, wh(Var, Rest1), Rest1).


        % aggs(+More_last, +Aggs, +Quant, +^R, +Var:Type,
        %               +Sem, +Preds, -Preds1, +^Rest, -^Rest1).

        /* aggregates are first extracted from the f-structure and then
           translated. */
 %1
aggs(last, _f, _fs, Aggs, _quant, Var:Type, Sem, Vars, Preds, _, Rest, _) :-
        aggs_are(Aggs, Type, Agg_preds, Agg_set, Var),
        join(Sem, Preds, Preds2),
        copy_term((Vars,Preds2, Var), (Vars,Preds3, Var1)),
        join(setof(Var1, Preds3, Agg_set), Agg_preds, Rest).
 %2
aggs(more, _f, _fs, Aggs, _quant, Var:Type, Sem, Vars,
                Preds, [/*null*/], setof(Var1, Preds2, Agg_set)
                                              & Agg_preds & Rest1, Rest1) :-
        join(Sem, Preds, Preds1),
        copy_term((Vars,Preds1,Var), (Vars,Preds2,Var1)),
        aggs_are(Aggs, Type, Agg_preds, Agg_set, Var).


 %1     the last/only aggregate.
aggs_are([[] ind _ stype [/*slots*/]^[/*quant*/]^Sem^(_var:_)^_],
                Type, aggreg(Sem, Type, Set, Ans_var), Set, Ans_var).

 %2     several aggregates.
aggs_are([[] ind _ stype [/*slots*/]^[/*quant*/]^Sem^(_var:_)^_|Aggs_rest],
                Type, aggreg(Sem, Type, Set, Ans_var) & Aggs, Set, Ans_var) :-
        aggs_are(Aggs_rest, Type, Aggs, Set, Ans_var).


        /* adjuncts are first attached to a function in the enclosing
           f-structure and they translated as normal functions. */
 %1
adjuncts(_ml, [], _var, Vars, Vars, Preds, Preds, Rest, Rest, I, I) :- !.
 %2
adjuncts(Ml, [[] ind _st stype Slots^Quant^Sem^(Var:T)^Fs^
                [_Pro,Num,Int,Adj,Aggs]], Var:_, Vars,
                        [Var|Vars2], Preds, Preds1, Rest, Rest1, I, I1) :- !,
        join(Preds, Sem, Preds2),
        slots(more, adjunc, Slots, Var:T, Fs, Vars, Vars2,
                                Preds2, Preds3, Rest, Rest2, I, I1),
        trans(Ml, adjunc, Fs, Quant, Var:T, [/*sem*/],
                Num, Int, Adj, Aggs, Vars2, Preds3, Preds1, Rest2, Rest1).
 %3
adjuncts(Ml, [[] ind _st stype Slots^Quant^Sem^(Var:T)^Fs^
                [_Pro,Num,Int,Adj,Aggs]|Rest_adj], Var:_, Vars,
                        [Var|Vars3], Preds, Preds1, Rest, Rest1, I, I1) :-
        join(Preds, Sem, Preds2),
        slots(more, adjunc, Slots, Var:T, Fs, Vars, Vars2,
                                Preds2, Preds3, Rest, Rest2, I, I2),
        trans(more, adjunc, Fs, Quant, Var:T, [/*sem*/], Num,
                        Int, Adj, Aggs, Vars2, Preds3, Preds4, Rest2, Rest3),
        adjuncts(Ml, Rest_adj, Var:T, Vars2, Vars3,
                                Preds4, Preds1, Rest3, Rest1, I2, I1).
```

```
%1
conjs(_, [], _, Vars, Vars, Preds, Preds, Rest, Rest, I, I) :- !.
%2
conjs(M1, [Val|Vals], Var, Vars, Vars1, Preds, Preds1, Rest, Rest1, I, I1) :-
        slot(more, con, Var, Val, Vars, Vars2,
                                  Preds, Preds2, Rest, Rest2, I, I2),
        conjs(M1, Vals, Var, Vars2, Vars1,
                                  Preds2, Preds1, Rest2, Rest1, I2, I1).

%1
join([/*null*/], [/*null*/], [/*null*/]) :- !.
%2
join([/*null*/], Rep, Rep) :- !.
%3
join(Rep, [/*null*/], Rep) :- !.
%4
join(\+ { Rep }, [], \+ { Rep }) :- !.
%5
join([], \+ { Rep }, \+ { Rep }) :- !.
%6
join(\+ { Rep }, \+ { Rep1 }, \+ { Rep } & \+ { Rep1 }) :- !.
%7
join(\+ { Rep }, Rep1, \+ { Rep } & Rep1) :- !.
%8
join(Rep, \+ { Rep1 }, Rep & \+ { Rep1 }) :- !.
%9
join(\+ Rep, Rep1, \+ { Rep2 }) :- join(Rep, Rep1, Rep2).
%10
join(Rep, \+ Rep1, \+ { Rep2 }) :- join(Rep, Rep1, Rep2).
%11
join(Rep & Reps, Rep1 & Reps1, Rep & Reps2) :- !,
        join(Reps, Rep1 & Reps1, Reps2).
%12
join(Rep & Reps, Rep1, Rep & Reps2) :- !, join(Reps, Rep1, Reps2).
%13
join(Rep, Rep1 & Reps, Rep & Rep1 & Reps) :- !.
%14
join(Rep, Rep1, Rep & Rep1).


