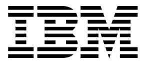


IBM Dictionary and Linguistic Tools

SC30-4039-00

Application Programming Interface Description

Version 2.7

Note

Before using this information and the product it supports, be sure to read the general information under Appendix A, “Notices” on page 251.

First Edition (May 2000)

This edition applies to Version 2.7 of the IBM Dictionary and Linguistic Tools.

IBM welcomes your comments on this publication. A form for readers’ comments is included at the back of this publication. You can also address comments to:

Department CGF
Design & Information Development
IBM Corporation
P.O. Box 12195
Research Triangle Park, NC 27709

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 2000. All rights reserved.**
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

About this manual xvii
Who should read this manual xvii
Conventions used in this manual xvii
Related Publications xvii

Part 1. IBM Dictionary and Linguistic Tools Description 1

Chapter 1. Introduction 3
Product Contents 4
Documentation 4
Linguistic Tools Version 2.7 Driver Package Description 4
Linguistic Tools Version 2.7 Driver 4
Linguistic Tools Version 2.7 Import Symbol File 5
Linguistic Tools Version 2.7 Testing Tool 5
Linguistic Tools Version 2.7 Header Files 5
Linguistic Tools Version 2.7 Sample Data 6
Linguistic Tools Version 2.7 Language Dictionaries 7
Platforms Supported 9
Languages Supported 10
Functional Categories 11

Chapter 2. Base Functions 13
Initializing 13
Terminating 13
Activating Dictionaries 13
Deactivating Dictionaries 14
Utilities 14
General Utilities 14
Token List Utilities 15

Chapter 3. Addenda (User) Dictionary Support Functions 17
Creating Addenda Dictionaries 18
Activating and Deactivating Addenda Dictionaries 19
Adding Words to Addenda Dictionaries 19
Addenda DBCS Considerations 23
Adding Morphological Information to Addenda Dictionaries 23
Removing Words from Addenda Dictionaries 25
Listing the Words in Addenda Dictionaries 25
Looking Up Words in Addenda Dictionaries 25
Saving Addenda Dictionaries to Disk 26
Addenda Portability Issues 26

Chapter 4. Text-Processing Functions 27
Spell Verify 27
Article Checking 27
Spell Aid 27
Fuzzy Spell Aid 27
Grade Level Analysis 28
Hyphenation 28

© Copyright IBM Corp. 2000

Ranked Hyphenation 28
Dehyphenation 29
Synonym Aid 29
Morphological Identification 30
Generate Inflected Forms 31
Inflect Word from Model 32
Simple Tokenization 33
Text Segment Identification 33
Sentence End Conditions 33
Abbreviation Processing 35
Lexical Analysis 37
Extract Keywords/Extract Query Terms 41
Processing Options 41
Keyword Order Number 42
Compound Word Processing 42
Hard Hyphen Forms 43
Word Normalization 44
Single-Word Data Element Creation 45
Compound Word Processing 45
Language-Specific Processing 47
Germanic Language Support 47
DBCS Support 47
Supported Functions by Language 48
System Dictionary Content by Language 55

Chapter 5. System Architecture 59
Linguistic Tools Main Entry Point 59
Linguistic Tools Initialization and Termination 60
Linguistic Tools Input 60
The Linguistic Tools Control Block 60
Service Area 69
Input Data Element List 69
Reply Area 76
Linguistic Tools Output 77
The Linguistic Tools Control Block 77
Service Area 77
Data Element List 77
Reply Area 77
Code Pages 78
Code Page Restrictions 78
Code Pages Supported 78
Definition of a Word in SBCS Text 80
Repeatable Punctuation 81
Sample Words 81
Definition of a Word in DBCS Text 81
Case Matching between Input Words and Dictionary Words 82
Reply Area 86
Continue Reply Capability 86
Reply Area Size 86
Tokens and Token Lists 89
The Token 89
User Data Tokens 91
Token List Utilities 91
Properties 95

DBCS Considerations . . . 96
Dictionaries . . . 100
Dictionary Use by Function . . . 100
Operating System-Specific Information . . . 103
Execution Environments . . . 103
Deliverables . . . 103
Software Customization . . . 104
Program Build Information . . . 104
Reentrancy of the Linguistic Tools . . . 106
File Names . . . 106
Dictionary Search Path . . . 108
Error Handling . . . 109
Memory Management . . . 109

Part 2. IBM Dictionary and Linguistic Tools Functions . . . 111

Chapter 6. Base Functions . . . 113
Initialize . . . 113
Input . . . 113
Output . . . 113
Return Codes . . . 113
Application Programming Guidelines . . . 113
Terminate . . . 114
Input . . . 114
Output . . . 114
Return Codes . . . 114
Activate Dictionary . . . 115
Input . . . 115
Output . . . 115
Return Codes . . . 117
Application Programming Guidelines . . . 117
Deactivate Dictionary . . . 118
Input . . . 118
Output . . . 118
Return Codes . . . 118
Application Programming Guidelines . . . 119

Chapter 7. Addenda (User) Dictionary Support Functions . . . 121
Create Addenda Dictionary . . . 121
Input . . . 121
Output . . . 121
Return Codes . . . 122
Application Programming Guidelines . . . 123
Add Word to Addenda . . . 124
Input . . . 124
Output . . . 125
Return Codes . . . 126
Application Programming Guidelines . . . 126
Add Morphology to Addenda . . . 128
Input . . . 128
Output . . . 128
Return Codes . . . 128
Application Programming Guidelines . . . 129

Remove Word from Addenda . . . 130
Input . . . 130
Output . . . 130
Return Codes . . . 130
Application Programming Guidelines . . . 131
List Addenda Words . . . 132
Input . . . 132
Output . . . 132
Return Codes . . . 132
Application Programming Guidelines . . . 133
Look Up Word in Addenda . . . 134
Input . . . 134
Output . . . 134
Return Codes . . . 135
Application Programming Guidelines . . . 135
Save Addenda . . . 137
Input . . . 137
Output . . . 137
Return Codes . . . 137
Application Programming Guidelines . . . 138

Chapter 8. Text-Processing Functions for Spelling Support . . . 139
Spell Verify . . . 139
Input . . . 139
Output . . . 139
Return Codes . . . 140
Application Programming Guidelines . . . 141
Article Checking . . . 141
Spell Aid . . . 142
Input . . . 142
Output . . . 142
Return Codes . . . 143
Application Programming Guidelines . . . 143
Fuzzy Spell Aid . . . 144
Input . . . 144
Output . . . 144
Return Codes . . . 145
Application Programming Guidelines . . . 146
Grade Level Analysis . . . 147
Input . . . 147
Output . . . 147
Return Codes . . . 148
Application Programming Guidelines . . . 149

Chapter 9. Text-Processing Functions for Hyphenation . . . 151
Hyphenation . . . 151
Input . . . 151
Output . . . 151
Return Codes . . . 152
Application Programming Guidelines . . . 152
Dehyphenation . . . 154
Input . . . 154
Output . . . 154
Return Codes . . . 154

Application Programming Guidelines 155

Chapter 10. Text-Processing Fuctions for Thesaurus 157
Synonym Aid 157
Input 157
Output 158
Return Codes 159
Application Programming Guidelines 160
Code Examples 161
Sample Output 164

Chapter 11. Text-Processing Functions for Morphology 167
Morphological Identification 167
Input 167
Output 168
Return Codes 169
Application Programming Guidelines 170
Generate Inflected Forms 171
Input 171
Output 172
Return Codes 173
Application Programming Guidelines 173
Inflect Word from Model 174
Input 174
Output 174
Return Codes 175
Application Programming Guidelines 175

Chapter 12. Text-Processing Functions for Text Analysis 177
Simple Tokenization 177
Input 177
Output 177
Return Codes 178
Application Programming Guidelines 178
Text Segment Identification 179
Input 179
Output 179
Return Codes 180
Application Programming Guidelines 180
Lexical Analysis 181
Input 181
Output 181
Return Codes 182
Application Programming Guidelines 182

Chapter 13. Text-Processing Functions for Text Extraction 185
Extract Keywords 185
Input 185
Output 186
Return Codes 189
Language-Specific Processing 190
Application Programming Guidelines 190
Example 191
Extract Query Terms 194

Input . . . 194
Output . . . 195
Return Codes . . . 198
Language-Specific Processing . . . 199
Application Programming Guidelines . . . 199
Output example . . . 199

Chapter 14. Text-Processing Functions for Other . . . 205
Single-Word Data Element Creation . . . 205
Input . . . 205
Output . . . 205
Return Codes . . . 206
Application Programming Guidelines . . . 206

Chapter 15. Text-Processing Functions for Language-Specific Support 207
Compound Word Component Isolation . . . 207
Input . . . 207
Output . . . 207
Return Codes . . . 208
Application Programming Guidelines . . . 209

Chapter 16. General Utility Functions . . . 211
NlpFindDicts . . . 211
Input . . . 211
Output . . . 211
Return codes . . . 212
Application Programming Guidelines . . . 213
NlpIsDelimiter . . . 215
Input . . . 215
Output . . . 215
Return Codes . . . 215
Application Programming Guidelines . . . 216
NlpMixedMode . . . 217
Input . . . 217
Output . . . 217
Return Codes . . . 217
Application Programming Guidelines . . . 217
NlpQryDelimTable . . . 219
Input . . . 219
Output . . . 219
Return Codes . . . 219
Application Programming Guidelines . . . 219
NlpQrySearchPath . . . 220
Input . . . 220
Output . . . 220
Return Codes . . . 220
Application Programming Guidelines . . . 220
NlpRegisterCodePage . . . 221
Input . . . 221
Output . . . 221
Return Codes . . . 221
Application Programming Guidelines . . . 222
Example . . . 222
NlpSetDelimTable . . . 223

Input . . . 223
Output . . . 223
Return Codes . . . 223
Application Programming Guidelines . . . 223
Example . . . 225
NlpSetDictDef . . . 226
Input . . . 226
Output . . . 226
Return Codes . . . 226
Application Programming Guidelines . . . 227
NlpSetSearchPath . . . 228
Input . . . 228
Output . . . 228
Return Codes . . . 228
Application Programming Guidelines . . . 228

Chapter 17. Token List Utility Functions . . . 229
NlpGetBeginType . . . 229
Input . . . 229
Output . . . 229
Return Codes . . . 229
Application Programming Guidelines . . . 230
NlpGetComponent . . . 231
Input . . . 231
Output . . . 231
Return Codes . . . 231
Application Programming Guidelines . . . 231
NlpGetContent . . . 232
Input . . . 232
Output . . . 232
Return Codes . . . 232
Application Programming Guidelines . . . 232
NlpGetFirstProperty . . . 234
Input . . . 234
Output . . . 234
Return Codes . . . 234
Application Programming Guidelines . . . 234
NlpGetNextProperty . . . 236
Input . . . 236
Output . . . 236
Return Codes . . . 236
Application Programming Guidelines . . . 236
NlpGetNextToken . . . 238
Input . . . 238
Output . . . 238
Return Codes . . . 238
NlpGetNumComponent . . . 239
Input . . . 239
Output . . . 239
Return Codes . . . 239
Application Programming Guidelines . . . 239
NlpGetNumTrail . . . 240
Input . . . 240
Output . . . 240

Return Codes 240
Application Programming Guidelines 240
NlpGetPrevToken 241
Input 241
Output 241
Return Codes 241
NlpGetPval 242
Input 242
Output 242
Return Codes 242
Application Programming Guidelines 242
NlpGetString 243
Input 243
Output 243
Return Codes 243
Application Programming Guidelines 244
NlpGetStringAddr 245
Input 245
Output 245
Return Codes 245
NlpGetStringFlag 246
Input 246
Output 246
Return Codes 246
Application Programming Guidelines 246
NlpGetTokenLen 247
Input 247
Output 247
Return Codes 247
Application Programming Guidelines 248
NlpGetTokenType 249
Input 249
Output 249
Return Codes 249
Application Programming Guidelines 249

Appendix A. Notices 251
Trademarks 252

Appendix B. Internal Character Codes 253
Constant Codes 253
Greek-Specific Codes 257
Russian-Specific Codes 259
Latin II Specific Codes 261
Turkish-Specific Codes 263
Arabic-Specific Codes 264
Hebrew-Specific Codes 266
Thai-Specific Codes 267
Collating Sequences for Linguistic Tools Internal Code Pages 269
Greek 269
Russian 271
Turkish 272
Icelandic: 273
Swedish and Finnish 274

Latin II languages (Polish, Hungarian, Czech). 275
All Other Languages 276

Appendix C. Return Codes 277

Appendix D. Language-Specific Processing 281
Arabic Language-Specific Processing 281
Morphological Identification 281
Catalan Language-Specific Processing 282
Spell Verify 282
Hyphenation/Dehyphenation 282
Lexical Analysis for Catalan 283
Word Normalization 283
Chinese Language-Specific Processing 284
Linguistic Overview 284
Supported Functions 284
Simple Tokenization for Chinese 285
Lexical Analysis for Chinese 285
Addenda Dictionary Support 285
Morphological Identification for Chinese 285
Keyword and Query Term Extraction for Chinese 286
Chinese Double-Byte Parsing Codes (DBPCs) 287
Danish Language-Specific Processing 290
Spell Verify 290
Lexical Analysis for Danish 290
Word Normalization 290
Dutch Language-Specific Processing 291
Spell Verify 291
Hyphenation 292
Lexical Analysis for Dutch 293
Word Normalization 293
English (US, UK, and Australian) Language-Specific Processing 294
Spell Aid 294
Lexical Analysis for English 294
Finnish Language-Specific Processing 305
Hyphenation 305
Lexical Analysis for Finnish 305
Word Normalization 305
French (National and Canadian) Language-Specific Processing 306
Spell Verify 306
Lexical Analysis for French 307
Word Normalization 308
National German Language-Specific Processing 309
Spell Verify 309
Spell Aid 310
Hyphenation/Dehyphenation 310
Lexical Analysis for German 311
Word Normalization 311
Swiss German Language-Specific Processing 312
Lexical Analysis in Swiss German 312
Word Normalization 312
Greek Language-Specific Processing 313
Greek Character Set 313
Spell Verify 313

Word Normalization . . . 314
Hungarian Language-Specific Processing . . . 315
Hyphenation/Dehyphenation . . . 315
Lexical Analysis for Hungarian . . . 315
Word Normalization . . . 315
Icelandic Language-Specific Processing . . . 316
Spell Verify . . . 316
Lexical Analysis for Icelandic . . . 316
Word Normalization . . . 316
Italian Language-Specific Processing . . . 317
Spell Verify . . . 317
Lexical Analysis for Italian . . . 319
Word Normalization . . . 320
Japanese Language-Specific Processing . . . 321
Linguistic Overview . . . 321
Supported Functions . . . 322
Simple Tokenization for Japanese . . . 322
Lexical Analysis for Japanese . . . 323
Synonym Aid Support . . . 323
Keyword and Query Term Extraction for Japanese . . . 324
Japanese Part of Speech (POS) Codes . . . 324
Addenda Dictionary Support . . . 327
Korean Language-Specific Processing . . . 328
Morphological Identification for Korean . . . 329
Korean Addenda Dictionary Support . . . 330
Keyword and Query Term Extraction for Korean . . . 330
Korean Double-Byte Parsing Codes (DBPCs) . . . 331
Norwegian Language-Specific Processing . . . 335
Spell Verify . . . 335
Hyphenation/Dehyphenation . . . 335
Lexical Analysis for Norwegian . . . 335
Word Normalization . . . 336
Portuguese Language-Specific Processing . . . 337
Enclitic Processing . . . 337
Hyphenation . . . 339
Lexical Analysis for Portuguese . . . 340
Morphological Identification . . . 340
Generate Inflected Forms . . . 341
Synonym Aid . . . 341
Word Normalization . . . 341
Spanish Language-Specific Processing . . . 342
Lexical Analysis for Spanish . . . 342
Word Normalization . . . 342
Russian Language-Specific Processing . . . 343
Spell Verify . . . 343
Swedish Language-Specific Processing . . . 344
Spell Verify . . . 344
Hyphenation/Dehyphenation . . . 344
Lexical Analysis for Swedish . . . 344
Word Normalization . . . 345

Appendix E. Morphology Grammar Masks . . . 347
Catalan . . . 347
Czech . . . 347

Danish . . . 348
Dutch . . . 349
English . . . 349
Finnish . . . 349
French (National) . . . 350
Canadian French . . . 351
German . . . 351
Greek . . . 352
Italian . . . 352
Norwegian, Bokmål . . . 353
Norwegian, Nynorsk . . . 353
Portuguese, National . . . 354
Brazilian Portuguese . . . 354
Russian . . . 355
Spanish . . . 356
Swedish . . . 356
Turkish . . . 357

Glossary . . . 359

Index . . . 367

Figures

1. Linguistic Tools Input/Output Structure 59
2. Sample Data Element Input 91
3. Sample code to traverse token list generated by Lexical Analysis 93
4. Sample code to display all text associated with a token list 94
5. Sample code to display all text associated with a token list 94
6. Sample code to traverse property list 96
7. Sample code to find a specific property 96
8. Sample Token List with DBCS Text 97
9. Sample Token List for Host DBCS Text 98
10. LX_SUBTERM example for “softwaredevelopment” 183
11. LX_SUBTERM example for “therealways” 184
12. Sample code to change treatment of a hyphen as a word delimiter . . 225
13. Greek Collating Sequence for Linguistic Tools Internal Code Page . . . 269
14. Russian Collating Sequence for Linguistic Tools Internal Code Page . 271
15. Turkish Collating Sequence for Linguistic Tools Internal Code Page . . 272
16. Icelandic Collating Sequence for Linguistic Tools Internal Code Page . 273
17. Swedish and Finnish Collating Sequence for Linguistic Tools Internal Code Page. 274
18. Latin II Collating Sequence for Linguistic Tools Internal Code Page. . . 275
19. Default (Other Languages) Collating Sequence for Linguistic Tools Internal Code Page. 276
20. Catalan Prefixes 282
21. French prefixes 306
22. French exception trailing words 307
23. Italian Prefixes 318

Tables

1. Language Dictionaries 7
2. Text Segment Terminating Punctuation Characters 33
3. Closing Punctuation Characters 34
4. Lexical Analysis function by language 38
5. Text-Processing Functions by Language—Spelling Support 50
6. Text-Processing Functions by Language—Hyphenation and Thesaurus . 52
7. Text-Processing Functions by Language—Morphology and Language-Specific Support 54
8. Linguistic Tools Dictionary Content 57
9. Linguistic Tools Control Block Fields (General) 62
10. Linguistic Tools Control Block Fields (Addenda Dictionary Support) . . . 65
11. Linguistic Tools Control Block Fields (Spelling Support) 66
12. Linguistic Tools Control Block Fields (Hyphenation) 67
13. Linguistic Tools Control Block Fields (Thesaurus) 67
14. Linguistic Tools Control Block Fields (Morphology) 68
15. Linguistic Tools Control Block Fields (Text Extraction) 68
16. Data Element Structure 75
17. Code Pages Supported 79
18. Linguistic Tools Delimiter Characters 80

© Copyright IBM Corp. 2000

19. Linguistic Tools Default Case Matching 82
20. Linguistic Tools Spell Aid Case Matching 83
21. Recommended Reply Area Size by Function 87
22. Linguistic Tools Token List Utilities 91
23. Linguistic Tools Internal Codes: Constant Codes 253
24. Linguistic Tools Internal Codes: Workstation and Host Greek Codes . . 257
25. Linguistic Tools Internal Codes: Workstation and Host Russian Codes 259
26. Linguistic Tools Internal Codes: Workstation and Host Latin II Codes . 261
27. Linguistic Tools Internal Codes: Workstation and Host Turkish Codes . 263
28. Linguistic Tools Internal Codes: Workstation and Host Arabic Codes . 264
29. Linguistic Tools Internal Codes: Workstation and Host Hebrew Codes . 266
30. Linguistic Tools Internal Codes: Workstation and Host Hebrew Codes . 267
31. Chinese Double-Byte Parse Codes (DBPCs) 287
32. Japanese Part of Speech Codes (JPOS, POS, DBPC/Macro POS). . . 325
33. Korean Double-Byte Parse Codes (DBPCs) arranged by Number . . . 331
34. Korean Double-Byte Parse Codes (DBPCs) arranged by PCODE . . . 332
35. Defined Constants for Korean Double-Byte Parse Codes (DBPCs) . . . 334
36. Additional Korean Double-Byte Parse Codes (DBPCs) Returned by MID 334
37. Portuguese Reflexive Pronoun (RP) 337
38. Portuguese Personal Pronouns (PP), Accusative and Dative Case . . . 337
39. Portuguese Impersonal Pronouns (IP), Accusative Case 337
40. Portuguese Indirect Object Pronouns (IO), Dative Case 337
41. Portuguese PP/IP Contractions (PPIPC), Dative + Accusative Case . . 337
42. Portuguese IO/IP contractions (IOIPC), Dative + Accusative Case . . . 338
43. Sample Application of Portuguese Enclitic Transformation Rules (a) . . 339
44. Sample Application of Portuguese Enclitic Transformation Rules (b) . . 339

About this manual

This manual describes the application programming interface for the IBM Dictionary and Linguistic Tools.

Note: The short name for the IBM Dictionary and Linguistic Tools is Linguistic Tools.

Who should read this manual

This manual is intended for software developers who work on products that use or plan to use Linguistic Tools.

Conventions used in this manual

The following conventions are used in this manual:

Convention	Usage	Example
Italics	Variables shown in the text, not in the code samples.	*SearchPath*
Bold	Commands (functions) shown in the text, not in the code samples. **Note:** Programmed commands are not case sensitive.	**NlpGetBeginType**
Monospaced text	Code samples	`/DLX_WIN`
All capitals	Keywords	LX_UCHAR
Quotation marks	Words out of context, as well as quotations	Neither “a” nor “an” ...

Related Publications

A companion manual that describes how to use the testing tool for the Linguistic Tools.

IBM Dictionary and Linguistic Tools User’s Guide for the Testing Tool, SC30-4040

© Copyright IBM Corp. 2000

Part 1. IBM Dictionary and Linguistic Tools Description

© Copyright IBM Corp. 2000

Chapter 1. Introduction

IBM Dictionary and Linguistic Tools Version 2.7 is an application that provides dictionary support and text-processing functions for multiple languages on a number of different platforms.

Note: The short name for IBM Dictionary and Linguistic Tools is Linguistic Tools.

Linguistic Tools includes both dictionary-access and text-processing functions. The dictionary-based functions provide spell checking (both verification and aid), hyphenation and dehyphenation, synonym aid, and morphological information such as the base forms or inflections of a word. U.S. English article checking and grade-level information are also available.

Some Linguistic Tools functions provide special processing for compound words in the Germanic languages. An application may also invoke this processing directly in order to obtain the component parts of a Germanic compound word.

The text-processing functions include simple tokenization/word isolation, text segment identification, lexical analysis, and single-word data element creation. For languages that use blanks as word delimiters, simple tokenization will return each word as a separate token. For the languages that do not use blanks to separate words (Chinese, Japanese, and Thai), simple tokenization will isolate words only in the portion of input text that has word delimiters; lexical analysis must be invoked to identify individual words within the text. Text segment identification uses punctuation to break up the text into units (generally sentences, although there are some exceptions). In languages other than Chinese, Japanese, and Thai, lexical analysis handles complex forms such as hyphenated forms, words with slashes, numbers containing commas or decimal points, and the like. Single-word data element creation takes block-format input text and creates a single-word data element list that can be used as input to other Linguistic Tools functions.

The keyword and query term extraction functions are designed to be used for text indexing and information retrieval. They isolate the content words in the input text, rejecting words that are in the stopword list and words that do not have the part of speech requested by the application. Base forms, normalized forms, and phonetic keys may be returned for each keyword/query term. In the Germanic languages, the components of a compound keyword or query term may be returned so that they, too, may be used as keywords/query terms. For query terms, Linguistic Tools can also return synonyms and inflected forms.

These functions have been designed to be modular in order to minimize code and data overhead. This and other changes from previous designs should result in improved performance. However, because Linguistic Tools is intended to operate under many different operating systems, speed and memory limitations must be taken into consideration.

The application will be able to make decisions at run-time concerning performance and memory usage. See “Operating System-Specific Information” on page 103 for additional information.

© Copyright IBM Corp. 2000

Product Contents

The following section defines the contents of the Linguistic Tools product. For the AS/400®, you should be able to locate all of these items after the installation.

Note: Documentation is not installed on the AS/400. Documentation can be found on the PDF documentation CD-ROM and on the AS/400 Web site at URL:

`http://www.as400.ibm.com`

Documentation

API Description

Title: *IBM Dictionary and Linguistic Tools Application Programming Interface Description*
Document number: SC30-4039-00
File name: GAO0MST.PDF

Tester User's Guide

Title: *IBM Dictionary and Linguistic Tools User's Guide for the Testing Tool*
Document number: SC30-4040-00
File name: GAP0MST.PDF

Linguistic Tools Version 2.7 Driver Package Description

File name: drvpkg.txt

This file also lists the compile and link options that are needed. For the AS/400, this file is located in the following directory:

/QIBM/ProdData/Dictionary

Linguistic Tools Version 2.7 Driver

File name: EFLNL27_

The underscore (_) is a character indicating the platform.

Platform	Driver name
32-bit Microsoft® Windows® (Windows NT®, Windows 95®, Windows 98®)	eflnl27w.dll
OS/2® Warp	eflnl272.dll
AIX® (IBM UNIX™)	libeflnl27x.so
HP-UX (Hewlett-Packard UNIX)	libeflnl27h.sl
Sun™ Solaris (Sun Microsystems UNIX)	libeflnl27s.so
OS/390® OpenEdition (OS/390 UNIX System Services)	libeflnl273.dll
OS/400®	Library: QDICT Object: EFLNL274 Type: *SRVPGM
Mac OS	eflnl27a.lib

Linguistic Tools Version 2.7 Import Symbol File

Not all platforms require a file listing the symbols to import.

Platform	Import Symbol file
32-bit Microsoft Windows (Windows NT, Windows 95, Windows 98)	eflnl27w.lib
OS/2 Warp	eflnl272.lib
AIX (IBM UNIX)	efxlnlps.imp
HP-UX (Hewlett-Packard UNIX)	N/A
Sun Solaris (Sun Microsystems UNIX)	N/A
OS/390 OpenEdition (OS/390 UNIX System Services)	libeflnl273.x
OS/400	N/A
Mac OS	N/A

Linguistic Tools Version 2.7 Testing Tool

Note: This is executable software and is provided "as is".

File name: TSTNL27_

The underscore (_) is a character indicating the platform.

Platform	Executable name
32-bit Microsoft Windows (Windows NT, Windows 95, Windows 98)	tstnl27w.exe
OS/2 Warp	tstnl272.exe
AIX (IBM UNIX)	tstnl27x
HP-UX (Hewlett-Packard UNIX)	tstnl27h
Sun Solaris (Sun Microsystems UNIX)	tstnl27s
OS/390 OpenEdition (OS/390 UNIX System Services)	tstnl273
OS/400	Library: QDICT Object: TSTNL274 Type: *PGM
Mac OS	tstnl27a

Linguistic Tools Version 2.7 Header Files

- For the AS/400:

  Library: QDICT
  Object: H
  Type: *FILE
  Members:
  EFZLNLPS
  EFZLEXTY
  EFZLEXCB
  EFZLEXRT
  EFZLNSRC

- For all other platforms:

  Members:
  efzlnlps.h

efzlexty.h
efzlexcb.h
efzlexrt.h
efzlnsrc.h

Linguistic Tools Version 2.7 Sample Data

Note: This data is provided “as is”.

Sample Test Case

- For the AS/400:

  /QIBM/ProdData/Dictionary/Samples/BASIC400.TSC

- For all other platforms:

  BASIC.TSC

Expected Output for Sample Test Case

- For the AS/400:

  Library: QDICT
  Object: TOU
  Type: *FILE
  Members:
  BASIC400

- For all other platforms:

  BASIC.TOU

Sample Code for Using the API

For the AS/400, these files are located in the following directory:

/QIBM/ProdData/Dictionary/Samples

The sample code consists of the following items:

- README.TXT—compilation instructions for the Sample Code
- nlpsamp.h
- sample.c
- mainsamp.c
- basesamp.c
- addsamp.c
- hyphsamp.c
- spelsamp.c
- synsamp.c
- textsamp.c

Linguistic Tools Version 2.7 Language Dictionaries

For the AS/400, these files are located in the following directory:

/QIBM/ProdData/Dictionary

Table 1 (Page 1 of 2). Language Dictionaries

Language	System Dictionary	Stopword Dictionary	Abbreviation Dictionary
Dutch (South Africa Afrikaan version)	afrikaan.dic	afrikaan.stb	
Arabic	arabic.dic	arabic.stb	
Catalan	catala.dic	catala.stb	
Chinese (Simplified, with error check/correct data)	clkchs.dic		
Chinese (Simplified, without error check/correct data)	eflchs.dic	eflchs.stb	
Chinese (Traditional, with error check/correct data)	clkcht.dic		
Chinese (Traditional, without error check/correct data)	eflcht.dic	eflcht.stb	
Czech	czech.dic		
Danish	dansk.dic	dansk.stb	dansk.abb
Dutch (Government Restrictive spelling version)	nederlnd.dic	nederlnd.stb	nederlnd.abb
Dutch (Permissive spelling version)	nedplus.dic		
English (Australian)	aus.dic	aus.stb	aus.abb
English (United Kingdom)	uk.dic	uk.stb	uk.abb
English (United States)	us.dic	us.stb	us.abb
U.S. English Computing Terms	us.cpt		
U.S. English Legal Terms	us.leg		
U.S. English Medical Terms	us.med		
Combined U.S. English dictionary and U.S. English Medical Terms dictionary	usmed.dic		
Finnish	suomi.dic	suomi.stb	
French (National)	francais.dic	francais.stb	francais.abb
French (Canadian)	canadien.dic	canadien.stb	canadien.abb
German (National pre-reform)	deutsch.dic	deutsch.stb	deutsch.abb
German (National reform)	deutsch2.dic		
German (DPA rules version of National reform)	deut2dpa.dic		
German (Swiss)	dschweiz.dic	dschweiz.stb	dschweiz.abb
Greek	hellas.dic	hellas.stb	hellas.abb
Hebrew	hebrew.dic	hebrew.stb	hebrew.abb
Hungarian	magyar.dic		
Icelandic	islensk.dic	islensk.stb	
Italian	italiano.dic	italiano.stb	italiano.abb
Japanese	efljpn.dic		
Korean	eflkor.dic		
Norwegian (Bokmål)	norbok.dic	norbok.stb	norbok.abb
Norwegian (Nynorsk)	nornyn.dic	nornyn.stb	

Table 1 (Page 2 of 2). Language Dictionaries

Language	System Dictionary	Stopword Dictionary	Abbreviation Dictionary
Polish	polska.dic		
Portuguese (National)	portugal.dic	portugal.stb	
Portuguese (Brazilian)	brasil.dic	brasil.stb	
Russian	russian.dic	russian.stb	russian.abb
Spanish	espana.dic	espana.stb	espana.abb
Swedish	svensk.dic	svensk.stb	svensk.abb
Thai	thai.dic		thai.abb
Turkish	turkiye.dic		

Platforms Supported

Linguistic Tools is available for the following operating systems:

- 32-bit Microsoft Windows (includes these OSs:)
  - Windows NT
  - Windows 95
  - Windows 98
- OS/2 Warp 4
- OS/400 Version 4 Release 5 Modification 0
- OS/390 OpenEdition (OS/390 UNIX System Services)
- AIX v. 4.1 and higher (IBM UNIX)
- HP-UX v. 10.10 and higher (Hewlett-Packard UNIX)
- Sun Solaris v. 2.5 and higher (Sun Microsystems UNIX, also known as SunOS 5.5)
- MacOS v. 9.0 and higher

Languages Supported

Linguistic Tools provides support for the following languages:

- Afrikaans (South African version of Dutch)
- Arabic
- Catalan
- Chinese (Simplified and Traditional)
- Czech
- Danish
- Dutch (both government restrictive and permissive spellings)
- English (Australian, U.K. and U.S.)
- Finnish
- French (National and Canadian)
- German (with the following sub-types)
  - National pre-reform
  - National reform
  - National reform conforming to the DPA rules
  - Swiss
- Greek
- Hebrew
- Hungarian
- Icelandic
- Italian
- Japanese
- Korean
- Norwegian (Bokmål and Nynorsk)
- Polish
- Portuguese (National and Brazilian)
- Russian
- Spanish
- Swedish
- Thai
- Turkish

Note: The Macintosh inplementation does not support the following languages:

- Japanese
- Chinese (both Simplified and Traditional)
- Korean
- Arabic
- Hebrew
- Thai

The extent of functional support varies from one language to another. See "Supported Functions by Language" on page 48 for details.

Functional Categories

The Linguistic Tools functions can be divided into categories such as base functions, user dictionary support functions, and various sets of text-processing functions.

Base Functions

- Initialize
- Terminate
- Activate dictionary
- Deactivate dictionary
- Utilities
  - General utilities (see “Utilities” on page 14)
  - Token List utilities (see “Token List Utilities” on page 91)

Addenda (User) Dictionary Support Functions

- Create addenda dictionary
- Add word to addenda (add word + data)
- Add morphology to addenda (add morphological information)
- Remove word from addenda
- List addenda words
- Look up word
- Save addenda

Text-Processing Functions

- Spelling support
  - Spell verify
  - Article checking (U.S. English only)
  - Spell aid
  - Fuzzy spell aid
  - Grade Level Analysis (U.S. English only)
- Hyphenation
  - Hyphenation
  - Dehyphenation
- Thesaurus
  - Synonym aid
- Morphology
  - Morphological identification
  - Generate inflected forms
  - Inflect word from model
- Text analysis
  - Simple tokenization
  - Text segment identification
  - Lexical analysis
- Text extraction
  - Extract keywords
  - Extract query terms
- Other
  - Single-word data element creation
- Language-specific support
  - Compound word component isolation (for Germanic languages only)

Chapter 2. Base Functions

The set of base functions is required in order to use all other Linguistic Tools functions. All functions require Linguistic Tools to be initialized, and every application should terminate Linguistic Tools when all of its functions are completed. Any application using a function that requires information from a dictionary must first activate the dictionary that it will use and deactivate the dictionary when it is no longer needed. Various Linguistic Tools utilities assist the application in obtaining output and tuning specific Linguistic Tools settings.

Initializing

The **Initialize** function initializes Linguistic Tools. The **Initialize** function must be called before an application can invoke any other linguistic function. There is also a **Terminate** function, which deallocates memory allocated by Linguistic Tools and deactivates any active dictionaries.

The chief task of the **Initialize** function is to set up the Service Area, an area where Linguistic Tools stores data that it must remember between calls. The function returns a Service Area Handle that refers to this area. The application must use this service handle when it calls any other Linguistic Tools function.

An application may maintain more than one active session with Linguistic Tools by making more than one call to the initialize function. Since every call to **Initialize** returns a new Service Area Handle, an application may maintain separate sessions by using separate Service Area Handles.

Terminating

The application is also required to terminate the Linguistic Tools session by using the **Terminate** function. This function ensures that Linguistic Tools releases all system resources under its control. If the application has opened more than one Linguistic Tools session, each must be terminated separately by passing the handle for the Service Area associated with it.

Note: Note that the terminate function does not save addenda dictionaries to disk before deactivating them. Any changes made to addenda dictionaries must be saved by calling **Save Addenda** prior to calling **Terminate**.

Activating Dictionaries

All dictionaries to be used by Linguistic Tools must be opened by using the **Activate Dictionary** function. A separate function call is required to activate each dictionary.

When a dictionary is activated, Linguistic Tools will return a dictionary handle which must be used to specify that dictionary to other Linguistic Tools functions. Linguistic Tools will maintain an active dictionary list, a table containing information about each dictionary that has been activated, including its name, type, the handle that has been assigned to it, and how to locate it. The application, however, must pass in a list of dictionary handles with each function call. This specifies which dictionaries may be used by this function; if a dictionary's handle is not included in

© Copyright IBM Corp. 2000

this list, that dictionary will not be available to the function. The order in which dictionaries will be accessed by the function is determined by their order in the input dictionary list. To exclude a dictionary from being accessed, the application simply omits the handle for that dictionary from the input dictionary list. (See "Dictionary Use by Function" on page 100 for information about how each function uses the input dictionary list.)

Deactivate Dictionary is the corresponding function which drops the dictionary, making its contents inaccessible to the other Linguistic Tools functions. The application may request that a dictionary be activated or deactivated at any time.

Deactivating Dictionaries

Deactivate Dictionary requests Linguistic Tools to drop a specified dictionary handle from the active dictionary list, thus making it inaccessible to other Linguistic Tools functions. It frees any system resources that Linguistic Tools has associated with the dictionary and makes the dictionary handle available for use in a later activation. Every dictionary that is activated must also be deactivated by calling the **Deactivate Dictionary** function.

Note: Note that this function does not save addenda dictionaries to disk. In order to save changes made to an addenda dictionary during a Linguistic Tools session, the application must call **Save Addenda** before calling **Deactivate Dictionary**.

Utilities

The base functions also include a set of utility functions to assist an application in interfacing with Linguistic Tools. Like any other Linguistic Tools functions, these require a valid Service Area Handle; that is, Linguistic Tools must first be initialized by calling the **Initialize** function before calling these or any other Linguistic Tools functions.

General Utilities

NlpQrySearchPath() can be used by the application to find out where Linguistic Tools will look for dictionary files. "Dictionary Search Path" on page 108 explains what Linguistic Tools will use as the default search path if none is specified by the application.

NlpSetSearchPath() may be used to specify the locations in which Linguistic Tools should look for dictionary files.

NlpFindDicts() determines which dictionaries are present on the user's system and where they are located. This function requires that dictionary file names either follow the conventions described in "Linguistic Tools Default Dictionary Naming Conventions" on page 108 or use a set of defaults specified by the application through the **NlpSetDictDef()** function.

NlpSetDictDef() will permit the application to specify its own set of default file types or extensions for dictionary file names if it does not wish to use the Linguistic Tools defaults listed in "Linguistic Tools Default Dictionary Naming Conventions" on page 108. Note that neither the Linguistic Tools defaults nor the application-specified file types/extensions will be enforced by any Linguistic Tools

function other than **NlpFindDicts()**; however, that function will not be able to identify files as dictionaries if the naming conventions are not observed.

NlpIsDelimiter() is useful for identifying word boundaries within the input text. "Definition of a Word in SBCS Text" on page 80 explains how Linguistic Tools uses delimiter characters to determine word boundaries.

