Information Extraction and Integration with Florid: The MONDIAL Case Study

Wolfgang May
may@informatik.uni-freiburg.de

TECHNICAL REPORT No. 131

Institut für Informatik
Albert-Ludwigs-Universität
Am Flughafen 17
79110 Freiburg, Germany

December 1999

Keywords: Information Extraction, Information Integration, WWW, HTML, Wrapping, Mediating, F-Logic.

Abstract

For accessing and processing this information provided on the Web, there is a need for integration of data from different, heterogeneous sources. Languages for this purpose have to serve for querying the web, extracting information from semistructured data, and restructuring the results. In [LHL+98] we argue that languages supporting deduction and object-orientation are particularly suited in this context; we proposed a formal model for querying structure and contents of Web data. A main advantage of our approach is that it brings together the above-mentioned issues in a unified, formal framework. The approach is implemented in the FLORID system [HLLS98] which is an implementation of the deductive objectorientes database language F-Logic [KLW95].

This report substantiates the above claims by a case-study using FLORID: We show how several information sources on the Web containing political and geographical data are integrated to a geographical database using FLORID. The case study illustrates the trade-off gained from an integrated Web-querying and data manipulation language, supporting a concise and elegant programming style. Using a deductive language, a process of rapid prototyping and refinement of the program – implementing both a wrapper and a mediator – can be easily followed: the program consists of a skeleton of generic wrapping rules [MHLL99], augmented by refining rules and application-specific rules.

Homepage of the MONDIAL Case Study:
`http://www.informatik.uni-freiburg.de/~may/Mondial/`.

Contents

1 Introduction

The Web can be seen as a large, still growing semi-structured data repository. For accessing and processing this information, there is a need for integration of data from different, heterogeneous sources. Since the field is rapidly growing and changing, semantically rich, flexible, and adaptable frameworks are required to cope with the evolving needs. In [LHL+98], we argue that languages supporting deduction and object-orientation are particularly suited in this context: *Object-orientation* provides a flexible and rich data model, and *deductive rules* provide a powerful framework for expressing complex queries in a high-level, declarative programming style. Since the information in the Web is often not very well-structured and its representation contains many exceptions and pitfalls, a combination of rapid prototyping and refinement is well-suited for this task. The FLORID system[1] [HLLS98] is an implementation of the deductive object-oriented database language F-Logic [KLW95] with additional features for accessing and managing Web documents.

The present case study with FLORID complements the theoretical work of the group on Information Extraction from the Web. Extending the work in [LHL+98], a formal model for representing structure and contents of the Web in a unified object-oriented model has been proposed in [May99]. In [MHLL99], we presented generic rules for typical extraction, integration, and restructuring tasks, such as HTML lists and tables, and syntactical markup such as commalists. The architecture of FLORID as a system for exploring the Web and extracting relevant information is described in [May00]. In the case-study, we employ FLORID for querying Web pages providing geographical and political data with different representation in order to create a geographic database. The main data sources are the CIA World Factbook, and a collection of data about administrative divisions and cities, called *Global Statistics*. The data is enriched by information about geographical coordinates.

On the wrapper level, the information is translated into an F-Logic model as a common framework, retaining an individual model for each of the data sources. The second task, which is usually performed by mediators, is the integration of the individual models, i.e., fusing the corresponding country and city objects.

The history of this case-study. The case study serves two purposes: this report documents the use of FLORID for extracting information from the Web – the second goal is the database itself: Since the end of the eighties, the TERRA[2] database [DR90] is used at the Computer Science department at Karlsruhe University for a practical database training in ORACLE. My first experience with TERRA was in the summer term '92 when I took part in this well-designed training. In 1997, then as a member of the staff of the database group at Freiburg University, I designed the database training at Freiburg around the TERRA database, which is freely available for educational purposes. At this time, there also was the idea to update the database with the political changes in Europe which have happened in the early 90s and to redesign it for ORACLE8.

At the same time, FLORID has been extended with built-ins for Web-Access and regular expressions. A small excerpt of the CIA world factbook was amongst

[1] available at `http://www.informatik.uni-freiburg.de/~dbis/florid/`.

[2] TERRA is Copyright by *Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe*.

the first examples (cf. [HLLS97]). In the meantime, the SGML parser "nsgmls" had been integrated into Florid [Sec98]. Then, for the second edition of the database training in winter term 98/99, the new database Mondial has been set up, using Florid for data extraction and integration.

2 The Main Data Sources

2.1 The CIA World Factbook

The CIA World Factbook (`http://www.odci.gov/cia/publications/pubs.html`) provides political, economic, and social and some geographical information about the countries. The 1996 Factbook pages consists of formatted text, thus, the wrapper consists mainly of pattern matching. The regular structure of the pages allows for very generic and easily extendible patterns for single-valued (both numeric and string) properties and multivalued properties. The latter comprise lists of strings (e.g., organizational memberships, bordering countries) and lists of pairs (name, percentage) of values – e.g., languages, ethnic groups, and religions. Here, we show the adequacy of matching with regular expressions.

A separate part of the CIA World Factbook provides information about political and economical organizations. The pages are based on HTML logical markup structured as nested HTML-lists. Here, the wrapper uses the built-in SGML Parser of Florid for mapping the lists to an internal tree-like object structure. For every organization, a list of properties is given. The list items are then queried by pattern matching, filling the respective methods of the organization. The members of an organization are also given as a set of commalists (divided by the type of membership) of the respective countries.

2.2 Global Statistics: Cities and Provinces

The *Global Statistics* Data (`http://www.stats.demon.nl`) collected by Johan van der Heijden provides information about administrative divisions (area and population) and main cities (population). The data is provided by tables, but with an irregular structure of columns. Here, given a table, it must be discovered if it is a table containing administrative divisions or cities, and the semantics of the columns has to be detected.

Again, the built-in SGML Parser is used for mapping the HTML tables to an internal tree-like object structure. This representation is then pre-evaluated into information about table headers. Based on the header-entries, the semantics of tables and their columns is analyzed by deductive rules and the data is transformed into an object-oriented representation.

2.3 Qiblih: Geographical Coordinates

The Qiblih pages (`http://www.bcca.org/misc/qiblih/latlong.html`) provide the geographical coordinates of many cities around the world, represented by HTML tables and structured text. Here, the data is extracted again by pattern matching.

2.4 The Original Terra Database

The Karlsruhe TERRA database can be seen as the origin and the "kernel" of the idea of the MONDIAL database. There, geographical and political information about countries, administrative divisions, cities, organizations, waters, mountains, deserts etc. is stored in a relational schema used for a practical training in ORACLE at Karlsruhe University. The data is provided as plain text, here again pattern matching is employed.

3 Data Extraction

3.1 The CIA World Factbook: Countries

```
?- sys.theOM.debugOn.     % trace Web access
?- sys.theOM.eqTraceOn.     % trace derived equalities
?- sys.prn.style@("bound"). % set output mode
```

First, several entry points for the data source are defined. We use the local mirror (with the Factbook from 1996):

```
cia[home -> "http://www.odci.gov/cia/publications/nsolo/factbook/" ;
    entry -> "file:/home/dbis/.public_html/.mir/ciawfb/wfb-all.htm";
    mirror -> "file:/home/dbis/.public_html/.mir/ciawfb/" ].
cia[src -> cia.mirror].
```

The continents are associated manually with the respective pages.

```
europe:continent[file@(cia)->"eur.htm"; name->"Europe"].
asia:continent[file@(cia)->"asia.htm"; name->"Asia"].
northAmerica:continent[file@(cia)->"namr.htm"; name->"North America"].
australia:continent[file@(cia)->"oce.htm"; name->"Australia/Oceania"].
southAmerica:continent[file@(cia)->"samr.htm"; name->"South America"].
africa:continent[file@(cia)->"afr.htm"; name->"Africa"].
C[url@(cia)->U] :-
    C:continent[file@(cia)->FN], strcat(cia.src,FN,U).
```

rel_continent can be initialized to restrict the evaluation to a single continent. Otherwise, all continents are made relevant.

```
exists_rel_continent :- C:rel_continent.
?- sys.strat.doIt.
C: rel_continent :- C:continent, not exists_rel_continent.
?- sys.strat.doIt.
```

For all relevant continents, the respective WWW pages are accessed by the built-in method url[get⇒webdoc].

```
U:url.get :- C:rel_continent[url@(cia)->U].
```

Several auxiliary formatting methods are defined, using FLORID's perl interface.

```
format[country_name->"$out[0]=$in[0];
   $out[0]=~s/\AUS/United States/g;
```

```
$out[0]=~s/UAE/United Arab Emirates/g;
$out[0]=~s/The Former Yugoslav Republic of Macedonia/Macedonia/g;
$out[0]=~s/'//g;
$out[0]=~s/([^,]*), South/South $1/g;
$out[0]=~s/([^,]*), North/North $1/g;
$out[0]=~s|\Athe \s*||g;
$out[0]=~s|\AThe \s*||g;
$out[0]=~s/([^(]*)\((.*)/$1/g;
$out[0]=~s/([^;]*);(.*)/$1/g;
$out[0]=~s/([^,]*),(.*)/$1/g;
$out[0]=~s|([^/]*)/(.*)|$1|g;
$out[0]=~s/\s*\Z//g";
    city_name->"$out[0]=$in[0];
$out[0]=~s/'//g;
$out[0]=~s/-/ /g;
$out[0]=~s/([^(]*)\((.*)/$1/g;
$out[0]=~s/([^;]*);(.*)/$1/g;
$out[0]=~s/([^,]*),(.*)/$1/g;
$out[0]=~s|([^/]*)/(.*)|$1|g;
$out[0]=~s/\s*\Z//g";
    org->"$out[0]=$in[0];
$out[0]=~s/G- /G-/g;
$out[0]=~s/\s*\Z//g";
    seas->"$out[0]=$in[0];
$out[0]=~s/'//g;
$out[0]=~s/\A/A,/g;
$out[0]=~s/Equator//g;
$out[0]=~s/Hawaii//g;
$out[0]=~s/ of / Of /g;
$out[0]=~s/ [a-z- ]* /,/g;
$out[0]=~s/\A[a-z-]* //g;
$out[0]=~s/ [a-z-]*\Z//g;
$out[0]=~s/\A[a-z-]*\Z/ /g;
$out[0]=~s/,Of/,/g;
$out[0]=~s/Of,/,/g;
$out[0]=~s/,,/,/g;
$out[0]=~s/,,/,/g";
    date->"$out[0]=$in[0];
$out[0]=~s/\A([0-9]) /0$1 /g;
$out[0]=~s/\ANA ([A-Z])/01 $1/g;
$out[0]=~s/\A([0-9]{4})/01 January $1/g;
$out[0]=~s/\A([0-9]{3}) AD/01 January 0$1/g;
$out[0]=~s/\A([0-9]{3}) BC/01 January -0$1/g;
$out[0]=~s/\A([0-9]{3}) /01 January 0$1/g;
$out[0]=~s/January/01/g;
$out[0]=~s/February/02/g;
$out[0]=~s/March/03/g;
$out[0]=~s/April/04/g;
$out[0]=~s/May/05/g;
$out[0]=~s/June/06/g;
```

```
   $out[0]=~s/July/07/g;
   $out[0]=~s/August/08/g;
   $out[0]=~s/September/09/g;
   $out[0]=~s/October/10/g;
   $out[0]=~s/November/11/g;
   $out[0]=~s/December/12/g;"].
```

`delete.little_words` deletes all non-capitalized words in commalists (used for lists of religions, languages etc).

```
delete[little_words->"$out[0]=$in[0];
   $out[0]=~s/'//g;
   $out[0]=~s/\s*[0-9]* / /g;
   $out[0]=~s/ [a-z ]* / /g;
   $out[0]=~s/\A[a-z]* //g;
   $out[0]=~s/ [a-z]*\Z//g;
   $out[0]=~s/\A[a-z]*\Z/ /g"].
```

`replace.spaces` replaces spaces by hyphens (needed later for XML id's).

```
replace[spaces->"$out[0]=$in[0];
   $out[0]=~s/ /-/g;
   $out[0]=~s/\s*[0-9]* / /g"].
?- sys.strat.doIt.
```

Evaluating CIA continent pages. Every continent page contains a list of links to the countries. The labels contain the country names (must be reformatted) and the `href` tags contain the respective urls. The country objects are defined with their urls, names, and continents, and the urls are accessed. Note that countries which belong to several countries are associated with only one continent.

```
cid(cia,C3):country
   [url@(cia) -> U; name@(cia)-> C2; continent -> CN ] :-
   CT:rel_continent[name->CN].url@(cia).get[hrefs@(Label) ->> U],
   pmatch(Label,"/(.*) \([0-9]/","$1", C),
   perl(format.country_name, C, C2),
   perl(replace.spaces, C2, C3).
U:url.get :- _:country[url@(cia)->U].
?- sys.strat.doIt.
```

Evaluating CIA country pages. The CIA pages contain structured text which is best evaluated by matching with (perl) regular expressions.

Patterns for scalar methods. The following patterns are used for defining scalar methods (of the given datatypes number and string). On the CIA country pages, each property is introduced by a keyword:

```
pattern(capital,"/Capital:.*\n(.*)/","","string").
pattern(independence,"/Independence:.*\n(.*) \(/","","string").
pattern(total_area,"/total area:.*\n(.*?) sq km/","NA","num").
```

For population it must be taken care that sometimes the item is called "Population", and sometimes "total population" (the latter for Serbia and Montenegro):

```
pattern(population,"/Population:.*\n([0-9,]+) \(/","","num").
pattern(population,
  "/total population:.*\n([0-9,]+) \(/","","num").
pattern(population_growth,
  "/Population growth rate:.*\n(.*)\%/","","num").
pattern(infant_mortality,
  "/Infant mortality rate:.*\n(.*) deaths.1,000 live births/",
  "NA","num").
pattern(gdp_total,
  "/GDP:.*\n.*purchasing power parity.*\$(.*) \(/","","extnum").
```

A special pattern for Germany (where total, west, and east are distinguished):

```
pattern(gdp_total,
  "/GDP:.*\n.*\npurchasing power parity.*\$(.*) \(/","","extnum").
pattern(gdp_agri,"/agriculture:.*\n(.*)\%/","NA","num").
pattern(gdp_ind,"/industry:.*\n(.*)\%/","NA","num").
pattern(gdp_serv,"/services:.*\n(.*)\%/","NA","num").
pattern(inflation,"/Inflation rate.*:.*\n(.*)\%/","NA","num").
pattern(datacode,"/Data code:.*\n(.*)/","","string").
pattern(government,"/Type of government:.*\n(.*)/","","string").
```

The location of a country is described in a string which gives the continent, and he bordering seas and countries.

```
pattern(location,"/Location:<.B>\n[^,]*,(.*)/","","extstring").
```

The raw results of matching have to be refined for storing them in the database: Numerical values are sometimes given as millions, billions, and trillions, and with the US decimal comma.

```
value("mi",1).
value("bi",1000).
value("tri",1000000).
```

Decimal commas and list-commas have to be distinguished. The former are enclosed directly by numbers whereas the latter are followed by a space and are replaced by semicolons.

```
replace[kommas->"$out[0]=$in[0];
   $out[0]=~s/([0-9]),([0-9])/$1$2/g;
   $out[0]=~s/, /;/g;
   $out[0]=~s/\s*;/;/g"].
is[num->"$_ = $in[0];
         if (/[a-zA-Z]/){ $out[0]=0 } else { $out[0]=1 };"].
?-sys.strat.doIt.
```

The patterns are applied to the WWW pages (accessible via url[get⇒webdoc]). String patterns, fully numerical patterns, numerical patterns with millions, numerical patterns with millions/billions/trillions, and dates are handled by specializations of a common schema:

```
C[Method -> X2] :-
   pattern(Method, RegEx, Excl,"string"),
   pmatch(C:country.url@(cia).get, RegEx, "$1", X),
   perl(format.city_name,X,X2),
   not X = Excl.
C[Method -> X2] :-
   pattern(Method, RegEx, Excl,"num"),
   pmatch(C:country.url@(cia).get, RegEx, "$1", X),
   not X = Excl, not substr("llion",X),
   perl(replace.kommas, X, X2),
   perl(is.num,X2,E), E = 1.
C[Method -> X3] :-
   pattern(Method, RegEx, Excl,"num"),
   pmatch(C:country.url@(cia).get, RegEx, "$1", X),
   pmatch(X, "/(.*) million/", "$1", X1),
   perl(replace.kommas, X1, X2), X3 = X2 * 1000000.
C[Method@(0) -> S2] :-
   pattern(Method, RegEx, Excl,"extnum"),
   pmatch(C:country.url@(cia).get, RegEx, "$1", S),
   not S = Excl,
   perl(replace.kommas, S, S2).
C[Method -> X2] :-
   pattern(Method,_,_,"extnum"),
   C:country[Method@(0) -> X],
   pmatch(X, "/(.*) ([a-z]*)llion/", ["$1","$2"], [Mant,ExpName]),
   value(ExpName,Exp), string2float(Mant,Mantfloat),
   X2 = Mantfloat * Exp.
```

The string-valued independence property is further evaluated to yield a valid date string.

```
C[indep_date->D2] :-
   C:country[independence->S],
   perl(format.date,S,D),
   pmatch(D,"/\A([0-9]{2} [0-9]{2} [0-9]{4})/", "$1", D2).
```

The location is evaluated wrt. the bordering seas: format.seas deletes all non-capitalized words and produces a comma-list of the geographic names in the string. The comma list is then split and all country and continent names are skipped – obtaining the bordering seas' names. Note that these data are not complete.

```
C[location -> X2] :-
   pattern(location, RegEx, _,"extstring"),
   pmatch(C:country.url@(cia).get, RegEx, "$1", X),
   perl(format.seas,X,X2).
C[coasts->>X] :-
   C:country[location->S],
   pmatch(S, "/,\s*([^,]*)/g", "$1", X),
   not _:country[name@(cia)->X], not _:continent[name->X].
?- sys.strat.doIt.
```

The CIA world factbook contains not only countries, but also islands with more or less (military) relevance – such as e.g. Glorioso Islands, consisting of a runway built upon a bunch of corals a runway. Real countries are only those where a capital is given.

```
C:real_country :-
   C:country[capital->CA], not substr("none", CA).
?- sys.strat.doIt.
```

Patterns for lists and multivalued methods. Ethnic groups, religions, languages, border countries and membership in political and economic organizations are given by comma-separated lists, again introduced by a keyword. The list items are e.g., built from a number and a percent symbol or "km".

```
commapattern(ethnicgroups,
    "/Ethnic divisions:.*\n(.*)/", "\%", "num").
commapattern(languages, "/Languages:.*\n(.*)/", "\%", "num").
commapattern(religions, "/Religions:.*\n(.*)/", "\%", "num").
```

Note that for border countries or a single border country, different keywords are used.

```
commapattern(borders, "/border countries:.*\n(.*)/", " km", "num").
commapattern(borders, "/border country:.*\n(.*)/", " km", "num").
commapattern(membership, "/participation:.*\n(.*)/", "", "enum").
```

In a first step, for every pattern, the whole answer string, i.e. the raw commalist is extracted (note that every commalist is prefixed with an additional comma for an easier handling):

```
C[str(M)->>L2] :-
  commapattern(M,S,_,_),
  pmatch(C:country.url@(cia).get, S, "$1", L),
  strcat(", ",L,L2).
```

Comments and details (enclosed in parentheses) are ignored – the intended stripped commalist does not contain any "(":

```
C[str(M)->>L2] :-
  C:real_country[str(M)->>L],
  pmatch(L, "/(.*)\([^\(]*\)(.*)/", ["$1$2"], [L2]).
?- sys.strat.doIt.
```

In the stripped commalists, decimal commas are deleted, whereas list-commas are replaced by semicolons:

```
C[str(M)->L2] :-
  C:real_country[str(M)->>L],
  not substr("(",L),
  perl(replace.kommas, L, L2).
```

The resulting semicolon-lists are broken into pieces at the semicolons to multivalued methods.

```
C[M->>X2] :-
  C:real_country[str(M)->S],
  pmatch(S, "/;([^;]*)/g", "$1", X),
  perl(format.org,X,X2).
?- sys.strat.doIt.
```

Depending on the "datatype" of the list items, the values are extracted from the individual list elements. Note that the resulting parametrized methods, e.g. country[religions@(religion)⇒number] are still multivalued. Different values result e.g. from Muslim(Sunni) and Muslim(Shi'a) which – stripping the parentheses – both result in Muslim.

```
C[M@(X2)->>Vfloat] :-
  commapattern(borders,_, S,"num"),
  strcat("/([^0-9]*)\s* ([0-9\.]+)",S,S2),
  strcat(S2,"/",Pattern),
  pmatch(C:real_country..M,
 Pattern, ["$1","$2"], [X,V]),
  perl(format.country_name, X, X2),
  string2float(V,Vfloat).
C[M@(X2)->>Vfloat] :-
  commapattern(M,_, S,"num"), not M = borders,
  strcat("/([^0-9]*)\s* ([0-9\.]+)",S,S2),
  strcat(S2,"/",Pattern),
  pmatch(C:real_country..M,
 Pattern, ["$1","$2"], [X,V]),
  perl(delete.little_words, X, X2), not X2 = " ",
  string2float(V,Vfloat).
```

If an element of the languages, religions, or ethnic groups list contains no explicit percentage, it is associated the null value.

```
C[M@(Str2)->>null] :-
  commapattern(M,_, S,"num"), not M = borders,
  C:real_country[M->>Str],
  not substr("%",Str),
  not substr("</I>",Str),
  perl(delete.little_words, Str, Str2), not Str2 = " ".
?- sys.strat.doIt.
```

The multivalued methods are summmed up.

```
C[M@(X)->V] :-
   commapattern(M,_,_,"num"), not M = borders,
   V = sum{Val [C,M,X]; C:country[M@(X)->>Val]},
   not V = 0, V < 101.
```

For country[borders@(country)⇛number], the country objects are linked.

```
C[borders@(C2)->V] :-
   V = sum{Val [C,X]; C:country[borders@(X)->>Val]},
   C2:country[name@(cia)->X],  not V = 0.
?- sys.strat.doIt.
```