%1
join_rest([/*null*/], Rest, Rest) :- !.
%2
join_rest(Sem & Sem1, Sem & Sem2 & Rest, Rest) :- !, join(Sem1, [], Sem2).
%3
join_rest(Sem, Sem & Rest, Rest).


%1
join_adjs(_, Preds, [], Preds) :- !.
%2
join_adjs(Var, Preds, [[] ind _ stype _^_^Pred^(Var:_)^_|Adjs1], Preds1) :-
        join(Preds, Pred, Preds2),
        join_adjs(Var, Preds2, Adjs1, Preds1).

%1      proportional(+Sema, +Vara:Type, +Preds, -Preds1, ^Rest, -Rest).
proportional(percentage(Quant, Of_obj), Memb:_Type, Of_obj_preds,
                         [/*null*/],
                         setof(Quant, Of_obj_preds, Set) &
                         numberof(Memb, pick(Memb, Set) & Rest, Num) &
                         card(Num_all, Set) &
                         ratio(Num, Num_all, Of_obj), Rest).

        /* ************************************************************ */
```

```
        /* ************************************************************ */
        /*      FILE    : type_system.pl                                */
        /*      PURPOSE : defines compatibility of domain types in the  */
        /*                domain hierarchy.                             */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(type_system, [
                compat/3,
                type/1
                ]).

        /* I M P O R T S */

:- use_module(geo_types, [
                sub_types/2
                ]).

:- use_module(fast_basics, [
                fmember/2
                ]).

        /* P R E D I C A T E S */

type(T) :- compat(T, _, _).

        /* determine if two types are compatible and return the new type for a
           variable (the most specialised type). */

compat(Type, Type1, New_type) :- compatible(Type, Type1, New_type), !.


compatible(Type, Type, Type) :- !.

compatible(Sub_type, Type1, Type1) :-
        sub_types(Type1, Sub_types), !,
          ( fmember(Sub_type, Sub_types), !
        |
            subs(Sub_type, Sub_types, Sub_type) ).

subs(_, [], _) :- !, fail.

subs(Sub_type, [First|Rest], Type2) :-
          ( compatible(Sub_type, First, Type2), !
        |
           subs(Sub_type, Rest, Type2) ).

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : unify.pl                                      */
        /*      PURPOSE : unification of information structures.       */
        /* ************************************************************ */

        /* E X P O R T S */

:- module(unify, [
                merge/6,
                new_fs/6,
                no_constraints_ptr/2
                ]).

        /* I M P O R T S */

:- use_module(fast_basics, [
```

```
                fappend/3,
                fmember/2,
                fdelete/3
                ]).

:- use_module(type_system, [
                compat/3
                ]).

:- use_module(fs_basics, [
                member_fs/2
                ]).

        /* P R E D I C A T E S */

        % merge(+Info, +Info1, +Arrow, +Ptrs, -Info2, -Ptrs1).

 %1       info's both full first with no quantifier or semantic
        % component but with different fs's.
merge(I ind full stype Slots^[]^[]^Var^Fs,
        I ind full stype Slots^Quant^Sem^Var^Fs1, Arrow, Ptrs,
                        I ind full stype Slots^Quant^Sem^Var^Fs2, Ptrs1) :- !,
        new_fs(Fs, Fs1, Ptrs, Arrow, Fs2, Ptrs1).

