A Rule-Based Data Standardizer for Enterprise Data Bases *

Abhik Roychoudhury I.V. Ramakrishnan Terrance Swift

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{abhik,ram,tswift}@cs.sunysb.edu

Abstract

Whenever a database permits textual entry of information — for example when data is copied from a paper form — the database is likely to contain duplicates and inconsistencies. These duplicates must be removed and inconsistencies resolved in order to mine the data or to use the data for decision support. We term the domain-specific solution to duplicate and inconsistency removal *data standardization*. In this paper, we describe a *Name-Address Standardizer*, one of a series of standardizers that have proven critical in creating a new enterprise-level database for the U.S. Customs Service. The standardizers were used to clean several legacy databases. These standardized databases were combined into a central database for which data is now standardized upon input.

In practice, a standardizer uses techniques both from natural language analysis and from rule-based expert systems. As a result Prolog is highly suitable as a basis for standardizers. All Customs standardizers were written almost entirely in Prolog and constitute a large programming effort: the Name-Address Standardizer contains about 100,000 lines of code, including generated parse tables and a fact base.

1 Introduction

Most large commercial databases are rife with inconsistencies and duplicate information. While there is little publically-available documentation to support this claim — businesses are reluctant to parade their dirty data in public — most programmers recognize this from their own experience. As one instance, a database for a well-known marketing firm had, at last count, seven different values in their sex field. As a second instance, consumers

*This paper reflects the opinions of the authors only, and does not represent policy of the U.S. Customs Service.

routinely receive duplicate catalogs addressed to variants of their own names. Duplicate information costs money to store and mail. Inconsistent or duplicate data either prevents decision support, or undermines its results. Querying such data can become complex and the cost of updating programs that manipulate dirty data can make an organization less adaptive to change.

There are many reasons for dirty data. Enterprises often use network or hierarchical databases management systems which may require data to be stored redundantly for efficiency and which do not provide high-level consistency controls. Even when current-generation relational databases are used, their schemata is often derived almost directly from that of a legacy hierarchical or network database system, and may preserve redundancy of information. Even if the schema is properly redesigned, the new database often inherits dirty legacy data, prohibiting the use of many types of integrity constraints and of consistency preserving triggers.

We term methods that addresses inconsistent or dirty data as *standardization* methods. Prolog itself is a superb means of creating standardization tools, and this paper discusses one such tool, a standardizer for names and addresses found within textual strings. This name-address standardizer is written almost entirely in Prolog. It was originally developed in XSB [5] for the U.S. Customs Service, and standardizes data for all imports coming into the U.S. in real time for a large Enterprise Database. The U.S. Customs Service name address standardizer is over 100,000 lines of code (see Section 3.3.1) and is currently being recreated at Stony Brook to standardize data for a large Manhattan investment bank.

Standardizers can be seen as supporting technology for *mediators* funded by the American Department of Defense over the past several years. While mediators attack the problem of combining heterogeneous databases, they require standardization methods: to remove duplicate data and to resolve inconsistent data.

This paper discusses the high-level architecture of the name-address standardizer and provides examples of its use. This structure of this paper is as follows. Section 2 overviews the use of the name-address standardizer and of Prolog in general in U.S. Customs Service; Section 3 discusses the high-level architecture of the name-address standardizer; and Section 4 discusses its functionality from a user's perspective. Finally, Section 5 discusses future directions for standardization technology currently undertaken at Stony Brook.

2 Data Standardization at the U.S. Customs Service

In 1993, around 20 million entries were filed for cargo imported into the United States, through about 100 official ports of entry. Accurate accounting of imports is important for three reasons. First, cargo — whether it is cocaine or mongooses — may be illegal to import. Secondly, even shipments of legal material are subject to quotas and duties. Finally, from a more general perspective, the number and kind of imports is valuable as economic data: vague or inconsistent information about imports lessens the quality of that data.

In 1991, the U.S. Customs Service began development of a series of expert systems to help their inspectors in prioritizing what shipments to inspect, and to facilitate passage of low-risk imports into the country. One of these, CCTIS [6] has been running continually in the two largest sea ports, Newark and Los Angeles since 1992. In 1993, a project was begun to combine with CCTIS another expert system to form the consolidated *Automated Targeting System (ATS)*. ATS is central to Custom's targeting efforts; it processes shipments and assesses their risk in pseudo-real time. ATS has been tested at the U.S. Customs Data Center, has moved into production, and is now being installed in field offices.

