TABARI
Textual Analysis by Augmented Replacement Instructions
Version 0.8.4

Philip A. Schrodt
Parus Analytical Systems
Charlottesville, VA 22901
Email: schrodt735@gmail.com

VERSION 0.8.4B3: August 30, 2014

Legal Stuff

Report bugs to: schrodt735@gmail.com

Updated copies of the TABARI program and manual can be found at `http://eventdata.parusanalytics.com/`

Please Cite Program As:

Philip A. Schrodt. 2014. TABARI Version 0.8.4

PDF File Generation Date: August 30, 2014

Contents

1 Introduction **1**

1.0.1 PETRARCH: The successor to TABARI 2

1.1 How TABARI Works 4

1.2 Advantages of machine coding 4

1.3 History 6

1.4 System Accuracy 7

1.5 Downloading and Running Version 0.8 on the Macintosh OS-X Operating System 8

1.5.1 Converting to the Unix file format 9

1.6 Downloading, Compiling and Running Version 0.8 on the Unix/Linux Operating Systems 9

1.7 Windows... 10

1.8 Using TABARI to generate event data 11

1.9 Typeface conventions used in this manual 13

1.10 Definitions 13

1.11 Bugs and Extensions 14

1.12 Suggested Readings 15

1.12.1 KEDS 15

1.12.2 Event Data 15

1.12.3 Computational Methods for Interpreting Text 16

1.13 Additional References 16

2 Generating Data with TABARI **17**

2.1 STEP 1: Locate and reformat a set of machine-readable texts 17

2.2 STEP 2: Develop the initial coding dictionaries 19
2.3 STEP 3: Fine-tune the dictionaries 19
2.4 STEP 4: Autocode the entire data set 20
2.5 STEP 5: Aggregate the data for statistical analysis 20
2.6 Using this manual 21

3 Running TABARI **23**

3.1 Setting Up the Environment 23
3.1.1 UNIX reference sites 24
3.2 Introductory Menu 24
3.2.1 Command line options 25
3.2.2 The Single Most Common Error When Reading Files 27
3.3 Main Coding Menu 28
3.3.1 N)ext 29
3.3.2 P)arsing 29
3.3.3 M)odify 29
3.3.4 R)ecode 29
3.3.5 A)utocode 29
3.3.6 O)ther 30
3.3.7 Q)uit 30
3.4 O)ther Menu 30
3.4.1 B)ackup 30
3.4.2 P)roblem 30
3.4.3 M)emory 31
3.4.4 U)sage 31
3.4.5 eX)it 31
3.4.6 V)alidate [invisible] 31
3.4.7 T)ags [invisible] 31
3.5 Dictionary Modification: M)odify 32
3.5.1 A)ctors and aG)ents 32

3.5.2 V)erbs . . . 33
3.5.3 Nouns and Adjectives . . . 34
3.6 Summary of Keyboard Shortcuts . . . 34
3.6.1 Main Menu . . . 34
3.6.2 A)utocoding Menu . . . 35
3.6.3 O)ther Menu . . . 35
3.6.4 Modify Menus . . . 35

4 Input Files **37**
4.1 .project File . . . 38
4.1.1 $< problemfile >$ and $< errorfile >$ . . . 39
4.1.2 $< coder >$ . . . 40
4.1.3 $< system >$ and $< run >$ . . . 40
4.2 Internal Data Set Documentation . . . 41
4.3 Text Files . . . 42
4.4 Event file . . . 45

5 .Actors, .Agents and .Verbs Dictionaries **47**
5.1 Purpose . . . 47
5.2 Phrase Input Formats for Dictionary Files . . . 48
5.2.1 Codes . . . 48
5.2.2 Stemming . . . 48
5.2.3 Regular Noun and Verb Endings . . . 50
5.2.4 Error Checking . . . 50
5.3 .Actor Dictionary . . . 51
5.3.1 Actor Synonyms and Multiple-line Date Restrictions . . . 52
5.3.2 Nouns and Adjectives . . . 53
5.4 .Agents Dictionary . . . 53
5.4.1 “Former” agents . . . 54
5.5 .Verbs Dictionary File . . . 54
5.5.1 Irregular Verb Forms . . . 56

5.5.2 What is a verb? . . . 56
5.5.3 Pattern Matching in Verb Phrases . . . 58
5.6 Synonym Sets . . . 60
5.7 Alternative Patterns . . . 61
5.7.1 Length Calculations . . . 62
5.7.2 A Note on Compound Verbs . . . 62
5.7.3 Designated Actors . . . 63
5.7.4 Precedence in phrase matching . . . 65
5.8 Default Actor Search Rules . . . 65

6 Special Purpose Codes 67

6.1 Purpose . . . 67
6.2 Null Code [- - -] . . . 67
6.3 Discard Code [# # #] . . . 69
6.4 Complex Code [+++] . . . 69
6.5 Special Purpose Codes for Actors . . . 70
6.5.1 Coded Compound Actors . . . 70
6.5.2 Date Restricted Codes . . . 70
6.6 Special Purpose Codes for Verbs . . . 72
6.6.1 Subordinate Codes . . . 72
6.6.2 Dominant Codes . . . 73
6.6.3 Paired Codes . . . 73

7 Parsing: How TABARI Looks at a Sentence 75

7.1 Purpose . . . 75
7.2 Color-Coded Parse Display . . . 76
7.3 Automatic Input Filtering . . . 78
7.4 Nouns . . . 79
7.5 Parsed Compound Actors . . . 79
7.5.1 Compound Noun and Compound Adjective Phrases . . . 81
7.6 Compound Sentences . . . 82

7.6.1 CODE BY... Command 82
7.7 Coreferencing Pronouns 83
7.8 Elimination of Comma-Delimited Nonrestrictive Clauses 84
7.9 Passive voice 85
7.10 Summary: Parsing 85

8 .Options File **87**
8.1 Purpose 87
8.1.1 .Options Command Notes 87
8.2 SET <parameter> = <value> 88
8.2.1 CODE BY... 88
8.2.2 SET: LABEL = FALSE 89
8.2.3 SET: MINIMUM WORDS = n 89
8.2.4 SET: ACTOR USAGE = OFF
SET: AGENT USAGE = OFF
SET: VERB USAGE = OFF 89
8.2.5 SET: DUP WARNING = FALSE 89
8.2.6 SET: CONVERT AGENTS = TRUE 89
8.2.7 SET: HAIKU = FALSE 90
8.2.8 SET: YYYY = FALSE 90
8.2.9 SET: TIME SHIFTING = FALSE 90
8.2.10 SET: SKIP RECORDS = FALSE 91
8.2.11 SET: FBIS = TRUE 91
8.2.12 SET: MATCH = TRUE 91
8.3 LABEL <code string> = <text> 91
8.4 FORWARD: sequence_format 92
8.4.1 Formatting the Sequence Numbers 93
8.5 DEFAULT: SOURCE [code] <PRIOR> TARGET [code] <AFTER> 94
8.6 COMMA 95
8.7 COMPLEX 97
8.8 NONEVENTS 98
8.9 OUTPUT 98

9 Content Analysis: ISSUES and FREQUENCIES 101

9.1 Introduction . . . 101

9.2 ISSUES . . . 102

9.3 FREQUENCY . . . 104

10 Special Features 107

10.1 Validation . . . 107

10.1.1 Modifying *.options* commands . . . 108

10.2 B)ackup . . . 109

10.3 Coding Nonevents . . . 110

10.4 ATTRIBUTION . . . 111

10.5 TIME SHIFTING . . . 112

A Utility Programs 115

A.1 NEXIS_Filter . . . 115

A.2 ACTOR_FILTER . . . 115

A.3 CountryInfo.txt . . . 116

A.4 NEXIS_VERIFY . . . 117

A.5 KEDS_Count . . . 117

B Project Management Suggestions 119

B.1 Selection, Training and Care of Coders . . . 119

B.1.1 Additional project management suggestions from Joe Pull's *Ode to Coding* . . . 120

B.2 Using TABARI: The Unix approach versus the Swiss Army Knife approach . 121

B.3 Skill sets your project may need . . . 122

B.4 Recommended utilities . . . 123

C High Volume Coding 125

C.1 Input File Management . . . 125

C.2 Anomalous Records . . . 125

C.3 Parallel Processing . . . 126

D TABARI versus KEDS **129**

D.0.1 Further Observations on System Speed: October 2009 131

D.0.2 Further Observations on System Speed: June 2009 131

E Program Release History **133**

E.1 Overviews of Major Releases 135

E.1.1 0.2 (31 March 2000) 135

E.1.2 0.3.10b1 (5 March 2002) 135

E.1.3 0.4.7B1 (14 July 2003) 135

E.1.4 0.7.3b3 (1 July 2009) 136

F Ode to Coding **137**

F.1 Meditations on the Dictionaries 138

F.1.1 The Good 138

F.1.2 The Bad 138

F.1.3 The Ugly 138

F.1.4 The Others 140

F.2 A Painfully Detailed Look at the Coding Process 141

F.2.1 Perfection. 143

F.2.2 Nobody's perfect. 143

F.2.3 When the world falls down around you. 143

F.2.4 When the computer throws up its hands in despair. 145

F.3 The End 148

F.4 Miscellaneous Thoughts 149

Acknowledgements

Over the years, the KEDS project has benefited immensely from a large number of individuals who have worked with the program, identified bugs, and suggested new features. These include:

1990 - 1999 (Keds):

Brad Bennett, Doug Bond, Joe Bond, Shannon Davis, Ronald Francisco, Deborah Gerner, Phillip Huxtable, Tony Nownes, Jon Pevehouse, Julia Pitner, Scott Savaiano, Uwe Reising, and Judy Weddle

2000 - 2012 (Tabari:

Rajaa Abu-Jabr, Abra Bron, Anna Gregory, Dennis Hermrick, Annie Ingham, Erika Hane, Matthias Heilke, Christian Hirschi, Milos Jekic, Baris Kesgin, Bradley Lewis, Peter Piccuci, Lee McMullen, Lauren Prather, Joe Pull, Jomana Quaddour, Vladimir Petroff, Taylor Price, Almas Sayeed, Steve Shellman, Erin Simpson, Marsha Sowell, Sarah Stacey, Brandon Stewart, Dale Thomas, David Van Brackle, and Ömür Yilmaz.

Program development of KEDS and TABARI has been funded by the Political Science and Methods, Measurement and Statistics programs of the National Science Foundation through Grants SES89-10738, SES90-25130 (Data Development in International Relations Project), SBR-9410023, SBE-0455158, SBE-0527564 (Human and Social Dynamics), SES-1004414 and SES-1259190. In 1996 and 2012, development was supported by the Fulbright-Hay Faculty Research Abroad Fellowship Program and in 2012, the Peace Research Center, Oslo. In various years, we received support from the University of Kansas General Research Fund Grant 3500-X0-0038. If you use this program to produce data for a paper or publication, we would appreciate receiving a copy of the paper.

This documentation was typeset using the LaTeXdocument processing system as implemented in the open-source TeXShop software. KEDS was developed in the Think Pascal environment. TABARI was initially developed using the MetroWerks CodeWarrior system; it is now compiled and debugged in the open-source GNU gcc/gdb environment, with some support from Apple's XCode development system. In short, this system was developed and deployed without the use of any Microsoft products.

LIST OF ACRONYMS

AFP	Agence France Presse [news wire service]
ASCII	American Standard Code for Information Interchange
CAMEO	Conflict and Mediation Event Observations (event data coding scheme)
COPDAB	Conflict and Peace Data Bank (event data coding scheme)
DDIR	Data Development in International Relations project
FBIS	Foreign Broadcast Information Service
GUI	graphical user interface
ICPSR	Inter-university Consortium for Political and Social Research (Ann Arbor, Michigan)
IDEA	Integrated Data for Events Analysis (event data coding scheme)
K	Kilobyte
KEDS	Kansas Event Data System
KU	University of Kansas
MB	Megabyte
NEXIS	A data service of Mead Data Central
NSF	National Science Foundation
OS-X	Apple Computer's Unix-based operating system for the Macintosh

PANDA	Protocol for the Analysis of Nonviolent Direct Action (Harvard University)
TEXT	The Macintosh "flat ASCII" text format
UN	United Nations
URL	uniform resource locator (event data coding scheme)
WEIS	World Events Interaction Survey (event data coding scheme)

Chapter 1

Introduction

> "The shortest answer is doing."
> George Herbert

> "If I gave people what they wanted, they would have asked for faster horses."
> Henry Ford

TABARI (Textual Analysis by Augmented Replacement Instructions) is a system for the machine coding of international event data based on pattern recognition. It is designed to work with short news summaries such as those found in the lead sentences of wire service reports. It is a successor to the "Kansas Event Data System" (KEDS) program that was developed by the eponymous project during the 1990s. TABARI has primarily been used to code events using the WEIS (McClelland 1976) and CAMEO event coding schemes (Gerner et al 2004) from the Reuters and *Agence France Presse* news services but in principle it can be used for other event coding schemes and text sources.

Historically, event data have usually been human coded by legions of bored undergraduates flipping through copies of the *New York Times*. Machine coding provides two advantages over these traditional methods:

- Coding can be done more *quickly* by machine than by humans; in particular the coding of a large machine-readable data set by a single researcher is feasible;
- Machine coding rules are applied with complete *consistency* and are not subject to inter-coder disparities caused by fatigue, differing interpretations of the coding rules or biases concerning the texts being coded;

The disadvantages of machine coding are that it cannot deal with sentences having a complex structure and it deals with sentences in isolation rather than in context. TABARI has a "complexity detector" that can divert linguistically complex sentences—for example those containing a large number of verbs or subordinate clauses—to a separate file for later human coding (or at least remove these from the coding stream).

This documentation assumes a basic familiarity with event data coding and is primarily a guide to the use of the TABARI program rather than as an introduction to developing an event

data coding scheme. The documentation also assumes familiarity with the basic operations of the Macintosh OS-X operating system.

1.0.1 PETRARCH: The successor to TABARI

In the early 2014, the TABARI project transitioned to be new coder named PETRARCH `https://github.com/openeventdata/petrarch` and work on TABARI effectively stopped at this point. The codebase for PETRARCH is entirely new, though at the present time [August 2014] the system still uses modified versions of several TABARI dictionaries, but we expect this will gradually change. In addition, the TABARI-formatted dictionaries are not being updated, whereas the PETRARCH dictionaries are being substantially updated to incorporate most current political actors. TABARI is still in active use—sometimes with, sometimes without, attribution—but in general if you are starting a new project, you should probably focus on TABARI, not TABARI: as of June 2014 PETRARCH has all of the functionality of TABARI[1] and has been very thoroughly tested.

The core, and motivating, innovation for the new program—which cascades through the entire system—is that PETRARCH uses fully-parsed Penn TreeBank input (`http://www.cis.upenn.edu/~treebank/`) : its coding is parser-based, whereas the coding of TABARI was largely pattern-based. This has a number of very substantial implications.

First, because TABARI was a pattern-based shallow parser, it could get the right answer for the wrong reason, and at least some of the dictionary entries—in particular those treating nouns as if they were verbs—depended on this. PETRARCH , in contrast, only matches true verbs: `(VP (VB` in the parse tree. While means that a small number of existing patterns that were erroneously based on nouns no longer work, the parser virtually eliminates the problem of noun-verb disambiguation (or, rather, relegates it to whatever parser is producing the Treebank output), which is a vastly more important issue. The problem of *word sense* disambiguation remains, particularly the non-trivial issue of distinguishing verbal from physical sense in words such as "attack," and dictionaries still need to handle this.

Parsed input is, however, typically less robust than pattern-based input, since the addition or deletion of words that seem trivial to a native speaker will sometimes change the parse (which is, of course, itself produced by a very complex program). This has two implications. First, it means that PETRARCH will be more conservative than TABARI, which was one of the motivations for the change, particularly as event data have gained increasing attention in the policy community where false positives are a major concern. Second, while the TABARI dictionaries provide a starting point, they will eventually need to be adapted. That said, there have also been some pleasant surprises where features that had to be dealt with as special cases in TABARI are taken care of automatically in PETRARCH , and the full parts-of-speech markup should ultimately simplify the dictionaries: a large number of the TABARI verb patterns existed solely to handle noun-verb disambiguation and can be eliminated.

Third, switching to one or more open-source parsers—in common with many contemporary projects, we are currently using the Stanford CoreNLP parser (`http://nlp.stanford.edu/software/corenlp.shtml`)—means that we are relegating the parsing to the computational linguists.[2] and more generally to a large community that

[1] As of August 2014 it is still missing the pronoun resolution, as we are still deciding whether we can use the CoreNLP facilities for this. This should be resolved in the near future.

[2] Resist impulse to insert remark here about political scientists writing parsers being similar to [MIT] linguists pontificating on politics...

develops parsers that can produce TreeBank output.[3] This has somewhat simplified the code of the coder, though not dramatically as the quirks of a full parse are, if anything, more complex than those of a pattern-based shallow parse. And the parse doesn't take care of everything: for example comma-delimited clause deletion and passive voice detection are essentially done the same way as in TABARI .

Nonetheless, the shift to Treebank input may allow PETRARCH to be easily adapted to other languages[4] since the TreeBank format is standard across many languages. It will still be necessary to adjust for some of the phrase and word-ordering rules, and of course the dictionaries would need to be translated, but except for passive-voice detection. PETRARCH works only with the Treebank tags, not the content.

Finally, Treebank identifies any noun phrase that could potentially be a political actor, whereas TABARI was restricted to identifying actors that were in the dictionaries. The `new_actor_length` parameter in the PETR_config.ini file allows arbitrary noun phrases to be recorded in the source and target slots of the event data whenever these occur in the subject and object positions of the verb phrase. These phrases can then be processed to extract the high-frequency named entities which are not in the dictionaries.

So what's not to like? Speed. TABARI could code very rapidly, typically around 1,000 to 2,000 sentences per second depending on the dictionaries. PETRARCH currently codes at only about 150 sentences per second, and the CoreNLP system parses at about 2 to 5 sentences per second. Consequently the computational demands are much higher, and high-volume coding requires substantial cluster computer resources. Presumably most of the performance hit in PETRARCH is due to the use of Python rather than C++. Python, however, is far better suited to processing text than C++, so the code is substantially shorter and easier to debug and modify. Python is also much more robust across platforms than C++—C++ proved to be a significant barrier to adoption for some users not familiar with Unix environments—and Python has a much larger, and younger, community of programmers.

A few other major changes:

1. PETRARCH does not use word stemming. Like the later versions of TABARI, it automatically produces regular verb forms and noun plurals, and allows irregular forms to be specified: the current *.verbs* dictionary has all of these.

2. PETRARCH makes much greater use of synonym sets than TABARI , and these are objects in the dictionaries in general, not just used in specific patterns.[5] The PETRARCH verb dictionary has organized around both verb and noun synonym sets derived from WordNet
<http://wordnet.princeton.edu/> and other sources

3. The functions of the TABARI *.project* and *.options* files are now incorporated into the PETR_config.ini file.

4. Earlier versions of TABARI incorporated a text-based user interface for machine-assisted coding and dictionary development. As the complexity of the TABARI dictionaries increased, this gradually broke down, and no comparable facility is planned for the

[3] But would someone please write a high-speed Treebank parser in Python rather than Java? And "high-speed" probably rules out "nltk."

[4] TABARI had been adapted to Spanish a couple of times, and KEDS to German and Spanish.

[5] TABARI version 0.8 finally implemented synonym sets but dictionaries using these were never developed.

PETRARCH program itself, though it would be possible to create one or more standalone programs for this task.

5. The text input format is considerably more complex, contains embedded XML, and follows a true XML structure.

6. The dictionary formats have changed substantially and in the fairly near future will not be compatible with TABARI in either direction.

7. Discard phrases—the TABARI "[###]" codes—are incorporated into a separate dictionary file rather than being part of the '.actors' and '.verbs' dictionaries.

Due to a series of unfortunate events (unrelated to anything involving Penn State[6]), the `http://event data.psu.edu` web site was discontinued, and the repository for the program and dictionaries is now at `http://event data.parusanalytics.com`. PETRARCH , and more generally the EL:DIABLO real-time coding suite of the Open Event Data Alliance are maintained at the GitHub repository `https://openeventdata.github.io/`.

1.1 How TABARI Works

TABARI uses a system of pattern recognition to do its coding. Three types of information are used:

Actors: These are proper nouns that identify the political actors recognized by the system.

Verbs: Because event data categories are primarily distinguished by the actions that one actor takes toward another, the verb is usually the most important part of a sentence in determining the event code.

Phrases: Phrases are used to distinguish different meanings of a verb—for example `PROMISED TO SEND TROOPS` versus `PROMISED TO CONSIDER PROPOSAL`—and to provide syntactic information on the location of the source and target within the sentence.

TABARI relies on *sparse parsing* of sentences—primarily identifying proper nouns (which may be compound), verbs and direct objects within a verb phrase—rather than using full syntactical analysis. TABARI will make errors on complex sentences or sentences using unusual grammatical constructions, but has proven to be quite robust in correctly coding the types of English sentences typically found in newswire reports. In contrast to full-parsing systems, TABARI is very fast—coding thousands of sentences per second—and consequently experimental re-coding of large data sets can be done easily.

1.2 Advantages of machine coding

We originally became involved with machine coding because it is dramatically faster and less expensive than human coding. Once a researcher has established vocabulary lists of

[6]despite its recent history as a magnet for unfortunate events

actors and verb phrases, the only cost involved in generating event data is the acquisition of machine-readable news reports, which are now widely available from data services such as NEXIS and Factiva, can be accessed at most academic institutions. Furthermore, a coding system developed at one institution can be used by other researchers through the sharing of vocabulary lists and coding software: Many of the dictionaries available at the KEDS project[7] web site are the result of multi-institution collaborations, or multiple projects that we have done over the years. The ability to modify the coding system quickly is essential in policy applications, since decision-makers must often deal with situations that may not have been addressed earlier by academic researchers.

In working with KEDS and later TABARI, we discovered additional advantages to machine coding. First, it is free of non-reproducible coding biases. Human coding is subject to systematic biases because of assumptions made by the coders. For example, Laurance (1990) notes that even expert coders at the U.S. Naval Postgraduate School tended to over-estimate the military capability of China because China was a large Communist country. Because most human event coding is done part-time by a rapidly changing group of students over a long period of time, coder biases are difficult to control. In contrast, when machine-coding is used, a set of words describing an activity will receive the same code irrespective of the actors or time period involved. Sparse parsing also tends to make *random* coding errors that a statistical analysis can rectify. in contrast, human coders tend to introduce systematic errors that are much more difficult to correct. Any biases embedded in the machine coding system are preserved *explicitly* in its vocabulary; there is no such record in human coding.

Weber (1990:17) notes, coding reliability in textual analysis consists of three components:

stability: the ability of a coder to consistently assign the same code to a given text;

reproducibility: intercoder reliability;

accuracy: the ability of a group of coders to conform to a standard.

The stability of machine coding is 100% because the machine will always code the same text in the same manner. This is particularly useful when a time series is being maintained for a number of years. Because the patterns used in coding are explicitly specified in the coding dictionaries rather than dependent on coder training, the same rules used in 1992 can be used in 2002. In our experience, the reproducibility of machine coding seems comparable to the inter-coder reliability of humans, and a machine is obviously not influenced by the context of an event or by intrinsic political or cultural biases. The dictionaries used by a coding program may reflect biases, but these will be explicit and can be examined by another researcher; the dictionaries are also applied consistently to all actors and in all contexts. Automated coding is not subject to errors due to fatigue or boredom,[8] and in contrast to human coders, a computer program does not require retraining after a coding vocabulary

Second, it is much easier to experiment with alternative coding rules using machine coding. The COPDAB (Azar 1982) and WEIS (McClelland 1976) event coding schemes are very general and for the most part presuppose a Cold War, Westphalian-Clausewitzian conflict

[7] Throughout this document, "KEDS" refers to the Kansas Event Data System project, which is currently housed at the Center for International Political Analysis in the Policy Research Institute of the University of Kansas. "KEDS " in the "small-caps" font refers to the computer program.

[8] As GENERAL INQUIRER's creator Philip Stone once phrased this issue, "Human coding is so tedious that it will reduce even the most committed post-modernist to pleading for computer assistance."

framework where the nation-state is the primary actor and event data are used to track whether state conflict is likely to escalate to the level of conventional war. This weakens the value of WEIS and COPDAB data when dealing with post-Cold War phenomena such as ethnic conflict, low-intensity conflict, and multilateral intervention. Using a machine-coding system, even a very large data collection such as our Arab-Israeli conflict data set (180,000 events) can be completely recoded in a minute or two. This is impossible with human coded data, which has severely restricted experimentation with new coding schemes. Both the CAMEO and IDEA coding schemes, which have been implemented using automated coding systems, have gone through a number of iterations as they were refined through experience.

1.3 History

The development of the original KEDS program began around 1990 as part of the National Science Foundation's "Data Development in International Relations" project (Merritt, Muncaster and Zinnes 1994). While the KEDS work for the DDIR project included some experimentation with German-language sources and foreign policy chronologies (Gerner et al 1994), most of our development focused on WEIS coding of interactions in the Middle East reported by the Reuters news service in English. We focused on this area both because it is very thoroughly covered in the international press, but also because we were doing field work in the area [9] and could therefore cross-check the validity of event data based on our experience in the region.

The development of the KEDS program benefited tremendously from a collaboration with the Protocol for the Assessment of Nonviolent Direct Action (PANDA) at the Program on Nonviolent Sanctions in Conflict and Defense at the Center for International Affairs at Harvard (Bond, Bennett, and Vogele 1994). This project used KEDS to code a superset of the WEIS categories (160 categories versus the 63 categories in WEIS) that provide far more detail on nonviolent events, substate actors and internal interactions such as strikes and protests. PANDA also coded several contextual variables in addition to the standard date-source-event-target variables of event data. The PANDA work eventually spun off a commercial event-coding operation—VRA, Inc. [`http://vra.com`]—which developed a coding program that used quite different principles than KEDS . The PANDA coding system, with the added collaboration of Craig Jenkins (Ohio State) and Charles Taylor (VPI) morphed into the IDEA coding scheme. Reuters reports dealing with the entire world have been coded for by VRA for 1985-2004; the resulting data set contains about 10,000,000 events and can be downloaded from

`http://gking.harvard.edu/data.shtml`

KEDS was written in the PASCAL programming language and worked only on Apple Macintosh computers. The choice of PASCAL made sense at the time—it was the core language for the Macintosh operating system—but by the late 1990s PASCAL had been largely superceded by the C/C++ programming language and compiler support for the language was dwindling. Furthermore, while KEDS was generally stable from 1995 to 2000, it contained some deep-seated idiosyncrasies that could only be eliminated by completely re-writing the program.

[9] In fact some of the final work on KEDS was done in Ramallah when that city was under a closure in 1996 while we were there on Fulbright grants.

In response to this, TABARI was created in the spring of 2000. It is based on the same sparse-parsing principles as KEDS but is written as "open-source" code in ANSI C++ and was immediately ported to the LINUX and WINDOWS operating systems. The conversion to C++ resulted in a program that was substantially faster than the PASCAL code, reducing the time required to recode a large data set from hours to minutes or even seconds.

TABARI is generally stable and has been used intensively in a number of projects at the University of Kansas Center for International Political Analysis over the past five years. We have periodically added some additional specialized features at the request of external funders, and are slowly eliminating a few remaining quirks, but for the most part are no longer doing active development. Instead, we are producing data sets, which was the reason we got into this project in the first place. Fifteen years ago.

The KEDS project maintains a web site at the URL

```
http://eventdata.parusanalytics.com/
```

At this site you will find the most recent versions of the software and this manual, assorted dictionaries, data sets and utility programs, a FAQ (frequently-asked-questions) section, and copies of papers from our project and related efforts.

1.4 System Accuracy

The accuracy of TABARI depends heavily on the source text, the event coding scheme and the type of event being coded. We have done a variety of different reliability checks using KEDS in early papers and articles (Gerner et al 1992, Schrodt et al 1992, Schrodt and Gerner 1993,1994 , Schrodt 1993). These papers are available on the KEDS web site.

With Reuters lead sentences and the WEIS coding scheme, KEDS assigned the same code as a single human coder in about 75% to 85% of the cases. Approximately 10% of the Reuters leads have a syntactic structure that is too complicated or too idiosyncratic for KEDS to handle properly, although some of the residual coding disagreement comes from ambiguities in the WEIS coding categories themselves.[10] In an experiment where dictionaries were optimized for the coding of a single day of Reuters leads, the PANDA project—using a coding scheme substantially more detailed than WEIS—achieved a 91.7% machine coding accuracy; this probably represents the upper limit of accuracy for Reuters leads and a program using KEDS's sparse parsing approach (Bond, Bennett & Vogele 1994:9). This level of coding accuracy is comparable to that achieved in event data projects using human coders: Burgess and Lawton (1972:58) report a mean intercoder reliability of 82% for eight projects where that statistic is known.[11] In a separate test of the VRA CODER, a system using full parsing,

[10] Examples of sentences that are too complex to code include the following:

```
The United States on Friday dismissed Israel's apparent rejection of an Egyptian
plan for talks with the Palestinians as 'parliamentary maneuvering' and said the
door was not closed to peace.

Resumption of ties between Egypt and Syria may spur reconciliation between Iraq
and Syria, and Syria and the PLO, the Qatari newspaper al-Raya said on Friday.
```

[11] KEDS had a somewhat lower agreement when compared to multiple human coders. In many cases this was

King and Lowe (2003) also found the accuracy of automated coding to be comparable to that of human coders.

Schrodt and Gerner (1993, 1994) assess the face validity of KEDS-generated data for the Middle East, 1982-1993; the time series produced by the program correspond closely to the patterns expected from narrative accounts of the interactions between the actors. In these papers, the KEDS data were also compared to the human-coded WEIS data set for 1982-1991. For almost all dyads, there was a statistically significant correlation between the number of events reported by the two series, as well as the number of cooperative events. On the net cooperation values aggregated using the Goldstein (1992) scale and number of conflictual events there was a statistically significant correlation in about half of the dyads. Many of the differences between the two series appear to be due to the higher density of events in KEDS compared to the *New York Times*-based WEIS: the Reuters series contained, on average, three times as many events as WEIS. The KEDS and WEIS data sets were also used in two statistical studies—one involving cross-correlation and the other spectral analysis—produced generally comparable results, although some idiosyncratic differences are found in specific dyads.

1.5 Downloading and Running Version 0.8 on the Macintosh OS-X Operating System

The Macintosh OS-X versions of TABARI run in the TERMINAL application, which uses a UNIX "command line" interface. Downside: no fancy eye-candy and menus. Upside: this is the standard interface for UNIX research software and it is easily maintained. Converting the 5-year-old LINUX version of TABARI to run in OS-X took only half an hour, and subsequent versions have run without modification in both OS-X and LINUX.

1. Use any browser to go to the current version of TABARI at the `http://eventdata.parusanalytics.com/software.html` site [12]
2. Click the link to download the TABARI.0.8.0B1.zip file.[13] This will probably be automatically decompressed and end up as a folder on your desktop named TABARI, though depending on your browser settings, could end up somewhere else. If it doesn't decompress automatically, double-click the *.zip* file icon.
3. Move the TABARI folder to the Documents folder [14]
4. Start the TERMINAL application (it is in the /Applications/Utilities folder). Set the size of the terminal window to at least 120 columns by 48 rows (just drag the lower right corner; size is shown in the window title)[15]

not because KEDS was coding poorly but because KEDS was more consistent and less likely to miss multiple events in a single story than some of the human coders—in other words, the program was actually doing better than the humans (Gerner et al. 1992, Schrodt, et al. 1992).

[12]This URL will not take you directly to TABARI but you should be able to find it from here. Like everyone, we periodically reorganize the web site and rather than providing a URL that may become out-dated, we are providing a plausible starting point.

[13]If are downloading a later version, make the appropriates changes in the name of the program.

[14]The program can actually be located anywhere; the directory suggestions given here just use ~/Documents/TABARI for simplicity.

[15]If you prefer XTERM to TERMINAL, the program works fine there as well. The width of the window is

5. Enter the command `cd ~/Documents/TABARI`
6. Enter the command `./TABARI.0.8.0B1`
7. You should now see a start-up screen identifying the program; entering <return> will run the "Demo" file, which is already in the folder.[16] The demonstration sentences in TABARI.Demo.Text illustrate a number of features of the system; use the <return> key to advance through the text. The text of the events uses the syntax and verb phrase vocabulary typical of Reuters newswire lead sentences (the noun phrases also are likely to be familiar, but are not typically found in Reuters leads).
8. If you get a message "Permission denied" following the ./, enter the command

   `chmod u+x TABARI.0.8.0B1`

   This adds permission to run the program to the file characteristics.

1.5.1 Converting to the Unix file format

To code real data, move your coding dictionaries and text files into this folder. If these were created in an earlier Macintosh system, they will need to be converted to the UNIX format used by TERMINAL applications. This can be accomplished by

- Using the commerical text editor BBEDIT: open the file, select the `Edit/Document Options...` menu item, under `Line Endings` select `UNIX (LF)` and re-save the file
- Using the free[17] program TEXTWRANGLER 3.1 from those wonderful folks at Bare Bones Software, then follow the instructions above.
- UNIX geeks will prefer using

```
tr '\r' '\n' < mac.txt > unix.txt
```

Further discussion of this issues can be found in section 3.2.2

1.6 Downloading, Compiling and Running Version 0.8 on the Unix/Linux Operating Systems

As noted above, version 0.8 (and future versions) have a code base that compiles in the UNIX gcc environment, which means that the program can usually be compiled and run on any Unix-based machine—we've done this on about half a dozen different systems. That file contains a Makefile that provides the instructions to the command MAKE to compile a version of the program for that machine. In order to work, the machine you are installing on needs

somewhat arbitrary: The program works okay with just an 80-column window, though given contemporary screen sizes there is little reason to be this restrictive. The 48-row size is essential: the ncurses formatting will not work properly with any smaller number.

[16] It is also possible to provide a project file name in the command line; this will skip the start-up screen. See Section 3.2.1 for details.

[17] !—and "free as in beer"

the `gcc` compiler suite and the `ncurses` terminal library, but in our experience most systems have these installed.

You can install this by doing the following:

1. The version of the program on the web site `http://eventdata.parusanalytics.com/software.html` is under the **Linux** heading; it is currently called TABARI.0.8.Linux.zip.

2. Using `sftp` or a file transfer program, upload this to the UNIX machine.

3. Decompress the file using the utility UNZIP, e.g.
   `unzip TABARI.0.8.Linux.zip`
   This will give you a new directory named TABARI.Source.0.8.