 %2       info's both full second with no quantifier or semantic
        % component but with different fs's.
merge(I ind full stype Slots^Quant^Sem^Var^Fs,
        I ind full stype Slots^[]^[]^Var^Fs1, Arrow, Ptrs,
                        I ind full stype Slots^Quant^Sem^Var^Fs2, Ptrs1) :- !,
        new_fs(Fs, Fs1, Ptrs, Arrow, Fs2, Ptrs1).
 %3
merge([] ind part stype []^[]^[]^Var^[],
                I ind Slots^Quant^Sem^Var^Fs, _, Ptrs,
                                I ind Slots^Quant^Sem^Var^Fs, Ptrs) :- !.
 %4
merge(I ind Slots^Quant^Sem^Var^Fs,
                [] ind part stype []^[]^[]^Var^[],
                        _, Ptrs, I ind Slots^Quant^Sem^Var^Fs, Ptrs) :- !.
 %5
merge(I ind Slots^Quant^Sem^Var^Fs,
                [] ind part stype []^[]^[]^Var^Fs1, Arrow, Ptrs,
                                I ind Slots^Quant^Sem^Var^Fs2, Ptrs1) :- !,
        new_fs(Fs, Fs1, Ptrs, Arrow, Fs2, Ptrs1).
 %6
merge([] ind part stype []^[]^[]^Var^Fs,
                I ind Slots^Quant^Sem^Var^Fs1, Arrow, Ptrs,
                                I ind Slots^Quant^Sem^Var^Fs2, Ptrs1) :- !,
        new_fs(Fs, Fs1, Ptrs, Arrow, Fs2, Ptrs1).
 %7
merge(I ind Slots^Quant^Sem^Var^Fs,
                [] ind Slots1^[]^[]^Var^Fs1, Arrow, Ptrs,
                                I ind Slots2^Quant^Sem^Var^Fs2, Ptrs1) :- !,
        new_fs(Fs, Fs1, Ptrs, Arrow, Fs2, Ptrs1), !,
        new_slots(Slots, Slots1, Slots2).
 %8
merge([] ind Slots^[]^[]^Var^Fs,
                I ind Slots1^Quant^Sem^Var^Fs1, Arrow, Ptrs,
                                I ind Slots2^Quant^Sem^Var^Fs2, Ptrs1) :- !,
        new_fs(Fs, Fs1, Ptrs, Arrow, Fs2, Ptrs1), !,
        new_slots(Slots, Slots1, Slots2).
 %9     merge on moved info. structure.
merge(moved Info temp Infos, moved Info temp Info1, _Arrow, Ptrs,
                                        moved Info temp Infos2, Ptrs1) :- !,
        merge(Infos, Info1, up, Ptrs, Infos2, Ptrs1).
 %10
merge(moved Info temp Infos, part stype []^[]^[]^_^[], _, Ptrs,
                                        moved Info temp Infos, Ptrs) :- !.
```

```
 %11
merge([] ind part stype []^[]^[]^_var^[], moved Infol temp Infos,
                                _, Ptrs, moved Infol temp Infos, Ptrs) :- !.
 %12
merge(moved Info temp Infos, ptr N, Arrow, Ptrs, ptr N, Ptrs1) :- !,
        merge_pointer(ptr N, moved Info temp Infos, Arrow, Ptrs, Ptrs1).
 %13
merge(ptr N, moved Info temp Infos, Arrow, Ptrs, ptr N, Ptrs1) :- !,
        merge_pointer(ptr N, moved Info temp Infos, Arrow, Ptrs, Ptrs1).
 %14
merge(moved Info temp Infos, Infol, _Arrow, Ptrs,
                                moved Info temp Infos2, Ptrs1) :- !,
        merge(Infos, Infol, up, Ptrs, Infos2, Ptrs1).
 %15
merge(Info, moved Infol temp Infos2, _Arrow, Ptrs,
                                moved Infol temp Infos3, Ptrs1) :- !,
        merge(Infos2, Info, up, Ptrs, Infos3, Ptrs1).
 %16
merge(Info, ptr N, Arrow, Ptrs, ptr N, Ptrs1) :- !,
        merge_pointer(ptr N, Info, Arrow, Ptrs, Ptrs1).
 %17
merge(ptr N, Infol, Arrow, Ptrs, ptr N, Ptrs1) :- !,
        merge_pointer(ptr N, Infol, Arrow, Ptrs, Ptrs1).

 %18    complete merge on infos.
merge(I ind Slots^Quant^Sem^Var^Fs,
                I1 ind Slots1^Quant1^Sem1^Var1^Fs1, Arrow,
                Ptrs, I2 ind Slots2^Quant2^Sem2^Var2^Fs2, Ptrs1) :-
        new_slots(Slots, Slots1, Slots2), !,
        new_quant_var(Var, Var1, Var2), !,
        new_quant(Quant, Quant1, Quant2), !,
        new_sem(Sem, Sem1, Sem2), !,
        new_index(I, I1, I2), !,
        new_fs(Fs, Fs1, Ptrs, Arrow, Fs2, Ptrs1), !.

 %1
new_quant_var(Var, Var, Var) :- !.
 %2
new_quant_var(Var:T, Var:T1, Var:T2) :- !,
        ( compat(T, T1, T2), ! | compat(T1, T, T2) ).


new_index([], [], []) :- !.      new_index([], I, I) :- !.

new_index(I, [], I).             new_index([], [], _).


        /* new_slots(+Slots, +Slots1, -Slots2). */
 %1
new_slots(S, S, S) :- !.
 %2
new_slots(full stype Slots, cc stype Slots, cc stype Slots) :- !.
 %3
new_slots(full stype Slots, part stype Slots1, full stype Slots2) :- !,
        slots_of(Slots, Slots1, Slots2).
 %4
new_slots(part stype Slots, full stype Slots1, full stype Slots2) :- !,
        slots_of(Slots1, Slots, Slots2).
 %5
new_slots(part stype Slots, part stype Slots1, part stype Slots2) :-
        join_slots(Slots, Slots1, Slots2).
 %6
new_slots(part stype Slots, part stype Slots1, full stype Slots2) :-
          ( var(Slots2), !, fail | join_slots(Slots, Slots1, Slots2) ).