In order to understand ATS and the role Prolog plays to support it, we present the two major sources of information to Customs about shipments: *manifests* and *entries*. Each carrier that transports cargo to the country must submit a manifest to Customs. This manifest contains a collection of *bills of lading* each of which:

- States the port of lading of the commodity, and the itinerary of the vessel.
- Contains information about the *shipper*, *consignee* and *notify party* for all cargo.
- Textually describes the cargo, its destination, weight, etc.
- Provides a transcription of shipping labels on each container.

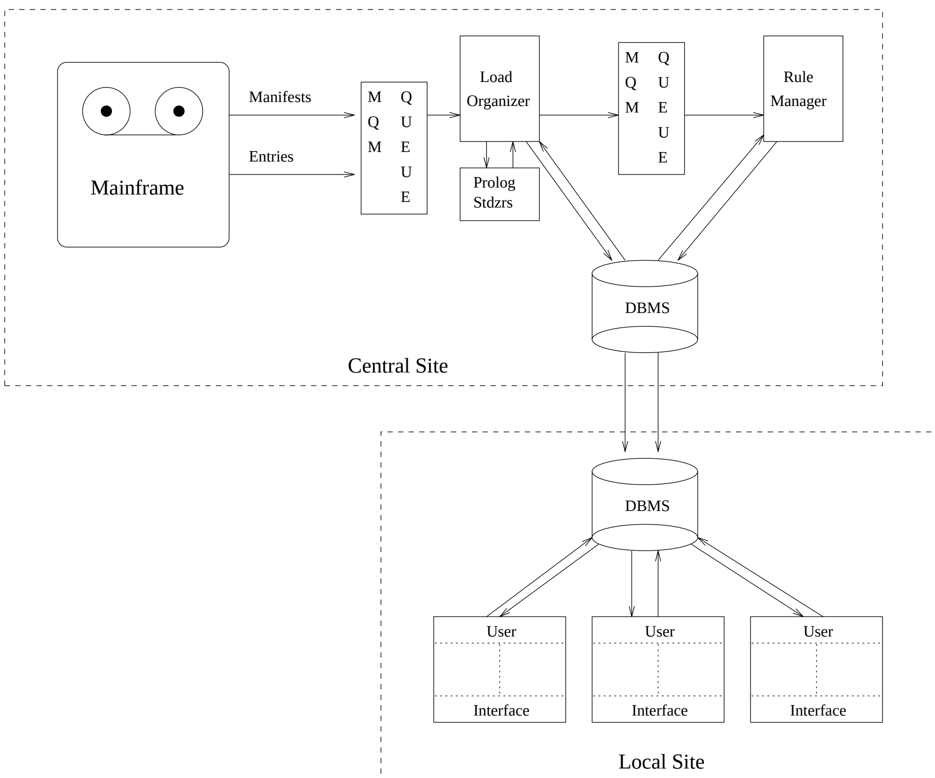

Figure 1: Top-level architecture of ATS

The manifest is usually delivered to Customs before the arrival of the voyage it describes. While it contains a few formatted fields, information in the manifest is mostly free text. Meanwhile, the U.S. Customs Service also receives *entry* information about the cargo from importers. A *filer*, usually a customs broker or agent, files a *cargo release form* stating the nature of the cargo and that the consignee is ready to receive it. This form covers most of the information in the manifest, but is compiled by different sources for different purposes. The data quality in entry forms is better than in manifests, although information from free text fields is still needed.

To sum up, the data about cargo comes to Customs both from filers and from shippers. Data about the same shipment may arrive asynchronously, and critical parts of this data reside in unformatted free text. Furthermore, the 20 million or so shipments may expand into about 100 million customs forms which need to be processed each year. A range of computational techniques are needed to wade through this morass of data, and ATS is crucial to extracting and interpreting the information, and in determining whether the sources are consistent. Figure 1 presents the high-level architecture of ATS. As data is entered into Customs' central mainframe, it is sent through an ESCON channel to an IBM SP2 running AIX. The data is entered onto an MQM queue and read by a Load Organizer, written mostly in C++, which performs a number of functions including correlating bills with entries (a task which is often not done on the mainframe) and updating the AIX database. Most importantly from our perspective, the load-organizer calls a series of Prolog standardizers.

- A *conveyance standardizer* that standardizes textual information about ship names, voyage numbers, flight numbers, etc.
- A *cargo standardizer* which standardizes cargo descriptions as well as shipping labels from cargo containers.
- A *port* standardizer which standardizes information about the itinerary of the voyage or flight.
- The *name-address standardizer* which standardizes name and address fields in manifests and entries.