Some lists contain items without explicit percentage. Above, this has been set to the null value. Elements of unary lists are now set to 100%, others remain nulls.

```
C[M@(X)->#100] :-
   commapattern(M,_, S,"num"), not M = borders,
   C:country..M@(X)[], not C.M@(X)[],
   N = count{X [C,M]; C:country..M@(X)[],
   commapattern(M,_, _,"num")}, N = 1.
C[M@(X)->null] :-
   commapattern(M,_, S,"num"), not M = borders,
   C:country[M@(X)->>null], not C.M@(X)[],
   N = count{X [C,M]; C:country.M@(X)[],
   commapattern(M,_, _,"num")}, N > 1.
?- sys.strat.doIt.
```

Output. Using the instance output format, the resulting atoms are selected for output to mondial-cia-facts.flp.

```
save[init->"‘rm ../Mondial-facts/mondial-cia-facts.flp‘; @out=@in;"].
```

for integration, capital is called capital@(cia). Note that is is a simple string-valued, not an object-valued method.

```
C[capital@(cia)->Cap] :- C:country[capital->Cap].
```

All numerical methods are ouptpu by a generic rule.

```
num(Method) :- pattern(Method,_,_,"num").
?- sys.strat.doIt.
?- perl(save.init,[""],[""]).
?- sys.prn.style@("instance").  % set output mode
?- sys[output->sys.open@("../Mondial-facts/mondial-cia-facts.flp")].
?- sys.answerChannel.setStream@(output)[].
```

The following output queries also define the schema of the CIA part of the database

```
 ?- C:continent[name->N].
 ?- C:country[name@(cia)->N;datacode->DC;continent->CT].
 ?- C[Method->V], num(Method).
 ?- C[gdp_total->V].
 ?- C[indep_date->D].
 ?- C[government->V].
 ?- C[capital@(cia)->V].
 ?- C[ethnicgroups@(X)->V].
 ?- C[religions@(X)->V].
 ?- C[languages@(X)->V].
 ?- C[borders@(X)->V].
 ?- C[coasts->>V].
?- sys.answerChannel.resetStream[].
?- sys.remove@("output").
```

3.2 The CIA World Factbook: Organizations

```
?- sys.theOM.debugOn.      % trace Web access
?- sys.theOM.eqTraceOn.    % trace derived equalities
?- sys.prn.style@("bound"). % set output mode
```

Define several entry points for the data source:

```
ext[cia_mirror -> "file:/home/dbis/.public_html/.mir/ciawfb/";
    organizations -> "wfb-appc.htm"].
?- sys.strat.doIt.
```

The CIA organization page is structured as a list of organizations. For every organization, there is an inner list, containing an item for each property. The page is parsed by the built-in HTML parser, obtaining a parse-tree of the page using a special tagfile (specifying which HTML tags are ignored or pruned).

```
?- sys.theOM.setTagFile@("mondial-orgs.tags").
U:url.parse :- strcat(ext.cia_mirror,ext.organizations,U).
?-sys.strat.doIt.
```

Several auxiliary formatting methods are defined, using FLORID's perl interface.

```
format[memberlist->"$out[0]=$in[0];
             $out[0]=~s/'//g;
             $out[0]=~s/\s*\\n//g;
             $out[0]=~s|,\s*|,|g;
             $out[0]=~s/,\s*and /,/g;
             $out[0]=~s/\A\The //g;
             $out[0]=~s/\A\The //g;
             $out[0]=~s/,\s*The /,/g;
             $out[0]=~s/Former Yugoslav Republic of //g;
             $out[0]=~s/\"//g;
             $out[0]=~s/ \([^\)]*\)//g;
             $out[0]=~s|;.*||g"].
format[membertype->"$out[0]=$in[0];
             $out[0]=~s/\s*\([^)]*\)//g;
             $out[0]=~s/ies\Z/y/g;
             $out[0]=~s/s\Z//g"].
format[organization->"$out[0]=$in[0];
             $out[0]=~s/'//g"].
format[seatcity->"$out[0]=$in[0];
             $out[0]=~s/'//g;
             $out[0]=~s/[-, ][A-Z0-9]*\Z/,/g;
             $out[0]=~s/[-, ][A-Z0-9]*[-, \Z]/,/g;
             $out[0]=~s/[-, ][A-Z0-9]*[-, ]/,/g;
             $out[0]=~s/[-, ][A-Z0-9]*[-, ]/,/g;
             $out[0]=~s|[0-9]*||g;
             $out[0]=~s/,\Z//g;
             $out[0]=~s/ \Z//g;
             $out[0]=~s/\A //g;
             $out[0]=~s/The Hague/s-Gravenhage/g;
```

```
              $out[0]=~s/Berne/Bern/g;
              $out[0]=~s/Teheran/Tehran/g;
              $out[0]=~s/.*,([^,]*)\Z/$1/g"].
format[seatcountry->"$out[0]=$in[0];
              $out[0]=~s/'//g;
              $out[0]=~s/St\./Saint/g;
              $out[0]=~s/Syrian AR/Syria/g;
              $out[0]=~s/Iran Islamic Republic/Iran/g;
              $out[0]=~s/UK/United Kingdom/g"].
format[date->"$out[0]=$in[0];
              $out[0]=~s|\A-\s*||g;
              $out[0]=~s|([0-9])-[0-9]*|$1|g;
              $out[0]=~s/NA 19/01 January 19/g;
              $out[0]=~s/NA ([A-Z]*)/01 $1/g;
              $out[0]=~s/\A([0-9]) /0$1 /g;
              $out[0]=~s/\([^)].*\)//g;
              $out[0]=~s|,.*||g;
              $out[0]=~s/January/01/g;
              $out[0]=~s/February/02/g;
              $out[0]=~s/March/03/g;
              $out[0]=~s/April/04/g;
              $out[0]=~s/May/05/g;
              $out[0]=~s/June/06/g;
              $out[0]=~s/July/07/g;
              $out[0]=~s/August/08/g;
              $out[0]=~s/September/09/g;
              $out[0]=~s/October/10/g;
              $out[0]=~s/November/11/g;
              $out[0]=~s/December/12/g;
              $out[0]=~s|([0-9]{2} [0-9]{2} [0-9]{4}).*|$1|g;"].
```

The possible types of membership are listed.

```
membertype("regional members").
membertype("nonregional members").
membertype("members").
membertype("associate members").
membertype("participating governments").
membertype("associate member").
membertype("observers").
membertype("observer").
membertype("pending members").
membertype("special members").
membertype("guests").
membertype("cooperating state").
membertype("membership applicants").
membertype("subbureaus").
membertype("Part I").
membertype("Part II").
membertype("Category I").
membertype("Category II").
```

```
membertype("Category III").
membertype("National Olympic Committees").
membertype("correspondent members").
membertype("subscriber members").
membertype("National Societies").
membertype("nonsignatory users").
membertype("special member").
membertype("partners for cooperation").
membertype("members (Board of Trustees)").
membertype("members (associated institutes)").
membertype("associate partners").
membertype("applicants").
```

For several countries, the name used on the CIA organization information page differs from that used in the rest of the Factbook.

```
substitute("Democratic Republic of the Congo","Congo").
substitute("Republic of the Congo","Zaire").
substitute("The Former Yugoslav Republic of Macedonia","Macedonia").
substitute("Russian","Russia").  %% Fipptehler
substitute("US","United States").
substitute("UK","United Kingdom").
substitute("NZ","New Zealand").
substitute("UAE","United Arab Emirates").
?- sys.strat.doIt.
```

For every letter, there is a list with organizations whose abbreviation starts with this letter.

```
L:pol_org_list :- L:(U.parse.ul), L = U.parse.root.html@(X),
               strcat(ext.cia_mirror,ext.organizations,U).
?- sys.strat.doIt.
```

The organisation-lists are now decomposed, yielding the data or the individual orgenizations. Every organization object is identified by its abbreviation. Additionally, the full name is extracted in the first step. The organization data itself is again a list. Additional checks are needed to ignore items of the form "X, see Y".

```
org(Short):org[abbrev->Short; name->Long2; data->DataList] :-
  L:pol_org_list[ul@(K)->OrgItem],
  OrgItem[li@(0)->Name], string(Name),
  L.ul@(K1)[li@(0)->DataList], K1 = K + 1,
  DataList.ul@(0)[],
  pmatch(Name,"/(.*) \(([^)]*)\)/",["$1","$2"],[Long,Short]),
  perl(format.organization,Long,Long2).
?- sys.strat.doIt.
```

For every organizations, several properties are listed (as an HTML list structure), e.g., its address, its establishment date, and a list of members of the different member types. For every property, the corresponding list item can be identified by a keyword ("address" or "established") or a member type. The list items are then matched with perl regular expressions, additionally using the above formatting methods.

```
O[seatcity->City;seatcountry->C2] :-
  O:org[data->D],
  D.ul@(X)[li@(0)->"address";li@(1)->Address],
  pmatch(Address,"/(.*)[,0-9] ([^,0-9]*)\Z/",
         ["$1","$2"],[CityLong,Country]),
  perl(format.seatcity,CityLong,City),
  perl(format.seatcountry,Country,C2).
O[established->Date2] :-
  O:org[data->D],
  D.ul@(X)[li@(0)->"established";li@(1)->EstablishDate],
  perl(format.date,EstablishDate,Date),
  pmatch(Date,"/\A([0-9]{2} [0-9]{2} [0-9]{4})/", "$1", Date2).
?- sys.strat.doIt.
```

For each of the membertypes, a commalist with the corresponding countries and the number of members is given.

```
O[memberlist@(Type2) -> Liste3; anzahl@(Type2) ->Anzahl2] :-
  O:org[data->D],
  D.ul@(X)[li@(0)->Type;li@(1)->Liste], membertype(Type),
  perl(format.membertype,Type,Type2), string(Liste),
  pmatch(Liste,"/\A-\(([0-9]*)[^)]*\)(.*)/",
         ["$1",",$2"],[Anzahl,Liste2]),
  perl(format.memberlist,Liste2,Liste3),
  string2float(Anzahl,Anzahl2).
```

The commalists are evaluated by matching, some country names are substituted.

```
O[member_names@(Type)->>Name] :-
    O:org[memberlist@(Type) -> Liste;anzahl@(Type) ->Anzahl],
    pmatch(Liste, "/,([^,]*)/g", "$1", Name),
    not substitute(Name,_).
O[member_names@(Type)->>Name2] :-
    O:org[memberlist@(Type) -> Liste;anzahl@(Type) ->Anzahl],
    pmatch(Liste, "/,([^,]*)/g", "$1", Name),
    substitute(Name,Name2).
?- sys.strat.doIt.
```

Output. Using the instance output format, the resulting atoms are selected for output to mondial-orgs-facts.flp.

```
save[init->"`rm ../Mondial-facts/mondial-orgs-facts.flp`; @out=@in;"].
?- sys.strat.doIt.
?- perl(save.init,[""],[""]).
?- sys.prn.style@("instance").  % set output mode
?- sys[output->sys.open@("../Mondial-facts/mondial-orgs-facts.flp")].
?- sys.answerChannel.setStream@(output)[].
```

The following output queries also define the schema of the organizations part of the database

```
 ?- O:org[abbrev->Short; name->Long].
```

```
 ?- O:org[established->Date].
 ?- O:org[seatcity->City; seatcountry->Country].
 ?- O:org[member_names@(Type)->>Name].
?- sys.answerChannel.resetStream[].
?- sys.remove@("output").
```

3.3 Global Statistics: Cities and Provinces

```
?- sys.theOM.debugOn. % trace Web access
?- sys.theOM.eqTraceOn. % trace derived equalities
?- sys.prn.style@("bound").  % set output mode
```

Define several entry points for the data source:

```
gs[home -> "http://home.worldonline.nl/~quark/" ;
    mirror -> "file:/home/dbis/.public_html/.mir/wol/" ].
gs[src->gs.mirror].
```

Several auxiliary formatting methods are defined, using FLORID's perl interface. Mainly, expressions

```
format[country_name->"$out[0]=$in[0];
             $out[0]=~s|\Athe \s*||g;
             $out[0]=~s|\AThe \s*||g;
             $out[0]=~s/\s*\Z//g;
             $out[0]=~s/([^,]*), South/South $1/g;
             $out[0]=~s/([^,]*), North/North $1/g;
             $out[0]=~s/,.*//g;
             $out[0]=~s/St\./Saint/g;
             $out[0]=~s/ & / and /g;
             $out[0]=~s/'//g;
             $out[0]=~s/ I\Z//g;";
       capital->"$out[0]=$in[0];
             $out[0]=~s/'//g;
             $out[0]=~s/([^(]*)\((.*)/$1/g;
             $out[0]=~s/([^;]*);(.*)/$1/g;
             $out[0]=~s/([^,]*),(.*)/$1/g;
             $out[0]=~s|([^/]*)/(.*)|$1|g;
             $out[0]=~s/\s*\Z//g";
       province->"$out[0]=$in[0];
             $out[0]=~s/'//g;
             $out[0]=~s/\?//g;
             $out[0]=~s/([^(]*)\((.*)/$1/g;
             $out[0]=~s/([^[]*)\[(.*)/$1/g;
             $out[0]=~s/-/ /g;
             $out[0]=~s/ & / and /g;
             $out[0]=~s/s[^a-z ]ka/ska /g;
             $out[0]=~s/z[^a-z ]ka/zka /g;
             $out[0]=~tr/áâãäåÉéèíïìÍÖóôöúüçñ/aaaaaEeeiiiIOooouucn/;
             $out[0]=~s/\s*\Z//g";
```

```
        city_name->"$out[0]=$in[0];
              $out[0]=~s/'//g;
              $out[0]=~s/`//g;
              $out[0]=~s/Adazapari/Adapazari/g;
              $out[0]=~s/Karkuk/Kirkuk/g;
              $out[0]=~s/Baranquilla/Barranquilla/g;
              $out[0]=~s/Ferrant/Ferrand/g;
              $out[0]=~s/Santafé de Bogotá/Bogota/g;
              $out[0]=~s/Metropolitan Lima/Lima/g;
              $out[0]=~s/Recklingshausen/Recklinghausen/g;
              $out[0]=~s/Petropavlovsk.\n.Kamchatsky/Petropavlovsk Kamchatsky/g;
              $out[0]=~s/Petropavlovsk...Kamchatsky/Petropavlovsk Kamchatsky/g;
              $out[0]=~s/([^,]*),(.*)/$1/g;
              $out[0]=~s|([^/]*)/(.*)|$1|g;
              $out[0]=~s/([^(]*)\((.*)\)/$1/g;
              $out[0]=~s/St.([A-Z])/St. $1/g;
              $out[0]=~s/-/ /g;
              $out[0]=~tr/áâãäåÉéèíïìÍÖóôöúüçñ/aaaaaEeeiiiIOooouucn/;
              $out[0]=~s/\s*\Z//g"].
```

Decimal commas are deleted.

```
replace[kommas->"$out[0]=$in[0];
                 $out[0]=~s|\s*||g;
                 $out[0]=~s|\..*||g;
                 $out[0]=~s/\(.*\)//g;
                 $out[0]=~s/([0-9]),([0-9])/$1$2/g;
                 if (!($out[0] =~/[0-9]/)) {$out[0] = 0};"].
replace[ixtodata->"$out[0]=$in[0];
                 $out[0]=~s/ix/data/g"].
replace[spaces->"$out[0]=$in[0];
   $out[0]=~s/ /-/g;
   $out[0]=~s/\s*[0-9]* / /g"].
```

GS contains an error, confusing Congo (Zaire, formerly Belgian Congo) and Congo (Democratic Republic, formerly Congo-Brazzaville, belonging to french central africa). The main cities and provinces of Zaire are attributed to the Congo.

```
repl[congo_zaire->"$out[0]=$in[0];
                 $out[0]=~s/Congo, Dem.Rep./Zaire/g;
                 $out[0]=~s/Congo/Zaire/g"].
```

the US state abbreviations (postal codes) are mapped to the state names.

```
us_postal_code("AL","Alabama").
us_postal_code("LA","Louisiana").
us_postal_code("OH","Ohio").
us_postal_code("AK","Alaska").
us_postal_code("ME","Maine").
us_postal_code("OK","Oklahoma").
us_postal_code("AZ","Arizona").
```

```
us_postal_code("MD","Maryland").
us_postal_code("OR","Oregon").
us_postal_code("AR","Arkansas").
us_postal_code("MA","Massachusetts").
us_postal_code("PA","Pennsylvania").
us_postal_code("CA","California").
us_postal_code("MI","Michigan").
us_postal_code("RI","Rhode Island").
us_postal_code("CO","Colorado").
us_postal_code("MN","Minnesota").
us_postal_code("SC","South Carolina").
us_postal_code("CT","Connecticut").
us_postal_code("MS","Mississippi").
us_postal_code("SD","South Dakota").
us_postal_code("DE","Delaware").
us_postal_code("MO","Missouri").
us_postal_code("TN","Tennessee").
us_postal_code("DC","Distr. Columbia").
us_postal_code("MT","Montana").
us_postal_code("TX","Texas").
us_postal_code("FL","Florida").
us_postal_code("NE","Nebraska").
us_postal_code("UT","Utah").
us_postal_code("GA","Georgia").
us_postal_code("NV","Nevada").
us_postal_code("VT","Vermont").
us_postal_code("HI","Hawaii").
us_postal_code("NH","New Hampshire").
us_postal_code("VA","Virginia").
us_postal_code("ID","Idaho").
us_postal_code("NJ","New Jersey").
us_postal_code("WA","Washington").
us_postal_code("IL","Illinois").
us_postal_code("NM","New Mexico").
us_postal_code("WI","Wisconsin").
us_postal_code("IN","Indiana").
us_postal_code("NY","New York").
us_postal_code("WY","Wyoming").
us_postal_code("IA","Iowa").
us_postal_code("NC","North Carolina").
us_postal_code("KS","Kansas").
us_postal_code("ND","North Dakota").
us_postal_code("KY","Kentucky").
```

Continents. The continents are associated manually with the respective pages.

```
europe:continent[file@(gs)->"europe/euix.htm"; name->"Europe"].
asia:continent[file@(gs)->"asia/asix.htm"; name->"Asia"].
america:continent[file@(gs)->"america/amix.htm"; name->"America"].
australia:continent[file@(gs)->"oceania/ocix.htm"; name->"Australia/Oceania"].
```

```
africa:continent[file@(gs)->"africa/afix.htm"; name->"Africa"].
C[url@(gs)->U] :- C:continent[file@(gs)->A], strcat(gs.src,A,U).
```

rel_continent can be initialized to restrict the evaluation to a single continent. Otherwise, all continents are made relevant.

```
exists_rel_continent :- C:rel_continent.
?- sys.strat.doIt.
C: rel_continent :- C:continent, not exists_rel_continent.
?- sys.strat.doIt.
```

For all relevant continents, the respective WWW pages are accessed by the built-in method url[get⇒webdoc]. Additionally, the continent data pages (urls obtained by replacing "ix" in the continent's url by "data") are parsed using the built-in HTML parser.

```
C:rel_continent[data_url@(gs)->U2] :-
   C:rel_continent[url@(gs)->U], perl(replace.ixtodata,U,U2).
U:url.get :- C:rel_continent[url@(gs)->U].
U:url.parse :- C:rel_continent[data_url@(gs)->U].
?- sys.strat.doIt.
```

General Country Data. The continent data pages contain tables with information about the countries. After parsing, this information is contained in the parse-tree: The class U.parse.table contains all tables of U.parse (here, this is exactly one table per continent page). For a table T, T.table@(0) contains the table body TB, TB.tbody@(n) is the n-th table row, from which tr@(k) contains the k-th column which is divided into several parts by the innermost tags. Depending how many parts this are, and which of them are strings, the contents of the table entry can be derived.

```
element(Table,Row,Column)[contents->Contents;type->Type] :-
  CT:continent[data_url@(gs)->U], Table:(U.parse.table),
  Table.table@(0)[tbody@(Row)->_[tr@(Column)->X
                  [Type@(0)->Contents]]],
  not X.Type@(1)[].
element(Table,Row,Column)[contents->Contents;type->Type] :-
  CT:continent[data_url@(gs)->U], Table:(U.parse.table),
  Table.table@(0)[tbody@(Row)->_[tr@(Column)->X
                  [Type@(0)->Contents;Type@(1)->I0]]],
  not string(I0), string(Contents).
element(Table,Row,Column)[contents->Contents;type->Type] :-
  CT:continent[data_url@(gs)->U], Table:(U.parse.table),
  Table.table@(0)[tbody@(Row)->_[tr@(Column)->X
                  [Type@(0)->I0;Type@(1)->Contents]]],
  not string(I0), string(Contents).
?- sys.strat.doIt.
```

The individual table entries give the name of the country (0th column), its capital name (1st column), its area (2nd column), and its population (3rd column).

```
cid(gs,CT,DZ):country[name_str->N2;
```

```
                   continent@(gs)->>CT;
                   population@(gs)->P2] :-
  CT:continent[data_url@(gs)->U; name->CN], T:(U.parse.table),
  element(T,DZ,0)[contents->N;type->td], string(N), not N="Total",
  element(T,DZ,3)[contents->P;type->td],
  perl(replace.kommas, P, P2),
  perl(format.country_name, N, N2).

C[capital_name->>Cap2] :-   C:country[name_str->N2],
  CT:continent[data_url@(gs)->U], T:(U.parse.table),
  element(T,DZ,0)[contents->N;type->td], string(N), not N="Total",
  element(T,DZ,1)[contents->Cap;type->td],
  perl(format.country_name, N, N2),
  perl(format.capital, Cap, Cap2), strlen(Cap,L), L > 2.
?- sys.strat.doIt.
C[name@(gs)->>"USA"] :- C:country[name@(gs)->>"United States"].
```

Country names of the form "Myanmar (Burma)" (appearing on the Myanmar page) or "Holy See (Vatican City)" have to be resolved into two alternative names.

```
C[name@(gs)->>N] :-
        C:country[name_str->N], not substr("(",N).
C[name@(gs)->>{N1,N2}] :- C:country[name_str->N],
         pmatch(N,"/([^(]*) \((.*)\)/", ["$1","$2"], [N1,N2]).
```

The continent pages given by continent.url contain links labeled by country names to pages containing administrative divisions and main cities for some of the countries. Other links should not be followed:

```
gs[exclude->>{"home", "general data", "cities",
              "population", "economy", "provinces",
              "Profiler", "Germany I" % excludes also Germany II!
}].
?- sys.strat.doIt.
```