NlpQryDelimTable() may be used to view Linguistic Tools' default word delimiter table for SBCS text. "Definition of a Word in SBCS Text" on page 80 explains how Linguistic Tools uses delimiter characters to determine word boundaries.

NlpSetDelimTable() permits the application to change the way in which Linguistic Tools uses non-alphanumeric characters to determine word boundaries in SBCS text. "Definition of a Word in SBCS Text" on page 80 explains how delimiter characters are used.

NlpMixedMode() allows the caller to specify whether Linguistic Tools will allow an all uppercase input word to match a mixed-case dictionary word. This function affects the results of any function which relies on matching input words against the dictionary (Spell Verify, Synonym Aid, Morphological Identification, and so on). For details see "Case Matching between Input Words and Dictionary Words" on page 82 and "NlpMixedMode" on page 217.

NlpRegisterCodePage() permits the application to provide code page-related information for code pages not included in the Linguistic Tools default set specified in Table 17 on page 79.

Token List Utilities

Functions for retrieving information about tokens and token lists are summarized in "Token List Utilities" on page 91.

Chapter 3. Addenda (User) Dictionary Support Functions

System Dictionary coverage is usually adequate for general vocabulary in each language supported by Linguistic Tools. However, there are some words that may not appear in the System Dictionary, such as proper names, acronyms, and specialized technical terms. Addenda dictionary support allows the application to save these terms with associated information such as hyphenation points, morphology, and other (optional) data supplied by the application and/or the user.

Since dictionary priority is established by the order of dictionary handles in the input dictionary list, as mentioned above, an application may wish to place the addenda dictionary handle in the list before the handle of the System Dictionary of the same language. This would indicate to the Linguistic Tools that the information in the addenda supercedes the information in the System Dictionary. For example, the user program may have a document which refers to a trademarked product “Entrée” and may wish to differentiate this from the capitalized form of the word “entrée” in order to prevent the product name from being hyphenated. To do this, the user program adds “Entrée” with no hyphenation information to an addenda and places the addenda handle first in the dictionary list. From then on, the word “entrée” is hyphenated, but the word “Entrée” is not.

Addenda dictionaries are also used in situations where only a subset of the System Dictionary is desired. Two uses specifically supported by Linguistic Tools are for the specification of stopwords (words which should not be identified as keywords by the Extract Keywords function) and the specification of abbreviations (for use in Text Segment Identification). In these two specific cases, separate input dictionary lists are used for Stopword Addenda Dictionaries and for Abbreviation Addenda Dictionaries.

To use an addenda dictionary, the application must:

1. Activate the dictionary
   - If the addenda dictionary does not exist—that is, if the application/user is creating a brand-new addenda dictionary—call **Create Addenda Dictionary**, which will return a handle to use in specifying this dictionary to other Linguistic Tools functions.
   - If using an existing addenda dictionary, call **Activate Dictionary** to obtain a handle to use in specifying this dictionary to other Linguistic Tools functions.
2. Use the dictionary in the desired functions:
   - **Spell Verify**
   - **Spell Aid**
   - **Fuzzy Spell Aid**
   - **Hyphenate**
   - **Morphological Identification**
   - **Generate Inflected Forms**
   - **Synonym Aid**
   - **Lexical Analysis** for DBCS languages
   - **Add Word to Addenda**
   - **Add Morphology to Addenda**
   - **List Addenda Words**

© Copyright IBM Corp. 2000

- **Look Up Word**
- **Remove Word from Addenda**
- **Text Segment ID**
- **Extract Keywords**
- **Extract Query Terms**

3. Call **Save Addenda** to save any changes to disk.
4. When the dictionary is no longer needed, free all resources associated with it by calling **Deactivate Dictionary**.

Creating Addenda Dictionaries

To create an addenda dictionary, the application should call the **Create Addenda Dictionary** function. The newly-created addenda dictionary will be a binary format file containing a standard header. Words may be added to it by means of the **Add Word** and **Add Morphology** functions. The **Add Word** function allows additional information (synonyms, double-byte parse codes, and other user- and/or application-supplied data) to be stored with the added word. The **Add Morphology** function provides a means for adding morphological data for a word to the addenda dictionary. The **Save Addenda** function must be called to save the file to disk. This function includes an option to save the file in flat (text) format.

In order to create an editable addenda dictionary file, the application must first call **Create Addenda**, then save the newly-created dictionary through the save-flat-file option of the **Save Addenda** function. The result will be a flat file which can be treated like any other editable file. If no words have been added to the dictionary, the contents of this file will consist of a one-line header containing an identifier string

```
IBM v1.0 t<type>-<language>
```

where <type> is one of the following:

- "4" - regular addenda dictionary
- "5" - stopword addenda dictionary
- "10" - abbreviation addenda dictionary

and <language> is one of the language names below:

- "Afrikaans" (Afrikaan)
- "Arabic" (Arabic)
- "Australian" (Australian English)
- "Bokmål" (Norwegian "book language")
- "Català" (Catalan)
- "CHT" (Traditional Chinese)
- "CHS" (Simplified Chinese)
- "Czech" (Czech)
- "Dansk" (Danish)
- "Deutsch" (German)
- "English - U.K." (U.K. English)
- "English - U.S." (U.S. English)
- "Español" (Spanish)
- "Français" (National French)
- "Français Canadien" (Canadian French)
- "HANKUK-O" (Korean)
- "Hebrew" (Hebrew)

- “Hellenikos” (Greek)
- “Íslensk” (Icelandic)
- “Italiano” (Italian)
- “Magyar” (Hungarian)
- “Nederlands” (Dutch—both government-restricted and permissive spellings)
- “NIHONGO” (Japanese)
- “Nynorsk” (Norwegian “new Norwegian”)
- “Polska” (Polish)
- “Português” (National Portuguese)
- “Português - Brasil” (Brazilian Portuguese)
- “Russian” (Russian)
- “Schweizer-Deutsch” (Swiss German)
- “Suomi” (Finnish)
- “Svensk” (Swedish)
- “Thai” (Thai)
- “Turkish” (Turkish)
- “No Language”

This header line should not be changed by the user. If the language name is removed from the file during a user edit session or if it does not match an entry in the above list, the dictionary will be considered to be language-less; it will be treated as if “No Language” had appeared on the first line. This means that no language-specific processing will be done for the words in the dictionary.

Activating and Deactivating Addenda Dictionaries

Addenda dictionaries are activated and deactivated by the same functions as System Dictionaries. The primary form of an addenda is a binary file that Linguistic Tools activates by copying into memory. The alternate form is a flat file that can be modified via an editor, transferred to another application, or used for archival purposes. When activating a flat file addenda, a binary addenda is always created, since this is the form of the addenda with which the tool will do its work.

For better performance, it is recommended that applications use only the binary form of the addenda. If the flat file addenda capability is required, it is recommended that the application save the addenda in both formats. Then when activating the addenda, the application should check the dates of both the binary and flat file addenda and activate the most recent. The application may choose to provide “import” and “export” options for the user so that the application only saves the flat file when exporting and erases the binary file when importing.

Note that **Deactivate Dictionary** does not save an addenda dictionary to disk. Changes not saved by means of the **Save Addenda** function will be lost when the dictionary is deactivated.

Adding Words to Addenda Dictionaries

The **Create Addenda** function will only place a header into the dictionary file. Words can then be added by using **Add Word to Addenda** to add one or more words with associated information or **Add Morphology** (if morphological information for the word is to be stored), or, if the file has been saved in flat (text) format, by editing the file directly. Words added through one of the function

interfaces may be any valid string of characters which satisfies the criteria stated in "Definition of a Word in SBCS Text" on page 80.

When words are added to a flat (text) file by editing, each word should appear on a line by itself. The words need not be in any particular order, as they will be ordered during activation. The data for the word must follow the word on the same line and/or the following lines. Each line may be up to 255 bytes in length.

If data is to be included, it must be tagged as follows:

```
word <SYN>synonym1, synonym2, synonym phrase
     <DBPC>code1, code2
     <MORPH>alias-term
     <ABBR-EOS>
     <DATA>user- or application-data
```

Linguistic Tools will use the tag to identify the type of data that follows. Each tag must be preceded by whitespace and must be in the above order, when present. The special <CONT> tag indicates that data has been split across lines and the current line is a continuation of the data at the end of the previous line. A <CONT> tag may occur only once on a line, and the tag may only be preceded on the line by whitespace. The word plus all its data may not exceed 64K bytes.

The following text is an example of what is allowed in an SBCS flat-file addenda dictionary:

```
brat_tice <SYN>bulwark, partition <MORPH>lattice
clep_sy_dra <SYN>water clock <DATA>from
             <CONT> ancient Greek "klepsudra"
em_prise <SYN>undertaking, adventure, enterprise
         <MORPH>enterprise
         <DATA>from Old French past-participle of
         <CONT>"emprendre", to undertake
fan_cy-free
  <SYN>carefree, uncommitted, unattached
macintosh <SYN>rubberized cloth, raincoat, personal computer brand
          <CONT>name
          <DATA>named after Scottish chemist Charles Macintosh - often
          <CONT>spelled "mackintosh"
PS/2
<DATA>short for Personal System/2, a product of IBM
Hwy. <SYN>Highway <ABBR-EOS>
```

The following, however, are not allowed in a flat file addenda dictionary:

```
brat_tice <SYN>bulwark, partition<MORPH>lattice[1]
clep_sy_dra <DATA>from ancient Greek "klepsudra" <SYN>water clock[2]
em_prise <SYN>undertaking,
              adventure,
              enterprise[3]
         <MORPH>enterprise
         <DATA>from Old French past-participle of
         <CONT>"emprendre", to undertake
fan_cy-free

  <SYN>carefree, uncommitted, unattached[4]
macintosh <SYN>rubberized cloth, raincoat, personal computer <CONT>[5]
          brand name
          <DATA>named after Scottish chemist Charles Macintosh -
  often <CONT>spelled "mackintosh"[5]
PS / 2[6]
<DATA>short for Personal System/2, a product of IBM
Hwy. <ABBR-EOS>Highway[7]
```

[1]Needs a space before the <MORPH> tag.
[2]Fields in the wrong order; <SYN> should precede <DATA>.
[3]Data split over multiple lines without <CONT> tags.
[4]Extra line between headword and data.
[5]<CONT> tag may only be preceded on a line by whitespace.
[6]No spaces allowed within the headword.
[7]No data is allowed for the <ABBR-EOS> tag.

In the word itself, any of the alphanumeric characters may be used, as well as periods, forward slashes, apostrophes, hard hyphens and ranked hyphenation symbols. The words may be in any code page supported for Linguistic Tools. Because the flat addenda dictionary does not contain any explicit code page declaration, the application must ensure that the appropriate code page is current when the addenda is to be activated. On activation of the addenda, each word is translated into the Linguistic Tools internal code page and checked to insure its validity. If a word is found to contain an unsupported character, it is skipped. Similarly, a duplicated word will be skipped.

Periods in a word are used to indicate that the word is an abbreviation. The forward slash is frequently used in trade names. Since periods and slashes are normally allowed in a word only when surrounded by numerics, the user should be aware that during dictionary activation, words in an addenda dictionary will be broken at the slashes and/or periods into pieces which will be used by Spell Verify. This will allow Spell Verify to recognize each piece of the word. Linguistic Tools currently does not allow hyphenation, synonym, morphology, or other data to be stored with word pieces created in this fashion.

Apostrophes in a word are used to form English contractions and possessives, and to indicate accented characters at the end of Italian words. In the latter case, special processing is performed to make a copy of the word, changing the apostrophe and its preceding character to the proper accented form of the character. No data is associated with these words.

Hard hyphens in a word are used to indicate where a hyphen must occur in the word when it appears in a document. In this way, they are treated identically to alphanumeric characters. These hyphens are not to be confused with syllable

hyphens (also known as soft hyphens), which are used in a document to indicate continuation when a word is broken at the end of a line.

If the application wishes to provide information for use by the **Hyphenate** function, it should be indicated by using the hyphenation symbols listed below to mark the optional hyphenation points in the word.

- (hard hyphen) required hyphen

+ (plus) compound word component break

, (comma) special processing hyphen

= (equals) other preferred hyphenation point

_ (underscore) syllable hyphen

Of these, the following are the ranked hyphenation marks, which are supported only for the Germanic languages:

+
=
,

For all other languages, only hard hyphens and syllable hyphens are recognized. (See "Ranked Hyphenation" on page 28 for additional information.)

The data for a word may consist of synonyms, double-byte parsing codes, morphology information, abbreviation classifications and user- or application-supplied data.

Synonym data may include several synonyms for the word, with individual synonyms separated by commas (whitespace may also be present between the comma and the next synonym). An individual synonym may be a single word or a phrase. This data will be returned by **Synonym Aid**. (See "Synonym Aid" on page 157 for details.)

Double-byte parsing code data consisting of a list of double-byte parsing codes (DBPCs) separated by commas may be entered to alter the parsing characteristics of the Linguistic Tools. These codes are used for Chinese, Japanese, and Korean language-specific processing and are documented in "Chinese Double-Byte Parsing Codes (DBPCs)" on page 287, "Japanese Part of Speech (POS) Codes" on page 324, and "Korean Double-Byte Parsing Codes (DBPCs)" on page 331.

Morphology information consists of a single alias term, whose pattern of inflectional spelling changes will be applied to the new word. (See "Adding Morphological Information to Addenda Dictionaries" on page 23 and "Add Morphology to Addenda" on page 128 for more information.)

Abbreviation classifications consist of a tag which should only be present when the word is an abbreviation delimited by a period. This tag indicates that the abbreviation may occur at the end of a sentence. If an abbreviation occurs in the addenda dictionary without this tag, Linguistic Tools will assume the abbreviation may not be used to end a sentence. See "Text Segment Identification" on page 33 for additional information.

The user—or application—data field may contain any string; Linguistic Tools will simply return it as is when the word is retrieved using **Look Up Word**.

Some consideration should be given to case when adding words to an addenda dictionary. There are four possible combinations of upper and lower case letters in any word:

- all lowercase
- all uppercase
- first letter only uppercase, rest lowercase (initial capital)
- "mixed" case (at least one letter other than the first is uppercase, while other letters in the word are lowercase).

Linguistic Tools will allow an all-lowercase dictionary word to match an input word which is all lowercase, all uppercase, or which has an initial capital. An all-uppercase dictionary word will match only an all-uppercase input word, while a word which has an initial capital letter in the dictionary can match either an all-uppercase input word or one which has only the initial capital. Mixed case words must match exactly, unless the **NlpMixedMode()** function is set to allow an all-uppercase input word to match a mixed case dictionary word.

Most words in an addenda dictionary probably will be all-lowercase. Abbreviations, acronyms, names, and other items with distinctive casing should be represented with uppercase (all, initial, or mixed) to ensure that input words verify only when casing is correct. "Case Matching between Input Words and Dictionary Words" on page 82 explains how each of the text-processing functions handles different case patterns in input words.

Addenda DBCS Considerations

All DBCS text passed to addenda functions may be mixed DBCS data. Thus, text may optionally include shift-in/shift-out information and/or other SBCS characters.

All DBCS output will be only in DBCS characters, except that host code page text will have a single SBCS shift-out character before the DBCS text and a single SBCS shift-in character after the DBCS text.

An entry in a DBCS flat-file addenda dictionary may take the form

```
@# <DBPC>19, 23 <DATA>Japanese addenda example
```

where @ and # represent Japanese DBCS characters. On the host, the DBCS text string must be surrounded by shift-out and shift-in characters.

The tags must be SBCS, as must the double-byte parsing codes, but the <DATA> field may contain SBCS, DBCS, or mixed text. Because the Linguistic Tools does not support the European-style morphological functions for the DBCS languages (Chinese, Japanese, and Korean), data bearing the <MORPH> tag will be ignored. Only the <SYN>, <DBPC>, and <DATA> information will be accessible through Linguistic Tools.

Adding Morphological Information to Addenda Dictionaries

Add Morphology to Addenda enables a linguistically-sophisticated user to associate morphological information with words in an addenda dictionary. The underlying assumption is that each of the added words will follow the inflectional pattern of some word in an active System Dictionary.

In order to do this, the application will have to ask the user to supply the model term or *alias*. The application will call **Generate Inflected Forms (GIF)** for the alias and save the output in a reply area. Then **GIF** must be called again for the new word, with the alias supplied. **GIF** will not find the new word in any existing dictionary, but it will find the alias and apply the pattern of spelling changes for the alias to the new word, producing a set of inflected forms for the new word. This output should be placed in a separate reply area from the first by the application.

Since it is possible for the user to have selected as an alias a word which has more than one part of speech and/or paradigm or one which turns out not to follow the desired inflectional pattern, the application should display both sets of output to the user to confirm that only the correct set of forms will be produced for the new word. If the user agrees, then the application may proceed to call **Add Morphology to Addenda**.

Add Morphology to Addenda will go through the output in both reply areas, associating each form in the second with the corresponding form in the first, and placing the resulting word/alias pairs in the specified addenda dictionary. In languages with many inflected forms, this may result in a considerable number of entries being added to the dictionary.

For example, suppose the user wants to put into an addenda dictionary a new word "attrit", which follows the same inflectional pattern as the existing verb *admit*. A call to **GIF** for *admit* will produce the following inflected forms:

```
admit
admits
admitted
admitting
```

Various kinds of morphological information (part of speech, PCODE, paradigm number, form ID, and morphology mask) will be associated with each form. These have been omitted here for simplicity's sake.

A second call to **GIF** for *attrit*, using *admit* as an alias, will produce the following inflected forms:

```
attrit
attrits
attritted
attritting
```

again with various kinds of morphological information (taken from the alias term) attached to each form.

If the user had selected another word as an alias—for example,"permit"— that has more than one part of speech (both verb and noun), the call to **GIF** would have produced two sets of inflected forms, one for the verb and one for the noun. Since "attrit" is only used as a verb, the second set of forms would be wrong. Because the spelling of the singular noun happens to coincide with the spelling of the base form of the verb, while the spelling of the noun plural coincides with the third person singular present tense form of the verb, it would not be immediately obvious from inspection of the addenda dictionary entries that there are extraneous forms

for the added word. However, attempts to use the alias for morphological identification or generation of inflected forms could produce undesirable results. For this reason, it is recommended that the application display the results of the second **GIF** call to the user to confirm that these are the correct forms. In the example, the presence of a second set of forms for a noun paradigm would cause the user to reconsider his/her choice of alias.

If the user agrees that the forms returned by the second **GIF** call are the correct forms, the call to **Add Morphology to Addenda** will result in the following entries being added to the dictionary:

```
attrit <MORPH>admit
attrits <MORPH>admits
attritted <MORPH>admitted
attritting <MORPH>admitting
```

Only the alias, not all of its associated morphological detail, needs to be stored in the dictionary. **Morphological Identification** and **Generate Inflected Forms** will look up the alias in the System Dictionary to obtain the relevant morphological information.

Removing Words from Addenda Dictionaries

The application may remove words from an addenda dictionary by calling **Remove Word from Addenda**.

Listing the Words in Addenda Dictionaries

The **List Addenda Words** function lists out all of the words in the specified addenda dictionary. For each word, a contents flag indicates whether there is data associated with it and what kind; however, the actual data is not included in the listing. To retrieve the data for a specific word, the application should call **Look Up Word**.

Looking Up Words in Addenda Dictionaries

A single word and its associated data may be retrieved by calling **Look Up Word**. To obtain a list of all the words in a specified addenda dictionary, the application may call **List Addenda Words**.

Look Up Word will also find words in System Dictionaries. If a System Dictionary containing the specified word precedes the addenda dictionary in the input dictionary list, this function will find the word in the System Dictionary and stop looking. In order to find the word and its data in the addenda dictionary, the application must place the handle of the addenda dictionary before the System Dictionary handle in the input dictionary list.

Note: Addenda dictionary files should be managed in the same fashion as any other application file that can be accessed by multiple users. Linguistic Tools will protect file integrity while the **Save Addenda** function is actually writing to it, but no other file locking mechanism is provided by the Linguistic Tools. It is the

application's responsibility to provide general write protection for addenda dictionary files.

Saving Addenda Dictionaries to Disk

To save changes that have been made to an addenda dictionary, the application should call the **Save Addenda** function. This function may be called at any time during a Linguistic Tools session to save changes made during that session.

An addenda dictionary may grow as large as the disk media allow. Linguistic Tools will page addenda dictionary information back and forth from a temporary file. Because of this, the results of calls to **Add Word to Addenda**, **Add Morphology to Addenda**, and **Remove Word from Addenda** will not be saved automatically to a permanent disk file. To ensure that all changes to an addenda dictionary are saved, the application should call the

Addenda Portability Issues

If a binary addenda dictionary is uploaded to a VM/CMS system, the file should be uploaded in binary, fixed-record-length format and the record length should be 2048.

If a flat-file addenda dictionary is to be transferred between platforms, it should be transferred as a text file with the appropriate code page translation performed.

Chapter 4. Text-Processing Functions

Spell Verify

The **Spell Verify** function attempts to find input words in the specified dictionaries. If a word is found, it is considered to be correctly spelled. If a word is not found, it is considered to be incorrectly spelled.

The user may indicate personal spelling preferences or add a word that is not in the System Dictionary by placing the word into an addenda dictionary.

"Case Matching between Input Words and Dictionary Words" on page 82 describes how Linguistic Tools' spell checking functions handle upper, lower, and mixed case input.

Article Checking

Linguistic Tools provides an option to check for article errors—incorrect use of "a" or "an"—as part of the spelling verification process for U.S. English only. This option is selected by a flag in the call to **Spell Verify**. See "Spell Verify" on page 139 for more details.

Spell Aid

The **Spell Aid** function provides suggested spellings for a given word by gathering from the specified dictionaries a list of words which resemble the input word. The words, or candidates, gathered from each dictionary are ordered so that those which most closely resemble the input word are put at the top of the list for that dictionary. Resemblance is based on both orthographic criteria, that is, how closely the letters of the input word match those of the dictionary word, and phonetic criteria, that is, how close the input word sounds to the dictionary word. Phonetic resemblance is determined by a set of rules loaded from the System Dictionary.

Normally an application will call **Spell Verify** first to locate words which may be misspelled, then call **Spell Aid** for each misspelled word. **Spell Aid** can also be called for words that have not been identified as misspelled—it will return candidates for any word for which it can find acceptable candidates. Note, however, that the **Spell Aid** function by itself does not first attempt to verify the word, so it cannot tell the user whether the word actually was misspelled. Only **Spell Verify** can do that.

Fuzzy Spell Aid

The **Fuzzy Spell Aid** function provides suggested spellings for a given word by gathering from the specified dictionary a list of words which resemble the input word. It is functionally similiar to regular **Spell Aid**, but it differs in the following ways:

- It is not as restrictive as regular **Spell Aid** in selecting candidate words

© Copyright IBM Corp. 2000

- The user can specify the maximum acceptable “distance measure” (a measure of dissimilarity) for candidate words
- It can select candidate words from **one and only one** specified **addenda** dictionary (**Spell Aid** can select from multiple system and/or addenda dictionaries)
- It returns up to 20 candidate words. (**Spell Aid** returns up to 6 per dictionary.)

Grade Level Analysis

Linguistic Tools also includes a function that will return grade-level information for U.S. English words. See “Grade Level Analysis” on page 147 for details.

Hyphenation

To assist the application in determining where to break words across lines, the **Hyphenation** function provides one or all hyphenation points in a word, as requested by the application. If the word cannot be hyphenated or if the word contains too few characters for the hyphenation function (that is, fewer than 5), the function will return zero hyphenation points.

Hyphenation may be either algorithmic or dictionary-based. Algorithmic hyphenation guarantees hyphenation points for any word. Dictionary-based hyphenation provides a higher level of accuracy. The application specifies which type of hyphenation is used when a dictionary is activated. (See “Supported Functions by Language” on page 48 to find out what kind of hyphenation support is available for each language.)

Ranked Hyphenation

Hyphenation has been traditionally indicated by required hyphens and syllable hyphens. Required hyphens (sometimes referred to as “hard” hyphens) are those which are required to be part of the word no matter where the word occurs, as opposed to syllable hyphens (sometimes referred to as “soft” hyphens) which are only used when a word at the end of a line is broken between syllables. For example, to show all of the possible hyphenation points for the word “high-spirited”, it would be marked as “high-spir_it_ed”, where hard hyphens are represented by “-” and soft hyphens are represented by “_”. This type of marking made it possible to choose between breaking words at the hard hyphen mark or at the soft hyphen marks. The introduction of Ranked Hyphenation incorporates additional marks to differentiate between several types of syllable hyphens which improves the quality of the resulting hyphenation.

The new marks are: equals (=) to indicate a preferred hyphen, plus (+) to indicate a component hyphen, and comma (,) to indicate a hyphenation point requiring special processing. Hard hyphens continue to be marked as before; when ranked hyphenation is not requested, underscore continues to be used as before for all syllable hyphens. In ranked hyphenation, the syllable hyphen character (represented by an underscore for code pages which have no syllable hyphen character) indicates a soft hyphen that is not preferred, does not occur at a component break, and requires no special processing. The hierarchy of all these marks in decreasing preference is:

1. Hard hyphen

2. Component hyphen
3. Special processing hyphen
4. Preferred hyphen
5. Soft hyphen

The first three types generally delimit word component boundaries, the preferred hyphen usually marks morpheme boundaries, and the soft hyphen separates syllable boundaries.

When ranked hyphenation information is available—that is, when all of the possible hyphenation points within a word have been classified according to precedence—the application may elect to have preferred hyphenation points returned over less-desirable points within a specified range. At present, ranked hyphenation is available for Danish, Dutch (all versions including Afrikaans), National German (all versions), Norwegian (both Bokmål and Nynorsk), and Swedish.

The user may indicate personal hyphenation preferences by placing a word with the preferred hyphenation points in an addenda dictionary. If the addenda dictionary precedes the System Dictionary in the input dictionary list, the user's hyphenation points will be returned.

Dehyphenation

The dehyphenation function removes all syllable hyphens from an input word and reverses any hyphenation-related spelling changes—for example, pre-reform German *Ak-ker* becomes *Acker* when dehyphenated, and *Schiff-fahrt* becomes *Schiffahrt*.

Synonym Aid

The **Synonym Aid** function compiles a list of candidate words which are synonyms for an input word. The input word is matched against words in dictionaries which contain synonym information and, when a match is found, the synonym information for the word is used to compile the candidate list. Only dictionaries which contain synonyms are searched, and only information from one dictionary (the first one found to contain synonym information) is used to compile the synonym candidate list.

The synonym returned may be for the word as input, for the base forms of the word, or for both. For example, “running” may have synonyms under both “run” and “running.” The headword (word that actually matched in the dictionary) or headwords are returned in the reply area, along with the synonyms for each.

To obtain an inflected form of a synonym word, the application should request the user to select one of the candidates returned by **Synonym Aid** and then call **Inflect Word from Model**. See “Inflect Word from Model” on page 32 for details.

Additional information may be associated with a synonym or synonym group by means of “qualifiers.” A definition is one type of qualifier. Other qualifiers may provide information about usage, context, and so on. Definitions currently are

available only for U.K. and U.S. English. See “Synonym Aid” on page 157 for details.

Words not in a System Dictionary but added to an addenda dictionary may also have synonyms associated with them. (See Chapter 3, “Addenda (User) Dictionary Support Functions” on page 17 for details on adding synonym information to an addenda dictionary.) When an addenda dictionary is passed to **Synonym Aid** and the input word is present with its accompanying synonyms in the addenda dictionary, the synonyms are retrieved from the addenda dictionary and output in the reply area. In addition, when both an addenda dictionary and a System Dictionary are passed to **Synonym Aid** and a morphology alias is present with the word in the addenda dictionary, the morphological information for the alias term is retrieved from the System Dictionary and applied to the input word to identify the base forms of the input word. These forms are then located in the addenda dictionary, and any synonyms for these forms are also retrieved and output. (See “Adding Morphological Information to Addenda Dictionaries” on page 23 for details.)

Morphological Identification

Linguistic Tools' morphology functions allow the application to obtain grammatical information about words as well as inflect words to specific forms.

Morphological Identification returns morphological information about an input word. This function processes one word per call and stops processing after information has been found in one dictionary. The morphological information returned by this function consists of the lemma (base form) of the input word, the part of speech, PCODE, DBPC, the inflective classification represented by a bit mask (for example, a word can be 1st, 2nd, or 3rd person, singular or plural, and so on), and the paradigm number (the combination of paradigm number and the word itself uniquely defines a set of inflections for the word). If the input word has multiple lemmas (base forms), then all lemmas will be returned. This function is intended to be used by text indexing and information retrieval systems, syntactic parsers, grammar checkers, and so on.

A ***paradigm*** consists of a set of inflected forms generated from the same base (lemma) form. For example, the paradigm of the verb “ring” includes the forms “ring, rings, rang, rung, ringing.” Each element of the paradigm has a different spelling, but some elements— such as “ring”—can represent more than one inflection—in this case, first and second person singular and all persons plural in the present tense. The pattern of spelling changes through this set of forms is represented by a numbered entry in a morphology table. The verb “sing” follows the same pattern, so it can be assigned the same paradigm number. However, the noun “ring” belongs to a different paradigm that has only the members “ring” and “rings.”

A paradigm number and a word uniquely identify a set of inflections. Warning: paradigm numbers will not be the same for the same word in every version of the dictionary. There is no intention of making these numbers upwardly compatible with future dictionaries. Likewise, paradigm numbers will differ from one language to another. There is no intention of making these numbers uniform between languages.

A ***PCODE*** is a seven-byte code keyed to the dictionary morphology tables in which paradigm information is stored. The PCODE serves as a cover term for a group of

paradigms. For example, there is one PCODE for all English verbs which can be used as the main verb in a sentence. The modal verbs —“can, may, must, shall, will, should, would, could”— follow a different pattern of grammatical behavior, so they are assigned a different PCODE. Similarly, those adjectives in English that have comparative (-er) and superlative (-est) forms have a different PCODE than the adjectives with which one must use “more” or “most” (for example, intelligent). Each language has a different set of PCODEs.

A ***double-byte parsing code (DBPC)*** is a two-byte language-specific hex value which contains part of speech information. This value is used to parse words or lemmas from text in double-byte languages. Currently DBPCs have been identified for Chinese, Japanese, and Korean. These are listed in “Chinese Double-Byte Parsing Codes (DBPCs)” on page 287, “Japanese Part of Speech (POS) Codes” on page 324, and “Korean Double-Byte Parsing Codes (DBPCs)” on page 331, respectively. Note that although the numeric values overlap, the significance of each value is different for each language. For all languages other than Chinese, Japanese, and Korean, the DBPC value is zero.

A ***morphology mask*** is a 32-bit field which is used to represent inflectional information about a word: person, number, tense, aspect, mood, gender, case, and so on. One or more masks may be used to indicate all of the possible inflections. All masks are atomic, that is, each mask represents only one inflectional class. A detailed listing of bit meanings for the morphology masks for the various languages supported for Linguistic Tools is given in Appendix E, “Morphology Grammar Masks” on page 347. Note that morphology mask information is available only for the SBCS languages.

Words not in a System Dictionary may be processed by identifying another word (or “alias”) that is contained in the System Dictionary and whose forms follow the same inflectional pattern. This alias term is used to retrieve morphological information which is then applied to the original word.

Similarly, words not in a System Dictionary but added to an addenda dictionary may also have morphological information (in the form of an alias term) associated with them. (See “Adding Morphological Information to Addenda Dictionaries” on page 23 for details on adding morphological information to an addenda dictionary.) When both an addenda dictionary and a System Dictionary are passed to the **Morphological Identification** function and the input word is present with its accompanying alias term in the addenda dictionary, the morphological information for the alias term is retrieved from the System Dictionary and applied to the input word.

Generate Inflected Forms

This function generates all or only the specified inflected forms of a word. Forms can be requested by specifying some subset of the following: paradigm number, PCODE, part of speech, form ID, inflectional classification (represented by the morphology mask). See “Morphological Identification” on page 30 for an explanation of paradigms, PCODEs, and so on. Only forms which satisfy all input parameters will be output. Note that the input word does not have to be a lemma form, but may also be an inflected form. The output from **Morphological Identification** can be used as input to the **Generate Inflected Forms** function.

Words not in a System Dictionary may also be processed in the **Generate Inflected Forms** function by identifying another word (or “alias”) which is contained in the System Dictionary and whose forms follow the same inflectional pattern. This alias term is used to retrieve morphological information which is then applied to the original word.

Words in an addenda dictionary are handled in much the same way as described for the **Morphological Identification** function. When both an addenda dictionary and a System Dictionary are passed to the **Generate Inflected Forms** function and the input word is present with its accompanying alias term in the addenda dictionary, the morphological information for the alias term is retrieved from the System Dictionary and applied to the input word in order to generate the desired forms.

Inflect Word from Model

This function is designed to be used in conjunction with the **Synonym Aid** function in order to return appropriately inflected forms of the selected synonyms. The **Inflect Word from Model** function takes a model word in a particular inflected form (such as the word from the user’s text which was passed as input to **Synonym Aid**) and an input word which is usually uninflected (such as a synonym returned from the call to **Synonym Aid**) and inflects the input word to match the inflection of the model word. More than one form may be output if there is ambiguity regarding the model word (that is, if “vocalized” is the model word and “sing” is the word to inflect, both “sang” and “sung” will be output).

Note: This function is often confused with the alias option of the **Generate Inflected Forms** function. Though both functions use morphological information to inflect an input word, there are a few basic differences:

- This function requires that both the input word and the model word exist with morphological information in a System Dictionary. The **Generate Inflected Forms** function only requires that the alias word be present with morphological information in the System Dictionary.
- This function assumes that the input word and the model word are not in the same tense, while the **Generate Inflected Forms** function assumes that the input word and the alias word are in the same tense.
- The **Generate Inflected Forms** function requires that the forms of the alias word follow the same inflectional pattern as the forms of the input word. For example, the forms of the word “sing” (sings, singing, sang, sung, and so on) follow the same pattern as the forms of “ring” (rings, ringing, rang, rung, and so on), but do not follow the pattern of the forms of “ping” (pings, pinging, pinged, and so on.). So, if “ping” is input and “sing” is the alias, the resulting forms will be “pings,” “pinging,” “pang” and “pung,” which are obviously incorrect. **Inflect Word from Model**, however, does not have the same requirement. For example, if “ping” is input and “sang” is the model, “pinged” is returned as the matching form.
- The **Generate Inflected Forms** function is intended to output all the forms of a word, unless the range of forms is limited by the values of the input parameters on the call. **Inflect Word from Model** is intended to output only the form of the input word which matches the form of the model word, unless there is ambiguity about which form of the model word is indicated.

Simple Tokenization

The **Simple Tokenization** function isolates a block of text into smaller components that are useful for further processing. Single-byte text is divided as specified in "Definition of a Word in SBCS Text" on page 80. Double-byte text is separated from the single-byte text, but all contiguous DBCS strings are returned as single tokens except for certain double-byte punctuation characters. Output text from **Simple Tokenization** is stored in a token list structure (see "Tokens and Token Lists" on page 89).

Text Segment Identification

A text segment is a sentence or sentence fragment. Linguistic Tools determines text segment boundaries based on punctuation rules and limited language-specific processing. Although some language-specific processing involves abbreviation processing, the level of function provided varies widely by language. Most languages that use single-byte code pages have an associated Abbreviation Addenda Dictionary which is provided with Linguistic Tools Since double-byte languages typically do not employ abbreviations with periods, Abbreviation Addenda Dictionaries are not available for these languages.

The output of the **Text Segment Identification** function uses a token list to represent the text of the segment; an LX_NSEN token is inserted before each text segment.

Text Segment Identification is designed to be run with or without the help of dictionary information. If a dictionary is available, Linguistic Tools will be able to use the following kinds of information in doing text segment identification:

- Spelling Verification
- Hyphenation/Dehyphenation
- Language-Specific Support

If no dictionary is available, the function can still be used, but may return less accurate results.

Sentence End Conditions

Determination of text segment boundaries is done primarily through punctuation matching. In addition, special input types and number of words are used as cues to mark the end of a text segment.

Table 2. Text Segment Terminating Punctuation Characters

GCGID of SBCS character	GCGID of DBCS character	Description
JQ700000	JQ700080	Double-byte Circle Period
SP020000	SP020080	Exclamation Point
SP110000	SP110080	Period
SP140000	SP140080	Semi-colon (Greek Question Mark)
SP150000	SP150080	Question Mark

Terminating punctuation characters (listed in Table 2) trigger end-of-sentence processing. If any of these characters are found, they will always be taken to mark the end of a sentence, unless one of the exception conditions below is true. Of the following exceptions, note that some apply to all terminating punctuation characters, while others apply only to single- and double-byte periods (not the circle period).

1. The terminating punctuation character is followed by a closing punctuation character.

   The list of recognized closing punctuation characters is given in Table 3. If a terminating punctuation character is followed by a closing punctuation character, the terminating punctuation character is not interpreted as the end of the sentence. Instead, the final closing punctuation character will be considered the sentence delimiter.

Table 3. Closing Punctuation Characters

GCGID of SBCS character	GCGID of DBCS character	Description
JQ720000	JQ720080	Single Square Quote
JQ720001	JQ720081	Double Square Quote
SM080000	SM080080	Closing Bracket
SM140000	SM140080	Closing Brace
SP040000	SP040080	Double Quote
SP050000	SP050080	Single Quote
SP070000	SP070080	Closing Parenthesis
SP070001	SP070081	Closing Carapace Bracket
SP070002	SP280000	Closing Single Angle Quote
SP070003	SP070083	Closing Double Angled Quote
SP070004	SP070084	Closing Cornered Parenthesis
SP200000	SP200080	Closing Single Hook Quote
SP220000	SP220080	Closing Double Hook Quote
SP250000	SP250080	Vertical Closing Parenthesis
SP350001	SP250081	Vertical Closing Carapace Bracket
SP250002	SP250082	Vertical Closing Single Angled Quote
SP250003	SP250083	Vertical Closing Double Angled Quote
SP250004	SP250084	Vertical Closing Cornered Parenthesis
SP250000	SP350080	Vertical Closing Brace
SP370000	SP370080	Vertical Closing Single Square Quote
SP370001	SP370081	Vertical Closing Double Square Quote

   If more than one closing punctuation character follows a terminating punctuation character, the end of sentence is marked by the final closing punctuation character in the series. For example, if a period is followed by a quotation mark and two closing parentheses, the second parenthesis marks the end of the sentence. The one exception to this rule is in National German, where a closing quotation mark is not considered to mark the end of a sentence if it is followed by a comma.

2. The terminating punctuation character is immediately followed by another terminating punctuation character.

   If a terminating punctuation character is followed by any terminating punctuation character, then it will not be considered to mark the end of the sentence. Rather, the last in the sequence of terminating punctuation characters will be regarded as the sentence delimiter. This handles cases such as “What did you tell her???!!!!”. The final exclamation point would be regarded as the end of sentence marker.