This paper concerns itself mostly with the name address standardizer which is by far the most sophisticated of the four standardizers, However, all standardizers share code for the tokenization phase and the bottom-up parsing phase (Section 3).

The standardized and correlated data from the load organizer is output to another MQM queue and read by one of a series of rule managers, which evaluate a shipment's risk by executing a series of 100-200 Prolog rules as well as a neural network. At periodic intervals, snapshots of the central database are downloaded into local field offices, where inspectors view the results of the system through PC-based user interfaces.

While this overview has greatly simplified the actions and architecture of ATS, it demonstrates that Prolog plays an essential role in a major information management systems for the U.S. Customs Service. The overview also highlights the central role of standardization: because a large amount of bill and entry data is textual, it is critical to understand this data to support the reasoning in ATS.

3 Standardizer architecture

Standardizing names and addresses of high-quality data is not difficult. For instance, if the delimiter `|` denotes a carriage return, extracting the company name, and address from the following string is simple:

```
ZZZ AUTOPARTS INC | 192 WASHINGTON STREET | EL SEGUNDO CA
```

The problem begins to be non-trivial when carriage returns have no relation to the format of the entity string or may even split tokens, as with:

```
ZZZ AUTOPARTS INC | 192 WASHING | TON STREET  EL SEGUNDO CA
```

Furthermore, the entity name and address may be buried within text,

```
CONSIGNED TO THE ORDER OF CITIZENS FIRST NATIONAL BANK  201-342-0096
```

or may be strung together,

```
1)HENRY I SALUS INC.,  2)RICHARD STACK LTD.,  C/O SEAWIDE BROKERS,ONE
WORLDTRADE    CENTRE,SUITE 2606, NEW YORK,   N.Y.10048,  U.S.A.
```

To support standardization of such data, the architecture of the name-address standardizer resembles that of natural language parsers and consists of four stages:

- An initial **tokenization** phase which converts the free text record into a stream of tokens.
- A **bottom-up parse** which corrects spelling of tokens and is responsible for grouping designated token sequences to supertokens.
- A **top-down frame-oriented parse**, which has been implemented using Definite Clause Grammars (DCGs).
- A final **post-processing** phase which corrects badly parsed entities and handles inconsistent or missing data.

3.1 Bottom-up Parser

The bottom-up parse is responsible for any correction and grouping of tokens that can occur without knowing what portion of the string — say, name or address — is being parsed. The bottom-up parse performs several functions:

- *Explicit Translation.* For instance, translating keywords designated in foreign languages such as `'AEROPORTO'` to `'AIRPORT'`;
- *Correcting Mispellings* such as correcting `'WISCONSON'` to `WISCONSIN'`;

- *Supertokenization* of sequences of tokens. One example of this is grouping the sequence `'SALT','LAKE','CITY'` into `'SALT LAKE CITY'` a town in Utah. If these tokens were not grouped, later stages of the parser would have to avoid recognizing `'LAKE CITY'`, a town in Pennsylvania, as the city field. We call this grouping *supertokenization.*

- *Correcting Line Breaks* such as correcting `'WASHING', | , 'TON'` to `'WASHINGTON'`, where `|` denotes a line-break carriage return pair.

3.1.1 Explicit Translation and Correcting Mispellings

From a procedural level, correcting mispelled tokens is done in the same manner as explicit translation. One recurses through a list of tokens transforming tokens that match the first argument of a Prolog table, and taking no action on other tokens. The Prolog table has entries of the form:

```
correct('AEROPORTO','AIRPORT').
correct('WISCONSON','WISCONSIN').
```

While procedurally simple, this solution is brittle: for instance the preceding table will not correct the token 'WISCONSEN' to 'WISCONSON'. Automatic generation of code to correct mispellings is needed to address this brittleness and is discussed in Section 3.1.3.

3.1.2 Supertokenization and Correction of Line Breaks

Bottom-up parsing is easily done in Prolog [3]. As an example, consider that San Francisco and San Lius Obispo are towns in California, while San Luis Potosi is a town (and a province) in Mexico. A simple approach to supertokenizing these input strings could consist of the rules:

```
cf_trans(['SAN','FRANCISCO'|Tail],Tail,'SAN FRANCISCO').
cf_trans(['SAN','LUIS','POTOSI'|Tail],Tail,'SAN LUIS POTOSI').
cf_trans(['SAN','LUIS','OBISPO'|Tail],Tail,'SAN LUIS OBISPO').
```

In the above representation, the functor `cf_trans` denotes that a “context free” translation is specified. The first argument is the input list, the second is the remainder of the input list after token recognition, and the third is the recognized token.