 %1     % check typing and unify vars in full and partial slot templates.
slots_of(Slots, [], Slots) :- !.
```

```
 %2
slots_of([F = Var:Type|Rest], [F = Var:Type1|Rest1],
                                        [F = Var:Type2|Rest2]) :- !,
        compat(Type1, Type, Type2), slots_of(Rest, Rest1, Rest2).
 %3
slots_of([F - F1 = Var:Type|Rest], [F = Var:Type1|Rest1],
                                        [F - F1 = Var:Type2|Rest2]) :- !,
        compat(Type1, Type, Type2), slots_of(Rest, Rest1, Rest2).
 %4
slots_of([F = Var:Type|Rest], [F - F1 = Var:Type1|Rest1],
                                        [F - F1 = Var:Type2|Rest2]) :- !,
        compat(Type1, Type, Type2), slots_of(Rest, Rest1, Rest2).
 %5
slots_of([First|Rest], Slots1, [First|Slots2]) :-
        slots_of(Rest, Slots1, Slots2).

        % join to partial slot templates.
 %6
join_slots([], Slots, Slots) :- !.
 %7
join_slots(Slots, [], Slots) :- !.
 %8
join_slots([F - F1 = Var:Type|Rest], [F = Var:Type1|Rest1],
                                        [F - F1 = Var:Type2|Rest2]) :- !,
         ( compat(Type, Type1, Type2), ! | compat(Type1, Type, Type2) ),
        join_slots(Rest, Rest1, Rest2).
 %9
join_slots([F = Var:Type|Rest], [F - F1 = Var:Type1|Rest1],
                                        [F - F1 = Var:Type2|Rest2]) :- !,
        ( compat(Type, Type1, Type2), ! | compat(Type1, Type, Type2) ),
        join_slots(Rest, Rest1, Rest2).
 %10
join_slots([F = Var:Type|Rest], [F = Var:Type1|Rest1],
                                                [F = Var:Type2|Rest2]) :- !,
        ( compat(Type, Type1, Type2), ! | compat(Type1, Type, Type2) ),
        join_slots(Rest, Rest1, Rest2).
 %11
join_slots([F = _:_|_], [F = _:_|_], _) :- !, fail.     % eg pred 'U' pred.
 %12
join_slots([F = Var:Type|Rest], [F1 = Var1:Type1|Rest1],
                                                [F2 = Var2:Type2|Rest2]) :- !,
         ( F @> F1 -> (F2,Var2,Type2) = (F1,Var1,Type1), !,
                    join_slots([F = Var:Type|Rest], Rest1, Rest2), !
        |
           (F2,Var2,Type2) = (F,Var,Type), !,
           join_slots(Rest, [F1 = Var1:Type1|Rest1], Rest2)
         ).

        % only one f-structure can have a quantifier (uniqueness).
new_quant([], Quant, Quant) :- !.       new_quant(Quant, [], Quant) :- !.

new_quant([], [], []) :- !.             new_quant([], [], _) :- !.

        % only one can have a predicate.
new_sem([], Sem, Sem) :- !.             new_sem(Sem, [], Sem) :- !.

new_sem([], [], []) :- !.               new_sem([], [], _) :- !.


        /* new_fs(+Fs, +Fs1, +Ptrs, +Arrow, -Fs2, -Ptrs1).
           produce, by unification, a new f-structure list 'Fs2' from
           the two f-structure lists 'Fs' & 'Fs1' */
 %1
new_fs([], [], Ptrs, _, [], Ptrs) :- !.
 %2
new_fs(Fs, [], Ptrs, up, Fs, Ptrs) :- !.
 %3
```

```
new_fs([], Fs, Ptrs, up, Fs, Ptrs) :- !.
 %4
new_fs(Fs, Fs, Ptrs, up, Fs, Ptrs) :- !.
 %5
new_fs([], Fs, Ptrs, down, Fs, Ptrs) :- !, no_constraints_ptr(Fs, Ptrs).
 %6
new_fs(Fs, [], Ptrs, down, Fs, Ptrs) :- !, no_constraints_ptr(Fs, Ptrs).
 %7
new_fs([F = Val|Rest], [F = Val|Rest1], Ptrs,
                                        Arrow, [F = Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).