3. The terminating punctuation character is not considered to be a word delimiter.

   As described in “Definition of a Word in SBCS Text” on page 80, a period is not considered to be a word boundary if it is preceded by either a numeric or a punctuation character and followed by a numeric character. In these cases the period will also not be considered to be an end of sentence marker. This prevents strings such as “1.25” and “.314” from ending a sentence.

4. The terminating punctuation character is a period and is part of an abbreviation that is not allowed at the end of a sentence.

   Limited abbreviation processing is performed for every language. If the handle of an Abbreviation Addenda Dictionary is supplied on input to the function, Linguistic Tools also checks the addenda to determine whether a given piece of the input text is an abbreviation. See “Activating Dictionaries” on page 13 for more information on obtaining a dictionary handle. Processing to determine whether an abbreviation ends a sentence is described in “Abbreviation Processing.”

5. The terminating punctuation character is a period and is not followed by any white-space characters.

   If the period is not followed by a space character or a new-line, it will not mark an end of sentence. This is to handle headings such as “III.IV”.

The end of a text segment may also be determined by the presence of certain data element types in the input data element list. If two or more consecutive elements of type LX_NLIN are encountered, they will be taken to mark a text segment boundary. Similarly, data elements of type LX_NSEN or LX_PARA also mark the end of a text segment. When a text segment is ended by one of these data elements rather than by punctuation, Linguistic Tools will mark it as a sentence fragment by placing an LX_FRAGMENT property on the LX_NSEN token that precedes it.

Linguistic Tools also limits the number of words and punctuation that may be included in a single sentence. The number of tokens contained in a single sentence may not exceed LX_MAX_INPUT_TOKENS. This limit is required for other Linguistic Tools functions such as **Extract Keywords** which deal with text one sentence at a time. If a large block of text is input without terminating punctuation, Linguistic Tools will break it at this limit.

Abbreviation Processing

Abbreviation processing occurs mainly for the purpose of disambiguating periods as sentence delimiters; Linguistic Tools does not attempt to recognize acronyms or other abbreviations that do not contain periods. Abbreviation processing is triggered by the presence of either a single- or double-byte period preceded by alphabetic text. Although Linguistic Tools attempts to find or mark all abbreviations

containing periods within a text, some abbreviations may not be recognized. Abbreviations of two or more characters followed by a period may be added to an Abbreviation Addenda Dictionary to cause them to be recognized. If no Abbreviation Addenda Dictionary is passed to the function, all single letters followed by periods will be marked as abbreviations; no other abbreviation processing will take place.

Whether or not a piece of text is an abbreviation is often ambiguous, since an abbreviation might be mistaken for a normal word followed by a period. For example, consider the characters “no.” in the following sentences:

> Enter the no. of exemptions you are claiming.
> Answer each question yes or no.

But even when a piece of text is known to be an abbreviation, there is still ambiguity as to whether it ends a sentence. Some abbreviations never end a sentence, while others sometimes do. For example, consider the use of the abbreviation “Hwy.” in the following sentences:

> The drive down Hwy. 1 to Santa Cruz was beautiful.
> Many people speak highly of the Pacific Coast Hwy.

Because abbreviations may be ambiguous and because some abbreviations may not occur at the end of a sentence, Linguistic Tools attempts to classify abbreviations found in the input text. If a period is found to be part of an abbreviation that sometimes ends a sentence, further processing will be performed. If Linguistic Tools determines that the abbreviation is not at the end of the sentence, the token representing the period is joined with the token for the abbreviation text. Otherwise, the token representing the period remains a separate token.

Linguistic Tools uses three sets of criteria to determine whether a period is part of an abbreviation:

1. All single letters followed by a period are considered to be abbreviations. These are often found in proper names—for example, E.M. Smith, U.S.A. Single-letter abbreviations are classified as possible sentence ends.
2. Words contained in the input Abbreviation Addenda Dictionary are always considered to be abbreviations. Whether an abbreviation found in the addenda may end a sentence is determined by the information associated with that word in the addenda. For example, “Mr.” is marked in the U.S. English Abbreviation Addenda Dictionary as an abbreviation that may not end a sentence, while “etc.” is an abbreviation that can sometimes end a sentence.
3. Any words from two to six characters in length followed by a period, and not found in any of the input dictionaries or Abbreviation Addenda Dictionaries, are also considered to be abbreviations. This is to handle cases such as “Jrnl. Comp. Ling.”. Abbreviations determined by dictionary lookup are always treated as possible end-of-sentence candidates.

If an abbreviation is identified as a possible end of sentence, Linguistic Tools examines the text following the abbreviation to determine whether the abbreviation marks the end of the current sentence. If the following word begins with an uppercase letter, the abbreviation is taken to mark the end of the sentence.

Note: One case in which abbreviation processing is known to produce an error in text segment identification is when a 2-6 character string followed by a period is, in turn, followed by an unabbreviated word beginning with an uppercase letter—for example, “Assoc. Comp. Linguistics.” This is due to the interaction of two sets of criteria: one for identifying possible abbreviations and one for identifying possible end of sentence. “Comp.” is not included in the U.S. English Abbreviation Addenda Dictionary, and can only be identified as an abbreviation by the criteria under (3) above. Because “Comp.” was determined to be an abbreviation by the dictionary lookup strategy, Linguistic Tools considers it to be a possible end-of-sentence abbreviation. The presence of an uppercase letter at the beginning of “Linguistics” causes Linguistic Tools to identify the period following “Comp” as the end of a sentence.

If an abbreviation is followed by two or more new-lines, a new-sentence, or new-paragraph data element, an end of sentence has been reached. Also if the following text is an inverted question mark or inverted exclamation point, an end of sentence marker is inserted in the output.

If Linguistic Tools determines that the period is part of an abbreviation that does not end a sentence, it continues its search for a sentence delimiter. Otherwise, it checks other terminating punctuation character exception conditions (such as following terminating punctuation or closing punctuation) before marking an end of sentence.

Lexical Analysis

The **Lexical Analysis** function identifies text segment boundaries (see “Text Segment Identification” on page 33 for details), analyzes the resulting token list, and, if necessary, changes the structure of simple tokens containing internal punctuation as follows:

1. Dehyphenates text containing hyphens, including text across new-lines
2. Divides certain forms containing apostrophes:
   - prefixed function words
   - contracted auxiliaries
3. Combines words surrounding “/” into a single term
4. Identifies terms that should not be looked up in a dictionary, such as punctuation

Specific details of processing vary according to the language being processed. Table 4 on page 38 summarizes the kinds of processing that can be performed for each language supported by Linguistic Tools.

Language-specific Processing

Table 4. Lexical Analysis function by language

Language	Hyphenated forms	Numbers	Prefixed forms	Alternatives with “/”	Contracted Auxiliaries
Afrikaans	X	X			
Arabic	X	X			
Catalan	X	X	X		
Chinese†					
Czech	X	X			
Danish	X	X			
Dutch	X	X			
English	X	X		X	X
Finnish	X	X			
French	X	X	X		
German	X	X		X	
Greek	X	X			
Icelandic	X	X			
Hebrew	X	X			
Hungarian	X	X			
Italian	X	X	X		
Hebrew	X	X			
Hungarian	X	X			
Japanese†					
Korean†					
Norwegian	X	X			
Polish	X	X			
Portuguese	X	X			
Russian	X	X			
Spanish	X	X			
Swedish	X	X			
Thai	X	X			
Turkish	X	X			

† See Appendix D, “Language-Specific Processing” on page 281 for details of Lexical Analysis in these languages.

Three general types of action may be taken:

1. A token may be left as a unit, and properties may be added to it
2. A single token may be broken into multiple tokens, and properties may be added to the resulting tokens
3. Two or more tokens may be combined into a single token, and properties may be added to the resulting token.

It is important to note that the analysis of tokens described here and in the various language-specific discussions assumes that the delimiters described in “Definition of a Word in SBCS Text” on page 80 have been used for tokenization. If other delimiters are present, **Lexical Analysis** may not yield the expected results.

Lexical Analysis is designed to be run with or without the help of dictionary information. If a dictionary is available, Linguistic Tools will be able to use the following kinds of information in doing **Lexical Analysis**:

- Spelling Verification
- Hyphenation/Dehyphenation
- Language-Specific Support

If no dictionary is available, the function can still be used, but may return less accurate results. For DBCS text special processing is performed, for which an active Chinese, Japanese, or Korean System Dictionary is required.

Forms Containing Hyphens

Lexical Analysis processes hyphenated forms from the input text by normalizing some forms and specifying the dominant portion of others. The normalized form of a hyphenated word is the string formed by removing all hyphens. If a normalized form of the word is found in the dictionary, an LX_CPXWORD property is associated with the token representing the hyphenated word. The value of the LX_CPXWORD property points to the normalized form of the word. If a hyphenated word may not be normalized, the dominant portion of the form is specified with an LX_CPXHEAD property.

For example, a hyphenated form such as “con-taining” (1a below), is reduced to a single token to correctly represent the word “containing.” Note that cases such as this, in which the hyphen is eliminated, must be distinguished from cases like “mother-in-law” (2a) and “machine-readable” (3a), in which the hyphen is retained. This can be done accurately only by consulting a dictionary to (a) identify lexicalized hyphenated forms such as “mother-in-law”, (b) determine whether a form like “con-taining” is a valid lexical item with the hyphen removed, or (c) determine whether the elements of a form such as “machine-readable” are valid lexical items.

```
1. a.  "con-"   (NEWLINE)   "taining"

   b.  LX_CPXWORD -> "containing"

2. a.  "mother-"   (NEWLINE)   "in-law"

   b.  LX_CPXWORD-> "mother-in-law"

3. a.  "machine-" (NEWLINE) "readable"

   b.  LX_CPXWORD-> "machine-readable"
       LX_CPXHEAD-> "readable"

4. a.  "contain" (SYLLABLE HYPHEN) "ing"

   b.  LX_CPXWORD-> "containing"
```

Note: If dehyphenation determines that the word contains only syllable hyphens or if the hyphenated form is in the dictionary, it does not add the property LX_CPXHEAD.

The process used to normalize hyphenated forms is determined by two factors: whether the hyphen is a syllable hyphen or a hard hyphen, and the position of the form in a line of input text. If a form contains a syllable hyphen, regardless of its position in the text, it is passed directly to the **Dehyphenate** function. If a form contains a hard hyphen at the end of a line, special processing is performed to determine whether or not the hard hyphen should be treated like a syllable hyphen (as is the case in many flat file formats). In all cases, any trailing blanks following an end-of-line hyphen character are ignored and forms hyphenated across new lines are combined into a single token.

Linguistic Tools determines whether a hard hyphen should be treated as a syllable hyphen in two stages. The form containing the hard hyphen is passed to **Look Up Word**. If it is found in the dictionary, Linguistic Tools will treat the hard hyphen as a required hyphen. Otherwise, a copy of the form is composed substituting a syllable hyphen for the end-of-line hard hyphen. This is necessary to preserve spelling changes that may occur as a result of hyphenation. If this form is found in the dictionary, the function determines that the hard hyphen should be treated as a syllable hyphen and an LX_CPXWORD property is generated.

Terms containing required hyphens are then processed in order to determine the dominant portion of the hyphenated form. Linguistic Tools makes the simplifying assumption that the dominant portion of a hyphenated form is the word following the final hyphen. When a hyphenated form is not found in any available dictionary, the application may choose to use the dominant portion of the form to determine lexical properties of the entire form. For example, the form “natural-language” would take on the properties of “language” and be called a singular noun. “Machine-readable” would take on the adjectival properties of “readable”. For other forms, however, this results in the assignment of inappropriate features to the hyphenated form. This is the case, for example, with forms such as “run-on”. In the sentence:

Run-on’s are considered stylistically poor.

it is incorrect to treat the hyphenated form as a preposition. Nonetheless, this simplified approach is usually adequate when an application needs to have lexical information associated with every form.

Forms containing apostrophes

Depending on the language, **Lexical Analysis** may divide forms containing apostrophes into two separate tokens. In English, the word “don’t” would be separated into “do” and “n’t.” In French, the form “l’enfant” is separated into “l’” and “enfant.” Separating apostrophe forms is useful for treating the two parts as the separate words that they represent.

Forms containing the slash character

Because the slash character (“/”) is often used to indicate a single alternative between two terms, **Lexical Analysis** combines forms separated by a slash into a single token. For example, the forms “and” “/” “or” is combined into a single token representing “and/or”. This approach also produces single tokens for terms such as “VM/CMS” and “AS/400”.

When a slash form is discovered, an LX_CPXHEAD property is added to the token. The value of this property will point to the string following the last slash in the combined form.

Identifying words that should not be looked up

Any word that does not contain any alphabetic characters will be marked with the LX_NOLOOKUP property.

Extract Keywords/Extract Query Terms

Linguistic Tools provides two high level functions to assist an application in extracting content words (that is, “keywords”) from input text. **Extract Keywords** returns content words from a large block of text. **Extract Query Terms** returns content terms present in a fragment of text, as well as providing alternate terms. For example, a sentence such as ***“This is an example”*** returns ***example*** as a content word. Both attach to the keywords information about attributes and base forms of the word.

Note: Neither of the text extraction functions includes punctuation in its output.

Processing Options

Extract Keyword and **Extract Query Terms** perform similar kinds of linguistic processing; however, **Extract Query Terms** has options to return synonyms and inflected forms and provides a weighting factor for each returned form. Because **Extract Keyword** is designed to operate on a large block of text, it divides the text into sentences, while **Extract Query Terms** does not.

The application can constrain the processing and output of these functions by selecting one or more of the following options:

- **Part of Speech Filtering**
  - Return all parts of speech
  - Return nouns only
  - Return nouns and adjectives only
  - Return nouns and verbs only
  - Return nouns, verbs, and adjectives only
  - JAIRS part of speech filtering (see “JAIRS Part of Speech Filtering” on page 324)
- **Stopword Filtering**. Ignore high-frequency words in the input text. They are poor discriminators and cannot be used by themselves to identify text content. A predefined stopword list is used for each supported language and stored in a Stopword Dictionary. Input words which are found in the stopword list will not be included in the output.
- **Word Reduction**. Return base forms of a keyword.
- **Word Decomposition**. For Germanic languages only, this option decomposes compound keywords which were not found in the dictionary and returns all their possible components. All information requested for keywords will be returned for each component.
- **Word Modification**. Return keywords and base forms in a normalized form. See “Word Normalization” on page 44 for details.
- **Phonetic Key**. Return a phonetic key for each keyword.
- **Word Expansion**. Return the valid inflected forms of a keyword (**Extract Query Terms** only).

- **Synonyms**. Return synonyms of a keyword (**Extract Query Terms** only).
- **Start of Sentence Processing**. Special processing for the first word in a sentence. See "Extract Keywords" on page 185 for details.

Only the first option (part of speech filtering) is available for Japanese.

Keyword Order Number

All words in an input sentence, including stopwords, are numbered from 0 to LX_MAXWORDNUM. The numbering begins with zero for each new sentence identified by the **Text Segment Identification** function. Keyword order numbers will not change even when stopwords are omitted during processing. For example:

```
stopword                 X       X  X   X
word order number        0   1   2  3   4    5
input sentence           I think it is all noise
```

"Think" and "noise" are keywords, numbered 1 and 5, respectively. All stopwords (marked with X) are filtered.

Compound Word Processing

Special processing is done for compound words in the Germanic languages. These words are analyzed into one or more decompositions which, in turn, may consist of two or more components. Each component of each decomposition is assigned a two-part number: the first part is a decomposition number, and the second part is the number of the component within a decomposition. For example:

```
Component order numbers           1/1     1/2
                    Staubecken = Staub + ecken   (1st decomposition)
                                  2/1     2/2
                    Staubecken = Stau  + becken  (2nd decomposition)
```

This numbering is done only for text extraction, not for other Linguistic Tools functions dealing with compound words.

It is possible to decompose some forms into components which themselves are further decomposable. This is not permitted in the Linguistic Tools text extraction functions. That is, a decomposition of a compound word should never contain subpieces of an earlier decomposition. For example:

```
                  1/1    1/2
Alpenabschnitt = Alpen + abschnitt    (1st decomposition, valid)
                  2/1    2/2   2/3
Alpenabschnitt = Alpen + ab + schnitt (2nd decomposition, not valid
                                       because 2/2 and 2/3 are
                                       subpieces of 1/2)
```

Hard Hyphen Forms

For Germanic languages, a word containing hard hyphens will be processed as follows:

- If it is in the dictionary, process it as usual.
- If it is not in the dictionary
  - Call compound word processing to decompose it
  - If the word cannot be decomposed, break it at the hyphens into components which, in turn, are processed as usual. These components may be compounds themselves, but no further analysis is performed.

```
Examples:
      input                                             output

  staub-ecken   (becomes a compound word after        staub-ecken
                 the hyphen has been removed)         staubecken
                                                      staub
                                                      ecken
                                                      stau
                                                      becken
  staub-xxxx    (not in the dictionary with the       staub-xxxx
                 hyphen, and not a compound word      staub
                 after the hyphen has been removed)   xxxx
```

For non-Germanic languages, a word containing hard hyphens will be processed as follows:

- If it is in the dictionary, process it as usual.
- If it is not in the dictionary, break it at the hyphens into components which, in turn, are processed as usual.

```
Examples:
      input                                             output

  mothers-in-law  (in the dictionary)                 mothers-in-law

  fast-track      (not in the dictionary)             fast-track
                                                      fast
                                                      track
```

Components of Hard-Hyphen and Slash Words

Components of a hard-hyphen or slash word are numbered from 0 to LX_MAXCOMPNUM.

```
Example:
         0   1    2    3       4  5       6        7       keyword number
         I bought a hat/scarf for my 18-year-old sister.

   Hat/scarf is a slash word with two components: hat and scarf.
   18-year-old is a hard-hyphen word with three components: 18, year, and old.
```

Note that component numbering begins with 0 for the first component.

Word Normalization

The purpose of normalization is to rewrite the input word in a "neutral" form, converting the input text to all-lowercase, eliminating most accents, and resolving spelling alternations—such as National German Eszett becoming "SS" when the word is uppercased. Thus, it can be determined whether two words are in fact the same word by comparing the normalizations of the two words.

Normalization is performed only after the input text has been processed linguistically (decompounded and lemmatized). Note that the normalized form of a valid word may or may not be a valid word itself. The normalized form is intended only for comparison to other normalized forms. Since the normalized text will never be processed further by the Linguistic Tools, a normalization scheme that is highly destructive is employed.

Normalization processing is language-specific. The logic that performs word normalization within the Linguistic Tools must be passed a language indicator to determine what language-specific processing should be performed:

- No accent stripping is performed for the following languages:
  - Arabic
  - Finnish
  - Hebrew
  - Icelandic
  - Korean
  - Thai
  - Japanese
  - Chinese
- In both National and Swiss German, the umlauted letters ä, ö, and ü are rewritten as "ae", "oe", and "ue", respectively.
- In National German, Eszett (ß) is replaced with "ss". Note that an Eszett represented by "sz" will not be normalized into "ss".
- In Danish and Norwegian the national characters a-overcircle (å), ae-ligature (æ), and o-slash (ø) are rewritten as "aa", "ae", and "oe", respectively.
- In Swedish the national characters a-overcircle (å), a-diaeresis (ä), and o-diaeresis (ö) are rewritten as "aa", "ae", and "oe", respectively.
- All remaining accents are removed. This includes all accents in the following languages:
  - Dutch (including tremas)
  - English
  - Greek
  - Romance languages, which include these languages:
    - Catalan
    - French
    - Italian
    - Portuguese
    - Spanish
  - Afrikaans
  - Czech
  - Hungarian
  - Polish
  - Russian

- Turkish
- Danish
- German
- Norwegian
- Swedish

Non-alphabetic characters such as numerals and punctuation will normalize into themselves. Word normalization is not performed on DBCS text.

Because normalization is performed only after lemmatization, the loss of accents in Romance languages is not crucial, since the lemma (base) forms are less likely to be ambiguous than the inflected forms. Consider the following Spanish forms:

```
input word:         página
lemma:              página
normalized output:  pagina

input word:         pagina
lemma:              paginar
normalized output:  paginar
```

Note that the removal of all accents (including trema) in Dutch, as opposed to the respelling of umlauted vowels in German, means that German words and names will be normalized differently in Dutch than in German—for example, Müller will become “muller” rather than “mueller”. The justification for this is that visually the German umlaut cannot be distinguished from the Dutch trema; however, the latter serves a very different function. It is used to distinguish between sequences of two vowels that should be regarded as separate vowels and sequences that should be regarded as diphthongs— for example, in the word “geïllustreerd” the “ei” sequence represents separate vowels (two syllables), while the “ei” in “eisen” represents a diphthong (one syllable). If a Dutch person does not have access to the trema on a keyboard, he or she will type the character without it, while a German person who does not have access to the umlaut on a keyboard will type the vowel with a following “e”.

Single-Word Data Element Creation

Single-Word Data Element Creation assists an application in converting block format data elements to single-word format data elements. Otherwise, the application must do its own word isolation in order to pass text data to Linguistic Tools functions which accept only single-word format input. See “Input Data Element List” on page 69 and “Use of Single-Word Format Data Elements” on page 71 for additional information.

Compound Word Processing

Some languages use a great deal of compounding, allowing many words to be put together more or less freely from a large set of component pieces. Because of the productive nature of this process, it is not possible to include all of the resulting compounds in any dictionary of these languages. For the Germanic languages (Danish, Dutch, German, Icelandic, Norwegian, and Swedish), Linguistic Tools includes a function to decompose these compounds into their constituent parts so that other functions such as spelling verification, keyword extraction, and so on, can operate on these words. This processing is automatically invoked by setting the

proper parameters when calling these functions. It can also be invoked independently to verify that an input string is a component of a compound by calling the **Compound Word Component Isolation** function.

In languages with compound words, some compound words can be decompounded in more than one way, each decomposition having a different meaning. The **Compound Word Component Isolation** function will present all possible decompositions for the input compound word.

Example of multiple decompositions (German):

Staubecken -> Staub + Ecken or Stau + Becken
Druckerzeugnis ->Drucker + Zeugnis or Druck + Erzeugnis

The function will also return data indicating the positions in which each component may appear within a valid compound word.

All Germanic dictionaries specified by the application will be searched in order to find decompositions for the input word. The components of a given decomposition may be from different dictionaries of the same language.

The components for each decomposition are listed just as they appear in the dictionary with respect to case and accenting. If elision has occurred at the juncture between two components of the input compound word, the **Compound Word Component Isolation** function returns the components with the elided character(s) present. Elision is the loss of one or more characters at the juncture between two components of a compound word. For example, the pre-reform German compound word “Schiffahrt” is the combination of “Schiff” and “fahrt”. One of the “f's” is elided (removed) when the compound word is formed. The **Compound Word Component Isolation** function would return the two components “Schiff” and “fahrt” in this case, and indicate that elision occurred at the end of the first component.

This processing is invoked automatically for the following functions in Germanic languages:

- Spell verify
- Hyphenation/dehyphenation
- Lexical analysis

For the following functions, a flag must be set in the control block to invoke this processing:

- Keyword/query term extraction (*lx_wordde_f*)

Other languages also make use of the compounding process; however, the Linguistic Tools **Compound Word Component Isolation** function is not supported for all languages which have compounds. To some extent, the way in which **Lexical Analysis** has been implemented for Japanese and Chinese results in compound words being broken down to their components; however, this function was not specifically designed to do that.

Language-Specific Processing

A number of Linguistic Tools functions perform language-specific processing. For dictionary-based functions such as spelling verification, hyphenation, or synonym aid, the application specifies the default type of processing by activating the appropriate dictionary. Dictionary content information is summarized in "System Dictionary Content by Language" on page 55.

The text analysis functions **Text Segment Identification** and **Lexical Analysis** may be called either with or without any language-specific information. For most accurate results, the appropriate System Dictionary should be available, so that the function may make use of spelling, hyphenation, and other language-specific information contained in it, and the appropriate Abbreviation Addenda Dictionary should also be available. To ensure that **Text Segment Identification** performs the proper language-specific processing, the application should specify the language code in the *lx_lang_code* field of the Linguistic Tools control block. If no language is specified, default (non-language-specific) processing is performed. A detailed description of language-specific processing for each language is in Appendix D, "Language-Specific Processing" on page 281.

Germanic Language Support

The cover label "Germanic" includes not only German (all versions), but also Danish, Dutch (including Afrikaans), Icelandic, Norwegian (both Bokmål and Nynorsk), and Swedish. All of these languages have similar word-compounding processes and require similar kinds of processing by Linguistic Tools.

Linguistic Tools support for Germanic compounds includes one specialized function and enhancements to several other functions. **Compound Word Component Isolation** separates compound words into components. This function is also included as an optional feature of **Extract Keywords** and **Extract Query Terms** in order to break Germanic compound words into their components, which are returned as key words in addition to the compound as a whole. **Spell Verify** contains special language-specific processing to verify that valid compound words have been produced.

The **Hyphenation** function includes an option to return preferred hyphenation points for those languages for which data are included in the dictionary (currently all but Swiss German and Icelandic).

Details are given for each language in Appendix D, "Language-Specific Processing" on page 281.

DBCS Support

Currently four languages which use DBCS (double-byte) code pages are supported for Linguistic Tools: Chinese (Traditional and Simplified), Japanese and Korean. Only limited support is available for these languages. Chinese and Japanese pose special difficulties because of the way in which they are written—in these languages normal text consists of a string of DBCS characters with nothing to separate one word from another. The only delimiters normally present in these languages are punctuation marks. Korean does use blanks between words.

Note that although all of the DBCS code pages include the Latin alphabetic characters, Linguistic Tools does not identify these with their SBCS counterparts. If

given a double-byte Latin text string, Linguistic Tools will not convert it to single-byte for lookup in a European-language dictionary.

None of these languages use hyphenation. If a word does not fit onto the end of a line, it is simply continued on the next line with nothing to indicate that the word was split at the line break.

Spell checknig is not applicable for these languages, because they are picture languages. Error checking, to determine if a series of SBCS characters makes sense, is applicable. For Chinese, error checking and error correction are provided by the spell verify and spell aid functions. Error checking is not supported for Japanese and Korean in Linguistic Tools.

Morphological information of the type provided for the European languages is not yet available for Chinese, Japanese, or Korean. Currently, what has been implemented for these languages is not part of speech in the European sense, but (for Chinese, Japanese, and Korean) a system of DBCS Parsing Codes (DBPCs) that are used in Japanese for word isolation and in Chinese and Korean by the **Morphological Identification** function. Note that the **Morphological Identification** function is not supported for Chinese and Korean in the Macintosh implementation of Linguistic Tools. Synonym Aid is available from the Chinese System Dictionaries on all platforms other than Macintosh. However, definitions and synonym inflection are not supported. Synonym Aid is not available for Japanese or Korean.

Lexical Analysis is used for Japanese and Chinese (both Traditional and Simplified) in order to determine the word boundaries in DBCS text. However, **Lexical Analysis** does not support these languages in the Macintosh implementation.

Details for each language are given in Appendix D, “Language-Specific Processing” on page 281.

Supported Functions by Language

The following base functions are supported for all languages:

- Initialize
- Terminate
- Activate dictionary
- Deactivate dictionary
- Utility functions

The following functions, which the are the *addenda dictionary support* functions, are available for all languages:

- Create addenda
- Add word to addenda
- Add morphology to addenda
- Remove word from addenda
- List words in addenda
- Save addenda
- Look up word in addenda

However, because the **Add morphology to addenda** function uses the output of the **Generate inflected forms** function, it is supported only for those languages supported by **Generate inflected forms**.

Of the text-processing functions, the following are available for all languages:

- Text analysis functions
  - Simple tokenization
  - Text segment identification
  - Lexical analysis
- Text extraction functions
  - Extract keywords
  - Extract query terms
- Other functions
  - Single-word data element creation

However, processing of these functions may vary according to language. See Appendix D, “Language-Specific Processing” on page 281 for details, especially for **Lexical Analysis**.

Other text-processing support is summarized in Tables 5, 6, and 7.

Table 5 (Page 1 of 2). Text-Processing Functions by Language—Spelling Support

Language	Dictionary Name	SV	AC	SA	FSA	PSA	GLA
Afrikaans	AFRIKAAN.DIC	X		X	X	X	
Arabic	ARABIC.DIC	X		X			
Catalan	CATALA.DIC	X		X	X		
Chinese (Simplified)	CLKCHS.DIC	X		X			
Chinese (Simplified)	EFLCHS.DIC						
Chinese (Traditional)	CLKCHT.DIC	X		X			
Chinese (Traditional)	EFLCHT.DIC						
Czech	CZECH.DIC	X		X			
Danish	DANSK.DIC	X		X	X	X	
Dutch (Government restrictive spelling)	NEDERLND.DIC	X		X	X	X	
Dutch (Permissive spelling)	NEDPLUS.DIC	X		X	X	X	
English (Australian)	AUS.DIC	X		X	X		
English (United Kingdom)	UK.DIC	X		X	X	X	
English (United States)	US.DIC	X	X	X	X	X	X
U.S. English (Computing terms)	US.CPT	X		X	X		
U.S. English (Legal)	US.LEG	X		X	X		
U.S. English (Medical)	US.MED	X		X	X		
U.S. English (Combined U.S. English and medical)	USMED.DIC	X		X	X	X	
Finnish	SUOMI.DIC	X		X			
French (National)	FRANCAIS.DIC	X		X	X	X	
French (Canadian)	CANADIEN.DIC	X		X	X	X	
German (National pre-reform)	DEUTSCH.DIC	X		X	X	X	
German (National reform)	DEUTSCH2.DIC	X		X	X	X	
German (DPA rules national reform)	DEUT2DPA.DIC	X		X	X	X	
German (Swiss)	DSCHWEIZ.DIC	X		X	X	X	
Greek	HELLAS.DIC	X		X	X	X	

The following abbreviations are used in this table:

SV Spell verify
AC Article checking
SA Spell aid
FSA Fuzzy spell aid
PSA Phonetic spell aid
GLA Grade level analysis

Table 5 (Page 2 of 2). Text-Processing Functions by Language—Spelling Support

Language	Dictionary Name	SV	AC	SA	FSA	PSA	GLA
Hebrew	HEBREW.DIC	X		X			
Hungarian	MAGYAR.DIC	X		X			
Icelandic	ISLENSK.DIC	X		X	X	X	
Italian	ITALIANO.DIC	X		X	X	X	
Japanese	EFLJPN.DIC						
Korean	EFLKOR.DIC						
Norwegian (Bokmål)	NORBOK.DIC	X		X	X	X	
Norwegian (Nynorsk)	NORNYN.DIC	X		X	X	X	
Polish	POLSKA.DIC	X		X			
Portuguese (National)	PORTUGAL.DIC	X		X	X	X	
Portuguese (Brazilian)	BRASIL.DIC	X		X	X	X	
Russian	RUSSIAN.DIC	X		X	X	X	
Spanish	ESPANA.DIC	X		X	X	X	
Swedish	SVENSK.DIC	X		X	X	X	
Thai	THAI.DIC	X					
Turkish	TURKIYE.DIC	X		X			

The following abbreviations are used in this table:

SV Spell verify
AC Article checking
SA Spell aid
FSA Fuzzy spell aid
PSA Phonetic spell aid
GLA Grade level analysis

Table 6 (Page 1 of 2). Text-Processing Functions by Language—Hyphenation and Thesaurus

Language	Dictionary Name	DH	PH	AH	DE	SYN	DEF
Afrikaans	AFRIKAAN.DIC	X	X	X	X	X	
Arabic	ARABIC.DIC						
Catalan	CATALA.DIC	X		X	X	X	
Chinese (Simplified)	CLKCHS.DIC					X	
Chinese (Simplified)	EFLCHS.DIC						
Chinese (Traditional)	CLKCHT.DIC					X	
Chinese (Traditional)	EFLCHT.DIC						
Czech	CZECH.DIC			X	X	X	
Danish	DANSK.DIC	X	X	X	X	X	
Dutch (Government restrictive spelling)	NEDERLND.DIC	X	X	X	X	X	
Dutch (Permissive spelling)	NEDPLUS.DIC	X	X	X	X	X	
English (Australian)	AUS.DIC	X		X	X	X	
English (United Kingdom)	UK.DIC	X		X	X	X	X
English (United States)	US.DIC	X		X	X	X	X
U.S. English (Computing terms)	US.CPT	X			X		
U.S. English (Legal)	US.LEG	X			X		
U.S. English (Medical)	US.MED	X			X		
U.S. English (Combined U.S. English and medical)	USMED.DIC	X		X	X		
Finnish	SUOMI.DIC			X	X	X	
French (National)	FRANCAIS.DIC	X		X	X	X	
French (Canadian)	CANADIEN.DIC	X		X	X	X	
German (National pre-reform)	DEUTSCH.DIC	X	X	X	X	X	
German (National reform)	DEUTSCH2.DIC	X	X	X	X	X	
German (DPA rules national reform)	DEUT2DPA.DIC	X	X	X	X	X	
German (Swiss)	DSCHWEIZ.DIC	X		X	X	X	
Greek	HELLAS.DIC	X		X	X	X	

The following abbreviations are used in this table:

DH Dictionary-based hyphenation
PH Preferred hyphenation
AH Algorithic hyphenation
DE Dehyphenation
SYN Display synonyms
DEF Display definitions

Table 6 (Page 2 of 2). Text-Processing Functions by Language—Hyphenation and Thesaurus

Language	Dictionary Name	DH	PH	AH	DE	SYN	DEF
Hebrew	HEBREW.DIC						
Hungarian	MAGYAR.DIC			X	X	X	
Icelandic	ISLENSK.DIC	X		X	X		
Italian	ITALIANO.DIC	X		X	X	X	
Japanese	EFLJPN.DIC						
Korean	EFLKOR.DIC						
Norwegian (Bokmål)	NORBOK.DIC	X	X	X	X	X	
Norwegian (Nynorsk)	NORNYN.DIC	X	X	X	X		
Polish	POLSKA.DIC			X	X	X	
Portuguese (National)	PORTUGAL.DIC	X		X	X	X	
Portuguese (Brazilian)	BRASIL.DIC	X		X	X	X	
Russian	RUSSIAN.DIC	X		X	X	X	
Spanish	ESPANA.DIC	X		X	X	X	
Swedish	SVENSK.DIC	X	X	X	X	X	
Thai	THAI.DIC						
Turkish	TURKIYE.DIC			X	X	X	

The following abbreviations are used in this table:

DH Dictionary-based hyphenation
PH Preferred hyphenation
AH Algorithic hyphenation
DE Dehyphenation
SYN Display synonyms
DEF Display definitions

Table 7 (Page 1 of 2). Text-Processing Functions by Language—Morphology and Language-Specific Support

Language	Dictionary Name	MID	GIF	IWFM	COMP
Afrikaans	AFRIKAAN.DIC	X	X	X	X
Arabic	ARABIC.DIC	X			
Catalan	CATALA.DIC	X	X	X	
Chinese (Simplified)	CLKCHS.DIC	X			
Chinese (Simplified)	EFLCHS.DIC	X			
Chinese (Traditional)	CLKCHT.DIC	X			
Chinese (Traditional)	EFLCHT.DIC	X			
Czech	CZECH.DIC	X			
Danish	DANSK.DIC	X	X	X	X
Dutch (Government restrictive spelling)	NEDERLND.DIC	X	X	X	X
Dutch (Permissive spelling)	NEDPLUS.DIC	X	X	X	X
English (Australian)	AUS.DIC	X	X	X	
English (United Kingdom)	UK.DIC	X	X	X	
English (United States)	US.DIC	X	X	X	
U.S. English (Computing terms)	US.CPT				
U.S. English (Legal)	US.LEG				
U.S. English (Medical)	US.MED				
U.S. English (Combined U.S. English and medical)	USMED.DIC	X	X	X	
Finnish	SUOMI.DIC	X			
French (National)	FRANCAIS.DIC	X	X	X	
French (Canadian)	CANADIEN.DIC	X	X	X	
German (National pre-reform)	DEUTSCH.DIC	X	X	X	X
German (National reform)	DEUTSCH2.DIC	X	X	X	X
German (DPA rules national reform)	DEUT2DPA.DIC	X	X	X	X
German (Swiss)	DSCHWEIZ.DIC	X	X	X	X
Greek	HELLAS.DIC	X	X	X	
Hebrew	HEBREW.DIC	X			

The following abbreviations are used in this table:

MID Morphological identification
GIF Generate inflected forms
IWFM Inflect word from model
COMP Compound word component isolation

Table 7 (Page 2 of 2). Text-Processing Functions by Language—Morphology and Language-Specific Support

Language	Dictionary Name	MID	GIF	IWFM	COMP
Hungarian	MAGYAR.DIC	X			
Icelandic	ISLENSK.DIC				X
Italian	ITALIANO.DIC	X	X	X	
Japanese	EFLJPN.DIC	X			
Korean	EFLKOR.DIC	X			
Norwegian (Bokmål)	NORBOK.DIC	X	X	X	X
Norwegian (Nynorsk)	NORNYN.DIC	X	X	X	X
Polish	POLSKA.DIC	X			
Portuguese (National)	PORTUGAL.DIC	X	X	X	
Portuguese (Brazilian)	BRASIL.DIC	X	X	X	
Russian	RUSSIAN.DIC	X	X	X	
Spanish	ESPANA.DIC	X	X	X	
Swedish	SVENSK.DIC	X	X	X	X
Thai	THAI.DIC	X			
Turkish	TURKIYE.DIC	X			

The following abbreviations are used in this table:

MID Morphological identification
GIF Generate inflected forms
IWFM Inflect word from model
COMP Compound word component isolation

System Dictionary Content by Language

The following table summarizes the types of data included in each of the Linguistic Tools System Dictionaries:

UHF Ultra-high frequency dictionary

The ultra-high frequency dictionary includes the most frequent words in the language. It is used to speed up spelling verification; on average approximately 50% of the words in general running text may be found in this dictionary.

HF High-frequency dictionary

The high-frequency dictionary is larger than the ultra-high frequency dictionary, but still much smaller than the main dictionary. It also is used to speed up spelling verification, but it contains other kinds of information as well. Up to 85% of the words in general running text may be found in the high-frequency dictionary.

Phon Phonetic spell aid rules

Syn
Synonyms

- X—Base support only
- D—Definitions available with .DEF file (See “Synonym Aid” on page 157 for details.)