4. Move into this directory—`cd TABARI.Source.0.8`—and enter the command `make`. This should compile the program

5. Enter the command `./TABARI.0.8.1B1`
   Note: you may need to adjust the version number.

6. You should now see a start-up screen identifying the program; entering <`return`> will run the "Demo" file, which is already in the folder.

1.7 Windows...

Despite innumerable pleas over the years, we've never found anyone willing to maintain a WINDOWS version of TABARI for more than brief periods of time and because we are a UNIX shop, we haven't made the investment ourselves.[18] Given that it is now possible to get a Macintosh Mini for about $600—this will use the same monitor, keyboard and mouse as your WINDOWS machine—most small projects would probably be better off investing in this minimal Macintosh hardware rather than trying to develop a WINDOWS version. Alternatively, you can run LINUX on your existing hardware, but LINUX presents a more challenging learning curve for most WINDOWS users than does the Macintosh. So, if you are in a WINDOWS environment but have some flexibility, probably the best approach would be to either

- Purchase a Mac Mini;
- Purchase system with LINUX already installed: `http://linuxpreloaded.com/` [accessed Jan-2012] lists a number of vendors;
- Download and install [for free] UBUNTU LINUX (or any other variant of LINUX) on your WINDOWS system;
- Download and install the [free] CYGWIN system for running LINUX applications under WINDOWS .

[18] There is a working version of TABARI 0.6 on the web site, but this is quite dated and does not have several important bug fixes nor newer features such as agents and synsets.

First two options will probably take less time; the second two less money.

Again, if someone would like to volunteer to fully convert the current version to WINDOWS —key additional work needed is to get functionality equivalent to that provided by "ncurses" (or better, like a WINDOWS GUI), the source code, as always, is available on the web site.

Finally, as of 2009, a multi-operating-system, proprietary, Java-based program called JABARI based on principles similar to TABARI—in fact it started life as a clone of TABARI , though subsequently has been very substantially extended—has been developed by the Lockheed-Martin Advanced Technology Laboratory, and they might be willing to license this.

1.8 Using TABARI to generate event data

While the KEDS project has generated and maintained a number of event data sets—see `http://eventdata.parusanalytics.com/data.html`—our primary intention has always been to provide a set of tools that will allow other research projects to produce (and, we would hope, share) their own data sets. Fundamentally, we are interested in creating fishing equipment, not just catching fish.

To date, this effort has been at least partly successful—TABARI has been used by several researchers (usually graduate students; occasionally undergraduate honors students) outside of the University of Kansas to create specialized data sets. It can be done! However, the process is considerably more complicated than downloading a data set from the ICPSR, plugging it into STATA, and running off the 789th statistical confirmation of the democratic peace hypothesis or 897th statistical refutation of the democratic peace hypothesis. Based on these experiences, a few bits of advice are in order:

1. You have to provide your own source texts! Due to intellectual property constraints, we cannot (and do not) provide the text of the news stories used to code the events—you have to acquire these on your own. At most North American research universities, this can be done relatively easily through data services such as NEXIS and Factiva, which are typically available at no cost through your library.

We have developed a number of filter programs that largely automate this downloading process; these are available at `http://eventdata.parusanalytics.com/software.dir/filters.html`. These are production programs and have been used to successfully download hundreds of thousands of stories from a variety of news sources. They work.

They work here. At the University of Kansas. They may not work at your institution because of subtle differences in the way that those data bases are accessed locally. These differences can be as simple as the difference between the use of the "`http:`" and "`https:`" protocols, to more complicated differences involving local caching of downloads.[19] Usually once one gets past the initial problem of connecting to the database, the remainder of the program will work unchanged for the specific news source it was designed for.

But there's more: news source formats change subtly (and without notice) over time, and a program that works fine for 1993-1995 may fail for 1997-1999. Again, these changes are

[19] Example: the Factiva/Reuters database worked identically at the University of Kansas and at Penn State, but the way that the *email systems* at the two institutions handled HTML-formatted texts transmitted from that site was different: KU allowed us to get the HTML directly; Penn State always interpreted it first. So, the formatter had to be rewritten.

usually small and may involve changing only one or two characters in the filter—for example the Reuters news service has variously identified itself as "Reuter" and "Reuters"—but those changes are essential in order for the filter to work correctly. Switching to an alternative news service will always involve changing the filter.

The upshot: in all likelihood you will need to allow for a period of experimentation to get the filtering programs to work. The current programs are written in the PERL text processing language, and you will either need to learn sufficient PERL or find someone who does. The filters are extensively documented internally, and designed to be modified; furthermore the flexibility of PERL makes most modifications quite easy, often just a line or two.

2. Use the production dictionaries, not the dictionaries used in the TABARI.Demo.Project suite! Due to intellectual property constraints, the demonstration file uses simulated news stories, and contains actors such as Gondor, Mordor, Gandalf and Frodo. This is not the actor list you want to be using to code contemporary political events! [20]

The KEDS project has been working on dictionaries for about 15 years and while these are not perfect, they've been subjected to hundreds of hours of work, and are probably a good place to start your development. At the present time, most of the dictionaries are incorporated in the *.zip* files that contain the *data sets*, though at some point we may put them onto a separate section of the web site. In addition to the dictionaries posted on the web site, we've got an assortment of others that have been developed for short-term projects; feel free to send an email asking about these.

The dictionaries are generally regionally-specific (the *.actors* dictionaries definitely; the *.verbs* dictionaries less so), although we have maintained a baseline "standard dictionary" since about 2000. In contrast to traditional human-coded event data, these dictionaries (and the resulting data) have changed almost continuously over the past decade. In the *.verbs* dictionaries, this first involved small additions to the WEIS coding system and then, starting around 2001, the incremental development of the CAMEO coding scheme. *.actor* dictionaries have involved a variety of abbreviations for sub-state and non-state actors and, starting about 2003, replacement of WEIS country codes with ISO country codes. Consequently a dictionary from 2002 will not simply be a subset of a dictionary in 2005.

3. Adjust your expectations to the fact that this is research software, and read the fact-filled manual (also see Appendix B.2 of this document). Microsoft can put 10,000 programmers on a major project, Apple puts 1,000, whereas we've typically got two, both part-time, handling both TABARI and the ancillary software. This software is not as complex as MS-OFFICE (although you will also never see a talking paper clip on your screen), nor as slick as ITUNES; it just does a pretty decent job of generating event data.

While TABARI is open-source, to date we've had very few individuals contributing to the software. But we'd like those contributions!—really! Something you absolutely hate about TABARI?—re-write it! Some feature you really want to see added?—add it! Can't stand the terminal interface?—write a GUI using the X-WINDOWS API. Can't start your day without

[20] This is a joke, right? No, people actually do this. About three years ago we got a call from someone working on a U.S. government contract who was comparing TABARI's effectiveness with that of some generic natural language processing software that one of the Beltway bandit consulting firms had adapted to do event data coding. TABARI was generating almost no valid events on a geographical domain where our work had generated thousands. How could that occur?—seems that in doing this "comparison", TABARI had been run using TABARI.Demo.actors, and, surprise, the conflict involving Elves and Men against the forces of the Dark Tower just wasn't a salient issue. Whether this was the result of rampant stupidity or deliberate deception remains an exercise for the reader—I'd give about 50/50 odds on the two interpretations.

removing viruses and spyware from your computer?—port TABARI to WINDOWS ! In the immortal words of Linus Torvalds, "RTFSC"—read the fact-filled source code.

4. Share your data and papers. In contrast to the thousands of papers that have been published using international economic time series and the various episodic data produced in the Correlates of War framework, event data analysis is still in its infancy. Frankly, it's all still a big experiment, and while it seems fairly likely that event data (or something like it) will eventually prove to be a very effective tool in analyzing political behavior, we're not there yet. We will be happy to post papers using event data (ours or others) on the KEDS web site, and we would really like to post additional data sets.

1.9 Typeface conventions used in this manual

`Menu options`, and `key entries` are in the LaTex ``typewriter'' font. The path to an option within a menu is shown using slashes, e.g.

`A)utocode/D)ate`

File types begin with a period and are in italics, e.g. *.verbs, .actors, .project*. The input file suffixes are those we've used in our project but the program does not actually require them; output file suffixes such as *.probs* and *.complex* are assigned by the program.

File names are in the LaTex "sans serif" font.

COMPUTER PROGRAMS are in the LaTex "SMALL CAPS" font.

`Input text`—usually either dictionaries or source text—is in ``typewriter'' font.

`Output text`—material that TABARI writes to the screen or to a file—is in ``typewriter'' font.

◇ represents a blank

1.10 Definitions

actor refers to any noun phrase in the *.actor* dictionaries.

source refers to the actor in an event data record who initiated an action; in some event data discussions this is called the "actor."

target refers to the actor in an event data record who is the object of an action.

Case-sensitive means that the program differentiates between upper- and lower-case letters. In a case-sensitive command, the keyword **SOURCE** will be recognized but **Source** and **source** would not be recognized.

string refers to any set of consecutive characters.

literal means a string that does not contain symbols that are interpreted by TABARI, for example

`*, $, +, %, |, {, }, ~`

token means a string of characters that has a special meaning to TABARI, for example

`*, $, +, %, <VERB>, <TEXT>, ->`

***x*-delimited** means a string that is between the characters 'x.' For example, in the sentence

`President Clinton, arriving in Dublin, said that US policy...`

the phrase "`arriving in Dublin`" is *comma*-delimited.

ASCII stands for American Standard Code for Information Interchange; it is the standard code through which characters are stored in a computer and in files. For example, A is represented by the number 65; a blank is the number 32 and so forth.

TEXT file means a file that contains only characters, without additional formatting information. Virtually all word processing programs will read and write a TEXT file, but we highly recommend using a text-only editor such as BBEDIT. If you are creating a new file on a general purpose word processor such as MS-WORD, use the `Save As...Text Only` option for any files that will be read by TABARI . These files are occasionally called *flat ASCII*.[21] TABARI uses the UNIX file format, not the older Macintosh file format (see section 1.5.1).

phrase refers to a set of words and tokens in the *.verbs, .actors* or *.options* files.

clause refers to an element of a text sentence that is delimited by either conjunctions or commas.[22]

A **default** is the value that a parameter will have unless it is changed by a command or menu option.

1.11 Bugs and Extensions

PLEASE REPORT BUGS! Versions of TABARI since 0.4 seem to be fairly stable and have been used extensively, but it *may* still contain a bug that periodically destroys the *.verb* list. Keep extra backups of the *.Verb* and *.actor* files; don't depend on TABARI's backups alone! Bug reports can be sent to `schrodt@psu.edu`

This is an on-going project and the dictionaries, in particular, are continually being updated. If you intend to use the program in an actual research project, check the web site for the latest copy of our dictionaries and the program, and a list of other projects that are willing to share their dictionaries.

[21] The files are "flat" because they are not structured with formatting information.

[22] In English grammar, a clause technically is any group of words—a subject and predicate. Allowing for the implied subject in compound sentences of the form

Arafat arrived in Jordan today and met with King Hussein

most clauses parsed by TABARI probably satisfy this criterion—at least in Reuters and AFP leads—but the markers used to delineate the clauses are commas and conjunctions.

Note: Bug reports are absolutely vital in the development of a complex program such as TABARI , so don't be shy! We've only been able to test the program on a small number of hardware configurations and we've done most of our work with the same set of texts and vocabulary lists. If you find a situation where TABARI crashes, or does not behave as described in the manual, please let us know—this might be symptomatic of a critical problem in the code or the documentation.

1.12 Suggested Readings

The annotated bibliography below gives citations to the primary papers from the KEDS project, as well as some surveys of event data analysis and computational methods for processing natural language.

1.12.1 KEDS

Gerner, Deborah J., Philip A. Schrodt, Ronald A. Francisco, and Judith L. Weddle. 1994. "The Machine Coding of Events from Regional and International Sources." *International Studies Quarterly* 38:91-119.

Description of the DDIR-sponsored KEDS research; including tests on German-language sources and a foreign affairs chronology.

Schrodt, Philip A. and Deborah J. Gerner. 1994. "Validity Assessment of a Machine-Coded Event Data Set for the Middle East, 1982-1992." *American Journal of Political Science* 38:825-854.

Statistically compares KEDS-generated data to a human-coded data set covering the same time period and actors.

Schrodt, Philip A. , Shannon G. Davis and Judith L. Weddle. 1994. "Political Science: KEDS–A Program for the Machine Coding of Event Data." *Social Science Computer Review* 12,3: 561-588.

General description of KEDS and an extended discussion of some of the problems encountered coding Reuters.

1.12.2 Event Data

Duffy, Gavin, ed. 1994. *International Interactions* 20,1-2

Special double-issue on event data analysis.

Merritt, Richard L., Robert G. Muncaster, and Dina A. Zinnes, eds. 1994. *Management of International Events: DDIR Phase II.* Ann Arbor: University of Michigan Press.

Reports from the DDIR projects.

1.12.3 Computational Methods for Interpreting Text

Advanced Research Projects Agency (ARPA). 1993. *Proceedings of the Fifth Message Understanding Conference (MUC-5).* Los Altos,CA: Morgan Kaufmann.

Reports from a large-scale ARPA project on developing computer programs to interpret news reports on terrorism in Latin America; these use a variety of different techniques.

Pinker, Steven. 1994. *The Language Instinct.* New York: W. Morrow and Co.

Excellent non-technical introduction to contemporary linguistics; extensive discussion of the problems of parsing English.

Salton, Gerald. 1989. *Automatic Text Processing.* Reading, Mass: Addison-Wesley.

General introduction to the use of computers to process text; covers a wide variety of methods.

1.13 Additional References

King, Gary and Will Lowe. "An Automated Information Extraction Tool For International Conflict Data with Performance as Good as Human Coders: A Rare Events Evaluation Design" *International Organization* 57,3: 617-642.

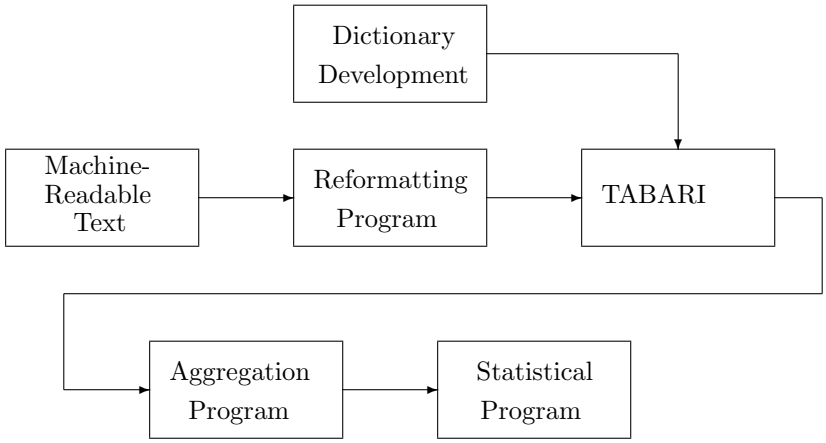
Laurance, Edward J. 1990. "Events Data and Policy Analysis." *Policy Sciences* 23:111-132.

Weber, Robert Philip. 1990. *Basic Content Analysis*, 2d ed. Newbury Park, CA: Sage Publications.

Chapter 2

Generating Data with TABARI

Because both the natural language text *input* and the event data *output* of TABARI are uncommon in the standard statistical data processing environment, working with TABARI requires a few more steps than, say, pulling variables off the Euro-Barometer surveys. The figure below shows what is involved in going from machine-readable text to data that can be analyzed with a statistical program.



2.1 STEP 1: Locate and reformat a set of machine-readable texts

The very first step in doing research with TABARI is finding a source of machine-readable text. This will usually come from an on-line data service or from documents downloaded from the Web. It may also be possible to generate machine-readable texts from printed documents using a scanner and optical character recognition (OCR) software but you should experiment first: some documents are more easily scanned than others.

In all likelihood, the original text will not be in the input format used by TABARI. For example, at one point the text provided by the NEXIS data service originally looked like [1]

```
] ] ]
]                            LEVEL 1 - 5 OF 914 STORIES
] ]             Proprietary to the United Press International 1981
] ]                      <December> 29, 1981, Tuesday, PM cycle
] ]HEADLINE: World News Summary
] ] BODY:
]   Sen. Charles Percy, R-Ill., chairman of the Senate Foreign
]Relations Committee, asked his Israeli hosts to avoid the annexed
]
^
^
]Golan Heights in a helicopter inspection tour today of the tense
]Israeli-<Lebanese>frontier.
]
```

This needs to be converted to the TABARI input format

```
811229  UPI001
Sen. Charles Percy, R-Ill., chairman of the Senate Foreign
Relations Committee, asked his Israeli hosts to avoid the annexed
Golan Heights in a helicopter inspection tour today of the tense
Israeli-Lebanese frontier.
```

We have developed a number of filtering and reformatting programs that remove all of the irrelevant information found in a NEXIS download. These programs are written in PASCAL, C and PERL—all of the recent ones are in PERL—and their source code can be found at the KEDS web site.

However, unless your original text is in exactly the same format as we found when we used the data service—at various times we've worked with NEXIS, Factiva, and FBIS—you will need to write your own filter or modify one of ours. Because machine-readable data are *usually* consistently formatted, this should not be very difficult provided you know (or know someone who knows) a programming language such as C or PERL, but filtering and reformatting are necessary steps before you can start using TABARI. If you aren't able to get a filter/reformatting program written in a programming language, the macro language in Microsoft's *Word* program provides another possibility for reformatting.[2]

[1] Raw NEXIS downloads now consist of HTML files, which are slightly—but only slightly—easier to reformat.

[2] If you have any choice, use PERL: unlike C, PERL is designed for text processing, and we've found that filters written in PERL are about one-tenth the size of those written in C or Pascal, and correspondingly easier to modify

2.2 STEP 2: Develop the initial coding dictionaries

TABARI uses large dictionaries of proper nouns and verb phrases to code the actors and events it finds in the source text. If you intend to code political events, you would probably find it easier to modify existing dictionaries rather than starting with a new dictionary. The TABARI dictionaries are the result of at least five person-years of coding, so a substantial amount of effort has already been invested in developing them. The advantage of this approach is that the existing dictionaries have already identified most of the English vocabulary used in news wire reports, so even if you expect to substantially change the coding scheme, you will know what phrases to expect. A variety of dictionaries can be found on the KEDS web site—these are usually found inside the compressed *.zip* files that contain the data.

If you are developing a dictionary for a new region (or a new time period for an existing region), be sure to check out the ACTOR_FILTER program. This detects proper noun phrases (typically, names) in a set of texts and compares these to an existing *.actors* dictionary; phrases that are not found in the dictionary are written to a file in order of frequency. The standard operating procedure for developing new dictionaries in our project is to first run texts through ACTOR_FILTER, then manually add the most frequent new actors to the dictionaries before coding.

2.3 STEP 3: Fine-tune the dictionaries

With the initial dictionaries incorporated into your system, the next step is fine-tuning—"tweaking"—the phrases to work correctly with your data and coding scheme. This is done by going through a large number of texts and modifying the vocabulary as needed; this process will also give you an indication of the accuracy of the system. Most vocabulary modifications involve the addition of specific individual actors (e.g. political leaders and geographical place names) and the addition of verb phrases describing behaviors specific to the problem you are considering.

Resist the temptation to tweak the vocabulary indefinitely! Tweaking should focus on finding *general* patterns that will occur on multiple occasions in the text, not on expanding the list of phrases to cover every possible contingency. Always remember that coding errors will be only *one source* of noise in your data: The source text is already an incomplete and biased record of the underlying events. The event coding scheme may be incorrectly aggregating some categories of events. Finally, the statistical models in which the data will eventually be used in capture only some of the possible forms of the political relationships.

Even if your coding was somehow "perfect," you are still dealing with a noisy process. Successfully identifying relationships amid that noise comes at the *analytical* stage of the project—both the development of the coding scheme and the statistical analysis—not in the coding stages. Know when the coding accuracy is "good enough," and don't fall into the trap of producing a project that does beer-budget analysis on champagne-budget data. If you can't cope with the fact that probably 15% of your data are erroneously coded, you shouldn't be doing event data analysis.[3] End of sermon.

Detailed and sage advice on the fine art of dictionary development can be found in Joe Pull's *Ode to Coding*, Appendix F.

[3] And this is true whether you are using human or machine coding.

2.4 STEP 4: Autocode the entire data set

Data should be autocoded once the accuracy of the dictionaries has reached a level you are comfortable with. Autocoding will ensure that the coding rules have been consistently applied across the entire data set; even a large data set can be completely recoded in a few minutes. Autocoding also insures that your coding can be replicated by later researchers, as well as extended to new texts at a later date in your own research.

If you cannot get the accuracy level of TABARI to an acceptable level, you still may be able to use the program for routine coding by using TABARI's complexity filter. This will automatically—and systematically—divert to a separate file any texts that appear too complex to machine code (for example texts containing an excessive number of verbs or actors, or containing ambiguous words such as ATTACK or GEORGIA). The complex texts can be processed using human coding, and then the two sets of event data can be merged. Coding with the complexity filter is not completely replicable—and the complexity filter will not catch every sentence that might be coded incorrectly—but it is more efficient and replicable than all-human coding, while being more accurate than all-machine coding.[4]

2.5 STEP 5: Aggregate the data for statistical analysis

TABARI produces standard event data of the form

```
<date> <source code> <event code> <target code> <auxiliary codes>
```

Because event data are an irregular, nominal-measure (categorical) time series, they must be aggregated before they can be used by standard statistical programs such as STATA and R or in graphical displays produced by a spreadsheets: all of these programs expect a regular, interval-measure (numerical) time series. This transformation is usually done by mapping each event code to an interval-level scale (for example, Goldstein 1993), and then aggregating the data by actor-pair and week, month or year using averages or totals.

It is relatively straightforward to do scaled aggregation by scripting the data transformation facilities of a statistical package. We also have developed an aggregation program, KEDS_COUNT, to automate this process; this program and its documentation are on the KEDS web site. In contrast to the text reformatting programs, which need to be customized, KEDS_COUNT should handle most situations requiring aggregation of event data into a time-series. In addition to KEDS_COUNT, the web site also has Dale Thomas's AGGREGATOR, a GUI aggregation program for WINDOWS, and STATA *.do* files written by Jon Pevehouse.

Although most existing research using event data has been done using scaled aggregates, in recent years we have been moving our own work toward event count models. We are doing this because scales, however widely used, are arbitrary and potentially distort the data. Fundamentally, event data are *nominal* and consequently counts, not interval-level measures, are the appropriate approach. The C program COUNT_TYPES can be used to produce these; the source code is available on the KEDS web site.

[4] Having given this advice for a number of years, we don't know anyone who actually has done it: generally the complex texts are simply discarded.

2.6 Using this manual

The remainder of this manual discusses the features of the TABARI program. Chapter 3 describes the actual operation of the program, including the windows, menus and program controls. Chapters 4 through 8 development of coding dictionaries and various optional features affecting the operation of the program. The appendices cover error messages, the numerical limitations of program and an assortment of other meditations on event data.

If you are just trying to get started with the program, or you will only be doing dictionary development, read through the sections in Chapters 3 through Chapter 7 on Input Files and *.verbs* and *.actors* dictionaries. You should also look through a *.verbs* dictionary to get an idea of how TABARI deals with verbs and verb phrases. Many of the optional features of TABARI —for example issues, validation, and time-shifting—are needed only in specialized applications and the program works fine without them. After you have spent some time working with dictionaries, you will find it useful to read (or re-read) Joe Pull's "Ode to Coding" (Appendix F).

If you are planning to implement a major project using TABARI, we would advise reading the entire manual, with particular attention to the material in the Appendices on the support requirements and utility programs. While TABARI is [generally] debugged and stable, it is not a simple "off-the-shelf" solution to data generation, and a full-scale project needs to be able to handle the downloading and filtering of source texts before coding can even begin. Utility programs such as PERL, BBEDIT, ACTOR_FILTER, SANITY.C, and CODE_INDEX will also make your project run more smoothly (or at least that is what they've done here). TABARI and the KEDS project utilities are now based in the command-line world of Unix rather than the GUI world of most commercial Macintosh (and WINDOWS) software, so basic familiarity with UNIX —or access to someone who has that expertise—is probably essential.

Due to the fact that we cannot assume that most users will read the manual cover-to-cover, there will be some redundancy in the information conveyed.[5] The probability is information being repeated is roughly proportional to the likelihood that a novice will need the information adjusted for the amount of incoherent havoc that will be caused by failure to implement the provision and the frequency with which we've encountered individuals running into the problem.

At a few points in the manual there are short sections and footnotes labelled "**Programming Notes**." These are relevant only if you are planning to modify the source code of the program: they usually indicate points in the program where it would be easy to make changes to alter its behavior. If you are only using the program, these sections can be ignored.

[5] For example, did we mention that files need to be saved as TEXT without formatting commands and in the UNIX file format...

Chapter 3

Running TABARI

3.1 Setting Up the Environment

TABARI runs in the TERMINAL program on the Macintosh, or comparable command-line programs in LINUX and WINDOWS. This is a "command line interface" (CLI) environment where actions of the program are controlled by the keyboard in a single window, rather than the GUI—"graphical user interface"[1]—that is now common on most personal computers. The CLI environment is older and simpler than the GUI environment, and for a specialized program like TABARI, which has relatively few options and is used only by trained individuals, it is substantially faster to use. The CLI also allows the program to be standardized across operating systems.

To start TABARI on the Macintosh, double-click the TERMINAL application—this is found in the Applications/Utilities folder on standard OS-X installations. Drag the lower-right corner of the window to create a window at least 48 rows high and 120 columns wide—the size of the window is shown in the title-bar of the window.[2]

Use the UNIX ``cd'' command to navigate to the directory containing TABARI and your dictionaries.[3] For example, if TABARI is in a folder named Documents/Coding/My_Data in your home folder, enter the command

```
cd Documents/Coding/My_Data
```

followed by the <return> key. You could also do this in three commands

```
cd Documents/
cd Coding
cd My_Data
```

[1] Occasionally known as "WIMP"—"windows, input, menus, and pointer", though curiously this acronym has not caught on with advocates of the GUI approach...

[2] This can also be done from the menu bar by selection `Terminal/Window Settings...` and then selecting the `Window` option in the pull-down menu at the top of that window.

[3] "Directory" is the term UNIX uses for the structures called "folders" (and represented by a folder icon) in the GUI interface. We will use the two terms inter-changably here, as they are in fact the same thing. The command "cd" comes from "change directory".

each taking you deeper into the folder hierarchy.

You can use the command

```
ls
```

to get a listing of the contents of the current directory, and

```
cd ..
```

to move back up to the previous folder. If you get thoroughly lost, the command

```
cd ~
```

will get you back to the "Home" directory that contains the standard Documents, Pictures, Music and other folders.[4] TERMINAL is a standard Macintosh application and you can switch back and forth to the FINDER to look at a visual representation of the folder structure.

3.1.1 Unix reference sites

There are an abundance of sites on the web that provide information on UNIX (and OS-X) at every imaginable level of detail; here are three that can get you started.[5]

`http://kb.iu.edu/data/afsk.html`

Basic introduction to the commonly used commands.

`http://helpdesk.ua.edu/unix/reference-card.pdf`

This is a link to PDF file with a reference sheet for basic UNIX commands.

`http://www.math.utah.edu/lab/unix/unix-commands.html`

Just another nice reference site.

3.2 Introductory Menu

To run TABARI, enter the command

[4]The "Home" directory does not, however, contain the Applications directory; this can be reached using the command

```
cd /Applications
```

where the initial "/" indicates that you are starting the search from the upper-most level of the file directory. The Applications directory is shared by all users of the computer, and therefore generally it is better to keep TABARI and the coding folders at the level of an individual user.

[5]All were accessed 26 January 2010 and were found by Googling "unix reference"

```
./TABARI.0.7.0B1
```

with the appropriate substitution for the ".0.7.0B1" part if you are running a different version. If you don't like typing the long name, you can just rename the file to something shorter—`TABARI` for example—and use that instead.

When TABARI is started, you will see the prompt[6]

```
Enter project file name, or <return> for the demonstration file:
```

To start a coding session, type in the name of a *.project* file (to avoid typing, you can also use the "paste" command to insert a file name that you have copied earlier). Alternatively, the <`return`> key will take you to the standard demonstration file.

The program will respond with information about the previous coding session recorded in the *.project* file, then give the prompt

```
Enter coder ID:
```

This can be up to 63 characters in length, but is typically much shorter: 3 or 4 characters (for example, the coder's initials) is the norm. The coder ID is attached to any new phrases the coder adds to a dictionary, as well as being recorded in the `<session>` fields of the *.project* file. Manual entry of the coder ID is skipped there is a $<$ *coder* $>$ command in the *.project* file (see Section 4.1).

Following this, the *.actors* *.verbs* and *.options* files are read (along with any additional optional files); this will usually take a few seconds. If problems are encountered in these files—this will usually occur only when the files have been manually edited—an error message will be provided. At that point you can either `Q)uit` the program and correct the error, or else (usually) ignore the error and continue reading the file. In a few instances—for example a missing *.actors* or *.verbs* file—the program will terminate when it finds an error, but in most instances the information on the erroneous line is simply ignored.

Once the dictionaries have been read, the system skips over all of the records in the first *.text* file that have been previously coded, using the $<$ *last* $>$ field of the final $<$ *session* $>$ record in the *.project* file to determine this information and codes the next record in the file.

3.2.1 Command line options

Project file (-p)

If you already know which project file you are planning to use—which is generally the case—you can skip the initial prompt by providing that file name after the program name. For example

```
./TABARI.0.8.4B1 LetCodeSomething.project
```

[6] In earlier versions of TABARI, the initial prompt was a menu offering the choice of coding or a little-used dictionary comparison function. This was removed as of version 0.7.

will start TABARI with the `LetCodeSomething.project` file. If this is the only option you want to specify, it can be used alone (that is, if the program sees only a file name in the command line, it is assumed to be a project file). using the command option $-p$ when specifying a project file to be used with the $-t, -o$ or $-f$ commands.

```
./TABARI.0.8.4B1 -p LetCodeSomething.project -t CodeThisText.txt -o ThisEventOutput.txt
```

Autocoding: -a

You can go directly into auto-coding mode (see section 3.3.5) by using one of the $-a$ options before the file name. These are

`-a, -an`: auto-code with no feedback
`-ad`: auto-code showing dates
`-af`: auto-code showing full text
`-aq`: quiet mode: no feedback and no usage statistics

Example:

```
./TABARI.0.7.0B1 -af TABARI.Demo.project
```

will autocode `TABARI.Demo.project` with full-text feedback. When the $-a$ option is used, you will not be asked whether to skip previously-coded records and the program exits without a key entry at the end: the assumption is that you are simply coding an entire file. However, you will be given the option to write the usage statistics unless the `SET: ACTOR USAGE = FALSE`, `SET: AGENT USAGE = FALSE` and `SET: VERB USAGE = FALSE` commands are used in the *.options* file (see Section 8.2.4).

By combining the $-aq$ option with a `<coder>` command in the *.project* file (see Section 4.1), autocoding can be done without *any* intervention from the keyboard. This allows autocoding to be run automatically from a script, often in combination with the $-t, -o$ or $-f$ commands.

Code single text file: -t

The command option $-t$ followed by a file name causes the program to code text from that single text file; this requires a $-p$ or $-a$ command specifying a project file. $-t$ *overrides* any `<textfile>` statements in the *.project* file and is equivalent to having only a single `<textfile>`statement.

Example:

```
./TABARI.0.8.4B1 -ad MyAutocoding.project -t CodeThisText.txt
```

Send output to file: -o

The command option $-o$ followed by a file name causes the program to put the coded events in the file; ; this requires a $-p$ or $-a$ command specifying a project file.. $-o$ *overrides* any `<outputfile>` statements in the *.project* file and is equivalent to having only a single `<outputfile>` statement.

Read output and text file names from file: -f

The command option $-f$ followed by a file name causes the program to read the `<outputfile>` and `<textfile>` names from a file. The `<outputfile>` name is in the first line, followed by one or more text file names. The file contains only the file names without the `<outputfile>` and `<textfile>` tags or comments. and `<textfile>` put the coded events in the file. This option; this requires a $-p$ or $-a$ command specifying a project file. These files override any `<outputfile>` and `<textfile>` statements in the *.project* file.

Validation: -c

The command option $-c$ is equivalent to `-p Validation.project`.

Version: -version

The command option $-version$ or $-vers$ shows the introductory screen with the version number, then exits after a key-press.

3.2.2 The Single Most Common Error When Reading Files

At the present time (originally written February 2006, and the situation has improved only modestly since that time), the single most common source of incoherent behavior in TABARI is the use of incorrect file formats.

Due to the decentralized nature of the development of personal computer operating systems,[7] otherwise-standard ASCII text files can have three different types of line endings (that is, characters indicating the end of a physical line)

UNIX: `\n`
original Macintosh: `\r`
WINDOWS: `\r\n`

where `\n` is the character with the numerical value 10 and `\r` has the numerical value 13.[8]

As if this were not sufficiently confusing, with Apple's transition to the OS-X operating system, the "Macintosh" line ending is being phased out of Macintosh software in favor of

[7] And, one suspects, the sheer cussedness and ego conflicts of programmers...
[8] Decimal, that is. Real programmers will prefer the hexidecimal—base 16—values $0A$ and $0D$ respectively.

the "UNIX " line ending. In the long run, this is a very good thing^{TM} but at the moment there are still a lot of Macintosh files lurking around—including some older files available on the KEDS web site—that use the older line endings.

Confusing things even further, many commercial programs—notably MS-WORD and BBEDIT—are "transparent" to the file type, conveniently working with files of any type without alerting the user to this fact. Consequently one can be looking at two files that *appear on the screen* to be identical but in fact they of two different formats.

Finally—and this seems to be happening less often—there are a few cases where *other programs* will change the file type. This usually occurs with file compression programs; we've very occasionally seen it occur when files are transferred through email attachments. This "User is clearly an idiot, do what we think he/she should have done however profound the consequences" (UICAIDWWTH/SSHDHPTC) style seems less common now, perhaps because companies producing this type of software have been selected out of the market.

When TABARI encounters a file in the WINDOWS or classical Macintosh formats, it will totally choke and, depending on the file contents, the error messages may not be helpful. It is also possible, for example, that the *.project* and *.actors* files will be in one format while the *.verbs* and *.options* files are in another: it depends on where the files came from. Again, with a suitably sophisticated text editor, the files will *appear* to be just fine and completely indistinguishable from files that have worked in the past.