 %8     both sets of f-structures.
new_fs([F = set Set|Rest], [F = set Set1|Rest1], Ptrs,
                                  Arrow, [F = set Set2|Rest2], Ptrs1) :- !,
        fappend(Set, Set1, Set2),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %9
new_fs([F = exists neg_c Vals|Rest], [F = exists neg_c Vals1|Rest1],
                Ptrs, Arrow, [F = exists neg_c Vals2|Rest2], Ptrs1) :- !,
        fappend(Vals, Vals1, Vals2),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %10
new_fs([F = exists neg_c Vals|Rest], [F = exists val_c Val|Rest1],
                Ptrs, Arrow, [F = exists val_c Val|Rest2], Ptrs1) :- !,
        \+ fmember(Val, Vals),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %11
new_fs([F = exists neg_c Vals|Rest], [F = neg_c Vals1|Rest1],
                Ptrs, Arrow, [F = exists neg_c Vals2|Rest2], Ptrs1) :- !,
        fappend(Vals, Vals1, Vals2),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %12
new_fs([F = exists neg_c Vals|Rest], [F = val_c Val|Rest1],
                Ptrs, Arrow, [F = exists val_c Val|Rest2], Ptrs1) :- !,
        \+ fmember(Val, Vals),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %13
new_fs([F = exists neg_c Vals|Rest], [F = exists|Rest1],
                Ptrs, Arrow, [F = exists neg_c Vals|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %14
new_fs([F = exists neg_c Vals|Rest], [F = atom Val|Rest1],
                          Ptrs, Arrow, [F = atom Val|Rest2], Ptrs1) :- !,
        \+ fmember(Val, Vals),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %15
new_fs([F = exists val_c Val|Rest], [F = exists neg_c Vals|Rest1],
                Ptrs, Arrow, [F = exists val_c Val|Rest2], Ptrs1) :- !,
        \+ fmember(Val, Vals),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %16
new_fs([F = exists val_c Val|Rest], [F =  val_c Val|Rest1],
                Ptrs, Arrow, [F = exists val_c Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %17
new_fs([F = exists val_c Val|Rest], [F = neg_c Vals|Rest1],
                Ptrs, Arrow, [F = exists val_c Val|Rest2], Ptrs1) :- !,
        \+ fmember(Val, Vals),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %18
new_fs([F = exists val_c Val|Rest], [F = exists|Rest1],
                Ptrs, Arrow, [F = exists val_c Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %19
new_fs([F = exists val_c Val|Rest], [F = atom Val|Rest1],
                          Ptrs, Arrow, [F = atom Val|Rest2], Ptrs1) :- !,
```

```
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).

 %20    first value constraint, second value.
new_fs([F = val_c Val|Rest], [F = atom Val|Rest1], Ptrs,
                                Arrow, [F = atom Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).

 %21    second value constraint, first value.
new_fs([F = atom Val|Rest], [F = val_c Val|Rest1], Ptrs,
                                Arrow, [F = atom Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).

 %22    two neg constraints.
new_fs([F = neg_c List|Rest], [F = neg_c List1|Rest1], Ptrs,
                                Arrow, [F = neg_c List2|Rest2], Ptrs1) :- !,
        fappend(List, List1, List2),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).

 %23    first neg constraint, second value.
new_fs([F = neg_c List|Rest], [F = atom Val|Rest1], Ptrs,
                                Arrow, [F = atom Val|Rest2], Ptrs1) :- !,
        \+ fmember(Val, List), !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).

 %24    second neg constraint, first value.
new_fs([F = atom Val|Rest], [F = neg_c List|Rest1], Ptrs,
                                        Arrow, [F = Val|Rest2], Ptrs1) :- !,
        \+ fmember(Val, List), !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %25
new_fs([Feat = exists|Rest], [Feat = atom Val|Rest1], Ptrs,
                                Arrow, [Feat = atom Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %26
new_fs([Feat = atom Val|Rest], [Feat = exists|Rest1], Ptrs,
                                Arrow, [Feat = atom Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %27
new_fs([Feat = exists|Rest], [Feat = neg_c Vals|Rest1], Ptrs,
                        up, [Feat = exists neg_c Vals|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, up, Rest2, Ptrs1).
 %28
new_fs([Feat = neg_c Vals|Rest], [Feat = exists|Rest1], Ptrs,
                        up, [Feat = exists neg_c Vals|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, up, Rest2, Ptrs1).
 %29
new_fs([Feat = neg_c Vals|Rest], [Feat = neg_c Vals1|Rest1],
                        Ptrs, up, [Feat = neg_c Vals2|Rest2], Ptrs1) :- !,
        fappend(Vals, Vals1, Vals2),
        new_fs(Rest, Rest1, Ptrs, up, Rest2, Ptrs1).
 %30
new_fs([Feat = exists|Rest], [Feat = val_c Val|Rest1],
                Ptrs, up, [Feat = exists val_c Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, up, Rest2, Ptrs1).
 %31
new_fs([Feat = val_c Val|Rest], [Feat = exists|Rest1],
                Ptrs, up, [Feat = exists val_c Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, up, Rest2, Ptrs1).
 %32
new_fs([Feat = val_c Val|Rest], [Feat = Val|Rest1],
                        Ptrs, Arrow, [Feat = Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %33
new_fs([Feat = atom Val|Rest], [Feat = exists val_c Val|Rest1],
                        Ptrs, Arrow, [Feat = atom Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %34
```