The remaining links on continent.url lead to country pages. Note that the slots url.get[hrefs@(label)⇒url] has been filled automatically when calling url.get.

```
CT[link@(Label2)->>U] :-
   CT:rel_continent.url@(gs).get[hrefs@(Label) ->> U],
   not substr(gs..exclude, Label),
   perl(repl.congo_zaire,Label,Label2), %%%<<<<<<<<
   not Label = "".
```

rel_country can be initialized to restrict the evaluation to some countries. Otherwise, all countries are made relevant.

```
exists_rel_country :- C:rel_country.
?- sys.strat.doIt.
Country:rel_country :- CT:continent[link@(Country)->>U],
                       not exists_rel_country.
```

Evaluating GlobalStatistics country pages. The GlobalStatistics country pages contain tables with information about administrative divisions and main cities. They are parsed by the built-in SGML parser, obtaining a parse-tree of the page using a special tagfile (specifying which HTML tags are ignored or pruned).

```
?- sys.theOM.setTagFile@("mondial-gs.tags").
```

For China and the USA, additional city pages are given.

```
asia[link@("China")->>U] :-
    strcat(gs.src,"asia/chinacy.htm",U),
    asia:rel_continent, "China":rel_country.
america[link@("USA")->>U] :-
    strcat(gs.src,"america/usacity.htm",U),
    america:rel_continent, "USA":rel_country.
?- sys.strat.doIt.
U:country_url.parse :-
    C:continent[link@(Country)->>U], Country:rel_country.
?- sys.strat.doIt.
```

Similar to the continent data pages' tables, the table entries on the country data pages are translated from the parse tree.

```
element(Table,Row,Column)[contents->Contents;type->Type] :-
  U:country_url, Table:(U.parse.table),
  Table.table@(0)[tbody@(Row)->_
                  [tr@(Column)->X[Type@(0)->Contents]]],
  not X.Type@(1)[].
element(Table,Row,Column)[contents->Contents;type->Type] :-
  U:country_url, Table:(U.parse.table),
  Table.table@(0)[tbody@(Row)->_[tr@(Column)->X
                  [Type@(0)->I0;Type@(1)->I1]]],
  string(I0), string(I1),
  not X.Type@(2)[], strcat(I0,I1,Contents).
element(Table,Row,Column)[contents->I1;type->Type] :-
  U:country_url, Table:(U.parse.table),
  Table.table@(0)[tbody@(Row)->_[tr@(Column)->X
                  [Type@(0)->I0;Type@(1)->I1]]],
  not string(I0), string(I1), not X.Type@(2)[].
element(Table,Row,Column)[contents->Contents;type->Type] :-
  U:country_url, Table:(U.parse.table),
  Table.table@(0)[tbody@(Row)->_[tr@(Column)->_
                  [Type@(0)->I0;Type@(1)->I1;Type@(2)->I2]]],
  string(I0), string(I1), string (I2),
  strcat(I0,I1,I01), strcat(I01,I2,Contents).
element(Table,Row,Column)[contents->Contents;type->Type] :-
  U:country_url, Table:(U.parse.table),
  Table.table@(0)[tbody@(Row)->_[tr@(Column)->_
                  [Type@(0)->I0;Type@(1)->I1;Type@(2)->I2]]],
  not string(I0), string(I1), string (I2),
  strcat(I1,I2,Contents).
```

```
element(Table,Row,Column)[colspan->Span2] :-
  U:country_url, Table:(U.parse.table),
  Table.table@(0)[tbody@(Row)->_
                  [tr@(Column)->X[colspan -> Span]]],
  not Span = "1", string2float(Span,Span2).
```

The header row(s) are determined by the <th> tags. Additionally, for every column, the header entry (attribute name) is extracted.

```
T[header_row->>HR] :- U:country_url,
   T:(U.parse.table), element(T,HR,_)[type->th].
T[header@(HR,Col)->>String] :-
   U:country_url, T:(U.parse.table)[header_row->>HR],
   element(T,HR,Col)[contents->String;type->th].
?- sys.strat.doIt.
```

Most country pages contain two tables: one with administrative tables, the other with main cities. Additionally, for each of them, the table structure is varying. For the administrative divisions, the capital column is not contained in every table; for the main city table, the *province* column is not contained in every table.

In the first step, the tables of administrative divisions and the tables of main cities are identified (searching a table entry which contains the appropriate header string). Additionally, the countries to which the tables refer are identified.

```
tabtype(maincitytab).
tabtype(admdivtab).
header(maincitytab,"/municipalities of (.*)/").
header(maincitytab,"/main cities of (.*)/").
header(admdivtab,"/administrative divisions of (.*)/").
T:TabType[int_country_name->>LN; int_country_name->>N2] :-
   CT:continent[link@(LN)->>U], T:(U.parse.table),
   element(T,_,_)[contents->I], not substr("II",I),
   header(TabType,Pattern), tabtype(TabType),
   pmatch(I,Pattern,"$1",Name),
   perl(repl.congo_zaire,Name,Name2), %<<<<<<<<<<<<<<<<
   perl(format.country_name, Name2, N2).
?- sys.strat.doIt.
```

References of the form “Myanmar (Burma)” have to be resolved into two alternative names, then the tables can be associated with the country objects.

```
T[country->>N] :- tabtype(TabType),
    T:TabType[int_country_name->>N], not substr("(",N).
T[country->>{N1,N2}] :-
    T:TabType[int_country_name->>N], tabtype(TabType),
    pmatch(N,"/([^(]*) \((.*)\)/", ["$1","$2"], [N1,N2]).
?- sys.strat.doIt.
C[TabType->T] :-
    C:country[name@(gs)->>N], T:TabType[country->>N], tabtype(TabType).
```

alternative country names known from table headers are propagated to countries.

```
?- sys.strat.doIt.
C[name@(gs)->>N] :-
    C:country[TabType->T[country->>N]], tabtype(TabType).
?- sys.strat.doIt.
```

Administrative Divisions. For the administrative divisions, the the area, population and capital (not present in all tables) columns are identified. Note that pop_col is multivalued.

```
less(0,1).
less(0,2).
less(1,2).
less(0,3).
less(1,3).
less(2,3).
less(0,4).
less(1,4).
less(2,4).
less(3,4).
?- sys.strat.doIt.
T[name_col->0;area_col->>Col] :- T:admdivtab[header@(Row,Col)->>Strg],
   substr("area",Strg), not substr("total",Strg).
   %% Netherlands: total and land area
T[pop_col->>Col] :-
   T:admdivtab[header@(Row,Col)->>Strg], substr("pop",Strg).
T[cap_col->Col] :-
   T:admdivtab[header@(Row,Col)->>Strg], substr("cap",Strg).
?- sys.strat.doIt.
```

Due to multispan's, the maximum value for area and capital column has to be taken (cf. Australia)

```
T[area_col->Col] :- Col = max{C [T]; T[area_col->>C]}.
T[cap_col->Col] :- Col = max{C [T]; T[cap_col->>C]}.
?- sys.strat.doIt.
```

for some of the countries, administrative divisions are organized in a two-level hierarchy, e.g., France and Italy. They can be detected by the existence of boldfaced table entries (table entries s.t. entry.b@(0) is defined.

```
T.two_level[] :-
   T:admdivtab[area_col->AS],
   element(T,Z,AS)[contents->I;type->td], I[b@(0)->P].
?- sys.strat.doIt.
```

for some countries, the population of administrative divisions is given at different years. This can be detected by the number of answers to C.adm_div_tab.pop_col $\twoheadrightarrow$X. In this case, for every population column, the years are derived and associates with the individual population columns.

```
T[morepops->>Yint;morepops@(Yint)->PS]   :-
   N=count{PS [T]; T[pop_col->>PS]}, N > 1,
   T:admdivtab[header_row->>HR;pop_col->>PS],
   element(T,HR,PS)[contents->I;type->th],
   pmatch(I, "/([0-9]{4})/", "$1", Y), string2integer(Y,Yint).
?- sys.strat.doIt.
```

For tables giving the population at several years, the most recent population value is relevant for MONDIAL. In every case, the *single-valued* method C.adm_div_tab.pop_col ⇒integer contains the relevant population column.

```
T[pop_col->TPS]   :-
   T:admdivtab..morepops[], TY=max{Y [T]; T[morepops->>Y]},
   T[morepops@(TY)->TPS].
T[pop_col->PS]   :-
   T:admdivtab, not T..morepops[], T[pop_col->>PS].
?- sys.strat.doIt.
```

After analyzing the table structure, the administrative divisions can be extracted. Non-two level administrative divisions:

```
C[adm_divs ->> prov(C,DRow):prov
       [country->C;name_str->PN2;population->P2;area->A2]] :-
   C:country[admdivtab ->
       T[name_col->NS;area_col->AS;pop_col->PS]],
   not T.two_level[],
   element(T,DRow,NS)[contents->PN;type->td],
   not substr("Total", PN), not substr("total", PN),
   not substr("*",PN),  % Russia: parts of other counties
   element(T,DRow,AS)[contents->A;type->td],
   perl(replace.kommas, A, A2),
   element(T,DRow,PS)[contents->P;type->td],
   pmatch(P,"/[0-9]/","$1",_),
   perl(replace.kommas, P, P2),
   perl(format.province, PN, PN2).
```

evaluating capital columns in non-two level tables.

```
C[adm_divs ->> prov(C,DRow)[capital_str->CN2]] :-
   C:country[admdivtab -> T[cap_col->CS]],
   prov(C,DRow):prov,
   not T.two_level[],
   element(T,DRow,CS)[contents->CN;type->td],
   perl(format.city_name, CN, CN2).
?- sys.strat.doIt.
```

For two-level administrative divisions, only the upper level is relevant. This is represented by bold entries.

```
C[adm_divs ->> prov(C,DRow):prov
           [country->C;name_str->PN2;population->P2;area->A2]] :-
   C:country[admdivtab -> T[name_col->NS;area_col->AS;
                                pop_col->PS;two_level->_]],
```

```
   element(T,DRow,NS)[contents->NI;type->td], NI[b@(0)->PN],
   not PN="Total", not PN="total",
   not substr("*",PN),  % Russia: parts of other counties
   element(T,DRow,AS)[contents->AI;type->td], AI[b@(0)->A],
   perl(replace.kommas, A, A2),
   element(T,DRow,PS)[contents->PI;type->td], PI[b@(0)->P],
   perl(replace.kommas, P, P2),
   perl(format.province, PN, PN2).
```

Capital columns in two-level tables:

```
C[adm_divs ->> prov(C,DRow):prov[capital_str->CN2]] :-
   C:country[admdivtab ->
        T[name_col->NS;cap_col->CS;two_level->_]],
   element(T,DRow,NS)[contents->NI;type->td], NI[b@(0)->PN],
   not substr("*",PN),   % Russia: parts of other counties
   element(T,DRow,CS)[contents->CI;type->td], CI[b@(0)->CN],
   not CN = " ", not CN  = "-",
   perl(format.province, CN, CN2).
```

There are some exceptions with the names and capitals of administrative divisions:
Provinces where no capital is given are no exceptions. Some provinces have no name since they are identical with the municipalities which are their capitals. In this case the name is set to "<name> (munic.)". In Great Britain, there seems to be a wrong capital entry for the province Rhondda.

```
P[name->PN] :-
   P:prov[name_str->PN], not P.capital_str[].
P[name->PN] :-
   P:prov[name_str->PN;capital_str->CN],
   not CN = " ", not CN = "", not CN  = "-", not CN  = "..".
P[capital_name->>CN] :-
   P:prov[name_str->PN;capital_str->CN],
   not substr("Rhondda",PN),
   not CN = " ", not CN = "", not CN  = "-", not CN  = "..".
P[capital_name->>"Rhondda"] :-   %% Data Inconsistency in GS.
   P:prov[name_str->PN], substr("Rhondda",PN).
?- sys.strat.doIt.
P[name->PN2; capital_name->>PN] :-
        P:prov[name_str->PN; capital_str->CN], not P.name[],
        strcat(PN," (munic.)",PN2).
?- sys.strat.doIt.
```

References of the form "Rangoon (Yangon)" have to be resolved into two alternative names.

```
?- sys.strat.doIt.
P[capital_name->>{N1,N2}] :-  P:prov[capital_name->>N],
    pmatch(N,"/([^(]*) \((.*)\)/", ["$1","$2"], [N1,N2]).
?- sys.strat.doIt.
```

Main Cities. For the main city tables, the header row(s) and the name and population columns (population columns together with the year) are identified. Note that header_row and pop_col are again multivalued.

```
city_col_header("cit").
city_col_header("name").
city_col_header("agglomeration").
?- sys.strat.doIt.
T[city_col->>Col] :- T:maincitytab[header@(_,Col)->>Strg],
   substr(CSH,Strg), city_col_header(CSH).
T[pop_col->>Col; pop_year@(HR,Col)->Y2] :-
   T:maincitytab[header@(HR,Col)->>Strg],
   substr("pop",Strg),
   pmatch(Strg,"/19([0-9][0-9])/","$1",Y), string2integer(Y,Y2).
```

Exception: Brazil has a slightly different layout.

```
T[pop_col->>Col; pop_year@(HR,Col)->Y2] :-
   T:maincitytab[header@(HR,Col)->>Strg],
   substr("municipality",Strg),
   pmatch(Strg,"/19([0-9][0-9])/","$1",Y), string2integer(Y,Y2).
```

Several main city tables contain a column which gives the province where the city belongs to:

```
T[prov_col->Col] :-
   T:maincitytab[header@(_,Col)->>Strg], substr("prov",Strg).
?- sys.strat.doIt.
```

There are also some main city tables which have a twocolumn layout.

```
T.twocolumn[]   :-
   N=count{CS [T]; T:maincitytab[city_col->>CS]}, N > 1.
?- sys.strat.doIt.
```

After analyzing the table structure, the main city information can be extracted. In case of tables with several header rows, one of them is chosen – they are all the same. Tables without province columns:

```
C[main_cities ->> cty(C,DZ):city
               [country->C;namestr->CN;population@(Y)->P2]] :-
   C:country[maincitytab -> T[city_col->>CS;pop_col->>PS]],
   not T.prov_col[],
   element(T,DZ,CS)[contents->CN;type->td],
   element(T,DZ,PS)[contents->P;type->td],
   perl(replace.kommas, P, P2),
   H=max{HR [T,DZ,PS,P];
            T[header_row->>HR;city_col->>CS;pop_col->>PS],
            element(T,DZ,CS)[contents->CN],
            element(T,DZ,PS)[contents->P], HR < DZ},
   T[pop_year@(H,PS)->Y],
   not substr("Moskovskaya.obl.", CN), %% Russia. Two Kaliningrads
   not T.twocolumn[].
```

Tables with a province column:

```
C[main_cities ->>
       cty(C,DZ):city[country->C;namestr->CN;
                      population@(Y)->P2;prov_name->Pr]] :-
   C:country[maincitytab ->
             T[city_col->>CS;pop_col->>PS;prov_col->PrS]],
   element(T,DZ,CS)[contents->CN;type->td],
   element(T,DZ,PS)[contents->P;type->td], perl(replace.kommas, P, P2),
   element(T,DZ,PrS)[contents->Pr;type->td], not Pr = "-",
   H=max{HR [T,DZ,PS,P];
            T[header_row->>HR;city_col->>CS;pop_col->>PS],
            element(T,DZ,CS)[contents->CN],
            element(T,DZ,PS)[contents->P], HR < DZ},
   T[pop_year@(H,PS)->Y],
   strcat(CN,Pr,CPS), not T.twocolumn[].
```

Twocolumn tables: important to check if the last element in the second column actually contains data or if it is empty (by strlen)

```
C[main_cities ->>
     cty(C,DZ,CS):city[country->C;namestr->CN;population@(Y)->P2]] :-
   C:country[maincitytab -> T[header_row->>HR;city_col->>CS]],
   element(T,DZ,CS)[contents->CN;type->td], P2 > 0,
   element(T,DZ,PS)[contents->P;type->td], PS = CS + 1,
   perl(replace.kommas, P, P2),
   T[pop_year@(HR,PS)->Y], T.twocolumn[].
?- sys.strat.doIt.
```

Names of the form “Vienna (Wien)” have to be resolved into two alternative names. Thus, city.name@(gs)⇒string is multivalued.

```
C[name@(gs)->>Name2] :-
        C:city[namestr->CNS],
        pmatch(CNS,"/\A([^(,]*)/","$1",Name),
        perl(format.city_name, Name, Name2).
C[name@(gs)->>Name2] :-
        C:city[namestr->CNS],
        pmatch(CNS,"/\(([A-Z][a-zA-Z- ]*)\)/","$1",Name),
        not _:country[name@(gs)->>Name],
        not substr("(Guadalajara)",CNS),
        perl(format.city_name, Name, Name2).
?- sys.strat.doIt.
```

The countries’ capitals are linked to city objects (where these already exist) by country[capital@(gs) ⇒city].

```
C[capital->Capcty] :-
        C:country[capital_name->>CN],
        Capcty:city[country->C;name@(gs)->>CN].
?- sys.strat.doIt.
```

For countries where the capital is not a main city (i.e., those where no main city data exist), the capital objects have to be created and country[capital⇒city] is defined.

```
C[capital->cty(C,CN2):city[country->C;name@(gs)->>CN]],
C[main_cities->>cty(C,CN2)] :-
        C:country[capital_name->>CN], not C.capital[], not CN = "none",
       perl(replace.spaces,CN,CN2).
?- sys.strat.doIt.
```

Linking Cities to Provinces. Simple cases:

```
Cty[province->>Prov] :- Cty:city[country->C;prov_name->PN],
                        Prov:prov[country->C;name->PN].
?-sys.strat.doIt.
```

In several cases, the province name given with the city is only one of to possible (e.g. "Kärnten (Carinthia)"):

```
Cty[country->C;province->>Prov] :-
     Cty:city[country->C;prov_name->PN], not Cty..province[],
     Prov:prov[country->C;name->N],
     strcat("/\(",PN,PN1),
     strcat(PN1,"\)/",PN2),
     pmatch(N,PN2,"$1",X).
?-sys.strat.doIt.
```

In case of abbreviations, the matching is done by the first five letters:

```
Cty[country->C;province->>Prov] :-
     Cty:city[country->C;prov_name->PN], not Cty..province[],
     Prov:prov[country->C;name->N],
     pmatch(PN,"/.{5}/","$1",_),
     pmatch(PN,"/\A(.*).\Z/","$1/",PN1),
     strcat("/\A",PN1,PN2),
     pmatch(N,PN2,"$1",_).
?-sys.strat.doIt.
```

For the US, the postal codes have to be translated to the states' names, e.g., "New York, NY"):

```
C[province->>Pr] :-
     C:city[namestr->CNS].country[name@(gs)->>"USA"],
     pmatch(CNS,"/, ([A-Z]{2})/","$1",Code),
     us_postal_code(Code,P),
     Pr[name->P].country[name@(gs)->>"USA"].
```

Note that Canadian provinces remain unresolved since non-official abbrevs are used.

```
?-sys.strat.doIt.
```

Linking Capitals of Provinces to Cities. The provinces' capitals are linked to city objects (where these already exist) by province[capital⇒city]. If the capital city is not yet associated with a province, set city[province⇒province].

```
P[capital->Capcty[province->>P]] :-
    P:prov[country->C;capital_name->>CN],
    Capcty:city[country->C;name@(gs)->>CN], not Capcty..province[].
P[capital->Capcty] :-
    P:prov[country->C;name->PN;capital_name->>CN],
    Capcty:city[country->C;name@(gs)->>CN;province->>P[name->PN]].
?- sys.strat.doIt.
```

For provinces where the capital is not a main city (i.e., those where no main city data exist), the capital objects have to be created.

```
P[capital->cty(C,CN2):city
   [country->C;name@(gs)->>CN;province->>P]],
C[main_cities->>cty(C,CN2)] :-
   P:prov[country->C;capital_name->>CN], not P.capital[],
   perl(replace.spaces,CN,CN2).
?-sys.strat.doIt.
```

Output. Using the instance output format, the resulting atoms are selected for output to mondial-gs-facts.flp.

```
save[init->"‘rm ../Mondial-facts/mondial-gs-facts.flp‘; @out=@in;"].
?- sys.strat.doIt.
?- perl(save.init,[""],[""]).
?- sys.prn.style@("instance").  % set output mode
?- sys[output->sys.open@("../Mondial-facts/mondial-gs-facts.flp")].
?- sys.answerChannel.setStream@(output)[].
```

The following output queries also define the schema of the GlobalStatistics part of the database

```
 ?- C:continent[name->N].
 ?- C:country[name@(gs)->>N;continent@(gs)->>CT].
 ?- C:country[capital->Cap;population@(gs)->Pop].
 ?- C:country[main_cities->>City:city[country->C;name@(gs)->>CN]].
 ?- C:city[population@(Y)->P].
 ?- C:country[adm_divs->>Pr:prov[country->C;name->PN;population->P]].
 ?- C:city[province->>Pr].
 ?- Pr:prov[capital->C].
 ?- Pr:prov[area->AR].
?- sys.answerChannel.resetStream[].
?- sys.remove@("output").
```

3.4 Qiblih: Geographical Coordinates

```
?- sys.theOM.debugOn.      % trace Web access
?- sys.theOM.eqTraceOn.      % trace derived equalities
?- sys.prn.style@("bound").  % set output mode
```

Define several entry points for the data source. The Qiblih data are distrubuted over three pages: USA, Canada, and other countries.

```
qiblih[home -> "http://www.bcca.org/misc/qiblih/";
       src -> "file:/home/dbis/.public_html/.mir/qiblih-geo/"].
qiblih[page@(us)->"latlong_us.html";
       page@(cdn)->"latlong_ca.html";
       page@(other)->"latlong_oc.html"].
```

Several auxiliary formatting methods are defined, using FLORID's perl interface. Mainly, expressions

```
qiblih[cityname->"$out[0]=$in[0];
             $out[0]=~s/ (\(.*\))\Z/ /g;
             $out[0]=~s/ AFB\Z//g;
             $out[0]=~s/ AP\Z//g;
             $out[0]=~s/ Co\Z//g;"].
qiblih[to_lower->"$out[0]=$in[0];
             $out[0]=~s/\A(.)([A-Z]*)/$1\L$2/g;
             $out[0]=~s/ (.)([A-Z]*)/ $1\L$2/g"].
```