The best way to check for this problem is to use BBEDIT or TEXTWRANGLER to check the file type using the `Edit/Document Options...` menu option. If either `Macintosh(CR)` or `Windows(CRLF)` is indicated, select `Unix(LF)` and re-save the file. This just takes a second or two and should solve the problem. Having done this, you will probably want to continue editing the file on a UNIX system—taking the file back to a WINDOWS or classical Macintosh machine will probably (but not necessarily, depending on the editor) change the format back again.

The single most common source of this problem in the second decade of the 21st century comes from editing files in MS-EXCEL—including versions as recent as MS-EXCEL 2008—which for reasons known only to the geniuses in Redmond, defaults to saving text files in the pre-OS-X "Macintosh" format which has not been used in most Apple programs for nearly ten years. Yet presumably hope springs ever eternal in Redmond, and this "Apple" thing is a mere passing fad, and thus we mustn't waste time converting to a new format when very shortly, the only people using Macs will be those playing "Brick-Out" in their parents' basement on legacy machines with fading 9-inch black-and-white screens. Or something like that. Anyway, as usual, you can probably blame Microsoft for the problem.

3.3 Main Coding Menu

The primary coding menu, which is always displayed following the coded record, contains the following options:

```
Select:  N)ext P)arsing M)odify R)ecode A)utocode O)ther Q)uit ->
```

3.3.1 N)ext

This simply moves to the next record in the *.text* file. The <return> key will also do this.

3.3.2 P)arsing

This option displays TABARI's parsing of the text, including parts of speech, clause markings, and the literals that have been associated with the words; the details of parsing are discussed in chapter 7. It also writes an HTML file with a color-coded representation of the sentence and other parsing infomation; see section 7.2 for details.

3.3.3 M)odify

This option is used to modify the phrases and codes of the *.actors* and *.verbs* dictionaries. It leads into a hierarchy of menus where you first choose the phrase to be modified (or added) and then make the modification. `M)odify` is described in detail in section 3.5.

3.3.4 R)ecode

This recodes the text without moving to the next record. This is primarily used when you have gone backwards in a file after making dictionary modifications and want to see how these affect an earlier record.

3.3.5 A)utocode

Autocoding codes the entire sets of files specified in the project $< textfile >$ commands without pausing; this is usually done in order to produce a final data set. Once the process has been initiated, it cannot be stopped except by terminating the program using Cmd-. (period), Cmd-Q or Ctrl-C. The `A)utocode` option presents the prompt

```
Feedback level:  N)one D)ate F)ull H)elp C)ancel ->
```

`H)elp` presents a fairly informative help screen; `C)ancel` exits the autocoding mode without doing anything.

The remaining three options initiate the autocoding at various levels of screen feedback. The speed of autocoding is dramatically affected by this choice: the less that is written to the screen, the faster the coding. There are three levels:

N)one No feedback on individual records; only the text file names are displayed as they are processed. [default]

D)ate The date and record ID of each record is displayed as the record is processed. This is useful in identifying a record that is causing the system to crash.

F)ull Display the text and the coded events as in standard coding. This slows the program substantially.

At the menu prompt, type the appropriate key— N, D, F, H or C —to select an option. <return> selects the default option, N)one.

When autocoding is finished, you will be given the option of writing *.usage* files for each of the dictionaries: these show the number of times that each phrase and pattern in each of the dictionaries has been identified in the texts that were just coded (see section 3.4.4. This facility can be cancelled by using the SET: ACTOR USAGE = FALSE, SET: AGENT USAGE = FALSE and SET: VERB USAGE = FALSE commands are used in the *.options* file (see Section 8.2.4).

3.3.6 O)ther

This brings up a second menu with the additional options described in section 3.4.

3.3.7 Q)uit

Quits the program. Prior to terminating the program will optionally write the any changes in the *.actors* and *.verbs* dictionaries (the default is to save the changes), and can also write a file that shows the number of times each phrase has been used.

The *.project* file is then updated and the program waits for a final keystroke (<return> will work) before returning to the prompt in the TERMINAL application.

3.4 O)ther Menu

3.4.1 B)ackup

This saves the current *.actors* and *.verbs* dictionaries with the suffix *.back*. In contrast to the process used to save files when the program terminates normally, B)ackup does not rename the original files. The results of this option are a bit complex and are described in detail in section 10.2.

3.4.2 P)roblem

This is active only if a $< problemfile >$ line is included in the *.project* file. It allows the user to enter a description of a problem encountered while coding: typically this involves some ambiguity in the codebook, a new actor, or a text where a dictionary entry has undesireable results. Multiple lines of a problem description can be entered, this is terminated by entering a blank line—that is, just hit the <return> key. The description is written to the *.probs* file, followed by the text of the event, the coded events, the parsed version and the tag listing.

3.4.3 M)emory

This gives a report of the amount of memory used in various arrays. This information could be used to adjust the amount of memory the program is using if you plan to recompile the program, or to determine that the dictionaries you are using exceed the capacity of the program. The existing memory sizes are based on generous allowances above the requirements of the current Levant dictionaries, but may need to be increased for other projects.[9]

3.4.4 U)sage

This option writes a file that shows the number of times each actor, agent, verb and verb phrase pattern was used by the program. The count for the actors and verbs shows the number of times the phrase was identified in a sentence; the count for the patterns shows the number of times the pattern was used to actually code an event. These files have the names of the *.actor .verb* and *.agent* files with the addition of the suffix "*.usage*." There are separate prompts for writing the *.actor .verb* and *.agent* dictionaries, though not the individual dictionaries within each type. You will also be asked whether you want to write these files at the end of processing a file and at the end of autocoding unless this has been deactivated in the *.options* file .

3.4.5 eX)it

This option exits the `O)ther` menu and returns to the main menu.

3.4.6 V)alidate [invisible]

This option is not listed in the display and is used only for debugging: it goes through a file and checks that the coded events correspond to the events that were supposed to be generated. The `V)alidation` facility is described in greater detail in 10.1.

3.4.7 T)ags [invisible]

This option is not listed in the display and is used only for debugging: it displays the list of head and tail tags for each words that were assigned during the syntactic markup. These are also included in the TABARI.Parse.html file which is described in 7.2.

[9] **Programming Note**: TABARI uses *fixed* blocks of memory for storing the dictionaries. Consequently, as dictionaries increase in size—prior to the ICEWS project, we considered 1000 actors, 10,000 verb patterns, and 500 codes to be large dictionaries, and when KEDS and TABARI was written, machines with multiple gigabytes of memory were still in the future—the various array sizes need to be increased as well. The program provides warnings when this needs to be done, but it does require changing some of the source code and re-compiling the program. You are also likely to find that when one dictionary-related array size needs to be increased, pretty much all of them will. Most modern programs, in contrast, would handle memory expansion automatically, and in fact it might not be that difficult to incorporate that into TABARI , but it has never been a priority. So just beware.

3.5 Dictionary Modification: M)odify

The `M)odify` option provides the same dictionary modification facilities. While it appears complicated at first, ten keystrokes will get you to the very deepest menu (modifying a code on a verb pattern), and we've found this is substantially faster than working with a mouse, the approach that was used in KEDS.

Important note regarding versions 0.7 and 0.8: A number of features have now been added to the dictionaries without corresponding changes to the `M)odify` menu options inside the program: This is due to the difficulty of writing an interface to those routines and the fact that we have been doing almost all of our dictionary development using utilities such as named-entity recognition programs and linguistic databases that operate outside of TABARI, rather than using the original scheme of developing the dictionaries from inside the program itself. You would, consequently, be advised to do likewise, or at the very least to keep good backup copies.

It would be best to experiment with the dictionary modification to understand the menu hierarchy before doing actual coding. The dictionary modification menus implement a variety of key-stroke filters. For example, while technically the `+` is used to signal adding a phrase, the `=` —which is the unshifted `+` on most keyboards—works as well. The filtered character, rather than the true character, is echoed to the screen.

All modifications are initially saved only in memory, not on the disk. The changes are saved to disk only if the `S)ave` option is used, or if you `Q)uit` the program and confirm that you want to save the changes. If you have not saved the modifications to disk and your computer suddenly shuts down due, for example, to some bozo driving a bulldozer into a power pole, or to some squirrel deciding to flash-fry itself on a transformer, you will lose all of your changes.[10] Conversely, if you have experimented with some changes that seem to have done more harm than good, you can simply `Q)uit` the program without saving the changes, and then restart using the dictionaries as they existed prior to those experiments.

The initial prompt following entering `M` is

```
Modify:  A)ctors V)erbs aG)gents N)ouns aD)jectives eX)it ->
```

Enter the option for the type of entry you want to modify; `X` will go back to the main menu without doing anything. The menus that follow will vary depending on that selection.

3.5.1 A)ctors and aG)ents

The next prompt is

```
Enter first two letters for list, + to add, <to exit menu ->
```

If you want to add a *new* actor or agent, enter `+`; this will take you immediately to the actor/agent entry screen described below. Actors/agents that are added are put in the first *.actors* or first *.agents* dictionary that was read from the *.project* file.

[10] Unless you are using a laptop or have a desktop machine plugged into an uninteruptable power supply that actually works. The two examples provided here, by the way, are based on actual events that occurred in our little ol'coding shop...

If you want to delete an existing actor/agent or change the code on an existing entry, enter the first two letters of the actor or agent you want to modify. This will produce a numbered list of 44 entries (or fewer if you are near the end of the alphabet) in alphabetical order, starting with the first actor that begins with the two letters you entered.

If the actor/agent you want to modify is not on the list, you can go further through the alphabet by entering >. If the actor/agent you want is further back in the alphabet, enter < to exit the `M)odify` sequence and start over with letters that are earlier in the alphabet.

If the actor/agent you want is on the list—which will be the usual case—enter the two digits corresponding to that entry (numbers less than 10 require a leading zero). If the entry you want to change is at the top of the list—entry 01—you can select it using <`return`> .

At this point the program re-displays the text being coded, the actor or agent and code that have been selected, and the prompt

```
Change:  C)ode D)elete eX)it ->
```

Entering `C` gives a prompt where a new code can be entered; follow this with <`return`> . Entering `D` asks for confirmation (`Y` for "yes" or `N` for "no"; the default is `N`) before deleting. `X` exits the process. If you are entering the code for an agent, the program checks to make sure that a ∼ connector was included. Entering a blank code—just hit <`return`> —will return you to the main menu without doing anything.

If you select `+` to add an actor/agent, the text of the current record is displayed, then you are first given a prompt where you can enter the text of the new actor or agent. After this is entered, followed by <`return`> , the program gives a prompt where a new *code* can be entered.

Following the entry of a new actor/agent or code, the current record will be automatically recoded and you will be back at the main menu.

Note: In order to change the *text* of an actor/agent, first add the new actor or agent, and then delete the old one.

3.5.2 V)erbs

The process of selecting a verb to add or modify is similar to that of selecting an actor. The only difference is that after a verb has been selected, the prompt has an option for adding patterns:

```
Change:  P)attern C)ode D)elete eX)it ->
```

Selecting `P` produces a numbered list of the first 44 patterns associated with a verb, in order of length. If the verb has no patterns, you are prompted to add one. Patterns are selected using the same process as the actor and verb selection: enter the two-digit number of the pattern to change its code or delete it; enter `+` to add a new pattern, or enter > to display

the next 44 patterns. When a pattern has been selected, you can either change the code or delete the pattern.

Verbs that are added are put in the first *.verbs* dictionary that was read from the *.project* file.

Note: In order to add a verb and a pattern, first add the verb, then go back through the `M)odify` sequence to add the pattern.

3.5.3 Nouns and Adjectives

These are similar to the actors and verbs modification, but simpler because nouns and adjectives are not associated with codes. When `N` or `D` is entered—note that changing an adjective uses `D`, not `A`—you are first asked to either enter the first two letters of the word (this is used if you want to delete an item), or `+` to add a word. If `+` is selected, you go directly to a prompt for adding the new word without going through a list.

If two letters are entered, you are shown an alphabetized list of 44 words beginning with the first word that begins with the two letters. If a two-digit number is entered, you are presented with the prompt

```
Change:  A)dd D)elete eX)it ->
```

`D` causes the word to be deleted after the usual confirmation prompt.

`A` provides another opportunity to add a word: this is typically used after one has examined the list to see whether a word is present. If `A)dd` is used, the word is entered at the appropriate point in the alphabetized list, rather than at the point in the list corresponding to the two digits that were entered to get this prompt. (In fact, if you are planning to `A)dd`, a simple approach is to just enter <`return`> in order to get to the prompt.) Nouns and adjectives that are added are put in the first *.actors* dictionary that was read from the *.project* file.

3.6 Summary of Keyboard Shortcuts

None of the alphabetical menu options are case sensitive: '`A`' and '`a`' will give the same results.

3.6.1 Main Menu

<`return`> , <`space`> : same as `N)ext`

`0` (zero) : same as `O)ther`

< or `,`: moves back to previous record. At present, the previous 32 records are stored in a circular array; you can't go further back than this (the 33rd reverse will take you back to the where you were). New records are not recoded until you select `R)ecode`. Recoded records will be out of chronological order in the *.events* file (but you're going to autocode everything anyway, right?)

3.6.2 A)utocoding Menu

`Q` and `X`: same as `C)ancel`

3.6.3 O)ther Menu

`E` and `Q`: same as `eX)it`

3.6.4 Modify Menus

<`return`> selects the first item displayed (same as entering "01")

, . and = are automatically shifted to <, > and + respectively (in other words, the menu responds to the key, not the character)

+ can be used to add a new term in the menu following the dictionary listing (this option isn't listed in the menu)

Chapter 4

Input Files

Every TABARI run involves the following files:

.project: The project file contains information about all of the other files used in the coding, as well as a coding history.

.text: The source text to be coded.

.verbs: One or more files containing a dictionary of the verbs and verb phrases used to code events.

.actors: One or more files containing a dictionary of the nouns used to identify the sources and targets of events.

.options: A file containing a variety of commands that modify the behavior of the TABARI program.

All of the files are in "flat ASCII" format and should only be edited using a program that produces a a file without embedded control codes—we recommend BBEDIT. The *.project*, *.text*, and *.events* files are described in this chapter; the remaining files are described in later chapters.

In addition to the required files, TABARI also can work with the following optional files; the details of the files are described later chapters.

Input

.agents: One or more files containing character strings are combined with entries in the *.actors* dictionary to produce composite codes [see Section 5.4]

.issues: These files contain character strings used to identify the context of the text [see Chapter 9]

.freqs: These files contain character strings used to do simple content analysis [see Chapter 9]

Output

.events: An output file containing the coded event data. This is required—it is specified using either a `<eventfile>` or `<outputfile>` statement— if you want to generate event data, but can be optional if you are only doing dictionary development and won't be using the coded data at that point.

.probs: record of problems encountered in the data that were noted in manual coding. [see section 4.1.1]

.errors: record of problems encountered in the data that were noted in automated coding. [see section 4.1.1]

.complex: source text that was too complex to machine-code by machine [see Chapter 8]

By convention, the files are identified by suffixes such as *.project*, *.actors*, and *.options*. That method of identifying files will be used throughout this manual but in fact the program does not use the information: A file name such as Balkans.ethnicgroups would work as an actor file name. However, unless your file names are approaching the limit of the length of the file names, there is little reason not to use Balkans.ethnicgroups.actors in order to make it clear that the file contains a TABARI actor dictionary.

4.1 .project File

The *.project* file is a text file with XML-like tags. The tag terminators
$< / \ldots >$ are optional, but if used, comments can follow the terminator. The tags that give information on the input files are followed by a series of
$<$ *session* $>$ lines that record who worked on the file and when they worked on it. There is no hidden or subtle information in the *.project* file—if information in the file is edited (typically by changing the $<$ *last* $>$ field to change the number of initial records that are skipped) the program should accommodate the changes without problems.

Format:

```
<actorfile> filename
<verbfile> filename
<optionsfile> filename
<textfile> filename
<agentfile> filename  [OPTIONAL]
<eventfile> filename  [OPTIONAL]
<outputfile> filename  [OPTIONAL]
<problemfile> filename  [OPTIONAL]
<errorfile> filename  [OPTIONAL]
<issuesfile> filename  [OPTIONAL]
<coder> code ID  [OPTIONAL]
<system> Unix system command  [OPTIONAL]
<session> filename
```

Notes:

1. The file names can occur in any order, and the program looks only at the first two letters of the tag.
2. There can be multiple $< actorfile >$, $< agentfile >$, $< verbfile >$ and $< textfile >$ files. The multiple dictionaries allow, for example, separate dictionaries for general and regionally-specific vocabulary;[1] the multiple text files allow texts that will be coded to be broken into files of manageable size.
3. Any line that does not begin with < will be ignored. This can be used to add comments. By convention, begin the comment lines with the UNIX # comment delimiter.
4. The final $< session >$ line contains a $< last >$ field that determines how many records will be skipped in the next coding session. The $< session >$ fields should be the last ones in the file.
5. The string following $< eventfile >$ is a prefix; the session number will be added to it. The string following $< outputfile >$ is the complete file name and does not change between runs. If both are present, $< outputfile >$ takes precedence. The event file is optional—if you are just doing dictionary development and arent interested in the resulting events, leave it out.

To code a set of files, just use multiple $< textfile >$ listings: The system will go through all of these in order. There is no practical limit on the number of these files; if a file is not found, the system will pause and report an error, then go to the next file. The `A)utocode` option goes through all of the files in the list.

Example:

```
<actorfile>  international.actors
<actorfile>  WAFR.actors
<agentfile>  generic.agents
<agentfile>  WAFR.agents
<verbfile>  WAFR.verbs
<optionsfile>WAFR.options
<textfile>  WAFR.1991.text
<textfile>  WAFR.1992.text
<textfile>  WAFR.1993.text
<eventfile>  WAFR.1991-93.evts
<coder>  rmn
```

Programming Note: The subfield `TYPE="XML"` in the $< textfile >$ tag indicates that the file will be interpreted using the `ReadXMLClass::readXMLRecord(void)` routines; otherwise it will be read using `ProcessorClass::readRecord(void)`.

4.1.1 $< problemfile >$ and $< errorfile >$

These two output files can be specified in the *.projects* file and provide feedback on errors encountered during the coding.

[1] This feature is new in version 0.7.

The < *problemfile* > line is optional and is needed only if you want coders to be able to use the `O)ther/P)roblem` menu option to save problematic records: see section 3.4.2.[2]

The < *errorfile* > line is also optional; it saves a record of recoverable errors that are generated by the parser and coder when these are in autocoding mode. This is primarily used for debugging, though it might also show where problems are occurring if the program has crashed, and also provides a list of the records that were not codeable (sometimes this is due to the story, sometimes poorly constructed dictionary entries). The file is overwritten every time the program is run—it is internally date-stamped—and is generated only if an error actually occurs. A maximum of 8 errors are reported for any given story, but usually the program will be unable to do further processing after only two or three errors.

4.1.2 < *coder* >

The < *coder* > command bypasses the prompt for a coder ID and sets this to the blank-trimmed string in the command line. It is useful when setting up programs to do automated coding without keyboard input, and in situations where you aren't concerned about differentiating between coders.

4.1.3 < *system* > and < *run* >

This runs the blank-trimmed string in the command line as an operating system command using the C++ `glibc system()`. This can, in principal, do almost anything—for example generate dictionaries from auxiliary files or move files into place from other dictionaries. There is no limit to the number of < *system* > commands, and these can occur anywhere in the project file.

If a < *run* > command is present, the < *system* > commands following < *run* > will be run *after* TABARI has finished running—this allows < *system* > commands to move or process output files. < *run* > also sets a string $SESSN to the completed session number so, for example,

```
<eventfile>  WAFR.93.evts
<run>
<system> mv WAFR.93.evts.$SESSN EventOutput
```

[2] Programming: Using the tag < *problemfile*! >—a '!' has been added—causes a *.debug* suffix, rather than the session number, to be added to the problem file name, and the file is always saved, even if the `O)ther/P)roblem` menu option is not used (under default circumstances, an empty < *problemfile* > is deleted at the end of a session.) This allows the `Processor.fprob` file to be used for debugging output. More generally, the `#define TAB_DEBUG = TRUE` flag in `TABARI.h` switches the program to use a series of shortcuts:

- immediately goes to the `C)oding` option
- reads a *.project* file hardcoded in `TABARI.cp`, unless overridden by a file name in the command line
- sets `sCoder` to "debug"
- at termination, deletes the new event file
- at termination, skip the over-writing of the *.actors* and *.verbs* files even if they have been changed.

would move the most recently coded event file to the directory EventOutput. The code

```
<errorfile>  WAFR.errors
<run>
<system> mv WAFR.errors WAFR.errors.$SESSN
```

appends a serial number of the most recent version of the error file, so that the versions for consecutive runs will not be overwritten.

The commands prior to $< run >$ are run before the dictionaries are read into TABARI . The commands following $< run >$ are the last thing that the program does—they occur just before the "Press any key to end TABARI" prompt (or just before the automatic exit in an autocode). If for some reason the command can't the executed, the error message "`Unable to execute` $< system >$ `command`" is displayed; otherwise the command is displayed following the text "`Ran command:`".

4.2 Internal Data Set Documentation

As the KEDS project has moved to the mass production of data sets, problems have arisen in identifying exactly the circumstances under which each data set has been produced. To deal with this, we've introduced internal documentation at the beginning of the event data set. This documentation consists of tab-delimited records that have the date of the first record to be coded. The source and target codes are "`DOC`", the event code is "`999`", and these fields are followed b various identifying information. The idea here is that these records will be transparent to any other programs used to process the event data, and can be either deleted or defined as missing values when the files are used in a statistical analysis.

The first two lines of the documentation block written by TABARI contain information on the TABARI version number, the date and time the file was generated, and the coder. This is followed by an arbitrary number of document header lines that are copied from $< dochead >$ records in the project file. The functional commands—$< actorsfile >, < verbsfile >$ and so forth—from the project file are copied, followed by $< doctail >$ records.

Example:

The following is a sample *.project* file with documentation. Note that the first line will be treated as a comment because it does not begin with <.

```
# Bosnia draft thematic coding file for TABARI/CAMEO, May 17, 2005
<dochead>This is the first header line from <dochead>
<dochead>This is the second header line
<actorsfile>  Balkans.actors.updated
<verbsfile>  BALK.VERBS
<optionsfile>CAMEO.OPTIONS
<textfile>  Bosnia.Reuters.leads
<textfile>  bosnia.AFP.AP.leads
<eventfile>  BOSNIA.THEME
# <freqfile>HH4.freq
```

```
<freqfile>HH4.1.freq
<doctail>This is the first tail line from <doctail>
<doctail>This is the second tail line
```

If the file Bosnia.Reuters.leads has the date 880501 in the first data record, this would produce the following header:

```
880501   DOC   DOC   999   Data generated by TABARI version 0.7.2b2
880501   DOC   DOC   999   Tue 14 Jun 2005 17:51 CDT
880501   DOC   DOC   999   Session 22; Coder pas
880501   DOC   DOC   999   This is the first header line from <dochead>
880501   DOC   DOC   999   This is the second header line
880501   DOC   DOC   999   <actorsfile>  Balkans.actors.updated
880501   DOC   DOC   999   <verbsfile>  BALK.VERBS
880501   DOC   DOC   999   <optionsfile>CAMEO.OPTIONS
880501   DOC   DOC   999   <textfile> Bosnia.Reuters.leads
880501   DOC   DOC   999   <textfile> bosnia.AFP.AP.leads
880501   DOC   DOC   999   <eventfile> BOSNIA.THEME
880501   DOC   DOC   999   <freqfile>HH4.1.freq
880501   DOC   DOC   999   FREQUENCIES: Positive Negative Neutral
880501   DOC   DOC   999   FREQS: POSV NEGT NEUT
880501   DOC   DOC   999   This is the first tail line from <doctail>
880501   DOC   DOC   999   This is the second tail line
```

Notes:

1. In order for internal documentation to be generated, at least one $<$ *dochead* $>$ or $<$ *doctail* $>$ record must be present—this signals TABARI to create the documentation.

2. Comments, $<$ *session* $>$ records and other lines from the *.project* file that do not contain commands are not copied—if you want this information copied, include it in a $<$ *dochead* $>$ or $<$ *doctail* $>$ record.

3. Usually the first $< textfile >$ record will contain the earliest date in the data set. If it doesn't—and if having this date correct is important (e.g. if the event data is going to be processed by a time series analysis program)— create an artificial record that contains the appropriate date but does not generate an actual event.

4. If internal documentation is generated and word frequency information is being tabulated from a $< freqfile >$, two tab-delimited lines labeled `FREQUENCIES:` and `FREQS:` are added that contain the strings in the `CATEGORY` and `ABBREV` fields of the $< FREQ >$ headers. These lines have the usual `DOC/999` fields.

4.3 Text Files

TABARI will process any flat ASCII file containing records in the following format

```
date idinfo any other information
source text line 1
source text line 2 ...
source text line n
blank line
```

Example:

```
000101  AFPN-0003-01
Israelis breathed a sigh of relief Saturday after none of the three
things they feared -- suicides by Christian cranks, the Y2K bug
and Palestinian unrest -- came to pass as the country moved into
the year 2000.

000102  AFPN-0005-01
Qatar and Yasser Arafat's Palestinian Authority resolved a tense
diplomatic stand-off on Sunday after an agreement to allow a banker
accused of embezzlement to return to Doha.

000102  AFPN-0011-01
Prime Minister Ehud Barak left Israel Sunday for a new round of
negotiations with Syria, saying he faced a difficult mission but
pledging to reach a peace deal that would guarantee Israel's security.

000102  AFPN-0012-01
A Russian airliner with 125 passengers on board flew into Gaza
international airport on Sunday, the first commercial flight between
Moscow and the Palestinian territories.
```

Your filter program should reformat the texts to be coded into this form. If you are using a word processor for the reformatting, be sure to save the file in flat ASCII "text" format rather than the normal format of the word processor.

date is the date of the sentence being coded in the 6 digit format YYMMDD. For example: 880318 is March 18, 1988. See comments in section 6.5.2 for the interpretation of dates that are incorrectly formatted and dates in the years 2000 to 2030; also see comments below on "dates" that don't begin with a digit. If the `SET: YYYY = TRUE` command has been used in the *.options* file (see section 8.2), years are 4-digit, i.e. `2007` rather than `07`

idinfo can be up to 31 characters in length; it follows the date and can be anything used to identify the source text (including blanks). In our projects it typically combines an identification of the source (REU for Reuters, AFP for AFP) and gives a serial number that uniquely identifies the source text in a specific day. This can also be used to identify whether records refer to sequential sentences; see the discussion of the FORWARD command in section 8.4.

text lines should be around 80 characters in length, although technically an individual input line can be up to 255 characters in length. However, punctuation filtering (for example

adding spaces around commas) can add some characters, so a line at the 255 character limit might exceed the allowed length after filtering, and would be truncated. To be on the safe side, keep the individual lines around 80 characters; short lines are also easier to read.

TABARI re-formats and "word wraps" the text before it is displayed. You can force line breaks in the record display by putting the sequence "`\n`" in the text where you want the break to occur The `\n` should go at the *beginning* of a line in order to avoid the input system adding a blank after it (which would happen if it is at the end of a line, where it is seen as a non-blank character). The comments in the *.demo* file are formatted in this manner.

The total size of any single text record is limited to a maximum of 2047 characters.

blank line is a line containing no non-blank characters. Ususally this is a line containing only a <`return`> , but TABARI will also respond appropriate to a line containing only blanks and tabs.

Filtering programs sometimes mangle texts that have an unusual structure, so the following rules have been introduced to reduce the potential problems with malformed texts:

1. No coding is done if there are fewer than 8 words in the sentence. While it is certainly possible to construct a grammatically-correct English sentence having a subject, verb and object using fewer than 8 words, we've never seen a codeable sentence of this length in actual news reports. But we've seen plenty of cases where our reformatting programs incorrectly split sentences at abbreviations and produced a sentence fragment that was this short. This threshold can changed using the `SET: MINIMUM WORDS` described in section 8.2.3.

2. If the record contains more than 127 words or more than 2047 characters, it is truncated but still coded; if an $< errorfile >$ has been specified in the *.project* file, this will be reported in that file. This situation generally occurs when several sentences are erroneously run together by a filter, and consequently the first part of the story may be codeable (more frequently, the entire text will be kicked out by the complexity filter if that option is being used).

3. If a "word" contains more than 63 consecutive characters, the 63rd character is changed to a blank to terminate the word; the next "word" begins after that blank. Since no legitimate non-technical English word has anywhere close to 63 letters–"antidisestablishmentarianism" has only 28—this occurs only when some sort of junk or uncodeable technical phrase is in the text, and splitting the word allows the remainder of the sentence to be parsed. (which might not be a good idea. . .)

4. TABARI uses the presence of a digit in the first character of a line to signal that it has found the date/ID line at the start of a text record, and a blank to signal the end of a record, so any block of text that does *not* contain a digit as the first character and is delimited by blank lines is, in effect, a comment block and will be ignored in the coding. For example

```
#---------------------------------------
this is a block of comments that will
```

```
be ignored. The '#---...' lines are not
required, but comments cannot contain a
line beginning with a digit [0-9]
#---------------------------------------
```

We have not used this feature in operational coding—and the first-digit rule exists primarily to deal filtering/formatting mistakes—but it could be useful under some circumstances such as test files, or for internal documentation of the texts.

4.4 Event file

The default event output format is

```
<date> \t <source code> \t <target code> \t <event> \t <label>
```

where $\backslash t$ is the tab character. If the option `SET: LABEL = FALSE` is used, the last field is suppressed. Additional identifying information information can be added using the `OUTPUT` command in the *.options* file; see sections 8.2.12 and 8.9 for details.

Example:

```
040401 ISRSET ISRGOV 220 (FORCE) FORCED
040401 USAGOV ISRGOV 031 (MEET) HOLD MEETINGS
040401 ISRGOV USAGOV 031 (MEET) HOLD MEETINGS
040401 JOR SYR 121 (CRITICIZE) SUSPECTS
040401 PALGAZ PAL 094 (CALL FOR) CALLING ON
040401 USAGOV ISR 042 (ENDORSE) ENCOURAGED
040401 BEL NTH 023 (NEUTRAL COMMENT) SAID
040401 SYR USAGOV 171 (UNSPECIF THREAT) STRAINED
040401 PAL USA 102 (URGE) URGED
040401 ISR PALPLO 160 (WARN) SHARON WARNED AGAINST YASSER ARAFAT
040401 ISR USAGOV 150 (DEMAND) DEMANDS
040402 ISRMIL PAL 223 (MIL ENGAGEMENT)  PALESTINIAN KILLED BY ISRAELI ARMY
040402 ISRGOV PALPLO 173 (SPECIF THREAT) NOT RULE OUT
```

(These data were generated using the `SET: MATCH = TRUE` option so the final field contains the text that generated the event coding: see section 8.2.12.)

Note: Because very few statistics programs are designed to work with categorical time series data, the output from TABARI is usually not read directly into a statistical program, but instead is aggregated in some fashion. In addition, a very large data set may be too large to be read by a spreadsheet program such as MS-EXCEL; if you want to examine the data visually, use a text editor such as BBEDIT or MS-WORD.

Chapter 5

.Actors, .Agents and .Verbs Dictionaries

5.1 Purpose

Most of the coding decisions made by TABARI are based on the dictionaries of words and phrases in the *.actors*, *.agents* and *.verbs* files. This chapter describes the basic features of those dictionaries.

In order to work with the dictionaries (as well as the `.options` file), you will need to be able to edit files using a word processor or text editor that produces flat ASCII, Unix-formatted "text only" files. The most reliable way to do this is with a text-only editor such as BBEDIT or TEXTWRANGLER or their LINUX or WINDOWS equivalents. Text-only editors will not inadvertantly add formatting codes to your files and often have search-and-replace capabilities that are far more sophisticated than those found in word processing programs.

If for some reason you want to use a general-purpose word processor, virtually all word processors can save in a text-only format. However, make sure that at every stage of your processing, the files that TABARI will be reading have been saved as "text only" rather than in a word processing format. If a file that was working earlier suddenly starts producing garbage or crashing the program, it was probably accidentally saved as a formatted document—for example MS-WORD will do this if you inadvertently format any part of the document.

While the *.actors*, *.agents* and *.verbs* dictionary files can be modified while the program is running, TABARI is not designed to build those files from nothing: working with empty *.verbs* and *.actors* files will have unpredictable results. It is generally more efficient to initially create or extensively modify the dictionaries with a text editor rather than building the entire vocabulary using the `M)odify` menu option inside the program.

Versions 0.7 and 0.8: See note concerning unimplemented features in the `M)odify` menu in Section 3.5

5.2 Phrase Input Formats for Dictionary Files

Except as noted in Section 5.3.1, the *.actors*, *.agents* and *.verbs* files use the same format:

```
PHRASE [CODE] {; comments}
```

The last line in the file should be ∼FINISH; this is the internal end-of-file mark. Lines after the ∼FINISH will not be read into the dictionary but will be copied into the saved file.

Initial and terminal white space (blanks and tabs) are trimmed from the input line and PHRASE. The comments after the semicolon are optional; this allows the origin of the pattern (e.g. the coder identification and date) and other remarks to be documented.

Any line *beginning* with / ("slash") or # is a comment and ignored. This can be used to temporarily eliminate phrases from the dictionaries. However, these are *not saved* and if the dictionary is re-written following modifications inside the program, these "commented" lines will be lost (they are still available in the *.old* version of the file).

Hyphens ('-') are replaced with underscores. As of version 0.7.3, TABARI replaces any hyphens in the input text with a blank: this change was made to deal with inconsistencies in transliteration—e.g. "al-Assad" versus "al Assad." Corresponding hyphens in the dictionaries can almost always be unambiguously replaced with an underscore "_", so this feature allows older dictionaries to be used. If a dictionary is saved, the revised version with the substitution will be used.

5.2.1 Codes

Codes can be any length, but can contain only letters and numbers. (i.e. no special characters such as '!', '[', or '$'; in particular avoid the special codes - - -, ***, ### and +++ and the agent connector ∼). Despite the absence of a length restriction on codes, they are commonly short: our project usually uses 3 or 4 digit codes for events, and 3 to 12 character codes for actors. The length of a code should balance the mnemonic advantages of long codes with the difficulties of using lengthy codes in complex code constructions and the increased size of the resulting output files.