The above form is correct, but inefficient. To improve indexing and reduce shallow backtracking, we factor it into a *parse trie* as shown by the code:

```
cf_trans('SAN',[H|T],Tail,Result):-
        cf_trans_san(H,T,Tail,Result).
cf_trans_san('FRANCISCO',Tail,Tail,'SAN FRANCISCO').
cf_trans_san('LUIS',[H|T],Tail,Result):-
        cf_trans_sanluis(H,T,Tail,Result).
cf_trans_sanluis('OBISPO',Tail,Tail,'SAN LUIS OBISPO').
cf_trans_sanluis('POTOSI',Tail,Tail,'SAN LUIS POTOSI').
```

All entries that share the token `'SAN''` are grouped into the first rule which then calls `cf_trans_san`. Generation of the above rules is performed automatically by the parse-table generator (Section 3.1.3). We note in passing that creation of a factored parse trie can be seen as an instance of the general problem of factoring clause heads, and has recently been explored fully in [1].

Line breaks in the middle of words are handled similarly, by creating a parse table entry such as

```
cf_trans(['THAI','|','LAND'|Tail],Tail,'THAILAND').
```

whose factored form is then added to the parse trie.

3.1.3 Automatically Generating Parse Tables

Clearly, automatic generation of parse tables is critical for the practical use of Prolog in supertokenization and in correcting mispellings and line breaks. The bottom-up parser makes a first pass at correcting context-insensitive mispellings by generating mispelling tables. Context-sensitive mispelling checks are handled in the later stages of top-down parse and post-processing.

Mispelling tables are generated using the concept of *minimum edit distance*. Minimum edit distance algorithms generally define as the distance between two strings as the number of insertions, deletions or replacements needed to transform one string to another. Indexing into parse tables is necessary for efficient bottom-up parsing. In this stage we therefore statically generate a set of possible mispellings that have edit distance 1 from a keyword, using what is sometimes called a *reverse minimum edit distance algorithm* [4]. Figure 2 provides a portion of the mispelling table for the token 'CORPORATION'. The mispelling table contains truncations of a keyword that have five letters or more, followed by deletions of a single token, followed by double occurrences of a token, followed by inversions of two tokens. Clearly the number of mispellings generated is linear in the length of the keyword.

The generation algorithm is complicated by the fact that that two different keywords may have common mispellings: both the city 'SOUTHFIELD' in Michagan, and the shipping line 'SOUTHFIELD' may truncate to 'SOUTH'. Since all three of these tokens are keywords, generating their mispellings may lead to incorrect results. As a result the parse table generator strips out those mispellings which may transform keywords or which may conflict with other transformations.

Corrections for carriage returns and supertokenization can be generated in a similar manner. Each of these generated predicates are then factored into parse tries as discussed above. At execution time, the bottom up parser first corrects carriage returns, then corrects mispellings, then supertokenizes, and finally performs explicit translation.

3.2 Top-down Parser

The goal of the top-down parser is to fill up an *entity frame* which in our name-address domain is abstractly represented by the Prolog term:

```
correct('CORPORATI','CORPORATION').
correct('CORPORA','CORPORATION').
correct('CORPO','CORPORATION').
correct('CORPORATIN','CORPORATION').
correct('CORPORAION','CORPORATION').
correct('CORPOATION','CORPORATION').
correct('CORORATION','CORPORATION').
correct('CRPORATION','CORPORATION').
correct('CORPORATIONN','CORPORATION').
correct('CORPORATIION','CORPORATION').
correct('CORPORAATION','CORPORATION').
correct('CORPOORATION','CORPORATION').
correct('CORRPORATION','CORPORATION').
correct('CCORPORATION','CORPORATION').
correct('CORPORATOIN','CORPORATION').
correct('CORPORTAION','CORPORATION').
correct('CORPROATION','CORPORATION').
correct('COPRORATION','CORPORATION').
correct('OCRPORATION','CORPORATION').
correct('CORPORAT','CORPORATION').
correct('CORPOR','CORPORATION').
correct('CORPORATIO','CORPORATION').
correct('CORPORATON','CORPORATION').
correct('CORPORTION','CORPORATION').
correct('CORPRATION','CORPORATION').
correct('COPORATION','CORPORATION').
correct('ORPORATION','CORPORATION').
correct('CORPORATIOON','CORPORATION').
correct('CORPORATTION','CORPORATION').
correct('CORPORRATION','CORPORATION').
correct('CORPPORATION','CORPORATION').
correct('COORPORATION','CORPORATION').
correct('CORPORATINO','CORPORATION').
correct('CORPORAITON','CORPORATION').
correct('CORPOARTION','CORPORATION').
correct('COROPRATION','CORPORATION').
correct('CROPORATION','CORPORATION').
```