Hyph
Hyphenation

- X—Dictionary-based hyphenation
- P—Preferred/ranked hyphenation
- A—Algorithmic hyphenation

Morph
Morphology information

- X—parts of speech, PCODEs, paradigm numbers, grammar masks
- DB—DBPCs (double-byte parse codes) only

BOFA
Flags indicating how the word can be used in compounds

Table 8 (Page 1 of 2). Linguistic Tools Dictionary Content

Language	Dictionary Name	UHF	HF	Phon	Hyph	Syn	Morph	BOFA
Afrikaans	AFRIKAAN.DIC	X	X	X	XPA	X	X	X
Arabic	ARABIC.DIC						X	
Catalan	CATALA.DIC	X	X		XA	X	X	
Chinese (Simplified)	CLKCHS.DIC					X	DB	
Chinese (Simplified)	EFLCHS.DIC						DB	
Chinese (Traditional)	CLKCHT.DIC					X	DB	
Chinese (Traditional)	EFLCHT.DIC						DB	
Czech	CZECH.DIC				A	X	X	
Danish	DANSK.DIC	X	X	X	XPA	X	X	X
Dutch (Government restrictive spelling)	NEDERLND.DIC	X	X	X	XPA	X	X	X
Dutch (Permissive spelling)	NEDPLUS.DIC	X	X	X	XPA	X	X	X
English Australian	AUS.DIC	X	X		XA	X	X	
English United Kingdom	UK.DIC	X	X	X	XA	XD	X	
English* United States	US.DIC	X	X	X	XA	XD	X	
U.S. English (Computing Terms)	US.CPT				X			
U.S. English (Legal)	US.LEG				X			
U.S. English (Medical)	US.MED				X			
U.S. English (Combined U.S. English and Medical)	USMED.DIC	X	X	X	XA		X	
Finnish	SUOMI.DIC				A	X	X	
French (National)	FRANCAIS.DIC	X	X	X	XA	X	X	
French (Canadian)	CANADIEN.DIC	X	X	X	XA	X	X	
German (National pre-reform)	DEUTSCH.DIC	X	X	X	XPA	X	X	X
German (National reform)	DEUTSCH2.DIC	X	X	X	XPA	X	X	X
German (DPA rules National reform)	DEUTSCH2DPA.DIC	X	X	X	XPA	X	X	X

Table 8 (Page 2 of 2). Linguistic Tools Dictionary Content

Language	Dictionary Name	UHF	HF	Phon	Hyph	Syn	Morph	BOFA
German (Swiss)	DSCHWEIZ.DIC	X	X	X	XA	X	X	X
Greek	HELLAS.DIC	X	X	X	XA	X	X	
Hebrew	HEBREW.DIC						X	
Hungarian	MAGYAR.DIC				A	X	X	
Icelandic	ISLENSK.DIC	X	X	X	XA			X
Italian	ITALIANO.DIC	X	X	X	XA	X	X	
Japanese	EFLJPN.DIC						DB	
Korean	EFLKOR.DIC						DB	
Norwegian (Bokmål)	NORBOK.DIC	X	X	X	XPA	X	X	X
Norwegian (Nynorsk)	NORNYN.DIC	X	X	X	XPA		X	X
Polish	POLSKA.DIC				A	X	X	
Portuguese (National)	PORTUGAL.DIC	X	X	X	XA	X	X	
Brazilian Portuguese	BRASIL.DIC	X	X	X	XA	X	X	
Russian	RUSSIAN.DIC	X	X	X	XA	X	X	
Spanish	ESPANA.DIC	X	X	X	XA	X	X	
Swedish	SVENSK.DIC	X	X	X	XPA	X	X	X
Thai	THAI.DIC						X	
Turkish	TURKIYE.DIC				A	X	X	

*For U.S. English only, **Grade Level Information** is also included in the dictionary.

Chapter 5. System Architecture

This chapter describes some of the most important architectural characteristics of the Linguistic Tools.

Linguistic Tools Main Entry Point

The Linguistic Tools has a main interface entry point and many utility functions that assist users of the tools to process the output data. See Chapter 16, “General Utility Functions” on page 211 and Chapter 17, “Token List Utility Functions” on page 229 for details of these utilities. All of the tool entry points use a calling convention that is defined for each environment supported. It is callable from languages other than C—for example, assembler or Java or others—only if the caller can use another language to simulate the C call interface. The Linguistic Tools do not claim to be an SAA service.

Information about building an application program with the Linguistic Tools is given in “Operating System-Specific Information” on page 103.

The entry point name of the Linguistic Tools is **NlpEntry**. To call the Linguistic Tools from a C Language program, the user program uses the statement:

```
NlpEntry(ltcb_p);
```

The argument *ltcb_p* can be replaced by any pointer variable pointing to the Linguistic Tools control block. The Linguistic Tools control block is a contiguous structure containing or pointing to all input and output data areas used for communication between the application and the Linguistic Tools. Some of the key input and output areas pointed to by the LTCB are the following:

- Service Area
- Input data element list
- Reply Area

The basic structure in which Linguistic Tools input and output is passed is shown in Figure 1.

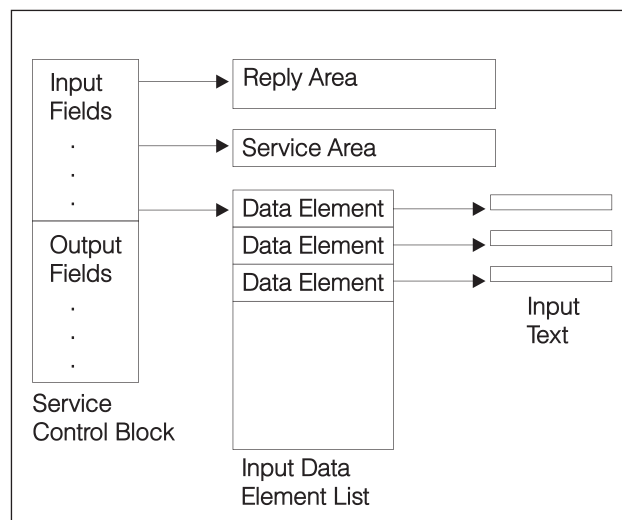


Figure 1. Linguistic Tools Input/Output Structure

© Copyright IBM Corp. 2000

Linguistic Tools Initialization and Termination

In order to use the Linguistic Tools, the application must call the initialization function, which allocates and initializes the Service Area. Upon return from the initialization call, the Linguistic Tools places the Service Area handle into the LTCB; the application must leave this handle in the Linguistic Tools control block on any subsequent tools calls.

There is an explicit termination function, which closes open dictionaries and frees any memory used by Linguistic Tools, including the Service Area. If the application wishes to use Linguistic Tools after termination, it must reinitialize the tools.

If the application calls the initialization function twice without issuing a terminate, it will force the creation of two Service Areas. The application could then pass either Service Area to the Linguistic Tools. When passed one Service Area, Linguistic Tools would not be aware of any processing it did when passed the other Service Area. Linguistic Tools would have to be terminated separately for each Service Area.

Linguistic Tools Input

Various structures handle the Linguistic Tools input.

The Linguistic Tools Control Block

Fixed length input and output data are placed in fields in the Linguistic Tools control block itself, while data that may vary in length are pointed to by the fields in the control block. The control block fields are summarized in the tables on pages 62 through 68. Note, however, that these tables ***do not*** represent the actual structure of the control block. The application should use the LX_CB structure supplied in the include files shipped with the product, rather than attempting to build the control block structure itself.

The field names used in the Linguistic Tools control block follow a regular convention. Fields that contain a pointer have names ending in `_p`. Fields that contain a count have names ending in `_ct`. Fields that contain a flag have names ending in `_f`. An `lx_` prefix precedes the names of all fields accessible by the application. The application should avoid using this prefix before its own variables and constants, to avoid conflicts with current or future field names.

LTCB input fields that are not used by the called function, and all LTCB output fields, may be set to any value by the application. However, the application can allow the Linguistic Tools to perform additional input error checking by setting unused input fields to binary zero. Suppose, for example, that an application mistakenly believes that a field required as input to the invoked Linguistic Tools function is not used. If the application sets the field to zero, and zero is not among the valid values for that field (for example, fields *lx_func_code*, *lx_reply_p*), Linguistic Tools will detect the error and not attempt any processing. If the application were to leave the required input field holding a random value, Linguistic Tools might try to process that random value, with unexpected results.

The Linguistic Tools defines its own data types using C typedef statements. The list below indicates how some of the Linguistic Tools data types correspond to standard C types. These and other data types used by the Linguistic Tools are defined in an include file which is shipped with Linguistic Tools.

- LX_HNLPSERV — unsigned short int
- LX_BOOLBYTE — unsigned char
- LX_UINT — unsigned int
- LX_USHORT — unsigned short int
- LX_PUSHORT — pointer to unsigned short int
- LX_ULONG — unsigned long int
- LX_PULONG — pointer to unsigned long int
- LX_UCHAR — unsigned char
- LX_PUCHAR — pointer to unsigned char
- LX_PVOID — void pointer
- LX_HTOKEN — unsigned short int

The following tables list the control block fields used by different Linguistic Tools functions:

Table 9 General fields used by many functions

Table 10 Fields used only by some of the addenda support functions

Table 11 Fields used only by the spelling support functions

Table 12 Fields used only by the hyphenation functions

Table 13 Fields used only by the thesaurus (synonym aid) function

Table 14 Fields used only by the morphology functions

Table 15 Fields used only by the keyword extraction and query term extraction functions

Table 9 (Page 1 of 3). Linguistic Tools Control Block Fields (General)

Field Name	Field Description	Data Type
lx_version_num	Linguistic Tools version number = 02. Linguistic Tools sets this field for the application's benefit, so that the application may check whether it is using the correct version of Linguistic Tools.	LX_USHORT
lx_func_code	Function code, indicating the function being invoked. • LX_ACTIVATE_DICT — Activate Dictionary • LX_ADDWORD2_ADDENDA — Add Word to Addenda • LX_ADD_MORPH — Add Morphology to Addenda • LX_COMPOUND_WD_ISOL — Compound Word Component Isolation • LX_CREATE_ADDENDA — Create Addenda • LX_DATA_ELE_LIST — Single-word Data Element Creation • LX_DEACTIVATE_DICT — Deactivate Dictionary • LX_DEHYPHENATION — Dehyphenation • LX_EXTRACT_KEYWORD — Extract Keywords • LX_EXTRACT_QRY_TERM — Extract Query Terms • LX_GEN_WORD_INFLECT — Generate Inflected Forms • LX_GRADE_LEVEL — Grade Level Analysis • LX_HYPHENATION — Hyphenation • LX_INFLECT_WORD_FROM_MODEL — Inflect Word from Model • LX_INITIALIZE_NLPS — Initialize Linguistic Tools • LX_ISOLATE_SEGMENT — Text Segment Identification • LX_ISOLATE_WORD — Simple Tokenization with token list output • LX_LEXICAL_ANALYSIS — Lexical Analysis • LX_LIST_ADD_WORD — List Addenda Words • LX_LOOKUP_WORD — Look Up Word • LX_MORPH_ID — Morphological Identification • LX_REMOVE_ADD_WORD — Remove Word from Addenda • LX_SAVE_ADDENDA — Save Addenda • LX_SPELL_AID — Spell Aid • LX_FUZZY_SPELL_AID — Fuzzy Spell Aid • LX_SPELL_VERIFY — Spell Verify and/or Article Checker • LX_SYNONYM_AID — Synonym Aid • LX_TERMINATE_NLPS — Terminate Linguistic Tools	LX_USHORT
lx_rqst_type	Request type • LX_NEW_REQ — New request • LX_CONT_REPLY — Continue reply	LX_BOOLBYTE
lx_serv_area_p	Handle for the Linguistic Tools Service Area, used to hold data to be preserved for the Linguistic Tools between calls. The area is allocated by Linguistic Tools upon initialization, and a handle for it is returned to the application in this field of the LTCB. The application must then ensure that the handle is in this field of the LTCB during subsequent Linguistic Tools calls.	LX_HNLPSERV
lx_dict_names_p	Pointer to a null-terminated string containing the name of a dictionary. Used by the Activate Dictionary, Create Addenda Dictionary, and Save Addenda functions.	LX_PUCHAR
lx_dict_tkns_ct	Number of dictionary handles in the input dictionary list. May be as large as 20.	LX_USHORT
lx_dict_tkns	Input dictionary list. When a dictionary is activated, a dictionary handle is assigned to it and returned by the Activate Dictionary function. Thereafter, the dictionary may be referred to by handle rather than by name. To specify which dictionaries are to be accessed by a function, the handles of the appropriate dictionaries must be placed into the input dictionary list. The list is a twenty-entry array, each entry of which may hold one dictionary handle. The first *lx_dict_tkns_ct* entries are assumed to contain the handles of dictionaries to be accessed.	LX_UCHAR[20]

Table 9 (Page 2 of 3). Linguistic Tools Control Block Fields (General)

Field Name	Field Description	Data Type
lx_lang_code	Code specifying a language to the invoked function. • LX_AFRIKAANS — Afrikaans • LX_ARABIC Arabic • LX_AUSTRALIAN — Australian English • LX_BOK_NORWAY — Norwegian — Bokmål • LX_BRAZ_PORT — Brazilian Portuguese • LX_CAN_FRENCH — Canadian French • LX_CATALAN — Catalan • LX_CHINESE_TRAD — Traditional Chinese • LX_CHINESE_SIMP — Simplified Chinese • LX_CZECH — Czech • LX_DANISH — Danish • LX_DUTCH — Dutch • LX_FINNISH — Finnish • LX_GERMAN — German • LX_GREEK — Greek • LX_HEBREW — Hebrew • LX_HUNGARIAN — Hungarian • LX_ICELANDIC — Icelandic • LX_ITALIAN — Italian • LX_JAPANESE — Japanese • LX_KOREAN — Korean • LX_NAT_FRENCH — French • LX_NAT_PORTGS — Portuguese • LX_NYN_NORWAY — Norwegian — Nynorsk • LX_POLISH — Polish • LX_RUSSIAN — Russian • LX_SPANISH — Spanish • LX_SWEDISH — Swedish • LX_SWISS_GERM — Swiss German • LX_THAI — Thai • LX_TURKISH — Turkish • LX_UK_ENGLISH — U.K. English • LX_US_ENGLISH — U.S. English	LX_USHORT
lx_num_cpg	IBM reference number of the caller's code page. If this number indicates one of a set of code pages recognized by the Linguistic Tools, code-page-related information is obtained from tables stored internally within the Linguistic Tools. If this number indicates a code page not included in the Linguistic Tools default set, the application must supply the appropriate tables through **NlpRegisterCodePage()**.	LX_USHORT
lx_elements_ct	Number of data elements in the data element list provided as input	LX_USHORT
lx_elements_p	Pointer to the data element list provided as input	LX_PELEMENT
lx_elt_format	Format of each data element in the input data element list: • LX_BLOCK_FORMAT — Text block format • LX_SINGLE_WORD_FMT — Single-word format	LX_BOOLBYTE
lx_out_units	Number of units of output (either tokens or text units) desired.	LX_ULONG
lx_rc	Return code	LX_USHORT
lx_reply_p	Reply Area address, pointing to the caller-provided area in which variable-length data is to be returned to the caller.	LX_PUCHAR
lx_reply_size	Reply Area size in bytes.	LX_USHORT
lx_cont_reply_f	Output flag indicating that the request could not be completed, but that the caller may issue a Continue Reply to continue the request: • LX_FALSE — Request completed • LX_TRUE — Request not completed; Continue Reply allowed	LX_BOOLBYTE
lx_reply_used	Space used within the Reply Area, in bytes. For some functions the space used is not necessarily contiguous within the Reply Area.	LX_USHORT

Table 9 (Page 3 of 3). Linguistic Tools Control Block Fields (General)

Field Name	**Field Description**	**Data Type**
lx_abbr_tkns_ct	Number of dictionary handles in the Abbreviation Addenda Dictionary list. May be up to 20.	LX_USHORT
lx_abbr_tkns	Abbreviation Addenda Dictionary list. This list specifies which Abbreviation Addenda Dictionaries are to be used by the Text Segment Identification, Lexical Analysis and Extract Keywords functions. The list is a twenty-entry array, each entry of which may hold the handle of one dictionary. The first *lx_abbr_tkns_ct* entries are assumed to contain the handles of the Abbreviation Addenda Dictionaries to be used.	LX_UCHAR[20]
lx_hFirstToken	Handle of first token in Reply Area produced on the most recent invocation of the Linguistic Tools. This may be passed to the Linguistic Tools token list utility functions.	LX_HTOKEN
lx_first_tkn_p	Pointer to first token in Reply Area produced on the most recent invocation of the Linguistic Tools. This should never be changed.	LX_PUCHAR
lx_next_elemt_p	Address of the data element associated with the next character to be processed in a Continue Reply situation.	LX_PELEMENT
lx_next_char_p	Address of the next character to be processed in a Continue Reply situation.	LX_PUCHAR
lx_delivered_out_units	Number of output units produced on this invocation. This may indicate the number of words returned in the Reply Area excluding the input word, the number of simple tokens broken out of the input text, or the number of output data elements created. Precise meaning varies by function.	LX_ULONG
lx_dict_found_in	Dictionary handle indicating the dictionary in which the input word was found.	LX_UCHAR
lx_unexpected_fn	Name of the function that generated the LX_UNEXPECTED_RC return code. This is useful to the support team for debugging.	LX_UCHAR
lx_unexpected_ln	Line number that generated the LX_UNEXPECTED_RC return code. This is useful to the support team for debugging.	LX_UINT

Table 10. Linguistic Tools Control Block Fields (Addenda Dictionary Support)

Field Name	Field Description	Data Type
lx_conf_res	Flag specifying how to resolve a conflict when a word to be added to an addenda dictionary is already present in that addenda dictionary. • LX_FLAG_ERROR — leave the data for the existing word unchanged, but set the *lx_rc* field of that data element to LX_DUP_WORD • LX_REPLC_DATA — replace the existing data with the new data pointed to by *lx_info_p* • LX_MERGE_DATA — merge the existing data with the new data pointed to by lx_info_p by using the data items in the input data to replace the same data items in the existing data, while leaving the rest of the existing data unchanged	LX_USHORT
lx_word_morph_reply_p	Pointer to reply area for inflection information for word being added to an addenda dictionary by the Add Morphology function.	LX_PUCHAR
lx_alias_morph_reply_p	Pointer to reply area holding inflection information for the alias of a word being added to an addenda dictionary by the Add Morphology function.	LX_PUCHAR
lx_lookup_exact_mt_f	Addenda Lookup Word matching flag: • LX_TRUE if case of addenda word and input word must match exactly • LX_FALSE otherwise.	LX_BOOLBYTE
lx_add_form_f	Addenda dictionary format flag, indicating the format in which the dictionary should be saved to disk by the Save Addenda function. Values are: • LX_SAVE_SAME — save in same format (binary or flat) as file from which this addenda was loaded • LX_SAVE_BINARY — save in binary format only • LX_SAVE_FLAT — save as flat file only	LX_USHORT
lx_add_type_f	Addenda dictionary type flag. Indicates whether the Save Addenda function should save this as a regular, Stopword or Abbreviation Addenda Dictionary. Values are: • LX_SAME_TYPE — save as same type as before • LX_ADDENDA — save as regular Addenda Dictionary • LX_STOPWORD — save as Stopword Addenda Dictionary • LX_ABBREVADD — save as Abbreviation Addenda Dictionary	LX_USHORT
lx_add_lang_f	Addenda dictionary language flag. Indicates the language that should be associated with the dictionary when it is saved. Values are: • LX_SAME_LANG — save as same language as before • Any of the values in the *lx_lang_code* list	LX_USHORT

Table 11. Linguistic Tools Control Block Fields (Spelling Support)

Field Name	Field Description	Data Type
lx_spell_ver_f	Spell verify flag. If the function code is LX_SPELL_VERIFY, this flag determines whether spell verify will be invoked: • LX_FALSE — Do not invoke spell verify • LX_TRUE — Invoke spell verify	LX_BOOLBYTE
lx_art_checker_f	Article checker flag. If the function code is LX_SPELL_VERIFY, this flag determines whether the article checker will be invoked. • LX_FALSE — Do not invoke article checker • LX_TRUE — Invoke article checker	LX_BOOLBYTE
lx_txt_cont_f	Text continuation flag. Indicates whether input text is to be treated as a continuation of text passed previously to the same function; set only on calls to the article checker function. • LX_FALSE — Text is not a continuation • LX_TRUE — Text is a continuation	LX_BOOLBYTE
lx_distance_meas	Maximum acceptable distance measure for candidate words. If the function code is LX_FUZZY_SPELL_AID then the input value in this field is used to select candidate words. The higher the value in lx_distance_meas the more words will qualify as candidates. The default value for this field is zero.	LX_USHORT
lx_comp_aid_f	Compound word component spell aid flag. If spell aid candidates are not found for the original input word, then decompound the word into three pieces (first, middle, and last), and get spell aid candidates by changing the specified component. • LX_TRUE — compound word component spell aid desired • LX_FALSE — compound word component spell aid not desired	LX_BOOLBYTE
lx_auto_aid_f	Automatic or manual compund word component position flag. • LX_TRUE — compound word component to change is determined automatically by the Linguistic Tools • LX_FALSE — compound word component to change is determined by the value of the lx_comp_position field in the control block	LX_BOOLBYTE
lx_comp_position	Determines which compound word component will be changed in the spell aid candidates • 0—simple word, no components • 1—first component • 2—middle component • 3—last component	LX_USHORT

Table 12. Linguistic Tools Control Block Fields (Hyphenation)

Field Name	Field Description	Data Type
lx_alg_hyph_f	Flag indicating whether algorithmic hyphenation rules should be loaded. Set when activating dictionary. • LX_FALSE — Algorithmic hyphenation rules should not be loaded • LX_TRUE — Algorithmic hyphenation rules should be loaded	LX_BOOLBYTE
lx_alg_only_f	Flag indicating whether algorithmic hyphenation rules are to be used exclusively: • LX_FALSE — algorithmic hyphenation used in conjunction with dictionary-based hyphenation • LX_TRUE — algorithmic hyphenation used exclusively	LX_BOOLBYTE
lx_chars_left	Number of characters remaining on line. Set for the hyphenation function.	LX_USHORT
lx_pref_hyphen_f	Flag specifying whether preferred hyphenation points are to be selected. • LX_FALSE — preferred hyphenation not desired • LX_TRUE — preferred hyphenation desired	LX_BOOLBYTE
lx_pref_range	Size of zone within which preferred hyphenation points are to be selected	LX_USHORT

Table 13. Linguistic Tools Control Block Fields (Thesaurus)

Field Name	Field Description	Data Type
lx_syn_pos	A mask that specifies to the synonym function the parts of speech that the returned candidates may be.	LX_UCHAR
lx_syn_inflect_f	Flag specifying whether synonyms are to be returned in the same inflection as the input word. • LX_FALSE — Return synonyms as stored in dictionary. This is usually, but not always, the base form of the input word. • LX_TRUE — Return synonyms in the same inflection as the input word.	LX_BOOLBYTE
lx_syn_outtype	Code specifying the type of output desired: • 0 — (default) Output synonyms. Also, output definitions if available, but do not output definitions that have no synonyms. • 1—Output definitions only. • 2—Output synonyms only. • 3—Output definitions and synonyms. Not all definitions will be associated with synonyms. This option is a combination of option 1 and 2.	LX_USHORT

Table 14. Linguistic Tools Control Block Fields (Morphology)

Field Name	Field Description	Data Type
lx_morph_mask_inp	Identifies to the Generate Inflected Forms function the inflection for which forms of the input word are desired.	LXT_PBITMASK
lx_morph_form_id	Identifies to the Generate Inflected Forms function which form of the input word is desired.	LX_USHORT
lx_alias_len	Length in bytes of word used as morphological alias. Set if an alias is being used with the Morphological Identification and Generate Inflected Forms functions.	LX_USHORT
lx_alias_p	Pointer to alias word. Set if an alias is being used with the Morphological Identification and Generate Inflected Forms functions.	LX_PUCHAR
lx_morph_pcode	Identifies to the Generated Inflected Forms function the PCODE for which forms of the input word are desired.	LX_UCHAR[7]
lx_morph_pos	Identifies to the Generated Inflected Forms function the parts of speech that returned forms of the input word may be.	LX_UCHAR

Table 15 (Page 1 of 2). Linguistic Tools Control Block Fields (Text Extraction)

Field Name	Field Description	Data Type
lx_stopw_f	Flag specifying whether stopwords in the input text are to be ignored by the Extract Keywords and Extract Query Terms functions. • LX_FALSE — Do not ignore stopwords. • LX_TRUE — Ignore stopwords.	LX_BOOLBYTE
lx_stopw_tkns_ct	Number of dictionary handles in the Stopword Addenda Dictionary list. May be up to 20.	LX_USHORT
lx_stopw_tkns	Stopword Addenda Dictionary list. This list specifies which Stopword Addenda Dictionaries are to be used by the Extract Keywords and Extract Query Terms functions. The list is a twenty-entry array, each entry of which may hold the handle of one dictionary. The first *lx_stopw_tkns_ct* entries are assumed to contain the handles of the Stopword Addenda Dictionaries to be used.	LX_UCHAR[20]
lx_pos_f	Flag specifying the parts of speech to be returned as keywords or query terms: • LX_XTRACT_POS_ALL — Return all parts of speech. • LX_XTRACT_POS_NOUNS_ONLY — Return nouns only. • LX_XTRACT_POS_NOUNS_ADJS — Return nouns and adjectives only. • LX_XTRACT_POS_NOUNS_VERBS — Return nouns and verbs only. • LX_XTRACT_POS_NOUNS_VERBS_ADJS — Return nouns, verbs, and adjectives only. • LX_XTRACT_JAIRS — JAIRS part of speech filtering for Japanese support. (words whose JPOS value is one of the following: 13, 18, 19, 23, 24, 97, 102, 103, 104, 106, 107. See Table 32 on page 325.) • LX_XTRACT_FREQ — Return Chinese frequency data.	LX_BOOLBYTE
lx_sentence_f	Flag specifying whether sentence determination is to be performed by the Extract Keywords function. • LX_FALSE — Sentence determination is not done. • LX_TRUE — Sentence determination is done.	LX_BOOLBYTE
lx_sent_begin_f	Flag specifying whether special processing should be done during dictionary lookup of the first word in a sentence. • LX_FALSE — Do not do any special processing for start of sentence. • LX_TRUE — Do special processing for start of sentence.	LX_BOOLBYTE

Table 15 (Page 2 of 2). Linguistic Tools Control Block Fields (Text Extraction)

Field Name	Field Description	Data Type
lx_phonetic_f	Flag specifying whether phonetic keys are to be returned for keywords found by the Extract Keywords and Extract Query Terms functions. • LX_FALSE — Phonetic keys not desired. • LX_TRUE — Phonetic keys desired.	LX_BOOLBYTE
lx_wordde_f	Flag specifying whether decompositions of Germanic compound words found by the Extract Keywords and Extract Query Terms functions are to be returned by those functions. • LX_FALSE — Decompositions not desired. • LX_TRUE — Decompositions desired.	LX_BOOLBYTE
lx_decompnd_further_f	Flag specifying whether to decompose further the components of a compound word: • LX_FALSE — New decompositions cannot be subpieces of earlier decompositions; that is, every decomposition is unique. • LX_TRUE — All possible decompositions will be given.	LX_BOOLBYTE
lx_word_mod_f	Flag specifying whether words returned by the Extract Keywords and Extract Query Terms functions are to be normalized: • LX_FALSE — Do not modify returned words. • LX_TRUE — Normalize returned words.	LX_BOOLBYTE
lx_wordre_f	Flag specifying whether base forms (lemmas) are to be returned for words found by the Extract Keywords and Extract Query Terms functions. • LX_FALSE — Base forms not desired. • LX_TRUE — Base forms desired.	LX_BOOLBYTE
lx_wordex_f	Flag specifying whether inflected forms are to be returned for query terms found by the Extract Query Terms function. • LX_FALSE — Inflected forms not desired. • LX_TRUE — Inflected forms desired.	LX_BOOLBYTE
lx_synonym_f	Flag specifying whether synonyms are to be returned for query terms found by by the Extract Query Terms function. • LX_FALSE — Synonyms not desired. • LX_TRUE — Synonyms desired.	LX_BOOLBYTE

Service Area

This area, referred to by the handle in field *lx_serv_area_p* in the Linguistic Tools control block, contains Linguistic Tools-owned data to be kept intact between calls to the Linguistic Tools. The application should never allocate, free, touch, or read the area. The Service Area is allocated by the Linguistic Tools on initialization, and is freed by the Linguistic Tools on termination. The following items are among those stored in the Service Area:

- Current state
- Data or pointers to data needed to complete a continue reply
- Active dictionary list

Input Data Element List

This list specifies text input to Linguistic Tools functions that process text. It is a contiguous list of data elements, each containing information about a portion of the input text. Linguistic Tools assumes that the data elements are listed in the same order as the text they reference.

The application must use one of two data element formats; all data elements in the same data element list must be in the same format. In the single-word format, each data element contains a pointer to a single word or fragment thereof. The single-word format element itself contains the *trailing delimiter* character that follows the word (or word fragment) in the text from which it came. The application must also indicate within each single-word format element's *lx_input_f* field whether the element is to be combined with the next element, and if so, how. How single-word format elements are used is described in "Use of Single-Word Format Data Elements" on page 71.

In the block-format data element list, each data element either points to a contiguous block of input text containing any number (other than zero) of words or characters, or specifies a special character such as a newline. Each block format element specifies this type by the *lx_type* field of the element. Note that control characters, such as newline, formatting codes, and so on should not be present in the data of a text-type data element.

Which data element format is appropriate depends on the function, although for some functions the application may choose either format. The application indicates the format of the input data element list and the number of data elements provided via fields in the LTCB.

Each data element contains various output as well as input fields. The input text and input fields in the input data element list will not be altered by the Linguistic Tools, so an editor may point the pointers in the data element list directly into the document to be processed.

The Linguistic Tools has two kinds of functions that accept textual input:

1. Single word functions

   Single word functions process only the first input word passed via the input data element list. The data element list must be in single-word format. The following functions are single word functions:

   - look up word
   - spell aid
   - fuzzy spell aid
   - hyphenation
   - dehyphenation
   - synonym aid
   - morphological identification
   - inflect word from model
   - generate inflected forms
   - compound word component isolation

2. Multiple word functions

   Multiple word functions process any number of words at a time. The following multiple word functions accept text input in single-word format elements:

   - add word to addenda dictionary
   - remove word from addenda dictionary
   - spell verify
   - article checking
   - grade level analysis

The following multiple word functions accept text input in block format elements:

- spell verify
- article checker
- grade level analysis
- text segment identification
- simple tokenization
- lexical analysis
- extract keyword
- extract query term
- single-word data element creation

Note that the spell verify, grade level analysis, and article checker functions accept either single-word or block format data elements.

Use of Single-Word Format Data Elements

This tells the application how to pass text to the Linguistic Tools using a single-word format data element list. A description is also given of how the Linguistic Tools sets certain data element fields when it creates a single-word format data element list. The rules governing how an application may pass data through a single-word format list are followed by Linguistic Tools when it creates such a list.

Field *lx_data_p* of the Data Element

This field points to the text of the element. Just how much text is pointed to by the field is indicated by field *lx_data_len* of the element.

The *lx_data_p* field of a single-word format data element may point to a single word or to a portion of a single word. "Definition of a Word in SBCS Text" on page 80 explains how the Linguistic Tools defines a word.

A single word may be split across multiple single-word format elements if necessary. The text pointed to by the *lx_data_p* field of a single-word format data element should never contain delimiter characters.

Combining of Data Elements

The combining or joining of data elements refers to the treatment of multiple consecutive single-word format elements as one element. The elements involved are not actually altered; the Linguistic Tools merely behaves as if they had been replaced by one element. A data element's *lx_input_f* field will be set to LX_ELMT_JOIN if the element is to be joined to the next one. Multiple consecutive elements are joined together if all but the last one have this setting. The text of the resulting combined element is the concatenation of the text pointed to by the *lx_data_p* fields of the constituent elements. A required hyphen is inserted after each element with *lx_input_f* set to LX_ELMT_REQ and LX_ELMT_JOIN. Any hyphenation-related spelling changes at the point of concatenation are reversed after each element with *lx_input_f* set to LX_ELMT_SYL and LX_ELMT_JOIN. The *lx_trail* field of the last constituent element becomes the combined element's trailing delimiter. Elements which have *lx_input_f* set to LX_ELMT_NO_HYPHEN and LX_ELMT_JOIN will be treated as if the application is unsure whether or not the elements should be joined. This is used when no hyphen is present at the end of a line to indicate whether the following line is a continuation of the same word or not.

Following are the only legal combinations of a data element's input flags in the *lx_input_f* field. LX_ELMT_JOIN indicates whether joining may occur at all. The LX_ELMT_SYL and LX_ELMT_REQ flags indicate how it is to occur.

- No flags set—Indicates that the element is not to be joined to the next element, although it may be joined to the previous element if the previous element's input flags are set accordingly.
- LX_ELMT_JOIN — Indicates that the element is to be joined to the next element, and that the text of the element is to be concatenated with the text of the next element. For example, if the two word fragments "bl" and "ue" are passed to Linguistic Tools by two consecutive data elements, and the first element's *lx_input_f* field is set to LX_ELMT_JOIN, Linguistic Tools behaves as if passed one element pointing to "blue". This would be useful if the word "blue" contained a control character that the application did not want the Linguistic Tools to see.
- LX_ELMT_JOIN and LX_ELMT_SYL — Same meaning as when only LX_ELMT_JOIN is set, except that any hyphenation-related spelling changes detected at the point of text joining are reversed. This allows a word hyphenated across the end of a line to be passed to the Linguistic Tools and processed in its unhyphenated state. For example, the pre-reform German word "Acker" is hyphenated as "Ak-" and "ker". If these two hyphenated pieces are passed to the Linguistic Tools via two data elements, with the first element's *lx_input_f* field set to LX_ELMT_JOIN and LX_ELMT_SYL, the Linguistic Tools acts as if passed one element pointing to "Acker".
- LX_ELMT_JOIN and LX_ELMT_REQ — Indicates that the element is to be joined to the next element and a required (hard) hyphen inserted where the text strings of the two elements are concatenated. This allows a word containing a required hyphen, usually a delimiter and not permitted in single-word format text, to be passed to the Linguistic Tools.

  For example, if the two words "able" and "bodied" are passed to the Linguistic Tools via two consecutive data elements, and the first element's *lx_input_f* field is set to LX_ELMT_JOIN and LX_ELMT_REQ, the Linguistic Tools will behave as if passed one element pointing to "able-bodied".
- LX_ELMT_JOIN and LX_ELMT_NO_HYPHEN — Indicates that the application is unsure of whether or not to join this and the following element. Linguistic Tools will try to determine whether or not the elements should be joined or not. This is currently used in DBCS languages where words may possibly wrap to the following line and no hyphenation points are specified.
- LX_ELMT_JOIN and LX_ELMT_SYL and LX_ELMT_REQ — Currently, this is treated as if the *lx_input_f* field were set to LX_ELMT_JOIN and LX_ELMT_REQ.
- LX_DB_FIRST — Used if the first character of the text is host DBCS. This flag can occur with any of the above combinations.

Combining More than Two Data Elements

If a group of contiguous elements all have *lx_input_f* set to LX_ELMT_JOIN, all the elements in the group, plus the element following the group, are together treated as one element by Linguistic Tools. Whether a given element's *lx_input_f* field has LX_ELMT_SYL and/or LX_ELMT_REQ set determines what happens where the text of that element meets the text of the next element.

Output Describing Combined Elements

Output information regarding a word is returned only in the first element associated with the word, unless the invoked function is spell verify or Korean morphology. Spell verify returns information in each data element, since the element making up a word may be verified separately. When passed a word containing required hyphens, spell verify first tries to verify each piece of the word between two hyphens or between a hyphen and either end of the word. If any of the pieces fails to verify, the entire word is then verified. Ultimately each piece is considered correctly spelled if either it or the entire word verifies as correct; a piece is marked as correct by setting the return code of each element making up the piece to LX_RC_OK.

In the Korean language, it is not clear whether or not a word at the end of a line continues on the following line or whether the words are to be separated. No hyphenation indicators are used. For this case, the application may set the LX_ELMT_JOIN and LX_ELMT_NO_HYPHEN flags for the morphological identification function. Return codes are put in all elements to indicate what part of the word was used to generate the given output. (See Appendix D, “Language-Specific Processing” on page 281 for details.)

Punctuation Filter Flag

During single-word data element creation, the process does not convert punctuation to data elements unless the punctuation is part of the word, as with an apostrophe. Data Element Creation tells the caller when some punctuation has been filtered by setting LX_ELMT_REMOVED in the *lx_input_f* field of the data element that follows the point where the punctuation was filtered. This flag is set for continue replies and new requests based on the last input seen.

Linguistic Tools-generated Single-Word Format Data Elements

Single-word format data elements created by functions accepting block format input, such as data element creation, spelling verification, and article checking, generally consist of one word per element, per the definition of a word given in “Definition of a Word in SBCS Text” on page 80. The only exception occurs when a single word is broken across two input block format elements, in which case it must be split across two output single-word format elements as well. The circumstances under which each possible *lx_input_f* flag combination is used by the Linguistic Tools are as follows:

- No flags set—Used when none of the conditions below are met.
- LX_ELMT_JOIN only
  1. Used when the element points to part of a word continued in the next element; such a word can only have been broken across two input block format elements; or
  2. Used when the element points to a word followed by a syllable hyphen in turn followed by another word, with no interceding characters; thus the Linguistic Tools will not be aware a syllable hyphen was ever present when passed the resulting single-word format element list.
- LX_ELMT_JOIN and LX_ELMT_SYL — Used when the text pointed to by the element is the first part of a normally non-hyphenated word broken across the

end of a line and so marked with one or more syllable hyphen characters. A syllable hyphen must immediately follow the first part of the word.

- LX_ELMT_JOIN and LX_ELMT_REQ — Used when a normally hyphenated word (such as “able-bodied”) is broken across the end of a line at the hyphen and is so marked with both required and syllable hyphen characters. The two hyphens may be together immediately after the first part of the word, or one may immediately follow the first part of the word and the other immediately precede the second part. Additional syllable hyphens appearing between the two parts may be present but will be ignored; only one required hyphen may be present.

  This flag combination is also used in an element pointing to a word followed by a required hyphen, in turn followed by another word, with no interceding characters.

- LX_ELMT_JOIN and LX_ELMT_NO_HYPHEN — Currently this legal combination is not generated by functions that accept block format input.
- LX_ELMT_JOIN and LX_ELMT_SYL and LX_ELMT_REQ — Currently the Linguistic Tools behaves as though only LX_ELMT_JOIN and LX_ELMT_REQ were set.
- LX_DB_FIRST — Used if the first character of the text is host DBCS. This flag may occur with any of the above combinations.