```
new_fs([Feat = exists val_c Val|Rest], [Feat = atom Val|Rest1],
                        Ptrs, Arrow, [Feat = atom Val|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %35
new_fs([Feat = atom Val|Rest], [Feat = exists neg_c Vals|Rest1],
                        Ptrs, Arrow, [Feat = atom Val|Rest2], Ptrs1) :- !,
        \+ fmember(Val, Vals),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %36
new_fs([Feat = exists neg_c Vals|Rest], [Feat = atom Val|Rest1],
                        Ptrs, Arrow, [Feat = atom Val|Rest2], Ptrs1) :- !,
        \+ fmember(Val, Vals),
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs1).
 %37    recursive unify of functions
new_fs([F = fs Val|Rest], [F = fs Val1|Rest1], Ptrs,
                                Arrow, [F = fs Val2|Rest2], Ptrs1) :- !,
        new_fs(Rest, Rest1, Ptrs, Arrow, Rest2, Ptrs2), !,
        merge(Val, Val1, Arrow, Ptrs2, Val2, Ptrs1), !.
 %38
new_fs([F = _|_], [F = _|_], _, _, _, _) :- !, fail.
 %39
new_fs([F = Val|Rest], [F1 = Val1|Rest1], Ptrs, Arrow,
                                        [F2 = Val2|Rest2], Ptrs1) :-
         ( F @< F1  ->  (F2,Val2) = (F,Val), !,
                new_fs(Rest, [F1 = Val1|Rest1], Ptrs, Arrow, Rest2, Ptrs1), !
        |
           (F2,Val2) = (F1,Val1), !,
           new_fs([F = Val|Rest], Rest1, Ptrs, Arrow, Rest2, Ptrs1)
         ).

 %1
no_constraints_ptr(Fs, _) :- no_constraints(Fs), !.

        % everything allowable in a 'complete' f-structure. (not 'exists').
no_constraints([]) :- !.

no_constraints([_ = fs _|R]) :- !, no_constraints(R).

no_constraints([_ = none|R]) :- !, no_constraints(R).

no_constraints([_ = atom _|R]) :- !, no_constraints(R).

no_constraints([_ = val_c _|R]) :- !, no_constraints(R).

no_constraints([_ = neg_c _|R]) :- !, no_constraints(R).

no_constraints([_ = set _|R]) :- !, no_constraints(R).

 %1
no_ptr_constraints([]) :- !.
 %2
no_ptr_constraints([_ = fs _ ind _^_^_^_^Fs|R]) :-
        no_constraints(Fs), !, no_ptr_constraints(R).

 %1
merge_pointer(ptr N, Info, Arrow, Ptrs, [N = fs Val1|Ptrs1]) :-
        fdelete(N = fs Val, Ptrs, Ptrs2),
        merge(Val, Info, Arrow, Ptrs2, Val1, Ptrs1).

        /* ************************************************************ */


        /* ************************************************************ */
        /*      FILE    : wf_fs.pl                                      */
        /*      PURPOSE : checks well-formedness of F-structures.       */
        /* ************************************************************ */
```

```
        /* E X P O R T S */

:- module(wf_fs, [
                well_formed/4,
                contains/2
                ]).

        /* I M P O R T S */

:- use_module(unify, [
                merge/6
                ]).

:- use_module(fast_basics, [
                fdelete/3,
                fmember/2
                ]).

        /* P R E D I C A T E S */

        % check that a complete feature-structure is wf.
 %1
well_formed(ptr N, Val1, Ptrs, [N = fs Val1|Ptrs3]) :-
        fdelete(N = fs Val, Ptrs, Ptrs2),
        well_formed(Val, Val1, Ptrs2, Ptrs3).
 %2
well_formed(I ind cc stype Slots^Quant^Sem^Var^Fs^[],
                I ind cc stype Slots^Quant^Sem^Var^Fs, Ptrs, Ptrs) :- !.
 %3
well_formed(I ind cc stype Slots^Quant^Sem^Var^Fs,
                I ind cc stype Slots^Quant^Sem^Var^Fs, Ptrs, Ptrs) :- !.
 %4
well_formed(_ ind full stype _^_^_^_^[]^[], _, _, _) :- !, fail.
 %5
well_formed(I ind full stype Slots^Quant^Sem^Var^Fs^[],
                I ind cc stype Slots^Quant^Sem^Var^Fs1, Ptrs, Ptrs1) :- !,
        complete(Slots, Fs, Fs1, Ptrs, Ptrs1), Fs1 \== [].
 %6
well_formed(_ ind full stype _^_^_^_^[], _, _, _) :- !, fail.
 %7
well_formed(I ind full stype Slots^Quant^Sem^Var^Fs,
                I ind cc stype Slots^Quant^Sem^Var^Fs1, Ptrs, Ptrs1) :- !,
        complete(Slots, Fs, Fs1, Ptrs, Ptrs1), Fs1 \== [].
 %8
well_formed(moved var Val temp Val1, Val2, Ptrs, Ptrs1) :-
        nonvar(Val), !, merge(Val, Val1, down, Ptrs, Val3, Ptrs2),
        well_formed(Val3, Val2, Ptrs2, Ptrs1).
 %9
well_formed(moved var Val1 temp Val, Val1, Ptrs, Ptrs1) :- !,
        well_formed(Val, Val1, Ptrs, Ptrs1).
 %10
well_formed(moved Val temp Val1, Val2, Ptrs, Ptrs1) :-
        nonvar(Val), !, merge(Val, Val1, down, Ptrs, Val3, Ptrs2),
        well_formed(Val3, Val2, Ptrs2, Ptrs1).
 %11
well_formed(moved Val1 temp Val, Val1, Ptrs, Ptrs1) :- !,
        well_formed(Val, Val1, Ptrs, Ptrs1).