Access all three pages.

```
U:url.get :- strcat(qiblih.src,qiblih.page@(X),U).
```

Every page is structured into groups, divided by the country or state names, respectively, which are enclosed in <A> tags.

```
U[group@(N2)->L]  :- strcat(qiblih.src,qiblih.page@(X),U),
   pmatch(U.get,"/<A NAME=[^>]*>([^<]*)<.A>([^<]*)/g",
          ["$1","$2"],[N,L]),
   perl(qiblih.to_lower,N,N2).
```

Inside the groups, every line contains a city and its longitude and latitude, low-level formatted with whitespace as <cityname><deg_lat min_lat>{N|S} <deg_long min_long>{W|E}. In a first step, longitude and latitude are extracted as strings.

```
U[group@(N)->>cty(N,C):
   city[name@(qiblih)->C2;country@(qiblih)->"United States";
             province@(qiblih)->N;lat->Lat;long->Long]] :-
   U[group@(N)->L], strcat(qiblih.src,qiblih.page@(us),U),
   pmatch(L,
     "/\n\s*([^0-9]*[a-zA-Z])\s*([0-9][^NS]*[NS])\s*([^WE]*[WE])/g",
     ["$1","$2","$3"], [C,Lat,Long]),
   perl(qiblih.cityname,C,C2).
U[group@(N)->>cty(N,C):
   city[name@(qiblih)->C2;country@(qiblih)->"Canada";
             province@(qiblih)->N;lat->Lat;long->Long]] :-
   U[group@(N)->L], strcat(qiblih.src,qiblih.page@(cdn),U),
   pmatch(L,
     "/\n\s*([^0-9]*[a-zA-Z])\s*([0-9][^NS]*[NS])\s*([^WE]*[WE])/g",
     ["$1","$2","$3"], [C,Lat,Long]),
   perl(qiblih.cityname,C,C2).
```

```
U[group@(N)->>cty(N,C):
   city[name@(qiblih)->C2;country@(qiblih)->N;
        lat->Lat;long->Long]] :-
   U[group@(N)->L], strcat(qiblih.src,qiblih.page@(other),U),
   pmatch(L,
     "/\n\s*([^0-9]*[a-zA-Z])\s*([0-9][^NS]*[NS])\s*([^WE]*[WE])/g",
     ["$1","$2","$3"], [C,Lat,Long]),
   perl(qiblih.cityname,C,C2).
```

North/East and South/West are translated into positive/negative values and the longitude/latitude strings are decomposed and transformed into numerical values. Note that an error in the data for Addis Ababa has to be corrected on-the-fly.

```
val("N") = 1.
val("S") = -1.
val("E") = 1.
val("W") = -1.
C[latitude->B] :-
   C:city[lat->Lat],
   pmatch(Lat,"/([0-9]*)&#176; ([0-9]*)..([NS])/g",
          ["$1","$2","$3"],[G,M,N]),
   string2float(G,G2), string2float(M,M2), val(N) = V,
   not C[name@(qiblih)->"Addis Ababa"],
   B = (G2 + M2 / 60) * V.
C[latitude->B] :-  %% Data for Addis Ababa is incorrect (90.2)
   C:city[lat->Lat],
   pmatch(Lat,"/([0-9]*)&#176; ([0-9]*)..([NS])/g",
          ["$1","$2","$3"],[G,M,N]),
   string2float(G,G2), string2float(M,M2), val(N) = V,
   C[name@(qiblih)->"Addis Ababa"], B = #9.
C[longitude->L] :-
   C:city[long->Long],
   pmatch(Long,"/([0-9]*)&#176; ([0-9]*)..([WE])/g",
          ["$1","$2","$3"],[G,M,E]),
   string2float(G,G2), string2float(M,M2), val(E) = V,
   L = (G2 + M2 / 60) * V.
?- sys.strat.doIt.
```

There are several cities for which two values are given, e.g., Baltimore AP vs. Baltimore CO (the AP and CO strings are already stripped when formatting. Here one of them is dropped by the following rules:

```
C1:useless :-
   C1:city[name@(qiblih)->N; country@(qiblih)->C;
           province@(qiblih)->S1;longitude->L1],
   C2:city[name@(qiblih)->N; country@(qiblih)->C;
           province@(qiblih)->S1;longitude->L2],
   L1 < L2.
C1:useless :-
   C1:city[name@(qiblih)->N; country@(qiblih)->C;
```

```
          province@(qiblih)->S1;longitude->L1;latitude->B1],
   C2:city[name@(qiblih)->N; country@(qiblih)->C;
          province@(qiblih)->S1;longitude->L2;latitude->B2],
   L1 = L2, B1 < B2.
?- sys.strat.doIt.
C:qcity :- C:city, not C:useless.
```

Output. Using the instance output format, the resulting atoms are selected for output to mondial-qiblih-facts.flp.

```
save[init->"‘rm ../Mondial-facts/mondial-qiblih-facts.flp‘; @out=@in;"].
?- sys.strat.doIt.
?- perl(save.init,[""],[""]).
?- sys.prn.style@("instance").  % set output mode
?- sys[output->sys.open@("../Mondial-facts/mondial-qiblih-facts.flp")].
?- sys.answerChannel.setStream@(output)[].
```

The following output queries also define the schema of the Qiblih part of the database

```
 ?- Cty:qcity[name@(qiblih)->N;country@(qiblih)->C;
            longitude->L;latitude->Lat].
 ?- C:qcity[province@(qiblih)->S].
?- sys.answerChannel.resetStream[].
?- sys.remove@("output").
```

3.5 The Original Terra Database

```
?- sys.theOM.debugOn.      % trace Web access
?- sys.theOM.eqTraceOn.      % trace derived equalities
?- sys.prn.style@("bound").  % set output mode
```

Define the entry point for the data source:

```
terra[home -> "file:/home/dbis/TERRA/datenbasis/"].
terra[src -> terra.home].
```

Every relation of Terra is made an object, relation[url@(terra) ⇒file] gives the files where the data can be found. For geo-relations, additionally the file of the geo-data is given.

```
stadt:relation[name->"stadt"].
land:relation[name->"land"].
landesteil:relation[name->"landesteil"].
umfasst:relation[name->"umfasst"].
liegt_an:relation[name->"liegt_an"].
geht_ueber:relation[name->"geht_ueber"].
lake:relation[name->"see"].
river:relation[name->"fluss"].
sea:relation[name->"meer"].
mountain:relation[name->"berg"].
```

```
desert:relation[name->"wueste"].
island:relation[name->"insel"].
water::geo_obj.
lake::water.
river::water.
sea::water.
mountain::geo_obj.
desert::geo_obj.
island::geo_obj.
R[url@(terra) -> U] :- R:relation,
   strcat(terra.src, R.name, U1), strcat(U1, ".sql", U).
R[url@(geo,terra) -> U] :-  R:relation, R::geo_obj,
   strcat(terra.src, "geo_", U1), strcat(U1, R.name, U2),
   strcat(U2, ".sql", U).
?- sys.strat.doIt.
```

All relations are accessed.

```
U:url.get :- R:relation[url@(terra)->U].
U:url.get :- R:relation[url@(geo,terra)->U].
?- sys.strat.doIt.
```

In the TERRA database, underscores and german names are used. Underscores are deleted, and the english names are used in MONDIAL.

```
delete[underscore->"$out[0]=$in[0];
             $out[0]=~s/_/ /g;"].
translate("Europa","Europe").
translate("Afrika","Africa").
translate("Asien","Asia").
translate("Amerika","America").
translate("Australien","Australia/Oceania").
translate("Soedermanslands laen","Södermanland").
translate("Oestergoetlands laen","Östergötland").
translate("Joenkoepings laen","Jönköping").
translate("Kronobergs laen","Kronoberg").
translate("Malmoehus laen","Malmöhus").
translate("Goeteborgs och Bohus laen","Göteborg och Bohus").
translate("Aelvsborgs laen","Älvsborg").
translate("Oerebro laen","Örebro").
translate("Vaestmanlands laen","Västmanland").
translate("Gaevleborgs laen","Gävleborg").
translate("Vaesternorrlands laen","Västernorrland").
translate("Jaemtlands laen","Jämtland").
translate("Vaesterbottens laen","Västerbotten").
translate("Norrbottens laen","Norrbotten").
translate("Niederoesterr.","Lower Austria").
translate("Arabisches Meer","Arabian Sea").
translate("Nordsee","North Sea").
translate("Ostsee","Baltic Sea").
translate("Europaeisches Nordmeer","Norwegian Sea").
```

```
translate("Japanisches Meer","Sea of Japan").
translate("Schwarzes Meer","Black Sea").
translate("Nordpolarmeer","Arctic Ocean").
translate("Asowsches Meer","Sea of Azov").
translate("Karibisches Meer","Caribbean Sea").
translate("Irische See","Irish Sea").
translate("Suedchinesisches Meer","South China Sea").
translate("Ostchinesisches Meer","East China Sea").
translate("Sundasee","Sunda Sea").
translate("Pazifischer Ozean","Pacific Ocean").
translate("Atlantischer Ozean","Atlantic Ocean").
translate("Golf von Mexiko","Gulf of Mexico").
translate("Persischer Golf","Persian Gulf").
translate("Indischer Ozean","Indian Ocean").
translate("Rotes Meer","Red Sea").
translate("Mittelmeer","Mediterranean Sea").
translate("Golf von Aden","Gulf of Aden").
translate("Gelbes Meer","Yellow Sea").
translate("Weisser Drin","White Drin").
translate("Schwarzer Drin","Black Drin").
translate("Schwarzer Irtysch","Black Irtysch").
translate("Noerdliche Dwina","Northern Dwina").
translate("Westliche Dwina","Western Dwina").
translate("Gelber Fluss","Hwangho").
translate("Themse","Thames").
translate("Mississippi River","Mississippi").
translate("Missouri River","Missouri").
translate("Arkansas River","Arkansas").
translate("Colorado River","Colorado").
translate("Nil","Nile").
translate("Blauer Nil","Blue Nile").
translate("Weisser Nil","White Nile").
translate("Kunene","Cunene").
translate("Kasai","Casai").
translate("Kwilu","Cuilo").
translate("Kuango","Cuango").
translate("Kuanza","Cuanza").
translate("Rowuma","Ruvuma").
translate("Sambesi","Zambezi").
translate("Shire","Chire").
translate("Newa","Neva").
translate("Wolga","Volga").
translate("Prespasee","Lake Prespa").
translate("Skutarisee","Lake Skutari").
translate("Ohridsee","Lake Ohrid").
translate("Vaenersee","Vaenern").
translate("Vaettersee","Vaettern").
translate("Maelarsee","Maelaren").
translate("Mjoesensee","Mjoesen").
translate("Ladogasee","Ozero Ladoga").
```

```
translate("Baikalsee","Ozero Baikal").
translate("Balchaschsee","Ozero Balchash").
translate("Onegasee","Ozero Onega").
translate("Taimyrsee","Ozero Taimyr").
translate("Chankasee","Ozero Chanka").
translate("Pskowsee","Ozero Pskovskoje").
translate("Tschanysee","Ozero Tschany").
translate("Kallasee","Lake Kalla").
translate("Inarisee","Inari").
translate("Saimasee","Lake Saima").
translate("Naesisee","Lake Naesi").
translate("Orisee","Lake Ori").
translate("Aralsee","Ozero Aral").
translate("Titicaca See","Lake Titicaca").
translate("Nicaragua See","Lake Nicaragua").
translate("Maracaibo See","Lake Maracaibo").
translate("Bodensee","Bodensee").
translate("Genfer See","Lac Leman").
translate("Comer See","Lago di Como").
translate("Mackay See","Lake Mackay").
translate("Gardasee","Lago di Garda ").
translate("Amadeussee","Lake Amadeus").
translate("Eyre See","Lake Eyre").
translate("Disappointment-See","Disappointment-Lake").
translate("Torrensee","Lake Torren").
translate("Kaspisches Meer","Caspian Sea").
translate("Urumiyeh See","Lake Urumiyeh").
translate("Namak See","Daryacheh ye Namak").
translate("Hamune Jazmurian See","Hamun e Jaz Murian").
translate("Tschadsee","Tschadsee").
translate("Nassersee","Lake Nasser").
translate("Voltastausee","Lake Volta").
translate("Tanasee","Lake Tana").
translate("Assalesee","Lake Assale").
translate("Kainji-Stausee","Lake Kainji").
translate("M Bakau-Stausee","Barrage de Mbakaou").
translate("Abayasee","Lake Abaya").
translate("Stefaniesee","Lake Stephanie").
translate("Suatsee","Lake Suat").
translate("Rudolfsee","Lake Rudolph").
translate("Albertsee","Lake Albert").
translate("Kiogasee","Lake Kioga").
translate("Victoriasee","Lake Victoria").
translate("Eduardsee","Lake Eduard").
translate("Kiwusee","Lake Kivu").
translate("Tumbasee","Lake Tumba").
translate("Leopold2-See","Lake Leopold-II").
translate("Tanganjikasee","Lake Tanganjika").
translate("Rukwasee","Lake Rukwa").
translate("Natronsee","Lake Natron").
```

```
translate("Ejassisee","Lake Eyasi").
translate("Mwerusee","Lake Mweru").
translate("Bangweolosee","Lake Bangweulu").
translate("Njassasee","Lake Nyasa").
translate("Chilwasee","Lake Chilwa").
translate("Cabora-Bassa-See","Lake Cabora-Bassa").
translate("Karibasee","Lake Kariba").
translate("Ngamisee","Lake Ngami").
translate("Etoschapfanne","Lake Etoscha").
translate("Makarikarisalzpfanne","Lake Makarikari").
translate("Kilimanscharo","Kilimanjaro").
translate("Djebel Marra","Jabal Marra").
translate("Cathkin-Peak","Cathkin Peak").
translate("Cathkin-Peak","Cathkin Peak").
translate("Pik Kommunismus","Pik Kommunizma").
translate("Pik Lenin","Pik Lenina").
translate("Pik Moskwa","Pik Moskva").
translate("Grosse Victoriawueste","Grand Victoria Desert").
translate("Gibsonwueste","Gibson Desert").
translate("Tanamiwueste","Tanami").
translate("Simpsonwueste","Simpson Desert").
translate("Grosse Sandwueste","Grand Sand Desert").
translate("Westl. Grosser Erg","Grand Erg Ouest").
translate("Oestl. Grosser Erg","Grand Erg Est").
translate("Libysche Wueste","Libyan Desert").
translate("Arabische Wueste","Arabian Desert").
translate("Nubische Wueste","Nubian Desert").
translate("Grosser Erg von Bilma","Erg de Bilma").
translate("Hamada des Draa","Hamada du Draa").
translate("Nord-Caicos","North Caicos").
translate("Ost-Caicos","East Caicos").

?- sys.strat.doIt.
```

The Terra Source Files consist of SQL `INSERT INTO <relation> VALUES <value-list>` statements. Thus, they can be easily matched by regular expressions.

```
land(N,L):terra_country[name@(terra)->N2; code@(terra)->L;
       population@(terra)->P2; area@(terra)->A2; capital@(terra)->Cap2] :-
  pmatch(land.url@(terra).get,
    "/VALUES \('([^']*)',\s*'([^']*)',([^,]*),[^,]*,([^,]*),[^,]*,\s*'([^']*)',/g",
    ["$1","$2","$3","$4","$5"], [N,L,P,A,Cap]),
  perl(delete.underscore,N,N2), string2float(P,P2), string2float(A,A2),
  perl(delete.underscore,Cap,Cap2).
?- sys.strat.doIt.
L[encompassed@(CT2)->P2] :-
  L:terra_country[code@(terra)->Code],
  pmatch(umfasst.url@(terra).get,
    "/VALUES \('([^']*)',\s*'([^']*)',\s*([0-9.]*)\)/g",
    ["$1","$2","$3"], [Code,CT,P]),
```

```
  string2float(P,P2), translate(CT,CT2).
?- sys.strat.doIt.
lteil(AB,C):pre_terra_prov[name_str@(terra)->N; country@(terra)->C;
      abbrev@(terra)->AB; capital@(terra)->Cap2] :-
  pmatch(landesteil.url@(terra).get,
    "/VALUES \('([^']*)',\s*'([^']*)',\s*'([^']*)',\s*[^,]*,[^,]*,\s*'([^']*)'\s*\)/g",
    ["$1","$2","$3","$4"], [N,AB,C,Cap]),
  perl(delete.underscore,Cap,Cap2).
L[name@(terra)->N2] :-
  L:pre_terra_prov[name_str@(terra)->N],
  not translate(N,_), perl(delete.underscore,N,N2).
L[name@(terra)->N2] :-
  L:pre_terra_prov[name_str@(terra)->N],
  translate(N,N2).
?- sys.strat.doIt.
lteil(AB,C)[pop@(terra)->E2] :-
   pmatch(landesteil.url@(terra).get,
     "/VALUES \('[^']*',\s*'([^']*)',\s*'([^']*)',\s*([^,]*)\s*/g",
     ["$1","$2","$3"], [AB,C,E]),
   string2float(E,E2).
?- sys.strat.doIt.
L[number_of_provs->A] :-
   L:terra_country[code@(terra)->Code],
   A=count{P [Code]; P:pre_terra_prov[country@(terra)->Code]}.
?- sys.strat.doIt.
L:terra_prov :- L:pre_terra_prov[country@(terra)->Code],
       C:terra_country[code@(terra)->Code;number_of_provs->A], A > 1.
?- sys.strat.doIt.
stadt(N,C,P):terra_city[name@(terra)->N2; country@(terra)->C;
                     prov@(terra)->>P;
   longitude@(terra)->Long2;latitude@(terra)->Lat2] :-
   pmatch(stadt.url@(terra).get,
     "/VALUES \('([^']*)',\s*'([^']*)',\s*'([^']*)',\s*[^,]*,\s*([-0-9.]*),\s*([-0-9.]*)\s*\)
     ["$1","$2","$3","$4","$5"], [N,C,P,Lat,Long]),
   perl(delete.underscore,N,N2),
   string2float(Long,Long2), string2float(Lat,Lat2).
stadt(N,C,P)[pop@(terra)->E2] :-
   pmatch(stadt.url@(terra).get,
     "/VALUES \('([^']*)',\s*'([^']*)',\s*'([^']*)',\s*([^,]*)\s*/g",
     ["$1","$2","$3","$4"], [N,C,P,E]),
   perl(delete.underscore,N,N2),string2float(E,E2).
?- sys.strat.doIt.
mount(N):mountain[name@(terra)->N2; height->H2;
             longitude->Long2; latitude->Lat2] :-
    pmatch(mountain.url@(terra).get,
      "/\('([^']*)',\s*[^,]*,\s*([^,]*),\s*[^,]*,\s*([-0-9.]*),\s*([-0-9.]*)\s*\)/g",
      ["$1","$2","$3","$4"], [N,H,Long,Lat]),
    perl(delete.underscore,N,N2),
    string2float(H,H2),
    string2float(Long,Long2), string2float(Lat,Lat2).
```

```
?- sys.strat.doIt.
desert(N):desert[name@(terra)->N2] :-
   pmatch(desert.url@(terra).get,
     "/\('([^']*)',/g", ["$1"], [N]),
   perl(delete.underscore,N,N2).
desert(N)[area->A2] :-
   pmatch(desert.url@(terra).get,
     "/\('([^']*)',\s*([^,]*)/g", ["$1","$2"], [N,A]),
   string2float(A,A2).
?- sys.strat.doIt.
island(N):island[name@(terra)->N2] :-
   pmatch(island.url@(terra).get,
      "/\('([^']*)',/g", ["$1"], [N]),
   perl(delete.underscore,N,N2).
island(N)[islands->I2] :-
   pmatch(island.url@(terra).get,
      "/\('([^']*)',\s*'([^']*)',/g", ["$1","$2",], [N,I]),
   perl(delete.underscore,I,I2).
island(N)[area->A2] :-
   pmatch(island.url@(terra).get,
     "/\('([^']*)',[^,]*,\s*([^,]*)/g", ["$1","$2"], [N,A]),
   string2float(A,A2).
island(N)[longitude->Long2; latitude->Lat2] :-
   pmatch(island.url@(terra).get,
     "/\('([^']*)',[^,]*,[^,]*,\s*([^,]*),\s*([^,]*)/g",
     ["$1","$2","$3"], [N,Long,Lat]),
   string2float(Long,Long2), string2float(Lat,Lat2).
?- sys.strat.doIt.
lake(N):lake[name@(terra)->N2] :-
   pmatch(lake.url@(terra).get,"/\('([^']*)',/g",["$1"],[N]),
   perl(delete.underscore,N,N2).
lake(N)[area->A2] :-
   pmatch(lake.url@(terra).get,
     "/\('([^']*)',\s*[^,]*,\s*([^)]*)\s*\)/g", ["$1","$2"],[N,A]),
   string2float(A,A2).
?- sys.strat.doIt.
sea(N):sea[name@(terra)->N2] :-
   pmatch(sea.url@(terra).get,"/\('([^']*)',/g", ["$1"], [N]),
   perl(delete.underscore,N,N2).
sea(N)[depth->D2] :-
   pmatch(sea.url@(terra).get,
     "/\('([^']*)',\s*([^)]*)/g", ["$1","$2"], [N,D]),
   string2float(D,D2).
?- sys.strat.doIt.
river(N):river[name@(terra)->N2] :-
   pmatch(river.url@(terra).get,"/\('([^']*)',/g", ["$1"], [N]),
   perl(delete.underscore,N,N2).
river(N)[length->L2] :-
   pmatch(river.url@(terra).get,
     "/\('([^']*)',\s*[^,]*,\s*[^,]*,\s*[^,]*,\s*([^,]*)/g",
```

```
     ["$1","$2"], [N,L]),
   string2float(L,L2).
?- sys.strat.doIt.
```