Block Format Data Elements

Block format data elements may be of the following *types*:

Text
Text elements point to text to be processed by the Linguistic Tools. Each text data element must have a text length of at least one. Text elements of length 0 will be ignored.

New line, new paragraph, or new sentence
New line, new paragraph, or new sentence elements contain no data but are used by the Text Analysis Processor heuristics which identify sentence boundaries.

User (private) data
User data elements are basically ignored by the Linguistic Tools. However, the presence of the User data element is regarded as an implicit delimiter between text of Text data elements. Whereas text in two consecutive Text data elements is considered to be logically contiguous, the Linguistic Tools does not consider the text of the Text data elements as logically contiguous if one or more User data elements intervene.

Data Element Structure

Each data element is structured as indicated below. Data types are defined in one of the include files shipped with the Linguistic Tools.

Field Name	Field Description	Data Type
lx_type	Type of element This field is used only in block format elements. Codes and possible types are as follows: LX_TEXT — Text LX_NLIN — New line LX_NSEN — New sentence LX_PARA — New paragraph LX_UDAT — User (private) data LX_IGNORE — User (private) data	LX_UCHAR
lx_input_f	Input flag. This field is used in single-word format elements or for initialization of host DBCS input. Possible values are: • LX_ELMT_JOIN — Element is to be joined to the next element • LX_ELMT_SYL — Element is to be joined to the next element and hyphenation spelling changes at the juncture are to be undone; ignored if not also LX_ELMT_JOIN • LX_ELMT_REQ — Element is to be joined to the next element and a required hyphen inserted at the juncture; ignored if not also LX_ELMT_JOIN • LX_ELMT_NO_HYPHEN — The Linguistic Tools is to determine if the current element and the following element are to be joined. No hyphenation information is specified; ignored if not also LX_ELMT_JOIN If LX_ELMT_JOIN, LX_ELMT_SYL and LX_ELMT_REQ are specified, the Linguistic Tools will behave as if only LX_ELMT_JOIN and LX_ELMT_REQ were set. • LX_DB_FIRST — If all of the following are true: – The input code page is a HOST DBCS code page – The first character of the first text element in the data element list is a DBCS character – It is the first call to one of: Text Segment Identification, Lexical Analysis, Extract Keyword, or Extract Query Term Data Element Creation will set LX_ELMT_REMOVED when the text contains some punctuation preceding the current word (represented by the current data element) that did not appear as a data element. This is for the benefit of article checker.	LX_UCHAR
lx_data_len	Length in bytes of data pointed to by this element. This field is required only for single-word format elements and for block format elements of type text. For text elements, this must be non-zero. Zero-length text elements are ignored.	LX_USHORT
lx_data_p	Pointer to logically first character of the text data. This field is always required. Set to NULL for non-text data.	LX_PUCHAR

Field Name	Field Description	Data Type
lx_info_p	Pointer to information to be associated with the word pointed to by lx_data_p. This field only applies to single-word format data elements, because it is only used by the **Add Word to Addenda Dictionary** function. The information is structured as follows: • AbbrEOS (type LX_USHORT) — flag: LX_TRUE indicates that the word is an abbreviation which may end a sentence. LX_FALSE indicates that the word is an abbreviation which may not end a sentence or that the word is not an abbreviation. • HyphLength (type LX_USHORT) — length of the Hyphenation data. • HyphData (type LX_PUCHAR) — pointer to the Hyphenation data, represented by a copy of the word from lx_data_p with all hyphenation points indicated. If this pointer is NULL and the word pointed to by lx_data_p contains hyphenation, the hyphenation points from lx_data_p will be used. If this pointer is NULL, and the word pointed to by lx_data_p contains no hyphenation, no hyphenation points will be stored. If this pointer is non_NULL, it will override whatever hyphenation may be present in the word pointed to by lx_data_p. • MorpLength (type LX_USHORT) — length of the Morphology Alias. • MorpData (type LX_PUCHAR) — pointer to the Morphology Alias. Only a single word is allowed; surrounding white space is ignored. • DBPCLength (type LX_USHORT) — length of the DBCS parse code data. • DBPCData (type LX_PUCHAR) — pointer to the DBCS parse code data. Multiple DBPCs are allowed, each separated by a comma; surrounding white space is ignored. • SynLength (type LX_USHORT) — length of the Synonym data. • SynData (type LX_PUCHAR) — pointer to the Synonym data. Multiple synonyms are allowed, each separated by a comma; white space is ignored before or after the comma. Each synonym may be a word or a phrase. • UserLength (type LX_USHORT) — length of the User-defined data. • UserData (type LX_PUCHAR) — pointer to the User-defined data. Any additional data to be stored with the word. White space is not ignored.	LX_PADDINFO
lx_trail	For single-word format data elements, this is the character following the input word in the text from which it came (in the code page of the input text). For DBCS input, *lx_trail* must be set to `\0`.	LX_UCHAR
lx_rc	For some functions this field contains output information regarding the processing of the input word.	LX_USHORT
lx_databyte	For some functions this field contains output data describing the input word.	LX_UCHAR

Table 16. Data Element Structure

Unused fields should be set to zero.

Reply Area

This area is used by some Linguistic Tools functions for the return of variable-length output. These functions expect the application to allocate the Reply Area and to place its address and size in the Linguistic Tools control block. See "Reply Area" on page 86 for more information.

Linguistic Tools Output

Linguistic Tools output is placed in the Linguistic Tools control block or in areas anchored to it. Most fixed length output is placed into fields at the end of the control block itself.

The Linguistic Tools Control Block

The control block fields are summarized in tables 9 through 15 on pages 62 through 68.

Service Area

The Service Area is an output area only in the sense that it is returned to the application; the area should never be read or modified by the application. The application is returned a handle for the Service Area in the Linguistic Tools control block by the Initialization function. The application must pass that handle back to the Linguistic Tools in the *lx_serv_area_p* field of the control block on each subsequent Linguistic Tools call.

Data Element List

Some Linguistic Tools functions that accept a data element list as input write output into output fields of each input element, without altering the input fields or text of the element. A few fields in each input data element are reserved for output describing the input text referenced by the element.

Some Linguistic Tools functions build an entire data element list within the Reply Area. This list is not the same as the input data element list, which is built by the application. The text pointers in such a list, however, will point into the same text referenced by the input data element list.

Reply Area

Variable-length output data from the Linguistic Tools is returned in the Reply Area, which is allocated by the application. Items that may be placed into the Reply Area are a data element list built by Linguistic Tools, a candidate list from spell aid, synonyms, and so on See “Reply Area” on page 86 for more information.

Code Pages

In order to perform certain functions, the Linguistic Tools needs information about the code page of the input text, in the form of code-page-specific tables.

The Linguistic Tools internally stores all of the necessary code-page-related tables for each of the most common code pages. If the input text is in one of these common code pages, the application may put the IBM reference number of that code page into the *lx_num_cpg* field of the Linguistic Tools control block. If the input text is in a code page not included in the Linguistic Tools default set listed below, the application must supply the appropriate information through the **NlpRegisterCodePage()** function.

By default, Linguistic Tools will assume at initialization that all input text and dictionaries will be in code page 850 (on the workstation), 500 (on the host OS/400 or OS/390 OpenEdition), or 1275 (on the Macintosh). If the application wishes to select another code page as the active code page, it must call a function which requires it to specify a code page number. If the desired code page is not one of the Linguistic Tools default set, the application must call **NlpRegisterCodePage()** before calling any other function that requires code page information; otherwise, any function for which a code page may be specified (generally by means of the *lx_num_cpg* field in the Linguistic Tools control block) will not work.

Code Page Restrictions

Note that **NlpRegisterCodePage()** will accept only 256-byte code pages, that is, single-byte character sets. For the double-byte character sets, an application ***must*** use the code pages supported by the Linguistic Tools.

Also note that other restrictions apply to Linguistic Tools' use of the code pages. These restrictions are included in the following list:

- Only the data in the elements can be included in a codepage other than the default. For example, dictionary names, which are used for activation, *must* be in codepage 850 (workstation), 500 (host), or 1275 (Mac).
- Unicode data *must* be in big-endian format.
- Unicode is not supported in the addenda dictionaries. That is, an addenda dictionary cannot be saved as a flat file in the Unicode codepage.
- When using the Unicode codepage, the language code (*lx_lang_code*) field of the control block must be specified. If it is not specified, then results can be unpredictable.

Code Pages Supported

Listed below, by language group and system, are the code pages for which the Linguistic Tools internally stores code-page-related tables.

Table 17. Code Pages Supported

Language Group	IBM EBCDIC	IBM ASCII	ISO 8859-x	MS Windows	Apple Macintosh
Latin 1[*]	LX_CP_LATIN_HOST 500	LX_CP_LATIN_PC 850	LX_CP_LATIN_ISO (ISO 8859-1) 819	LX_CP_LATIN_WIN 1252	LX_CP_LATIN_MAC 1275
Note: [*]Latin 1 includes these languages: Afrikaans, Catalan, Danish, Dutch, English, Finnish, French, German, Icelandic, Italian, Norwegian, Portuguese, Spanish, and Swedish.					
Latin 2 (Czech, Hungarian, Polish)	LX_CP_LATIN2_HOST 870	LX_CP_LATIN2_PC 852	LX_CP_LATIN2_ISO (ISO 8859-2) 912	LX_CP_LATIN2_WIN 1250	LX_CP_LATIN2_MAC 1282
Latin 5 (Turkish)	LX_CP_TURKISH_HOST 1026	LX_CP_TURKISH_PC 857	LX_CP_TURKISH_ISO (ISO 8859-2) 920	LX_CP_TURKISH_WIN 1254	LX_CP_TURKISH_MAC 1281
Latin/Cyrillic (Russian)	LX_CP_CYRILLIC_HOST 1025	LX_CP_CYRILLIC_PC 866	LX_CP_CYRILLIC_ISO (ISO 8859-2) 915	LX_CP_CYRILLIC_WIN 1251	LX_CP_CYRILLIC_MAC 1283
Latin/Arabic	LX_CP_ARABIC_HOST 420	LX_CP_ARABIC_PC 864	LX_CP_ARABIC_ISO (ISO 8859-2) 1089	LX_CP_ARABIC_WIN 1256	N/A
Latin/Greek	LX_CP_GREEK_HOST 875	LX_CP_GREEK_PC 869	LX_CP_GREEK_ISO (ISO 8859-2) 813	LX_CP_GREEK_WIN 1253	LX_CP_GREEK_MAC 1280
Latin/Hebrew	LX_CP_HEBREW_HOST 424	LX_CP_HEBREW_PC 862	LX_CP_HEBREW_ISO (ISO 8859-2) 916	LX_CP_HEBREW_WIN 1255	N/A
Latin/Thai	LX_CP_THAI_HOST 838	LX_CP_THAI_PC 874	LX_CP_THAI_PC (ISO 8859-11) 874	LX_CP_THAI_PC 874	N/A
Chinese, simplified	LX_CP_S_ CHINESE_HOST 935	LX_CP_S_ CHINESE_PC 1381	N/A	N/A	N/A
Chinese, traditional	LX_CP_T_ CHINESE_HOST 937	LX_CP_T_ CHINESE_PC 948 LX_CP_T_ CHINESE_BIG5 (BIG-5) 950	N/A	N/A	N/A
Japanese	LX_CP_ JAPANESE_HOST 939	LX_CP_ JAPANESE_PC1 932 LX_CP_ JAPANESE_PC2 942	N/A	N/A	N/A
Korean	LX_CP_KOREAN_HOST 933	LX_CP_KOREAN_PC 944 LX_CP_KOREAN_KS 949	N/A	N/A	N/A
Unicode (UCS-2)	LX_CP_UCS2 13488	LX_CP_UCS2 13488	LX_CP_UCS2 13488	LX_CP_UCS2 13488	LX_CP_UCS2 13488

Definition of a Word in SBCS Text

The Linguistic Tools defines a word as a contiguous string of **non-delimiter** characters. Characters that may not appear in words (and therefore delimit them) are called **delimiter** characters. Linguistic Tools' **Simple Tokenization** function determines word boundaries in SBCS text by classifying the conditions under which specific characters act as word delimiters. Alphanumeric characters never act as delimiters. All other characters are regarded as delimiters except the following:

Table 18. Linguistic Tools Delimiter Characters

Character	Exception condition
Catalan high-dot (·) Apostrophe/Single-quote (') **Note:** Apostrophe is double-quote (") where language is Hebrew Accent acute (´)	Not a delimiter if surrounded by alphabetics
Pound Sterling Sign (£) Yen sign (¥) EuroCurrency Sign Thai Baht Sign Yuan Sign Dollar sign ($)	Not a delimiter if preceded by a punctuation character and followed by a numeric
Exclamation Point (!) Asterisk (*) Percent sign (%) Question mark (?) Number Sign (#) Equals sign (=)	Not a delimiter if preceded by the same character.
Hard hyphen (-) Slash character (/) Colon (:)	Not a delimiter if preceded by the same character or if surrounded by numerics.
Comma (,) Period (.)	Not a delimiter if preceded by the same character or if surrounded by numerics or if preceded by punctuation and followed by a numeric.

The absence of a character is treated as a blank, for example, the last character in a string is considered to be followed by a blank.

This definition of a word is used by all functions except **Lexical Analysis** which does further analysis of forms containing slashes, hyphens, and apostrophes.

Several utility functions are available for the application to use in examining and, if necessary, altering the way in which Linguistic Tools determines word boundaries:

- **NlpIsDelimiter()** — indicates whether a specific character is regarded as a delimiter in the current context
- **NlpQryDelimTable()** — may be used to view the current default delimiter table
- **NlpSetDelimTable()** — may be used to change the default delimiter table so that particular characters behave differently in determining word boundaries

Repeatable Punctuation

Characters in the last three groups (from exclamation point down) in Table 18 on page 80 may be repeated in text. That is, a text may contain a series of these, as shown in the following example:

That's great!!!
What???
from here...to there

In these examples only the first of the repeated punctuation characters actually marks a word boundary. The others are not considered to be delimiters. The *fWordBoundary* flag is provided in the **NlpIsDelimiter()** function to help determine whether a particular instance of one of these repeatable punctuation characters should be identified as a word boundary.

Sample Words

Each of the following strings would be considered a single word by Linguistic Tools:

- book
- don't
- $10,223.95
- mother-in-law
- 10/22/90

Definition of a Word in DBCS Text

In the DBCS languages supported for Linguistic Tools (Chinese, Japanese, and Korean) a word consists of one or more DBCS text characters: *hanzi, kanji, hiragana, katakana, hangeul*, or DBCS Latin alphabetic characters. *Katakana* transliterations of non-Japanese words may contain center dots (GCGID JQ740080) to indicate word boundaries within the string being transliterated. Only DBCS Latin words may contain numerics, hyphens, periods, and/or apostrophes, either DBCS or SBCS.

For these languages Linguistic Tools' **Simple Tokenization** function separates SBCS text from DBCS text and applies the criteria listed above to isolate words within the SBCS portion only. **Lexical Analysis** must be called in order to identify the words in DBCS text.

Case Matching between Input Words and Dictionary Words

The following section applies only to SBCS text. In DBCS text, the input word always must match the dictionary word exactly.

There are four possible combinations of upper- and lower-case letters in any word:

- all lowercase
- all uppercase
- first letter only uppercase, rest lowercase (initial capital)
- mixed case (at least one letter other than the first is uppercase, while other letters in the word are lowercase).

Linguistic Tools will allow an all-lowercase dictionary word to match an input word which is all-lowercase, all-uppercase, or which has an initial capital. An all-uppercase dictionary word will match only an all-uppercase input word, while a word which has an initial capital letter in the dictionary can match either an all-uppercase input word or one which has only the initial capital. Mixed case words must match exactly, except when the Mixed Mode flag has been set (see "NlpMixedMode" on page 217 for details). An input word which has the correct letters in the correct order, but which does not fit the case pattern(s) allowed for the dictionary word, will result in a mismatch.

Table 19. Linguistic Tools Default Case Matching

	All-lowercase Dictionary Word	**Initial Capital Dictionary Word**	**All-uppercase Dictionary Word**	**Mixed Case Dictionary Word**
All-lowercase Input Word	All-lowercase output	No match	No match	No match
Initial Capital Input Word	Initial capital output	Initial capital output	No match	No match
All-Uppercase Input Word (MM off)*	All-uppercase output	All-uppercase output	All-uppercase output	No match
All-Uppercase Input Word (MM on)*	All-uppercase output	All-uppercase output	All-uppercase output	All-uppercase output
Mixed Case Input Word	No match	No match	No match	Exact match only

*Mixed Mode flag set by **NlpMixedMode()**.

Some letter sequences may appear in the dictionary in more than one case pattern—for example, "may" (the English auxiliary verb) and "May" (the month). If Linguistic Tools receives the input word "May", it will find both "may" and "May" as possible matches (because both the all-lowercase dictionary word and the one with the initial capital may match an input word with an initial capital). Because "May" is an exact match, that is the one that will be returned. Preference is always given to an exact match. However, if there were no word "May" in the dictionary, then "may" would be returned as the best match.

In some languages, accents may be stripped when a letter is uppercased, or the form of the letter may change. Linguistic Tools makes allowance for this—Appendix D, "Language-Specific Processing" on page 281 describes what is done for each language.

The **Spell Verify** function will flag as misspelled all words which do not match according to the pattern described above.

Spell Aid will attempt to find candidates for any single-word-delimited input string that does not contain invalid characters. The candidates returned by this function generally will match the case pattern of the input word. An all-lowercase input word will receive all-lowercase spell-aid candidates, unless the dictionary word has an initial capital, is all-uppercase or mixed case, in which event the dictionary form is returned. An all-uppercase input word will always receive all-uppercase candidates.

With initial capital and mixed case input words, the case pattern of the returned candidates depends on two factors:

1. The case pattern of the dictionary word
2. The case of the initial letter of the input word.

If the dictionary word is all-lowercase, whether the candidates will have initial capitals or be all-lowercase is determined by the case of the initial letter of the input word. If the initial letter is uppercase, the candidates will have initial capitals—for example, spell aid for “Testr” or “TesTr” returns “Tester”. If the initial letter is lowercase, the candidates will be all-lowercase—for example, spell aid for “tESTR” returns “tester”.

If the dictionary word has an initial capital or is in mixed or all-uppercase, any case pattern in the input word other than all-uppercase will receive the dictionary word with its given case pattern as a candidate. An all-uppercase input word will receive all-uppercase candidates. For example, spell aid for “ibmr” , “Ibmr”, “IBMr”, or any other mixed case variant will all return “IBMer” as the top candidate, while spell aid for “IBMR” will return “IBMER”.

Table 20. Linguistic Tools Spell Aid Case Matching

	All-lowercase Dictionary Word	**Initial Capital Dictionary Word**	**All-uppercase Dictionary Word**	**Mixed Case Dictionary Word**
All-lowercase Input Word	All-lowercase output	Initial capital output	All-uppercase output	Mixed case output
Initial Capital Input Word	Initial capital output	Initial capital output	All-uppercase output	Mixed case output
All-Uppercase Input Word (MM off)*	All-uppercase output	All-uppercase output	All-uppercase output	Mixed case output
All-Uppercase Input Word (MM on)*	All-uppercase output	All-uppercase output	All-uppercase output	All-uppercase output
Mixed Case Input Word (initial lowercase)	All-lowercase output	Initial capital output	All-uppercase output	Mixed case output (exact match)
Mixed Case Input Word (initial capital)	All-lowercase output	Initial capital output	All-uppercase output	Mixed case output (exact match)

*Mixed Mode flag set by **NlpMixedMode()**.

The **Grade Level Analysis** function will return information only for words which match according to the pattern described above.

Look Up Word and **Remove Word from Addenda** both require an exact match between the input word and the dictionary word.

If a word input to **Morphological Identification** (MID) or **Generate Inflected Forms** (GIF) matches a dictionary word according to the pattern described above, the lemma (for MID) or inflected forms (for GIF) will be returned in the case pattern of the dictionary word. If more than one match is possible—for example, the English input word "May" will match both "may" and "May" in the dictionary—all matching lemmas will be returned in the case patterns of the dictionary words—in this case both "may" and "May". Otherwise, these functions will return LX_WORD_NOT_FOUND.

Most of the words for which synonym information exists are either all-lowercase or (for German) have initial capitals. If an all-lowercase input word matches an all-lowercase dictionary word, it returns all-lowercase synonyms. In languages other than German, an input word with initial capital may match either an initial capital or an all-lowercase dictionary word and return synonyms with initial capitals. In German, because all nouns have initial capitals and other words normally do not (except in sentence-initial position), input words with initial capitals only match dictionary words with initial capitals (while all-lowercase input words match only all-lowercase dictionary words). An all-uppercase input word may match only an initial capital dictionary word; it may not match an all-lowercase dictionary word in German. In other languages an all-uppercase input word may match any case pattern except for mixed case (unless the Mixed Mode flag has been set) and will return all-uppercase synonyms. **Synonym Aid** will return LX_WORD_NOT_FOUND when there is a case mismatch.

If algorithmic hyphenation rules are available and are loaded when the dictionary is activated, **Hyphenate** will accept almost any single-word-delimited string as input and attempt to hyphenate it. The output string will match the case pattern of the input string. However, if algorithmic hyphenation rules are not available or are not loaded when the dictionary is activated, then **Hyphenate** will return LX_WORD_NOT_FOUND when there is a mismatch. If a matching word is found in the dictionary, the output string will match the case pattern of the input string.

Dehyphenate will also accept almost any single-word-delimited string as input. It simply removes the hyphens from the input string and returns the result, in the same case pattern, making no attempt to verify whether this is a correctly-spelled word.

The text extraction functions (**Extract Keyword** and **Extract Query Terms**) follow the pattern described above for matching input words with dictionary words. For the first word in a sentence, if the start of sentence flag is set, these functions will use all matching dictionary words found (see "Extract Keywords" on page 185 for details). If an input word does not match any dictionary word, the function will return it as a keyword as long as it is appropriately delimited. This includes both case mismatches and misspellings. However, such a keyword will have the input string as its lemma and its part of speech will be LX_POS_UNKNOWN.

If the normalization option is selected, keywords will be returned in all-lowercase, with language-specific processing as described in "Word Normalization" on page 44. If normalization is not selected, keywords will be returned in the same case pattern as the input word.

Compound Word Component Isolation currently accepts any of all-lowercase, all-uppercase, or initial capital input as matching either an all-lowercase or initial capital dictionary word (although an all-lowercase input word should not match an initial capital dictionary word). Only mixed case input is rejected (returns LX_WORD_NOT_FOUND).

Case is not significant to the **Simple Tokenization**, **Lexical Analysis**, and **Single Word Data Element Creation** functions. **Text Segment Identification** uses the case of the initial letter in the following word to determine whether a period marking an abbreviation also marks the end of a sentence.

Reply Area

Many of the Linguistic Tools functions output information via a block of memory referred to as the reply area. The specific structure of the data contained in the reply area depends on the type of function that was called.

The application should use the *lx_reply_p* field of the Linguistic Tools control block to provide the address of the reply area which it has allocated. The *lx_reply_size* field tells Linguistic Tools how large the reply area is. Table 21 on page 87 provides recommended reply area sizes for different functions.

Continue Reply Capability

Certain Linguistic Tools functions that place output in the reply area have the ability to suspend processing if the reply area fills up before processing is complete, and return whatever output fit. The application may then issue a **continue reply** to collect the balance of the output, or issue a **new request** to cancel the unfinished function and invoke a new function. If the application issues a continue reply, Linguistic Tools restarts processing where it left off and returns as much output as it can in the reply area. Sometimes multiple continue replies must be issued to collect all output.

The application may use Linguistic Tools' continue reply capability to regulate the way in which output is returned by the Simple Tokenization, Text Segment Identification, and Lexical Analysis functions. By setting LTCB field *lx_out_units* to the maximum number of tokens or sentences that the application wishes to obtain at one time, it can force Linguistic Tools to return only that number of tokens or sentences, regardless of the amount of text that the function had processed. To obtain the next group of tokens or sentences, the application would issue a continue reply.

On every call to Linguistic Tools, the application must set field *lx_rqst_type* of the LTCB to indicate whether it is issuing a continue reply or new request. Every call to Linguistic Tools that is not a continue reply is considered a new request.

When issuing a continue reply, the application must be careful to pass the Linguistic Tools the same function code, data element list, and (if used) the same spell verify and article checker flags as on the previous call. When fulfilling a continue reply, Linguistic Tools will begin by processing the input data element list just after the last element processed on the previous call. The Linguistic Tools will use the reply area as if it were empty, overwriting any output from the previous call.

Reply Area Size

Applications using Linguistic Tools must allocate space to hold a reply area. Table 21 on page 87 gives size estimates (in bytes) for the reply area by function. The definition of minimum size is the minimum amount of memory which will contain the largest single unit of output data. Note that the definition of an output unit will vary according to the function. For functions not supporting Continue Reply capability, the recommended size is the same as the minimum size, since all of the output constitutes a single unit and cannot be broken over several calls.

For functions supporting Continue Reply capability the recommended size will differ from the minimum size, since the minimum size is calculated to hold only one unit of output. These functions typically return more than a single unit of output,

however. Additional output units could be accessed by repeated calls, one per unit, to the same function with *lx_rqst_type* set for continue reply; however, it may be more efficient to allocate a reply area large enough to hold more than one unit of output. The recommended size given below is an estimate based on the expected size of one output unit multiplied by the average number of output units that a caller might require. This means that most requests can be handled by one call, thereby eliminating the overhead of multiple calls and increasing operating efficiency. An asterisk (*) in the recommended size column indicates that the caller can find the recommended size by multiplying the size of the structure to be returned by the maximum number of output units desired from a single call.

Some functions do not use the reply area at all. None of the token list utilities update or change the reply area, so they are not included in the table below. All of the text-processing functions are included, but not all use the reply area. Those which do not contain N/A in the columns for minimum and recommended size to indicate that the information is not applicable.

Application developers may wish to allocate one reply area and reuse it for each call to the Linguistic Tools. In this case, the developer should note the recommended reply area size for the functions which it intends to use and take the largest of these sizes for the amount of memory to allocate.

Note that the **Add Morphology to Addenda** function requires as input the data returned from two separate calls to **Generate Inflected Forms**. In this case, the application ***must*** allocate two separate reply areas.

It is assumed that the application can achieve a certain amount of performance tuning by varying the size of the reply area. The numbers provided below are to be taken more as suggestions than requirements.

Table 21 (Page 1 of 2). Recommended Reply Area Size by Function

Function Name	**Minimum Size**	**Recom-mended Size**	**Continue Reply Support**
Initialize	N/A	N/A	No
Terminate	N/A	N/A	No
Activate Dictionary	64	64	No
Deactivate Dictionary	N/A	N/A	No
Create Addenda Dictionary	64	64	No
Add Word to Addenda	N/A	N/A	No
Add Morphology to Addenda	N/A	N/A	No
List Addenda Words	68	*	Yes
Look Up Word	512	512	No
Remove Word from Addenda	N/A	N/A	No
NlpFindDicts()	1K	1K	No
Spell Verify	64	64	Yes
Spell Aid	394	394	No
Fuzzy Spell Aid	1304	1304	No
Article Checking	64	64	Yes

Table 21 (Page 2 of 2). Recommended Reply Area Size by Function

Function Name	Minimum Size	Recom-mended Size	Continue Reply Support
Grade Level Analysis	64	64	Yes
Hyphenate	82	82	No
Dehyphenate	65	65	No
Synonym Aid (and Word Definitions)	8K	8K	No
Inflect Word from Model	4K	4K	No
Compound Word Component Isolation	2K	2K	No
Morphological Identification	1276	1276	No
Generate Inflected Forms	39K	39K	No
Single-Word Data Element Creation	64	1K	Yes
Simple Tokenization	90	1K	Yes
Text Segment Identification	550	1K	Yes
Lexical Analysis	1K	*	Yes
Extract Keyword	7K	64K	Yes
Extract Query Terms	44K	64K	Yes

Tokens and Token Lists

Several Linguistic Tools functions return data in the reply area in the form of a token list. A **token list** is an ordered collection of objects (**tokens**) that describe text and its attributes to the application. In general, each text token in the list represents a word. Every token contains information about a portion of text such as the string that token represents. General information is provided below; more specific output is described in the output section for each Linguistic Tools function. Information represented by the token list must be retrieved by using the token list utilities described in "Token List Utilities" on page 91.

The Token

The token is a basic unit of output used to describe a portion of text. As a data structure, the token allows Linguistic Tools to describe text in a common format, regardless of the input data structure. In addition, it provides a very flexible method for representing attributes of the text through the use of properties (see "Properties" on page 95).

Token Characteristics

Every token contains, by default, some standard information. This includes:

- The string of characters associated with a token
- The length of the string associated with a token
- A pointer to the input text represented by the token
- The number of basic delimiter bytes that follow the string represented by the token.
- A bit-mask describing the types of characters (content) in the string associated with a token
  - Uppercase
  - Lowercase
  - Numeric
  - Punctuation
- Type of input text the token represents

  If text is input through the data-element structure, this type will equal the lx_type field of the data element that contains the text associated with this token. Otherwise, the type will always equal LX_TEXT.
- The handle of the next token in the token list
- The handle of the previous token in the token list
- A bit-mask (string flags) describing

  right-to-left—
  - Whether characters in the string part of the token are in single- or double-byte format
  - Whether the token is basic or joined

Basic, Joined, and Component Tokens

Normal tokens are limited by the fact that they may only represent contiguous text of the same type. Because this restriction is violated if the application inputs text containing control characters and/or uses more than one data element or word list data structure, a special type of token was defined.

A token may be either basic or joined. A **basic** token is one which represents a single contiguous string (including any trailing blanks). A **joined** token is a group of basic tokens, each called a **component** when part of a joined token, where each component represents a contiguous piece of the word. Logically, a joined token still represents a word and should be regarded as a single unit, a *virtual* token, so to speak. Some situations in which Linguistic Tools returns joined tokens include:

- text across discontiguous memory
- New-lines between parts of words
- New-lines followed by blanks
- Some SO/SI situations
- Mixed SBCS/DBCS text (Linguistic Tools separates these at the lowest level)

A token list can consist of any combination of basic and/or joined tokens. The application may determine whether a token is joined or basic by performing a bit-wise AND operation with the token's string flag and the constant LX_JOIN.

It is important for the application to know whether a token is joined or not, because that determines how the application may evaluate the string associated with it. To work with the string associated with a basic token, the application need only call **NlpGetTokenLen()** and **NlpGetStringAddr()** to obtain a pointer and a length to the token's string. However, for a joined token the application must call **NlpGetTokenLen()** and **NlpGetString()** to obtain a pointer and length to the token's string. Although **NlpGetString()** will return the string for either a basic or joined token, it has the additional overhead of always copying the string into a buffer, as well as requiring the caller to maintain a buffer in which to hold the string.

NlpGetComponent() and **NlpGetNumComponent()** are functions designed specifically for joined tokens (if these functions are passed the handle to a basic token, their output is undefined). **NlpGetNumComponent()** returns the number of component tokens that make up the joined token. **NlpGetComponent()** returns the token handle of a specific component token so that the caller may pass that component's handle to other token list utilities. All other token list utilities are able to operate on either basic or joined tokens.

Note: Because a component token is merely a basic token that is a subset of a joined token, all functions that are valid for basic tokens are also valid for component tokens.

Treatment of Trailing Delimiters

Currently, Linguistic Tools defines a basic delimiter to mean the space character. When tokens are created to represent the input text, all basic delimiters are separated from other text because the token length does not include any basic delimiter bytes. To obtain information about how many basic delimiters follow (or trail) the current token, the application must call **NlpGetNumTrail()**. This function will tell the user how many basic delimiter characters follow the string represented by this token.

Because Linguistic Tools assumes that the basic delimiters of a basic token are consecutive to the token string, special cases need to be explained. When the input text begins with spaces, the first token in the token list will have a string length of zero, a trailing delimiter count equal to the number of leading spaces, and the string pointer (obtained from **NlpGetStringAddr()**) will point to the beginning of the input text.

When data element input contains a text data element with leading spaces, Linguistic Tools will create a basic token to represent those spaces. The basic token will then be joined with any previously created token (if one exists) to make sure that the basic delimiters that follow a given string are grouped together. For example, the input in Figure 2 would produce three tokens when processed by Linguistic Tools. The first token would have a string length of five and a type of LX_TEXT. The second token would be a joined token with two components. The first component would have a string length of eleven, trailing delimiter count of zero, and a type of LX_UDAT. The second component would have a string length of zero, a trailing delimiter count of two, and a type of LX_TEXT. Therefore, the joined token also would have a trailing delimiter count of two. The third token produced would have a string length of four, a trailing delimiter count of one, and a type of LX_TEXT.

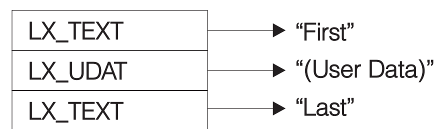


Figure 2. Sample Data Element Input

User Data Tokens

User data tokens contain information that the application wishes Linguistic Tools to ignore, such as formatting tags—for example, ***:hp2.*** or ***:ehp2***.

Linguistic Tools does not do any processing of text contained in an input user-data element. However, it does regard the occurrence of a user-data element of type LX_UDAT between two input text elements as a word delimiter. A user-data element of type LX_IGNORE between two input text elements will not be considered a word delimiter.

Token List Utilities

Certain Linguistic Tools functions return data in the form of a token list. When one of these functions has been invoked, it will return data in the area indicated by lx_reply_p. The token list utilities assume that the data they will use is in the reply area used on the last call to NlpEntry(). These functions are summarized in Table 22; more detailed information is provided in Chapter 17, "Token List Utility Functions" on page 229.

Table 22 (Page 1 of 2). Linguistic Tools Token List Utilities

Function	Reference	Description
NlpGetBeginType	Page 229	Returns character type of first character in a token string
NlpGetNumComponent	Page 239	Find out how many components are associated with a given token

Table 22 (Page 2 of 2). Linguistic Tools Token List Utilities

Function	Reference	Description
NlpGetComponent	Page 231	Get handle of token component—this handle is needed in order to get detailed information about a token
NlpGetTokenLen	Page 247	Returns length of string associated with a token
NlpGetTokenType	Page 249	Returns type of a given token
NlpGetContent	Page 232	Returns description of token contents
NlpGetString	Page 243	Places string associated with a token into a buffer from which application can retrieve it
NlpGetStringAddr	Page 245	Returns location of token string in input text
NlpGetStringFlag	Page 246	Returns flags indicating whether • token string is single- or double-byte • token is basic or joined
NlpGetNumTrail	Page 240	Returns number of basic delimiters (generally blanks) that trail a given token
NlpGetFirstProperty	Page 234	Returns first property associated with a token
NlpGetNextProperty	Page 236	Returns handle to property following the current property
NlpGetPval	Page 242	Return value of a property
NlpGetNextToken	Page 238	Returns next token handle on token list
NlpGetPrevToken	Page 241	Returns preceding handle on token list

Examples

The following two figures contain code samples that demonstrate how to navigate through a token list using the Token List Utilities. These examples are based on the assumption that the token list to be processed was generated by the Linguistic Tools Lexical Analysis function. Some modifications to the sample code would be necessary if it were intended to process a token list generated by other Linguistic Tools functions (for example, Extract Keyword, Extract Query Terms).

Figure 3 on page 93 is a code sample that advances through a token list and queries information about each token. This example assumes that there are no joined tokens in the token list.

Figure 4 on page 94 is a code sample that prints the exact input text represented by a given token list. In order to output the trailing delimiters in the text, calls to **NlpGetStringAddr()** and **NlpGetNumTrail()** are made. **NlpGetString()** would not allow us to output all of the text since it does not include any trailing delimiters in its

output. Each component of a joined token must be accessed individually since the code uses the string pointer to directly access the characters of the text.

```
LX_USHORT     rc;
LX_HNLPSERV   hNlpService;      /* Handle of NLP Service   */
LX_HTOKEN     hToken;           /* Handle of current token */
LX_PUCHAR     pString;
LX_USHORT     StringLen;
LX_USHORT     x;
LX_TOKCONTENT TokContent;
/*--------------------------------------*/
/* Because this is a bottom-tested loop, */
/* this code assumes that hToken        */
/* contains a valid value               */
/*--------------------------------------*/
do
  {
  NlpGetStringAddr(hNlpService, hToken, &pString);
  NlpGetTokenLen(hNlpService, hToken, LX_TEXT, &StringLen);

  /* Print out the string associated with token */
  for(x=0; x<StringLen; ++x)
    printf("%c", pString[x]);

  printf("\n");   /* Make sure new-line follows token string */

  NlpGetContent(hNlpService, hToken, &TokContent);

  /* Print a special message if */
  /* string is all upper-case   */
  if (TokContent == LX_UPPERCASE)
    printf("It's all upper-case !!\n");

  /*-------------------------------------------*/
  /* By passing hToken as both an input and an */
  /* output parameter, we set its value to the */
  /* next token present in the token list.     */
  /*-------------------------------------------*/
  rc = NlpGetNextToken(hNlpService, hToken, &hToken);
  } while(rc == LX_RC_OK);
```

Figure 3. Sample code to traverse token list generated by Lexical Analysis

```
LX_HNLPSERV  hNlpService;
LX_HTOKEN    hToken,
             hComponent;
LX_PUCHAR    pString;
LX_STRFLAG   StringFlag;
LX_UINT      x,y, NumComponent;
LX_USHORT    StrLen,
             NumTrails,
             TotalLen,
             rc;

do
  {
  NlpGetStringFlag(hNlpService, hToken, &StringFlag);

  /*----------------------------------------------*/
  /* If the current token is joined, then print    */
  /* every one of its components individually.     */
  /* Otherwise, print the basic token information. */
  /*----------------------------------------------*/
  if (StringFlag & LX_JOIN)
    {
    NlpGetNumComponent(hNlpService, hToken, &NumComponent);
    for(y=1; y<=NumComponent; ++y)
      {
      NlpGetComponent(hNlpService, hToken, y, &hComponent);
      NlpGetStringAddr(hNlpService, hComponent, &pString);
      if (pString != NULL)
        {
```

generated by Lexical Analysis

Figure 4. Sample code to display all text associated with a token list

```
        NlpGetTokenLen(hNlpService, hComponent, LX_ALL_TEXT, &StrLen);
        NlpGetNumTrail(hNlpService, hComponent, &NumTrail);
        TotalLen = StrLen + NumTrail;
        for (x=0; x<TotalLen; ++x)
          printf("%c", pString[x]);
        }
      }
    }
  else
    {
    NlpGetStringAddr(hNlpService, hToken, &pString);
    if (pString != NULL)
      {
      NlpGetTokenLen(hNlpService, hToken, LX_ALL_TEXT, &StrLen);
      NlpGetNumTrail(hNlpService, hToken, &NumTrail);
      TotalLen = StrLen + NumTrail;
      for (x=0; x<TotalLen; ++x)
        printf("%c", pString[x]);
      }
    }

  /* Advance to next token in list */
  rc = NlpGetNextToken(hNlpService, hToken, &hToken);
  } while (rc == LX_RC_OK);
```

generated by Lexical Analysis

Figure 5. Sample code to display all text associated with a token list

Properties

Any token may optionally have a property list associated with it. A **property list** is an ordered list of one or more properties. A **property** is an object that may be associated with a token to hold additional information about that token. Each property is composed of an identifier and an optional value. For example, a token representing the word “truck” could have a property that indicates that this word is a noun. Alternatively, the same token could have a property for part-of-speech whose value points to a constant that indicates that this word is a noun. The ways in which properties represent information are determined by the functions that create them.