        % a partial f-structure may be well_formed if it contains no
        % functions and no constraints.
 %12
well_formed(_ ind part stype [/*slots*/]^_^_^_^[], _, _, _) :- !, fail.
 %13
well_formed(I ind part stype [/*slots*/]^Quant^Sem^Var^Fs,
                        I ind cc stype []^Quant^Sem^Var^Fs1, Ptrs, Ptrs1) :-
        no_variables(Fs, Fs1, Ptrs, Ptrs1), Fs1 \== [].
```

```
        % remove any remaining value and neg-existential constraints and
        % check no existential constraints remain.
 %1
no_variables([], [], Ptrs, Ptrs) :- !.
 %2
no_variables([Feat = atom A|Rest], [Feat = atom A|Rest1], Ptrs, Ptrs1) :- !,
        no_variables(Rest, Rest1, Ptrs, Ptrs1).
 %3
no_variables([_ = neg_c _|Rest], Rest1, Ptrs, Ptrs1) :- !,
        no_variables(Rest, Rest1, Ptrs, Ptrs1).
 %4
no_variables([_ = val_c _|Rest], Rest1, Ptrs, Ptrs1) :- !,
        no_variables(Rest, Rest1, Ptrs, Ptrs1).
 %5
no_variables([Funct = set Vals|Rest],
                                [Funct = set Vals1|Rest1], Ptrs, Ptrs1) :- !,
        wf_set(Vals, Vals1), no_variables(Rest, Rest1, Ptrs, Ptrs1).
 %6
no_variables([_ = none|Rest], Rest1, Ptrs, Ptrs1) :- !,
        no_variables(Rest, Rest1, Ptrs, Ptrs1).
 %7
no_variables([Funct = fs Val|Rest], [Funct = fs Val1|Rest1], Ptrs, Ptrs1) :-
        no_variables(Rest, Rest1, Ptrs, Ptrs2),
        well_formed(Val, Val1, Ptrs2, Ptrs1).


 %1
complete([], Fs, Fs1, Ptrs, Ptrs1) :- !, no_variables(Fs, Fs1, Ptrs, Ptrs1).
 %2
complete(Functs, [_ = none|Rest], Rest1, Ptrs, Ptrs1) :- !,
        complete(Functs, Rest, Rest1, Ptrs, Ptrs1).
 %3
complete(domain = _, Fs, Fs1, Ptrs, Ptrs1) :- !,
        no_variables(Fs, Fs1, Ptrs, Ptrs1).
 %4
complete([F - _ = _:_|Rest], [F = fs I ind cc stype Info|Rest1],
                        [F = fs I ind cc stype Info|Rest2], Ptrs, Ptrs1) :- !,
        complete(Rest, Rest1, Rest2, Ptrs, Ptrs1).
 %5
complete([F - F1 = V:T|Rest], [F = fs I ind part stype
                        Slots^Quant^Sem^Var^Fs|Rest1],
                                [F = fs Val1|Rest2], Ptrs, Ptrs1) :- !,
        has_funct(F1 = V:T, part stype Slots),
        well_formed(I ind full stype Slots^Quant^Sem^Var^Fs,
                                                Val1, Ptrs, Ptrs2),
        complete(Rest, Rest1, Rest2, Ptrs2, Ptrs1).
 %6
complete([F - F1 = V:T|Rest],
                [F = fs I ind Slots^Quant^Sem^Var^Fs|Rest1],
                                [F = fs Val1|Rest2], Ptrs, Ptrs1) :- !,
        has_funct(F1 = V:T, Slots),
        well_formed(I ind Slots^Quant^Sem^Var^Fs, Val1, Ptrs, Ptrs2),
        complete(Rest, Rest1, Rest2, Ptrs2, Ptrs1).
 %7
complete([F = _:_|Rest], [F = fs ptr N|Rest1],
                                [F = fs Val1|Rest2], Ptrs, Ptrs1) :- !,
        fmember(N = fs Val, Ptrs),
        well_formed(Val, Val1, Ptrs, Ptrs2),
        fdelete(N = fs Val, Ptrs2, Ptrs3),
        complete(Rest, Rest1, Rest2, [N = fs Val1|Ptrs3], Ptrs1).
 %8
complete([F - F1 = V:T|Rest], [F = fs ptr N|Rest1],
                                [F = fs Val1|Rest2], Ptrs, Ptrs1) :- !,
        fmember(N = fs I ind _ stype Slots^Quant^Sem^Var^Fs, Ptrs),
        has_funct(F1 = V:T, full stype Slots),
        well_formed(I ind full stype Slots^Quant^Sem^Var^Fs,
                                                Val1, Ptrs, Ptrs2),
        fdelete(N = fs _, Ptrs2, Ptrs3),
```