For rivers, also the river/lake/sea they flow into is given.

```
to_pattern[river->"/\('([^']*)',\s*'([^']*)'/g";
   lake->"/\('([^']*)',\s*[^,]*,\s*'([^']*)'/g";
   sea->"/\('([^']*)',\s*[^,]*,\s*[^,]*,\s*'([^']*)'/g"].

river(N)[to@(Class)->O] :-
   to_pattern[Class->Pattern],
   pmatch(river.url@(terra).get, Pattern, ["$1","$2"], [N,To]),
   perl(delete.underscore,To,To2),
   O:Class[name@(terra)->To2].
?- sys.strat.doIt.
```

for all classes of geographic objects, the geo_<type> relations give the country and province(s).

```
O[place->>pair(C,P)] :-
   Geo::geo_obj, O:Geo[name@(terra)->N2],
   pmatch(Geo.url@(geo,terra).get,
     "/\('([^']*)',\s*'([^']*)',\s*'([^']*)'\s*\)/g",
     ["$1","$2","$3"], [C,P,N]), not C = "CS",
   perl(delete.underscore,N,N2).
?- sys.strat.doIt.
```

The relation liegt_an(city,country,province,river,lake,sea) contains the cities which are located at a river/lake/sea.

```
loc_pattern[river->
     "/\('([^']*)',\s*'([^']*)',\s*'([^']*)',\s*'([^']*)'/g";
   lake->
     "/\('([^']*)',\s*'([^']*)',\s*'([^']*)',\s*[^,]*,\s*'([^']*)'/g";
   sea->
    "/\('([^']*)',\s*'([^']*)',\s*'([^']*)',\s*[^,]*,\s*[^,]*,\s*'([^']*)'/g"].
?- sys.strat.doIt.
S[located_at@(Class)->>O]:-
   loc_pattern[Class->Pattern],
   pmatch(liegt_an.url@(terra).get, Pattern,
          ["$1","$2","$3","$4"], [N,P,C,X]),
   perl(delete.underscore,N,N2),
   perl(delete.underscore,X,X2),
   S:terra_city[name@(terra)->N2;country@(terra)->C;prov@(terra)->>P],
   O:Class[name@(terra)->X2].
?- sys.strat.doIt.
```

The relation geht_ueber contains pairs of seas.

```
 ?- sys.echo@("*** merges_with ***").
bordering_seas(M1,M2) :-
      pmatch(geht_ueber.url@(terra).get,
```

```
      "/VALUES \('([^']*)',\s*'([^']*)'\)/g",
      ["$1","$2"], [N1,N2]),
      perl(delete.underscore,N1,N12),
      perl(delete.underscore,N2,N22),
      M1:sea[name@(terra)->N12],
      M2:sea[name@(terra)->N22].
?- sys.strat.doIt.
```

The geo objects' names are translated into English.

```
O[name->NEngl] :-
     O:geo_obj[name@(terra)->N],
     translate(N,NEngl).
O[name->N2] :-
     O:geo_obj[name@(terra)->N],
     not translate(N,_), perl(delete.underscore,N,N2).
?- sys.strat.doIt.
```

Output. Using the instance output format, the resulting atoms are selected for output to mondial-terra-facts.flp.

```
save[init->"`rm ../Mondial-facts/mondial-terra-facts.flp`; @out=@in;"].
?- sys.strat.doIt.
?- perl(save.init,[""],[""]).
?-sys.strat.doIt.
?- sys.prn.style@("instance").  % set output mode
?- sys[output->sys.open@("../Mondial-facts/mondial-terra-facts.flp")].
?- sys.answerChannel.setStream@(output)[].
 ?- L:terra_country[name@(terra)->N; code@(terra)->CC;
          area@(terra)->A; population@(terra)->P;
          capital@(terra)->Cap;encompassed@(CT)->Perc].
 ?- P:terra_prov[name@(terra)->N; country@(terra)->C;
            abbrev@(terra)->AB; capital@(terra)->Cap].
 ?- P:terra_prov[pop@(terra)->E].
 ?- S:terra_city[name@(terra)->N; country@(terra)->C; prov@(terra)->>P;
           longitude@(terra)->Long; latitude@(terra)->Lat].
 ?- S:terra_city[pop@(terra)->P].
 ?- M:mountain[name->N; height->H; longitude->Long; latitude->Lat].
 ?- D:desert[name->N].
 ?- D:desert[area->A].
 ?- I:island[name->N].
 ?- I:island[islands->N].
 ?- I:island[area->A].
 ?- I:island[longitude->Long; latitude->Lat].
 ?- R:river[name->N; to@(M)->X].
 ?- R:river[length->L].
 ?- S:sea[name->N].
 ?- S:sea[depth->D].
 ?- L:lake[name->N].
 ?- L:lake[area->A].
 ?- O[place->>pair(C,P)].
```

```
 ?- S:terra_city[located_at@(GT)->>G].
 ?- bordering_seas(M1,M2).
?- sys.answerChannel.resetStream[].
?- sys.remove@("output").
```

3.6 Countries, Names, and Car Codes

The CIA World Factbook does not provide the commonly used and well-known car codes of the countries. These are needed for integration with the TERRA Database. Additionally, the german country names proved to be very helpful for integrating the TERRA Database.

```
?- sys.theOM.debugOn. % trace Web access
?- sys.theOM.eqTraceOn. % trace derived equalities
?- sys.prn.style@("bound").  % set output mode
```

Define several entry points for the data source:

```
ext[country_codes_mirror ->
       "file:/home/dbis/.public_html/.mir/terraplus/";
    country_codes ->"flags+carcodes.html"].
```

An auxiliary formatting method is defined, using FLORID's perl interface.

```
format[country_name->"$out[0]=$in[0];
     $out[0]=~s|\Athe \s*||g;
     $out[0]=~s|\AThe \s*||g;
     $out[0]=~s|\A\s*||g;
     $out[0]=~s/\s*\Z//g;
     $out[0]=~s/([^,]*) \(South\)/South $1/g;
     $out[0]=~s/([^,]*) \(North\)/North $1/g;
     $out[0]=~s/([^,]*) \(S&uuml;d\)/Sued-$1/g;
     $out[0]=~s/([^,]*) \(Nord\)/Nord $1/g;
     $out[0]=~s/'//g;
     $out[0]=~s/Cte/Cote/g;
     $out[0]=~s/Dnemark/Dänemark/g;
     $out[0]=~s/Sdafrika/Südafrika/g;
     $out[0]=~s/\Aquatorial- guinea/Aequatorialguinea/g;
     $out[0]=~s/\Agypten/Ägypten/g;
     $out[0]=~s/So Tom/Sao Tome/g;
     $out[0]=~s/Prncipe/Principe/g;
     $out[0]=~s/sterreich/Österreich/g;
     $out[0]=~s/Rumnien/Rumänien/g;
     $out[0]=~s/Ruland/Russland/g;
     $out[0]=~s/Grobritannien/Großbritannien/g;
     $out[0]=~s/Trkei/Türkei/g;
     $out[0]=~s/Great Britain and Northern Ireland/United Kingdom/g;
     $out[0]=~s/Luxemburg/Luxembourg/g;
     $out[0]=~s/Vatican/Holy See/g;
     $out[0]=~s/United States of America/United States/g;
     $out[0]=~s/Cape Verte/Cape Verde/g;
```

```
     $out[0]=~s/Yugoslavia/Serbia and Montenegro/g;
     $out[0]=~s/Comores/Comoros/g;
     $out[0]=~s/ I\Z//g;"].
?- sys.strat.doIt.
```

The page with country names in english, german, and the local name, and the car code is structured as a list of tables for every letter. The page is parsed by the built-in HTML parser, obtaining a parse-tree of the page using a special tagfile (specifying which HTML tags are ignored or pruned).

```
?- sys.theOM.setTagFile@("mondial-codes.tags").
U:url.parse :- strcat(ext.country_codes_mirror,ext.country_codes,U).
?- sys.strat.doIt.
```

The individual tables are extracted

```
tab(N):country_codes_table[thetable->T;nr->N]  :-
      T:(U.parse.table), T = X.html@(N),
      strcat(ext.country_codes_mirror,ext.country_codes,U).
```

the 0.th column contains the car code, the first column gives the englich name, the third one gives the german name, and the forth one gives the name in the local language.

```
language(1,"english").
language(3,"german").
language(4,"local").
ctry(N,Row):country[name@(Language)->Name2;car_code_str->Code] :-
  T:country_codes_table[nr->N].thetable.table@(Row)
        [tr@(Column)->_[td@(0)->Name];tr@(0)->_[td@(0)->Code]],
  language(Column,Language), string(Name),
  perl(format.country_name,Name,Name2).
```

The car code of Finland is not FIN but SF. All other codes are correct.

```
C[car_code->Code] :-
   C:country[car_code_str->Code], not Code = "FIN".
C[car_code->"SF"] :-
   C:country[car_code_str->Code], Code = "FIN".
?- sys.strat.doIt.
```

Output. Using the instance output format, the resulting atoms are selected for output to mondial-codes-facts.flp.

```
save[init->"`rm ../Mondial-facts/mondial-codes-facts.flp`; @out=@in;"].
?- sys.strat.doIt.
?- perl(save.init,[""],[""]).
?- sys.prn.style@("instance").  % set output mode
?- sys[output->sys.open@("../Mondial-facts/mondial-codes-facts.flp")].
?- sys.answerChannel.setStream@(output)[].
```

The following output query also defines the schema of the country names and country code part of the database

```
 ?- C:country[car_code->Code; name@("english")->NE;
    name@("german")->NG; name@("local")->NL].
?- sys.answerChannel.resetStream[].
?- sys.remove@("output").
```

4 Integration

The program mondial-integra integrates the databases generated by the above *wrappers* into a common fully object-oriented schema. The resulting Object Manager (OM) is then dumped to be used by follow-up programs.

```
?- sys.theOM.debugOn. % trace Web access
?- sys.theOM.eqTraceOn. % trace derived equalities
?- sys.prn.style@("bound").     % set output mode
```

First, the data-sources provided by the wrappers are loaded:

```
?- sys.load@("../Mondial-facts/mondial-cia-facts.flp").
?- sys.load@("../Mondial-facts/mondial-gs-facts.flp").
?- sys.load@("../Mondial-facts/mondial-orgs-facts.flp").
?- sys.load@("../Mondial-facts/mondial-qiblih-facts.flp").
?- sys.load@("../Mondial-facts/mondial-codes-facts.flp").
?- sys.load@("../Mondial-facts/mondial-terra-facts.flp").
?- sys.strat.doIt.
```

The following cities are not contained in gs. They are needed as seats for organizations:

```
C[main_cities->>cty("Abidjan"):city
      [country->C; name@(gs)->>"Abidjan"]] :-
   C:country[name@(cia) -> "Cote dIvoire"].

C[main_cities->>cty("Jeddah"):city
      [country->C; name@(gs)->>"Jeddah"]]:-
   C:country[name@(cia) -> "Saudi Arabia"].
```

Cities names like "Mexico", "Mexico City" and "Ciudad de Mexico" denote the same object – same for Panama:

```
C1 = C2 :-
   C1:city[country->C; name@(gs) ->> N1],
   C2:city[country->C; name@(gs) ->> N2],
   strcat(N1," City", N2).
?- sys.strat.doIt.
C1 = C2 :-
   C1:city[country->C; name@(gs) ->> N1],
   C2:city[country->C; name@(gs) ->> N2],
   strcat("Ciudad de ", N1, N2).
?- sys.strat.doIt.
```

Make cid(gs,"USA") also accessible via cid(gs,"United States"):

```
cid(gs,"USA") = cid(gs,"United States").
```

CIA-relevant countries are those for which a capital is given:

```
C:cia_rel_country :-
   C:country[capital@(cia)->Cap], not Cap = "none".
?- sys.strat.doIt.
```

Fusing CIA and GlobalStatistics Countries.

- Fuse two countries if they have the same name:
  ```
  C1 = C2 :-
     C1:country[name@(cia)->N],
     C2:country[name@(gs)->>N],
     not C1=C2.
  ?- sys.strat.doIt.
  ```
- Fuse two countries if they belong to the same continent and the CIA capital's name is the same as name of the GlobalStatistics capital:
  ```
  C1 = C2 :-
     C1:country[continent@(gs)->>CT1[name->CN1]],
     C1.capital[name@(gs)->>N],
     not C1[name@(cia) ->_],
     C2:country[name@(cia)->_; continent->CT2[name->CN2]; capital@(cia)->N],
     not C1=C2, substr(CN2,CN1). % North/South America vs America
  ?- sys.strat.doIt.
  ?- sys.strat.doIt.
  ```

Note that some gs country-names are not matched; the unmatched names occur only on the link page, the cities and provinces are matched instead with other names: "Myanmar" → "Myanmar (Burma)", "Congo, Dem Rep" → "Congo", "Czech Rep." → "Czech Republic", "Korea" → "South Korea", "North Korea", "UAE" → "United Arab Emirates".

Fusing CIA and GlobalStatistics Cities. In MONDIAL-CIA, the only cities mentioned are the capitals. Some of them are stored under different names than in GlobalStatistics.

```
city_synonym("Bucharest","Bucuresti").
city_synonym("Warsaw","Warszawa").
city_synonym("New Delhi","Delhi").
city_synonym("Tashkent","Toshkent").
city_synonym("Addis Ababa","Addis Abeba").
city_synonym("La Hague","s'-Gravenhage").
?- sys.strat.doIt.
C1 = C2 :-
   C1:city[country->Ctry1;name@(gs)->>CN1],
   C2:city[country->Ctry1;name@(gs)->>CN2], city_synonym(CN1,CN2).
?- sys.strat.doIt.
```

Fusing CIA and mondial-codes Countries. The mondial-codes data source contains the countries' english, german, and local names and their car-codes.

Countries are objects.
For most countries, the CIA and english name coincides:

```
C1 = C2 :-
   C1:country[name@("english")->X],
   C2:cia_rel_country[name@(cia)->X].
?- sys.strat.doIt.
```

For other countries, the CIA name coincides with the german or local name, or the GlobalStatistics name coincides with the english name:

```
C1 = C2 :-
   C1:country[name@("german")->N2], not C1.name@(cia)[],
   C2:country[name@(cia)->N2].
C1 = C2 :-
   C1:country[name@("local")->N2], not C1.name@(cia)[],
   C2:country[name@(cia)->N2].
C1 = C2 :-
   C1:country[name@("english")->X], not C1.name@(cia)[],
   C2:country[name@(gs)->>X].
?- sys.strat.doIt.
```

By this, for all CIA-relevant countries, also the method country.car_code is defined.

Linking Qiblih-Cities to Countries. In mondial-qiblih, the countries are only represented by the string-valued method qcity[country@(qiblih)⇒country-name]. All Qiblih cities for which qcity.country@(qiblih) matches an N which is any of country.name@(cia) or country.name@(gs) are linked to the respective country.

```
City[country->Country] :-
   City:qcity[country@(qiblih)->N], Country:country[name@(cia)->N].
```

There are some CIA capitals which are also contained in Qiblih, but not with the exact country name. They are matched by name, excluding the US and Canadian Cities (which can be detected since for them, province@(qiblih) is defined).

```
City[country->Country] :-
   City:qcity[name@(qiblih)->N], not City.country[],
   not City.province@(qiblih)[],
   (Country:cia_rel_country).capital[name@(gs)->>N].
?- sys.strat.doIt.
```

US and Canadian Qiblih cities can be linked to the corresponding province object using qcity[province@(qiblih)⇒prov-name]:

```
C[province->>P] :-
   C:qcity[country->S; province@(qiblih)->N],
   P:prov[country->S; name->N].
?- sys.strat.doIt.
```

Merging Qiblih-Cities with GlobalStatistics main cities. If no province is known in Qiblih (all except US and CDN), cities are matched by name (comparing with both GlobalStatistics or CIA name – note that capitals can have both of them) and country:

```
C1 = C2 :-
   C1:qcity[name@(qiblih)->N;country->Ctry],
   C2:city[name@(gs)->>N;country->Ctry],
   not C1.province@(qiblih)[].
C1 = C2 :-
   C1:qcity[name@(qiblih)->N;country->Ctry],
   C2:city[name@(cia)->N;country->Ctry],
   not C1.province@(qiblih)[].
?- sys.strat.doIt.
```

In the USA, Qiblih and GlobalStatistics main cities are matched by name/state/country:

```
C1 = C2 :-
   C1:qcity[name@(qiblih)->N;province->>P;country->Ctry],
   not C1..name@(gs)[],
   C2:city[name@(gs)->>N;province->>P;country->Ctry].
?- sys.strat.doIt.
```

For Canada, Qiblih provides provinces, but GlobalStatistics does not. Cities are matched by name and country:

```
C1 = C2 :-
   C1:qcity[name@(qiblih)->N;country->Ctry[name@(gs)->>"Canada"]],
   not C1..name@(gs)[],
   C2:city[name@(gs)->>N;country->Ctry].
?- sys.strat.doIt.
```

Organizations and Memberships. Note that the CIA country data and the CIA organizations data is not fully consistent: there are 71 memberships which are not mentioned in the Country data, and there are 6 organizations which are mentioned in the country data which do not exist in the organization data.

Since the organizations data is more detailed (membership type), the membership information is taken from there. The memberships are linked to the countries. Note that there are members which are no legal countries, mostly international commissions or local interest groups. Note that the member lists contains some countries which are both members and associate members/observers (IFRCS). Here, we take only the full membership.

```
O[members@("member")->>C] :-
   O[member_names@("member")->>CN], C:country[name@(cia)->CN].
O[members@(Type)->>C] :-
   O[member_names@(Type)->>CN], C:country[name@(cia)->CN],
   not O[member_names@("member")->>CN].
```

The seats of organizations are linked to the respective cities:

```
O[seat->Cty] :-
   O:org[seatcity->N;seatcountry->CN],
   Cty:city[name@(gs)->> N], Cty.country[name@(gs)->>CN].
O[seat->Cty] :-
   O:org[seatcity->N;seatcountry->CN],
   Cty:city[name@(_)-> N], Cty.country[name@(gs)->>CN].
?- sys.strat.doIt.
```

Karlsruhe Terra: Fusing Countries. The TERRA country objects are fused with the country objects. In most cases, the objects can be merged using terra_country[code@(terra)⇒car_code] and country[car_code⇒car_code]:

```
L = C :- L:terra_country[code@(terra)->Code], C:country[car_code->Code].
?- sys.strat.doIt.
```

Others are fused by matching terra_country[name@(terra)⇒name] with country[name@("german")⇒german_name] or even with terra_country[name@(gs)⇒name]:

```
L = C :-
   L:terra_country[name@(terra)->N], not L.car_code[],
   C:country[name@("german")->N].
L = C :-
   L:terra_country[name@(terra)->N], not L.car_code[],
   C:country[name@(gs)->>N].
```

If the name of a capital is the same in CIA/GlobalStatistics and TERRA, the countries are also the same.

```
L = C :-
   L:terra_country[capital@(terra)->CN], not L.car_code[],
   C:country[capital@(gs)->>CN].
?- sys.strat.doIt.
```

The only remaining TERRA country objects are those of the SU, GDR and CS (countries which have been dissolved).

Fusing Administrative Divisions. TERRA contains data about cities belongig to administrative divisions which is not contained in GlobalStatistics. This information has to be integrated. Some of the administrative divisions can be fused by country/name:

```
L = P :-
    L:terra_prov[country@(terra)->C;abbrev@(terra)->AB;name@(terra)->N],
    not C  = AB,
    P:prov[name -> N], P.country[code@(terra)->C].
?- sys.strat.doIt.
```

For the remaining administrative divisions, the capital coincides with the capital of an administrative division in GlobalStatistics of the same country (which is already merged). Here, terra_prov[country@(terra)⇒country_id; capital@(terra)⇒city_name] and prov[country.code@(terra)⇒country_id; capital.name@(gs)⇒⇒city_name] are used. Since TERRA contains the country itself as a province if no other provinces are known, these have to be excluded.

```
L = P :-
    L:terra_prov[country@(terra)->C;abbrev@(terra)->AB;
            capital@(terra)->CN],
    not C  = AB, not L:prov,
    P:prov, P.capital[name@(gs)->>CN],
    P.country[code@(terra)->C].
?- sys.strat.doIt.
```

Finland and Norway's provinces are only contained in Terra. They are made administrative divisions of their countries, the capital Cap is made a city and the methods Cap.province and Cap.country are defined:

```
C[adm_divs->>L:prov[country->C;name->N;
                   capital->Cap;population->E]],
Cap:city[province->>L;country->C] :-
   L:terra_prov[country@(terra)->"SF";abbrev@(terra)->AB;
           name@(terra)->N;capital@(terra)->CN;pop@(terra)->E],
   C:country[code@(terra)->"SF"],
   Cap:terra_city[country@(terra)->"SF";prov@(terra)->>AB;
             name@(terra)->CN].

C[adm_divs->>L:prov[country->C;name->N;
                   capital->Cap;population->E]],
Cap:city[province->>L;country->C] :-
   L:terra_prov[country@(terra)->"N";abbrev@(terra)->AB;
           name@(terra)->N;capital@(terra)->CN;pop@(terra)->E],
   C:country[code@(terra)->"N"],
   Cap:terra_city[country@(terra)->"N";prov@(terra)->>AB;
             name@(terra)->CN].
```

For the former USSR, all its administrative divisions (the SSR's) are now independent and are fused with the respective country objects. Here, the matching is done by the first three letters of the (german) name. Note that this leads to objects which are countries and tprovs, thus, o.ltid@(terra) is defined.

```
L = C :-
   L:terra_prov[name@(terra)->N1;country@(terra)->"SU"],
   C:country[name@("german")->N2], not C:terra_country,
   pmatch(N1,"/\A(...)/","$1",N1short),
   pmatch(N2,"/\A(...)/","$1",N2short), N1short = N2short.
?- sys.strat.doIt.
```

Continent Information. Terra contains information about the distribution of countries over continents in terra_country[encompassed@(continent)⇒percentage]. Note that this is not contained in CIA or GlobalStatistics for countries which belong to two continents (Russia, Egypt, Turkey).

```
C[continent@(CT)->Perc] :-
   C:country[encompassed@(CN)->Perc], CT:continent[name->CN].
C[continent@(asia)->80;continent@(europe)->20] :-
   C:country[name@(cia)->"Russia"].
?- sys.strat.doIt.
```

all other countries belong with 100% to the continent which is given in CIA/GlobalStatistics:

```
C[continent@(Cont)->#100] :-
   C:country[continent@(gs)->>Cont], not C.continent@(_).
?- sys.strat.doIt.
```

Fusing Cities. TERRA contains geographical coordinated for all its cities, many of them are also contained in GlobalStatistics without coordinates and Qiblih also does not provide their coordinates. For countries where no administrative divisions are distinguished in TERRA(i.e. terra_city.country = terra_city.prov), cities are merged by country/name.

```
S = C :-
   S:terra_city[name@(terra)->N;country@(terra)->CC;prov@(terra)->>CC],
   Country:country[code@(terra)->CC],
   C:city[country->Country;name@(gs)->>N].
```

Cities where GlobalStatistics knows the province (US cities and capitals of provinces) are fused using name/province/country:

```
S = C :-
   S:terra_city[name@(terra)->N;country@(terra)->CC;prov@(terra)->>P],
   not CC = P,
   Country:country[code@(terra)->CC],
   LT:terra_prov[name@(terra)->PName;country@(terra)->CC;
            abbrev@(terra)->P],
   C:city[country->Country;name@(gs)->>N],
   C..province[name->PName].
?- sys.strat.doIt.
```

There are cities where TERRA knows the province but GlobalStatistics does not know it although the province as an object is known. These can be fused and linked using the TERRA information.

```
S = C, C[province ->> P] :-
   S:terra_city[name@(terra)->N;country@(terra)->CC;prov@(terra)->>AB],
   not CC = P,
   Country:country[code@(terra)->CC],
   P:prov[country@(terra)->CC;abbrev@(terra)->AB],
   C:city[country->Country;name@(gs)->>N], not C..province[].
?- sys.strat.doIt.
```

TERRA cities which are capitals are fused with the CIA/GlobalStatistics capital of the same country. Here, all countries where the capital changed must be excluded!

```
S = C :-
   S:terra_city[name@(terra)->N;country@(terra)->CC],
   L:terra_country[code@(terra)->CC; capital@(terra)->N],
   L:country[capital->C], not S:city, not CC="D".
```

the same for provinces

```
S = C :-
   S:terra_city[name@(terra)->N;country@(terra)->CC;prov@(terra)->>AB],
   L:terra_prov[country@(terra)->CC;abbrev@(terra)->AB;capital@(terra)->N],
   L:prov[capital->C], not S:city.
```

Again, the former USSR needs special treatment: capitals of SSRs are now the capitals of the new countries:

```
S = C :-
   L:terra_prov[country@(terra)->"SU";capital@(terra)->N;capital->C],
   S:terra_city[name@(terra)->N].
?- sys.strat.doIt.
```

non-capital USSR cities can only be matched by name

```
S = C :-
   S:terra_city[country@(terra)->"SU";prov@(terra)->>P;name@(terra)->N],
   L:country[code@(terra)->"SU";abbrev@(terra)->P],
   C:city[country->L;name@(gs)->>N].
?- sys.strat.doIt.
```

Additional Terra Information. TERRA contains some more data which has not yet been integrated, e.g. cities which are not mentioned in any of the other databases, or additional information about cities which are known.