Number of distinct codes is potentially as high as $2^{16} - 1$, but in the current version of TABARI the maximum is set at 4096 (2^{12}).[1]

5.2.2 Stemming

Stemming can be used to match different forms of the same word using a single string called a "stem." For example:

ACCEPT:	**ACCEPTS**	**ACCEPTED**	**ACCEPTING**
SYRIA:	**SYRIA'S**	**SYRIAN**	**SYRIANS**

[1] **Programming Note:** This maximum is set by the constant MAX_CODES in TABARI.h

A word is considered to match a stem provided every character in the stem matches. In other words, **SYRI** will match all four forms of **SYRIA** but it will not match **SYRACUSE**.

When phrases have the same initial letters, TABARI checks long phrases before shorter ones: For example **SIGNALLED, SIGNED and SIGN** have the search order:

```
SIGNALLED
SIGNED
SIGN
```

To prevent a character string from being used as a stem, put an underscore (_) after the string: this means that the string will match only if it is followed by a space. The phrase `OF_` will only match "`OF`◇" whereas the stem `OF` would match "`OF`", "`OFFER`" and "`OFFICIAL`."[2]

Considerable effort may be involved in the dictionary tweaking to determine which stems can be used without causing problems. For example `TOUR` looks like a useful stem for TOUR, TOURED and TOURING, but it also matches TOURIST, which can cause problems. The phrase:

```
HEAD
- * FOR
```

correctly handles the verb forms `HEADING FOR` and `HEADED FOR` but inconveniently also matches `HEADQUARTERS FOR`. To force an entire word to be used in a match, end it with an underscore; alternatively, problematic words such as `HEADQUARTERS` can be eliminated using null codes (see section 6.2) or designated as nouns (see section 5.3.2)[3].

Note: In our experience, stemming is the most frequent cause of wildly inaccurate coding errors, and it is not entirely clear that this feature is worth the trouble. In a recent case[4], for example, the verb `BEAT` – which is generally problem-free—matched BEATY, the name of an American temporarily held captive in Iraq.

"Stemming" is a quick-and-dirty approach to a more general linguistic problem called "lemmaization," which involves finding the root from which various forms of a word can be constructed. In some languages, this can involve a large number of suffixes, as well as the possibility of prefixes and even in-fixes (changes in the middle of the word). Fortunately for our purposes, English relies almost solely on suffixes, and, in the case of regular verbs, there are only two: "s" and "ed". Suffixes are also generally used to modify nouns and adjectives; the negation "un" (as in "unimportant") is the primary exception. Consequently simple stemming will handle lemmaization for most common English words.

[2] ◇ stands for a blank in these examples

[3] Yet another linguistic side note, October 2006: As it happens, the example choosen here is particularly appropriate, as we heard at a recent conference that "HEAD" is in fact the English language word with the greatest number of multiple meanings, around 60. These range from the common uses as a body part ("He put his hat on his head") and rank ("She was at the head of her class.", "The head of government") to relatively obscure nouns such as those referring to alcoholic aero-gels ("Nice head of foam on that beer"), a marine toilet ("The head is clogged") and—my favorite—one of the very few instances of a counter word in English, "The ranch had 300 head of cattle").

[4] It was recent when this text was written for the KEDS manual in 1992. Mr. Beaty's kids are probably now assuming a similar risk.

5.2.3 Regular Noun and Verb Endings

Beginning with Version 0.7, TABARI now automatically checks for regular plural (S) and possessive ('S) endings on nouns, actors and agents, and the following regular verb endings

```
...S
...ED
...ES
...EN
...ING
```

These endings (but *only* these endings...!) are recognized even if the underscore _ is added to the end of the word. The following rules apply to this facility:

1. This only applies to the final word in a multi-word actor, agent or noun phrase.

2. These endings are *not* employed if the word is terminate with "="

Iregular noun endings such as `CITY_` → `CITIES`, `THIEF_` → `THIEVES`, `KNIFE_` → `KNIVES`, `CRISIS_` → `CRISES`,and `MOUSE_` → `MICE` should just be included as separate entries[5]

With the addition of this facility and the irregular verb forms discussed in section 5.5.1, it should be possible to avoid stems in *most* instances, which should both simplify the dictionaries and reduce the errors from stemming. Which is to say, most words should now end with _.[6]

5.2.4 Error Checking

TABARI does *some* error checking of the dictionary elements as it reads in the dictionaries, but this is by no means extensive. When an error is found, the program will stop, describe the error, display the line where the error occurred, and if the error involves only a single line, give you the option of either quitting or continuing to read the dictionaries. The line containing the error is ignored. Errors involving the overall file structure—for example a missing `</NOUN>` terminator—cause the program to stop.

At the present time, the following errors are detected

- Phrase missing or fewer than three characters in length
- No code in `[...]`
- Missing left-bracket [
- Missing right-bracket]
- Missing ~ connector in an agent code

[5] At present there is no "noun form" facility similar to the verb forms because actor, agents and nouns are just associated with at single set of codes, rather than a set of patterns.

[6] The "=" ending usually isn't required since the regular endings on a noun or verb having an irregular form will just be a spelling error and, very occasionally, actually match spellings from individuals whose command of English and spell-checking software is less than optimal.

- Missing verb marker '*' in a pattern
- Patterns in the *.verbs* file prior to a verb
- Blank line
- Pattern that exceeds 255 characters
- Missing `</NOUN>, </ADJ>, </ATTRIB>` and `</TIME>` terminators
- No verb forms inside ...
- Duplicate entries: identical phrase entered multiple times[7]

This list is not comprehensive and undetected errors are possible—in general the program pre-supposes that the input is correct. If TABARI suddenly starts acting strangely after previously cooperating, particularly if you have just manually edited the dictionaries, an incorrectly formatted dictionary entry is likely to be causing the problem.

5.3 .Actor Dictionary

The *.actors* file contains proper nouns and their associated codes. Multiple nouns can refer to the same code; for example in our system the actor code `ISR` (Israel) corresponds to `ISRAEL, ISRAELI, ARENS, PERES, JERUSALEM, TEL AVIV` and `SHAMIR`, among others.

Example:

```
ABU_SHARIF [PLO]
ACQUINO [PHL]
AL-WAZIR [PAL]
AMMAN [JOR]
AMNESTY_INTERNATIONAL [NGO]
ANKARA [TUR]
ANTIGUA [ATI]
AQUINO [PHL]
ARAB [ARB]
ARAFAT [PAL]
ARCHBISHOP_OF_CANTERBURY [PVO]
ARENS [ISR]
```

It is occasionally necessary to use two or more spellings for a single actor, particularly if multiple (or poorly edited) text sources are used, as with `ACQUINO` and `AQUINO` in the example above (to say nothing of `QADDIFY, GADDIFY, QADHFI, KHADAFI`...).

The codes assigned to actors can be quite complex, and can include both multiple codes assigned to a single phrase (see section 6.5.1) and codes that change depending on the date of the event (see section 6.5.2).

[7] This can be deactivated using the `SET: DUP WARNING = FALSE` command in the *.options* file: see section 8.2.5.

5.3.1 Actor Synonyms and Multiple-line Date Restrictions

TABARI version 0.7.5 introduced an expanded form of the actors dictionary that allows all of the synonyms for an actor to be placed in a single set of entries, and date restrictions to be placed on separate lines. This modified format is intended to enhance the ability to both read and maintain the dictionaries.

Example:

```
FRODO  [HOB]
GALADRIEL  [ELF]
GANDALF  [WIZ]

GOLLUM  ; pas 27 Apr 03
+SMEAGOL
+SLINKER
+STINKER
  [HOB <950601]
  [HOBGOV 950601-990815]
  [HOBREB >990816]

GONDOR
+MINAS_TIRITH
+ANORIEN
  [GON]

ITHILEN  [ITH]
KIRITH_UNGOL  [LOR]
```

Actor synonyms—set of actor phrases with the same code, typically multiple spellings of a name—are designated by an initial actor phrase followed by a synonyms beginning with +. The blank lines separating the synonym groups are optional and there to improve readability. When used in combination with the `OUTPUT: ACTORS PARENT` facility in the *.options* file (see Section 8.9), this can be used to disambiguate multiple forms of an actor name.

Multiple-line date restrictions can follow the synonyms (or a single actor): these lines must begin with either a space or a tab, and contain a single [...] code block, either simple or date restricted. The multiple-line date restrictions follow the same rules as the single-line date restrictions—in fact all the program does is assemble these into a single line for processing—but are easier to read, particularly since comments can be added to the individual lines.[8]

As of Version 0.7.5, this formatting is *not* preserved when the dictionary is written: this feature is primarily designed for files that are externally formatted.

[8] Consequently, the total length of the date restrictions, with spaces added to separate them, must be less than 256 characters, the input line length limitation. This situation produces a reasonably information error message.

5.3.2 Nouns and Adjectives

The actors file can contain a list of nouns and adjectives. The sections of the dictionary containing these are delimited by `<NOUN>...</NOUN>` and `<ADJECTIVE>...</ADJECTIVE>` respectively. Nouns and adjectives do not have codes; they are primarily used to increase the accuracy of the parsing of compound noun and adjective phrases, which in turn increases the likelihood that the compound clauses in a sentence will be correctly identified.

The `<NOUN>` category can be used in place of many the null-coded actors that were used in KEDS ; the `<ADJECTIVE>` category can be used in place of the [rarely-used] `TITLE` facility in KEDS . If you are working with older dictionaries, it is generally useful to move any words that were null-coded because they were nouns and adjectives to `<NOUN>` and `<ADJECTIVE>` lists because this will improve the effectiveness of the parsing. Null-coding simply removes a word from consideration, whereas designation as a noun or adjective allows it to be used in the identification of compound phrases. This in turn insures those conjunctions will not be incorrectly used to delimit compound *clauses*.

5.4 .Agents Dictionary

The optional *.agents* dictionary is new to Version 0.7 and contains common nouns that can be associated with actor codes. This facility was added to get around the necessarity of redundantly specifying roles for multiple states, e.g. `ISRAELI_MILITARY`, `EGYPTIAN_MILITARY`, and `JORDANIAN_MILITARY`.

Example:

```
AMBASSADOR  [~GOV]
FOREIGN_MINISTER  [~GOVFRM]
PRESIDENT  [~GOVPRS]
AIR_FORCE  [~MIL]
OFFICIAL_MEDIA  [~GOVMED]
ATTORNEY_GENERAL  [~GOV]
FOREIGN_MINISTRY  [~GOV]
SECURITY_SERVICES  [~GOVFRM]
OPPOSITION_GROUPS  [~OPP]
MILITARY  [~MIL]
OFFICIALS  [~GOV]
OPPOSITION_JOURNALIST  [~MEDOPP]
POLICE  [~COP]
NEWSPAPER  [~MED]
ACADEMIC  [~EDU]
GUERRILLA  [~REB]
HUMAN_RIGHTS_ACTIVISTS  [NGM~]
HUMAN_RIGHTS_BODY  [NGO~]
```

Agents are coded when they occur immediately before an actor, immediately after an actor, or before a preposition and an actor. Assuming the code`ISR` was assigned to the actors

ISRAEL, ISRAELI and , PERES and the *.agents* dictionary contained PRESIDENT [∼GOVPRS] , the following phrases would all generate the code ISRGOVPRS

```
PRESIDENT OF ISRAEL
ISRAELI PRESIDENT
PRESIDENT PERES
```

At present only the prepositions defined in the system are IN_ and OF_.

Agent codes require a ∼ either at the beginning or the end of the agent code to indicate whether the actor code is added before the code or after it.[9] The placement of the actor code is always determined by the location of the ∼ in the agent code, not by whether the actor occurs before or after the agent in the text.

If an agent code is already part of the actor code, it is not added. For example if the *.actors* dictionary contained TAMIL_TIGER [LKAREB], and the *.agents* dictionary contained GUERRILLA [∼REB], the phrase "Tamil Tiger guerrillas..." would code as "LKAREB," not "LKAREBREB." A similar rule applies to prefixed codes. This applies only if the duplicate codes are aligned on the 3-character boundaries used in CAMEO coding, i.e. if there was an actor code "AREBKK," the suffix "REB" could be added to this.

If the command SET: CONVERT AGENTS = TRUE is used in the *.options* file, an agent which is not associated with an actor is converted to an actor, and the agent code (e.g. ∼REB) is used: see Section 8.2.6 for further discussion. This option is intended for situations where the location of the event is either known—for example coding from local news sources where knowledge of the location is assumed rather than explicit—or can be determined by processing with other programs. The risk with this approach is that a number of common nouns likely to be used as agents may also be found as adjectives in generally unrelated noun phrases, e.g. MILITARY_SALES or POLICE_STATE, so it should be used with caution.

5.4.1 "Former" agents

The word "Former" occurring immediately prior to the actor name or the agent name in an actor-agent pair automatically converts an agent into an "elite"—that is, the agent code is converted to the code ELI. This is an implementation of a CAMEO coding rule and avoids the alternative of putting the modifier "FORMER" in front of every actor and agent.

5.5 .Verbs Dictionary File

The *.verbs* file contains a combination of simple verbs (e.g. PROMISED) and verbs plus associated words (e.g. PROMISED FUNDS). For example

```
ACCEPT
```

[9] This is specifically designed for use with the CAMEO sub-state actor coding system, but presumably with the appropriate use of embedded markers and some perl programming, the output could be converted to other systems as well.

```
- PROPOSAL WAS * [081]
- * FORMULATION [041]
- REFUSED TO * [112]
- * INVITATION [082]
- * PROPOSAL [081]
- * CHARGES [013]
```

In this example, the *root* verb is `ACCEPT` and with stemming it will match `ACCEPT`, `ACCEPTS`, and `ACCEPTED`. The phrases that start with "-" are the associated phrases along with their codes, e.g.

```
ACCEPTED PROPOSAL
```

will be coded 081 (WEIS "agree") while

```
REFUSED TO ACCEPT
```

will be coded 112 (WEIS "reject"). The phrases tend to involve the direct object of the verb, though this is not always the case. The "*" indicates where the verb itself should appear, so the first phrase would match

```
PROPOSAL WAS ACCEPTED
```

If a root verb by itself uniquely identifies a code—which is commonly the case for WEIS and CAMEO—it is on a line by itself along with the code. A verb can also have a default code followed by exception phrases:

```
AGREE [081]
- * TO_LET_ESCAPE  [066]
- * LOAN           [071]
- * AID            [071]
```

In this case, the verb `AGREE` normally codes a 081 ("Agree") category but the `AGREED TO LOAN` would code 071 ("Extend economic aid") and `AGREED TO LET ESCAPE` would code 066 ("Release persons or property").

Tabari matches phrases in the order of longest to shortest; phrases are automatically sorted by length as they are entered. Details on the length calculations are described in section 5.7.1.

Note:: The verb token * shows the location of the verb root as a separate string and it cannot be attached to prefixes, suffixes or verb endings. The only characters that can legally precede and follow the * are the underscore and blank. Other characters will cause a syntax error. Prefixed and suffixed forms of verb roots (e.g. `PATROL/PATROLMAN`) should be entered as nouns, adjectives, or separate, null-coded verb roots.

5.5.1 Irregular Verb Forms

Beginning with Version 0.7, TABARI allows irregular verb forms to be explicitly specified. These are given in a set of {...} immediately following the main verb. The left-bracket { must be the first character of the line, and the verb-form line does not contain a code. Verb forms are separated by blanks and consequently must be either single words or phrases that are connected by underscores. For example:

```
BREAK_     [1717]
{BROKE_ BROKEN_}
- * TREATY [161]
_ * BONE   [182]
```

provides the irregular past tense and past participle forms. By using `BREAK_` rather than `BREAK=`, the third person singular `BREAKS` will be recognized. As will `BREAKED`, though this is unlikely to be encountered in a context that will cause errors. A useful list of irregular verbs can be found at `http://www2.gsu.edu/~wwwesl/egw/verbs.htm`.[10]

At the present time, additional verb forms cannot be added from inside the program, though the irregular verb forms are preserved in the flat verb file. All of the verb forms share the same pattern list, so the patterns can be edited from any form.

5.5.2 What is a verb?

In the formal languages used in computer programming and statistical packages, words are associated with a single meaning or a small set of related meanings. While this is often true when dealing with natural language—for example the English words "accuse" and "deny" are almost never incorrectly coded in news wire leads—there are exceptions. In English, there are *lots* of exceptions.

Those exceptions are due to the fact that English is primarily an "isolating" language where the grammatical role of a word can change depending on its position in a sentence.[11]

```
The treaty was broken
```

[10] Accessed 30 December 2008

[11] See Pinker (1994 , chapter 12) for an extended discussion of these issues. The problem is further complicated by the fact that English is derived from an inflected Germanic language (Old English) that evolved over a millenium into a language that is now largely isolating. Vestiges of inflection remain at quirky points in the language—for example "correct" English retains (barely...) the distinction between the nominative "who" and the accusative *whom* while dropping that distinction in "you/ye" , and the inflected "-ed" is used to indicate past tense. Relics of ancient alternatives to subject-verb-object ordering are still found in a small number of phrases that have been strongly-preserved through social convention, notably "With this ring I thee do wed." (subject-object-verb ordering. Warning: if you mention this fact the next time you hear this phrase in context, you will either get hit or be considered an utterly hopeless nerd.)

To further complicate matters, a series of arbitrary grammatical rules derived from Latin (an inflected language) were incorporated into formal English during the 18th century by socially-mobile London elites seeking to differentiate their use of the vernacular from that of the masses: The two notorious examples of this is the prohibition against split infinitives and against ending a sentence with a prepositions. These two cases of "bad grammar" beloved of "grammar Nazis" are in fact completely consistent with the underlying grammar of English, but not that of Latin.

the word *broken* is a verb, while in the sentence

```
The diplomats discussed the broken treaty
```

the word *broken* is an adjective. In contrast, in inflected languages—for example Old English, Latin, Russian or American Sign Language—a root is usually modified by prefixes, suffixes or vowel changes to indicate that it is being used for a different purpose. For example, in Latin either of these phrases correspond to "Man bites dog":

```
Homo canem mordet
Canem homo mordet
```

The order of the subject and object are irrelevant; to create the sentence "Dog bites man," the nouns themselves must be changed:

```
Canis hominem mordet
Hominem canis mordet
```

English, in contrast, uses the same word whether a noun is a subject or object[12] and the role of the noun as subject versus object is determined by its position. Those positions can be changed by the use of passive voice:

```
The dog was bitten by the man
```

but "dog" and "man" are still unchanged.

Words can also change from verbs to nouns without modification:

```
When Jill returned from the car wash, she parked her car in the drive.
```

The only indication here that *wash* and *drive* are nouns rather than verbs comes from their position in the two prepositional phrases. In the sentence

```
Jill said ‘‘I’m going to drive the car across town so I can wash it’’
```

"drive" and "wash" are verbs. As Pinker notes, English contains a extraordinary number of *homonyms*—words that sound identical but have different roles and meanings depending on how they are used in a sentence.

What does this mean for TABARI? It is not a full parser, and generally assigns each string of characters to only a single class of words. This can lead to ambiguities. However, TABARI's patterns are oriented towards looking at the location of various words with respect to each other, and this facility is very important in determining the meaning of words in English. The upshot is that you will find that dictionary development involves a lot of effort in finding

[12] It retains this distinction in pronouns: "I" versus "me", "she" versus "her" and so forth.

phrases that must be null-coded[13] or otherwise assigned an interpretation distinct from the root. This is not a feature of the program but rather a feature of the English language. On the positive side, after you have worked with TABARI for some time, you will have a much deeper appreciation of how the grammar of English actually works, as distinct from the much more simplified grammar that is commonly taught to native speakers. Joe Pull's "Ode to Coding" (Appendix F) discusses this is detail.

The key to the effective use of phrases is to make sure that phrases are sorted into the transitive verb that determines the event code. Do not code indicators of tense (e.g. HAS, WILL) or forms of "to be" (IS, WAS etc.) as verbs; use the transitive verb that indicates the action (this will often be an infinitive, e.g. in `WILLING TO NEGOTIATE` the verb is `NEGOTIATE` rather than `WILLING`). Having done that, the next step is to null-code phrases where the word is not being used as a verb, either because it occurs in an idiomatic expression or because the string matches a homonym that is not a verb.

In our work with news wire reports on the Middle East, two words stand out as particularly problematic: FORCE and ATTACK. Both words can be used either as nouns ("A guerrilla force launched an attack") or as verbs ("Rebel radio said guerrillas would attack in order to force concessions") and occur frequently in reports about military conflict. FORCE and ATTACK are further complicated because they can be used to refer both to verbal actions (persuasion and criticism) and to uses of force; both uses are common in news reports. In our dictionaries, a large number of patterns are associated with each of these words to try to distinguish the noun usage from the verb usage. ARMS, BATTLE, FIRE, HELP, ORDER, PLAN, PLEDGE, STRIKE and SUPPORT are other examples of words that are used both as verbs and as nouns.

A final complication arises from the tendency of the English to base some adjectives on verbs—for example "the wrecked car" or "the broken treaty." Usually these do not cause problems—in fact occcasionally they allow TABARI to assign the correct code based on an adjective rather than a verb—but they should be kept in mind.

5.5.3 Pattern Matching in Verb Phrases

Verb phrases can use either simple or composite patterns. By default, phrase matching stops when a conjunction (AND_, BUT_, OR_ and NOR_) is encountered, unless the conjunction is in a compound actor phrase. In other words, a verb phrase cannot match words that occur after a conjunctions. Phrase matching also stops at commas, which eliminates matching inside, or across, subordinate phrases.

By default, pattern matching skips over intermediate words: the phrase

```
PROMISED
- * DOLLARS
```

will match the phrases

```
PROMISED TWO MILLION DOLLARS
PROMISED A GRANT OF 30-MILLION DOLLARS
```

[13] English also, of course, allows verbs to be created from nouns:
null-cod'ed. *verb.* 1. the act of assigning a null code [- - -] to a verb phrase in TABARI.

If two words *must* be consecutive, connect them with an underscore, e.g.

```
BURNED_DOWN
```

The underscore character and concatenation can be used to force a word or set of consecutive words to be directly before or after the verb; this is particularly useful for words that can be interpreted as either nouns or verbs. For example:

```
ATTACK [122]
- * CRITICISM OF   [042]
- *_HELICOPTERS    [- - -]
...
POUND [223]
- BRITISH_* [- - -]
```

where "[- - -]" is a "null code" that causes a phrase *not* to be coded; it is described in section 6.2.

The ending construction "XXXX~_YYYY" means that XXXX can have a wild-card ending but must be followed immediately by YYYY.[14] For example, the pattern

```
- * BACK~_POLICY [062]
```

would match the phrases

```
BACK POLICY
BACKS POLICY
BACKED POLICY
```

but fail on the phrase

```
BACKED ANOTHER POLICY
```

due to the intermediate term.

In summary, TABARI has the following possibilities for partial matches (wild-card endings) and consecutive placement:

XXXX YYYY	XXXX can partially match, YYYY does not need to follow immediately
XXXX~_YYYY	XXXX can partially match, YYYY must follow immediately
XXXX_ YYYY	XXXX must match exactly, YYYY does not need to follow immediately
XXXX_YYYY	XXXX must match exactly, YYYY must follow immediately

[14] Curiously, this logical possibility was missing in KEDS ...

5.6 Synonym Sets

Beginning with Version 0.8, TABARI allows the specification of synonym sets ("synsets") that can be used in patterns. These are a generalization of the older alternative patterns facility described in Section 5.7. Synsets slightly more efficient than alternative patterns, and certainly a simpler notation if a set of synonyms is going to be used in more than one pattern.

Synsets are specified in a `<SYNSET>...</SYNSET>` section at the beginning of the *.verbs* file. The designation for the sunset begins with the character `&` and the synonyms follow one per line, prefixed with `+` in a syntax similar to the synonyms used in the *.actors* file.

```
 <SYNSETS>
&TIETYPES
+MILITARY
+CULTURAL
+ECONOMIC
&RELATIONS
+RELATIONS
+&TIETYPES_EVENTS
+OFFICE_OF_&TIETYPES_AFFAIRS
&CURRENCY
&CURRENCY
+DOLLAR
+EURO
+GOLDEN_GOBLIN_GALLEONS
+KRONER
+YEN
+SWISS_FRANC
</SYNSETS>
```

To specify a synset in a pattern, simply use the designator where the set of synonyms can occur:

```
ABANDON  [345]
- * EFFORT  [987]
- * DIPLOMATIC TIES [901]
- * &TIETYPES TIES [902]
CONTRIBUT  [070]
- * MILLIONS_OF_&CURRENCY [905]
- * &CURRENCY [903]
- * TROOPS  [071]
```

Synsets follow the same connector rules as used in other patterns, except that a synset designator cannot be stemmed and the automatic noun/verb endings are not applied. The designated actor tokens **$** and + (Section 5.7.3) are allowed, although this feature has not been extensively tested. As shown in the example, synset patterns can refer to other synsets, provided these have been declared earlier. However, this cannot be used recursively and a pattern within a synset that refers to that synset designator will generate an input error.

Alternative patterns (Section 5.7) can contain synsets except for an isolated synset designator as the final element of the alternative.[15] However, synsets *cannot* contain alternative patterns.

Synsets cannot be added from inside the program, though patterns using a previously specified designator can be added.

5.7 Alternative Patterns

Patterns can contain sets of alternative matches; these are done using the following notation:

```
{ pattern1  | pattern2  | ... | patternN  }
```

This will match a string containing

pattern1 OR pattern2 OR ... OR patternN .

The {, |and } should be space-delimited. An underscore on a *word* inside an alternative set forces the string to match completely—in other words, it deactivates stemming. To force the element following a set to be consecutive, put an underscore *after* the closing bracket. For example

```
{ WAS | IS | WILL_BE }_HERE
```

is equivalent to

```
{ WAS_HERE | IS_HERE | WILL_BE_HERE }
```

There can be a connector prior to the pattern:

```
MILLIONS_OF_{ DOLLARS | EUROS | YEN }
```

is equivalent to

```
{ MILLIONS_OF_DOLLARS | MILLIONS_OF_EUROS | MILLIONS_OF_YEN }
```

[15] If you encounter this contingency, just move the synset to earlier in the list of alternatives, or terminate the alternatives with a nonsense string such as `HVORFOR_IKKE`. While I have not tested it extensively, the existing system seems to work except in this situation and attempting to modify the code handling alternatives—which is completely separate from that handling synsets—is potentially bug-inducing.

5.7.1 Length Calculations

.

Patterns are tested in reverse order of length—the system attempts to match longer patterns first. When length is calculated, open spaces (blanks rather than _) and tokens ($, +, %, *, and ^) are not included in the count, so that

```
- + * TO THE NEXT DAY $
```

has a shorter length than

```
- * THEIR MEETINGS
```

because the first pattern has only 12 alphanumeric characters, whereas the second pattern has 13. When a *.verbs* dictionary is saved in TABARI , the ordering of the pattern list reflects the calculated length. If a dictionary is modified outside of the program, the patterns will be stored internally in the correct order, even if the order was not correct in the input file. In other words, when adding phrases, don't worry about calculating the correct order—TABARI will handle this for you.

The length of a pattern containing alternatives is the length of the fixed text, plus the length of the longest alternative. When a match is made with a pattern containing alternatives, the number of characters in the actual match is calculated, and all additional patterns longer than that length are evaluated before the match to the alternative is accepted.

5.7.2 A Note on Compound Verbs

TABARI is generally pretty good about correctly distinguishing between compound noun phrases and compound clauses. However, it sometimes runs into problems with compound verbs, e.g.

```
SHOT
- *_AND_KILLED
```

In general, this type of construction should be avoided because the conjunction AND_ will not be eliminated. This, in turn, means that the sentence will be incorrectly split into clauses at this point. If a conjunction is used in a compound verb, include both parts as a distinct *verb root* , rather than as a phrase, in order to keep the sentence from being coded as compound. Compound verbs become particularly problematic when a phrase occurs before the verb. The construction

```
  KILLED [678]
- SHOT_AND_* [123]

SHOT [345]
```

creates two separate events:

```
  SHOT [345]
SHOT_AND_KILLED [123]
```

because the SHOT_AND_* patterns is matched across the conjunction, and SHOT is coded because it is in the first part of a compound sentence.

5.7.3 Designated Actors

Phrases can specify the location of the source and target by using the tokens **$** and + respectively. For example, the phrase

```
ADVISE
- +_WAS_*_BY_$
```

would do the correct assignments on the phrases

```
Egypt was advised by the United States
```

$ and + are assigned to the first actor that is encountered in the appropriate location. In other words in matching

```
- + WAS * BY $
```

the system will search for the *target* using the first actor before WAS...ADVISE and search for the *source* using the first actor after BY. In this example, the actor locations are used to reverse the source and target when a verb phrase is in passive voice.[16]

The symbol % specifies a *compound actor* that should be assigned to both the source and target; it works with either coded or parsed compounds. This is typically used when dealing with consultations:

```
Representatives of Syria and Jordan will meet In Cairo
```

The phrase

```
MEET
- % * IN   [031]
```

would do the correct assignments

[16] In TABARI , passive voice is handled automatically by a parsing rule, so this pattern is unnecessary. However, it is the most straightforward way to explain the principle...

```
SYR  JOR 031
JOR  SYR 031
```

rather than

```
SYR  EGY  031
JOR  EGY  031
```

If no compound actor is found in the sentence, the phrase containing the compound assignment fails, and then any shorter phrases in the verb's phrase list will be checked.

The symbol ^(caret) can be used in a pattern to skip over an actor without assigning it to a source or target. For example, the sentence

```
Russia's president will meet in Switzerland with the U.S. Secretary of
State
```

coded with the pattern

```
MEET
 - * IN ^ WITH + [031]
```

will generate the event

```
RUS USA 031
```

rather than

```
RUS SWZ 031
```

A $ or + token must follow the ^. Multiple skips are allowed by using multiple ^tokens. A single ^will skip over both actors in a compound actor phrase, so

```
Russia's president will meet in Bern and Geneva with the U.S. Secretary
of State
```

will still generate the event

```
RUS USA 031
```

because the + token will match `U.S.` rather than `Geneva`.[17]

The source and target tokens are optional; if they are not specified then default rules (see section 5.8) are used to locate the source and target. However, if a source or target is specified in a phrase and then not found in the appropriate location, the phrase as a whole fails. Similarly, if a string in a phrase is missing, then the phrase as a whole fails.

[17] Bug or feature?—you decide...

5.7.4 Precedence in phrase matching

A phrase is matched in the forward direction from the beginning of the clause. Each element of the phrase is matched in order. For example, if one had a sentence of the form

```
<actor1> s1 <actor2><verb>
```

and the phrase

```
- + s1 *
```

the target would be assigned to $< actor1 >$, because this occurs before `s1`.

If the sentence had the form:

```
s1 <actor2> <verb>
```

the phrase match would fail, because there is no actor prior to `s1`.

Because phrase matches are stopped by conjunctions, a phrase match will only apply inside a clause in a compound sentence.

5.8 Default Actor Search Rules

If a verb phrase does not designate a source and target—which is the usual situation—these are filled in using default searches. The *default source search* is done first: it starts at the beginning of the text and the first actor encountered *prior to the verb* is designated the source. If a target has already been found through a pattern match, the source is the first actor with a code distinct from the code of the target. If there is no actor prior to the verb, the source is set to missing.[18]

Following the assignment of the source, the *default target search* then tries to find an actor *after the verb* that has a code distinct from the code of the source. If no such actor is found, a search is made for the first actor *before the verb* that has a code that is distinct from the code of the source.

If no such target is found and the source is compound, the compound source is also designated as the target; the system automatically eliminates codes which would have the same source and target. Actors with null codes are ignored in both searches.

Finally, if a default source and/or target code has been specified using a `DEFAULT` command in the *.options* file (see Section 8.5), that code is assigned.

[18]The absence of a source prior to the verb usually means that the subject of the sentence is not in the dictionary: in our coding we skip such sentences by using a `COMPLEX: NOACTPRIOR` condition in the *.options* file. This command is discussed in section 8.7

Note:: Assigning a default source will guarantee that the first verb in each clause (or the sentence) is coded. Because English words that look like verbs may actually be nouns or adjectives (see the discussion in section 5.5.2 above) this can result in coding that is significantly different from that generated by the default options, where a verb must be preceded by an actor.

Chapter 6

Special Purpose Codes

6.1 Purpose

Most of the codes used in the TABARI dictionaries are the simple codes assigned by the coding scheme to actors and events. However, TABARI has three special codes—the null, discard and complex codes—that can be attached to phrases to change how TABARI deals with a sentence. Codes can also be assigned priorities, they can be restricted in time, and multiple events can be coded from a single phrase.

The null, discard and complex codes can be can be included in either the *.actors* or *.verbs* files. In the *.actors* file, they can also be included as date-restricted codes. In the *.verbs* file, they can be assigned to either roots or phrases. These codes do not generate events.

> **Note:** In the *.verbs* file, it is generally better to assign discard and complex codes to roots rather than to phrases. When the codes are assigned to a root, a sentence meeting the complex or discard criteria can be identified as soon as all of the words in the text are classified. If the codes are assigned to phrases, the verb roots in the text must be evaluated first, and in some instances the phrases of a root may not be evaluated, for example if the verb occurs late in a phrase or evaluation is halted by a dominant code. The only advantage to assigning the codes in phrases is to keep all of the phrases dealing with a verb in one place. This problem does not apply to null codes, which simply eliminate a phrase from being coded and are very common in the dictionaries.

6.2 Null Code [- - -]

The null code [- - -][1] is used to eliminate phrases that would otherwise be confused with actors or verbs. Phrases with null codes can be in either the *.actors* or *.verbs* file. Experience in both the KEDS and PANDA projects has shown that finding the appropriate phrases to be

[1] The null code consists of three *consecutive* hyphens (ASCII 45): these tend to mush together in printed text so they are separated by spaces here.

null coded is a key element in bringing the accuracy of a coding system above the 75% level. Null-coded phrases are a substantial part of the dictionaries of both projects.

Words that have only a null code—all actors and null-coded verbs without patterns—are converted to type "null" and are henceforth not considered actors or verbs. The choice of putting null-coded words in the *.actors* or *.verbs* file is one of convenience and does not affect the efficiency of the coding; usually it is best to put a null-coded word in the list containing words with a similar stem.

Example:

Using the actors

```
ISRAEL [ISR]
WEST BANK [PAL]
```

the phrase

```
Israeli-occupied West Bank and Gaza
```

will generate both ISR and PAL as actors. By adding the null code

```
ISRAELI-OCCUPIED [- - -]
```

only PAL is generated as an actor.

Example:

The phrase

```
The head of Lebanon's Catholic community
```

generates a verb identification for HEAD, since it is more commonly a verb, e.g.

```
Egyptian President Mubarak headed for a meeting with...
```

Adding the code

```
THE_HEAD_OF [- - -]
```

eliminates this problem.