Linguistic Tools properties are defined in one of the include files (EFZLEXPR.H) shipped with Linguistic Tools.

Examples

Figure 6 on page 96 contains sample code to traverse an imaginary property list. This example examines every property associated with a given token (note that this example assumes that hToken has been given a valid value from elsewhere). After obtaining the handle for each property, it obtains the identifier and value associated with the property. If the property identifier is equal to either SMILE_PROP or WORRY_PROP (imaginary constants), then a special message is printed.

Code like that found in Figure 6 on page 96 is useful when looking for more than one property on a token. However, if the application is looking only for a property with a specific identifier, a simpler approach can be taken. By using the RqstPropID parameter of **NlpGetFirstProperty()**, the application can ask Linguistic Tools to perform the work of looking through the property list. If the application provides the desired property identifier instead of passing the Linguistic Tools constant LX_ANYPROP, then the function will return LX_RC_OK only if a property with that identifier was found. Figure 7 on page 96 shows an example of passing a value other than LX_ANYPROP to **NlpGetFirstProperty()**.

```
LX_HNLPSERV hNlpService;   /* NLP Service handle                   */
LX_HTOKEN   hToken;        /* Handle of current token              */
LX_HPROP    hCurrProp;     /* Handle of the current property       */
LX_PROPID   PropID;        /* Identifier of the current property  */
LX_PVOID    PropValue;     /* Value of the current property        */
LX_USHORT   rc;            /* Return code                          */

rc = NlpGetFirstProperty(hNlpService, hToken, LX_ANYPROP, &hCurrProp);
while (rc == LX_RC_OK)
  {
  /*-----------------------------------------------------*/
  /* Get the identifier and value for the current property */
  /*-----------------------------------------------------*/
  NlpGetPval(hNlpService, hCurrProp, &PropID, &PropValue);

  /*----------------------------------------------------------*/
  /* If the property ID is the one we want, print out a message */
  /*----------------------------------------------------------*/
  if (PropID == SMILE_PROP)
    printf("Be happy. \n");

  else if (PropID == WORRY_PROP)
    printf("Don't worry \n");

  /* Advance to the next property in the list */
  rc = NlpGetNextProperty(hNlpService, hCurrProp, LX_ANYPROP, &hCurrProp);
  }
```

Figure 6. Sample code to traverse property list

```
LX_HNLPSERV hNlpService;
LX_HTOKEN   hToken;
LX_HPROP    hProperty;
LX_USHORT   rc;              /* Return code */

rc = NlpGetFirstProperty(hNlpService, hToken, SMILE_PROP, &hProperty);
if (rc == LX_RC_OK)
  printf("Found the one we wanted! \n");
```

Figure 7. Sample code to find a specific property

DBCS Considerations

When working with a token list that represents double-byte text, several considerations must be taken into account. The meaning of string length, string content information, and string flags must be clarified for a double byte environment. In addition, differences between PC and HOST code page architectures must be accounted for. Because host code pages require that DBCS strings be bracketed by shift-in/shift-out characters, a special token type was created to represent them. Except for the presence of SO/SI characters in the host DBCS text, PC and host DBCS text is processed in the same way.

The first consideration of DBCS tokens involves the way in which string lengths are specified. The token utilities assume that ***all*** lengths are indicated in units of bytes. Therefore, when specifying the length of a 4-character double byte string, the string

length should be passed as eight, not four. Also, it should be noted that string pointers always start with the first byte of a DBCS string.

The content-bits of a double byte string are usually set to zero. Linguistic Tools does not attempt to distinguish between upper-case, lower-case, and numerics in double-byte text. Although the content-bits of some tokens will indicate punctuation, Linguistic Tools does not attempt to distinguish all double-byte punctuation characters. Only those characters mentioned in the language-specific processing sections will be marked as punctuation.

DBCS Token Examples

Figure 8 shows a sample DBCS token list. In this example, five tokens of type LX_TEXT are produced from a hypothetical input text sequence of two DBCS characters, two SBCS characters, a SBCS trailing punctuation character, two SBCS characters, two DBCS characters, a DBCS trailing punctuation character, and a final DBCS character. The token list shown is representative of what would be output from the Simple Tokenization function.

This example shows how Simple Tokenization separates double-byte text from single-byte text. In addition, any trailing characters are also determined, whether or not they are single-byte characters. Note that the lengths provided are measured in bytes, not numbers of characters. A single double-byte character always has a length of two.

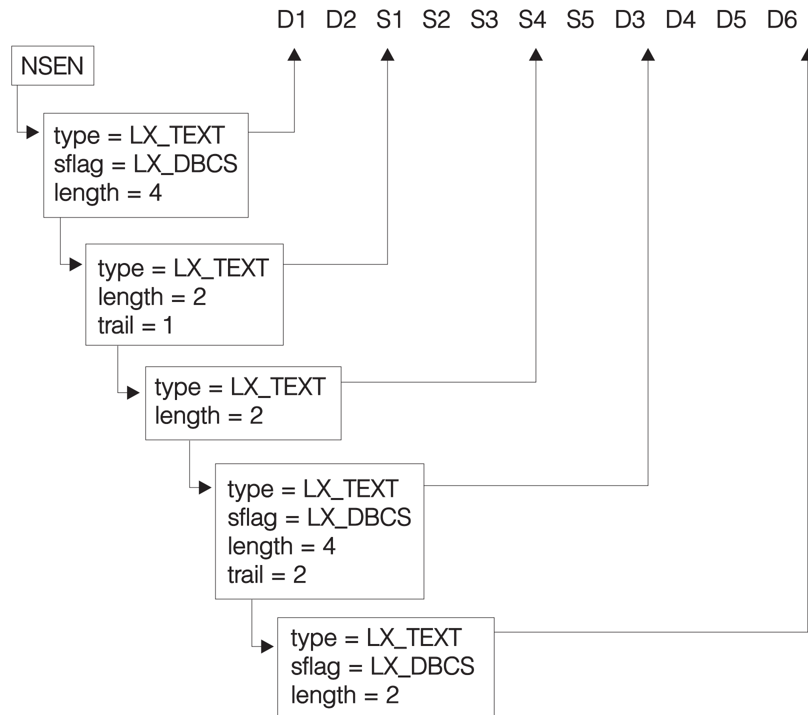


Figure 8. Sample Token List with DBCS Text

Host DBCS Text Tokens

Because host double-byte text is surrounded by shift-out and shift-in (SO/SI) characters, special considerations have been taken by Linguistic Tools. A special token type LX_SOSI has been implemented to separate the control characters from the actual text. Obtaining the string associated with a token works slightly differently for host DBCS text. A special string flag has been implemented to indicate whether the text associated with a token is in host double-byte form. These factors are designed to make the calling application's job easier when dealing with host DBCS text.

SO/SI characters are isolated as separate tokens of type LX_SOSI to prevent the control characters from interfering with actual text. A token of type LX_SOSI is created for every SO/SI character that was present in the input text. Because SO/SI characters are placed in separate tokens, it is easier to find any control characters in the input.

Because the control characters are separated from the text, Linguistic Tools performs special processing when the user requests the string associated with a token using the NlpGetString() utility. When a string is returned from this routine, only valid host DBCS strings will be produced. Thus, any host double-byte text will be surrounded by the necessary SO/SI characters, whether or not the original input text contained the SO/SI characters at that exact position. Because these characters are added, output from the NlpGetString() routine is suitable for immediate display. Without the insertion of the extra control characters, the text would not be displayable.

For any given token, the application may always determine the context of any double-byte text associated with a token by that token's string flags. The LX_DBCS string flag indicates if there is double-byte text associated with a token. In addition, the LX_HOSTDBCS string flag indicates whether the text associated with a token is in host DBCS form. (Note that the LX_DBCS flag will always be present if the LX_HOSTDBCS flag is set.) These string flag settings are provided so that an application need not separately maintain the state of the output text when processing token list information.

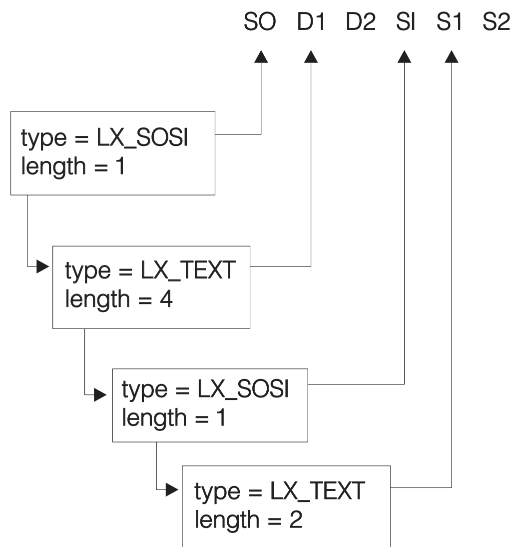


Figure 9. Sample Token List for Host DBCS Text

Figure 9 shows a sample token list for host DBCS text. Unlike the previous example, this text contains SO/SI control characters. Each SO/SI character is represented as a token of type LX_SOSI, separate from the other text tokens.

Dictionaries

The majority of the Linguistic Tools involves access to dictionaries. Dictionaries may be in either text or binary format. System, Legal and Medical Dictionaries are very large files stored in a compacted binary format, which cannot be edited by the user. Addenda dictionaries may be stored in either binary or flat file (text) format. Only the flat file format is user-editable.

Note: Linguistic Tools 2.1 Japanese main dictionaries (.DCJ files) and Linguistic Tools 2.1 Japanese user dictionaries (.URJ files) are ***not*** supported in Linguistic Tools 2.7.

On the workstation, System Dictionaries are stored as binary files. Under VM/CMS, System Dictionaries should be stored as F (fixed) format files with LRECL=512.

The application indicates which dictionaries should be used by means of an input dictionary list which is passed in with the function call. Dictionary handles are assigned by the **Activate Dictionary** function. The order of dictionary handles in the list should reflect the importance of each active dictionary, from the application's point of view. For example, if the application is dealing with a document which is written primarily in English with a few French words and phrases interspersed throughout, the application may place the English dictionary handle first in the list, followed by the handle of the French dictionary. In this way, the application instructs the Linguistic Tools to assume that a word is English and search for it in the English dictionary, only searching the French dictionary when the English dictionary contains no matches. "Dictionary Use by Function" describes how different Linguistic Tools functions make use of the input dictionary list.

Dictionary Use by Function

The input dictionary list in the Linguistic Tools control block indicates which of the active dictionaries are to be used by the invoked function. Dictionary usage by function is as follows:

- ***Activate dictionary***—The input dictionary list is not used.
- ***Add Morphology to Addenda***—Only one dictionary handle is allowed in the input dictionary list, indicating the addenda dictionary to which the words should be added.
- ***Add Word to Addenda***—Only one dictionary handle is allowed in the input dictionary list, indicating the addenda dictionary to which the word should be added.
- ***Article Checking***—The input dictionary list is not used.
- ***Compound Word Component Isolation***—All Germanic dictionaries in the input dictionary list will be checked for decompositions of the input word.
- ***Create Addenda***—The input dictionary list is not used.
- ***Deactivate Dictionary***—Only one dictionary handle is allowed in the input dictionary list, indicating which dictionary to deactivate.
- ***Dehyphenation***—The language of the first dictionary in the input dictionary list determines whether language-specific spelling changes must be made to the word. Otherwise, dehyphenation makes no use of the dictionaries.

- ***Extract Keywords***— This function scans the dictionaries in the input dictionary list to verify and decompose words, and to extract morphological information for the words.
- ***Extract Query Terms***— This function scans the dictionaries in the input dictionary list to verify and decompose words, and to extract synonyms and morphological information for the words.
- ***Generate inflected forms***—Scans dictionaries in the order of the input dictionary list and extracts word forms from the first dictionary which contains both morphology data and the word.
- ***Grade level analysis***—Scans dictionaries in the order of the input dictionary list until the input word is found or all dictionaries have been scanned.
- ***Hyphenation***—Scans dictionaries in the order of the input dictionary list and extracts the hyphenation point from the first dictionary which contains both hyphenation data and the word. If no hyphenation point can be determined from the initial scan, algorithmic hyphenation is performed, based on the language of the first dictionary in the list.
- ***Inflect word from model***—Scans dictionaries in the order of the input dictionary list and extracts word forms from the first dictionary which contains both morphology data and the word.
- ***Initialize Linguistic Tools***—The input dictionary list is not used.
- ***Lexical Analysis***—Scans the dictionaries in the input dictionary list until the input word is found or all dictionaries have been scanned. If no dictionaries are available—that is, there are no dictionary handles in the input dictionary list—this function will perform default (no language-specific) processing for SBCS languages. For Chinese and Japanese, the appropriate dictionary must be specified.
- ***List Addenda Words***—Only one dictionary handle is allowed in the input dictionary list, indicating the addenda dictionary to be listed out.
- ***Look Up Word***—Scans all active dictionaries in the order of the input dictionary list and returns information from the first dictionary in which the word is found.
- ***Morphological identification***—Scans dictionaries in the order of the input dictionary list and extracts morphological information from the first dictionary which contains both morphology data and the word. If this function is being called for Korean text, the first dictionary in the input dictionary list must be a Korean dictionary, as the function will use the language code of the first dictionary in the dictionary list to determine what language-specific processing to perform.
- ***Remove word from addenda***—Only one dictionary handle is allowed in the input dictionary list, indicating the addenda dictionary from which to remove the word.
- ***Save addenda***—Only one dictionary handle is allowed in the input dictionary list, indicating the addenda dictionary to be saved to disk.
- ***Simple tokenization***—The input dictionary list is not used.
- ***Single-word data element creation***—The input dictionary list is not used.
- ***Spell aid*** — Scans each dictionary in the input dictionary list for candidate spellings.

- ***Fuzzy Spell aid***—Scans the addenda dictionary in the input dictionary list for candidate spellings.
- ***Spell verify***—Scans the dictionaries in the input dictionary list until the input word is found or all dictionaries have been scanned.
- ***Synonym aid***—Scans dictionaries in the order of the input dictionary list and extracts synonyms (and word definitions, if requested) from the first dictionary which contains both synonym data and the word.
- ***Text Segment Identification***—Scans the dictionaries in the input dictionary list until the input word is found or all dictionaries have been scanned. If no dictionaries are available—that is, there are no dictionary handles in the input dictionary list—this function will perform default (no language-specific) processing for SBCS languages. DBCS text will be tokenized, but DBCS terminal punctuation will not be handled correctly unless the language code is specified.
- ***Terminate Linguistic Tools***—The input dictionary list is not used. All active dictionaries, whether in the list or not, are deactivated upon termination of Linguistic Tools.

Operating System-Specific Information

The following information is specific to particular operating systems.

Execution Environments

See “Platforms Supported” on page 9.

Deliverables

See “Product Contents” on page 4 for a list of the deliverables. In addition to the information listed there, a summary of the Linguistic Tools deliverables is included in the following information.

For all operating systems Linguistic Tools includes:

- The API description (this document)
- A drvpkg.txt file containing operating system-specific information
- The driver file (explained in “Driver Files”)
- The Import Symbol File, if needed
- A set of include files
- A set of System, Stopword Addenda, and Abbreviation Addenda Dictionaries for the desired languages

In addition, the following items are available by request:

- The Testing Tool user’s guide
- The Testing Tool executable software
- A sample test case for use with the Testing Tool
- Sample code showing how to call the various functions

Driver Files

The following list includes the actual driver files provided for each of the platforms:

- 32-bit Microsoft Windows, compiled with IBM VisualAge C++ v3.5

  Dynamic Link Library file: eflnl27w.dll
  Import Library file: eflnl27w.lib
  Map file: eflnl27w.map

- OS/2, compiled with IBM VisualAge C++ v3.0

  Dynamic Link Library file: eflnl272.dll
  Import Library file: eflnl272.lib
  Map file: eflnl272.map

- AIX (v4.1 and higher), compiled with IBM AIX XL C Compiler/6000 v3.1.4.6

  Shared Object file: libeflnl27x.so
  Import Library file: efxlnlps.imp

- HP-UX (v10.10 and higher) 45, compiled with HP SoftBench ANSI C Compiler

  Shared Library file: libeflnl27h.sl

- Sun-OS (v2.5 and higher), compiled with Sun Sparc ANSI C Compiler

  Shared Object file: libeflnl27s.so

- OS/390 OpenEdition (OS/390 UNIX System Services), compiled with IBM C/370 v2.1.0

  Dynamic Link Library file:
  libeflnl273.dll
  Definition Side-Deck file:
  libeflnl273.x
  Prelinker Composite Object file:
  libeflnl273.p

- OS/400 (V4R5 and higher), compiled with ILE-C Cross Compiler under AIX

  Service Program file: QDICT/EFLNL274

- MacOS (v9.0 & higher), compiled with MetroWerks CodeWarrior 5.0

  Static library eflnl27a.lib

Software Customization

The Linguistic Tools driver can be customized by removing code for unused functions to reduce the size of the driver. This is done only by request and for all operating system platforms except the OS/400. To accomplish this, the Linguistic Tools has been arranged in groups of related functions. The Linguistic Tools can then be customized by combining the desired groups and leaving out the unnecessary groups. Contact IBM support to learn more about Linguistic Tools software customization.

Program Build Information

When compiling with the Linguistic Tools, the application should use the set of include files provided by the Linguistic Tools developers. The calling program needs only to include the single file EFZLNLPS.H (which includes several other include files) in its code to obtain the following information:

- Data Type definitions
- Control Block structure definition
- Data Element structure definition
- Function Code constants
- Return Code constants
- Linguistic Tools function prototypes
- Structure definitions used in Linguistic Tools reply area output
- Other useful constant definitions

For a 32-bit Microsoft Windows application to be built with the Linguistic Tools, using the Microsoft Visual C++ compiler:

- The **/DLX_WIN32** compile option should be specified to ensure that the EFZLNLPS.H include file expands correctly.
- The **/Zp1** compile option must be used in modules using structures passed to or returned by the Linguistic Tools, because the Linguistic Tools assumes that all structures passed to it are packed.

For a 32-bit Microsoft Windows application to be built with the Linguistic Tools, using the IBM VisualAge C++ compiler:

- The **/DLX_WIN32** compile option should be specified to ensure that the EFZLNLPS.H include file expands correctly.

- The **/Sp1** compile option must be used in modules using structures passed to or returned by the Linguistic Tools, because the Linguistic Tools assumes that all structures passed to it are packed.

For an OS/2 application to be built with the Linguistic Tools, using the IBM VisualAge C++ compiler:

- The **/DLX_OS232** compile option should be specified to ensure that the EFZLNLPS.H include file expands correctly.
- The **/Sp1** compile option must be used in modules using structures passed to or returned by the Linguistic Tools, since the Linguistic Tools assumes that all structures passed to it are packed.

For an AIX application to be built with the Linguistic Tools:

- The **-DLX_AIX** compile option should be specified to ensure that the EFZLNLPS.H include file expands correctly.
- The application should link using the **-bI:/<impfile_path>/efxlnlps.imp** option.

  Note: It is assumed that the shared library file is in a directory that is specified by the LIBPATH environment variable.

For an HP-UX application to be built with the Linguistic Tools,

- The **-DLX_HPUX** compile option should be specified to ensure that the EFZLNLPS.H include file expands correctly.
- The **+u1** compile option must be used in modules using structures passed to or returned by the Linguistic Tools, since the Linguistic Tools assumes that all structures passed to it are packed. The Linguistic Tools does not assume that structures passed to it begin on naturally aligned boundaries.
- The application should link using the **-ashared** option to force the linker to search for a shared library.
- The application should link using the **+s** option to allow the shared loader to use the SHLIB_PATH environment variable to locate shared libraries that are needed by the executable object file.
- The application should link using the **-L<directory of shared library>** option to specify additional library search directories at link time.
- The application should link using the **-leflnl27h** option.

  Note: It is assumed that the shared library file is in a directory that is specified by either the LIBPATH environment variable or the -L link option.

For a Sun Solaris application to be built with the Linguistic Tools,

- The **-DLX_SUN** compile option should be specified to ensure that the EFZLNLPS.H include file expands correctly.
- The **-misalign** compile option must be used in modules using structures passed to or returned by the Linguistic Tools, because the Linguistic Tools assumes that all structures passed to it are packed. The Linguistic Tools does not assume that structures passed to it begin on naturally aligned boundaries.
- The application should link using the **-R<directory-list>** option to specify library search directories to the runtime linker.

- The application should link using the **-L<directory of shared library>** option to specify additional library search directories at link time
- The application should link using the **-leflnl27s** option.

  Note: It is assumed that the shared library file is in a directory that is specified by either the LIBPATH environment variable or the **-L link** option.

For an OS/390 OpenEdition (OS/390 UNIX System Services) application to be built with the Linguistic Tools:

- The **-DLX_OS390_OE** compile option should be specified to ensure that the EFZLNLPS.H include file expands correctly.
- The **-Wc,dll** compile option must be used to indicate that the application will be linked with a DLL.

For an OS/400 application to be built with the Linguistic Tools, using the ILE-C compiler:

- The **define(LX_OS400)** compile option should be specified to ensure that the EFZLNLPS.H include file expands correctly.
- The **sysifcopt(*IFSIO)** compile option should be specified to ensure that the IFS path names are treated properly.
- The application should bind with the Linguistic Tools service program by using the **bndsrvpgm(QDICT/EFLNL274)** option.
- Applications written in languages other than ILE-C may need to define "Named Activation Group" for the ILE-C programs that interface to the Linguistic Tools service program, in order to have persistent memory between calls to the Linguistic Tools. (The Linguistic Tools service program is defined to use the *CALLER activation group.)

For a MacOS application to be built with the Linguistic Tools:

- The **/DLX_MAC** compile option should be specified to ensure that the EFZLNLPS.H include file expands correctly.
- The align arrays of characters compile option must be set **OFF** in modules using structures passed to or returned by Linguistic Tools, because Linguistic Tools assumes that all structures passed to it are packed.
- The MACTRAPS library must be linked with the application.

The drvpkg.txt file, which is part of the Linguistic Tools deliverables, provides an overview of these instructions.

Reentrancy of the Linguistic Tools

For all platforms, Linguistic Tools is reentrant on the thread level, provided that each concurrent thread of the Linguistic Tools has been initialized separately.

File Names

The conventions described below have been developed with portability in mind. They are somewhat more restrictive than the conventions of any one operating system.

In general, it is recommended that:

- File names consist of 1-8 characters, of which the first must be alphabetic, while the rest may be alphanumeric.
- Extensions consist of 1-8 characters, depending on the operating system, of which the first must be alphabetic, while the rest may be alphanumeric.

For portability between the host and the workstation, it is recommended that extensions not exceed 3 characters in length.

Fully-Qualified File Names

For 32-bit Microsoft Windows and OS/2, a fully-qualified file name consists of:

```
drive:\dir_1\dir_2\ _ _ _ \dir_x\filename.ext
```

where the drive identifier consists of a single alphanumeric character, and subdirectory names can consist of 1-8 characters, followed by an optional extension of 1-3 characters. The filename may consist of 1-8 characters, and the extension may be up to 3 characters in length.

The filename should be separated from the extension by a period. The subdirectory names should be separated by backslashes; Linguistic Tools will regard everything up to the last backslash as the path name. A fully-qualified file name may be up to 256 characters long.

For OS/400 and the UNIX platform systems (AIX, HP-UX, Sun Solaris, OS/390 OpenEdition), a fully-qualified file name consists of :

```
/dir_1/dir_2/ _ _ _ /dir_x/name_1.name_2. _ _ _ .name_x.extension
```

where each directory name can consist of 1-8 characters, as can each portion of the file name, while the extension can consist of 1-3 characters. The subdirectory names should be separated by slashes; Linguistic Tools regards everything up to the last slash as the path name. Although these systems permit the use of multiple period delimiters, Linguistic Tools groups together everything up to the last period as the file name and regard only the string following the last period as the file extension.

The case of file names is significant for the UNIX systems only. An all-lowercase file name *fname* is considered to be different from the all-uppercase *FNAME* and the mixed case *Fname*. For OS/400, the case of the filename does not matter.

Under **MacOS** a fully-qualified file name consists of:

```
device:folder1:folder2: _ _ _ :folderX:filename
```

where the device and folder names may consist of 1-8 characters. The filename can consist of 32 characters.

The device/folder names should be separated by colons; Linguistic Tools will regard everything up to the last colon as the path name. A fully-qualified file name may be up to 256 characters long.

On the Macintosh, there are no file extensions, as with the other workstation platforms. Instead, the type of the file is maintained as a separate file attribute.

Unqualified File Names

For the Macintosh, unqualified file names consist of a Macintosh filename. For all other platforms, unqualified file names consist of a file name and extension.

Linguistic Tools Default Dictionary Naming Conventions

Most Linguistic Tools functions make no particular assumptions about file names beyond the definitions of fully-qualified and unqualified file names given above. For example, **Activate Dictionary** will accept any file name and attempt to activate that file as a dictionary.

There is one Linguistic Tools function, however, that ***must*** assume that dictionary files can be identified by their names: **NlpFindDicts()**. By default, this function assumes that the name of a dictionary or addenda has one of the following file types/extensions associated with it:

- ADD — Addenda Dictionary (flat file/text format)
- ADB — Addenda Dictionary (binary format)
- DIC — System Dictionary (v1.5 and 1.7)
- STW — Stopword Addenda (flat/text format)
- STB — Stopword Addenda (binary format)
- ABR — Abbreviation Addenda (flat/text format)
- ABB — Abbreviation Addenda (binary format)
- MED — System Medical Dictionary
- LEG — System Legal Dictionary
- CPT — System Computing Terms Dictionary

The application is not required to use these. If it chooses to use a different set of extensions and it wishes to be able to make use of **NlpFindDicts()**, it must inform Linguistic Tools through **NlpSetDictDef()**. Otherwise, **NlpFindDicts()** will not be able to identify dictionary files correctly.

Note: The Macintosh platform is an exception—rather than using a file extension, a separate file attribute is used. The three-letter dictionary types, given above, will be stored in the file-type attribute, which will be referenced by **NlpFindDicts()**.

Dictionary Search Path

If an input file name is fully qualified (as described in "Fully-Qualified File Names" on page 107), Linguistic Tools will search for the file in the specified location. If it is not found, no further search is made.

If an input filename is unqualified and a search path has been specified using **NlpSetSearchPath**, Linguistic Tools will search for the filename in all specified locations. Note that when a search path is present, only the locations from the search path are examined. This means that not even the current directory is searched, unless it is part of the search path. The routines **NlpSetSearchPath()** and **NlpQrySearchPath()** are used to set and query Linguistic Tools search path settings.

Specifying a Search Path

For Microsoft Windows and OS/2, the search path consists of a series of directory names separated by semicolons:

```
dir1;dir2;dir3; _ _ _ ;dirx;
```

where each dir name may include a drive specification and multiple subdirectory levels.

On the **Macintosh** platform, the search path consists of a series of folder sets separated by semicolons:

```
device1:folder1:folder2; _ _ _ ;device2:folder1:folder2;
```

where each folder set consists of multiple levels of device and/or folder names.

For OS/400 and the UNIX platforms, the search path is specified by a series of directory names separated by colons:

```
dir1:dir2:dir3:_ _ _:dirx:
```

where each **dir** name can include multiple subdirectory levels.

Search Path Defaults

If no search path has been set through **NlpSetSearchPath()**, Linguistic Tools will assume a default search path for unqualified file names:

- For the Macintosh, the search path will default to the path specified in the NLPS Preferences file.
- For all other platforms, if the application or the user has set an environment path variable *DICTPATH* to indicate where dictionaries are located on the system, Linguistic Tools will use this as its default search path.
- For OS/400, if the *DICTPATH* environment variable is not set, the default search path is the current directory and the directory where the product dictionaries are installed.

The **NLPQrySearchPath()** function will return the current search path. If the application has specified its own search path through **NLPSetSearchPath()**, that is what will be returned. If no path has been set by the application, the function will return the default search path that was described previously.

Error Handling

Linguistic Tools returns an eror code for those operating system errors that are trappable, for example, File Not Found, Memory Allocation Error, and so on. It is assumed that the application calling Linguistic Tools handles ABEND errors, those that result in termination of the process that generated the error, such as Memory Access Violation and others of this type.

Memory Management

Except for the reply area, Linguistic Tools will own and manage all memory areas required.

Part 2. IBM Dictionary and Linguistic Tools Functions

© Copyright IBM Corp. 2000

Chapter 6. Base Functions

Initialize

This function initializes the Linguistic Tools so it can be called to carry out other functions. It sets up the Service Area, returning a handle (in field *lx_serv_area_p* of the Linguistic Tools control block). This handle must be provided in subsequent calls to Linguistic Tools.

This function may be called more than once, in effect starting up multiple instances of the service, each with its own Service Area.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_func_code* — set to LX_INITIALIZE_NLPS
- *lx_rqst_type* — set to LX_NEW_REQ

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated:

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_version_num*
- *lx_cont_reply_f* — always set to LX_FALSE
- *lx_serv_area_p* — a handle for the allocated Service Area

Return Codes

A return code that indicates the outcome of processing is passed back to the application in the *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_MEM_ALLOC_ERR — memory allocation error

Application Programming Guidelines

By default, Linguistic Tools will assume at initialization that all input data is in code page 850 or 500 (according to the platform). If the application wishes to use another code page, it must call some function for which a code page value is expected on input in the *lx_num_cpg* field of the Linguistic Tools control block. If the application wishes to use a code page or translate tables not included in the Linguistic Tools default set, then it must use **NlpRegisterCodePage()** to provide this information to Linguistic Tools.

© Copyright IBM Corp. 2000

Terminate

Deactivate all dictionaries and deallocate all memory associated with the Service Area specified in the function call. If the application has called Linguistic Tools more than once, it will need to terminate each instantiation of Linguistic Tools separately.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function.
- *lx_func_code* — set to LX_TERMINATE_NLPS
- *lx_rqst_type* — set to LX_NEW_REQ

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated:

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_serv_area_p* — set to null, since the Service Area is deallocated

Return Codes

A return code that indicates the outcome of processing is passed back to the application in the *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle

Activate Dictionary

Makes a specified dictionary available to Linguistic Tools for processing. Returns a handle which the application must use to specify that dictionary to other Linguistic Tools functions.

This function activates only one dictionary at a time. Additional dictionaries must be activated by additional calls to the activate function. A maximum of twenty dictionaries may be active at any one time. The order in which dictionaries are activated is reflected in the value of the dictionary handle that is returned; however, for any Linguistic Tools function which uses dictionaries, an application may arrange the handles in the input dictionary list to use the dictionaries in any order desired.

The same function is used to activate both system and addenda dictionaries. Note that the maximum of twenty active dictionaries stated above includes both system and addenda dictionaries.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area handle returned from the **Initialize** function
- *lx_func_code* — set to LX_ACTIVATE_DICT
- *lx_rqst_type* — set to LX_NEW_REQ
- *lx_num_cpg* — number of code page that input text is in. Required for flat-file addenda and stopword dictionaries.
- *lx_reply_size*
- *lx_reply_p*
- *lx_dict_names_p* — pointer to a null-terminated string containing the dictionary name.

  If an unqualified file name is given for the dictionary, the default search path will be used in an attempt to locate the dictionary. For more details, see "File Names" on page 106 and "Dictionary Search Path" on page 108.
- *lx_alg_hyph_f* — flag indicating whether algorithmic hyphenation rules should be loaded
  - LX_FALSE — do not load algorithmic hyphenation rules
  - LX_TRUE — load algorithmic hyphenation rules

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_rc* — return code
- *lx_reply_used*
- *lx_reply_p* — will be empty if return code is anything other than LX_RC_OK, LX_BAD_WORD, or LX_DUP_WORD

  The reply area will contain a structure of type LX_ACTIVATE_REPLY (defined in one of the files included under EFZLNLPS.H), which contains following fields:

– Dictionary token field — contains a handle which will be used to refer to a particular dictionary in all other functions (type LX_UCHAR).

– Dictionary type field — indicates the type of dictionary activated (type LX_UCHAR). This field can have one of the following values:

  LX_LDDTGEN — general system dictionary
  LX_LDDTMED — medical system dictionary
  LX_LDDTLEGL — legal system dictionary
  LX_LDDTADD — regular (flat file) addenda dictionary
  LX_LDDTADB — binary addenda dictionary
  LX_LDDTSTW — (flat file) stopword addenda dictionary
  LX_LDDTSTB — binary stopword dictionary
  LX_LDDTABR—flat file abbreviation dictionary
  LX_LDDTABB—binary abbreviation dictionary

– Dictionary content field — describes the information contained in the dictionary (type LX_UCHAR). This field is a bit-mask, with some combination of the following values bitwise ORed together:

  LX_LDCTHYP — hyphenation data present
  LX_LDCTGRD — grade level data present
  LX_LDCTSYN — synonym data present
  LX_LDCTPOS — part-of-speech data present
  LX_LDCTBOF — compound word data present (BOFA data)
  LX_LDCTCLS — morphology data present
  LX_LDCTALG — algorithmic hyphenation rules present
  LX_LDCTPHN — phonetic spellaid rule data present

– Dictionary version number field — provides a version number to identify the level of a dictionary (type LX_UCHAR).

– Language code field — Code specifying the language of the active dictionary (type LX_USHORT).

  LX_AFRIKAANS — Afrikaans
  LX_ARABIC — Arabic
  LX_AUSTRALIAN Australian English
  LX_CATALAN — Catalan
  LX_CHINESE_TRAD — Chinese (Traditional)
  LX_CHINESE_SIMP — Chinese (Simplified)
  LX_CZECH — Czech
  LX_DANISH — Danish
  LX_DUTCH — Dutch
  LX_UK_ENGLISH — U.K. English
  LX_US_ENGLISH — U.S. English
  LX_FINNISH — Finnish
  LX_NAT_FRENCH — National French
  LX_CAN_FRENCH — Canadian French
  LX_GERMAN — German
  LX_SWISS_GERM — Swiss German
  LX_GREEK — Greek
  LX_HEBREW — Hebrew
  LX_HUNGARIAN — Hungarian
  LX_ICELANDIC — Icelandic
  LX_ITALIAN — Italian
  LX_JAPANESE — Japanese

LX_KOREAN — Korean
LX_NO_LANGUAGE — No language
LX_NYN_NORWAY — Norwegian—Nynorsk
LX_BOK_NORWAY — Norwegian—Riksmal/Bokmål
LX_POLISH — Polish
LX_NAT_PORTGS — Portuguese
LX_BRAZ_PORT — Brazilian Portuguese
LX_RUSSIAN — Russian
LX_SPANISH — Spanish
LX_SWEDISH — Swedish
LX_THAI — Thai
LX_TURKISH — Turkish

– Pointer to dictionary name—Pointer to null-terminated dictionary name (type LX_PUCHAR)

Return Codes

A return code that indicates the outcome of processing is passed back to the application in the *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT — bad dictionary; no recognizable header in file submitted as a dictionary
LX_BAD_DICT_NME — invalid dictionary name
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_DUP_DICT — duplicate dictionary
LX_DUP_WORD — duplicate word added to addenda dictionary
LX_END_OF_STORAGE — end of system storage reached
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_FL_ACC_DEND — file access denied
LX_FL_NO_FND — file not found
LX_FL_OP_FAILD — file open failed
LX_FL_SHR_VIOLTN — file sharing violation
LX_IO_ERROR — file input/output error
LX_MEM_ALLOC_ERR — memory allocation error
LX_NO_REPLY_P — missing reply area
LX_REPLY_2_SMALL — reply area is too small to allow any output
LX_2_MANY_DICTS — too many dictionaries open

Application Programming Guidelines

Typically an application would display a list of available dictionaries, indicating which ones are currently active and in what order they will be accessed. From this, the application would allow the user to activate, deactivate or reorder entries on the list. If any dictionaries from the updated list need to be picked up, these would be specified to the Linguistic Tools via the activate dictionary function.

On the workstation an application may wish to improve the performance of dictionary searches by limiting the number of directories searched when a

dictionary name is unqualified. The application may do this by finding the directories which are known to contain dictionaries and requesting that Linguistic Tools restrict further searches to only these directories. For more details, see "NlpFindDicts" on page 211 and "NlpSetSearchPath" on page 228.

In order to use algorithmic hyphenation rules in the **Hyphenate** function, these rules must be loaded when the dictionary is activated. This is done by setting *lx_alg_hyph_f* to LX_TRUE.

In order to improve performance when activating addenda dictionaries, it is recommended that the application use binary format addendas whenever possible. (See "Activating and Deactivating Addenda Dictionaries" on page 19 for details.)

Deactivate Dictionary

Deactivates the specified dictionary, making it inaccessible to Linguistic Tools.

This function deactivates only one dictionary at a time.

This function does not save addenda dictionaries before deactivating them. In order to save changes made via **Add Word**, **Add Morph**, or **Remove Word**, the application must call **Save Addenda** before calling **Deactivate**. See "Save Addenda" on page 137 for more information.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_DEACTIVATE_DICT
- *lx_rqst_type* — set to LX_NEW_REQ
- *lx_dict_tkns_ct* — must be one
- *lx_dict_tkns* — input dictionary list (can contain only one dictionary handle)

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated:

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in the *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

- LX_BAD_FUNCT_CODE — invalid function code
- LX_BAD_RQST_TYPE — invalid request type
- LX_RC_OK — processing successfully completed
- LX_UNEXPECTED_RC — unexpected or unknown condition
- LX_BAD_SERV_HANDLE — invalid Service Area Handle
- LX_BAD_DICT_TKN — invalid dictionary token

- LX_BAD_DICT_TKN_CNT — invalid dictionary token count

Application Programming Guidelines

An application may wish to make accessible only some of the dictionaries currently activated by Linguistic Tools. For instance, a multilingual document may require different dictionaries for different sections. Rather than pay the performance price of activating and deactivating dictionaries for each section, the application can alter the dictionary token list to include only those dictionaries needed at any given time.