There are some countries where GlobalStatistics gives no main cities (except the capital), but TERRA gives some additional cities. Then, these cities can be added to the database without any risk to have a city under two different names. This also holds when all GlobalStatistics cities are capitals of provinces and TERRA also knows these provinces/capitals (Then the capitals are already fused).
First, all GlobalStatistics cities are determined which are no capitals:

```
C[notcapital->true] :-
   C:city[country->X], not X[capital->C],
   not X..adm_divs[capital->C].
?- sys.strat.doIt.
```

For countries which have provinces in TERRA and GlobalStatistics knows no non-capital cities (effectively, this implies that GlobalStatistics knows no provinces, these are only N and SF), all TERRA cities can be added with their provice information.

```
S:city[country->C;province->>P] :-
   C:country[code@(terra)->CC],
   not C..main_cities[notcapital->true],
   S:terra_city[country@(terra)->CC; prov@(terra)->>AB],
   not CC = AB, not S:city,
   P:prov[country->C;abbrev@(terra)->AB].
```

For countries (except the former USSR) where GlobalStatistics knows no main cities and TERRA knows no provinces, these cities are added.

```
S:city[country->C] :-                   %% other countries
   C:country[code@(terra)->CC], not CC = "SU",
   N = count{City [C]; C[main_cities->>City]}, N = 1,
   S:terra_city[country@(terra)->CC; prov@(terra)->>CC], not S:city.
?- sys.strat.doIt.
```

TERRA city longitude and latitude data is added if there is no other (Qiblih) information:

```
C[longitude->Long;latitude->Lat] :-
   C:city[longitude@(terra)->Long;latitude@(terra)->Lat],
   not C.latitude, not C.longitude.
?- sys.strat.doIt.
```

Those TERRA-cities which are the capital of something are made cities: (currently this is only for El Alaiun, the capital of Western Sahara):

```
C:city :-
    CT:country[capital->C],
    C:terra_city[name@(terra)->CN], not C:city.
?- sys.strat.doIt.
```

The TERRA city population data is added if there is no other (GlobalStatistics) information:

```
C[population@(87)->E] :-
    C:city[pop@(terra)->E], not C[population@(_)->_].
?- sys.strat.doIt.
```

TERRA capital data is added if there is no other (CIA/GlobalStatistics) information – effectively, this is only Western Sahara:

```
C[capital->City:city[country->C]] :-
   C:country[capital@(terra)->N], not C.capital,
   City:city[name@(terra)->N].
?- sys.strat.doIt.
```

For countries where the car-code is not yet given, it is taken from TERRA (currently this is only French Guiana)

```
C[car_code->CC] :-
  C:country[code@(terra) -> CC], not C.car_code[].
```

Terra geo objects' locations. For TERRA geo objects (rivers, lakes, seas, deserts, mountains, islands), the country/province data is translated.

```
mountain::geo_obj.
desert::geo_obj.
island::geo_obj.
water::geo_obj.
lake::water.
river::water.
sea::water.
```

First, the countries are determined, for the former USSR, (SU,prov) is changed into (prov):

```
Obj[country->>C] :-
   Obj:geo_obj[place->>pair(AB,CC)], not CC =  "SU",
   C:country[code@(terra)->CC].
Obj[country->>C] :-
   Obj:geo_obj[place->>pair(AB,"SU")],
   C:country[abbrev@(terra)->AB].
Obj[country->>C] :-
   Obj:geo_obj[place->>pair(AB,"CS")],
   C:country[car_code->"CZ"].
```

As far as provinces are given, this information is also added:

```
Obj[prov@(C)->>P] :-
   Obj:geo_obj[country->>C;place->>pair(AB,CC)],
   C:country[code@(terra)->CC;adm_divs->>P],
   P:prov[abbrev@(terra)->AB].
```

The resulting database is dumped to *mondial-all-facts.flp*:

```
?- sys.strat.doIt.
?- sys.theOM.saveAs@("../Mondial-facts/mondial-all-facts.flp").
```

The Object Manager saved in *mondial-all-facts.flp* can be used by follow-up programs, transforming the database into different representations.

5 The Object-Oriented Model

The program *mondial-clean.flp* loads the dumped OM and resolves multiple names. The object-oriented database containing only the resulting objects and methods is written to *mondial-oo-facts.flp*:

```
?- sys.theOM.debugOn. % trace Web access
?- sys.theOM.eqTraceOn. % trace derived equalties
?- sys.prn.style@("bound").  % set output mode
?- sys.load@("../Mondial-facts/mondial-all-facts.flp").
?- sys.load@("mondial-addprovs.flp").
?-sys.strat.doIt.
format[name->"$out[0]=$in[0];
             $out[0]=~tr/áâãäåÉéèíïÍÖóôöúüçñ/aaaaaEeeiiIOooouucn/"].
C[province->>P] :-
  C:city[name@(gs)->>N2].country[car_code->CC],
  P:prov[name->PN2].country[car_code->CC],
  addprov(N,CC,PN),
  perl(format.name, N, N2),
  perl(format.name, PN, PN2).
?-sys.strat.doIt.
addprov2(N2,CC,PN2) :- addprov(N,_,CC,PN),
  perl(format.name, N, N2),  perl(format.name, PN, PN2).
?-sys.strat.doIt.
```

```
C[province->>P] :-
  C:city[name@(gs)->>N].country[car_code->CC],
  P:prov[name->PN].country[car_code->CC],
  addprov2(N,CC,PN).
?-sys.strat.doIt.
```

For some geo objects, additional countries are given.

```
G[country->>C] :-
  addgeocountry(Geo,Name,Code),
  G:Geo[name->Name], C:country[car_code->Code].
```

For geo objects, for every country, a semicolon-list of provinces is given.

```
addgeoprov(Geo, Name, Code, ProvName) :-
  addgeo(Geo, Name, Code, Strg),
  pmatch(Strg, "/([ÄA-Za-z][^;]*)/g", "$1", ProvName).
G[prov@(C)->>P] :-
  addgeoprov(Geo, Name, Code, ProvName),
  G:Geo[name->Name; country->>C[adm_divs->>P]],
  C[car_code->Code], P[name->ProvName].
?- sys.strat.doIt.
```

Data Cleaning. Until now, data has only been integrated in the sense of collecting. There has been no treatment of different names for the same objects used in the individual sources or of missing values.

For New Caledonia (a french overseas territory, which is a seat of an organization), there is no country code.

```
C[car_code->"NCA"] :-
   C:country[name@(cia)->"New Caledonia"].
```

The official name of a country is the name used in the CIA World Factbook.

```
C[name->Name] :- C:country[name@(cia)->Name].
?-sys.strat.doIt.
```

If multiple names for a city are given, we try the CIA name, the Qiblih name, the shortest GeoStatistics name, or the MONDIAL name.

```
C[name->N] :- C:city[name@(cia)->N].
?- sys.strat.doIt.
C[name->N] :- C:city[name@(qiblih)->N], not C.name.
?- sys.strat.doIt.
C[name->N] :-
   C:city[name@(gs)->>N], not C.name, strlen(N,L),
   L=min{SL [C]; C[name@(gs)->>NN], strlen(NN,SL)}.
?- sys.strat.doIt.
C[name->N] :- C:city[name@(terra)->N], not C.name.
?- sys.strat.doIt.
```

Several cities belong to several administrative divisions These are mainly cities s.t. the municipality itself is an administrative division which is often for its own and it is the capital of the surrounding province. If there is no province which is obviously the municipality, as a heuristics, we take the smaller one.

```
C[theProvince->P] :-
   C:city[province->>P],
   Anz=count{NN [C]; C..province[name->NN]}, Anz = 1.
C[theProvince->P] :-
   C:city[province->>P;name@(gs)->>N], not C.theProvince[],
   P[name->PN], strcat(N," (munic.)",PN),
   Anz=count{NN [C]; C..province[name->NN]}, Anz > 1.
?-sys.strat.doIt.
C[theProvince->P] :-
   C:city[province->>P], not C.theProvince[], P[area->A],
   A = min{AA [C]; C..province[area->AA]}.
?-sys.strat.doIt.
C[theProvince->P] :-
   C:city[province->>P;name@(gs)->>N], not C.theProvince[],
   P[name->PN], N = PN,
   Anz=count{NN [C]; C..province[name->NN]}, Anz > 1.
?-sys.strat.doIt.
```

The total area of continents is computed from the countries' information. Nothe that for the continent *America*, there is no area since countries distinguish between *North AMerica* and *South America*.

```
C[area@(CT)-> AP] :-
  C:country[car_code->Code; continent@(CT)->Perc; total_area->A],
  CT[name->CN], not Code = null, AP = A * Perc / 100.
?- sys.strat.doIt.
C[area->A] :- C:continent,
  A = sum{AA [C]; Country:country[area@(C)->AA]}.
```

Output. The resulting object-oriented database is written to the file *mondial-oo-facts.flp*.

```
save[init->"‘rm ../Mondial-facts/mondial-oo-facts.flp‘; @out=@in;"].
C::basic_geo_obj :- C::geo_obj, not C = geo_obj, not C = water.
?-sys.strat.doIt.
?- perl(save.init,[""],[""]).
?-sys.strat.doIt.
?- sys.prn.style@("instance").  % set output mode
?- sys[output->sys.open@("../Mondial-facts/mondial-oo-facts.flp")].
?- sys.answerChannel.setStream@(output)[].
 ?- C:continent[name->N; area ->A].
 ?- C:country[name->Name; car_code->Code;
            total_area->Area; population -> Pop;
            capital -> Cap].
 ?- C:country[adm_divs->>P].
 ?- C:country[main_cities->>MC].
 ?- C:country[continent@(Cont)->Perc].
 ?- C:country[population_growth->PopGrowth;
             infant_mortality->InfMort].
 ?- C:country[gdp_total->GDP].
 ?- C:country[gdp_agri->Agri].
```

```
 ?- C:country[gdp_ind->Ind].
 ?- C:country[gdp_serv->Serv].
 ?- C:country[inflation->Infl].
 ?- C:country[indep_date->Indep].
 ?- C:country[government->Gov].
 ?- C:country[ethnicgroups@(Egrp)->Perc].
 ?- C:country[religions@(Rel)->Perc].
 ?- C:country[languages@(Lang)->Perc].
 ?- C:country[borders@(Country2)->Length].
 ?- P:prov[name->Name; country->C].
 ?- P:prov[capital->Cap].
 ?- P:prov[area->A].
 ?- P:prov[population->Pop].
 ?- C:city[name->Name; country->Country].
 ?- C:city[theProvince->P].
 ?- C:city[population@(Year)->Pop].
 ?- C:city[longitude->Long; latitude->Lat].
 ?- O:org[abbrev->Abbrev; name->Name].
 ?- O:org[established->Est].
 ?- O:org[seat->City].
 ?- O:org[members@(Type)->>C].
 ?- C::geo_obj.
 ?- C::water.
 ?- M:mountain[name->N].
 ?- M:mountain[height->H].
 ?- M:mountain[longitude->Long; latitude->Lat].
 ?- D:desert[name->N].
 ?- D:desert[area->A].
 ?- I:island[name->N].
 ?- I:island[islands->N].
 ?- I:island[area->A].
 ?- I:island[longitude->Long; latitude->Lat].
 ?- R:river[name->N; to@(M)->X].
 ?- R:river[length->L].
 ?- S:sea[name->N].
 ?- S:sea[depth->D].
 ?- L:lake[name->N].
 ?- L:lake[area->A].
 ?- Object:basic_geo_obj[country->>C].
 ?- Object:basic_geo_obj[prov@(C)->>P].
 ?- C:city[located_at@(WC)->>W].
 ?- bordering_seas(M1,M2).
?- sys.answerChannel.resetStream[].
?- sys.remove@("output").
```

The ER-Diagram of the resulting MONDIAL database is given in Figure 1. The object-oriented schema is defined as follows:

country[name⇒string;	name of the country (english)
adm_div⇒⇒prov;	administrative provinces
main_cities⇒⇒city;	cities in this country

```
    capital⇒city;                        the capital
    car_code⇒string;                     the car code
    continent@(continent)⇒percentage;
                    the percentage of the country belonging to a continent
    total_area⇒number;                   its total area
    population⇒number;                   its population
    population_growth⇒number;            the population growth rate
    infant_mortality⇒permille;           mortality of children in first year in permille.
    gdp_total⇒number;                    Gross domestic product
    gdp_agri⇒percentage;                 percentage of agriculture of the gdp
    gdp_ind⇒percentage;                  percentage of industry of the gdp
    gdp_serv⇒percentage;                 percentage of services of the gdp
    inflation⇒percentage;                inflation rate
    indep_date⇒date_string;              date of independence
    government⇒string;                   type of government
    ethnicgroups@(race_string)⇒percentage;    percentage of ethnic groups
    religions@(religion_string)⇒percentage;   percentage of religions
    languages@(language_string)⇒percentage;   percentage of languages
    borders@(country)⇒number;            length of the border with another country
    coasts⇒⇒string]                      bordering seas

prov[name⇒string;                        name of the province
    capital⇒city;                        its capital city
    country Fd country;                  the country it belongs to
    area⇒number;
    population⇒number;

city[name⇒string;                        name of the province
    province⇒⇒prov;                      the province it belongs to
    country⇒country;                     the country it belongs to
    population@(year)⇒number;            its population in a given year
    longitude⇒number;                    the geographic longitude
    latitude⇒number;                     the geographic latitude
    located_at@(geo_class)⇒⇒geo_water]   the rivers/seas/lakes where it is located at.

org[name⇒string;                         name of the organization (in english)
    abbrev⇒string;                       the official abbreviation
    established⇒date_string;             date of establishment
    seat⇒city;                           its seat
    members@(membertype)⇒⇒country].      its members (with the membership type).

mountain :: geo_object[name⇒string; name of the mountain
    height⇒number;                       the height of a mountain
    longitude⇒number;                    its longitude
    latitude⇒number;                     and latitude
    country⇒⇒country;                    all countries where the mountain
                                         is located.
    prov@(country)⇒⇒prov]                all pairs (country,province) where the mountain
                                         is located.

desert :: geo_object[name⇒string; name of the desert
```

area⇒number;	its area
country⇒⇒country;	all countries where the desert is located.
prov@(country)⇒⇒prov]	all pairs (country,province) where the desert is located.

island :: geo_object[name⇒string;	name of the island
islands⇒string;	the group of islands it belongs to
area⇒number;	ita area
longitude⇒number;	its longitude
latitude⇒number;	and latitude
country⇒⇒country;	all countries where the island is located.
prov@(country)⇒⇒prov]	all pairs (country,province) where the island belongs to.

geo_water :: geo_object

sea :: geo_water[name⇒string;	name of the sea
depth⇒number;	its maximal depth
country⇒⇒country;	all countries which border the sea.
prov@(country)⇒⇒prov]	all pairs (country,province) which border the sea.

lake :: geo_water[name⇒string;	name of the lake
area⇒number;	its area
country⇒⇒country;	all countries where the lake is located.
prov@(country)⇒⇒prov]	all pairs (country,province) where the lake is located.

river :: geo_water[name⇒string;	name of the river
length⇒number;	its length
to@(geo_water)⇒water;	the water(s) it flows into
country⇒⇒country;	all countries where the river is located.
prov@(country)⇒⇒prov]	all pairs (country,province) where the river is located.

6 Mapping to a Relational Model

6.1 Datalog

The MONDIAL database has been transformed into a relational model (as used e.g., in Datalog and SQL database systems). The mapping implemented by *mondial-rel.flp* follows the usual transformation rules from an ER model to a relational model and writes the relational database to *mondial-rel-facts.flp* (Datalog representation):

```
?- sys.theOM.debugOn. % trace Web access
?- sys.theOM.eqTraceOn. % trace derived equalties
?- sys.prn.style@("bound").  % set output mode
```

```
?- sys.load@("../Mondial-facts/mondial-oo-facts.flp").
?-sys.strat.doIt.
```

Completing the database. For those cities or geo-objects where it is not known to which province they belong, the country is made a province of itself, and the city/geo-object is linked to this pseudo-province. Provinces are only considered for countries where for (nearly) all cities, the province is known (in the current database, this excludes, e.g., MA, VN, THA)

```
C[provinces->true] :-
   C:country,
   AnzCit = count{X [C]; C[main_cities->>X]},
   AnzCitProv = count{Y [C]; C[main_cities->>Y], Y.theProvince},
   #0.8 * AnzCit < AnzCitProv.
?- sys.strat.doIt.
C[adm_divs->>C:prov[country->C]] :-
   C:country, not C.provinces.
?- sys.strat.doIt.
```

Null is made an object which is member of all classes which can be referenced. This allows a unified handling of null values and non-null values.

```
null:country[car_code->null].
null:province[country->null;name->null].
null:city[theProvince->null;name->null].
null:fluss.
null:see.
null:meer.
```