Figure 2: Portion of the Mispelling Table used by the Bottom-up Parser

frame(Type, entity(Name,Title,Rest),
address(Room,Building,Street,PoBox,Town,State,Country,Zip,Rest),
Telephone,Attention,Other)

For instance the token string

```
XYZ INC ATTENTION MANUFACTURING DEPARTMENT 4 GATEWAY AVENUE MAPLE NC 27956
```

Would be parsed as

```
Entity:  (base)
         Name:  XYZ
         Title:  INC
Address:
         Attn:  MFRG DEPT
         Str/Dist:  4 GATEWAY AVE
         Town:  MAPLE
         Town:  NC
         Zip:  27956
```

The top down parser is structured as an LL(K) parser and coded using DCGs. Effectively, each stage of the parse is associated with a given part of the frame into which tokens are placed by default. In the above example, `XYZ` is not a recognized keyword, so the top-down parser begins by assuming that it is parsing an entity name. When the parser encounters the token INC it recognizes that INC may constitute the end of an organization name. Next, the parser encounters the token ATTN (transformed from ATTENTION by an explicit transformation in the bottom-up parse). By default it then enters a state in which it adds unknown tokens to the Attention field, and remains in that state until it hits the number 4 which is the first token of various address elements, including street number. Parsing continues until the entire string has been consumed, and in this case, requires no post-processing.

The top-down parse also performs context-sensitive correction of mispellings. Figire 3, illustrates this type of correction in the context of parsing of post office boxes. BP is a French abbreviation (*boite postale*) sometimes used for post office boxes. If explicit transformation of the token BP into the more common 'POB' were done in the bottom-up parsing stage, it is likely that the name of the company British Petrolium, which sometimes uses the acronym B.P. would be improperly standardized. To prevent this possibility, the mispelling is corrected by the portion of code that recognizes post office boxes.

```
pobox(['POB',Number|Rest]) -->
        pobox_1,!,
        box_desig(Number,Rest).
pobox(['GEN DELEVERY']) -->
        ['GENERAL'],['DELIVERY'],!.

pobox_1 --> ['POB'],opt(['BOX']),!.
pobox_1 --> ['PF'],!.
pobox_1 --> ['POST'],opt(['OFFICE']),opt(['BOX']),!.
pobox_1 --> ['BOX'],!.
pobox_1 --> ['CP'],!.
pobox_1 --> ['BP'],!.
pobox_1 --> ['BX'],!.
pobox_1 --> ['POSTAL'],!.
pobox_1 --> ['APARTADO'],opt(['POSTAL']),!.
pobox_1 --> ['B'],['P'],!.
pobox_1 --> ['G'],['P'],!.
pobox_1 --> ['P'],['O'],!.
```

Figure 3: DCG fragment for Recognizing Post Office Boxes in Various Languages

3.3 Post-processing

The top-down parser attempts to disambiguate information by using its present context plus a short lookahead of the input token stream. Such guesses turn out to be wrong in a significant minority of cases, and need to be rectified in the post-processing phase. As an example, consider:

```
ALLIED INDUSTRY PA
```

Perhaps the most natural way for a human to parse this string is to take the organization name as ALLIED INDUSTRY, the city as empty, and the state as Pennsylvania. However, Industry is in fact a town in Pennsylvania, and this information may lead us to conclude that the company name is actually ALLIED. Such post processing is done by rules which have the (somewhat simplified) form

```
post_process_entity_name(ea(Rel,entity(Name,Title,Rest),
                              address(Rm,Bld,Str,Po,City,State,Country,Zp,Rst),
                              Tel,Attn,Flags,Other),
                           ea(Rel,entity(Newname,Title,Rest),
                              address(Rm,Bld,Str,Po,City,State,Country,Zp,Rst),
                              Tel,Attn,Flags,Other) ):-
        is_null(City),is_null(State),
        last_two(Name,City,State,Newname),
        consistent_city_state(Penult,Ult).
```

This rule can be read as follows. If neither a city nor a state were found during the top-down parse, the rule checks to see whether the last the last two tokens of the name field form a consistent city state pair. If this is the case, they are stripped from the entity name and added to the appropriate fields of the address. Abstracting from the foregoing example, the fact that no city had been parsed was used to disambiguate the parse. Indeed, when global information is needed to disambiguate a parse, it is most easily done in the post-processing stage.