In older dictionaries, many of the nulled-coded entries are nouns and adjectives that could otherwise be mistaken for verbs and actors due to stemming. As discussed in section 5.3.2, it is better to move these into the `<NOUN>` and `<ADJECTIVE>` lists because this will allow them to be used in the identification of compound phrases.

6.3 Discard Code [# # #]

Source texts are likely to contain some events which involve multiple international actors but which are non-political: sports events are the most common; traffic accidents and natural disasters involving tourists a close second. These can be automatically discarded by using the discard code **[###]**.

Example:

```
MARIJUANA  [###]
MICHAEL\_JORDAN  [###]
FRANK\_JORDAN    [###]
REAGAN [### > 890120] [USA]
SOCCER  [###]
WORLD\_CUP [###]

KILLED [223]
- * IN ACCIDENT [###]
- WILDLIFE * [###]
```

The presence of a discard condition causes most of the other processing of the text to be terminated. Discards should therefore be used only when you are certain that the record contains nothing of interest; if you might want to look at it, use a **complex** code instead (see section 8.7).

6.4 Complex Code [+++]

The complex code can be used to identify words and phrases in the *.actors* or *.verbs* dictionaries that will cause the program to automatically divert the text to a *.complex* file, provided the **COMPLEX:** command has been included in the *.Options* file.

For example, the word GEORGIA can refer to a violence-prone region afflicted by ethnic conflict and political demagogues in the southern region of the former Soviet Union, or to a violence-prone region afflicted by ethnic conflict and political demagogues in the southern region of the United States. To a human coder, the choice of the two GEORGIAs is usually clear from the context of the text, but it may be difficult to distinguish the cases using only the sparse parsing available in TABARI. Diverting the cases to the *.complex* file allows these to be later coded by a human. In some coding systems, the complex code might be used with verb phrases or idiomatic expresses whose coding almost always require information beyond that available in the literal structure of the sentence.

6.5 Special Purpose Codes for Actors

6.5.1 Coded Compound Actors

In some circumstances, it is useful to have a single phrase generate multiple actor codes. This is referred to as a *coded* compound actor, as distinct from a *parsed* compound, which is based on the structure of the sentence.

Coded compound actors are entered by separating the actor codes with a slash ("/"). For example

```
EAST_AND_WEST_GERMANY [GME/GMW]
NORTH_AND_SOUTH_KOREA [KON/KOS]
```

This method can also be used to expand the membership of alliances when that is appropriate:

```
G7 [USA/GMW/FRN/ITL/UK/JAP/CAN]
```

Note that coded compound actors can generate very large numbers of events when combined with other compound actors. For example the text

```
Ministers from Poland, Hungary and the Czech Republic attended
the meeting of the G7
```

would produce 21 events. TABARI currently allows for a maximum of 64 events. Some combinations of compound events can exceed this capacity. In that situation, the program codes the first 64 events and then issues an error message, then continues coding. Lengthy lists of actors are not uncommon in wire service stories, particularly those dealing with meetings.

Programming Note: The 64 event maximum can be increased by changing the constant `MAX_EVENTS` in the TABARI.h header file. An even more elegant solution would be to allocate new storage dynamically when this situation occurs.

6.5.2 Date Restricted Codes

In a data set covering a long time period, some individuals will change their role. For example, Boutros Boutros-Ghali was in the foreign ministry of Egypt prior to 1 January 1992, then became Secretary General of the United Nations on that date. To code the period 1990-1994, Boutros Ghali needs to be assigned the code EGY for part of the period, and UNO for the remainder.

This problem is handled by assigning multiple codes, and then putting a date restriction on each code. Date restrictions have the following formats; each restricted code goes in its own set of brackets.

<YYMMDD Assign the code for events prior to and including this date

>YYMMDD Assign the code for events after and including this date

YYMMDD-YYMMDD Assign the code for events between the two dates

Using these formats, the actor entry for Boutros-Ghali would be

```
BOUTROS-GHALI [EGY < 911231] [UNO > 920101]
```

A code with *no* date restriction will be used as a default; this must be the *final* code in the list.[2] If Boutros-Ghali has a UN position until 31 December 1999, then returns to a position in the Egyptian government, the following coding would work:

```
BOUTROS-GHALI[UNO 920101-991231] [EGY]
```

The codes used with date restrictions can involve compound codes as well as simple codes.

Notes on date restrictions:

1. Years 00 through 30 are assumed to be 2000 through 2030, so in the TABARI coding system the year "01" is greater than "99." If you are still using TABARI after 2030, get a better program... [3]

2. If a date cannot be interpreted (e.g. is less than six characters; contains non-numeric characters or month/day out of range), the record is skipped. A warning error is displayed during manual coding with the option to quit. In both manual and autocoding modes, a message is written to the $< errorfile >$ provided one has been specified.[4]

3. The total length of the codes and date restrictions must be less than or equal to 256 characters; any codes beyond this length will be ignored.

4. If there are inconsistent restrictions, for example

   `[XXX <920101] [YYY <930101]`

   the first restriction that is satisfied will be applied. Restrictions are evaluated in the order they are listed, except for the default (unrestricted) code, which is applied only if none of the restrictions are satisfied. If none of the restrictions are satisfied and there is no default code, the null code is assigned.

   This feature can be used to provide very general date restrictions. Consider an individual who was born in 1930, became politically active around 1950, and remainded

[2] This is a change from KEDS , where a default code could occur anywhere in the list. Consequently some date-restricted actor entries developed in KEDS may not work correctly in TABARI unless this is corrected.

[3] **Programming Note:** 8-digit codes, which are required for "Y2K" compliance in some contracts, can be accomodated by changing the variable `GlobalFlags.f4DigitYear` in the file TABARI.h. In fact, changing the pivot date to something later than 2030 only involves the modification of a couple of lines of code.

[4] In KEDS and in TABARI versions prior to 0.7.5, the date 140101 was assigned to any dates that could not be interpreted, but the sentence was still coded on the assumption that these records would be identified later. Using the $< errorfile >$ provides the same information but isolates the potentially problematic records—if the date is bad, the text is likely to be bad as well.

active until their death in 2005, served in government 1960-1965 and was active in the opposition for 1966-1985. The date restrictions

```
[XXXGOV 600101-651231] [XXXOPP 660101-851231]
[XXXELI 500101-050630] [---]
```

codes the active periods in government and opposition, labels the individual as an "elite" during the remainder of their adult life, but null-codes them if they are mentioned after their death.

5. If only a single date restriction is used, e.g.

   EAST GERMANY [GME <901103]

   the system will automatically add a null code as the default:

   EAST GERMANY [GME <901103] [- - -]

   These null codes will appear in the modify dialog and in the *.actors* file when it is saved.

6.6 Special Purpose Codes for Verbs

6.6.1 Subordinate Codes

A subordinate code is used only if no other events are found. It is denoted by adding ? to the end of the event code, e.g.

```
SAID [023?]
```

The main use for this is coding attribution.[5] Many news wire stories begin with a sentence having the structure

```
<actor_1> SAID <pronoun> <verb> <actor_2>
```

For example

```
George Bush said he Rejected Syria's assertion...
```

The relevant event is

USA rejected Syria

rather than

USA said Syria

[5]Attribution can also be coded as a separate field: see section 10.4.

The combination of coreferencing the pronoun HE and using a subordinate code will handle this.

Subordinate events are coded only if there are *no* non-subordinate events in the sentence—a single non-subordinate event will eliminate all subordinate events.

6.6.2 Dominant Codes

A dominant code is the opposite of a subordinate code: a pattern with a dominant code is always coded in a clause, even it it occurs after some other pattern. When a dominant code is found in a sentence, only that event and other events with dominant codes are coded. A dominant code is specified by adding ! to the end of the code; for example:

```
  REJECT
- * EFFORT BY [111!]}
```

Dominant codes are largely a legacy feature: we used these in the early years of KEDS when the dictionaries were relatively small and some of the coding accuracy depended on coding nouns as if they were verbs. In recent years it has been used very little now in our work.

6.6.3 Paired Codes

There are an assortment of circumstances where the event coding schemes generates symmetric events of the form

```
<Actor_1> Event_1 <Actor_2>
<Actor_2> Event_2 <Actor_1>
```

For example, in the WEIS coding system a meeting between Israel and Egypt would generate the pair

```
ISR  EGY  031 (meet with)
EGY  ISR  031 (meet with)
```

A visit by a Jordanian official to Syria would generate the pair

```
JOR  SYR  032 (visit; go to)
SYR  JOR  033 (receive visit; host)
```

These combinations can be coded automatically by using a pair of codes separated by a colon (:); for example

```
FLEW
- $ * TO + [032:033]
```

would do the visit-and-receive pair, while

```
MEET [031:031]
```

generates two events with the same code but with the source and target reversed.

Note: In our tests comparing TABARI to human coders, the program's consistency in correctly assigning multiple codes goes a long way towards compensating for its inability to interpret complex sentences. Human coders tend to miss some of the source-target combinations implied by paired codes and compound actors; TABARI gets them all. In Reuters and AFP lead sentences, these situations generate a *lot* of events.

Chapter 7

Parsing: How TABARI Looks at a Sentence

7.1 Purpose

In order to code event data, TABARI must parse a sentence to identify the subject, verb phrase, and object of the sentence. To do this, TABARI employs a number of simple but general grammatical rules for parsing English sentences. These are sufficient to handle most of the sentence structures encountered in news wire lead sentences, and do a reasonably good job on other literal text.

TABARI is not a general purpose natural language parser: The problem of creating a general machine parser, even for a single language such as English, has defied the efforts of linguists and computer scientists for forty years[1], and TABARI is no breakthrough in this regard. Instead, the approach taken by TABARI is to use a small number of fairly robust parsing techniques that provide sufficient information about a sentence from newswire reports to enable event data to be correctly coded most of the time. If the source text uses complex sentence structures—as might be encountered, for example, in political speeches or legal documents—then TABARI is probably not the appropriate coding program.

TABARI's parser assigns a number of different types to words in a sentence, as well as delineating several different clauses. Some of these tags appear when the `P)arse` option is used; others are purely internal.

[1] See Pinker (1994) for a wide variety of examples of why machine-parsing is a more difficult problem than it first appears to be.

TABARI Word and Clause Types

Words	
Null	Word has not been classified (not displayed)
Litr	Literal: string of characters that occurs as part of an actor, verb or pattern
Actor	Actor
Agent	Agent
Verb	Verb
Time	Time-shifting word
Attrib	Attribution word
Determ	Determiner: A_, AN_, THE_
Noun	Explicit noun; verb shifted to noun by determiner or capitalization; and null-coded actors
Adjctv	Adjective
Auxil	Auxilary verb—WAS_, WERE_, BEEN_—used in passive voice detection
Byword	BY_, used in passive voice detection
Comma	Comma ",", used in subordinate clause detection and compounds
Pronoun	Pronoun: HE_, SHE_, HIM_, HER_, IT_, THEY_, THEM_, THEIR_
Conj	Conjunction: AND_, BUT_, OR_, NOR_
Number	Strings that begin with a digit except when part of a phrase, as in Group of 77
Issue	Word in ISSUES set
NullTag	Type was cancelled because it overlapped with a subordinate clause

Clause tags	
Clause	Clause in compound sentence
Compound	Compound noun phrase
Reference	Pronoun coreference
Subord	Subordinate clause
Replace	Used to standardize actor names in XML input
NullTag	Deactivates tags in subordinate clauses (not displayed)
Halt	End of text (not displayed)

7.2 Color-Coded Parse Display

One of the more popular items in KEDS that was missing from earlier versions of TABARI was the color-coded parse, which shows the sentence as TABARI "sees" it. This is particularly useful when puzzling through odd coding, since it makes obvious, for example, nouns that have been tagged as verbs (or vice versa), and text that has been eliminated (or not) in subordinate clauses.

Because the display capabilities of the TERMINAL are limited, TABARI provides this in a separate file named TABARI.Parse.html. In order to access this after you have used the `P)arse` command, start a browser (for example, SAFARI, FIREFOX, or MS-EXPLORER) and open the file (not location) TABARI.Parse.html which will be in the folder where TABARI is running. The file is re-generated each time you use the `P)arse` command, so once you've got it running, use "`Reload`" (Cmd-R in FIREFOX) to display the revised file.

Once you are familiar with TABARI's parsing and word types, the display should be more or

less self-explanatory. {...} indicates clauses; <...>indicates compounds, and subordinate clauses are grey with a line through them. The following table gives the key to the colors of the various word types

Color	Label	Content
Blue	verb	Verb with code assigned to root
Red	actor	Actor
Black	agent	Agent
Maroon	determ	Determiner (A, AN, THE)
Magenta	noun	Noun
Lime	prnoun	Pronoun
Green	conj	Conjunction
Black	adjctv	Adjective
Gray	null	null-coded root; this will only be coded if one of its phrases is matched
Black	litr	"literal": a word or stem found in a pattern but but not in an actor, verb, noun or adjective list
Black		Unclassified words

In the "Parsed text" display, compound actor, noun and adjective phrases are delineated with `<...>` and clauses are delineated with `{...}`. In the "Parse table", clauses are delimited with XML-like tags: `<...>` marks the beginning of the clause and `</...>` marks the end. The clause tags are:

compnd	compound actor, noun or adjective phrase
clause	clause in a compound sentence
subord	comma-delimited subordinate clause

If a root contains multiple words connected by underscores, all of the words in the root will be colored. If the multiple words are not connected by underscores, only the first word will be colored. The "Parse table" display will show the complete root.

Coreferenced pronouns are followed by a reference to the index of the actor they refer to: the index is the left-most column in the parse table.

7.3 Automatic Input Filtering

The following changes are made in the text before it is processed:

1. All characters are converted to upper-case and any diacriticals are removed (i.e. ö becomes O, å becomes A, é becomes E and so forth).
2. Periods following lower-case letters are replaced with blanks; this means that the punctuation at the end of a sentence is eliminated but periods in abbreviations (e.g. U.N.) are retained.
3. '?', '!' and '-' are replaced with blanks.[2]
4. Commas inside numbers are eliminated, so "1,500" becomes "1500".
5. Semicolons (';') are replaced with commas.
6. Commas and double-quotes (") are delimited with blanks, so the sentence

   `Clinton◇responded,◇‘‘I don’t know.’’`

   becomes

   `Clinton◇responded◇,◇‘‘◇I don’t know◇’’◇`

   where ◇ represents a blank.

All of this filtering is done before the text is entered into the system, so these changes will be reflected in the text displayed on the screen. Generally TABARI formats the source text to fit the size of the text window displayed on the screen. If you want to have a "hard-coded" return in the text, use "◇\n" (note the space before the "\n ")

Any text between the delimiters /*...*/ will not be coded; this allows irrelevant material (for example subordinate clauses) to be eliminated from coding and for comments to be added to the text, as is done in the *.demo* texts.

TABARI does nothing with the following punctuation issues:

1. Hyphenated words are the end of a line, for example

   ```
   ..... confronting antidisestab-
   lishmentarianism, the government .......
   ```

   These should be combined into single words when filtering the original text.
2. Any other forms of hyphenation, e.g. "Arab-Israel", "al-Asad"
3. Apostrophes, e.g. "didn't", "Shi'a"
4. The asymmetrical "smart quotes" **“** and **”** – if these occur in your source text, replace them with a simple double-quote " using a word processor. The same goes for "smart apostrophes". This is most easily done as part of the initial filtering process. These are almost never found in text downloaded from an HTML file, but are common in text transferred from word-processing documents.

[2] In versions prior to 0.7.3, '-' [hyphen] was preserved. The change was made to deal with inconsistencies in transliteration—e.g. "al-Assad" versus "al Assad"—and more generally because a hyphen could almost always be unambiguously replaced with an underscore "_" in the dictionaries. Versions after 0.7.3 do the hyphen-to-underscore replacement when dictionaries are read.

7.4 Nouns

As discussed in section 5.5.2, one of the major challenges involved in parsing English is determining whether a word is a noun or a verb. TABARI designates words that are in the *.verbs* dictionary as "nouns" under the following circumstances:

a. word was preceded by THE, A or AN

b. word was capitalized in the middle of a sentence

This handles about 80% of the required cases of noun-verb disambiguation.

Actors are also changed into "nouns" when they are null-coded. Nouns are not included in the calculations of complexity conditions, so this means that a word which is simply being eliminated to avoid confusion with another word with the same stem—the purposes of most null-coded words in the dictionaries—will not cause the text to be rejected as too complex. When the word is in the *.actors* dictionary, date-restrictions (see section 6.5.2) are first resolved when determining whether the word is null-coded.

The capitalization rule [b] is not applied if the record contains ALL CAPITAL LETTERS, a situation found in some older new service reports. In earlier versions of TABARI, files containing all-capitalized text had to be designated using the `SET: ALLCAPS TRUE` command in the *.options* file, but this is now handled automatically. Capitalized and mixed-case records can also be combined in a single file.

7.5 Parsed Compound Actors

TABARI recognizes compound nouns and does the appropriate duplication of events. This is a *parsed* compound actor, as distinct from the *coded* compound discussed in section 6.5.1. A compound actor is treated as a regular actor during the coding (e.g. during the search for sources and targets) and then expanded during the creation of events.

Parsed compounds have the form

$$< np_1 > \_AND\_ < np_2 >$$
$$< np_1 > \_,\_ < np_2 > \_,\_ \ldots\_ < np_{n-1} > \_AND\_ < actor_n >$$

where

$$np = < actor > \mid < adjctv >< np > \mid < determ >< np >$$

In other words, the compounds of a compound phrase can be either actors or actors preceded by any number of adjectives or determiners.

Example:

The sentence

```
The United States and Egypt approved of efforts by Israel and Jordan
to...
```

would code to

```
USA  approved  ISR
USA  approved  JOR
EGY  approved  ISR
EGY  approved  JOR
```

With the verb pattern

```
CLASH
-% * [221:221]
```

the sentence

```
Palestinian Police and Israeli Settlers Clashed...
```

would code to

```
PSEPOL  <221>  ISRSET
ISRSET  <221>  PSEPOL
```

The compound phrase

```
Israeli Prime Minister Sharon and President Bush
```

would be recognized assuming the actors file contained

```
<ADJECTIVE>
PRIME
MINISTER
PRESIDENT
</ADJECTIVE>
BUSH [USA]
ISRAEL [ISR]
SHARON [ISR]
```

In this example, the system will identify all of the actors in the sequence, but will subsequently filter out the duplicates, so this will be treated the same as

```
ISRAEL AND BUSH
```

However, if there are distinct codes for the actors in the noun phrases

```
ISRAEL [ISR]
SHARON [ISRGOV]
BUSH [USAGOV]
```

the ISR and ISRGOV code generate distinct events, so this becomes equivalent to

```
ISRAEL, SHARON AND BUSH
```

Depending on your application, this may or may not be what you want, and some subsequent filtering may be required.

Note: The system will not detect a parsed compound actor in combination with a coded compound actor containing a conjunction. This is not done as a general parsing rule because coded compounds—e.g. `G7, MAJOR_POWERS, GANG_OF_FOUR`—do not necessarily contain embedded conjunctions. For example, if `EAST_AND_WEST_GERMANY` was a coded compound, then the phrase

```
Poland, Hungary, East and West Germany agreed to coordinate
```

would not correctly identify the compound subject because the `AND_` is absorbed in the coded compound. To get around this problem, use a word processor to replace `EAST AND WEST GERMANY` with `EAST GERMANY AND WEST GERMANY`.

7.5.1 Compound Noun and Compound Adjective Phrases

TABARI recognizes compound noun and compound adjective phrases: these are used to eliminate conjunctions that would otherwise cause the sentence to be split into multiple clauses (see section 7.6).

A compound noun phrase is defined by

```
<noun> <conj> <adjctv>* <noun>
```

where `<adjctv>*` means zero or more adjectives. This allows phrases such as "`bombs and rockets`" or "`black sheep and lazy white goats`" to be ignored when dividing the sentence into clauses, provided the actors file contains

```
<ADJECTIVE>
LAZY
WHITE
BLACK
</ADJECTIVE>
<NOUN>
BOMBS
```

```
GOATS
ROCKETS
SHEEP
</NOUN>
```

Note that the adjectives prior to the first noun don't need to be in the list. More generally, note that this feature doesn't mark anything that will eventually be coded; instead it simply *prevents* a conjunction from marking a clause break.

A compound adjective phrases is defined by

```
<adjctv> <conj> <adjctv>
```

This allows phrases such as "`black and white goats`" to be ignored when dividing the sentence into clauses.

7.6 Compound Sentences

By default, if a sentence is compound – that is, it contains multiple clauses separated with conjunctions – then each clause will be coded, so the sentence could generate multiple events. This can be changed using the **SET: CODE BY ...** command discussed below. The first event in a clause that is recognized will be coded (with adjustments for subordinate and dominant codes). This means events are prioritized by:

- left to right order of verbs in the clause
- phrases within a verb by length

Conjunctions are `AND_`, `BUT_`, `OR_` and `NOR_`. `AND`s that are used in a compound actor, noun or adjective phrase formation or by a verb phrase that itself contains a compound (e.g. `SHOT_AND_KILLED`) are not used to delimit clauses.

When a compound sentence is encountered, the source actor is kept the same unless it is explicitly overridden by a $ operator in a pattern or if an actor occurs *immediately* after the conjunction. In coding after a conjunction, the *target* is reset using the default rules for assigning targets (e.g. the code of the target must be distinct from the code of the source).

If no event is found in the first clause, the source actor is set to the first actor in that clause, provided one exists. This usually identifies the correct subject of a sentence even when the verb in the first clause is not in the dictionary.

7.6.1 CODE BY... Command

The `SET: CODE BY ...` parameter, set in the *.options* file, determines when the system stops looking for events. There are three options:

CODE BY CLAUSE [default]
The system codes the first event it finds in each conjunction-delimited clause in the sentence. A subordinate event is promoted only if no other event is found in the clause containing the subordinate event.

CODE BY SENTENCE
The system codes only the first event it finds in the sentence. A subordinate event is promoted only if no other event is found in the sentence.

CODE ALL.
The system codes every event that it finds in the sentence. A subordinate event is promoted only if no other event is found in the sentence.

Example:

If `BOMBED, SAID` and `ATTACK` are verbs, then the sentence

```
Israel bombed guerrilla bases in southern Lebanon and said this was in
retaliation for earlier rocket attacks...
```

would be coded differently depending on the `CODE BY` condition

`CODE BY CLAUSE` would generate two events

```
ISR   bombed   LEB
ISR   said     LEB
```

`CODE BY SENTENCE` would generate one event

```
ISR   bombed   LEB
```

`CODE ALL` would generate three events

```
ISR  bombed   LEB
ISR  said     LEB
ISR  attack   LEB
```

7.7 Coreferencing Pronouns

Pronouns occur fairly frequently in news wire reports, for example in the phrase:

```
Turkey believes Iraq and Syria can cope with a decrease in vital water
they receive from the mighty Euphrates river when a major dam is filled
next month.
```

Coreferencing pronouns involves ascertaining which noun or nouns a pronoun refers to; in this example `THEY` refers to `IRAQ AND SYRIA`. Coreferencing is a very general problem in parsing.[3] The techniques used in TABARI are quite simple, but they are fairly effective when applied to Reuters and AFP leads.

In compound sentences, coreferencing is important in bringing actor references into the appropriate clause of the sentence. The pronoun `IT_` is most likely to be incorrectly coreferenced since `IT_` sometimes refers to an action, an abstract concept, or a direct object.[4]

TABARI uses the following rules to handle pronoun coreferencing:

`HE, SHE, IT` assign the first actor in the sentence

`ITS, HIS, HER` assign to the first actor prior to the pronoun

`THEY, THEM, THEIR` assign either:

- First compound actor if one exists
- If there is no compound actor, then the first actor is used. This deals with most situations when the actor is, in fact, plural. (e.g. `ISRAELI POLICE`).

In the `P)arse` display, coreferenced pronouns are with a reference to the index of the actor they refer to. In the example above, `THEY` would appear as

```
THEY_ <refernc = 2>
```

because the compound actor phrase IRAQ AND SYRIA begins at index 2.[5]

When full stories, rather than only lead sentences, are being coded, pronoun coreferencing can also occur across source texts; see the discussion of the `FORWARD` command in section 8.4 of the *.options* chapter.

7.8 Elimination of Comma-Delimited Nonrestrictive Clauses

In Reuters and AFP leads, short clauses delimited by commas and clauses between commas and the end of the sentence are usually irrelevant to coding events:

```
President Hosni Mubarak, in a grim warning underlining Egypt's deepening
economic crisis, will request emergency assistance from the International
Monetary Fund, the official UAE news agency WAM said on Thursday ..
```

[3] It is also a problem that cannot be solved on a purely syntactic basis: in the sentence
`Baker will meet with Mubarak when he goes to Geneva`
the pronoun `HE` could refer to either `BAKER` or `MUBARAK` depending on who is going to Geneva.

[4] Because an grammatically correct English sentence always requires a subject, the interpretation of IT is further complicated by its use as a placeholder in sentences that have *no* subject—"`It is raining`"—or as the object of a transitive verb—"`The ambassadors discussed it over dinner.`"

[5] Real programmers always begin indices at zero, not one.

Following the vocabulary used in Turabian (1987:51), these are called *nonrestrictive clauses*, or *comma-delimited clauses*.[6]

TABARI's default action is to eliminate these phrases if the number of words between commas is greater than two and less than or equal to ten; the minimum allows the preservation of lists with commas in them. The maximum and minimum length of an eliminated phrase can be adjusted using the `COMMA` command in the *.options* file; this feature can also be turned off. Commas inside numbers – for example 10,000 – do not trigger this feature, nor do commas inside lists of actors that were converted into parsed compounds.

7.9 Passive voice

Passive voice (e.g. "Bosnia was attacked by Serbia") detection is automatic and does not require specific patterns for each verb. The following rules are used to detect passive voice: [7]

1. Auxiliary verb WAS_, WERE_ or BEEN_ occurs one or two words prior to verb
2. BY_ immediately follows verb;
3. Source and target have not been set by $, + or % in verb patterns

When the passive voice construction is detected, the word ordering is assumed to be object-verb-subject, and therefore the source and target are reversed. Passive voice detection also forces the subject of any later verbs to be re-evaluated if multiple verbs are coded in a sentence.

7.10 Summary: Parsing

Parsing operations occur in the following order; operations in *italics* are optional:

1. Filter the text for punctuation and diacriticals
2. Assign types to all words that are in the dictionary; if a word cannot be found it is classified <null>
3. Identify compound actors
4. Identify compound noun and adjective phrases
5. Coreference pronouns

[6] Some comma-delimited clauses found in event reports are not nonrestrictive by Turabian's definition—Turabian actually distinguishes 14 different types of comma-delimited clauses—but this term will suffice for purposes of machine coding...

[7] There are persistent reports that this feature does not always work, even though it works in all of our validation tests. If you find instances where it fails, we would appreciate samples of the relevant texts and dictionaries.

6. Eliminate comma-delimited nonrestrictive clauses
7. Identify the clauses in a compound sentence
8. Check the phrases for each verb in each clause of the sentence
9. When a verb phrase is found and does not have designated actors, identify source and target:
   (a) Source is the first actor in the sentence
   (b) Target is the first actor after the verb that has a code distinct from that of the source; if this does not exist, it is the first actor before the verb

Chapter 8

.Options File

8.1 Purpose

The *.options* file provides processing information and contains a variety of specific commands. The commands in this file can have a *substantial* impact on how TABARI does coding, and a dictionary that has been developed under one set of *.options* settings may work very poorly under another set.

8.1.1 .Options Command Notes

1. As with all TABARI files, the *.options* file should be created using a text editor or saved in flat ASCII format if produced using a word processor. The file cannot be changed from within the TABARI program.

2. The command names are given here in full for mnemonic purposes, but the program usually only looks at the first four characters and the colon. For example,

   DEFAULT: TARGET [WLD]

   could be entered as

   DEFA: TARGET [WLD]

3. There is only limited error checking in most of the commands in the *.options* file. If an improper format is used, the value of the parameter will usually stay at the default but no indication of an error will be given. In addition, the command-line processing presupposes a modicum of sanity on the part of the user—in particular, if the command is setting variables using a text string, don't include that text string in a code: For example `DEFAULT: SOURCE [SCRAFTER] TARGET [THETAR]` will activate the `AFTER` option. More generally, if you try to do something clever that you think might have unexpected side-effects, it probably will.

4. The *.options* file can take input lines up to 255 characters in length. If some lines are longer than the 80-character line length used in many word processors, be sure to save these as TEXT without the line feeds so that the command is read as a single string

of characters. The commands and parameter names are *case-sensitive* but in the spirit of "do-what-I-mean," most are forced to upper case.

5. Any text following '#' or '//' is treated as a comment and ignored. These can be used to temporarily "comment-out" commands.

8.2 SET <parameter> = <value>

This is a general command that sets various switches and numerical parameters affecting the operation of the program. Logical switch can take the values **TRUE (T)** or **FALSE (F)**. 3-way switches can either be OFF—in which case they have no effect at all—or take a logical value. Numerical parameters have various formats.

List of SET switches and parameters

Parameter	Values	Default
CODE BY ...	CLAUSE ALL SENTENCE	CLAUSE
LABEL	logical	TRUE
MINIMUM WORDS	numerical	FALSE
ACTOR USAGE	3-way	OFF
AGENT USAGE	3-way	OFF
VERB USAGE	3-way	OFF
DUP WARNING	logical	TRUE
CONVERT AGENTS	logical	FALSE
FBIS	logical	FALSE
YYYY	logical	FALSE
TIME SHIFTING	logical	set by `.verbs` file
SKIP RECORDS	logical	TRUE
HAIKU	logical	TRUE
MATCH	obsolete	use OUTPUT

8.2.1 CODE BY...

See discussion in section 7.6.1.

8.2.2 SET: LABEL = FALSE

This option suppresses adding the event code label as the last field in the event string. It affects both the screen display and the file.

8.2.3 SET: MINIMUM WORDS = n

This sets the minimum length (in words) for a valid sentence. If a sentence contains fewer than n words (where n is any integer), it will not be parsed or coded; note that in this calculation, a comma is a "word." This is useful for skipping sentence fragments that got through (or were created by) your text filters. The default value is 8.

8.2.4 SET: ACTOR USAGE = OFF
SET: AGENT USAGE = OFF
SET: VERB USAGE = OFF

This allows an automatic response to the "Write dictionary usage files?" query that is presented when the program reaches the end of a file. If the command is absent or set to OFF, the decision is made via a query to the user. If it is set to TRUE or FALSE, the query is skipped and the file is saved for TRUE, not saved for FALSE.

8.2.5 SET: DUP WARNING = FALSE

By default, TABARI issues a warning when it encounters a duplicate actor or verb root. This option will turn off that warning, which can be useful if multiple overlapping dictionaries are being used temporarily. Whether or not the warning is turned on, only the first entry is entered in the dictionaries, and any duplicates will be removed when dictionaries are saved.

8.2.6 SET: CONVERT AGENTS = TRUE

If set to TRUE, agents which do not have an associated actor according to the rules in Section 5.4 are converted to actors and their agent codes (rather than actor+agent codes) are used. This is allows a sentence such as

```
Students and police fought in the Egyptian capital
```

to be coded as

```
~EDU fought ~COP
```

or the sentence

```
Sudanese tudents and police fought in the Egyptian capital
```

to be coded as

```
SUDEDU fought ~COP
```

This option is intended for system where the location of the event is either known—for example coding from local news sources where knowledge of the location is assumed rather than explicit—or can be determined by processing with other programs. Once the location is known, an auxiliary program can be used to combine that code with the agent code.

The potential risk with this approach is that some agents can be used as adjectives rather than nouns (e.g. MILITARY), and this can cause coding errors with agent dictionaries that contain generic terms.

8.2.7 SET: HAIKU = FALSE

Most of the error messages in TABARI are accompanied by commentary in the form of [bad] haiku.[1]. For example:

File will not open
It remains a mystery
Much like life itself.

If you find these annoying (or are setting up the project options for someone who will), this option will turn off the automatic display.[2]

8.2.8 SET: YYYY = FALSE

If set to TRUE, years are specified using four digits rather than two, and pivoting on the years 00 to 30 (see section 6.5.2) is suppressed.

8.2.9 SET: TIME SHIFTING = FALSE

If set to FALSE, time shifting (see Section 10.5) is disabled even if a set of `<TIME>...</TIME>` phrases were read in the *.verbs* file. This is primarily used for `V)alidation` checking (Section 10.1); by default time shifting is activated if the `<TIME>` phrases are present, deactivated otherwise. If this is deactivated at the point when a *.verbs* file is written, the `<TIME>` phrases will not be written.

---

[1] Such weirdness is this!
It is open source software
And not Microsoft

[2] **Programming Note:** The texts for all of the haiku can be found in the set of strings named `shErrorNN` in TABARI.h.

8.2.10 SET: SKIP RECORDS = FALSE

If set to TRUE, the query

```
Skip the NN records coded in previous sessions ->
```

is displayed after the dictionaries are read; NN is the number of records already coded in the file as determined from the final `<session>` record in the *.project* file. This is typically done when manually checking a file. If the file is going to be re-coded from the beginning—for example during debugging or validation—this can be set to `FALSE`. This feature is also turned off when the autocoding command-line option $-a$ is used.

8.2.11 SET: FBIS = TRUE

This invokes some custom processing that handles leads downloaded from the Foreign Broadcast Information Service "Open Source Portal" and formatted using the KEDS project FBIS.REFORMAT.PL program. It does the following:

1. Material inside [brackets] is skipped
2. Non-alphabetic material (numbers, —, **, etc) at beginning of sentence is skipped
3. Commas in the construction `<actor><comma><actor>` (as in "`Afghan, Tajek rebels`") are treated as conjunctions. This does not apply in a comma-delimited list (e.g. ''`Lions, tigers, and bears, oh my`").

Note: When processing FBIS, it useful to set `MINIMUM WORDS` to a smaller value than the default because many valid FBIS reports are far shorter than most Reuters and AFP reports.

8.2.12 SET: MATCH = TRUE

As of version 0.8, this is no longer available as a `SET:` option: use OUTPUT: MATCH instead (Section 8.9).

8.3 LABEL <code string> = <text>

This command associates a text string with a code. By default, the text for event codes will appear in the output strings displayed in the event window of the program and in the data. Providing some text to describe each event code is strongly recommended when doing dictionary development, unless you have coders with *very* good memories. Text can be assigned to either actor or event codes.

The text string may be any length (within the general 255 character limit for any string read by TABARI), but generally it is a short description in order to fit within the width of the window. Leading and trailing blanks of the code and the text will be skipped; blanks within the label will be retained.