In the following, all undefined methods are linked to "null" and the object-oriented model is encoded into a relational schema.

```
C[area->null] :- C:country, C2:country, C2.M[], not C.M[],
    not M = provinces.
country(Name, Code, Cap, Prov, Area, Pop) :-
  C:country[name->Name; car_code->Code;
           total_area->Area; population -> Pop],
  not Code = null, C.provinces,
  C.capital[name->Cap],
  C.capital.theProvince[name->Prov].
country(Name, Code, Cap, Name, Area, Pop) :-
  C:country[name->Name; car_code->Code;
           total_area->Area; population -> Pop],
  not Code = null, not C.provinces,
  C.capital[name->Cap].
population(Code, PopGrowth, InfMort) :-
  C:country[car_code->Code;
           population_growth->PopGrowth; infant_mortality->InfMort],
  not Code = null.
politics(Code, Indep, Gov) :-
  C:country[car_code->Code; indep_date->Indep;government->Gov],
  not Code = null.
```

```
economy(Code, GDP, Agri, Ind, Serv, Infl) :-
  C:country[car_code->Code; gdp_total->GDP; gdp_agri->Agri;
            gdp_ind->Ind; gdp_serv->Serv; inflation->Infl],
  not Code = null.
language(Code, Lang, Perc) :-
  C:country[car_code->Code; languages@(Lang)->Perc],
  not Code = null.
ethnic_group(Code, Egrp, Perc) :-
  C:country[car_code->Code; ethnicgroups@(Egrp)->Perc],
  not Code = null.
religion(Code, Rel, Perc) :-
  C:country[car_code->Code; religions@(Rel)->Perc],
  not Code = null.
borders(Code1, Code2, Length) :-
  C:country[car_code->Code1; borders@(Country2)->Length; total_area->A1],
  Country2[car_code->Code2; total_area->A2],
  A1 < A2, not Code1 = null, not Code2 = null.
?- sys.strat.doIt.
borders(Code2, Code1, Length) :-
  C:country[car_code->Code1; borders@(Country2)->Length; total_area->A1],
  Country2[car_code->Code2; total_area->A2],
  A1 > A2, not borders(Code2, Code1, _),
  not Code1 = null, not Code2 = null.
encompasses(Code,CN,Perc),
C[area@(CT)-> AP] :-
  C:country[car_code->Code; continent@(CT)->Perc; total_area->A],
  CT[name->CN], not Code = null, AP = A * Perc / 100.
?- sys.strat.doIt.
continent(N,A) :- C:continent[name->N; area ->A].
P[capital->null] :- P:prov, not P.capital[].
P[area->null] :- P:prov, not P.area[].
P[population->null] :- P:prov, not P.population[].
?- sys.strat.doIt.
province(Name, CountryCode, Pop, Area, Capital, Capprov) :-
  P:prov[name->Name; area->Area; population->Pop],
  P.country[car_code->CountryCode; name->CountryName;
            provinces->true],
  P.capital[name->Capital],
  P.capital.theProvince[name->Capprov],
  not CountryCode = null.
```

countries occur as provinces if provinces are not known:

```
province(CountryName, Code, Pop, Area, Cap, CountryName) :-
  C:country[name->CountryName; car_code->Code;
            total_area->Area; population->Pop],
  not C.provinces, not Code = null, C.capital[name->Cap].
?- sys.strat.doIt.
C[longitude->null] :- C:city, not C.longitude[].
C[latitude->null] :- C:city, not C.latitude[].
C[population@(0)->null] :- C:city, not C[population@(_)->_].
```

```
?- sys.strat.doIt.
city(Name, CountryCode, Province, Pop, Long, Lat) :-
  C:city[name->Name; population@(Year)->Pop;
         longitude->Long; latitude->Lat],
  C.country[car_code->CountryCode; provinces->true],
  C.theProvince[name->Province],
  not Name = "none", not CountryCode = null,
  Year = max{Y [C]; C[population@(Y)->_]}.
city(Name, CountryCode, CountryName, Pop, Long, Lat) :-
  C:city[name->Name; population@(Year)->Pop;
         longitude->Long; latitude->Lat],
  C.country[name->CountryName; car_code->CountryCode],
  not C.country.provinces,
  not Name = "none", not CountryCode = null,
  Year = max{Y [C]; C[population@(Y)->_]}.
?- sys.strat.doIt.
O[established->null] :- O:org, not O.established[].
O[seat->null] :- O:org, not O.seat[].
?- sys.strat.doIt.
organization(Abbrev, Name, City, CountryCode, Prov, Est) :-
  O:org[abbrev->Abbrev; name->Name; established->Est],
  O.seat[name->City],
  O.seat.country[car_code->CountryCode; provinces->true],
  O.seat.theProvince[name->Prov].
organization(Abbrev, Name, City, CountryCode, CountryName, Est) :-
  O:org[abbrev->Abbrev; name->Name; established->Est],
  O.seat[name->City],
  O.seat.country[name->CountryName; car_code->CountryCode],
  not O.seat.country.provinces.
is_member(CC, Abbrev, Type) :-
  O:org[abbrev->Abbrev; members@(Type)->>C[car_code->CC]],
  not CC = null.
?- sys.strat.doIt.
O[M->null] :- GC::geo_obj, O:GC, O2:GC.M[], not O:GC.M[].
O[to@(C)->null] :- O:water, C::water, not O.to@(C)[].
?-sys.strat.doIt.
O[provUnknown->true] :-  O:GC[country->>C], C.provinces,
    GC::geo_obj, not GC = geo_obj, not GC = water,
    not O..prov@(C).
?-sys.strat.doIt.
mountain(Name, Heigth, Long, Lat) :-
  B:mountain[name->Name; height->Heigth;
         longitude->Long; latitude->Lat], not B.provUnknown.
desert(Name, Area) :-
  D:desert[name->Name; area->Area], not D.provUnknown.
island(Name, Islands, Area, Long, Lat) :-
  I:island[name->Name; islands->Islands; area->Area;
          longitude->Long; latitude->Lat], not I.provUnknown.
river(Name, River, Lake, Sea, Length) :-
  R:river[name->Name; length->Length],
```

```
  R.to@(river)[name->River], R.to@(lake)[name->Lake],
  R.to@(sea)[name->Sea], not R = null, not R.provUnknown.
lake(Name, Area) :-
  L:lake[name->Name; area->Area], not L = null, not L.provUnknown.
sea(Name, Depth) :-
  S:sea[name->Name; depth->Depth], not S = null, not S.provUnknown.
?- sys.strat.doIt.
geo(Type, Name, Country, Prov) :-
  Type::geo_obj, O:Type[name->Name; prov@(C)->>P],
  not Type = geo_obj, not Type = water,
  C:country[car_code->Country; provinces->_],
  P[name->Prov], not O.provUnknown.
geo(Type, Name, Country, CountryName) :-
  Type::geo_obj, O:Type[name->Name; country->>C],
  not Type = geo_obj, not Type = water,
  C:country[car_code->Country;name->CountryName],
  not C.provinces, not O.provUnknown.
?- sys.strat.doIt.
geo_lake(Name, Country, Prov) :-
  geo(lake, Name, Country, Prov).
geo_mountain(Name, Country, Prov) :-
  geo(mountain, Name, Country, Prov).
geo_river(Name, Country, Prov) :-
  geo(river, Name, Country, Prov).
geo_sea(Name, Country, Prov) :-
  geo(sea, Name, Country, Prov).
geo_desert(Name, Country, Prov) :-
  geo(desert, Name, Country, Prov).
geo_island(Name, Country, Prov) :-
  geo(island, Name, Country, Prov).
?-sys.strat.doIt.
?- sys.strat.doIt.
O[located_at@(G2)->>null] :-
    O:city, G1::water, G2::water,
    O..located_at@(G1), not O..located_at@(G2).
?-sys.strat.doIt.
located(City,Prov,CountryCode,River,Lake,Sea) :-
    C:city[name->City],
    C.country[car_code->CountryCode; provinces->true],
    C.theProvince[name->Prov],
    C..located_at@(river)[name->River],
    C..located_at@(lake)[name->Lake],
    C..located_at@(sea)[name->Sea].
located(City,CountryName,CountryCode,River,Lake,Sea) :-
    C:city[name->City],
    C.country[name->CountryName;car_code->CountryCode],
    not C.country.provinces,
    C..located_at@(river)[name->River],
    C..located_at@(lake)[name->Lake],
    C..located_at@(sea)[name->Sea].
```

```
?-sys.strat.doIt.
merges_with(N1,N2) :-
    bordering_seas(M1,M2),
    M1[name->N1], M2[name->N2].
?-sys.strat.doIt.
save[init->"`rm ../Mondial-facts/mondial-rel-facts.flp`; @out=@in;"].
?-sys.strat.doIt.
?- perl(save.init,[""],[""]).
?-sys.strat.doIt.
?- sys.prn.style@("instance").  % set output mode
?- sys[output->sys.open@("../Mondial-facts/mondial-rel-facts.flp")].
?- sys.answerChannel.setStream@(output)[].
?- sys.answerChannel.resetStream[].
?- sys.remove@("output").
```

The relational schema of *mondial-rel-facts.flp* is given as follows:

Country: the countries of the world with some data.
name: The country name
code: The car code
capital: The name of the capital
province: The province where the capital belongs to
area: the total area
population: the population number

Economy: economical information about the countries.
country: the country code
GDP: gross domestic product
agriculture: percentage of agriculture of the GDP
service: percentage of services of the GDP
industry: percentage of industry of the GDP
inflation: inflation rate (per annum)

Population: information about the population of the countries.
country: the country code
population_growth: population growth rate (per annum)
infant_mortality: infant mortality (per thousand)

Politics: political information about the countries.
country: the country code
independence: date of indepenedence
government: type of government

Language: infomation about the languages spoken in a country
country: the country code
name: name of the languange
percentage: percentage of the language in this country

Religion: information about the religions in a country
country: the country code
name: name of the religion
percentage: percentage of the language in this country

Ethnic_Group: information about the ethnic groups in a country
country: the country code
name: name of the religion
percentage: percentage of the language in this country

borders: informations about neighboring countries. Note that in this relation, for every pair of neighboring countries (A,B), only one tuple is given – thus, the relation is *not* symmetric.
country1: a country code
country2: a country code
length: length of the border between country1 and country2

Continent: Information about continents.
name: name of the continent
area: total area of the continent

encompasses: information to which continents a country belongs.
country: the country code
continent: the continent name
percentage: percentage, how much of the area of a country belongs to the continent

City: information about cities.
name: the name of the city
country: the country code where it belongs to
province: the name of the province where it belongs to
population: population of the city
longitude: geographic longitude
latitude: geographic latitude

Province: information about administrative divisions.
name: the name of the administrative division
country: the country code where it belongs to
area: the total area of the province
population: the population of the province
capital: the name of the capital
capprov: the name of the province where the capital belongs to
note that *capprov* is not necessarily equal to *name*. E.g., the municipality of *Bogota (Columbia)* is a province of its own, and *Bogota* is the capital of the surrounding province *Cudinamarca*.

Organization: information about political and economical organizations.
name: the full name of the organization
abbreviation: its abbreviation
city: the city where it is seated
country: the country code of its seat
province: the province of its seat
established: date of establishment

is_member: memberships in political and economical organizations.
organization: the abbreviation of the organization
country: the code of the member country

type: the type of membership

Lake: information about lakes.
name: the name of the lake
area: the total area of the lake

River: information about rivers.
name: the name of the river
length: the length of the river
river: the river where it flows to
lake: the lake where it flows to
sea: the sea where it flows to

Sea: information about seas.
name: the name of the sea
depth: the maximal depth of the sea

Mountain: information about mountains
name: the name of the mountain
height: the height of the mountain
coordinates: its geographical coordinates as (longitude,latitude)

Island: information about islands
name: the name of the island
islands: the group of islands where it belogs to
area: the area of the island
coordinates: its geographical coordinates as (longitude,latitude)

Desert: information about deserts.
name: the name of the desert
area: the total area of the desert

geo_Mountain: geographical information about mountains
mountain: the name of the mountain
country: the country code where it is located
province: the province of this country

Note that for a country there can be several provinces where the mountain is located in.
analogous for *geo_island*, *geo_desert*, *geo_river*, *geo_lake*, *geo_sea*.
merges_with: information about neighboring seas.
sea1: a sea
sea2: a sea

located: information about cities located at rivers, lakes, and seas.
city: the name of the city
country: the country code where the city belongs to
province: the province where the city belongs to
river: the river where it is located at
lake: the lake where it is located at
sea: the sea where it is located at
Note that for a given city, there can be several lakes/seas/rivers where it is located at.

The referential dependencies are given in Figure 2.

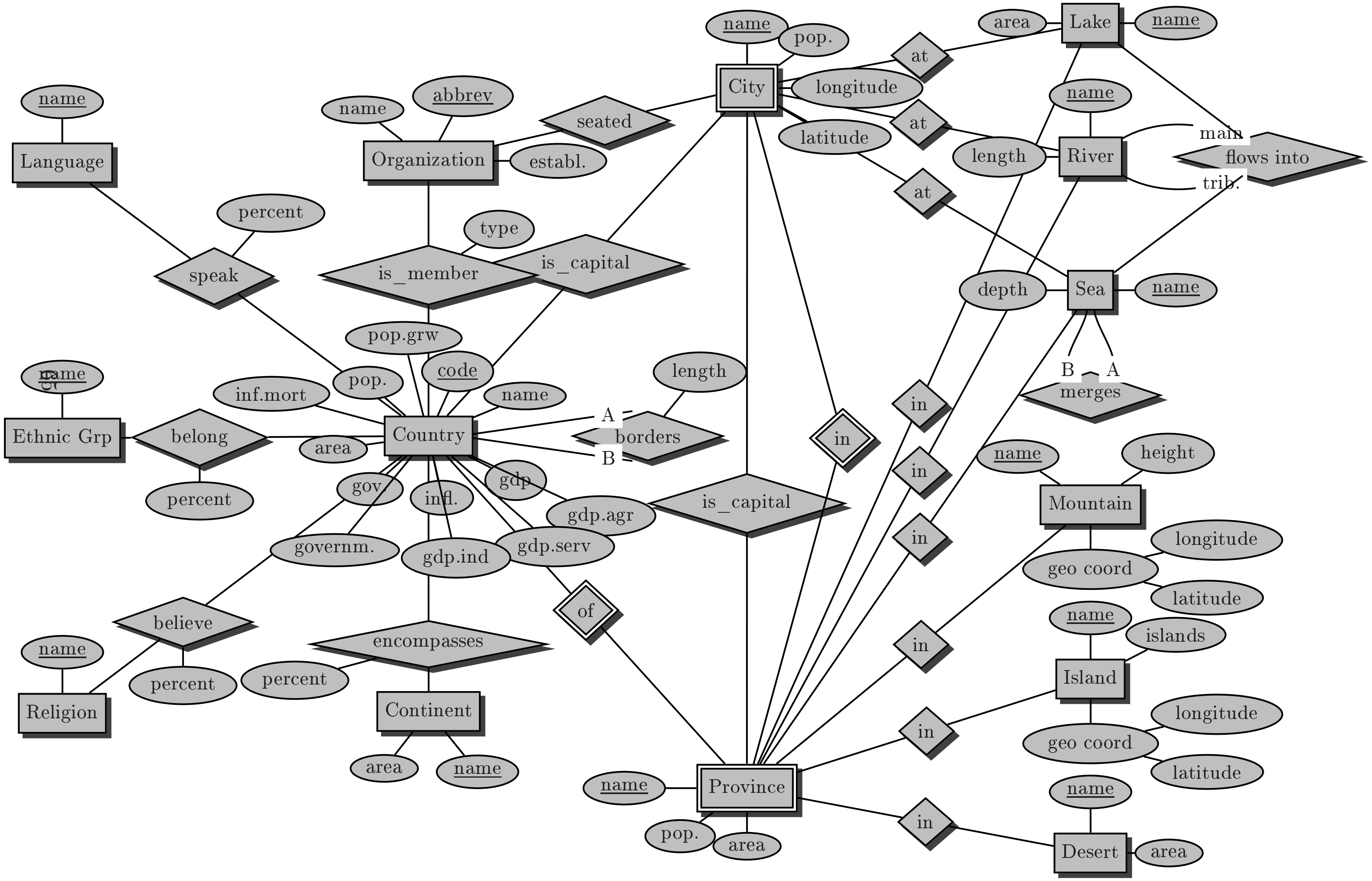
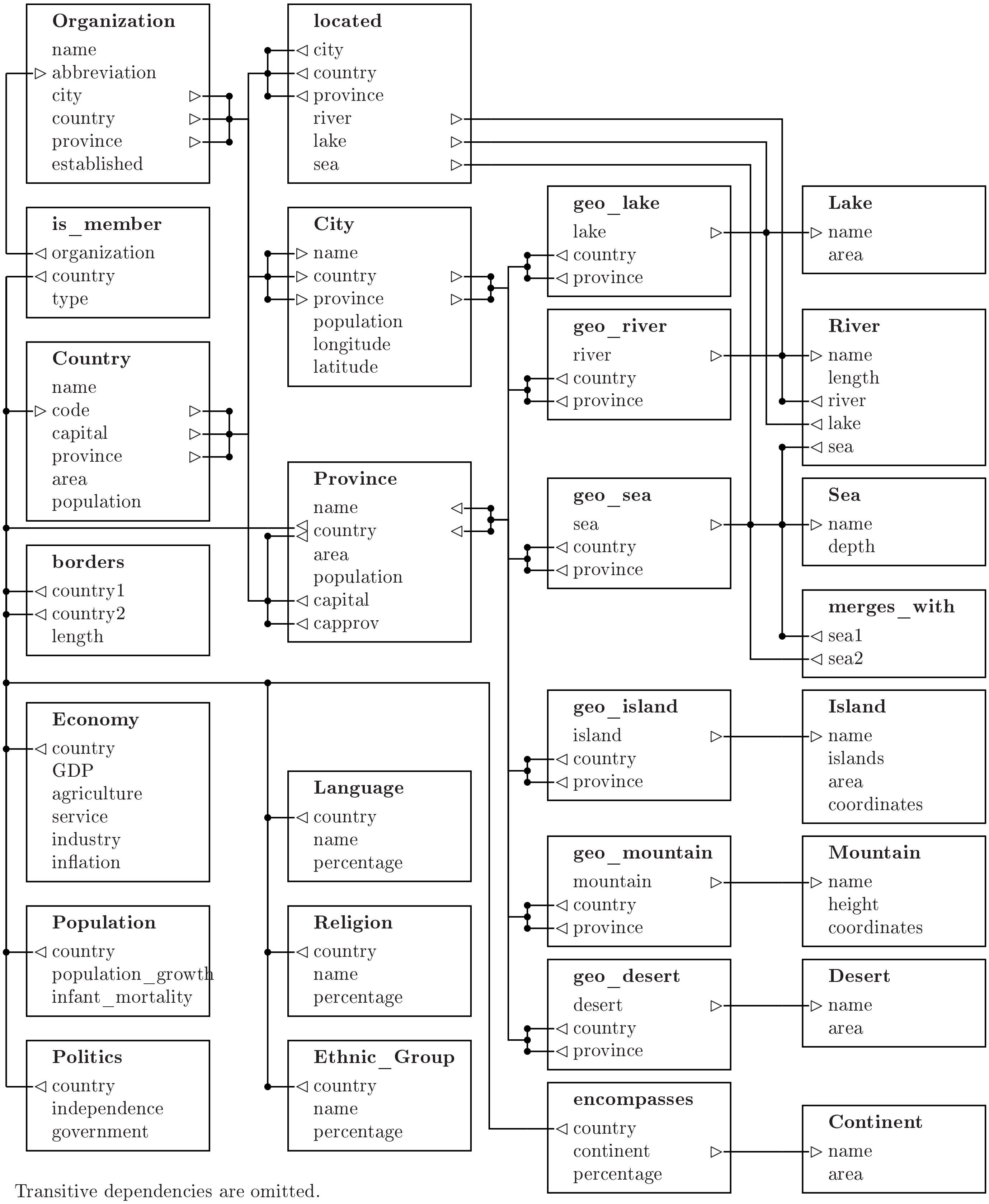


Figure 1: ER-Diagram of the Mondial database



Transitive dependencies are omitted.

Figure 2: Referential Dependencies of the MONDIAL Database

6.2 Oracle

For its second purpose, the database training on ORACLE at the IIF, the MONDIAL database as been converted to ORACLE/SQL. The SQL MONDIAL schema (containing the relations given in the previous section) is generated by the following commands:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM SYYYY';
-- SYYYY means 4-digit-year, S prefixes BC years with "-"

CREATE TABLE Country
(Name VARCHAR2(32) NOT NULL UNIQUE,
 Code VARCHAR2(4) CONSTRAINT CountryKey PRIMARY KEY,
 Capital VARCHAR2(35),
 Province VARCHAR2(32),
 Area NUMBER CONSTRAINT CountryArea
   CHECK (Area >= 0),
 Population NUMBER CONSTRAINT CountryPop
   CHECK (Population >= 0));

CREATE TABLE City
(Name VARCHAR2(35),
 Country VARCHAR2(4),
 Province VARCHAR2(32),
 Population NUMBER CONSTRAINT CityPop
   CHECK (Population >= 0),
 Longitude NUMBER CONSTRAINT CityLon
   CHECK ((Longitude >= -180) AND (Longitude <= 180)) ,
 Latitude NUMBER CONSTRAINT CityLat
   CHECK ((Latitude >= -90) AND (Latitude <= 90)) ,
 CONSTRAINT CityKey PRIMARY KEY (Name, Country, Province));

CREATE TABLE Province
(Name VARCHAR2(32) CONSTRAINT PrName NOT NULL ,
 Country  VARCHAR2(4) CONSTRAINT PrCountry NOT NULL ,
 Population NUMBER CONSTRAINT PrPop
   CHECK (Population >= 0),
 Area NUMBER CONSTRAINT PrAr
   CHECK (Area >= 0),
 Capital VARCHAR2(35),
 CapProv VARCHAR2(32),
 CONSTRAINT PrKey PRIMARY KEY (Name, Country));

CREATE TABLE Economy
(Country VARCHAR2(4) CONSTRAINT EconomyKey PRIMARY KEY,
 GDP NUMBER CONSTRAINT EconomyGDP
   CHECK (GDP >= 0),
 Agriculture NUMBER,
 Service NUMBER,
 Industry NUMBER,
```

```
 Inflation NUMBER);

CREATE TABLE Population
(Country VARCHAR2(4) CONSTRAINT PopKey PRIMARY KEY,
 Population_Growth NUMBER,
 Infant_Mortality NUMBER);

CREATE TABLE Politics
(Country VARCHAR2(4) CONSTRAINT PoliticsKey PRIMARY KEY,
 Independence DATE,
 Government VARCHAR2(120));

CREATE TABLE Language
(Country VARCHAR2(4),
 Name VARCHAR2(50),
 Percentage NUMBER CONSTRAINT LanguagePercent
   CHECK ((Percentage > 0) AND (Percentage <= 100)),
 CONSTRAINT LanguageKey PRIMARY KEY (Name, Country));

CREATE TABLE Religion
(Country VARCHAR2(4),
 Name VARCHAR2(50),
 Percentage NUMBER CONSTRAINT ReligionPercent
   CHECK ((Percentage > 0) AND (Percentage <= 100)),
 CONSTRAINT ReligionKey PRIMARY KEY (Name, Country));

CREATE TABLE Ethnic_Group
(Country VARCHAR2(4),
 Name VARCHAR2(50),
 Percentage NUMBER CONSTRAINT EthnicPercent
   CHECK ((Percentage > 0) AND (Percentage <= 100)),
 CONSTRAINT EthnicKey PRIMARY KEY (Name, Country));

CREATE TABLE Continent
(Name VARCHAR2(20) CONSTRAINT ContinentKey PRIMARY KEY,
 Area NUMBER(10));

CREATE TABLE borders
(Country1 VARCHAR2(4),
 Country2 VARCHAR2(4),
 Length NUMBER
   CHECK (Length > 0),
 CONSTRAINT BorderKey PRIMARY KEY (Country1,Country2) );

CREATE TABLE encompasses
(Country VARCHAR2(4) NOT NULL,
 Continent VARCHAR2(20) NOT NULL,
 Percentage NUMBER,
   CHECK ((Percentage > 0) AND (Percentage <= 100)),
 CONSTRAINT EncompassesKey PRIMARY KEY (Country,Continent));
```

```
CREATE TABLE Organization
(Abbreviation VARCHAR2(12) PRIMARY KEY,
 Name VARCHAR2(80) NOT NULL,
 City VARCHAR2(35) ,
 Country VARCHAR2(4) ,
 Province VARCHAR2(32) ,
 Established DATE,
 CONSTRAINT OrgNameUnique UNIQUE (Name));

CREATE TABLE is_member
(Country VARCHAR2(4),
 Organization VARCHAR2(12),
 Type VARCHAR2(30) DEFAULT 'member',
 CONSTRAINT MemberKey PRIMARY KEY (Country,Organization) );