The post-processing phase is also responsible for applying consistency checks on the output of the top-down parser. These consistency checks are based largely on the following fact bases:

- 42,000 United States cities with their states, and 5-digit zip codes;
- The 500 largest Canadian cities with their provinces;
- 10,000 additional city-country pairs.

Depending on the fact base used, the post-processing phase can check the validity of a city country pair, a city, state/province and country, or a city, state/province and zip code. If the standardizer does not recognize a valid location it attempts to correct the spelling of the city name using a more aggressive algorithm than permitted in earlier stages. To take a concrete example, if the city name in the parsed output is PITSBURG, the zipcode is 15123, and the country is US, we determine that the city corresponding to zipcode 15123 is PITTSBURGH. If the distance is less than a predefined threshold (which is a function of the string length), the misspelled city PITSBURG is transformed to the correct city, PITTSBURGH. For foreign cities a similar algorithm is used. Suppose LONDO were derived for the city and GB for the country. In this case, the country is known but not the city. To attempt to correct the city, the set of cities known to be in the United Kingdom is obtained from the fact base (e.g. *LONDON, MANCHESTER,..*). If the putative city (LONDO) has a unique minimum edit distance from any city in this set, and moreover if that distance is below a predefined threshold, the city name is corrected. In the case of the truncated token `LONDO` a unique minimum edit distance of 1 is found for `LONDON`, and the token is transformed.

3.3.1 Prolog Code Summary

Table 1 provides a breakdown of generated and handwritten Prolog code as grouped by its function in the name-address standardizer. As an aside we note that the conveyance standardizer, described briefly in Section 1, contains about 15,000 additional lines of Prolog code, mostly in generated parse tables and in a specialized fact base. The cargo and port standardizers each contain about 500 additional lines of Prolog code. All four standardizers share code whenever possible.

Function	Subfunction	Clauses	Lines
Tokenization		94	412
Bottom-up Parse			
	Explicit Translation	2362	3857
	Mispellings	7725	7725
	Carriage Returns	3618	3618
	Spaces	10005	10005
Top-Down		724	2082
Post-processing		604	2838
Domain Information			
	City Information	51898	51898
		1165	1165
	Other	6087	6087
Control and Utilities		727	1345
Total			91032

Table 1: Prolog Standardizer Code

4 Functionality of the Name-Address Standardizer

Below, we illustrate some of the functionality of the name-address standardizer.

Detecting An Entity in a Text String As mentioned in Section 3 the standardizer must recognize an entity name buried within a string, for example:

```
CONSIGNED TO THE ORDER OF      CITIZENS FIRST NATIONAL BANK 201-342-6900
Entity: (base)
     Name: CITIZEN 1ST NATIONAL BANK
Address:
     Telephone: 2013426900
```

This parsing is done by recognizing that the phrase `TO THE ORDER OF` and its variants are commonly used in manifests and entries. This phrase and its variants are transformed into a single token `'ORDER OF'` by the bottom-up parse. During the top-down parse, the DCG productions for entity names take account of `'ORDER OF'` and may restart the parse of an entity name after encountering this token.

As an extreme case, the name-address standardizer may never encounter a true entity name:

```
TO ORDER OF THE HOLDER OF  ORIGINAL THRU B/L NO. PST-22310  (TO THE ORDER OF SHIPPER)

Entity: (base)
Address:
```

Detecting Multiple Entities in a Text String The standardizer also must handle input strings containing multiple names. In a given record, the relationship between the entities may or may not be specified. If specified, the relation may be have the type *care of*, *on behalf of dba* (doing business as) and several others. If the type of relationship is unspecified, it is denoted as a *sequence*. The following example illustrates these concepts.

```
1)HENRY I SALUS INC.,  2)RICHARD STACK LTD.,  C/O SEAWIDE BROKERS,ONE
WORLDTRADE     CENTRE,SUITE 2606, NEW YORK,   N.Y.10048,  U.S.A.

Entity: (base)
     Name: HENRY I SALUS
     Title: INC
Address:
org_type = 1  one_more = 1
Entity: (seq)
     Name: RICHARD STACK
     Title: LTD
Address:
org_type = 1  one_more = 1
Entity: (co)
     Name: SEAWIDE BROKER
Address:
     Room: SUITE 6202
     Str/Dist: 1 WORLDTRADE CTR
     Town: NEW YORK
     State/Province: NY
     Country: US
     Zip: 10048
cityxstatexcountry valid
```

This example illustrates several aspects of standardization. There are three entities in this sequence, and one of the two relations is specified as a *care of* relationship. The address keyword `CENTRE` is standardized to the abbreviation `CTR`. The entity `SEAWIDE BROKERS` has its name changed to reduce duplicates. Because like is always standardized to like, this minor change in spelling will prove harmless.