```
        complete(Rest, Rest1, Rest2, [N = fs Val1|Ptrs3], Ptrs1).
 %9
complete(Slots, [Funct = set Vals|Rest],
                                [Funct = set Vals1|Rest2], Ptrs, Ptrs1) :- !,
        wf_set(Vals, Vals1), complete(Slots, Rest, Rest2, Ptrs, Ptrs1).
 %10
complete([F = _:_|Rest], [F = fs Val|Rest1],
                                [F = fs Val1|Rest2], Ptrs, Ptrs1) :- !,
        well_formed(Val, Val1, Ptrs, Ptrs2),
        complete(Rest, Rest1, Rest2, Ptrs2, Ptrs1).
 %11
complete(Slots, [F = fs Val|Rest], [F = fs Val1|Rest2], Ptrs, Ptrs1) :- !,
        well_formed(Val, Val1, Ptrs, Ptrs2),
        complete(Slots, Rest, Rest2, Ptrs2, Ptrs1).
 %12
complete(Slots, [Feat = atom A|Rest],
                        [Feat = atom A|Rest1], Ptrs, Ptrs1) :- !,
        complete(Slots, Rest, Rest1, Ptrs, Ptrs1).
 %13
complete(Slots, [_ = neg_c _|Rest], Rest1, Ptrs, Ptrs1) :- !,
        complete(Slots, Rest, Rest1, Ptrs, Ptrs1).
 %14
complete(Slots, [_ = val_c _|Rest], Rest1, Ptrs, Ptrs1) :- !,
        complete(Slots, Rest, Rest1, Ptrs, Ptrs1).


 %1
wf_set([], []) :- !, fail.              % an empty set is not accepted.
 %2
wf_set([First|Rest], [First1|Rest1]):-
        wf_set_memb(First, First1), wf_rest_set(Rest, Rest1).


 %1
wf_rest_set([], []) :- !.
 %2
wf_rest_set([First|Rest], [First1|Rest1]) :-
        wf_set_memb(First, First1), wf_rest_set(Rest, Rest1).


 %1
wf_set_memb(_ ind _ stype _^_^_^[], _) :- !, fail.    % empty f-structure.
 %2
wf_set_memb(I ind _ stype Slots^Quant^Sem^Var^Fs,
                        I ind cc stype Slots^Quant^Sem^Var^Fs1) :- !,
        no_set_variables(Fs, Fs1).


        % set members may contain functions which in the case of
        % 'case' functions (of-obj etc) will have 'part' type slots
        % which are not allowed in non-set members which have had
        % their designator functions removed.

no_set_variables([], []) :- !.

no_set_variables([Feat = atom A|Rest], [Feat = atom A|Rest1]) :- !,
        no_set_variables(Rest, Rest1).

no_set_variables([F = set Vals|Rest], [F = set Vals1|Rest1]) :- !,
        wf_set(Vals, Vals1), no_set_variables(Rest, Rest1).

no_set_variables([_ = none|Rest], Rest1) :- !, no_set_variables(Rest, Rest1).

no_set_variables([_ = val_c|Rest], Rest1) :- !, no_set_variables(Rest, Rest1).

no_set_variables([_ = neg_c|Rest], Rest1) :- !, no_set_variables(Rest, Rest1).

no_set_variables([Funct = fs I ind part stype Slots^Quant^Sem^
                                Var^Fs|Rest], [Funct = fs Val|Rest1]) :-
        no_set_variables(Rest, Rest1),
```

```
        well_formed(I ind full stype Slots^Quant^Sem^Var^Fs, Val, [], _).

no_set_variables([Funct = fs Val|Rest], [Funct = fs Val1|Rest1]) :-
        no_set_variables(Rest, Rest1), well_formed(Val, Val1, [], _).

        % check that an f-s contains a slot for a newly added function.

contains(_type stype [], _f) :- !, fail.

contains(_type stype [F = _:_|_], F) :- !.

contains(Type stype [_|Rest], F) :- contains(Type stype Rest, F).

has_funct(F = V:_, _type stype [F = V:_|_]) :- !.

has_funct(F = V:T, Type stype [F1 = _:_|Rest]) :-
        F @> F1, !, has_funct(F = V:T, Type stype Rest).

        /* ************************************************************** */
```