Example:

LABEL: 04=APPROVE
LABEL: 041=PRAISE
LABEL: 042=ENDORSE
LABEL: 05=PROMISE
LABEL: 051=PROMISE POLICY SUPPORT
LABEL: ISR=Israel
LABEL: PSE=Palestinians
LABEL: JOR=Jordan

8.4 FORWARD: sequence_format

NOTE 2012.02.08: We are in the process of phasing out this feature since there are now open-source pronoun coreferencing systems that are far more accurate than the simple rules used here. In the relatively near future, TABARI will be modified to allow some form of parsed input to handle the cross-sentence references, and those systems will also improve on the within sentence references as well. At this time, the basic system works and should not cause any program crashes (though it has not been tested extensively). In addition, combining the record documentation and the `FORWARD/OUTPUT` systems was awkward, and will not be repeated. Still, it works.

The `FORWARD` command activates the forwarding of pronoun coreferences between text records: this is used when coding full stories. When `FORWARD` is active, when a pronoun occurs at the beginning of a sentence, prior to any actors, it is coreferenced as the first actor in the *previous* sentence, where "previous" is determined by the sequence format given in the command, provided that the actor in the previous sentence occurred before a verb.

The plural pronouns THEY_, THEM_ and THEIR_ are only replaced if the forwarded actor is compound. This rule, while somewhat conservative, is quite effective at handling pronoun coreferences across sentences in Reuters and AFP.

The sequencing of sentences is given by the *idinfo* string in the date header. The *sequence_format* string in the command (discussed below) shows where the sequencing information is found. Forwarding is done if the following conditions hold:

- The story number of the current text is *equal* to the story number of the previous source text
- The sentence number of the current text is equal to the sentence number of the previous source text *plus one*

Note that the second condition allows your formatting program to specify situations where forwarding should not be done (e.g. across paragraphs) by increasing the sentence number more than 1.

The record identification can be followed by an additional document identification of up to 255 characters—the extended length is intended to accommodate long URLS, a typical use for this feature—which is copied from the first non-blank character following the record identification to the end of the line. This is turned on with the `DOC` keyword.

Example:

FORWARD: ^^^***NNNN*SSS DOC

Unlike the record ID, the document ID does not have to be a constant length for each record. Note that if FORWARD is not being used, the record ID can just be used as the document ID.

The keyword `OFF` disables forwarding even when a story ID and sentence serial number has been specified. This allows the identification information to be written (see `OUTPUT`) without using forwarding.

Example:

FORWARD: ^^^***NNNN*SSS OFF

Pronouns can be forwarded across multiple stories provided a chains of coreferences is maintained. For example, in the sequence

```
810726 REUT-045-001
James Baker begins a Middle East shuttle trip...
810726 REUT-045-002
He will first fly to Israel to...
810726 REUT-045-003
Afterwards, he plans to visit Jordan to...
```

The "he" in the third sentence (45-003) will still refer to Baker.

8.4.1 Formatting the Sequence Numbers

The *sequence_format* string is composed of the following characters:

^ = leading blanks between the date and the start of the idinfo string

***** = any character (including blanks after the start of the string)

N = serial number of the story

S = serial number of the sentence within the story

For example, if your text formatter had generated story id's consisting of a 3-character source identifier, a 3-digit story identifier and a 3-digit sentence identifier:

890227REU312023

the serial format would be

***NNNSSS

If this format included some blanks and hyphens for readability

890227◇◇REU-312-023

(where ◇=blank) the serial format would be

^^****NNN*SSS

When the FORWARD command is used, the *idinfo* string is copied for the first non-blank (i.e. *, N or S) character to the end of the string, so in the first example the *idinfo* string would be nine (9) characters in length; in the second example it would be eleven (11) characters in length. The maximum length of an *idinfo* string—not including leading blanks—is 15 characters.

The story and sentence serial numbers must consist of a sub-string of consecutive N's and S's (i.e. ***NNSSNN is not allowed; the second set of NN will be ignored) but can be in any order (i.e. SSS***NNN is allowed). If a non-numeric value is found in either sequence number, the entire number is treated as equal to zero.

The *idinfo* string length is 31 characters. The story ID segment (i.e. the SSS...SS part) does not need to be numeric. The sentence serial number (NN...NN) must be numeric (i.e. integer). Sentence numbers must be less than or equal to 32,767.

8.5 DEFAULT: SOURCE [code] <PRIOR> TARGET [code] <AFTER>

The **DEFAULT** command provides a default code for the source and/or target. When activated, these codes are used if the regular searches for actors has failed. If it is not activated, a text with a missing source or target will not generate an event.

Example:

DEFAULT: SOURCE: [USA] PRIOR TARGET: [WORLD] AFTER

DEFAULT: SOURCE: [SRCDEF] TARGET: [TARDEF]

DEFAULT assigns a default source and/or target code even when no identifiable actor or agent can be found associated with the verb. The `PRIOR` option after `SOURCE` means that the default will be applied any time there is no actor prior to the verb;[3] The `AFTER` option following `TARGET` means that the default will be applied any time there is no actor following the verb.

Default codes are useful in one or more of the following circumstances:

- Dictionaries are incomplete and you want to find situations where an codeable event has been reported but the actors involved are not in the dictionary;
- The default target is useful for events such as comments and policy statements that are not directed to a specific audience;
- It will sometimes allow the coding of abbreviated, syntactically incomplete sentences—for example chronologies or direct quotes—where the subject or object is implied rather than explicit.[4]
- It allows TABARI to code only verb phrases without identifying (or requiring) sources and targets.

Default codes that refer to specific actors—as opposed to special codes that are only used in the context of a default—will probably substantially increase the number of incorrectly coded events; it is generally a good idea to make the default codes distinct (e.g. `SRCDEF, TARDEF`) so that you know they have been assigned by the default procedure rather than through regular coding.

If the `OUTPUT: LOC` option is used—see section 8.9—default actors are given a "location" of 127.

As noted earlier, assigning a default source will guarantee that the first verb identified in each clause (or the sentence) is coded. This means that the coding is more sensitive to disambiguation errors: the first "verb" may actually be a noun or an adjective.

The `COMPLEX NOACTPRIOR NOACTAFTER` options (see Section 8.7) should not be used in conjunction with default codes, since these will cause the sentence to be classified as complex and the coding will be bypassed rather than using the default codes. The is some error-checking on the **DEFAULT** command—this assumes that the default codes 15 or fewer characters in length, and in also shifts the codes to upper case.

`DEFAULT: OFF` deactivates all of the `DEFAULT` options; this is only used in the `V)alidation` texts (Section 10.1) since the *.options* file is only read at the beginning of the run.

8.6 COMMA

This command sets the minimum and maximum length of comma-delimited nonrestrictive clauses that are eliminated before coding. The syntax is

[3] You usually want to use this option: in regular coding, TABARI is "greedy" in locating actors, and if a source is not found prior to the verb, it will look for it after the verb, and—less frequently—a target can be located prior to the verb. Consequently you might get a default target when in fact the source is missing

[4] See the discussion in Gerner, et al. 1994.

```
COMMA: MIN=<minimum value> MAX=<maximum value
EMIN=<minimum value EMAX=<maximum value
BMIN=<minimum value BMAX=<maximum value
```

where the values are the number of words between commas, or between a comma and the beginning or end of a sentence. The comma locations are "chained"—when there are multiple commas in a sentence, counting for the length of the next comma-delimited phrase starts at the comma which ended the previous phrase. Commas inside syntactic compound phrases are not evaluated.

The BMIN/BMAX limits are used only on phrases at the beginning of the sentence and the EMIN/EMAX limits are used only on phrases at the end of the sentence.

The defaults are

BMIN =	128	BMAX =	0
MIN =	2	MAX =	10
EMIN =	2	EMAX =	10

The default BMIN/BMAX settings cause the first comma-delimited phrase to *never* be deleted.

This feature can be turned off for all phrases using the command

```
COMMA: OFF
```

When this command is used, all of the text—including the material in comma-delimited clauses—is checked for events. This will usually substantially change the coding. To code from the nonrestrictive clauses only when no events were found in the regular text, use the `SET: CODE NONRESTRICTIVE=TRUE` command.

Example:

COMMA: MIN = 1 MAX = 8 EMIN=3 EMAX=10
COMMA: OFF

NOTE 2012.02.14: This system works "okay" with the defaults, but these are just rules of thumb and a more sophisticated parser will almost certainly produce better results, particularly on complex sentences. A system for incorporating this information will be added in the near future. We have also done almost no experimentation with activating the initial clauses, and our sense is that this may result in incorrect (or no) identification of the subject quite a few sentences due to structures such as "Arnor is about to restore complete, full diplomatic ties with Gondor..."

8.7 COMPLEX

The **COMPLEX** command provides a complexity filter: it defines the conditions under which a sentence is considered too complex to code. By default all options are inactive.[5] When a complex text is found, the program takes the following actions:

In autocoding mode: By default, the text is diverted to a *.complex* file that can be later examined. If the NOSAVE option is used, the record is skipped and no file is written.

In normal mode: No events are coded and the message **Complex text** is displayed.

The COMPLEX command has the following options:

Parameter	**Condition**
VERBS[n]	Diverted if text contains *n* or more verbs
ACTORS[n]	Diverted if text contains *n* or more actors
PRONOUNS[n]	Diverted if text contains *n* or more pronouns
CONJ[n]	Diverted if text contains *n* or more conjunctions
COMMA[n]	Diverted if text contains *n* or more comma-delimited nonrestrictive clauses
LATEVERB[n]	Diverted if no verb is found within *n* words of the beginning of the sentence. If **LATEVERB** is active, a text will also be diverted if it contains no verb.
NOACTPRIOR	Diverted if no actor occurs *prior* to the first verb
NOACTAFTER	Diverted if no actor occurs *after* the first verb
NOVERB	Diverted if there is no verb
NOSOURCE	Diverted if there is no source
NOTARGET	Diverted if there is no target
NOEVENT	Diverted if there is no event
EXPLAIN	Inserts an explanation of why the source was diverted into the output file inside /*...*/ delimiters; the program also displays this explanation on the screen after the **Complex text** message when not autocoding
NOSAVE	Do not save the complex files to a *.complex* file when autocoding.

The text is diverted if any of the conditions listed in the **COMPLEX** command are met. The first six commands require a number in the brackets; there are no default values, and a value of zero indicates that the condition is not active. The last six are logical; if they are present the condition is checked; otherwise it isn't. As noted above, text will also be diverted to the *.complex* file (or skipped if the NOSAVE option is used) if it contains a phrase that has a complex code +++

Example:

[5] Implicitly, the complex (+++) and discard (###) codes also provide some complexity filtering: these are always active.

COMPLEX: VERBS[5] ACTORS[8] LATEVERB[8] NOVERB NOACTPRIOR EXPLAIN

Almost all of the coding we have done uses the following complexity conditions:

COMPLEX: VERBS[6] NOACTPRIOR EXPLAIN

This has worked quite well on both Reuters and AFP texts.

Note: Null-coded actors and verbs that are null-coded and have no pattern are converted to type NULL and are not considered in the evaluation of complexity.

8.8 NONEVENTS

When TABARI tries to code a record and does not find a verb phrase that generates a non-null code, usually it does nothing with that record. This can be over-ridden using the `NONEVENTS` command; the treatment of nonevents is described in detail in section 10.3. The command can take three values:

`NONEVENTS: TRUE`	Sentences that do not contain a verb phrase that generates a non-null code are included in the event file with the event code ***;regular events also written to the file. Coding of actors in nonevents is described in section 10.3.
`NONEVENTS: ONLY`	Only nonevent records are written to the event files; sentences generating regular events are ignored.
`NONEVENTS: FALSE`	Sentences that do not contain a verb phrase that generates a non-null code are ignored [default]

If `NONEVENT` is set to either `TRUE` or `ONLY`, it automatically cancels any `SET: MATCH = TRUE` command, since the `MATCH` facility requires a verb phrase to anchor the construction of the match (see section 8.2.12).

Note: to get the opposite effect—situations where there is a verb phrase but the source and/or target is missing—use the default `SOURCE:` and/or `TARGET:` commands discussed in Section 8.5.

8.9 OUTPUT

The default event output format is

```
<date> \t <source code> \t <target code> \t <event> \t <label>
```

where `\t` is the tab character. If the option `SET: LABEL = FALSE` is used, the last field is suppressed.

Additional fields can be added using the OUTPUT command

Keyword	Content
ACTORS	Text used to match the source and target actors
AGENTS	Text used to match the source and target agents
PARENT	Output the *first* phrase in a list of actor synonyms
SHIFT	Adds SHIFT and number of days if the date was time-shifted; otherwise empty field
MATCH	This adds the text of the matched string to the event record: see notes below
LOCS	Three tab-delimited fields giving the word location for the source, target and verb: see notes below
PATTERN	Patterns used to match the source, target, and verb phrase
ID	Entire record ID string, without leading blanks
TEXT	Text of the sentence

Notes on Additional fields

1. Actors and agents phrases occur in tab-delimited fields between the LABEL and MATCH phrases in the order

```
source-actor \t source-agent \t target-actor \t target-agent \t
```

2. PARENT can be used to disambiguate multiple forms of an individual actor's name—for example, JIM_SMITH, JAMES_''JIMMY-THE-RAT''_SMITH, and SENATOR_JAMES_HARBRIG_WALLACE_SMITH_IV—by outputting a single phrase for all forms of the name. This is the first phrase in a list of actor synonyms (see Section 5.3.1).

3.The PATTERNs will be written in three tab-delimited fields immediately following the standard fields. If a verb pattern—as distinct from a simple verb root—is used, the verb is inside <...>, e.g.

```
<TRADE > ACCUSATIONS
```

4. The TEXT field is always the last one in the output record.

5. The MATCH field still has a few quirks: Multi-word actors located with pattern tokens (+, $, %) prior to the verb (*) will be in reverse order. But you'll get the idea.... In addition, MATCH is automatically deactivated, even if it is included in a OUTPUT command, when the NONEVENT command is set to either TRUE or ONLY (see section 8.8).

6. The numbers produced by LOCS give the index of the word where the phrase begins. The first word is 0, second word is 1, etc.[6] Commas and most other punctuation are "words" in this scheme. The third field gives the location of the beginning of the *verb*—that is, the word which is anchored by the * in the verb pattern—rather than the start of the verb phrase. This information can be used for post-processing of the texts, for example in trying to locate relevant geographical information. If the DEFAULT option is used—see section 8.5—default actors are given a "location" of 127.

If the FORWARD: command has been used, one can optionally output three other identification fields:

[6] Zero-based indexing used in most programming languages. If you prefer 1-based indexing, this can be changed in the final lines of the routine CoderClass::getEventString(). Though if you are comfortable changing that code, you probably prefer zero-based indexing anyway. Unless you program in FORTRAN.

Keyword	Content
STORY	Story serial number
SENTENCE	Sentence serial number
DOCUMENT	Document identification

These will be written in tab-delimited fields following the `LABEL` or `PATTERNS` field in the order given above. `TEXT, STORY, SENTENCE` and `DOCUMENT` can be abbreviated to three characters. Note that if you only want to output these fields, without pronoun forwarding, this can be done by specifying `FORWARD: ... OFF`—see section 8.4.

Example:

```
OUTPUT: ID SENTENCE
OUTPUT: DOC
OUTPUT: PATTERN ID
OUTPUT: ID TEXT
```

Programming Note: The format on an output string for `LABEL, ACTORS, AGENTS, MATCH` and `PATTERN` is set in the function `CoderClass::getEventString()` and can be readily modified there. The order of the `ID, STORY, SENTENCE` and `DOCUMENT` fields can be changed in the function `ProcessorClass::writeEvents`.

Chapter 9

Content Analysis: ISSUES and FREQUENCIES

9.1 Introduction

TABARI is primarily a program for coding events using the subject-verb-object structure of a sentence. It is *not* intended as a general-purpose content analysis program, and there are far better programs available for that purpose.[1]. However, on occasion it is useful to be able to do some basic content analysis on the texts being coded: the `<ISSUES>` and `<FREQUENCIES>` facilities allow this.

`<ISSUES>` are typically used to used to determine the context of an event; it searches for phrases rather than individual words. `<ISSUES>` can also be used to extract numerical information, such as the number of demonstrators or number of people killed. The `<ISSUES>` codes are appended to the regular output; depending on the text, several issues may be found (e.g. the typical modern urban demonstration in an industrialized society covers every issue from nuclear disarmament to endangered toads), or none.

`<FREQUENCIES>` simply counts individual words and writes these numbers to a file in a rectangular data matrix containing the number of words found in each category. This is typically used as input in a further statistical analysis. Unlike `<ISSUES>`, the number of data fields is the same for each record—if no words are found, the field contains a zero—and the fields are integers rather than codes.

Both of these facilities have been validated on a simulated set of test cases, and used in a couple of large-scale experiments. However, we have not employed them extensively, and you should be alert for the possibility of bugs.

[1] Athough none of these programs, as of February 2006, are open-source: someone needs to take care of that...

9.2 ISSUES

The TABARI `<ISSUES>` facility allows additional content analysis to be done on the sentence; this allows some of the "context" of the event to be determined. For example, the general-purpose code for "`consult`" might also code for the content of the consultation by looking for strings such as "`arms control`", "`ceasefire`", "`trade`" and so forth.

Issues are done using simple string-matching without regard for syntax. It uses the same rules as are used for actor and verb phrases—in particular, a multi-word phrase is matched to the longest phrase that starts with the first word. In the current version of the program, a phrase used as an issue cannot also be used as an actor or a verb root, although it can appear in a verb *pattern*. The *issues* codes are appended in a tab-delimted list to each event generated from a record; if no issue in a category is found, this is denoted by two consecutive tabs unless a textbfDEFAULT code is specified.

Issues coding is specified in an *issues* file that is designed in the project file using the command

```
<issuesfile> name of file
```

The issues file is divided by XML tags `<ISSUE...>...</ISSUE>`; these must appear on individual lines. The following fields are allowed in the tag:

CATEGORY = "string"

This is used to identify the category. At the present time, nothing is done with this string; eventually there will be some way to use it in a display. Please do not include the keywords `TYPE` and `DEFAULT` in this; at our XML parser is very simple.

TYPE = "ALL"

The codes for all of the issues found in the string will be included in the output; the default is that only the code for the first issue encountered in the sentence in each category will be in the output. Multiple codes are separated by spaces.

TYPE = "NUMERIC"

This looks for the first phrase of the form `<number><issue>` (for example "`500 demonstrators`") and puts the numerical value in the output file. At present, the number must be an integer.

A specific number can also be associated with a phrase, e.g.

```
HUNDREDS_OF_PROTESTORS [367]
```

Any code following a numerical issue will be interpreted as an integer.[2] If you are only expecting to look for patterns of the form `<number><issue>`, the numerical value is optional.

Order of precedence in evaluation of numerical issues:

[2] **Programming Note:** Specifically, the C function atoi() is used to interpret the string—check your friendly local C manual for details on this.

1. Look for a number immediately prior to the issue; if present, interpret it as an integer;
2. If no number is found before the issue, add the coded numerical value (which can be zero)

Notes:

1. If a number was entered as part of a phrase—for example "Group of 77"—it is interpreted as a character string ("literal") rather than as a number, and therefore will not be picked up in a numeric issue. If this is a problem, globally recode the number in the phrase—e.g. replace "Group of 77" with "Group of N77"—so that it is distinct from the number.
2. TABARI currently interprets any string that has a digit as the first character as a number, so for example "23Skidoo" would be a number, though atoi() would not know how to interpret this.[3]

TYPE = "ALL NUMERIC"

This looks for all phrases of the form `<number><issue>` (for example "`500 demonstrators`") and puts the *total* of the numerical values in the output file.

DEFAULT = "string"

Default code for the category. If no default code is specified, the field will be empty if no issue is found. A non-numeric code can be specified even if the issue is numeric; this allows a missing-value

Example

```
<ISSUE CATEGORY = "1">
ISSUE1A [I1AA]
ISSUE_ISSUE1B [I1BB]
ISSUE ISSUE1C [I1CC]
ISSUE1D [I1DD]
</ISSUE>
<ISSUE CATEGORY = "2" TYPE = "ALL">
ISSUE2A [I2AA]
ISSUE_ISSUE2B [I2BB]
ISSUE ISSUE2C [I2CC]
ISSUE2D [I2DD]
</ISSUE>
<ISSUE CATEGORY = "3">
ISSUE3A [I3AA]
ISSUE_ISSUE3B [I3BB]
ISSUE ISSUE3C [I3CC]
ISSUE1D [I3DD]
</ISSUE>
```

[3] **Programming Note:** Code for a more rigorous assessment of numbers can be added in the routine `ParserClass::makeLex(void)`.

9.3 FREQUENCY

The TABARI <FREQUENCY> facility is a second mode for adding content analysis which counts the occurrence of specific classes of words in the sentences. Frequencies are simpler than issues: they involve single words and do not have codes. The facility was designed to be used with word sets such as those associated with the GENERAL INQUIRER program. Frequencies are tabulated at the literal level without any pattern matching and involve almost no additional computational overhead. There is no limit (other than available memory) on the number of words coded.

A literal can appear in multiple frequency lists, and frequency counts for all categories (including those that are zero) are appended in a tab-delimted list to each event generated from a record. These will typically be used as input to a statistical program.

Frequency coding is specified in a frequencies file in the project file using the command

```
<freqfile> name of file
```

The *.freqs* file is divided by XML tags <FREQ ...>...</FREQ>; these must appear on individual lines. The following fields are allowed in the tag

CATEGORY = "string"

This is used to identify the category. If a documentation header is being written to the event file, these strings are written in a tab-limited line labeled FREQUENCIES: ("- -" is written if the CATEGORY string is absent for a particular category). Please do not include the keywords TYPE and DEFAULT in this; at the moment our XML parser is still very simple.

ABBREV = "string"

This string appears in the screen output above the list of issues. The display is formatted on the assumption that ABBREV strings are four characters long. If a documentation header is being written to the event file, these strings are also written in a tab-limited line labeled FREQS: ("- -" is written if the string is absent for a particular category).

Example:

```
<FREQ CATEGORY = "Action" ABBREV = "ACTN">
ABIDE [ACTV]
ABOLISH [ACTV]
ABSCOND [ACTV]
ABSOLVE [ACTV]
ABUSE [ACTV]
ACCEDE [ACTV]
ACCELERATE [ACTV]
ACCELERATION [ACTV]
ACCENTUATE [ACTV]
ACCOMMODATE [ACTV]
```

```
ACCOMPANY [ACTV]
ACCOMPLISH [ACTV]
ACCOMPLISHMENT [ACTV]
</FREQ>
<FREQ CATEGORY = "Hostile" ABBREV = "HOST">
ABHOR
ABOLISH
ABRASIVE
ABSCOND [HSTL]
ABSENTEE [HSTL]
ABUSE [HSTL]
ACCOST [HSTL]
ACCURSED [HSTL]
ACCUSATION [HSTL]
ACCUSE [HSTL]
ACRIMONIOUS [HSTL]
ACRIMONY [HSTL]
ADMONISH [HSTL]
ADVERSARY [HSTL]
</FREQ>
```

Chapter 10

Special Features

10.1 Validation

The `V)alidation` facility goes through a set of pre-coded records to confirm that TABARI is [still] working correctly after changes have been made in the source code. The Validate.project file invokes a project containing about 500 such records; this project can be loaded by entering the full name or using the shortcut ∼v when prompted for a project file name (provided that the required files are available). The `V)alidation` file ends with a small number of records which cause the program to terminate, so successful validation ends at the record labeled `FINAL-01` rather than the end of the file; use `Q)uit` at this point. This facility has been used extensively during the development of TABARI —more generally, the validation suite should always be run after any changes to the code—and should be thoroughly debugged.

Validation is invoked by typing `V` in the `O)ther` menu. This option is not shown in the menu since it would not be employed by most users. Validation operates similar to autocoding—it will go through any correctly coded record without pausing, then stops when it encounters a record where the events listed in the record and those produced by the coding do not match. Events from validation are not saved.

The validation records are placed in brackets {...} in a comment section—delimited by /* . . . */) —following the text. Each event should have a separate, bracketed record. The correct codes are entered in the order

```
{ <source> / <target> / <event> }
```

separated by slashes. The codes are blank-trimmed before matching, so blanks can be added for clarity. Text outside of the brackets is ignored unless time-shifting is active, in which case the first record should contain a date. If an `<ATTRIBUTION>` list is being used, the attribution code is also checked. At present, `<ISSUE>` codes are not checked.

Example:

```
950103 DEMO-04
Dagolath's first Deputy Prime Minister Telemar left for
Minas Tirith for meetings of the joint transport committee
with Arnor, the Dagolathi news agency reported.
/*\n-----------------------------------------------------
\n[Paired events: LEFT_ generates a "visit" and "receive
\n visit" events]
\n{ DAG / GON / 032 }
\n{ GON / DAG / 033 } */
```

As in a comment section, the "\n" at the beginning of a line [awkwardly] generates a line break; this has no effect on the validation but makes the record easier to read on the screen.

The following special codes can be used in validation:

{- - -}	No events were generated; this also catches some but not all of the input and parsing errors.
{###}	Discard code was found
{+++}	Complex code was found or complexity conditions exceeded
-{...	Do not check additional records—this can be used to comment-out the validation of some records, or to avoid having to list all of the records found when compound actors are present

These special codes are checked first, and if there is a match, any source/target/event records are not checked.[1]

If time-shifting is activated, the first record should contain a date of the form `YYMMDD` following the "}" in the first record; this will be checked against the shifted date. If there is a discrepancy or if the date is missing, the validation records will not match.

10.1.1 Modifying *.options* commands

Validation records of the form

{OPTION: <*.options* command line>}

can be used to change settings that would normally be set at the beginning of the run using the *.options* file. Note that an <`option`> `OFF` command may need to be used to restore the original options before the new values are set.[2]

Example:

```
950101 DEFAULT-CMD-01
```

[1] Bug or feature?: you decide...

[2] This *only* works during validation: options cannot be reset in the text itself, though it would be simple to add this if for some reason you really need it.

```
Initiating a change in the .options setting:
Switch DEFAULT: turn off PRIOR. DEFAULT : OFF is needed first to reset this
/*\n-----------------------------------------------------------------
\n{ OPTION:  COMPLEX: VERBS[5] CONJ[4] EXPLAIN  }
\n{ OPTION: DEFAULT: OFF }
\n{ OPTION: DEFAULT: SOURCE [DEFSRC]  TARGET [DEFTAR] AFTER  } */
```

The validation process stops and returns to the regular menu when it encounters a record without any validation fields.

10.2 B)ackup

This is an O)ther menu option that saves the current *.actors* and *.verbs* dictionaries with the suffix *.back*. Unlike the file update routine that is used when the program is terminated normally, this does not rename the original files. (It *does* over-write any previously saved file with the *.back* suffix.)

The actions of this command are rather complex, and once TABARI got through its initial shake-down period, we find it isn't used very much. Compared to versions of KEDS prior to 0.8, TABARI also does not seem [as?] prone to spontaneous crashes, but it is still not immune to power failures. If you are making extensive changes, it would be a good idea to periodically backup the files.[3]

The implication of saving a file to *.back* rather than to the original name is that you need to manually rename the *.back* file if you want to use it later as the regular input file. When `B)ackup` is used and the program terminates normally, you will have three copies of your file. For example, suppose you were using a project with

```
<actorsfile> Enedwaith.actors
<verbsfile> Enedwaith.verbs
```

At the end of a coding session where the program terminated normally, you would have the following files

Enedwaith.actors	Files with the changes made during the coding session
Enedwaith.verbs	
Enedwaith.actors.back	Files with the changes that had been made during the coding
Enedwaith.verbs.back	session at the point B)ackup was last used
Enedwaith.actors.old	Files that were read at the beginning of the coding session
Enedwaith.verbs.old	

In contrast, if the coding session terminated abnormally due to a power failure, or an as-yet-undetected fatal error in TABARI , you would have the following files

[3] Or, as tends to be more commonly done, just quit the program and let the program do its normal saving of the files on exiting. Start-up in TABARI takes only a couple of seconds, whereas in KEDS it could take minutes—the logo for the program was a tapping tennis shoe—so coders tended to avoid re-starting KEDS.

Enedwaith.actors Enedwaith.verbs	Files that were read at the beginning of the coding session
Enedwaith.actors.back Enedwaith.verbs.back	Files with the changes that had been made during the coding session at the point B)ackup was last used
Enedwaith.actors.old Enedwaith.verbs.old	Files that were read at the beginning of the previous coding session

In this circumstance, you would probably want to rename the file Enedwaith.actors.back as Enedwaith.actors and Enedwaith.verbs.back as Enedwaith.verbs. However, prior to doing this, it is highly advisable to make sure that the *.back* files look okay, particularly if TABARI has crashed, because there is a slight possibility that the program crashed because the dictionaries were corrupted and the "backed-up" files contain garbage. This is the reason that `B)ackup` saves files with the *.back* suffix rather than overwriting the primary files.

Note also that *.back* files are not necessarily the most recent backup—they may be older than the *.old* files. This is likely to occur if some coders are using `B)ackup` and others are not. Look at the file modification dates to find the most recent version.[4]

10.3 Coding Nonevents

TABARI provides for the ability to "code" sentences that contain multiple actors but do not contain a verb phrase that generates a non-null event. This is typically combined with the `OUTPUT: TEXT` command in the *.options* file in order to produce a file of sentences that might contain relevant verb phrases that are not in the current dictionary.

Nonevent coding is activated using the `NONEVENTS: TRUE` or `NONEVENTS: ONLY` commands in the *.options* file. `NONEVENTS: TRUE` includes coded events along with the nonevents; `NONEVENTS: ONLY` produces a file that contains only the nonevents, which is the more typical use of this facility.

Nonevents are coded as follows:

1. The source is the first actor in the sentence. If that actor is compound, only the first actor in the compound is used;
2. The target is the next actor that follows the source (and is outside of a compound phrase containing the source) that has a code distinct from the source;
3. The event code is "***"

If a target that is distinct from the source is not found, the record is skipped.

If `NONEVENT` is set to either `TRUE` or `ONLY`, it automatically cancels any `SET: MATCH = TRUE` command, since the `MATCH` facility requires a verb phrase to anchor the construction of the match (see section 8.2.12).

[4]You may also want to look at the files themselves: At the moment OS-X is not always reliable at updating the file modification times it shows in the Finder windows, and at the very least you should click on the file name to make sure that the modification date is current. Presumably Apple will correct this at some point.

10.4 ATTRIBUTION

TABARI provides for a separate "attribution" field that indicates who made a statement. This is included as a third actor field in the output record. The attribution patterns are in the verbs dictionary. The set begins with a line containing `<ATTRIB>` and ends with a line containing `</ATTRIB>`. The roots and patterns are specified the same way as verbs. Attribution patterns generally will not contain a code segment. However, the code segment can contain a discard or complexity code—these have the same effect in attribution patterns as they do in verbs.

Attribution can be set in either a verb or attribution pattern using the symbol `@`—this works in a fashion similar to the `$` and + symbols for source and target.

The following rules are used to determine default attribution:

1. The last attribution verb is checked. This is the reverse of the rule used in evaluating verbs but is most likely to correctly identify the source, since in news reports this tends to occur in a subordinate clause at the end of the report.
2. Subordinate clauses are searched for attribution
3. The following rules are used to locate the source of the attribution

    a. `@` token in an attribution or verb pattern
    b. actor prior to attribution verb and before a comma
    c. first actor in the sentence

Note that the attribution actor will frequently also be either the source or target of the event.

The attribution code is output as the fourth tab-delimited field in the event record (following the date, source and target); if no attribution phrase was found, this is set to the blank code `***`. [5]

Limitations:

1. The root in an attribution phrase cannot also be a verb. Since `SAID_` has the largest number of patterns of any verb in most dictionaries, and is the most common attribution verb, this will require to a very substantial reorganization of that part of the *.verbs* dictionary. An attribution verb such as `SAID_` can still be used as part of a pattern for another verb; for example to code the phrase "`said would break relations`"as an event rather than attribute, use a pattern of the form

    ```
    BREAK
    - SAID WOULD * RELATIONS [171]
    ```

2. At present, the attribution patterns cannot be changed from inside the program; this part of the *.verbs* file should be modified using a text editor.

[5] **Programming note:** This format can be changed in the procedure `CoderClass:: getEventString(char *s, int ievt)`.

3. If the @ symbol is used in a verb pattern but there is no <ATTRIB>... </ATTRIB> segment in the verbs file, the @ will be assigned an actor (assuming one is found) but the code of the attribution actor will not be written to the event record. This can be used to skip over an actor (or multiple actors, by using multiple @s) in a phrase before assigning the source or target, although the standard way of doing this is using the ^operator.

4. At present, attribution does not duplicate records for compound actors, since these situations are very rare (that is, it is very uncommon for a statement to be made by a compound actor).[6] In the case of a syntactic compound, the first actor encountered before in the attribution phrase is used if the compound prior to the attribution phrase (and prior to a comma in the case of a terminal); the first actor in the sentence is used if the attribution is not found by looking before the attribution phrase. In the case of a coded compound, the first code is used.

5. The <ATTRIBUTION> facility was developed at the request of an external sponsor, and it has not been tested in our project beyond basic validation of the specified features. It may still have some bugs.

10.5 TIME SHIFTING

Time shifting patterns allow the date to be changed when words such as "yesterday", "tomorrow", "next week" and the like are encountered. These only change the date; they do not affect the coding.

The time-shift patterns are in the *.verbs* dictionary. The set begins with a line containing <TIME> and ends with a line containing </TIME>. The roots and patterns are specified the same way as verbs. The code segment generally contains the number of days to increment or decrement the date. However, the code segment can also contain a null, discard, or complexity code—these have the same effect in time-shift patterns as they do in verbs.

Limitations:

1. The number of days to be incremented or decremented must be less than or equal to 8192. This is more than 22 years, and generally using any change this large is ill-advised. If a value greater than this is specified, it will be converted to zero; a warning message will be given.

2. The *root* in a time-shift phrase cannot also be a verb, actor or agent. Consequently, these patterns should focus on the temporal modifiers, with any accompanying verbs or nouns being included in patterns.

3. The first non-null time-shift phrase found in the sentence is used; subsequent phrases are ignored.

4. Time-shifting patterns cannot be changed from inside the program; this part of the *.verbs* file should be modified using a text editor.

[6] More specifically, because this is so unusual, it has been accommodated by making minimal changes to the rest of the program, with speed and maintainance being a higher priority. If compound attribution is of major concern, you'll probably want to modify the source code.