CREATE OR REPLACE TYPE GeoCoord AS OBJECT
(Longitude NUMBER,
 Latitude NUMBER);
/

CREATE TABLE Mountain
(Name VARCHAR2(20) CONSTRAINT MountainKey PRIMARY KEY,
 Height NUMBER CONSTRAINT MountainHeight
     CHECK (Height > = 0),
 Coordinates GeoCoord CONSTRAINT MountainCoord
     CHECK ((Coordinates.Longitude >= -180) AND
            (Coordinates.Longitude <= 180) AND
            (Coordinates.Latitude >= -90) AND
            (Coordinates.Latitude <= 90)));

CREATE TABLE Desert
(Name VARCHAR2(25) CONSTRAINT DesertKey PRIMARY KEY,
 Area NUMBER);

create table Island
(Name VARCHAR2(25) CONSTRAINT IslandKey PRIMARY KEY,
 Islands VARCHAR2(25),
 Area NUMBER CONSTRAINT IslandAr
  check ((Area >= 0) and (Area <= 2175600)) ,
 Coordinates GeoCoord CONSTRAINT IslandCoord
     CHECK ((Coordinates.Longitude >= -180) AND
            (Coordinates.Longitude <= 180) AND
            (Coordinates.Latitude >= -90) AND
            (Coordinates.Latitude <= 90)));

CREATE TABLE Lake
(Name VARCHAR2(25) CONSTRAINT LakeKey PRIMARY KEY,
 Area NUMBER CONSTRAINT LakeAr CHECK (Area >= 0));
```

```
CREATE TABLE Sea
(Name VARCHAR2(25) CONSTRAINT SeaKey PRIMARY KEY,
 Depth NUMBER CONSTRAINT SeaDepth CHECK (Depth >= 0));

CREATE TABLE River
(Name VARCHAR2(20) CONSTRAINT RiverKey PRIMARY KEY,
 River VARCHAR2(20),
 Lake VARCHAR2(20),
 Sea VARCHAR2(25),
 Length NUMBER CONSTRAINT RiverLength
   CHECK (Length >= 0));

CREATE TABLE geo_Mountain
(Mountain VARCHAR2(20) ,
 Country VARCHAR2(4) ,
 Province VARCHAR2(32) ,
 CONSTRAINT GMountainKey PRIMARY KEY (Province,Country,Mountain) );

CREATE TABLE geo_Desert
(Desert VARCHAR2(25) ,
 Country VARCHAR2(4) ,
 Province VARCHAR2(32) ,
 CONSTRAINT GDesertKey PRIMARY KEY (Province, Country, Desert) );

CREATE TABLE geo_Island
(Island VARCHAR2(25) ,
 Country VARCHAR2(4) ,
 Province VARCHAR2(32) ,
 CONSTRAINT GIslandKey PRIMARY KEY (Province, Country, Island) );

CREATE TABLE geo_River
(River VARCHAR2(20) ,
 Country VARCHAR2(4) ,
 Province VARCHAR2(32) ,
 CONSTRAINT GRiverKey PRIMARY KEY (Province ,Country, River) );

CREATE TABLE geo_Sea
(Sea VARCHAR2(25) ,
 Country VARCHAR2(4)  ,
 Province VARCHAR2(32) ,
 CONSTRAINT GSeaKey PRIMARY KEY (Province, Country, Sea) );

CREATE TABLE geo_Lake
(Lake VARCHAR2(25) ,
 Country VARCHAR2(4) ,
 Province VARCHAR2(32) ,
 CONSTRAINT GLakeKey PRIMARY KEY (Province, Country, Lake) );

CREATE TABLE merges_with
(Sea1 VARCHAR2(25) ,
```

```
 Sea2 VARCHAR2(25) ,
 CONSTRAINT MergesWithKey PRIMARY KEY (Sea1,Sea2) );

CREATE TABLE located
(City VARCHAR2(35) ,
 Province VARCHAR2(32) ,
 Country VARCHAR2(4) ,
 River VARCHAR2(20),
 Lake VARCHAR2(25),
 Sea VARCHAR2(25));
```

From *mondial-rel-facts.flp*, a file *mondial-inputs.sql* containing `INSERT` statements for SQL has been generated by a perl script. Then, the ORACLE database is generated by running *mondial-schema.sql* and *mondial-inputs.sql*.

7 Mapping to an XML Model

The MONDIAL database is also available in XML [XML98].

7.1 A Flat XML Model

With the sys.theOM.saveAs@(<filename>) command, FLORID provides a means for dumping the current object-manager object by object. This output has been transformed by a simple perl script into a flat XML file *mondial-flat.xml* with the following DTD *mondial-flat.dtd*:

```
<!ELEMENT mondial (country|city|prov|org|continent|
                   river|lake|sea|mountain|desert|island)*>
<!ATTLIST mondial restriction CDATA #IMPLIED>
<!ELEMENT country (name|car_code|total_area|population|
                   encompassed+|border*|capital|
                   population_growth?|infant_mortality?|
                   gdp_total?|gdp_ind?|gdp_agri?|gdp_serv?|
                   inflation?|indep_date?|government?|
                   ethnicgroup*|religion*|language*|
                   adm_div*|main_city*)>
<!ATTLIST country id ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT car_code (#PCDATA)>
<!ELEMENT total_area (#PCDATA)>
<!ELEMENT population (#PCDATA)>
<!-- note that population is also a
     subelement of province and city -->
<!ATTLIST population year CDATA #IMPLIED>
<!ELEMENT capital EMPTY>
<!ATTLIST capital ref IDREF #REQUIRED>
<!ELEMENT population_growth (#PCDATA)>
<!ELEMENT infant_mortality (#PCDATA)>
<!ELEMENT gdp_total (#PCDATA)>
<!ELEMENT gdp_ind (#PCDATA)>
```

```
<!ELEMENT gdp_agri (#PCDATA)>
<!ELEMENT gdp_serv (#PCDATA)>
<!ELEMENT inflation (#PCDATA)>
<!ELEMENT indep_date (#PCDATA)>
<!ELEMENT government (#PCDATA)>
<!ELEMENT encompassed (#PCDATA)>
<!ATTLIST encompassed continent CDATA #REQUIRED>
<!ELEMENT ethnicgroup (#PCDATA)>
<!ATTLIST ethnicgroup name CDATA #REQUIRED>
<!ELEMENT religion (#PCDATA)>
<!ATTLIST religion name CDATA #REQUIRED>
<!ELEMENT language (#PCDATA)>
<!ATTLIST language name CDATA #REQUIRED>
<!ELEMENT border (#PCDATA)>
<!ATTLIST border country IDREF #REQUIRED>
<!ELEMENT main_city EMPTY>
<!ATTLIST main_city ref IDREF #REQUIRED>
<!ELEMENT adm_div EMPTY>
<!ATTLIST adm_div ref IDREF #REQUIRED>

<!ELEMENT prov (name|in_country|population|area|capital?)>
<!ATTLIST prov id ID #REQUIRED>
<!ELEMENT in_country EMPTY>
<!ATTLIST in_country ref IDREF #REQUIRED>

<!ELEMENT city (name|in_country|in_province?|longitude?|latitude?|
                population*|located*)>
<!ELEMENT in_province EMPTY>
<!ATTLIST in_province ref IDREF #REQUIRED>
<!ELEMENT longitude (#PCDATA)>
<!ELEMENT latitude (#PCDATA)>
<!ELEMENT located EMPTY>
<!ATTLIST located at (river|sea|lake) #REQUIRED
                  ref IDREF #REQUIRED>

<!ELEMENT org (name|abbrev?|established?|seat?|member*)>
<!ATTLIST org id ID #REQUIRED>
<!ELEMENT abbrev (#PCDATA)>
<!ELEMENT established (#PCDATA)>
<!ELEMENT seat EMPTY>
<!ATTLIST seat ref IDREF #REQUIRED>
<!ELEMENT member EMPTY>
<!ATTLIST member type CDATA #REQUIRED
                 ref IDREF #REQUIRED>

<!ELEMENT continent (name)>
<!ATTLIST continent id ID #REQUIRED>

<!ELEMENT river (name|to|length?|in_country*|location*)>
<!ATTLIST river id ID #REQUIRED>
```

```
<!ELEMENT length (#PCDATA)>
<!ELEMENT to EMPTY>
<!ATTLIST to type (river|sea|lake) #REQUIRED
             ref IDREF #REQUIRED>
<!-- better in XML: location country REQ|prov IMPL -->
<!-- similar for cities ?? -->

<!ELEMENT location EMPTY>
<!ATTLIST location country IDREF #REQUIRED
                   prov IDREF #REQUIRED>

<!ELEMENT lake (name|area?|depth?|in_country*|location*)>
<!ATTLIST lake id ID #REQUIRED>
<!ELEMENT depth (#PCDATA)>

<!ELEMENT sea (name|area?|depth?|in_country*|location*)>
<!ATTLIST sea id ID #REQUIRED>

<!ELEMENT desert (name|area?|in_country*|location*)>
<!ATTLIST desert id ID #REQUIRED>

<!ELEMENT island (name|islands?|area?|longitude?|latitude?|
                  in_country*|location*)>
<!ATTLIST island id ID #REQUIRED>
<!ELEMENT islands (#PCDATA)>
```

The XML file *mondial-flat.xml* serves as a base for XSL(T) scripts to transform MONDIAL for other DTDs.

7.2 A Hierarchical XML Model

The DTD. In the sample hierarchical model, where provinces are nested into countries, and cities are nested into provinces with the following DTD *mondial.dtd*:

```
<!ELEMENT mondial (country|org|continent|
                   river|lake|sea|mountain|desert|island)*>
<!ELEMENT country (name|car_code|total_area|population|
                   encompassed+|border*|capital|
                   population_growth?|infant_mortality?|
                   gdp_total?|gdp_ind?|gdp_agri?|gdp_serv?|
                   inflation?|indep_date?|government?|
                   ethnicgroup*|religion*|language*|
                   (prov*|city*))>
<!ATTLIST country id ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT car_code (#PCDATA)>
<!ELEMENT total_area (#PCDATA)>
<!ELEMENT population (#PCDATA)>
<!-- note that population es also a subelement of city -->
```

```
<!ATTLIST population year CDATA #IMPLIED>
<!ELEMENT capital EMPTY>
<!ATTLIST capital ref IDREF #REQUIRED>
<!ELEMENT population_growth (#PCDATA)>
<!ELEMENT infant_mortality (#PCDATA)>
<!ELEMENT gdp_total (#PCDATA)>
<!ELEMENT gdp_ind (#PCDATA)>
<!ELEMENT gdp_agri (#PCDATA)>
<!ELEMENT gdp_serv (#PCDATA)>
<!ELEMENT inflation (#PCDATA)>
<!ELEMENT indep_date (#PCDATA)>
<!ELEMENT government (#PCDATA)>
<!ELEMENT encompassed (#PCDATA)>
<!ATTLIST encompassed continent CDATA #REQUIRED>
<!ELEMENT ethnicgroup (#PCDATA)>
<!ATTLIST ethnicgroup name CDATA #REQUIRED>
<!ELEMENT religion (#PCDATA)>
<!ATTLIST religion name CDATA #REQUIRED>
<!ELEMENT language (#PCDATA)>
<!ATTLIST language name CDATA #REQUIRED>
<!ELEMENT border (#PCDATA)>
<!ATTLIST border country IDREF #REQUIRED>
<!ELEMENT adm_div EMPTY>
<!ATTLIST adm_div ref IDREF #REQUIRED>
<!ELEMENT main_city EMPTY>
<!ATTLIST main_city ref IDREF #REQUIRED>

<!ELEMENT prov (name|population|area|capital?|city*)>
<!ATTLIST prov id ID #REQUIRED>

<!ELEMENT city (name|longitude?|latitude?|
                population*|located*)>
<!ELEMENT longitude (#PCDATA)>
<!ELEMENT latitude (#PCDATA)>
<!ELEMENT located EMPTY>
<!ATTLIST located at (river|sea|lake) #REQUIRED
                  ref IDREF #REQUIRED>

<!ELEMENT org (name|abbrev|established?|seat?|member*)>
<!ATTLIST org id ID #REQUIRED>
<!ELEMENT abbrev (#PCDATA)>
<!ELEMENT established (#PCDATA)>
<!ELEMENT seat EMPTY>
<!ATTLIST seat ref IDREF #REQUIRED>
<!ELEMENT member EMPTY>
<!ATTLIST member type CDATA #REQUIRED
                 ref IDREF #REQUIRED>

<!ELEMENT continent (name)>
<!ATTLIST continent id ID #REQUIRED>
```

```
<!ELEMENT river (name|to|length?|in_country*|location*)>
<!ATTLIST river id ID #REQUIRED>
<!ELEMENT length (#PCDATA)>
<!ELEMENT to EMPTY>
<!ATTLIST to type (river|sea|lake) #REQUIRED
             ref IDREF #REQUIRED>
<!-- better in XML: location country REQ, prov IMPL -->
<!-- similar for cities ?? -->
<!ELEMENT in_country EMPTY>
<!ATTLIST in_country ref IDREF #REQUIRED>
<!ELEMENT location EMPTY>
<!ATTLIST location country IDREF #REQUIRED
                   prov IDREF #REQUIRED>

<!ELEMENT lake (name|area?|depth?|in_country*|location*)>
<!ATTLIST lake id ID #REQUIRED>
<!ELEMENT depth (#PCDATA)>

<!ELEMENT sea (name|area?|depth?|in_country*|location*)>
<!ATTLIST sea id ID #REQUIRED>

<!ELEMENT desert (name|area?|in_country*|location*)>
<!ATTLIST desert id ID #REQUIRED>

<!ELEMENT island (name|islands?|area?|longitude?|latitude?|
                  in_country*|location*)>
<!ATTLIST island id ID #REQUIRED>
<!ELEMENT islands (#PCDATA)>
```

The Transformation Script. The flat XML file *mondial-flat.xml* is converted to a hierarchical one *mondial.xml* by the following XSL(T) script *mondial.xsl* (we use XT [Cla] for this):

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xt="http://www.jclark.com/xt"
                version="1.0"
                extension-element-prefixes="xt">
<xsl:output type="xml" indent="yes" xml-declaration="yes"/>
```

The script applies to the whole database as well as to its restrictions to individual continents. In case of a restriction, the *restrictions* attribute of the `<mondial>` tag contains the continent. The name of the output document depends on these restrictions. If the whole MONDIAL database is handled (i.e, the file is called *mondial.xml*, otherwise, the name is, e.g., *mondial-europe.xml*. The following lines show how to use an `<xsl:variable>` for this.

```
<xsl:template match="mondial">
  <xsl:variable name="output_file_name">
    <xsl:choose>
```

```
      <xsl:when test="@restrictions">
        <xsl:variable name="restrictions" select="@restrictions"/>
        <xsl:value-of select="concat('mondial-',$restrictions,'.xml')"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="'mondial.xml'"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xt:document href="{$output_file_name}">
```

The new document uses the external DTD file *mondial.dtd*. Note that this information is written by `<xsl:text>` with attribute `disable-output-escaping="yes"`, replacing `<` and `>` by the tag-delimiters < and >.

```
  <xsl:text disable-output-escaping="yes">
       &lt;!DOCTYPE mondial SYSTEM "mondial.dtd"&gt;
  </xsl:text>
  <mondial>
  <xsl:apply-templates select="country"/>
  <xsl:apply-templates
       select="continent|org|river|lake|sea|mountain|desert"/>
  </mondial>
  </xt:document>
</xsl:template>
```

for each country, the country information is copied. If administrative divisions are given - referenced inside *adm_div* elements - they become subelements of the country which in turn contain the cities as subelements. Note that the provinces elements are explicitly selected by using the reference to their id. If no administrative divisions are given, cities become direct children of the country:

```
<xsl:template match="country">
  <country>
    <xsl:copy-of select="@id|name|car_code|total_area|population|
                 population_growth|infant_mortality|
                 gdp_total|gdp_ind|gdp_agri|gdp_serv|inflation|
                 indep_date|government|encompassed|ethnicgroup|
                 religion|language|border"/>
    <xsl:choose>
      <xsl:when test="adm_div">
        <xsl:apply-templates select="id(adm_div/@ref)">
          <xsl:with-param name="countrycap" select="capital/@ref"/>
        </xsl:apply-templates>
      </xsl:when>
      <xsl:otherwise>
        <xsl:apply-templates
                   select="/mondial/city[in_country/@ref=current()/@id]">
             <xsl:with-param name="countrycap" select="capital/@ref"/>
        </xsl:apply-templates>
```

```
      </xsl:otherwise>
    </xsl:choose>
</country>
</xsl:template>
```

The following rule is applied to every province selected by the above rule. The province information is copied, and all cities which belong to this province become subelements. Here, the selection is controlled by comparing the *in_Province* attribute of the cities with the id of the province currently under consideration:

```
<xsl:template match="prov">
    <xsl:param name="countrycap"/>
    <province>
      <xsl:copy-of select="@id|name|population|area"/>
      <xsl:apply-templates
                 select="/mondial/city[in_province/@ref=current()/@id]">
           <xsl:with-param name="countrycap" select="$countrycap"/>
           <xsl:with-param name="statecap" select="capital/@ref"/>
      </xsl:apply-templates>
    </province>
</xsl:template>
```

For every city (copied inside the province elements inside the country elements), the city information is copied, and attributes *is_country_cap* and *is_state_cap* distinguish capitals:

```
<xsl:template match="city">
    <xsl:param name="countrycap"/>
    <xsl:param name="statecap"/>
    <city>
      <xsl:if test="@id=$countrycap">
           <xsl:attribute name="is_country_cap">yes</xsl:attribute>
      </xsl:if>
      <xsl:if test="@id=$statecap">
           <xsl:attribute name="is_state_cap">yes</xsl:attribute>
      </xsl:if>
      <xsl:copy-of
           select="@id|name|longitude|latitude|population|located"/>
    </city>
</xsl:template>
```

All other objects are copied:

```
<xsl:template match="continent|org|river|lake|sea|
                     island|mountain|desert">
  <xsl:copy-of select="current()"/>
</xsl:template>
</xsl:stylesheet>
```

Acknowledgements. First, I want to thank the people who created the TERRA database at Karlsruhe the late 1980s, and those who supervised the training in summer 1992, especially the tutor of our group, Dietmar Kottmann.

In 1997, I got the TERRA database from Birgitta König-Ries who supervised the database training at Karlsruhe University at this time. Concerning Freiburg, I want to thank my former colleague Rainer Himmeröder who implemented large parts of FLORID, especially the Web access features (with a *Studienarbeit* by Bernhard Seckinger), and who was a great help when organizing the ORACLE training together with Stefan Mayer in winter term 97. Additionally, I want to thank the student members of the ORACLE training team of winter 1998/99, Erik Behrends, Aldar Dougarjapov, Marco Koch, and Bülent Yilmaz. The data has been completed by Jessica Spengler, using a *Kümmerly and Frey* atlas which has been provided by the library of the geographics department. I also want to thank the leader of the F-Logic team, Georg Lausen, and the former team members Paul-Thomas Kandzia and Bertram Ludäscher.

I also want to thank those unknown people at CIA who collected and still maintain the Factbook for putting it on the Web. Finally, I want to thank Johan van der Heijden for allowing me to use the GlobalStatistics pages.

Available Files. The sources and results of this case study are freely available for educational purposes:

- The FLORID system:
  `http://www.informatik.uni-freiburg.de/~dbis/florid`.
- The MONDIAL Case Study:
  `http://www.informatik.uni-freiburg.de/~may/Mondial`.
  - The F-Logic programs for information extraction from the Web: `mondial-{cia|gs|orgs|qiblih|terra|codes}.flp` and the extracted data `mondial-{cia|gs|orgs|qiblih|terra|codes}-facts.flp`. Note that the programs are designed for the 1996 versions of the data sources. Depending on changes of the sources, adaptations could be necessary.
  - The cleaned F-Logic database: `mondial-oo-facts.flp`.
  - The Datalog database: `mondial-rel-facts.flp`.
  - The ORACLE database: `mondial-schema.sql` and `mondial-inputs.sql`.
  - The XML files: `mondial-flat.dtd` and `mondial-flat.xml` for the flat model, the XSL script `mondial.xsl` for the transformation from the flat model to the hierarchical model, and `mondial.xml` and `mondial.dtd` for a hierarchical model.
- The Freiburg ORACLE training:
  Contact `dbis@informatik.uni-freiburg.de`.

References

[Cla] J. Clark. XT: an implementation of XSL Transformations. `http://www.jclark.com/xml/xt.html`.

[DR90] M. Dürr and K. Radermacher. *Einsatz von Datenbanksystemen.* Springer Verlag, 1990.

[HLLS97] R. Himmeröder, G. Lausen, B. Ludäscher, and C. Schlepphorst. On a Declarative Semantics for Web Queries. In F. Bry, K. Ramamohanarao, and R. Ramakrishnan, editors, *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1341 in LNCS, pp. 386–398, Montreux, Switzerland, 1997. Springer.

[HLLS98] R. Himmeröder, G. Lausen, B. Ludäscher, and C. Schlepphorst. FLORID: A DOOD-System for Querying the Web. In *Demonstration Session at EDBT*, Valencia, Spain, 1998.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[LHL+98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. *Information Systems*, 23(8):589–612, 1998.

[May99] W. May. Modeling and Querying Structure and Contents of the Web. In *Workshop on Internet Data Modeling (IDM'99), Proc. DEXA 99 Workshops*, pp. 721–725. IEEE Computer Science Press, 1999.

[May00] W. May. An Integrated Architecture for Exploring, Wrapping, Mediating and Restructuring Information from the Web. In *Australasian Database Conference (ADC 2000).* IEEE Computer Science Press, to appear, 2000.

[MHLL99] W. May, R. Himmeröder, G. Lausen, and B. Ludäscher. A Unified Framework for Wrapping, Mediating and Restructuring Information from the Web. In *International Workshop on the World-Wide Web and Conceptual Modeling (WWWCM)*, number 1727, pp. 307–320, 1999.

[Sec98] B. Seckinger. Transformation von SGML-Dokumenten in FLORID-Datenbanken. Studienarbeit, Universität Freiburg, 1998.

[XML98] Extensible Markup Language (XML). `http://www.w3.org/XML/`, 1998.