The standardizer also has knowledge of first names, allowing it to parse out sequences of personal names in addition to sequences of organization names. In the following example, the top-down parser catches the pattern of first name, last name, new first-name, last-name and infers that there are two entities present. In this case, the standardizer also associates the address with each name. We stress that this rule is only one of several for extracting personal names from a sequence.

```
ELANOR SMITH  PETER SMITH  JTWROS APT 4-B  175 W 34TH STREET NEW YORK NY 10022
Entity: (base)                      Entity: (seq)
     Name :                               Name :
        First name     : ELANOR              First name     : PETER
        Last name      : SMITH               Last name      : SMITH
     Title: JTWROS                        Title: JTWROS
Address:                             Address:
     Room: APT 4 B                        Room: APT 4 B
     Str/Dist: 175 W 34TH ST              Str/Dist: 175 W 34TH ST
     Town: NEW YORK                       Town: NEW YORK
     State/Province: NY                   State/Province: NY
     Country: US                          Country: US
     Zip: 10022                           Zip: 10022
cityxstatexcountry valid             cityxstatexcountry valid
```

Using the Fact Base to Infer and Correct Information In addition to inconsistent and duplicate information, missing information is also possible. For instance, the following example shows how the fact base is used to add the proper Canadian Province to a record.

```
QUICK BUILDERS LTD 400 QUEBEC STREET SASKATOON CA
   Entity: (base)
           Name: QUICK BUILDERS
           Title: LTD
   Address:
           Str/Dist: 400 QUEBEC ST
           Town: SASKATOON
           State/Province: SK
           Country: CA
  cityxstatexcountry valid
```

We also note that 'CA' can denote California in addition to Canada. However, the standardizer uses its knowledge that Saskatoon is a Canadian city to disambiguate its use.

In addition, the standardizer also has a capability of correcting wrong country codes. In the following example, CURACAO was improperly entered as a city in Namibia (ISO code NA) rather than in the Netherlands Antilles (ISO code AN).

```
E. MORENO BRANDAO    | |P.O. BOX 3037  CURACAO  | NA
```

```
Entity: (base)
        Name: E MORENO BRANDAO
Address:
        Po: POB 3037
        Town: CURACAO
        Country: AN
```

Standardizing Non-English Data The name-address standardizer makes an attempt to standardize addresses in certain foreign languages, The name-address standardizer is not as powerful for non-English names and addresses as for English. Still, it is used by Customs for all data because standardization generally provides an improvement even when the data is not in English. We provide an example of a successful parse of a Brazilian address. Similar features have been added for French, Spanish, and German.

```
ACOS FINOS PIRATINI S.A. RUA CANCIO GOMES, 127-CX POSTAL 2118  SEDE PORTO ALEGRE RS BR

Entity: (base)
     Name: ACOS FINOS PIRATINI
     Title: SA
Address:
     Str/Dist: RUA CANCIO GOMES 127
     Po: POB 2118
     Town: PORTO ALEGRE
     State/Province:
     Country: BR
org_type = 1
cityxcountry valid
```

While many Pacific rim countries, such as Japan, Korea, China, and Taiwan use English as a commercial language, the problem of transliteration arises. For instance a particular Korean port appears as "Pusan" and "Busan" with equal frequency. The name address standardizer attempts to standardize Korean city names according the the *"McCune-Reischauer romanization"* thereby replacing `'PUSAN'` for `'BUSAN'`, `'TAEGU'` for `'DAEGU'` as in the example below. This transliteration is performed automatically according to phonetic rules.

```
F CHO KWANG LIGHT BULBS IND.CO., 42-32 YIHYEON-DONG,  DAEGU, KOREA

Entity: (base)
     Name: CHO KWANG LIGHT BULBS IND
     Title: CO
Address:
     Str/Dist: 42 32 YIHYEON DONG
     Town: TAEGU
     Country: KR
org_type = 1
```

For cities on mainland China, the standardizer attempts to use the pin-yin romanization, but this is done on a city-by-city basis.