5. The time shifting facility was developed at the request of an external sponsor, and it has not been tested in our project beyond basic validation of the specified features. It may still have some bugs.

Example:

```
<TIME>
forever_and_a_day [- - -]
kan_wa_ma_kan [###]
last_week [-7]
last_month [-30]
next_week [+7]
next_month [+30]
once_upon_a_time [###]
tomorrow [+1]
- * and tomorrow and tomorrow [+3]
- * and a week [+8]
when_peace_comes [+++]
yesterday [-1]
</TIME>
```

NOTE: In versions prior to 0.8, the *.events* file was re-written when the program exited so that the time-shifted records in the correct chronological order, and prior to re-writing, time-shifted events were identified by a '`*`' before the date. In the current version, time-shifted events are simply included at the point they were encountered in the input file and it is the responsibility of the user to get the file into chronological order: this can be done using any sorting utility, with the additional step of moving the records with dates after 1-Jan-2000 to follow 31-Dec-1999 if only two-digit years are used. For example, in OS-X 10.6, the command

```
sort -k 1,1 -t \t -o sort.output REULE.201111.evt.txt
```

will sort the file *REULE.201111.evt.txt* on the basis of the date field, and put the result in *sort.output*.[7] This change was made due to the need to accommodate extremely large input files, which were usually split for parallel processing and recombined anyway. If you need to know whether a date has been shifted, use the `OUTPUT: SHIFT` option. (Section 8.9).

[7] Based on the `man` page, it would seem that a simpler `sort -n...` should be sufficient, but that wasn't working for me, and this version did. Computers...

Appendix A

Utility Programs

In addition to TABARI , our project uses several utility programs for the pre-processing of source texts and the post-processing of event data. All of these programs are available at the web site.

A.1 NEXIS_Filter

Over the years we have developed a series of different filter programs that convert downloads from NEXIS, FBIS and Factiva into the appropriate format for processing by TABARI . The recent versions have been written in PERL and can do both lead-sentence filtering and full-story filtering, though only for AFP stories. The source code as well as the compiled program is posted at the web site. Earlier versions (with source code) are available in C and Pascal, and some of these handle wire sources other than Reuters and AFP.

Managing the source text for a long time series is a major task, and if you are not familiar with the Macintosh, you might want to do these on a WINDOWS or UNIX platform. Because TABARI input and output are both in flat ASCII files, this is not difficult: The PANDA project, for example, did all of their reformatting and text management work in a WINDOWS environment and switched to the Macintosh only for the actual coding with KEDS.[1]

A.2 ACTOR_FILTER

This program locates potential new actor names in a file of TABARI input records by looking for strings of consecutive capitalized words and comparing these against an existing sets of actor names and a list of stop words. The output of the program is a keyword-in-context (KWIC) file sorted by the frequency of the actor name. The program basically runs in batch model, controlled by an optional text file.

[1] then subsequently wrote their own coding program for WINDOWS. We have also heard from credible sources that deep in the bowels of a certain U.S. government agency sat a lone Macintosh expressly for the purpose of running KEDS .

We've used this program to develop a number of new actor dictionaries for coding internal events, and we've found that it significantly decreases the time required to modify the *.actor* files, although the names then must be manually entered . The KWIC index typically has quite a few false positives – in particular names that are preceded by distinct descriptive adjectives – but it is a fast way to figure out who is who in a new set of data. The program is in beta status, but seems to be stable. The compiled program is posted in the "Software" section of the web site.

NOTE 2011.12.01: This program has largely been superseded by a pair of Python programs, `PoliNER` and `CodeCatcher`. `PoliNER` does systematic named-entity-recognition, using a broader set of stop words and abbreviations that were used in `Actor_Filter`. `CodeCatcher` take the output of `PoliNER`, sorted by frequency, and uses a keyboard-based interface to generate the appropriate new dictionary entries. These are not quite ready for posting on the web site but are available on request.

A.3 CountryInfo.txt

`CountryInfo.txt` is a general purpose file intended to facilitate natural language processing of news reports and political texts. It was originally developed to identify states for the text filtering system used in the development of the Correlates of War project Militarized Interstate Dispute data set MID4, then extended to incorporate *CIA World Factbook* and *WordNet* information for the development of TABARI dictionaries.

File contains about 32,000 lines, covering about 240 countries and administrative units (e.g. American Samoa, Christmas Island, Hong Kong, Greenland). It is internally documented and almost but not quite XML: The major fields are delimited with tags of the form $< tag >$ ... $< /tag >$ but elements inside are delimited with line feeds. Converting this to strict XML would be a relatively simple programming exercise for anyone who should be working with the file in the first place. File is UTF-8 with Unix line feeds and will need to be converted if used in a Windows system.

Fields include

- Country name in English
- Adjectival forms and synonyms of the country name, including some non-English versions of the name
- ISO-3166 numeric, alpha2 and alpha3 codes, FIPS-10 code, IMF code, COW alpha and numeric codes
- Capital city
- Cities with populations over 1-million
- Regions and geographical features (WordNet meronyms)
- Leaders, 1960-2008 (rulers.org)
- Members of government, 2003-2010 (CIA World Leaders)

The web site has both the original file and an *.actors* dictionary derived from it. Note, however, that the leader names are typically complete names, rather than the form of the name generally encountered in news reports, and will need further processing; a later version of the file may incorporate this.

A.4 NEXIS_VERIFY

This utility program goes through a list of TABARI records and checks the dates for missing intervals, bad date formats and the like. Date strings are tagged on any of the following conditions:

1. Date is not in the range 790415 to 970610, the current NEXIS limits on Reuters
2. Consecutive dates are separated by more than 4 days
3. A date occurs before the date of the previous record.

These conditions can be modified by setting parameters in the source code. Both the compiled program and the C source code are posted.

Based on long experience with the vagaries of Reuters and NEXIS records, we strongly recommend running this routine before coding, particularly if you subsequently intend to aggregate data using KEDS_COUNT.

As of February 2006, we are working on a more general replacement for this program called SANITY.C, which checks for the proper construction of CAMEO actor and event codes as well as the consistency of dates. In our experience, it is *very* easy to get a large event data set messed up and this, in turn, will cause problems in analytical programs, so you are well-advised to run some sort of consistency check before using a data set (including our data sets...).

A.5 KEDS_Count

This program aggregates event data by time period and dyad. The program was originally designed to work with output from TABARI but will work with any tab-delimited event data. The program operates in a "batch" mode by first reading a "command file"—created using a word processor—that defines the characteristics of the aggregation. It then reads through the event data files and produces a tab-limited output file for each dyad. These output files can be read into a spreadsheet or statistics program.

NOTE 2012.01.07: This program has largely been superseded by Will Lowe's R package, *events*, which is available from the CRAN website.

Appendix B

Project Management Suggestions

B.1 Selection, Training and Care of Coders

Most of the value-added from the KEDS project has been provided by the twenty or so coders[1] who have been involved with the project and devoted thousands of hours to refining the *.actors* and *.verbs* dictionaries that are essential to producing data. Without this effort, KEDS and TABARI would merely have been useful proof-of-concept exercises; with them, we have produced a lot of data sets that can be used in political science research and policy applications.

Dictionary development requires a great deal of skill and training. Fortunately, compared to most data-generation processes in the social sciences, it is relatively interesting, presenting an ever changing set of puzzles. For the right person, it is a very interesting job: in contrast to many projects we are acquainted with, we have *very* low turn-over in personnel.

We have recruited coders almost exclusively from KU's undergraduate honors program, in many cases from students we have identified in undergraduate honors classes we have taught. In most cases, the students are political science or international studies majors, though we have never made this a pre-requisite. Most are native speakers of English, although we have had some excellent graduate student coders who are not native speakers. The training in English grammar that non-native speakers have had can actually give them some advantages over the intuitive knowledge of native speakers, and international news wire stories are always written with the fluent non-native speaker as the target audience. [2]

Beyond that, we've had a difficult time figuring out what does or does not make a good coder. One of the smartest and most politically-saavy undergraduates we've known—someone we were certain would be perfect for the job—turned out to be absolutely incapable of coding (we found other tasks for her. . .). When we hire a cohort of six new coders, we can figure that

[1] We consistently have used this term, although "dictionary developers" is probably a more accurate term.

[2] Often as not news wire stories are produced by "stringers" who are native to the area they are reporting on. The stereotype that Reuters and AFP reports are produced by "Western white guys" ignores the reality that very few stringers are Western or white, and quite a few of them are not guys.

one or two will disappear after a month or two. The rest will stay around until graduation (or after).

We have discovered that the best way to hire coders is to have our existing coders do much of the interviewing (with one faculty member and either the data manager or a graduate student also participating). The "millenium generation" of honors students have spent their short lifetimes perfecting their resumés: "High school senior honors project: Amnesty International delegate to the Bonzo district of Jerkmenistan; personally intervened to stop ethnic cleansing campaign against the Calabash minority." These tactics have been finely/expensively honed for the express purpose of conning potential Boomer employers and admissions committees, but fail miserably when subjected to the assessment of peers. Peer interviewing—with the occasional principal investigator over-ride—is also useful in increasing the likelihood that the individuals can work as a team.

Once hired, coders typically go through a period of about six weeks (10 to 15 hours per week) of practice coding on sample texts before being unleashed on actual dictionary development. This typically begins with a day-long "coding camp" that provides both an overview of event data research in general, an introduction to TABARI , and an extended discussion of the event and actor coding schemes. After this, most of the training is done by peers: for this reason, we've always tried to make sure that we are training the next cohort of coders at least a semester before the previous cohort has graduated.[3]

In the traditional structure of social science research projects, the undergraduates would be supervised by a graduate student. We have done this at some points, but in recent years we have been able to hire a 30 to 40 hour-per-week "data manager," usually selected from among the graduating undergraduates. This provides both continuity and an individual who is acquainted with the nuances of learning to code as an undergraduate. Our data managers have usually worked one or two years before moving on to law school. We also have had several graduate students associated with the project for multiple years, particularly in the KEDS phase of the project.

Our projects have generally been run using a teamwork model rather than a strictly hierarchical model. We typically have project meetings every two to three weeks where we review current tasks, keep students appraised of our ever-erratic funding situation, identify bugs or possible new features for TABARI, and review leads that seem problematic with respect to our coding schemes. "Senior staff" meetings involving only the principal investigators, data manager, and graduate students are held when appropriate. Staff turnover is reduced by the liberal applications of cookies, donuts, birthday cakes, and other inducements to attend meetings that, due to scheduling problems, are invariably either early in the morning or late in the afternoon.

B.1.1 Additional project management suggestions from Joe Pull's *Ode to Coding*

- Working alone allows much higher productivity. This is especially noticeable when working in the evening, away from the assorted noises of the office environment. There are some pretty good views of Lawrence in the dark from the windows, too.

[3]This strategy requires a steady flow of funding, which we've been fortunate to have. An interuption in coding generations is not a disaster—we experienced one in the late 1990s—but if, for example, internal "bridging grants" are available so that this can be avoided, these are worth pursuing."

- Know the coding scheme. Spending time learning it allows for much faster coding and saves time in the long run.
- Don't try to code for too long at a time. No one can do it well. Two hours is about the limit for most people in one sitting. Even making it that long is sometimes a challenge.
- Consistency is crucial when assigning event codes to phrases; this requires good communication between individual coders and between project directors and coders. This is the most important during the initial training phase.
- A cooler room temperature helps prevent coders from falling asleep during late summer afternoons on the 6th floor of Blake Hall.
- Having other projects to break up the coding is a good idea. Researching date restrictions, downloading new data, etc. give coders a chance to use different parts of their brains.

B.2 Using TABARI: The Unix approach versus the Swiss Army Knife approach

In the 40 or so years of software development, two general but diametrically opposed approaches have emerged that define the limits of what a particular program should be able to do. One approach—epitomized by the tools found in the Unix operating system (see Raymond 2003) —is to sub-divide tasks among a large number of small programs that are optimized for specific tasks but compatible with each other. In Unix, a complex task is usually performed by linking a series of programs in a script, with each small program taking as its input the output of the previous program.

The alternative approach is to put as much functionality as possible into a single program. This approach has been used in most consumer-oriented software, which consequentially boast "feature lists" that run for multiple pages. This approach has been vociferously defended in both U.S. and European courts by Microsoft when it (supposedly) integrated an internet browser previously considered a stand-alone programinto the Windows operating system.[4]

In a perfect world of flawless programming and unlimited computer resources, the fully-integrated approach appears to be preferable from the perspective of the user. In the real world, it is a nightmare for programmers, leading to bloated programs that continually stress the limits of the user's hardware, and to software instability where, for example, a flawed grammar checker will cause an otherwise debugged text editor to crash. Programmers also complain that while integrated software tends to do a lot of things, it does all of them less

[4] As of February 2006, Microsoft has successfully countered anti-trust efforts in the U.S. but is still having serious legal problems in the E.U. This may or may not be related to the differences in the two systems with regard to public financing of political campaigns.

Meanwhile for past two years Microsoft has been running a very extensive set of advertisements portraying its users as [literally] dinosaurs. They said it, not us. In 1986, Apple famously portrayed WINDOWS users as lemmings. Poor WINDOWS users don't get no respect.

To continue the tirade across the decades, in 2011 Apple managed both to briefly become the most valuable company in the world based on market capitalization, and in July, briefly held larger cash reserves than the U.S. Treasury. Not bad for a bunch of hippies. Now, if we could just get the Penn State College of Liberal Arts information technology support—Motto: Ignore it and it will go away—to pay attention to Macintoshes.

well than a specialized program could do, just as none of the tools on a Swiss army knife works as well as the same tool on its own.[5]

Contemporary software design is consequently a product of the tension between marketers who wish to maximize their feature lists and programmers who wish to keep individual components isolated and simple. There are limits at both ends. For example while MS-WORD has fully incorporated (with varying degrees of success) spelling and grammar checking, mail-merge, equation construction and a graphics editing—all of which were once found in free-standing programs—and WORD itself is integrated into MS-OFFICE, OFFICE maintains word processing, spreadsheets, presentation and email/virus-propagation as distinct programs. On the UNIX side, the tremendous popularity of the PERL language is undoubtedly due to part to its integration of a number of functions that were previously available only in specialized routines such as GREP and AWK, and the feature lists of programming tools such as EMACS and GCC match the complexity of WORD or EXCEL.

It will come as no surprise that the design philosophy of TABARI is closer to that of UNIX than to that of Microsoft, and in fact will be closer to the UNIX approach than KEDS was. This is partly out of necessity—to date, virtually all of the programming of TABARI is still being done by only one person—and partly out of an attempt to impose discipline on the aforementioned programmer. Most "user friendly" features are actually a lot more fun to write, and definitely are easier to debug, than the core parsing and coding functions. But parsing and coding can be done only by TABARI , whereas (for example) the reformatting of input and output can be done by many programs.

B.3 Skill sets your project may need

Features in TABARI have been (doubtlessly imperfectly...) prioritized on the assumption that the user have the following sets of tools and skills:

Level 0

User knows only the basic features of their operating system and sufficient knowledge of the English-language, and motivation, to read the fact-filled TABARI manual (or has been instructed by someone else who has). Project will be using input text that has already been formatted, and TABARI alone will be used. Level 0 is the typical skill set for a beginning undergraduate coder.

Level 1

Level 0 skills plus the ability to use a text-only editor with sophisticated search-and-replace capabilities, such as BBEDIT on the Macintosh or emacs in Linux. Level 0 is the typical skill set for a graduate student research assistant who does not have programming skills, or an experienced undergraduate coder.

Level 2

[5] One can continue this analogy by noting as more tools are incorporated into a single Swiss Army knife, it becomes increasingly difficult to use any of them effectively.

Level 1 skills plus the ability to writes programs in the PERL language that will handle text filtering and formatting. We recommend that any project making extended use of TABARI have at least one person with Level 2 skills. Knowledge of PERL (as distinct from, say, C++ or VISUAL BASIC, either of which are perfectly adequate for writing text filters) is not absolutely necessary, but the advantages of PERL are so great – and virtually all KEDS project filters are now written in PERL – that it is worthwhile to have a technically-proficient research assistant learn PERL if they do not already know it.

Level 3

Level 2 skills plus sufficient knowledge of C++ to be able to modify the source code of TABARI. By the standards of the professional programming community, the required C++ skills are very limited (in fact, knowledge of C and just enough familiarity with C++ syntax to follow the implementation of classes is probably sufficient); experience in navigating through a large program familiarity with linked-lists and manipulating null-terminated strings[6] and debugging skills are probably more important. We would recommend that any multi-investigator project planning to use TABARI for more than a year should try to hire (or train) such a person.

B.4 Recommended utilities

A well-equipped project should have available (and know how to use effectively) the following software:

- text-only editor such as BBEDIT or TEXTWRANGLER
- PERL
- The open-source GCC compiler suite

For the immediate future, this is the only ancilliary software that we will assume.

These skill sets apply only to the use of TABARI itself—they do not encompass the linguistic skills required to effectively develop dictionaries, or the statistical skills required to properly analyze event data once these have been produced. We hope to systematically address these issues at a later date.

References:

Raymond, Eric S. 2003. *The Art of Unix Programming.* Addison-Wesley.

[6] Which is to say, familiarity with *pointers*, a topic no longer taught in JAVA-oriented computer science programs.

A programmer familiar only with JAVA will have a very difficult time following the TABARI source code. See `http://www.joelonsoftware.com` for an extended discussion of this issue.

Appendix C

High Volume Coding

In 2007-2009, we became involved in a project that could be characterized as "high-volume coding"—instead of data sets involving tens of thousands of records, this project involved tens of millions, and the original text files were about 27-gigabytes in size. We successfully managed this (more so the second time around) and a few notes from this experience might be useful for other researchers

C.1 Input File Management

The KEDS web site (`Software/Utilities`) contains a series of small PERL programs called the TABARI High Volume Suite that we developed for high volume coding. These programs—which presume that the records are already in TABARI input format—chronologically sort the data, remove duplicate records within the same day, and break a large file into more manageable pieces. The Read.Me.txt file explains how the programs are used.

The other "lesson" we've learned from high-volume coding exercises is that using machines with adequate disk capacity is absolutely essential, and all the more so now that disk space is very inexpensive. Input file processing usually involves generating multiple copies of the texts, and if you find yourself running low on disk space, your productivity will slow dramatically. Having disk space available that is ten times the size of your initial file is probably a good rule-of-thumb.

We also did most of our high-volume work on a Unix server rather than a desktop: some of the processes take a fair amount of time to complete, and tend to run substantially more quickly on a server which is not juggling the dozen or so graphics-intensive programs which are now running on the typical desktop.

C.2 Anomalous Records

Despite all of the debugging we have done to date, it is still possible that a very large data set will cause TABARI to crash: the 26-million record set we coded in 2009 ran without

problems on our Levant dictionaries, but when we switched to some more comprehensive new dictionaries, three records (out of 26-million...) caused a crash.

In the short term, the solution is to just get rid of these. The most efficient way to do this is run commands of the form

```
./TABARI.0.7.2b2 -ad LN.coding.project.n
```

where the `-ad` option will display the date and record serial number (assuming you have these... they are a very good idea...) as the data are coded. Parallel processing—discussed below—is particularly useful for this phase of coding.

If the system crashes, usually the last visible date will be the offending record (sometimes the problem is in the following record). The actual record can then be located using a global search on the serial number.

Reading million-record files into a text editor can be time-consuming: if you just want to look at the record, the UNIX `grep` command is considerably faster. The command

```
grep -A 20 02371261 LN.date00.06.15
```

looks for all instances of the string 02371261—this was a unique story serial number—in the file LN.date00.06.15 and prints out 20 lines following the match. The UNIX `sed` utility could then be used to remove these lines, or the file read into a text editor at this point.

C.3 Parallel Processing

High-volume event coding is very well suited for parallel processing, since the coding of each story (except where pronoun forwarding is used) is independent. Consequently "parallel" processing involves nothing more than breaking the input data into chunks, processing each of these simultaneously, and then recombining the coded files.

In our coding of the 26-million sentences. These were first split in 26 files using LN.DATA.DIVIDE.PL from the TABARI High Volume Suite, then uploaded onto a 13-node LINUX parallel cluster. We created a series of distinct project files that contained the same dictionaries, but differed in the $< textfile >$, $< eventfile >$ and $< errorfile >$ fields, e.g.

```
<textfile>    /var/tmp/LN.date00.06.3
<textfile>    /var/tmp/LN.date00.06.4
<eventfile>   LN.4.evt
<errorfile>   ICEWS.4.errors
```

In the system we were using, it made sense to move the individual text files to the `/var/tmp` directory, which is separate for each node. Your configuration may differ. [1] The various *.verbs*, *.actors*, and *.options* dictionaries, however, were read from directory common to all of the processors, so these were common to all of the runs.

An example of an overall program file

[1] Your configuration may also have documentation...this being the University of Kansas, ours didn't...

```
#ICEWS Phase 1 recode, beta 1, 24 june 2009
<dochead> ICEWS Phase 1 recode, beta 1
<dochead> Text source: ICEWS Phase I 4-sentence set
<doctail> For further information on the TABARI automated coding program and CAMEO
<doctail> coding system, consult http://web.ku.edu/keds
<coder> Kansas
<actorsfile>National.fixed.actors
<actorsfile>CountryCodes.fixed.actors
<actorsfile>Internatnl.actors
<actorsfile>nouns.adject.null.fixed.txt
<agentfile>  ICEWS.agents
<verbsfile>CAMEO.080612.verbs
<optionsfile> CAMEO.09b5.options
<textfile> /var/tmp/LN.date00.06.17
<textfile> /var/tmp/LN.date00.06.18
<eventfile>   LN.11.evt
<errorfile>   ICEWS.11.errors
```

Once this has been set up, just run

```
./TABARI.0.7.2b2 -aq LN.coding.project.n
```

in each node, where n corresponds to the project number. The `-aq` option invokes the `A)utocode N)one` command and skips the `Usage` query following the autocode, which, in combination with a < *coder* > line in the *.project* file, means the program codes with no further intervention. Then run HV.EVENTMERGE.PL from the TABARI High Volume Suite to combine the resulting output files.

In terms of actual coding time, the 26-million sentences were processed in about 6 minutes, an effective coding speed of about 70,000 sentences per second. The initial set-up, of course, takes quite a bit longer, but this was particularly useful for weeding out anomalous records, which in an earlier project, before we were using a parallel implementation, was the most time-consuming part of the project.

It should be relatively easy to add formal parallel processing code to TABARI using the MPI system[2], but this method is quite workable. Once we've got access to a better maintained system, we expect to add the appropriate MPI code.

[2] which is not properly installed on our machine...

Appendix D

TABARI versus KEDS

Generally, TABARI works the same way that KEDS worked, so KEDS *.text*, *.actors* and *.verbs* files can be used without any changes to the files, beyond translating them from the Macintosh "Classic" format to the OS-X UNIX format (see section 1.5.1). This compatibility will be maintained indefinitely. However, TABARI *.actors* and *.verbs* files are not compatible with KEDS if new word types such as nouns, adjectives, attribution verbs and time-shifting are used. The *.project* file is quite different—see section 4.1—and only a small number of the commands in the KEDS *.options* file have been implemented. TABARI does not implement the optional (and little-used) CLASSES and RULES facility of KEDS , but instead implementes some new classes of word (notably adjectives and nouns) and has a built-in passive-voice grammatical transformation.

Because the event data community has generally reached a consensus that machine-assisted coding is not a good idea because it is neither transparent nor reproducible, TABARI does not implement the machine-assisted coding facilities of KEDS. Again, if you need these facilities, use KEDS. (Or, since this is open-source code, write the appropriate routines.)

TABARI is now substantially more reliable and more stable than KEDS. While at least some research projects at other institutions are still using KEDS for purposes of continuity—it has some features TABARI does not have—we have been using TABARI for all of our dictionary development and coding work for about five years. It is certainly faster—the program codes about 8,500 records per second even on an inexpensive machine, a $500 1.2 Ghz G4 Mac Mini. Upgrading to a 1.6 Ghz G5 gives a speed of 10,500 events/second.[1] The coding speed on a Mini is about 300-times faster than KEDS, and about 33-million times faster than human coding.[2]

In order to maximize compatibility between operating systems, TABARI uses a very simple UNIX-style keyboard-driven "terminal" interface rather than the GUI—Graphical User Interface—approach of KEDS . This is designed to work on a screen that is 48 lines high by 120 characters wide. At one point we considered adding a GUI, but coders who have worked

[1] This test involved coding around 100,000 records using dictionaries with about 700 actors and 4,000 verb phrases. See further discussion in Section D.0.2

[2] In a large-scale project, humans working from paper sources will typically code about 6 to 10 events per hour. They *initially* code at a faster rate, but this pace soon slows as fatigue and boredom set in. When re-training and personnel replacement are taken into consideration, the 6 to 10 event pace is probably an optimistic estimate.

with both KEDS and TABARI consistently prefer the simpler interface, which is much faster once one has memorized a few simple menus. The interface is implemented using the UNIX "ncurses" terminal library, and consequently we now have 100% compatibility between the Macintosh and UNIX /LINUX versions, as well as allowing the program to run remotely from a server.

TABARI codes in a fashion largely comparable to—but not *identical* to—KEDS. The following table gives correlation r between WEIS-coded data produced with TABARI version 0.4 and an earlier KEDS data set on 238-months of Levant data (April 1979 to December 1998) and on the 1990-91 Iraq-Kuwait crisis by major WEIS event category.[3]

Table 1
WEIS Coding of 1979-1998 Levant Data

All events	correlation	N KEDS	N TABARI
Verbal cooperation	0.95	54,543	58,259
Material cooperation	0.90	6,208	7,593
Verbal conflict	0.94	13,262	15,296
Material conflict	0.95	18,673	22,512

Table 2
WEIS Coding of 1990-91 Iraq-Kuwait Crisis

Mediator activity[4]	correlation	N KEDS	N TABARI
Verbal cooperation	0.87	4,818	5,066
Material cooperation	0.74	598	583
Verbal conflict	0.82	950	971
Material conflict	0.85	727	821

These two data sets were coded with somewhat different dictionaries, so some of the differences are due to dictionary changes. However, there are at least four major potential sources of differences between KEDS and TABARI

- TABARI disambiguates verbs that can also be nouns (e.g. ATTACK and FORCE) by detecting the presence of the articles A_, AN_, and THE_. It also disambiguates proper names that could be verbs by looking at mid-sentence capitalization. Nouns so identified do not go into the verb count in the complexity filter, so TABARI codes a number of stories that KEDS skipped.
- Subordinate and dominate codes are handled in a more straightforward fashion than in KEDS (see section 6.6.1).
- TABARI eliminates an assortment of pathological coding behaviors in KEDS (mostly involving unlimited word skipping when matching phrases).

[3]Major WEIS categories
Verbal cooperation: WEIS categories 02, 03, 04, 05, 08, 09, 10;
Material cooperation: WEIS categories 01, 06, 07;
Verbal conflict: WEIS categories 11, 12, 13, 14, 15, 16, 17;
Material conflict: WEIS categories 18, 19, 20, 21, 22

- TABARI is considerably more sophisticated at handling compound phrases, which may account for the consistently higher event counts in TABARI .

In short, even with identical dictionaries, TABARI will not exactly duplicate KEDS coding. (If you want to exactly duplicate KEDS coding, use KEDS. . .).

D.0.1 Further Observations on System Speed: October 2009

During the summer of 2008, TABARI.0.6. was used in a major coding project that involved coding 25-Gb of text (6.7 million stories, 253 million lines of text) for around 30 countries. This project used *.actors* dictionaries with extensive lists of sub-state actors (around 14,000 entries), and a considerably more extensive CAMEO *.verbs* dictionary (around 10,000 entries) than we had used in the earlier tests.

These larger dictionaries slowed the program considerably, though not proportionately: coding dropped to "only" about 2,500 records per second, though obviously for a coding task of this magnitude, the resulting coding now required hours rather than minutes.[5] Meanwhile, of course, the consistent increases in CPU speeds that had been seen during the first two decades of the development of personal computers have, for the most part, ended, and the speeds we see on 2008 equipment are pretty much what we saw on 2004 equipment.

Various enhancements in version 0.7—notably agents and regular word endings—are explicitly designed to reduce the size of the dictionaries, which should provide greater speed as well as greater consistency. However, as with many problems, the long-term solution is simply to do the coding in parallel. There are few obvious ways to do parallel processing inside TABARI—though contemporary compilers probably are finding a few shortcuts—but trivial parallelism can be achieved by simply splitting the files across multiple processors and running these simultaneously.

D.0.2 Further Observations on System Speed: June 2009

The summer of 2009 saw the high-volume coding exercise described in Appendix C, this time involving about 26-million records. We gained access to a 16-node, 64-processor cluster computer running Red Hat Linux—which is quite small by parallel computing standards—and with large dictionaries achieved throughput of around 70,000 events per second. This will continue to scale more or less linearly depending on the architecture of the machine you are using, but the short answer to the problem of coding very large data sets does appear to be parallel processing at the level of the input files. The details of this are described in the Appendix "High Volume Coding."

While the bulk of this speed is achieved through the application of multiple processors, even the single node on the cluster was coding around 6,000 records per second despite using large, comprehensive dictionaries. Two factors seem to account for this. First, LINUX machines are substantially faster than MACINTOSH machines—all that eye-candy comes at a price.[6] Second, the use of *.agents*—available in TABARI .0.7—does seem to be increasing the coding

[5] We were doing full-story coding, so each story generated anywhere from a dozen to a hundred or more sentences.

[6] The WINDOWS version of the program also runs substantially faster than the OS-X version.

speed by eliminating dictionary entries that have similar initial characters (specifically the country name), which increases the speed at which phrases are identified.

Appendix E

Program Release History

Release 0.2 **April 2000**

Major features of KEDS

Release 0.3 **January 2002**

XML input files
Time-shifting
Issues
Problems file
LABEL: option
Output options for record ID information

Release 0.4 **March 2002 - July 2003**

Pronoun coreference forwarding
Automatic validation; validation suites in TABARI.Demo file
Automatic passive voice detection
Numerous small interface and input bugs corrected
Alternative pattern matching : `{ xxx | yyy | zzz }`
< *nouns* > and < *adjectives* >, compound noun phrases
Text of matching string displayed
< *Attribution* > facility
Pattern length calculations correctly deal with alternative patterns
Additional validation records in TABARI.Validate file
Compound adjective phrases
FBIS options
DOCUMENT option in OUTPUT

Release 0.5 **April 2005 - June 2005**

Ncurses interface; OS-X compatibility
TABARI.Parse.html output

Complete re-write of `checkPattern` routine
Elimination of all code generating gcc 4.0 warnings
Frequencies
Event file internal documentation
Correction of some bugs in compound phrases

Release 0.6 August 2007 - July 2008

command line options
additional input diagnostics
Mac OS-X and Linux versions merged
$< errorfile >$
OUTPUT: PATTERN
Duplicate entry detection when reading dictionaries
Automatic detection of ALL CAPS conditions
intensive debugging of compound phrase marking; debugged program successfully coded 25-Gb of text (6.7 million stories, 253 million lines of text)

Release 0.7 January 2010 - April 2011

.agents dictionaries
regular noun and verb endings
irregular verb forms
multiple dictionaries
initial menu and mergefiles.cp eliminated
nonevent coding
$< system >$ and $< run >$ commands in *.project* file
.errors file no longer contains redundant warnings
synonyms and multi-line date restrictions in *.actors*
ACTORS, AGENTS, PARENT options in OUTPUT:
-version and -c command line options

errors identified in JABARI conversion corrected:
- passive voice
- incorrect partial matches due to verb endings
- additional agent debugging

Release 0.8 January 2012 - February 2014

synonym sets
Elimination of all code generating gcc 4.2.1 and Xcode 4.2 warnings
Input errors report line where the error occurred
-p, -t, -o, -f command line options `<outputfile>` command DEFAULT *.options* command
{OPTION:...} in `V)alidation` records
OUTPUT: SHIFT added; sorting of time-shifted records eliminated
OUTPUT: LOC
COMMA:
SET: CONVERT AGENTS

E.1 Overviews of Major Releases

These were the descriptions posted on the KEDS web site when the various releases were available.

E.1.1 0.2 (31 March 2000)

This release of the TABARI source code is fully functional for both the Linux and Macintosh operating systems. TABARI is completely compatible with KEDS *.actor* and *.verbs* dictionaries, though there are a few pathological "features" in KEDS pattern-matching that have not been duplicated in TABARI . TABARI *.project* files are text, rather than customized Macintosh files—see the documentation or the TABARI.demo.project files for an example.

The TABARI.Demo.project will run through the basic features of the program and is a good place to start. This program has been extensively tested using the demo file and successfully run on a 26,000 sentence corpus of actual newswire texts, but it has still had only limited use for actual coding.

E.1.2 0.3.10b1 (5 March 2002)

This release of the TABARI source code implements all of the major components of the KEDS system. For production coding, TABARI is probably now better than KEDS , unless you are using KEDS rules and classes. It is certainly faster—the program codes about 2000 records per second (Mac 350Mhz G3 and Dell 450Mhz Pentium II). This is 70-times faster than KEDS , and about 7.8-million times faster than human coding.

The program has been tested in both the Macintosh and Linux versions against 26,000 Reuters lead sentences from the Levant for the period 1987 through 1990. It codes these without crashing, and gives results that, when aggregated using the Goldstein scale, are very comparable to those of KEDS. Running the TABARI.Demo.project file will demonstrate most of the parsing and coding features that have been implemented.

E.1.3 0.4.7B1 (14 July 2003)

This release of the TABARI source code implements a number of bug fixes and new features that have been incorporated into the program since we began using it in December 2001 for all dictionary development and production work in the KEDS project. Version 0.4.05B2 has been "stress-tested" by auto-coding about 230,000 sentences in preparation for our 2002 International Studies Association paper. This work uncovered an assortment of bugs involving obscure interactions between dictionaries and unusual grammatical constructions; while the program is still not completely crash-proof, it is certainly reasonably stable now. Dictionary development work by our coders uncovered a number of interface bugs, and we've also added several major new features.

E.1.4 0.7.3b3 (1 July 2009)

This version corrected a couple more 1-in-x-million crashes, improved the error reporting, and added substantial new functionality, including multiple actor dictionaries, agents, regular noun and verb endings, and system calls in the *.project* file. It has been used in high-volume ($N > 100-$million records) coding on a cluster computer, as well as in conventional applications.

Appendix F

Ode to Coding

Joseph Pull,
KEDS Project Coder 2000 - 2003 [1]

July 2003

Coding with TABARI is a balancing process. Almost every aspect of good coding walks a line between opposite pitfalls: phrases that are too specific and phrases that are too general; spending too little time on a lead and giving up on it after ten seconds; trying to code for five hours straight and coding for two hours per week. The trick is to stay between the extremes.