Chapter 7. Addenda (User) Dictionary Support Functions

Create Addenda Dictionary

Allows a user to create an addenda dictionary using the file name passed by the calling function.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_CREATE_ADDENDA
- *lx_rqst_type* — set to LX_NEW_REQ
- *lx_reply_size*
- *lx_reply_p*
- *lx_dict_names_p* — Pointer to a null-terminated string containing the dictionary name (type LX_PUCHAR).
- *lx_lang_code* — the language of words to be added to the addenda dictionary.

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_reply_used*
- *lx_reply_p* (type LX_PUCHAR) — not used if the return code is anything other than LX_RC_OK

  The reply area will contain a structure of type LX_ACTIVATE_REPLY (defined in one of the files included under EFZLNLPS.H), which contains the following fields:

  – Dictionary token field — contains a handle which will be used to refer to this dictionary in all other functions (type LX_UCHAR).

  – Dictionary type field — indicates the type of addenda dictionary created (type LX_UCHAR). This field will only have the following value:

    LX_LDDTADB — binary addenda dictionary

  – Dictionary content field — describes the information contained in the dictionary (type LX_UCHAR). This field is a bit-mask, with some combination of the following values bitwise ANDed together:

    LX_LDCTHYP — hyphenation data present
    LX_LDCTSYN — synonym data present
    LX_LDCTPOS — part-of-speech data present

© Copyright IBM Corp. 2000

LX_LDCTCLS — morphology data present

– Dictionary version number field — provides a version number to identify the level of the dictionary (type LX_UCHAR).

– Language code field — Code specifying the language of the active dictionary (type LX_USHORT).

LX_AFRIKAANS — Afrikaans
LX_AUSTRALIAN Australian English
LX_CATALAN — Catalan
LX_CHINESE_TRAD — Chinese (Traditional)
LX_CHINESE_SIMP — Chinese (Simplified)
LX_CZECH — Czech
LX_DANISH — Danish
LX_DUTCH — Dutch
LX_UK_ENGLISH — UK English
LX_US_ENGLISH — US English
LX_FINNISH — Finnish
LX_FRENCH — French
LX_CAN_FRENCH — Canadian French
LX_GERMAN — German
LX_SWISS_GERM — Swiss German
LX_GREEK — Greek
LX_HUNGARIAN — Hungarian
LX_ICELANDIC — Icelandic
LX_ITALIAN — Italian
LX_JAPANESE — Japanese
LX_KOREAN — Korean
LX_NYN_NORWAY — Norwegian — Nynorsk
LX_BOK_NORWAY — Norwegian — Riksmal/Bokmål
LX_POLISH — Polish
LX_NAT_PORTGS — Portuguese
LX_BRAZ_PORT — Brazilian Portuguese
LX_RUSSIAN — Russian
LX_SPANISH — Spanish
LX_SWEDISH — Swedish
LX_TURKISH — Turkish

– Pointer to dictionary name — Pointer to null-terminated dictionary name (type LX_PUCHAR)

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

- LX_BAD_FUNCT_CODE — invalid function code
- LX_BAD_RQST_TYPE — invalid request type
- LX_RC_OK — processing successfully completed
- LX_UNEXPECTED_RC — unexpected or unknown condition
- LX_BAD_SERV_HANDLE — invalid Service Area Handle
- LX_BAD_DICT_NME — invalid dictionary name
- LX_BAD_LANG_CODE — invalid language code
- LX_BAD_PARM — missing or illegal input value for parameter
- LX_DUP_DICT — duplicate dictionary

- LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
- LX_FL_OP_FAILD — file open failed
- LX_INVLD_NUM_DICT — illegal Number of dictionaries
- LX_MEM_ALLOC_ERR — memory allocation error

Application Programming Guidelines

The application should allow the user to determine enough of the addenda dictionary name so the user can take advantage of any security that exists on a system (such as RACF).

If the application wishes to be able to use the **NlpFindDicts()** utility, it must the observe the naming conventions described in "Linguistic Tools Default Dictionary Naming Conventions" on page 108. If the application chooses to use its own default extensions/file types/qualifiers, it must specify these by means of the **NlpSetDictDef()**function. If the application does not wish to make use of the **NlpFindDicts()** utility, Linguistic Tools will not require it to follow any particular naming conventions. However, if the input file name has no extension/file type/qualifier, Linguistic Tools will assume that the default extension/file type/qualifier (set via **NlpSetDictDef()**) should be used.

The **Create Addenda** function can accept either an unqualified or a fully-qualified file name file name. On workstations, if an unqualified file name is given, the file will be placed in the current folder or directory. On VM systems, if no file type or file mode is given, Linguistic Tools will supply the default file mode of A.

The newly-created addenda dictionary will be a binary file containing just a header. **Create Addenda** will leave this dictionary active; the application need not call **Activate Dictionary** in order to use the new addenda.

To put data into the dictionary the application should call **Add Word to Addenda**. If morphological information for a word is to be stored, the **Add Morphology to Addenda** function should be used. In order to create a new addenda dictionary in editable flat-file format, the application must first call **Create Addenda**, then use the flat-file option with the **Save Addenda** function. When the addenda is saved as a flat file, the header is converted to one line containing some identifying information followed by a language name. This line should not be removed or modified.

If the input file name duplicates an existing file name, LX_DUP_DICT will be returned.

Add Word to Addenda

Allows the application to store words and their related data in an active addenda dictionary. The user must specify the handle of an active addenda dictionary. The words will be available for later use by other Linguistic Tools functions during the current instantiation of the Linguistic Tools.

The data for the words may include synonyms, a morphology alias, and user- or application-supplied data, for use by **Look Up Word**, **Synonym Aid**, **Generate Inflected Forms** and **Morphological Identification**. Syllable hyphenation points may also be indicated, for use by the **Hyphenate** function. For Chinese, Japanese, and Korean only, the data may include a DBCS parsing code (DBPC) for use by **Lexical Analysis**. European-language style **Morphological Identification** and **Synonym Aid** are not supported for the double-byte languages (Chinese, Japanese, and Korean). If the addenda dictionary is one of the special abbreviation addenda dictionaries, a flag may be set to indicate that a particular abbreviation is allowed at the end of a sentence.

The word and its data may be saved to the addenda dictionary on disk by calling the **Save Addenda** function at any time during a Linguistic Tools session.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_ADDWORD2_ADDENDA
- *lx_rqst_type* — set to LX_NEW_REQ
- *lx_num_cpg* — Number of code page that data is in
- *lx_dict_tkns_ct* — Must be 1 for this function.
- *lx_dict_tkns* — input dictionary list. Must contain the dictionary handle assigned by the Activate Dictionary function.
- *lx_conf_res* — specifies how to resolve a conflict when the word to be added is already present in the addenda dictionary:
  - LX_FLAG_ERROR — leave the data for the existing word unchanged, but set the data element's lx_rc field to LX_DUP_WORD.
  - LX_REPLC_DATA — replace the existing data with the new data pointed to by *lx_info_p*.
  - LX_MERGE_DATA — merge the existing data with the new data pointed to by *lx_info_p* by using the data items in the input data to replace the same data items in the existing data, while leaving the rest of the existing data unchanged. See Chapter 3, "Addenda (User) Dictionary Support Functions" on page 17 for more information regarding the types of data items which may be stored in an addenda dictionary.
- *lx_ele_format* — must be LX_SINGLE_WORD_FMT
- *lx_elements_ct* — Contains the number of words to be added to the addenda dictionary

- *lx_elements_p* — Pointer to the data element list which will contain the words to be added to the addenda dictionary as well as the data to be associated with each word. The fields in the data element list structure that are used for the Add Word function are as follows:

  lx_data_len — length in bytes of the word to be added.

  lx_data_p — pointer to the word to be added.

  lx_info_p — pointer to a structure (type LX_ADD_INFO) containing the data to be associated with the word. The fields of the structure are as follows:

  – *HyphLength* (type LX_USHORT) — length of the Hyphenation data.

  – *HyphData* (type LX_PUCHAR) — pointer to the Hyphenation data, represented by a copy of the word from lx_data_p with all hyphenation points indicated. If this pointer is NULL, and the word pointed to by *lx_data_p* contains hyphenation, the hyphenation points from *lx_data_p* will be used. If this pointer is NULL, and the word pointed to by *lx_data_p* contains no hyphenation, no hyphenation points will be stored. If this pointer is non-NULL, it will override whatever hyphenation may be present in the word pointed to by *lx_data_p*.

  – *MorpLength* (type LX_USHORT) — length of the Morphology Alias.

  – *MorpData* (type LX_PUCHAR) — pointer to the Morphology Alias. Only a single word is allowed; surrounding white space is ignored.

  – *DBPCLength* (type LX_USHORT) — length of the DBCS parse code data.

  – *DBPCData* (type LX_PUCHAR) — pointer to the DBCS parse code data. Multiple DBPCs are allowed, each separated by a comma; surrounding white space is ignored.

  – *SynLength* (type LX_USHORT) — length of the Synonym data.

  – *SynData* (type LX_PUCHAR) — pointer to the Synonym data. Multiple synonyms are allowed, each separated by a commoa; white space is ignored before or after the comma. Each synonym may be a word or a phrase.

  – *AbbrEOS* (type LX_USHORT) — flag: LX_TRUE indicates that word is an abbreviation which may end a sentence. LX_FALSE indicates that word is an abbreviation which may not end a sentence.

  – *UserLength* (type LX_USHORT) — length of the User-defined data.

  – *UserData* (type LX_PUCHAR) — pointer to the User-defined data. Any additional data to be stored with the word. White space is not ignored.

    See Chapter 3, “Addenda (User) Dictionary Support Functions” on page 17 for more information regarding the types of data items which may be stored in an addenda dictionary.

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function

- *lx_elements_p* — the following field of each input data element is updated:
  – *lx_rc*

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_DICT_NO_UPDT — non-updateable dictionary cannot be used with this function
LX_END_OF_STORAGE — End of system storage reached
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library

Application Programming Guidelines

This function can accept multiple single-word format data elements as input.

The primary purpose of this function is to allow the application to add words to an addenda dictionary which do not exist in a system dictionary. However, the application may also update existing information in an addenda by calling **Look Up Word** to retrieve the existing data for a word, making the desired modifications to the data, and calling **Add Word** with lx_conf_res set to LX_REPLC_DATA or LX_MERGE_DATA to place the new data for the word into the addenda dictionary.

If the application wishes to provide information for use by the **Hyphenate** function, it should be indicated by including the optional hyphenation points in the word. (See "Ranked Hyphenation" on page 28 for information on how to indicate different kinds of hyphenation points).

Synonym data can include several synonyms for the word, with individual synonyms separated by commas. An individual synonym can be a single word or a phrase. This data will be returned by **Synonym Aid**. (See "Synonym Aid" on page 157 for details.)

For a Chinese, Japanese, or Korean word it is possible to have multiple DBCS parsing codes (DBPCs) separated from each other by commas. These parse codes will be used by the **Lexical Analysis** function. Care must be exercised in the assignment of parse codes. The choice of code for a particular word may affect more than just that word; it may change the way in which the preceding or following text is parsed.

Morphology information consists of a single alias term, whose pattern of inflectional spelling changes will be applied to the new word. The intent of allowing morphology information to be input is to provide the application with a means of adding new morphology information or altering or removing existing morphology

information. The preferred means for entering morphology information for a word is through a call to **Add Morphology to Addenda**. (See "Adding Morphological Information to Addenda Dictionaries" on page 23 and "Add Morphology to Addenda" on page 128 for more information.)

The user- or application-supplied data field may contain any data; Linguistic Tools will simply return it as-is when the word is retrieved using **Look Up Word**. However, if the application wishes to allow the user to convert a binary addenda to flat (text) format, the user-data must contain only flat text.

The word and its data will be converted to the Linguistic Tools internal code page when they are added to the addenda dictionary and converted back when retrieved. This allows the binary addenda dictionary to be portable to other systems. The only exception to this is application-supplied data, which is not altered in any way. An application which intends to use binary addenda dictionaries across multiple platforms should be sensitive to this.

The addenda words may be entered in any supported code page; however, all words passed on a single call must be in the same code page. For Latin I Linguistic Tools supports code pages 850 and 500; both are converted to the same internal code page. The **Add Word** function can accept host code page input to add to a binary addenda dictionary originally created from a workstation code page flat file, or vice versa. However, attempting to add words to this addenda dictionary in a code page which represents a different language from the code page used to create the addenda is not guaranteed to produce the expected result. For example, an ideogram which is used by all DBCS languages (Traditional and Simplified Chinese, Japanese, and Korean) may be represented by a different internal code point for each language.

If an addenda dictionary is to be saved to a flat file, all words will be converted to the code page specified in the call to the **Save Addenda** function.

If any serious error occurs, the function stops processing the input words. If a word-level error occurs (such as invalid characters), that word is not inserted, but the function continues processing with the remaining input words. The return code of the function will be the return code for the last word that was not added to the addenda dictionary. If all specified words are successfully added, the return code will be LX_RC_OK.

Add Morphology to Addenda

Allows a user to add to an active addenda dictionary one word with associated morphological information. This information is specified by means of an alias — that is, a word in the system dictionary which has the same set of inflections.

The user must specify the handle of an active addenda dictionary. Words which are added in this way will be available for use during the current instantiation of the Linguistic Tools. The words may be saved to the addenda dictionary on disk by calling the **Save Addenda** function at any time during a Linguistic Tools session.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_ADD_MORPH
- *lx_rqst_type* — set to LX_NEW_REQ
- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_dict_tkns_ct* — Always one
- *lx_dict_tkns* — input dictionary list. Must contain the dictionary handle assigned by the Activate Dictionary function.
- *lx_word_morph_reply_size* — number of bytes used in the **Generate Inflected Forms** (GIF) reply area for new word
- *lx_word_morph_reply_p* — pointer to GIF reply area for new word
- *lx_alias_morph_reply_p* — pointer to GIF reply area for alias word

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_ALIAS_NO_MATCH — mismatch between alias and new word forms — the **Add Morphology to Addenda** function expects the same number of forms in both input reply areas

LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_DICT_NO_UPDT — non-updateable dictionary cannot be used with this function
LX_END_OF_STORAGE — end of system storage reached
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_NO_REPLY_P — missing reply area

Application Programming Guidelines

This function is intended to be used by a linguistically-sophisticated user. An incorrect choice of morphological alias can result in incorrect results from other Linguistic Tools functions such as **Spell Verify**, **Morphological Identification** and **Generate Inflected Forms**.

To use this function the application should:

1. Request the user to supply an alias term.
2. Call **Generate Inflected Forms (GIF)** with the user-supplied alias as the input word.
3. Display the output to the user. If more than one set of inflected forms was produced, this means that the alias term has more than one part of speech and/or paradigm. If not all of these sets are desired, the application should request the user to supply another alias which will produce only the desired forms.
4. Call **GIF** again with the new word as the input word and the address of the user-supplied alias in the *lx_alias_morph_reply_p* field. A different reply area should be allocated for this call. **GIF** will apply the spelling changes from the alias paradigm to the new word and output the results to the second reply area.
5. Display the output of the second reply area and request the user to confirm that this is what was desired.
6. Call **Add Morphology to Addenda (AddMorph)** with pointers to both reply areas. **AddMorph** will go through both reply areas, matching each form of the new word to the corresponding form of the alias word, and put the resulting word/alias pairs into the specified addenda dictionary. If the number of forms in the two reply areas is not the same, there will be a mismatch between forms. In this case, **AddMorph** will return error code LX_ALIAS_NO_MATCH and not attempt to pair up the forms.

Note: In languages with many inflected forms, using this function may result in a considerable number of entries being added to the addenda dictionary.

Remove Word from Addenda

Allows an application to remove words and associated data from an active addenda dictionary. The application must specify the handle of an active addenda dictionary. These changes will only be saved to disk by calling the **Save Addenda** function.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_REMOVE_ADD_WORD
- *lx_rqst_type* — set to LX_NEW_REQ
- *lx_num_cpg* — Number of code page that data is in
- *lx_dict_tkns_ct* — Always one
- *lx_dict_tkns* — input dictionary list. Must contain a dictionary handle assigned by the Activate Dictionary function.
- *lx_ele_format* — must be LX_SINGLE_WORD_FMT
- *lx_elements_ct* — number of words to be removed from the addenda dictionary
- *lx_elements_p* — Pointer to the data element list which will contain the words to be removed from the addenda dictionary. The fields in the data element list structure that are used for the Remove Word function are as follows:
  - *lx_data_len* — length in bytes of the word to be removed.
  - *lx_data_p* — pointer to the word to be removed.

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_elements_p* — the following field of each input data element structure is updated:
  - *lx_rc*

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count

LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_DICT_NO_UPDT — non-updateable dictionary cannot be used with this function
LX_END_OF_STORAGE — end of system storage reached
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_WORD_NOT_FOUND — word not found in dictionary

Application Programming Guidelines

This function can accept multiple single-word format data elements as input.

A word will be removed only if it is an exact match for a word in the specified dictionary.

List Addenda Words

Lists out all words in the specified addenda dictionary.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — LX_LIST_ADD_WORD
- *lx_rqst_type*
- *lx_dict_tkns_ct*
- *lx_dict_tkns* — input dictionary list. Must contain a dictionary handle returned by the Activate Dictionary function.
- *lx_num_cpg* — Number of code page that data should be returned in
- *lx_reply_p* — pointer to reply area
- *lx_reply_size*

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_cont_reply_f*
- *lx_reply_used*
- *lx_delivered_out_units* — number of words placed in the reply area
- *lx_reply_p* — For each word the reply area will contain:
  - A contents flag

    This is a bit-flag indicating the types of data available for the word (type LX_USHORT):
    - LX_HYPHSET = Hyphenation Data present
    - LX_MORPSET = Morphology (alias) Data present
    - LX_DBPCSET = DBCS parse code (DBPC) Data present
    - LX_SYNOSET = Synonym Data present
    - LX_USERSET = User Data present
    - LX_ABBRSET = Abbreviation End-of-Sentence Flag present
  - A word length (type LX_UCHAR)
  - The word itself (a string of LX_UCHAR)

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed

LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_CPG_NOT_SUPPORTED — code page not supported
LX_DICT_NO_UPDT — non-updateable dictionary cannot be used with this function
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_FUNC_CODE_CHG — change of function code not allowed
LX_NO_REPLY_P — missing reply area
LX_REPLY_2_SMALL — reply area is too small to allow any output
LX_REPLY_FULL — reply area is full

Application Programming Guidelines

This function has *continue reply* capability. If there is more output than can fit into the reply area, a return code of LX_REPLY_FULL and an lx_cont_reply_f value of LX_EXPECT_CONT_REP will be returned. The application should process the output in the reply area and then call the function again with a request type of LX_CONT_REPLY to obtain additional output from Linguistic Tools.

This function will list only the entry words in the specified addenda dictionary. It will not pick up any data associated with them. This is not intended to be a means to “unfold” the addenda dictionary for the user to view or edit. In order to retrieve individual entries, complete with associated data, the application should call **Look Up Word**.

To obtain a user-editable version of an addenda dictionary, the application should call the **Save Addenda** function with the flat-file option.

Look Up Word in Addenda

This function will search through all active dictionaries, in the order in which they occur in the input dictionary list, until it finds the word or reaches the end of the dictionary token list. If the word is found first in an addenda dictionary, the word and all associated information will be retrieved and put in the reply area. If the word is found first in a system dictionary, or if lx_reply_p is null, only the token of the dictionary in which it was found will be returned; nothing is put into the reply area.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p*—Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_LOOKUP_WORD
- *lx_rqst_type* — set to LX_NEW_REQ
- *lx_dict_tkns_ct*
- *lx_dict_tkns* — input dictionary list
- *lx_num_cpg* — number of code page which data is in
- *lx_elt_format* — must be LX_SINGLE_WORD_FMT
- *lx_elements_ct* — contains the number of words to be looked up; must be one
- *lx_elements_p* — pointer to the data element list containing the word to be looked up
- *lx_reply_p* — pointer to reply area (optional)
- *lx_reply_size* (optional)
- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function

Output

The updated Linguistic Tools control block. The following output fields or the areas they point to are updated:

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_dict_found_in* — dictionary token indicating the dictionary in which the word was found
- *lx_delivered_out_units* — should be 1 if the word was found in an addenda dictionary
- *lx_reply_used*
- *lx_reply_p*

  If the word was found in an addenda dictionary, the reply area will contain a structure (type LX_ADD_INFO) having the following fields:

  – *HyphLength* (type LX_USHORT) — length of the Hyphenation data.

- *HyphData* (type LX_PUCHAR) — pointer to the Hyphenation data, represented by a word with all hyphenation points indicated.
- *MorpLength* (type LX_USHORT) — length of the Morphology Alias.
- *MorpData* (type LX_PUCHAR) — pointer to the Morphology Alias.
- *DBPCLength* (type LX_USHORT) — length of the DBCS parse code data.
- *DBPCData* (type LX_PUCHAR) — pointer to the DBCS parse code data. Multiple DBPCs may be present, each separated by a comma.
- *SynLength* (type LX_USHORT) — length of the Synonym data.
- *SynData* (type LX_PUCHAR) — pointer to the Synonym data. Multiple synonyms may be present, each separated by a comma. Each synonym may be a word or a phrase.
- *AbbrEOS* (type LX_USHORT) — flag: LX_TRUE indicates that word is an abbreviation which may end a sentence. LX_FALSE indicates that word is an abbreviation which may not end a sentence.
- *UserLength* (type LX_USHORT) — length of the User-defined data.
- *UserData* (type LX_PUCHAR) — pointer to the User-defined data. Any additional data stored with the word, returned exactly as entered.

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_IO_ERROR — file input/output error
LX_NO_REPLY_P — missing reply area
LX_REPLY_2_SMALL — reply area is too small to allow any output
LX_WORD_NOT_FOUND — word not found in dictionary

Application Programming Guidelines

In order for a dictionary word to be found, the input word must be an exact match, including case. This function will not convert between upper and lower case, nor does it perform any language-specific processing.

This is a single word function, meaning that only the first word passed to the function will be processed and that the input data element list must be in single-word format. See “Use of Single-Word Format Data Elements” on page 71 for a description of how input words may be passed via a single-word format data element list.

The hyphenated word in the reply area will be a copy of the word with all optional hyphenation points indicated. (See "Ranked Hyphenation" on page 28 for information on how different kinds of hyphenation points are indicated).

Synonym data may include several synonyms for the word, with individual synonyms separated by commas. An individual synonym may be a single word or a phrase. This data will be returned by **Synonym Aid**.

Double-byte parse code (DBPC) data may include one or more parsing codes, with individual codes separated by commas. DBPCs have been identified only for Chinese, Japanese, and Korean. These are documented in "Chinese Double-Byte Parsing Codes (DBPCs)" on page 287, "Japanese Part of Speech (POS) Codes" on page 324, and "Korean Double-Byte Parsing Codes (DBPCs)" on page 331, respectively. DBPC information will not be present for any other languages.

Morphology information will consist of a single alias term, whose pattern of inflectional spelling changes will be applied to the new word. (See "Adding Morphological Information to Addenda Dictionaries" on page 23 and "Add Morphology to Addenda" on page 128 for more information.)

The user- or application-supplied data field may contain any data; Linguistic Tools will simply return as-is whatever was stored on a previous call to **Add Word**.

See Chapter 3, "Addenda (User) Dictionary Support Functions" on page 17 for general information about Linguistic Tools addenda dictionary support and for information about entering addenda dictionary data in a flat addenda dictionary.

Note that the input dictionary list is searched in order. If a word is in a system dictionary, but has also been put into an addenda dictionary with associated data, the word in the system dictionary will be found first and no data returned if the system dictionary precedes the addenda dictionary in the input dictionary list. In order to obtain the information stored in the addenda dictionary entry for this word, it is necessary to put the addenda dictionary token ahead of the system dictionary token in the input dictionary list (in the *lx_dict_tkns* field of the LTCB).

Save Addenda

Allows the application to save to disk any changes which have been made to an active addenda dictionary via the **Add Word**, **Add Morphology**, and **Remove Word** functions.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_SAVE_ADDENDA
- *lx_rqst_type* — set to LX_NEW_REQ
- *lx_num_cpg* — number of code page that dictionary data is in. Required only when lx_add_form is LX_SAVE_FLAT.
- *lx_dict_tkns_ct* — must be one
- *lx_dict_tkns* — specifies the addenda dictionary which is to be saved to disk
- *lx_dict_names_p* — pointer to a null-terminated dictionary name. Set to NULL to retain the same file name.
- *lx_add_form_f* — specifies the format in which the addenda dictionary is to be saved
  - LX_SAVE_BINARY — save in binary format only
  - LX_SAVE_FLAT — save as flat file only
  - LX_SAVE_SAME — save in same format as file from which it was loaded
- *lx_add_type_f* — specifies the type of addenda dictionary to be saved
  - LX_ADDENDA — save as regular addenda
  - LX_STOPWORD — save as stop word addenda
  - LX_ABBREVADD — save as abbreviation addenda
  - LX_SAME_TYPE — save as same type as file from which it was loaded

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle

LX_BAD_DICT_NME — invalid dictionary name
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_PARM — missing or illegal input value for parameter
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_FL_ACC_DEND — file access denied
LX_FL_OP_FAILD — file open failed
LX_FL_SHR_VIOLTN — file sharing violation

Application Programming Guidelines

This function ensures that any changes made to an addenda dictionary are saved to a permanent file. After this function is called, all changes that have been made to the dictionary via **Add Word**, **Add Morphology**, and **Remove Word** are saved to disk.

Note: The **Deactivate** function does not save addenda dictionary changes before deactivating, nor does **Terminate** save changes before terminating the session. **Save Addenda** is the only way for an application to save addenda dictionary changes to disk.

If the application has specified that the dictionary should be saved as a flat file, that file will be created in the code page specified by *lx_num_cpg*.

Saving the dictionary in binary format is much faster than saving as a flat file. It also ensures that the dictionary will be loaded much more quickly when it is activated again. Saving the dictionary in flat-file format ensures that the the user and/or application will have easy access to the file, but saving and reloading will be much slower than for a binary file.

An application may convert between binary and flat file by activating a dictionary in whatever format it was last saved in and saving it in the other format.

It is also possible to save the same addenda dictionary in both formats. This requires two calls to the **Save Addenda** function with different file names.

See “Create Addenda Dictionary” on page 121 for file naming guidelines.

The *lx_add_type_f* flag is used to select whether this file should be saved as a regular addenda dictionary (used by a number of functions), as a stop word addenda (used by the text extraction functions to filter out words which should never be used as keywords), or as an abbreviation addenda (used by the Text Segment Identification function to determine sentence boundaries). For most languages, stop word and abbreviation addendas are shipped as part of the Linguistic Tools; however, it is possible that a user might want either to edit the existing stop word or abbreviation dictionary or to create his/her own dictionary.

If the file name input to the function duplicates an existing file name, the existing file will be overwritten.

Chapter 8. Text-Processing Functions for Spelling Support

This chapter describes the text-processing functions of the Linguistic Tools.

Spell Verify

Attempts to find input words in specified dictionaries. If a word is found, it is considered to be correctly spelled; otherwise, it is marked as misspelled. All words are marked one way or the other.

This function is invoked through the same function code used for article checking; both may be invoked at the same time by setting both *lx_spell_ver_f* and *lx_art_checker_f* to LX_TRUE.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_rqst_type*
- *lx_func_code* — set to LX_SPELL_VERIFY
- *lx_spell_ver_f* — must be set to LX_TRUE for this function
- *lx_dict_tkns_ct* — number of dictionary handles in input dictionary list
- *lx_dict_tkns* — input dictionary list
- *lx_num_cpg* — number of code page that data is in
- *lx_elt_format* — may be either LX_SINGLE_WORD_FMT or LX_BLOCK_FORMAT
- *lx_elements_ct* — number of data elements in input data element list
- *lx_elements_p* — pointer to data element list containing the input text
- *lx_reply_p* — a Reply Area is only needed if lx_elt_format is set to LX_BLOCK_FORMAT
- *lx_reply_size* — used only if a Reply Area is provided

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated by the Spell Verify function (if the Article Checking function is invoked simultaneously, other fields may be updated as well):

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_cont_reply_f* — used to indicate whether the request is complete or more output is available
- *lx_reply_used* — used only if a Reply Area was provided

© Copyright IBM Corp. 2000

- *lx_delivered_out_units* — If lx_elt_format was set to LX_BLOCK_FORMAT, this indicates the number of output data elements placed into the Reply Area.
- *elements_p* — If lx_elt_format was set to LX_SINGLE_WORD_FMT, the following field of each data element is updated (unless an error prevents processing of all of the elements):
  - *lx_rc* — set to indicate whether the input word was found in any of the specified dictionaries, as follows:
    - LX_RC_OK — word found (spelled correctly)
    - LX_WORD_NOT_FOUND — word not found in dictionary
    - LX_BAD_WORD — invalid word

    Note that a data element may be "joined" by the Spell Verify function if its input flag field (*lx_input_f*) is set accordingly. The joining of elements may affect the *lx_rc* fields of the joined elements, as described in "Use of Single-Word Format Data Elements" on page 71.
- *lx_reply_p* — If lx_elt_format was set to LX_BLOCK_FORMAT, the Reply Area will be filled with single-word format data elements, each corresponding to a word or part of a word in the input text. The way in which the Linguistic Tools decides how the input text will be broken into single-word format elements is documented in "Use of Single-Word Format Data Elements" on page 71.

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_IO_ERROR — file input/output error
LX_WORD_NOT_FOUND — word not found in dictionary

If the input data element list format is in block format, the following return codes are also possible:

LX_ART_CHECKER_F_CHG — flag change not allowed on continue reply
LX_BAD_ELE_TYPE — invalid element type
LX_BAD_STR_ADDR — invalid or NULL string address provided
LX_ELT_FORMAT_CHG — change of element format not allowed
LX_END_OF_STORAGE — end of system storage reached
LX_FUNC_CODE_CHG — change of function code not allowed
LX_MEM_ALLOC_ERR — memory allocation error
LX_NO_REPLY_P — missing reply area
LX_REPLY_2_SMALL — reply area is too small to allow any output

LX_SPELL_VER_F_CHG — spell verify flag change not allowed on continue reply

Application Programming Guidelines

The data pointers within the data element list passed as input to Spell Verify may point directly into the document to be checked, rather than to strings containing copies of text from the document. The function will not alter the input text in any way.

If the input data element list contains block format elements, the Spell Verify function will build a single-word format data element list describing the input text, and put it into the Reply Area. If the same text is to be passed again as input to some Linguistic Tools function, the single-word format list provided by the service should be passed to avoid unnecessary processing.

The return indicator from Spell Verify is typically either "found" or "not found" (unless some other error occurs). No indication is given as to how close a match was found in the dictionary, which dictionary the matching word was found in, and so on

See "Case Matching between Input Words and Dictionary Words" on page 82 for details of how Linguistic Tools handles a difference in case between input words and dictionary words.

Applications which have the display capability should mark with highlighting or color those words which are not found. This indicates to the user the need to correct the word or add the word to an addenda dictionary. The usual scenario is for the user to correct the word directly or to request Spell Aid on a marked word and then select a correctly spelled word from the aid candidate list to replace the marked word.

The usual user options for spell verify include verifying an entire document, a page or range of pages, or a block of text (such as a paragraph).

See "Use of Single-Word Format Data Elements" on page 71 for a description of how input words are to be passed to the spell verify function through a single-word format data element list. "Input Data Element List" on page 69 contains information about the use of both single-word and block format data element lists. This function does not return LX_REPLY_FULL; the application must check the continue reply flag (lx_cont_reply_f) to determine whether there is additional output to be retrieved.

Article Checking

Linguistic Tools provides an option to check for article error — incorrect use of "a" or "an" — as part of the spelling verification process for U.S. Engilsh only. This option is selected by a flag in the call to **Spell Verify**.

Spell Aid

Attempts to collect from the specified dictionaries a list of words which resemble the input word. Both orthographic (letter-matching) and phonetic (similarity in pronunciation) criteria are used to determine the order in which candidates are presented.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_func_code* — set to LX_SPELL_AID
- *lx_rqst_type* — must be set to LX_NEW_REQ; Spell Aid does not support continue reply
- *lx_dict_tkns_ct* — number of dictionary handles in input dictionary list
- *lx_dict_tkns* — input dictionary list
- *lx_num_cpg* — number of code page that data is in
- *lx_elements_ct* — must be one
- *lx_elements_p* — pointer to data element list containing the word for which aid is being requested
- *lx_elt_format* — must be LX_SINGLE_WORD_FMT
- *lx_reply_p*
- *lx_reply_size*
- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_cont_reply_f* — always set to LX_FALSE
- *lx_reply_used*
- *lx_delivered_out_units* — set to the total number of candidates returned in the Reply Area (not including the input word)
- *lx_reply_p* — Spell Aid returns any candidates found in the Reply Area. A list of candidates is returned for each dictionary in which candidates were found. The first item in the Reply Area, if candidates were found, is always the length of the input word (type LX_UCHAR), followed by the input word. Then the following information is given for each dictionary in which candidates were found:
  - A header corresponding to the dictionary. This header contains:
    - The dictionary token (type LX_UCHAR)
    - An unused field (type LX_USHORT)
    - The input word's prefix length assumed when the dictionary was searched (type LX_UCHAR)

- The number of candidates from the dictionary (type LX_UCHAR) (Up to six candidates may be listed per dictionary.)

– Then, for each candidate from the dictionary:

- The candidate's length (type LX_UCHAR)
- The candidate itself (a string of LX_UCHAR)

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_IO_ERROR — file input/output error
LX_NO_AID_AVAIL — no aid available
LX_NO_REPLY_P — missing reply area
LX_REPLY_FULL — reply area is full

Application Programming Guidelines

Spell Aid is a single word function, meaning that only the first word passed to the function will be processed, and that the input data element list must be in single-word format. See "Use of Single-Word Format Data Elements" on page 71 for a description of how input words may be passed to the Spell Aid function via a single-word format data element list. See "Case Matching between Input Words and Dictionary Words" on page 82 for information on how this function handles different case patterns in input words.

In order to determine how much Reply Area space has been used by Spell Aid, the application may look at the *lx_reply_used* field of the Linguistic Tools control block. By counting bytes while interpreting Reply Area data, the application can know when to stop looking for more candidate lists in the Reply Area. The number of candidate lists in the Reply Area is not otherwise specified.

Fuzzy Spell Aid

The **Fuzzy Spell Aid** function provides suggested spellings for a given word by gathering from the specified addenda dictionary a list of words which resemble the input word. It differs from regular Spell Aid in the following ways:

- It is not as restrictive as regular Spell Aid in selecting candidate words
- The user can specify the maximum “distance measure” value to use in selecting candidates words (The “distance measure” is a numeric value computed to determine how distant (different) a candidate word is from the input word. A candidate word that is exactly the same as the input word will have a distance measure of zero.)
- It selects candidate words from a specified **addenda** dictionary only (Spell Aid selects from system and addenda dictionaries)
- It returns up to 20 candidate words (Spell Aid returns up to 6)

The candidate words selected from the addenda are ordered so that those which are thought to most closely resemble the input word are put at the top of the list.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following control block input fields are used:

- *lx_func_code* — set to LX_FUZZY_SPELL_AID
- *lx_rqst_type* — must be set to LX_NEW_REQ; fuzzy spell aid does not support continue reply
- *lx_dict_tkns_ct* — number of dictionary handles in input dictionary list (must be 1)
- *lx_dict_tkns* — input dictionary list (one addenda dictionary only)
- *lx_num_cpg* — number of code page that data is in
- *lx_elements_ct* — must be one
- *lx_elements_p* — pointer to data element list containing the word for which aid is being requested
- *lx_elt_format* — must be LX_SINGLE_WORD_FMT
- *lx_reply_p*
- *lx_reply_size*
- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_distance_meas* — maximum “distance measure” canadidates can have and still be included in the list of candidate words

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function

- *lx_cont_reply_f* — always set to LX_FALSE
- *lx_reply_used*
- *lx_delivered_out_units* — set to the total number of candidates returned in the Reply Area (not including the input word)
- *lx_reply_p* — Fuzzy Spell Aid returns any candidates found in the Reply Area. A list of candidates found in the addenda dictionary is returned. The first item in the Reply Area, if candidates were found, is always the length of the input word (type LX_UCHAR), followed by the input word. Then the following information is given for the addenda dictionary in which candidates were found.
  - A header corresponding to the dictionary. This header contains the dictionary token (type LX_UCHAR), an unused field (type LX_USHORT), the input word's prefix length assumed when the dictionary was searched (type LX_UCHAR), and the number of candidates from the dictionary (type LX_UCHAR).
  - Then, for each candidate from the dictionary, the candidate's length (type LX_UCHAR) and the candidate itself are given. Up to twenty candidates may be listed for the dictionary.

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

Upon processing completion, the Fuzzy Spell Aid function will pass a return code back to the application in field *lx_rc* of the LTCB. The following list describes the possible return codes.

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_DICT_NO_UPDT — non-updateable dictionary cannot be used with this function
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_FL_NO_FND — file not found
LX_IO_ERROR — file input/output error
LX_NO_AID_AVAIL — no aid available
LX_NO_REPLY_P — missing reply area
LX_REPLY_FULL — reply area is full

Application Programming Guidelines

Fuzzy Spell Aid is a single word function, meaning that only the first word passed to the function will be processed, and that the input data element list must be in single-word format. See "Use of Single-Word Format Data Elements" on page 71 for a description of how input words may be passed to the fuzzy spell aid function through a single-word format data element list. See "Case Matching between Input Words and Dictionary Words" on page 82 for information on how this function handles different case patterns in input words.

In order to determine how much Reply Area space has been used by fuzzy spell aid, the application may look at the *lx_reply_used* field of the LTCB. The number of candidate lists in the Reply Area is one.