5 Discussion

We discuss two avenues for extending and improving the architecture of the name-address standardizer: context-sensitive correction of entity names, and the incorporation of chart parsing. Both avenues are now under development.

Correction of Entity Names Section 3 outlined an approach in which correction of tokens becomes more aggressive as contextual information is determined during the course of the parse. We may term this approach *context-sensitive correction*, and tests at Customs indicate that the technique works well for domains that are well-known such as cities. In the case of cities, bottom-up parse handles certain errors through its misspelling, carriage return, and supertokenization tables, while the post-processing step handles others through a minimum edit distance match of a city against other cities in its state or country.

To support each of these features, the standardizer relies on fact bases in Prolog. While suitable for domains of tens or hundreds of thousands of strings, the present standardizer architecture cannot easily be extended to maintain a fact base containing millions of strings as would be needed for a comparable level of correction of entity names for Customs. In order to perform context-sensitive correction of entity names, an efficient and robust access to an external database is required. Nearly all commercial and most research Prolog systems offer database interfaces. However, it must be noted that, in general, the quality of these interfaces is inferior to that of the Prolog systems themselves. For instance, the Oracle interface for one well-known commercial Prolog has two serious flaws. First, the interface's mapping of Prolog predicates to database tables is restrictive in that the order of arguments in a Prolog predicate must be the same as the order of fields in the corresponding database table. The result of this restriction is that the Prolog program may need to be substantially rewritten upon a database change. Second, repeated queries to the database are re-parsed each time they are executed leading to a significant loss of speed over Oracle's direct Pro*C interface. Such flaws limit the usefulness of the external database acess.

Addition of Chart Parsing Elements of an address can occur in various conceivable orders, for instance both

`ATTENTION JULIE, ACME STEAMROLLERS, STE 12, 13 WARNER BROS LANE HOLLYWOOD CA`

and

`ACME STEAMROLLERS, ATTENTION JULIE, 13 WARNER BROS LANE STE 12, HOLLYWOOD CA`

are forms of addresses that occur frequently. Using Prolog DCGs to parse such strings leads to code that is relatively difficult to maintain. This difficulty arises because the grammar must have left-recursion eliminated in order to terminate, and for each non-terminal, first and follow elements factored to provide efficiency. Even more seriously, the top-down parser attempts to eliminate ambiguities as it parses in order to prevent inordinate backtracking. In a minority of cases the top-down parse makes the wrong choice in attempting to eliminate these ambiguities and its mistakes are corrected in the post processor (See the example in Section 3.3).

Ambiguity is common in natural language analysis and is traditionally solved using chart parsing [2]. If chart parsing were used by the name-address standardizer, an address could be parsed into all possible elements: street addresses, post-office boxes, telephone numbers, and so on. Consistent sets of these elements could be gathered and prioritized during the post-processing phase. In theory the resulting standardizer would be somewhat more declarative than the present standardizer, and easier to maintain. Tabling in XSB is tightly integrated with the Prolog engine, so that XSB forms a highly suitable platform for experimenting with chart parsing in Prolog. Initial incorporation of tabling within standardizers is encouraging, although a great deal of work remains to fully rewrite the name-address standardizer to use tabling.

Acknowledgements

Recreation of the Customs' name-address standardizer at Stony Brook was partially funded by Multivariate Decision Processes, Stony Brook, NY and by NSF grants CCR-9404921, CCR-9510072, CDA-9303181, CDA-9504275 and INT-9314412.

References

[1] Dawson, S., Ramakrishnan, C. R., Skiena, S., and Swift, T. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems* (September 1996).

[2] Earley, J. An efficient context-free parsing algorithm. *Communications of the ACM 13*, 2 (1970), 94–102.

[3] Gazdar, G., and Mellish, C. *Natural Language Processing in Prolog.* Addison Wesley, 1989.

[4] Kukich, K. Techniques for automattically correcting words in text. *ACM Computing Surveys* (1992), 377–441.

[5] Sagonas, K., Swift, T., and Warren, D. XSB as an efficient deductive database engine. In *Proc. of SIGMOD 1994 Conference* (1994), ACM.

[6] Swift, T., Henderson, C., Holberger, R., Murphey, J., and Neham, E. CCTIS: an expert transaction processing system. In *Sixth Conference on Industrial Applications of Artificial Intelligence* (1994).