It is helpful when coding to keep the long-range goal in mind. With TABARI , coding by hand is a means, not an end. Hand coding [2] is done only to develop the dictionaries that the program runs on; no useful data is generated until the dictionaries are complete and the program autocodes all of the leads over again. Therefore, the point of dictionary development is development, not coding individual leads correctly. Extracting a useful actor or verb dictionary entry from an uncodable or incorrectly coded lead is just as useful as getting an individual lead to code correctly. Additionally, even leads that are coded correctly can be used to extract useful actors and phrases to add to the dictionaries.

This fact is both very freeing and very sobering. It means that any single lead does not so much matter whether it codes correctly or not; the coder need not worry over the inevitable "bad leads" he or she runs into. On the other hand, this also means that correctly coding a single lead accomplishes nothing useful if it requires changing the dictionaries in ways that will cause future leads to miscode. Every potential dictionary alteration must be evaluated as to whether it will cause more harm or more good when it is applied across thousands of leads.

[1] Joe, an undergraduate honors student, was one of the most sophisticated coders to have worked on the project. Before he left for law school (Yale!) we asked him to write down advice for future coders. This essay has been updated slightly to reflect current CAMEO codes and TABARI features.

[2] By"hand coding" is meant the process of dictionary development by human coders to prepare TABARI for autocoding the entire collection of leads. In the context of this document, "Hand coding" does not refer to the use of human coders to manually develop event data without the use of a machine coding program. That would be nasty, brutish, and painfully long.

Before diving into the assorted problems that coders face, it should be pointed out that TABARI is a very good program. Simple leads and straightforward events get coded correctly the vast majority of the time.[3] For this reason, our primary attention is focused on those cases where regularly occurring structures in leads cause TABARI to systematically miscode. A focus on the inevitable occasional problem should not cause despair; when confronting a problem lead it is easy to forget that one just finished breezing through ten correct leads in a row.

F.1 Meditations on the Dictionaries

The coder's task is to add entries to four lists of words and phrases, called dictionaries. The two primary ones are the ACTORS and VERBS dictionaries. The NOUNS and ADJECTIVES dictionaries are supplementary lists used to weed out common problem words and phrases.

F.1.1 The Good

The simple structure of the actors dictionary—an alphabetized list of names—is reflected in the simplicity of using it. The coder, so long as he or she is sober, need not worry much about causing problems by adding to the actor dictionary. The most likely trap the coder is likely to encounter with the actor dictionary is not adding to it enough. TABARI's conjunction-recognition feature and the way it searches for actors make it desirable that all permutations of a leader's country, title, and name be included as separate entries. For example, KING_ABDULLAH, KING_ABDULLAH_II, and JORDAN'S_KING_ ABDULLAH, should all be separate listings in the actor dictionary with the code [JORGOV]. It is tempting when coding to see that the phrase "Iraqi Prime Minister Tariq Aziz met with Yasser Arafat" correctly generates the event IRQ 022 PALPLO (from IRAQI MET WITH YASSER_ARAFAT) and move on to the next lead, but the coder needs to recognize that it ought to generate IRQGOV 022 PALPLO and add IRAQI_PRIME_ MINISTER_TARIQ_AZIZ [IRQGOV] to the actors dictionary. Adding multiple spellings and arrangements of a name risks no miscodes and offers the potential for better interpretation of leads.

F.1.2 The Bad

There are no bad TABARI dictionaries, only bad TABARI coding schemes...

F.1.3 The Ugly

Verb phrases are more complicated. There are only a limited number of ways a person's name can be written (though the number of different ways journalists refer to "Lebanon's Shiite Muslim fundamentalist militia Hezbollah" is astounding), but the English language's flexibility allows for much more variation in wording to describe the same event. For example, "met," "consulted," "lunched with," "arrived for talks with," "visited," "headed to," and

[3] That is, after the dictionaries have reached a reasonably advanced stage.

"hosted" are just a few of the ways that meetings between heads of state are indicated. Additionally, the same word can mean very different things; "fired" can mean a termination of employment or a military engagement.

This creates the quandary that the verbs dictionary must be large and contain many patterns, while at the same time the coder must be more cautious about changes made to the verbs dictionary than to the actors dictionary. When creating a verb entry, two questions must be asked. First, will the entry result in a correct coding of the lead in question (making the assumption that the current lead is fairly representative of future leads)? Second, could the entry create miscodes if a word in it is used in a different way in the future? The verb "attacked" by itself cannot be coded as [190] "use conventional military force" because it is often used to describe verbal criticism, as in "attacked the Russian policy." Instead, longer phrases including the verb must be separately entered, such as TANK ATTACK [194] and ATTACK IN_ SPEECH.

The problem of multiple interpretations tends to decrease as the length of a verb phrase entry increases, but the danger still needs to be considered. Unfortunately, using only long verb phrases introduces a different problem: applicability and recurrence. The point of machine coding is to work from a list of common words and phrases. Using only long verb phrases will allow a coder to correctly treat individual leads, but long phrases are less to appear in future leads, and thus the effort in creating them is wasted. A short, common phrase but one that excludes incorrect interpretations is the goal of a verb dictionary entry.

Verb phrases, then, are another example of the balancing act in coding. Walking the line between phrases too short and too long, too general and too specific, is the task at hand. A good way to approach the creation of a verb phrase is to take the lead and "boil it down," letting extraneous words and phrases evaporate until only a core of the lead that corresponds to one and only one CAMEO category remains.

For example, take the lead "Israeli Prime Minister Ehud Barak headed here Sunday to discuss the Middle East peace process with Egyptian President Hosni Mubarak, whose top advisor said a Syrian-Israeli settlement could take a year." ISRAELI_PRIME_ MINISTER_EHUD_BARAK is clearly the source; this makes these words unlikely to be useful for inclusion in the verb phrase. The words "here" and "Sunday" are irrelevant to the CAMEO coding scheme, so they can be dropped. The opinion of Mubarak's "top advisor" is extraneous to the event of the meeting and also not of much significance to the CAMEO scheme, so it can be dropped as well. This leaves us with "ISRAELI_PRIME_ MINISTER_EHUD_BARAK *headed here* Sunday to discuss the Middle East peace process with Egyptian President Hosni Mubarak*, whose top advisor said a Syrian-Israeli settlement could take a year.*" The target is now clear as EGYPTIAN_PRESIDENT_ HOSNI_MUBARAK. Closer consideration of the CAMEO scheme reveals that the topic of discussions is irrelevant; what we are interested in is the fact of the meeting. "The Middle East peace process" can thus be discarded.

Now we have "ISRAELI_PRIME_ MINISTER_EHUD_BARAK *headed here* Sunday to discuss *the Middle East peace process* with EGYPTIAN_PRESIDENT_HOSNI_ MUBARAK, *whose top advisor said a Syrian-Israeli settlement could take a year.*" There isn't any extraneous information left; the lead has been boiled down to its essence, which we see as "ISRAELI_PRIME_ MINISTER_EHUD_BARAK headed to discuss with EGYPTIAN_PRESIDENT_HOSNI_MUBARAK." Substituting our TABARI symbols, we have "$ headed to discuss with +". This is our verb dictionary entry. We use the first verb in the phrase as the anchor[4], since TABARI uses the

[4]The verb heading in the dictionary under which the verb phrase in entered

first verb it encounters in the lead as the preferred one. We enter "* TO_ DISCUSS WITH_ +" in the verb dictionary under the verb HEAD.[5]

Now all that remains is to assign the phrase a CAMEO code. In assigning the code, our primary reference is the verb dictionary entry itself, not the lead in its entirety.[6] We are developing the dictionary, not coding the lead, and the dictionary entry will remain long after this particular lead is forgotten. This means the CAMEO code must be derived from the text of the verb phrase, not from the lead itself. Our lead above refers to discussions, so it will fall under the CAMEO category for "Consult," the 020 category. More specifically, our phrase uses the verb HEAD, indicating travel, so the phrase receives the code 022, "Make a visit." Since visits are always hosted by someone, the 022 code is paired with the 023 code, "Host a visit." Our complete verb dictionary entry thus looks like this: [7]

HEAD - * TO_ DISCUSS WITH_ + [022:023]

Clearly, adding verb dictionary phrases is much more involved than simply typing in the name of a new actor, or a new combination of title and name. With practice, though, the process becomes intuitive, and the coder soon loses consciousness of going through each individual step.

F.1.4 The Others

The actor and verb dictionaries are used to generate events. In contrast, the noun and adjective dictionaries are used to prevent the generation of incorrect events, and to prevent useful leads from being thrown out. There are two ways in which they do this. The simpler way is by providing a place to file words that look like verbs or actors but really aren't. For example, "Middle East peace process" is a common phrase that would trigger the actor MIDDLE_EAST, as in "Anthony Zinni traveled to the Middle East today." The "Middle East peace process," though, is neither a destination nor an actor—just a theoretical concept. By adding MIDDLE_EAST_PEACE_PROCESS to the noun dictionary the phrase is ignored as a noun—a noun cannot simultaneously be an actor—and the problem is avoided. (This process is even more important regarding false verbs, but that will be discussed later.)

[5]Notice that the $ operator has been left off. This is because TABARI assumes without being told that the source comes before the verb. TABARI searches for "$ * +" as the default pattern with respect to every verb, and when other words are added to a verb phrase it continues to assume that the source comes first unless it is ordered otherwise in the phrase. It normally accomplishes nothing to include the $ as the first character in a verb phrase. $ is primarily intended for phrases where the source does not come first, as in "+ was killed by $." One exception to this rule is when an entry is intended to force the source to be in a specific location with respect to another word, as in "$_WARPLANES *" under the verb RAID. This would prevent a lead like "In the Gaza Strip Israeli warplanes raided office buildings that the Israeli military said housed Hamas forces" from being coded as PALGZ RAID ISRMIL. Instead, it would be coded ISRMIL RAID PALHM.

[6]Of course, a well-crafted verb phrase will capture the essence of the lead in such a way that the code for the two will be identical. This is not true absolutely because sometimes the idiosyncrasies of wording in a lead will suggest something that a verb phrase cannot capture. In these cases, preference should be given to the text of the new verb phrase over the text of the lead in assigning a code, again because the verb phrase is what will endure, not the lead.

[7]Those familiar with the CAMEO scheme will note that the phrase might suggest the code 024:024, "Meet in third location." In this case 022:023 is preferable because there was no location given to suggest a site outside of Egypt. The phrase "* TO_ ^TO_ DISCUSS WITH_ +" would be a better candidate for 024:024, although even this is not certain because a lead could say "Israeli Prime Minister Ehud Barak headed to Cairo to discuss the Middle East peace process with Egyptian President Hosni Mubarak."

The second function of the noun and actor dictionaries (and the reason that they are separate lists) is the weeding out of false clause conjunctions. TABARI is set up to consider conjunctions like AND and OR as the boundaries between separate parts of a compound sentence. "President Clinton met with Binyamin Netanyahu yesterday AND the Israeli Prime Minister agreed to open negotiations with the PLO" is an example of such a sentence containing two events, USAGOV MET WITH ISRGOV and ISRGOV AGREED TO NEGOTIAT WITH PALPLO. However, AND is also often used to join two things together that are not separate clauses, as in "President Clinton met on this bright and sunny day with Binyamin Netanyahu." The danger here is that TABARI will interpret the AND to mean there are two separate sentences. Since neither side of the lead has two actors, it will not be able to generate a $ VERB + structure, and it will conclude, "No events found." The event data from the lead is thus lost.

With the adjective dictionary, this problem is solved. TABARI has a feature that recognizes structures that consist of the sequence ADJECTIVE_AND_ADJECTIVE or NOUN_AND_NOUN. [8] These structures are then treated as parts of a single clause rather than the border between two separate ones. If the coder adds "bright" and "sunny" to the adjective dictionary, TABARI will read "CLINTON MET on this adjective_and_adjective day with NETANYAHU" rather than "CLINTON MET on this bright AND sunny day with NETANYAHU," which is useless. The noun and adjective dictionaries, then, offer a "bridge" over troubled conjunctions. As it turns out, this bridge is useful in a remarkable number of leads.

The noun and adjective dictionary features are relatively recent additions to TABARI, and so right now they remain rather small. However, they are powerful tools. The conjunction problem was one of the biggest sources of systematic errors in TABARI coding of newswire leads, and these two dictionaries go far toward minimizing it. The coder simply needs to remember to use the function, and to remember that TABARI only recognizes the strict format NOUN_AND_NOUN or ADJECTIVE_AND_ADJECTIVE. This means that occasionally multiple-word phrases must be used to trigger recognition, such as APPEALS_COURT. In general, creative use of nounizing and adjectivizing is a good thing, so long as consideration of what potential problems might be created by doing it is retained.

F.2 A Painfully Detailed Look at the Coding Process

Having plumbed the depths of the dictionaries and delved deeply their mysteries, it is time to consider the process that creates them, bit by bit extracting useful names and phrases from the raw material of daily newswire reports. "Coding," it is called, that curious activity that exposes one to the mundane details of what happened each day during a period several years ago. At times quite enjoyable, at times frustrating, at times simply slow, human hand coding is fundamental to the machine coding project. Ironically, it turns out that human

[8] There are separate dictionaries for nouns and adjectives rather than just throwing all the words together into one big list because it is not useful to search for NOUN_AND_ADJECTIVE or ADJECTIVE_AND_NOUN; these structures, if they occur, do not usually need to be bridged. The exception to this—and a point that may allow future addition to the program—is the structure NOUN_AND_ADJECTIVE_NOUN, as in "the president and two advisors." A variation of this structure is the sequence ACTOR_AND_TITLE_ACTOR, as in, "French President Jacques Chirac and Prime Minister Tony Blair." Here the intervening "Prime Minister" will interrupt recognition of the compound actor JACQUES_CHIRAC_AND TONY_BLAIR unless PRIME_MINISTER_TONY_BLAIR is a separate actor dictionary entry from TONY_BLAIRonce again highlighting the need for extensive inclusion of name and title variations in the actors dictionary.

coders and the TABARI program have something in common: neither of them can continue coding for too long a sustained period. [9]

For the human coder, the basic approach to attacking a lead has four steps:

1. Read the sentence from beginning to end.

2. Mentally pluck from among the extraneous words in the sentence the essence of what is being reported. What actually happened that can be translated into event data, as opposed to editorial commentary, prognostication, or useless human-interest detail? Some leads will contain nothing useful; others, multiple pieces of information that can be turned into event data. When something useful is found it will take the form ACTOR ACTION ACTOR (at least within the WEIS or CAMEO framework), where one entity is doing something to another entity. [10]

3. Having extracted the essence of the lead, turn to the TABARI output line, which will show how the program has interpreted the text. Conveniently enough, the TABARI output will also exist in the format ACTOR ACTION ACTOR, where each actor is a represented by a three- or six- letter sequence and the action is represented by a three or four digit number, followed by text to describe what that number represents. Do the essence of the lead and the TABARI interpretation match? If they do, pat the monitor on the top and move on to the next lead.

4. However, often they will not match, or an error code will be generated. This prompts a quick diagnosis as to whether the lead is (a) worth correcting and (b) correctable. If both of these are the case, a trial-and-error session is initiated, where entering new words and phrases into the actors and verbs dictionaries hopefully results in the correct coding of the lead. As has already been seen, actors can be entered without a second thought; they are straightforward and rarely cause recognition or stemming problems. Verb phrases are more difficult; it may take several tries to find an accurate verb pattern that the program will match to the lead. Don't hesitate to try several different dictionary changes while searching for one that will work. Failed changes can always be deleted without causing any harm, and sometimes a new entry that does not fix the current lead might be left in the dictionary because it seems likely to be applicable to future entries.

With the general picture of coding established, a more detailed look at typical coding scenarios is next to consider. The step-three comparison of a lead with TABARI's interpretation of it usually yields one of four situations: a perfect match, an almost match, a waytysm match, or an error code.

[9] Of course, this is for two very different reasons. Human coders cannot code for very long at a stretch because their brains tire and attention spans run out, and they begin to fall asleep, watch the clock, compulsively check their email, and daydream. TABARI cannot code for too long at a stretch because it rapidly completes all the leads it has been supplied with and then has no material on which to continue coding. Still, the statement that the two are similar stands because in the literal sense it is true, and in the literary sense drawing such comparisons is all about justifying the most unlikely statements on whatever grounds possible, level of flimsiness notwithstanding.

[10] The KEDS project uses the word "actor" to refer to a person, country, or organization that can initiate or receive an action: visits, criticisms, economic aid, etc. In a particular lead, an actor will be either the SOURCE ($) or the TARGET (+), depending on whether it is doing the action or receiving it. An action is an incident between two actors that is defined by the CAMEO codebook and referred to by a number, called a "code."

F.2.1 Perfection.

The most desirable of these, obviously, is a perfect match. If TABARI nailed the lead—not only correctly extracting the event described, the source, and the target but also extracting all events in the text and not adding in any extra ones—then the lead can be passed up for the next one. In advanced stages of a coding project, after the actor, verb, noun, and adjective dictionaries have been well developed, perfect matches are quite common, though certainly not universal.

F.2.2 Nobody's perfect.

The second possibility is an almost match. This is a situation where the program has done well, but conceivably could have done better. Almost matches are sometimes close enough and sometimes not, but regardless of how "almost" the result is, the coder should at least briefly try to improve it. Examples of almost matches are instances where the actors are both correct but the event code is not—either it is completely wrong or it is semi-right but there is a more appropriate code available.[11] Alternatively, the event code may be correct but one of the actors is incorrect or missing.

Almost matches can be relatively simple to fix. If the actors are both correct, simply adding a new verb pattern will almost always fix the problem. It may be that a short verb pattern is being picked up inside a longer phrase that has a much different meaning: "said will comply" versus "said will not comply." Simply adding the longer phrase will solve the difficulty. If the actors are not correct, one of them may be missing from the dictionary altogether; adding it will fix the problem. Sometimes an event will be generated between a state and that state's government—ISR and ISRGOV, for example. Often, these will be cases where a politician and his or her title are being picked up separately. "Israeli prime minister Ariel Sharon" may be coded as an event between ISRAELI and ARIEL_SHARON. Here, adding the new actor phrase ISRAELI_PRIME_MINISTER_ ARIEL_SHARON will fix the mistake.

One last type of almost match is the classic reversal of source and target: Palestinian protestors arresting Israeli police. The latest versions of TABARI greatly reduce the incidence of actor reversal, but it still occurs with some frequency. Simply adding a "+ * BY_ $" (or similar) phrase in the verb dictionary will usually solve the problem.

F.2.3 When the world falls down around you.

The third type of match is the "what are you thinking, you stupid machine" match: *waytysm*. These matches are usually generated by leads with unusual or complicated grammatical structures or by idioms or direct quotes. Additionally, there are systematic newswire lead devices that generate *waytysm* matches, the most common being the "So-and-so did thus and such, said $" structure. The COMMA SOURCE SAID sequence at the end of the lead gets dropped by TABARI's feature that eliminates subordinate clauses and then the source's opinion, command, or interpretation gets coded as an actual event.

[11] A semi-right code, in the light of the statistical application of event data, is probably good enough most of the time, especially if it is in the same category (02, 05, etc.) as the desired code. Still, in the pursuit of the best data possible, in view of future leads that may require a new phrase to be even remotely correct, and in view of the negligible effort required to do so, it is a good idea to try to get the code perfectly correct by adding a new verb pattern.

Waytysm leads quite often are simply beyond repair. If the problem is a structural one, as in the example above, there is little a coder can do. Sometimes a useful verb dictionary entry can be extracted from the lead so that this particular lead may be miscoded but future similar leads might be correctly interpreted. Other times, the word sequence that creates the problem can be entered and null-coded, or given a useless code like "make neutral comment." In this way, the lead is not coded correctly, but it is not coded incorrectly either. Removing a source of problems is useful, though not nearly as satisfying as adding a phrase that leads to correct coding.

Sometimes a *waytysm* lead happens simply because a new type of event is reported. Having never been seen before, the event is incompatible with the information the dictionary has been built upon. These types of problems are easy to solve; simply creating the necessary language will do the trick. Sometimes this may be as simple as adding a new verb to the dictionary.

One possible outcome of a confrontation with a *waytysm* lead is an epic struggle between TABARI and the coder with a happy ending. Occasionally, wrestling with a lead and making five trial changes to the dictionary that are undone followed by eight lasting changes results in a difficult lead being correctly (and oh-so-elegantly) coded. There is a drawback to this scenario. One lead, even changed from grossly miscoded to nailed-tight correctly coded, is still only one lead. In the statistical analysis of the data, it will not show up, and there is a risk that such radical dictionary surgery will result in ten leads that were previously coded correctly now being butchered. Also, the investment of fifteen minutes on a single lead is simply not economical, and if the resulting verb patterns added to the dictionary are long and complex, they will never help in the coding of another lead. This defeats the purpose of dictionary development and machine coding.

On the other hand, the epic struggle does have some benefits. For the novice coder, the epic struggle may actually involve a straightforward lead, and the process of working through it yields insight into how TABARI works, how leads get coded, and what works and what does not in the dictionaries. This is learning by doing, and after a few repetitions the epic struggle may shrink to a ten-second surgical strike that epitomizes the purpose of hand coding. For the advanced coder, other benefits accrue. Someone who knows what they are doing will create dictionary entries that, though they may be fairly long, contain common words and that are applicable to future leads. There are also significant morale advantages to an epic struggle[12], both from personally conquering a twisted-grammar Matterhorn and from the future sense of awe that bringing up a new, complex lead that TABARI nails will bring to some fortunate coder. Further, the experienced coder will conduct his or her war against the dictionary in such a manner as to not initiate the miscoding of hundreds of other leads in order to address the eccentricity of one. Considering the benefits and risks, then, the key word regarding the epic struggle is (surprise, surprise) "balance."

In short, the *waytysm* lead can be frustrating, but it offers an opportunity to learn and to glean useful phrases for the verb dictionary. A rational approach where the coder gives the lead a good shot while refraining from obsession with "solving" it is the goal. No matter what the outcome of that particular lead is, there are hundreds more following it. This depressing thought serves to highlight again the purpose behind the whole exercise: dictionary development. If useful dictionary entries are extracted, all is well. It's like working out; how far a basketball player runs or how much weight he or she lifts in any particular practice

[12]Though probably not of the sort that could inspire a blockbuster movie and make one wealthy and famous for having done very little.

session is irrelevant as long as the outcome is better skill during the game (though there is usually a correlation between them).

Two last thoughts about *waytysm* miscodes (and miscodes in general). First, a certain number of problems are the direct result of features of TABARI—the way it interprets a type of word (like verb, conjunction, actor, etc.), the way it addresses punctuation, the order in which it searches for events. In diagnosing a problem like this, it is crucial that the coder understand how the program works: what it looks for, what it ignores, and the order in which it searches. This allows the quick recognition of which problems are fixable and which are beyond hope. Just as how people can often predict how their friends will react to a certain situation, the coder should be able to make a good guess at how TABARI will interpret a lead even before seeing it done. Essentially, the coder must think like the computer. Developing the ability to turn off one's human flexibility—the inherent knowledge, contextual understanding of sentences, and intuition about how things ought to work—and instead process a sentence in the mechanical, rigidly systematic, senseless way that a computer program does is crucial to figuring out why some leads work out right and why others just don't cut it. This, in turn, allows the coder to turn his or her creativity back on to think of a solution to the problem. Once again balance is crucial, this time between mechanical processing and creative problem solving.

Second, even errors that are the result of TABARI's structure are not necessarily "forever problems." If a regular structure causes the same problem over and over, an alteration of the program itself might be able to fix it. For this reason TABARI includes the "Problem" feature under the "Options" menu. Use it! Selecting this feature allows coders to type a description of whatever difficulty they are encountering. This description is then saved to a separate word-processing file outside of the program, along with the current lead and the parsing of that lead. These problem files can then be forwarded to a project leader. Over time, an accumulation of various problem files can suggest potential, which then can be incorporated into TABARI's source code itself—though this obviously requires a good programmer. Accumulated experience and familiarity with common problems was what allowed TABARI to incorporate important grammatical features such the nouns and adjectives dictionaries that were missing in KEDS. Outside the level of programming changes, problem files can also be used to highlight difficulties with the coding scheme or simply as a convenient way of recording questions about how to code certain words or types of events.

F.2.4 When the computer throws up its hands in despair.

The three types of matches are common, and the coder will do much work with them. However, sometimes TABARI does not even reach the point of trying to code the sentence—it just gives some lame excuse about why the lead is "just too hard" and "it makes my chip hurt" and begs to be allowed to skip on to the next one. In these cases where no match is generated, the coder has to do triage:

1. Assess the type of problem from the error code generated
2. Determine whether this type of problem is fixable
3. Fix it or move on

There are several different reasons TABARI might refuse to code a lead. Some are valuable. Discard codes, for example, are words or phrases that are entered by the coder into the actor

dictionary to trigger the immediate, perfunctory dismissal of any lead that contains them. This allows the efficient discarding of leads about sporting events and presidential debates, as well as more idiosyncratic events like the spilling of massive amounts of cyanide into the Tisza River in Romania that killed thousands of fish and contaminated downstream into Hungary in early 2000. Other examples of useful non-codes are leads that do not contain a verb, or leads that do not contain both a source and a target.

Other times, though, a non-code is not the best outcome. Sometimes the coder can fix this, most commonly in the "too many verbs" case. TABARI is set up to ignore any lead that has six or more individual words with the "verb" tag. Presumably, any lead containing six verbs is too complicated for the program to code correctly, so it refrains from even trying. However, TABARI often incorrectly labels nonverbs as verbs. If a coder can "deverb" the false verbs in a lead, he or she can bring the lead below the verb limit and make it codable. This has the additional benefit of making future leads that contain the same false verbs codable as well. The cumulative effect of deverbing one or two words in a lead over hundreds of leads can be enormous, allowing the coding of many leads that would have otherwise been lost to the verb limit.

False verbs often arise from the stemming feature of the program. For example, if the verb "sold" is in the verb dictionary, the words "solder" and "soldiers" will be identified as forms of that verb. Even ignoring the verb limit, this is problematic because these words will trigger the recognition of verb patterns that are not accurate, leading to miscodes. While a little caution in the creation of new verb entries can help prevent such problems (will "sold_" work just as well as "sold"?), some stemming is necessary for the recognition of ed, -s, and ing forms, and occasional unforeseeable problems arise as well. Basically, some false verbs cannot be systematically prevented—desperate measures are required. A classic example of problematic stemming is the AFP lead

> Sharon's apparent turnaround came after Israel sent its tanks and helicopters into the Gaza Strip and the West Bank, resulting in heavy clashes with gunmen and scenes of carnage, and ahead of new missions to the region by a US Middle East peace envoy and Vice President Dick Cheney.

A lead of this complexity is exactly what the verb limit is designed to throw out. It does not even report a "hard" event that can be coded, just an "apparent turnaround." However, there are not enough verbs to trigger the limit, so the lead is coded

> 020308 ISR PALGZ 014 (ACCOMODATE, CEASEFIRE) 020308 ISR PALWB 014 (ACCOMODATE, CEASEFIRE)

A ceasefire! It's certainly good that we have TABARI to interpret the news for us.

Closer analysis reveals the problem.

> Sharon's apparent turnaround came after Israel sent its tanks and helicopters into the Gaza Strip and the West Bank, resulting in heavy clashes with gunmen and scenes of carnage, and ahead of new missions to the region by a US Middle East peace envoy and Vice President Dick Cheney.

The phrase "* IN GUN" is entered in the verb dictionary under the verb TURN, and turning in weapons is a form of accommodation. Thus "turnaround" is recognized as the verb, "into" as the preposition "in," and "gunmen" as the word "gun."

The error makes perfect sense, and it is easily fixable through the desperate measures of the noun and adjective dictionaries. Entering TURNAROUND into the noun dictionary prevents the false verb from being triggered, and the result is that lead is not coded under TURN[13]. But there is no way that anyone could have predicted such a misapplication of the verb phrase when TURN IN GUN was entered in the verb dictionary. It was a pretty good entry[14].

De-verbing, then, is a crucial task of the coder. Watching for English's random words like "turnaround" is the easy way of eliminating false verbs, though. Life gets more complicated with systematic sources of false verbs, like the ending "ing." Here again, English's use of identical words in multiple capacities creates quandaries. Often, -ing forms of verbs are used as nouns and are irrelevant to the coding of a lead: "Egypt's president discussed the fighting in the West Bank with Lebanon's prime minister." TABARI will code "fighting" as a stem off the verb FIGHT, even though it does not act as a verb in the sentence. This is not a problem in this particular lead. However, it becomes problematic if the lead is longer:

```
Egypt's president DISCUSSED the FIGHTING in the West Bank with Lebanon's
prime minister when he LANDED in Beirut after he VISITED Turkey to SIGN
a cooperation accord regarding freight SHIPPED though Suez to Istanbul.
```

Now the sentence has six words recognized as verbs; it will not be coded. Entering "fighting" into the noun dictionary will prevent it from being recognized as a verb and allow the lead to generate the events

```
EGY LEB 020 (Consult) EGYPT DISCUSSED WITH LEBANON'S LEB EGY
020 (Consult) EGYPT DISCUSSED WITH LEBANON'S
```

On the surface it would appear that all words ending in *-ing* could be coded as nouns. However, this would cause other problems. Often, it is useful to have ing forms recognized as verbs; the classic example of this is "meeting." Leads commonly take the form "Egypt's president is meeting with Lebanon's prime minister today." Here there is no other verb to

[13] It will probably end up being coded as ISRAEL SENT TANKS INTO GAZA_STRIP AND THE_WEST_BANK. This is not correct, strictly speaking, because the lead is reporting an "apparent turnaround" instead of an attack. On the other hand, the attack did occur at some point, so coding an attack is probably more accurate than what was coded at first. One could argue that Sharon's "apparent turnaround" from the use of military force must have been a form of accommodation, in which case the lead had the right code for the wrong reason in the first case. Even then, though, it would be best to make the noun dictionary addition. Erroneous codes that happen to be right are not systematic; it is preferable to have the better-developed dictionary and one semi-correct lead than to leave the dictionary unaltered—causing who knows what problems in future leads—and get lucky in this one case. And you thought coding looked easy at first glance...

[14] "Pretty good" because there is one problem with the phrase. The word IN should have been followed by an underscore: IN_. This would have prevented "into" from triggering it, though it would not have mattered in this particular lead because there is another "in" later on in the sentence. The bigger picture, though, is that prepositions, conjunctions, and any word that cannot be stemmed in English should be followed by a terminal underscore when entered in the verb dictionary. This prevents "intestine," "insoluble," and "intergalactic" from being recognized as "in." The most common (though certainly not the only) words this should be used for include "in," "on," "of," "with," "no," "up," and the ubiquitous "to."

grab on to other than "meeting." It turns out that while occasionally "meeting" causes trouble by being coded as a verb, more often it is beneficial to call it a verb, especially as meetings make up such a large proportion of the events in international diplomacy. Rather than being a false verb, then, "meeting" is a "useful (pseudo-)verb." The example given above with "fighting" is not the best, either, because the verb "fighting" might be useful as a trigger for the CAMEO code 190, the use of conventional military force. On the other hand, it might not. The question is whether more leads are coded correctly when "fighting" is left as a verb or when it is considered a noun.

The trick, of course, is to know which *-ing* words are false verbs and which are useful (pseudo-)verbs. This is a matter of experience and the writing style of the text being coded. All a coder can do is be sensitive to the trends in the text and accommodate them with verb and noun dictionary entries. Once again the issue is a balancing act where the coder is constantly seeking the approach that will maximize useful codes and minimize miscodes and rejected leads.

Creative phrasing can also reduce the number of false verbs in a sentence. For example, "arm," "force," "free," and "trade" are all words that might usefully be included in the verbs dictionary. However, entering "armed_forces" and "free_trade_zone" as nouns will prevent these sequences of words not used as verbs from being recognized as multiple verbs. In one fell swoop several problems have been eliminated.

F.3 The End

Specific approaches to specific scenarios are a useful starting point, but no finite list can ever address every situation the coder will confront. Hopefully these suggestions have given a sketch of the mindset and approach that might allow a coder to quickly and effectively defeat new problems as they arise. This mindset boils down to a few basic points:

Balance: Walk the line between opposite extremes.

Perspective: Always keep the big picture in mind. Don't give undue attention to single eccentric leads. Don't spend too much effort on internal political events within a single country if the project is aimed at international relations. Remember what the project wants to discover and concentrate on extracting it from amidst the extraneous information.

Balance and Perspective: When making dictionary changes, consider what effect they will have when applied over thousands of leads. Don't be too specific or too general in wording and codes. Don't make large changes without consideration of the consequences, both intended and unintended.

Creativity: Solve problems in flexible ways, keeping in mind the above considerations. Be willing to use one part of speech as another, if it will systematically work to solve a problem.

Happy coding!

F.4 Miscellaneous Thoughts

Here are some suggestions, in no apparent order:

- Working alone allows much higher productivity. This is especially noticeable when working in the evening, away from the assorted noises of the office environment. There are some pretty good views of Lawrence in the dark from the windows, too.
- Know the coding scheme. Spending time learning it allows for much faster coding and saves time in the long run.
- Don't try to code for too long at a time. No one can do it well. Two hours is about the limit for most people in one sitting. Even making it that long is sometimes a challenge.
- Parse, parse, parse. It's not quite magic, but it's close. And know what the parse output symbols mean. Parsing quickly reveals which words the program is reading and which are being ignored; how each word is being treated, and how many verbs there are. It doesn't yet highlight which words are being matched to a pattern in the dictionary, but maybe some day
- Use terminal underscores on prepositions, etc. If you don't want to force two words to be next to each other, use an underscore and then a space.
- Watch out for stemming and words with multiple meanings. These are the sources of a large portion of miscodes.
- "Meet" codes are almost always reciprocal. Don't forget the second half of 022:023, 020:020, 021:021, and 026:026. Also, "Agree": 060:060, etc.
- Discard codes are fun and profitable. Just don't use them lightly: make sure that the discard phrase is sufficiently specific that it will almost always be used in a sentence that should be discarded.
- Consistency is crucial when assigning event codes to phrases; this requires good communication between individual coders and between project directors and coders. This is the most important during the initial training phase.
- A cooler room temperature helps prevent coders from falling asleep during late summer afternoons on the 6th floor of Blake Hall.
- Having other projects to break up the coding is a good idea. Researching date restrictions, downloading new data, etc. give coders a chance to use different parts of their brains.