Grade Level Analysis

This function attempts to find input words in specified dictionaries. If a word is found and grade level information is available for it, that information is returned. This function is supported only for U.S. English.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_rqst_type*
- *lx_func_code* — set to LX_GRADE_LEVEL
- *lx_dict_tkns_ct* — number of dictionary handles in the input dictionary list
- *lx_dict_tkns* — input dictionary list
- *lx_num_cpg* — number of code page that data is in
- *lx_elt_format* — may be either LX_BLOCK_FORMAT or LX_SINGLE_WORD_FMT
- *lx_elements_ct*
- *lx_elements_p* element is an 'S' format element
- *lx_reply_p* — the Reply Area needs to be provided only if lx_elt_format is set to LX_BLOCK_FORMAT
- *lx_reply_size* — used only if Reply Area provided

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_cont_reply_f*
- *lx_reply_used* — set only if lx_elt_format was set to LX_BLOCK_FORMAT
- *lx_delivered_out_units*—If lx_elt_format was set to LX_BLOCK_FORMAT, this indicates the number of output data elements placed into the Reply Area.
- *elements_p* — If *lx_elt_format* was set to LX_SINGLE_WORD_FMT, the following fields of each data element are updated (unless an error prevented processing of all of the elements):
  - *lx_rc* — set to indicate whether the input word was found in any of the specified dictionaries, as follows:
    - LX_RC_OK — word found and has grade level information
    - LX_NO_AID_AVAIL — no aid available
    - LX_NO_DATA_AVAIL — no data available (function is only supported for U.S. English)
    - LX_BAD_WORD — invalid word

- LX_WORD_NOT_FOUND — word not found in dictionary

  Note that a data element may be "joined" by the grade level function if its input flag field (*lx_input_f*) is set accordingly. The joining of elements may affect the *lx_rc* fields of the joined elements, as described in "Use of Single-Word Format Data Elements" on page 71.

– *lx_databyte* — contains the numeric value of the grade-level associated with this word

- *lx_reply_p* — If lx_elt_format was set to LX_BLOCK_FORMAT, the Reply Area will be filled with single-word format data elements, each corresponding to a word or part of a word in the input text. The way in which the Linguistic Tools decides how the input text will be broken into single-word format elements is documented in "Use of Single-Word Format Data Elements" on page 71. "Use of Single-Word Format Data Elements" on page 71. The fields of each data element contain information as described above.

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_ELE_TYPE — invalid element type
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ELT_FORMAT_CHG — change of element format not allowed
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_FUNC_CODE_CHG — change of function code not allowed
LX_IO_ERROR — file input/output error
LX_NO_AID_AVAIL — no aid available level information available for input word
LX_NO_DATA_AVAIL — no data available
LX_WORD_NOT_FOUND — word not found in dictionary

If the input data element list is in block format, the following return codes are also possible:

LX_BAD_STR_ADDR — invalid or NULL string address provided
LX_END_OF_STORAGE — end of system storage reached
LX_MEM_ALLOC_ERR — memory allocation error
LX_NO_REPLY_P — missing reply area
LX_REPLY_2_SMALL — reply area is too small to allow any output

Application Programming Guidelines

If the application has already isolated the input text into separate words, then the text should be passed as single-word format elements. Otherwise, the text should be passed as block format elements, causing Linguistic Tools to isolate the words as described in "Definition of a Word in SBCS Text" on page 80. If the input data element list contains block format elements, the Grade Level function will build a single-word format data element list describing the input text, and put it into the Reply Area. If the same text is to be passed as input to a Linguistic Tools function later, the single-word format list provided by the service should be passed to avoid unnecessary processing.

The data pointers within the data element list passed as input to the grade level function may point directly into the document to be checked, rather than to strings containing copies of text from the document. The function will not alter the input text in any way.

The lx_databyte field of the data-element structure contains the numeric value for the grade level associated with this word. Currently, these values range from 4 to 16 (grade 4 through college senior). If a word is in the dictionary but has no grade level associated with it, Linguistic Tools will return a grade level value of zero for it and a return code of LX_NO_AID_AVAIL. If the input word is unknown, the grade level value will be zero for that word, and it will have a return code LX_WORD_NOT_FOUND.

Lower and upper case forms are handled the same way as in the Spell Verify function ("Spell Verify" on page 139).

See "Use of Single-Word Format Data Elements" on page 71 for a description of how input words are to be passed to the Grade Level function via a single-word format data element list. "Input Data Element List" on page 69 also contains information about the use of both single-word and block format data element lists.

This function does not return LX_REPLY_FULL; the application must check the continue reply flag (lx_cont_reply_f) to determine whether there is additional output to be retrieved.

Chapter 9. Text-Processing Functions for Hyphenation

Hyphenation

Determines one (preferred) or all hyphenation points in an input word, as requested by the application.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_HYPHENATION
- *lx_rqst_type*
- *lx_reply_p*
- *lx_reply_size*
- *lx_num_cpg* — number of code page that data is in
- *lx_dict_tkns_ct* — number of tokens in dictionary list
- *lx_dict_tkns* — dictionary token list
- *lx_elt_format* — must be LX_SINGLE_WORD_FMT
- *lx_elements_ct* — number of words to be hyphenated; must be one
- *lx_elements_p* — pointer to the data element list containing the word to be hyphenated
- *lx_pref_hyphen_f* — flag indicating whether preferred hyphenation points are to be selected if available
  - LX_FALSE — do not select preferred hyphenation
  - LX_TRUE — select preferred hyphenation
- *lx_pref_range* — width of zone in which preferred hyphenation points are to be selected
- *lx_chars_left* — number of spaces remaining on line
- *lx_alg_only_f* — flag indicating whether algorithmic hyphenation rules should be used exclusively
  - LX_FALSE — algorithmic hyphenation should be used with dictionary-based hyphenation
  - LX_TRUE — algorithmic hyphenation should be used exclusively

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated:

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function

© Copyright IBM Corp. 2000

- *lx_delivered_out_units* — always 1
- *lx_reply_used*
- *lx_reply_p* — The reply area will contain:
  - hyphen position offset (LX_UCHAR)

    If only one hyphenation point is selected, this will indicate the position of that point within the word. If all hyphenation points were requested, the value of this will be x'FF'.
  - Word length (LX_UCHAR)
  - The word with all hyphenation-related spelling changes implemented (string of LX_UCHAR)

    If only one hyphenation point was requested, this form will not contain embedded hyphen characters. If all hyphenation points were requested, this form will contain embedded hyphenation information encoded as follows (ranked from highest to lowest):

  - (hard hyphen) required hyphen

  + **(plus)** compound word component boundary

  , (comma) special processing required

  = (equals) other preferred hyphenation point

  (syllable hyphen) syllable hyphen

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_NO_REPLY_P — missing reply area
LX_REPLY_2_SMALL — reply area is too small to allow any output
LX_SPEL_CHNG — dehyphenation or hyphenation changed spelling
LX_WORD_NOT_FOUND — word not found in dictionary

Application Programming Guidelines

No validity checking is performed on the input to this function. If a misspelled word (one that cannot be found in the dictionary) is input, the function will either apply the algorithmic hyphenation rules, or if algorithmic hyphenation is not available, will return LX_WORD_NOT_FOUND. If invalid input such as all-numeric strings is given, the function will attempt to process it; however, the results will be invalid.

If the input word is less than five characters in length, no processing will be done and no hyphenation points will be returned.

In order to apply algorithmic hyphenation rules, these rules must have been loaded when the dictionary was activated.

If *lx_chars_left* is set to a non-zero value, this function will return only one hyphenation point, determined by the value of this parameter. The location of this point within the word will be returned as the hyphen position offset.

If *lx_chars_left* is set to zero, all hyphenation points in the word will be returned within the word, and the hyphen position offset will have a value of x'FF'. If, in addition, preferred hyphenation has been selected, this information will be included in the word by means of the scheme described below. If preferred hyphenation has not been specified, all hyphenation points except for hard hyphens will be indicated by the syllable hyphen character. A return code of LX_SPEL_CHNG signals that a required spelling change has been made. For details of language-specific spelling changes, see Appendix D, “Language-Specific Processing” on page 281.

If preferred hyphenation is selected and *lx_pref_range* is equal to *lx_chars_left*, then only the “best” (highest-ranking) hyphenation point within the span of *lx_chars_left* will be returned. If preferred hyphenation is selected, but *lx_pref_range* is set to zero, Linguistic Tools will use the preferred hyphenation data, but ranking will not apply—that is, preferred hyphenation points will be found, but will not outrank other possible hyphenation points. If preferred hyphenation is selected and *lx_pref_range* is set to a value in the 3—5 range, Linguistic Tools will return a good, but not necessarily the best, hyphenation point within the word.

Preferred hyphenation points can only be selected if ranked hyphenation data are available, either in a system dictionary or in an active addenda dictionary. At present, ranked hyphenation information is included only in the Danish, Dutch (both preferred and modern), National German, Norwegian (both Bokmål and Nynorsk), and Swedish system dictionaries.

Dehyphenation

Dehyphenates a word and places the result in the reply area.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_DEHYPHENATION
- *lx_rqst_type*
- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_reply_p*
- *lx_reply_size*
- *lx_num_cpg* — number of code page that data is in
- *lx_elt_format* — must be LX_SINGLE_WORD_FMT
- *lx_elements_ct* — the number of words to be dehyphenated; must be one
- *lx_elements_p* — pointer to the data element list containing the word to be dehyphenated

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_reply_used*
- *lx_reply_p* — The reply area will contain:
  - hyphen position count (LX_UCHAR) — always 0
  - word length (LX_UCHAR)
  - dehyphenated word with any necessary spelling changes implemented (string of LX_UCHAR)

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_NO_REPLY_P — missing reply area
LX_REPLY_2_SMALL — reply area is too small to allow any output
LX_SPEL_CHNG — dehyphenation or hyphenation changed spelling

Application Programming Guidelines

This function will remove all hyphens from a word and make any language-specific spelling changes.

Chapter 10. Text-Processing Fuctions for Thesaurus

Synonym Aid

Compiles a list of possible synonyms for the input word. Only dictionaries containing synonyms are searched, and only information from the first dictionary containing a match is presented.

If the dictionary has definitions, or if a companion definition dictionary is available, (UK.DEF for U.K. English, US.DEF for U.S. English and AUS.DEF for Australian English), definitions will be returned for each synonym group, if requested. If the *lx_syn_inflect_f* flag is set to LX_TRUE and the input word is found in the corresponding system dictionary, inflected synonyms will be returned.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_SYNONYM_AID
- *lx_dict_tkns_ct* — number of dictionary handles in input dictionary list
- *lx_dict_tkns* — input dictionary list
- *lx_num_cpg* — number of code page that data is in
- *lx_rqst_type* — must be set to LX_NEW_REQ; synonym aid does not support continue reply
- *lx_elements_ct* — number of elements in the data element list; must be one
- *lx_elements_p* — pointer to input data element list containing the word for which synonyms are being requested
- *lx_elt_format* — must be LX_SINGLE_WORD_FMT
- *lx_reply_p* — pointer to reply area (allocated by caller)
- *lx_reply_size* — size in bytes of reply area
- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_sent_begin_f* — set to zero
- *lx_syn_pos* — requested part of speech:
  - LX_NOUN
  - LX_VERB
  - LX_ADJECTIVE
  - LX_ADVERB
  - LX_PREPOSITION
  - LX_INTERJECTION
  - LX_CONJUNCTION
  - LX_PRONOUN
  - LX_MASCULINE_NOUN

© Copyright IBM Corp. 2000

- LX_FEMININE_NOUN
- LX_NEUTER_NOUN
- LX_PLURAL_NOUN
- LX_VERB_PRONOMINAL
- LX_TRANSITIVE_VERB
- LX_INTRANSITIVE_VERB
- LX_ALL_POS — all parts of speech

- *lx_syn_outtype* — Code specifying the type of output desired:
  - 0 — (default) Output synonyms. Also, output definitions if available, but do not output definitions that have no synonyms.
  - 1 — Output definitions only.
  - 2 — Output synonyms only.
  - 3 — Output definitions and synonyms. Not all definitions will be associated with synonyms. This option is a combination of option 1 and 2.
- *lx_syn_inflect_f*
  - LX_FALSE — to request return of synonyms as stored in dictionary. This will usually, but not always, be the base form of the synonym word.
  - LX_TRUE — to request inflection of synonyms to match the morphology (form) of the input word.

    This option has no effect if the word is found in an addenda dictionary.

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated:

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_reply_used*
- *lx_dict_found_in*
- *lx_delivered_out_units* — number of fields returned in the reply area
- *lx_reply_p* — pointer to variable-length data returned in the Reply Area as a number of fields, each of which has the following format:
  - field type (LX_UCHAR)
    - 0x01 — dictionary headword
    - 0x02 — part of speech
    - 0x03 — change-sense
    - 0x04 — end-of-scope
    - 0x05 — concatenate next qualifier to previous qualifier
    - 0x06 — qualifier
    - 0x07 — synonym
  - field length (LX_USHORT) — length of following value
  - field value (a block of LX_UCHAR) — the interpretation of this field depends on the value of field type, as described below.

Fields are output in the same order as they appear in the dictionary. The change-sense field appears immediately before any synonym field which begins a new synonym sense group. A sense-level qualifier, likewise, is output before the synonym group(s) it qualifies. A word-level qualifier is output after the synonym it qualifies. The qualifier concatenation field indicates that the next qualifier should be concatenated to the previous qualifier. The end-of-scope field ends the current nesting level. See “Code Examples” on page 161 for details. The field types and their allowed values are documented below:

Type	*Value*
headword	a string representing the dictionary word which matched the input word
part of speech	a single byte with the same values as lx_syn_pos (see input section)
change-sense	(empty)
end-of-scope	(empty)
qualifier concatenation	(empty)
qualifier	subtype + qualifier data (see below)
synonym	a string representing the synonym word or phrase

The field value of the qualifier field contains:

- subtype (LX_UCHAR) — indicates type of qualifier:
  - 0x01 — sense-level qualifier
  - 0x02 — word-level qualifier
- qualifier data (a string of LX_UCHAR) — formatted as a tag, followed by a blank, followed by information about the synonym or group

If the input word is found in an addenda dictionary, there will be one synonym field (type 07) in the reply area for each synonym stored with the word in the addenda dictionary. Additionally, if the word in the addenda dictionary is accompanied by a morphological alias, this alias will be used to determine the base forms of the addenda word. Any synonyms stored in the addenda for these base forms will be returned as well. (See “Adding Morphological Information to Addenda Dictionaries” on page 23 for details on morphological information in an addenda dictionary.) Automatic synonym inflection is not supported for synonyms found in an addenda dictionary.

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

- LX_BAD_FUNCT_CODE — invalid function code
- LX_BAD_RQST_TYPE — invalid request type
- LX_RC_OK — processing successfully completed
- LX_UNEXPECTED_RC — unexpected or unknown condition
- LX_BAD_SERV_HANDLE — invalid Service Area Handle
- LX_BAD_DICT_TKN — invalid dictionary token
- LX_BAD_ELE_FORMAT — invalid data element format

- LX_BAD_ELEMENTS_CT — invalid element count
- LX_BAD_PARM — missing or illegal input value for parameter
- LX_BAD_WORD — invalid word
- LX_CPG_NOT_SUPPORTED — mode page not supported
- LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
- LX_IO_ERROR — file input/output error
- LX_MEM_ALLOC_ERR — memory allocation error
- LX_NO_AID_AVAIL — no aid available
- LX_NO_DATA_AVAIL — no data available
- LX_REPLY_FULL — reply area is full
- LX_SYN_OK_NO_MID — morphology error in synonym aid; output not inflected
- LX_WORD_NOT_FOUND — word not found in dictionary

Application Programming Guidelines

Before performing actual synonym lookup, this function calls **Morphological Identification (MID)** to determine the lemma (base form) of the input word. This is necessary because synonym data are stored under the lemma rather than under the inflected forms.

On rare occasions this call to MID may fail. This will not cause a hard error which will terminate the **Synonym Aid** function. Rather, Linguistic Tools will treat the input word as if it were a lemma and will look for synonym data for it. If, in fact, the input word is a lemma form, the desired synonyms may be returned. If the input word is only an inflected form, no synonym data will be found—because MID failed, its lemma cannot be determined—and a return code of LX_NO_AID_AVAIL will be returned.

Sometimes, however, an input word may be ambiguous—that is, it could be either a lemma form of one word or an inflected form of another. For example, “lay” could be either a lemma form *lay* (“Lay the cloth on the table.“) or the past tense of *lie* (”He lay down on the couch and fell asleep.“). In such a case, a failure of MID would result in only the first form being looked up. The synonym data returned would be valid (for the first form) but incomplete (because the second lemma could not be identified and looked up).

In any case, if MID fails, any synonyms returned will be in lemma form, even if the inflected synonyms option was selected.

The application should regard a return code of LX_SYN_OK_NO_MID as a warning that the results of this function call might not be correct; it should warn the user that the returned data may be incomplete.

Definitions currently are available only for U.K., U.S. and Australian English. In order to return definitions for groups of synonyms (definitions are given for sense groups, not for individual synonyms), the companion definition dictionary (UK.DEF, US.DEF or AUS.DEF) must be present. On workstations, the companion dictionary must be located in the same directory as the main dictionary. On VM, the companion dictionary must be located on the same minidisk as the base dictionary.

Inflected synonyms will only be returned if *lx_syn_inflect_f* is set to LX_TRUE ***and*** the user has the companion definition dictionary in the same location as the base dictionary. Although, the synonyms may be inflected to match the input word through the use of the *lx_syn_inflect_f* flag, this feature is only available for U.K.,

U.S. and Australian English and is slow when many synonyms are to be inflected. Therefore, it is strongly recommended that the **Inflect Word From Model** function be called to obtain an inflected form of a single synonym word selected by the user, which works with any language for which morphological information is available. Note that **Inflect Word From Model** can only inflect a single-word synonym, not a multiword phrase. The application should pass as input to **Inflect Word from Model** only the word that was input to **Synonym Aid** (to serve as an inflectional model) and the synonym selected by the user.

Code Examples

```
/*============================*/
/* NLP synonym call samples   */
/*============================*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "efzlnlps.h"      /* Contains NLP constants and typedefs */
#include "nlpsamp.h"

/* Strings used to print out      */
/* US English part-of-speech info */
LX_PUCHAR SynPos [] =
  {
  "Pronoun",
  "Verb",
  "Noun",
  "Adjective",
  "Adverb",
  "Preposition",
  "Interjection",
  "Conjunction",
  "Noun-Masculine",
  "Noun-Feminine",
  "Verb-Pronominal",
  "Verb-Transitive",
  "Verb-Intransitive",
  "Noun-Plural",
  "Noun-Neuter"
  };

/*==================================================================*/
/* This function makes a sample call to the Synonym Aid function.  */
/* It assumes that the following fields in the NLP control block   */
/* have already been set:                                          */
/*              - lx_dict_tkns                                     */
/*              - lx_dict_tkns_ct                                  */
/*              - lx_num_cpg                                       */
/*              - lx_serv_area_p                                   */
/*==================================================================*/
void Synonym(LX_CB_P   pNLPCB,
             LX_PUCHAR pWord)
{
  LX_ELEMENT  DataEle;    /* Data Element structure for input to Poe */
  LX_PUCHAR   pReplyArea; /* Dynamically allocated reply area.       */
  LX_ULONG    xx, yy;     /* loop counters */
```

```
LX_PUCHAR   pCurrChar;  /* Used to traverse output data in reply area */
LX_UCHAR    FieldType;
LX_PUSHORT  pFieldLen;
LX_USHORT   FieldLen;   /* *pFieldLen without the indirection */
LX_PUCHAR   pFieldData;
LX_UINT     SenseNum;
LX_UCHAR    HeadWord[LX_MAX_WORD_LEN];
LX_BOOL     fConcatNext;
LX_BOOL     fQualPrint;       /* Did we print a qualifier ? */

/*---------------------------------------------------------*/
/* 2K should be enough to hold synonym output for most words. */
/* NOTE:                                                   */
/* A regular application would probably not allocate a new    */
/* area every time a word is passed.  Instead re-use another  */
/* area.                                                   */
/*---------------------------------------------------------*/
pReplyArea = malloc(2048);

/* Create data element structure based on input string */
DataEle.lx_data_p   = pWord;
DataEle.lx_data_len = strlen(pWord);
DataEle.lx_input_f  = '\0';
DataEle.lx_trail    = '\0';

/* Set up parms in NLP control block */
pNLPCB->lx_rqst_type   = LX_NEW_REQ;
pNLPCB->lx_func_code   = LX_SYNONYM_AID;
pNLPCB->lx_num_cpg     = 850;
pNLPCB->lx_elements_ct = 1;
pNLPCB->lx_elements_p  = &DataEle;
pNLPCB->lx_elt_format  = LX_SINGLE_WORD_FMT;
/* Single word input format */
pNLPCB->lx_reply_p     = pReplyArea;
pNLPCB->lx_reply_size  = 2048;
pNLPCB->lx_syn_pos     = LX_ALL_POS;      /* All parts of speech */
pNLPCB->lx_syn_outtype = 0;               /* Synonyms and definitions */

NlpEntry(pNLPCB);

printf("Synonym rc = %d\n", pNLPCB->lx_rc);

/* Print reply area information */
if (pNLPCB->lx_delivered_out_units)
  {
  pCurrChar = (LX_PUCHAR) pNLPCB->lx_reply_p;

  fConcatNext = LX_FALSE;
  fQualPrint  = LX_FALSE;
  for (xx = 0; xx < pNLPCB->lx_delivered_out_units; xx++)
    {
    FieldType = *pCurrChar;
    pFieldLen = (LX_PUSHORT) (pCurrChar + 1);
    pFieldData = (LX_PUCHAR) (pFieldLen + 1);

    FieldLen = *pFieldLen;

    switch (FieldType)
```

```
{
case 1:    /* Dictionary Headword */
  strncpy(HeadWord, pFieldData, FieldLen);
  HeadWord[FieldLen] = '\0';  /* NULL-delimit string */
  break;

case 2:    /* Part Of Speech */
  printf("\n\nSynonyms of \"%s\" (%s)",
         HeadWord,
         SynPos[*pFieldData]);
  SenseNum = 0;
  break;

case 3:    /* Change-Sense */
  if (fQualPrint == LX_FALSE)
    printf("\n\t\t--------");
  fQualPrint = LX_FALSE;
  break;

case 5:    /* Qualifier Concatenation */
  fConcatNext = LX_TRUE;
  break;

case 6:    /* Qualifier */
  fQualPrint = LX_TRUE;

  /* if it is a Sense-level qualifier */
  if (*pFieldData == '\x01')
    {
    /*---------------------------------------------------*/
    /* A context qualifier tag resets sense-number & causes */
    /* its data to be enclosed in angle-brackets.          */
    /*---------------------------------------------------*/
    if (pFieldData[1] == 'C')
      {
      SenseNum = 0;
      printf("\n\n\t<");
      }
    else
      {
      /*-------------------------------------------------*/
      /* If a type '5' field was encountered previously,    */
      /* don't start a new definition line.                */
      /*-------------------------------------------------*/
      if (fConcatNext != LX_TRUE)
        {
        ++SenseNum;
        printf("\n\n\t %d. ", SenseNum);
        }
      else
        printf("  ");

      if (pFieldData[1] == 'U')
        {
        printf("(");
        }
      }
```

```
          /* Output the data following the tag */
          for (yy = 3; yy < FieldLen; yy++)
            printf("%c", pFieldData[yy]);

          /*----------------------------------------------------*/
          /* Output appropriate closing punctuation based on tag  */
          /*----------------------------------------------------*/

          /* if tag indicates a "usage" qualifier */
          if (pFieldData[1] == 'U')
            {
            printf(")");
            }

          /* if tag indicates a "context" qualifier */
          else if (pFieldData[1] == 'C')
            printf(">");

          fConcatNext = LX_FALSE;
          }

        /* otherwise it is a Word-level Qualifier */
        else
          {
          printf("  (");

          /* Output the data following the tag */
          for (yy = 3; yy < FieldLen; yy++)
            printf("%c", pFieldData[yy]);

          printf(")");
          }

        break;

      case 7:    /* Synonym */
         printf("\n\t\t");
         for (yy = 0; yy < FieldLen; yy++)
           printf("%c", pFieldData[yy]);
      }

    /* Advance pointer to the next field in the reply area. */
    pCurrChar = pFieldData + FieldLen;
    }

  printf("\n\n");
  }

  free(pReplyArea);
}
```

Sample Output

```
----------------- Sample output of code for word "test" ----------------

Synonyms of "test" (Noun)

 1. A set of questions or exercises designed to determine knowledge or skill
           catechism
           catechization
           exam  (Informal)
           examination
           quiz

 2. An operation employed to resolve an uncertainty
           experiment
           experimentation
           trial

 3. A procedure that ascertains effectiveness, value, proper function,
       or other quality
           assay
           essay
           proof
           trial
           tryout  (Informal)

 4. A means by which individuals are compared and judged
           standard
           benchmark
           criterion
           gauge
           mark
           measure
           touchstone
           yardstick

Synonyms of "test" (Verb)

 1. To subject to a test of knowledge or skill
           check
           examine

 2. To subject to a procedure that ascertains effectiveness, value,
       proper function, or other quality
           assay
           check
           essay
           examine
           prove
           try
           try out

 3. To engage in experiments
           experiment
           try out

Synonyms of "test" (Adjective)

 1. Constituting a tentative model for future experiment or development
```

```
pilot
experimental
trial
```

Chapter 11. Text-Processing Functions for Morphology

Morphological Identification

Returns morphological information for an input word. This information may consist of a lemma (base form), part of speech, PCODE, DBPC (double-byte parse code), inflective classification (morphology bit mask representing information about person, number, tense, and so on), and/or a paradigm number. If the input word has multiple lemmas, all lemmas will be returned.

If the input word is not included in the system dictionary, but an addenda dictionary containing alias information for the input word is available, this function can apply the morphological information from the alias to the input word.

Output from Morphological Identification can be used as input to Generate Inflected Forms.

For DBCS languages, this function will accept mixed SBCS/DBCS input. (Currently Chinese and Korean are the only DBCS languages supported for Morphological Identification.) Also for DBCS languages, the DBPC value will be returned in the reply area. For SBCS languages, the DBPC field will be present but will contain zeros. See “Chinese Language-Specific Processing” on page 284 and “Korean Language-Specific Processing” on page 328 for details about Chinese and Korean language differences and an explanation of the language-specific DBPC values.

Alias processing and morphology bit masks are not supported for DBCS languages.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_rqst_type*
- *lx_func_code* — set to LX_MORPH_ID
- *lx_dict_tkns_ct*
- *lx_dict_tkns*
- *lx_num_cpg* — number of code page that data is in
- *lx_elements_ct*
- *lx_elt_format* — must be LX_SINGLE_WORD_FMT
- *lx_elements_p* — pointer to data element list containing the input word
- *lx_alias_len* — length of morphological alias from addenda dictionary
- *lx_alias_p* — pointer to alias term
- *lx_reply_p*
- *lx_reply_size*
- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_morph_pcode_f* — always LX_TRUE
- *lx_sent_begin_f* — set to zero

© Copyright IBM Corp. 2000

Output

The updated Linguistic Tools control block. The following output fields or the areas they point to are updated:

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_reply_used*
- *lx_dict_found_in*
- *lx_reply_p* — Variable length output data is returned in the Reply Area, as follows:
  - Length of input word (1 LX_UCHAR)
  - Input word
  - Number of paradigms returned (1 LX_UCHAR)
  - For each paradigm the following is returned:
    - PCODE (7 LX_UCHARs)
    - Part of speech (1 LX_UCHAR). Possible values for this field are:
      - LX_PRONOUN
      - LX_VERB
      - LX_NOUN
      - LX_ADJECTIVE
      - LX_ADVERB
      - LX_PREPOSITION
      - LX_INTERJECTION
      - LX_CONJUNCTION
      - LX_POS_UNKNOWN
    - DBPC (1 LX_USHORT) — For SBCS words this will always be zero.
    - Paradigm number (1 LX_USHORT)
    - Number of lemmas (1 LX_UCHAR)
    - For each lemma:
      - Length of the lemma (1 LX_UCHAR)
      - The lemma
    - Number of morphology bit masks for this paradigm (1 LX_UCHAR)
    - for each morphology bit mask:
      - The morphology bit mask (4 LX_UCHARs). See Appendix E, "Morphology Grammar Masks" on page 347 for the format of the morphology bit masks. These masks differ from language to language.

Sample Output

Following is the reply area output from MID when all of the information for the word "wound" is requested:

```
x'05'       length of the following word
wound       input word (in code page specified by calling application)
x'04'       number of paradigms

x'C5D50000000000'  seven LX_UCHARs of PCODE
x'02'              part of speech (noun)
x'0000'            DBPC
x'0200'            paradigm number (2)
```

```
x'01'             number of following lemmas
x'05'             length of the following lemma
wound             lemma for this paradigm
x'01'             number of morphology masks that follow
x'00800000'       singular

x'C5E50000000000' seven LX_UCHARs of PCODE
x'01'             part of speech (verb)
x'0000'           DBPC
x'0300'           paradigm number (3)
x'01'             number of following lemmas
x'05'             length of the following lemma
wound             lemma for this paradigm
x'06'             number of morphology masks that follow
x'80000000'       infinitive
x'44800000'       present tense, first person, singular
x'42800000'       present tense, second person, singular
x'44400000'       present tense, first person, plural
x'42400000'       present tense, second person, plural
x'41400000'       present tense, third person, plural

x'C5C9D100000000' seven LX_UCHARs of PCODE
x'03'             part of speech (adjective)
x'0000'           DBPC
x'0500'           paradigm number (5)
x'01'             number of following lemmas
x'05'             length of the following lemma
wound             lemma for this paradigm
x'01'             number of morphology masks that follow
x'00000000'       no bits set

x'C5E50000000000' seven LX_UCHARs of PCODE
x'01'             part of speech (verb)
x'0000'           DBPC
x'2800'           paradigm number (40)
x'01'             number of following lemmas
x'04'             length of the following lemma
wind              lemma for this paradigm
x'07'             number of morphology masks that follow
x'24800000'       past tense, first person, singular
x'22800000'       past tense, second person, singular
x'21800000'       past tense, third person, singular
x'24400000'       past tense, first person, plural
x'22400000'       past tense, second person, plural
x'21400000'       past tense, third person, plural
x'08000000'       past participle
```

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle

LX_BAD_ADDRESS — invalid address (pointer argument)
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_PARM — missing or illegal input value for parameter
LX_BAD_WORD — invalid word
LX_BLOCK_SPAN — unexpected block spanning attempt
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ELE_P — missing element list
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_IO_ERROR — file input/output error
LX_MEM_ALLOC_ERR — memory allocation error
LX_NO_AID_AVAIL — no aid available
LX_NO_BIT_MASK_TABLE — no bit mask table in the dictionary
LX_NO_BIT_MASK_ENTRY — no bit mask entry for this PCODE
LX_NO_DATA_AVAIL — no data available
LX_NO_DATA_SAT — no data satisfies input parameter request
LX_NO_REPLY_P — missing reply area
LX_RC_OK_NHS — processing successfully completed; non-Hangeul stem returned by Morphological Identification for Korean
LX_REPLY_FULL — reply area is full
LX_WORD_NOT_FOUND — word not found in dictionary

Application Programming Guidelines

See “Use of Single-Word Format Data Elements” on page 71 for a description of how input words are to be passed to this function via a single-word format data element list.

MID will use the language of the first dictionary in the input dictionary list to determine whether Korean-specific processing should be performed. Note that for Korean, lx_dict_found_in is not set, as Korean MID may return multiple lemmas from multiple dictionaries. See “Korean Language-Specific Processing” on page 328 for details of Korean MID.

When both an addenda dictionary and a system dictionary are passed to this function and the input word is present with its accompanying alias term in the addenda dictionary, the morphological information for the alias term is retrieved from the system dictionary and applied to the input word. If the alias term is not present in the addenda dictionary, a return code of LX_NO_AID_AVAIL will be returned. If only an addenda dictionary is passed to this function, a return code of LX_NO_DATA_AVAIL will be returned.

See “Morphological Identification” on page 30 for an explanation of paradigms, PCODEs, DBPCs, and morphology bitmasks.

Generate Inflected Forms

Returns a specific inflected form or all inflected forms of the input word, as requested through input fields specifying paradigm number, PCODE, part of speech, form ID, and/or inflective classification (bit mask representing information about person, number, tense, and so on). If a field is set to zero, it will not be used to limit the information returned.

If a word is not included in the system dictionary, but morphological alias information for it is available in an active addenda dictionary, this information will be used, in conjunction with the morphological information given in the system dictionary for the alias term, to generate inflected forms.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle return from the **Initialize** function
- *lx_rqst_type*
- *lx_func_code* — set to LX_GEN_WORD_INFLECT
- *lx_dict_tkns_ct*
- *lx_dict_tkns*
- *lx_num_cpg* — number of code page that data is in
- *lx_elements_ct*
- *lx_elt_format* — must be LX_SINGLE_WORD_FMT
- *lx_elements_p* — pointer to data element list containing input word
- *lx_alias_len* — length of the alias word (for using morphology alias information from an addenda dictionary)
- *lx_alias_p* — pointer to the alias word
- *lx_reply_p*
- *lx_reply_size*
- *lx_morph_pcode* — identifies PCODE desired (***NOTE:*** This field must be set to LX_FALSE if not used.)
- *lx_morph_mask_inp* — pointer to an array of masks that identifies the inflections desired

  This pointer points to user-allocated memory in the form of an N by 4 array. N is the number of masks sent as input and there is no limit on this number. The last mask must be equal to all null-characters. In other words, the user sets up a block of memory that contains an array of masks (each mask is 4 characters) with the last mask being "\0\0\0\0". Then the user sets lx_morph_mask_inp to point to this memory. (***NOTE:*** This field must be set to all null-characters if not used.) be output)
- *lx_morph_form_id* — requested form id (***NOTE:*** This field must be set to LX_FALSE if not used.)

- *lx_morph_paradigm* — requested paradigm number (***NOTE:*** This field must be set to LX_FALSE if not used.)
- *lx_sent_begin_f* — set to zero

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_reply_used*
- *lx_dict_found_in*
- *lx_reply_p* — Variable length output data is returned in the Reply Area as follows:
  - length of input word (1 LX_UCHAR)
  - input word
  - number of paradigms returned (1 LX_UCHAR)
  - for each paradigm the following is returned:
    - PCODE (7 LX_UCHARs)
    - part of speech (1 LX_UCHAR). The values for this field are:
      - LX_PRONOUN
      - LX_VERB
      - LX_NOUN
      - LX_ADJECTIVE
      - LX_ADVERB
      - LX_PREPOSITION
      - LX_INTERJECTION
      - LX_CONJUNCTION
      - LX_POS_UNKNOWN
    - DBPC (1 LX_USHORT) — reserved
    - paradigm number (LX_USHORT)
    - number of forms returned (one or more) (1 LX_UCHAR)
    - for each form returned:
      - form id (LX_USHORT)
      - number of spellings (one or more) (1 LX_UCHAR)
      - for each spelling returned:
        - length of the word (1 LX_UCHAR)
        - the word
      - number of morphology masks for the word (1 LX_UCHAR)
      - for each morphology mask returned:
        - the morphology mask (4 LX_UCHARs)

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_ADDRESS — invalid address (pointer argument)
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_FLAG — invalid flag value
LX_BAD_PARM — missing or illegal input value for parameter
LX_BAD_SERV_HANDLE — invalid Service Area Handle input
LX_BAD_WORD — invalid word
LX_BLOCK_SPAN — unexpected block spanning attempt
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ELE_P — missing element list
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_IO_ERROR — file input/output error
LX_MEM_ALLOC_ERR — memory allocation error
LX_NO_AID_AVAIL — no aid available
LX_NO_BIT_MASK_TABLE — no bit mask table in the dictionary
LX_NO_BIT_MASK_ENTRY — no bit mask entry for this PCODE
LX_NO_DATA_AVAIL — no data available
LX_NO_DATA_SAT — no data satifies input parameter request
LX_NO_REPLY_P — missing reply area
LX_REPLY_FULL — reply area is full
LX_WORD_NOT_FOUND — word not found in dictionary

Application Programming Guidelines

See “Use of Single-Word Format Data Elements” on page 71 for a description of how input words are to be passed to this function via a single-word format data element list.

When both an addenda dictionary and a system dictionary are passed to this function and the input word is present with its accompanying alias term in the addenda dictionary, the morphological information for the alias term is retrieved from the system dictionary and applied to the input word. If the alias term is not present in the addenda dictionary, a return code of LX_NO_AID_AVAIL will be returned. If only an addenda dictionary is passed to this function, a return code of LX_NO_DATA_AVAIL will be returned.

Inflect Word from Model

Returns the inflected form or forms of a word based on a model. May be used in conjunction with the **Synonym Aid** function in order to provide synonyms in the same inflected form as the input word.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_INFLECT_WORD_FROM_MODEL
- *lx_dict_tkns_ct* — number of dictionary handles in the input dictionary list
- *lx_dict_tkns* — input dictionary list
- *lx_num_cpg* — number of code page that data is in
- *lx_rqst_type*
- *lx_elements_ct* — must be 2

  The first data element should contain the word to use as a model, while the second should contain the word to be inflected.
- *lx_elements_p* — pointer to the data element list.
- *lx_elt_format* — must be LX_SINGLE_WORD_FMT
- *lx_reply_p* — pointer to reply area (allocated by caller)
- *lx_reply_size* — size in bytes of reply area

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_reply_used*
- *lx_dict_found_in*
- *lx_reply_p* — variable-length output data is returned in the Linguistic Service reply area as follows:
  - Length of model word (1 LX_UCHAR)
  - Model word
  - Length of word to inflect (1 LX_UCHAR)
  - Word to inflect
  - Number of inflected forms (1 LX_UCHAR)
  - For each inflected form:
    - Length of the form (1 LX_UCHAR)
    - The form itself

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_PARM — missing or illegal input value for parameter
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_IO_ERROR — file input/output error
LX_MEM_ALLOC_ERR — memory allocation error
LX_NO_AID_AVAIL — no aid available
LX_NO_REPLY_P — missing reply area
LX_REPLY_FULL — reply area is full
LX_WORD_NOT_FOUND — word not found in dictionary

Application Programming Guidelines

This function is designed to be used in conjunction with the **Synonym Aid** function. **Synonym Aid** returns synonyms in their base (lemma) form. If an inflected form is desired—for example, to replace an inflected form in the input text — the application should call **Inflect Word from Model** to obtain the appropriate inflected form. In this case, the first data element should contain the word that was input to the **Synonym Aid** function, while the second contains the user-selected synonym to be inflected. This function can only be used for single-word synonyms. Although **Synonym Aid** may return phrases as synonyms, this function cannot inflect a phrase properly.

Normally the word that was input to the **Synonym Aid** function would serve as the model, since the goal is to replace it with a correctly inflected synonym. However, it is possible to call this function to inflect any word using any other word as a model.

This function requires that the dictionary contain both the model word and the word to be inflected along with their associated morphological information.

In some cases it is possible for more than one inflected form to be returned from this function. This occurs when the model word can represent more than one inflected form. For example, the English word “set” can be either the present (non-third person) or the past form of the verb, as well as a singular noun. Suppose that the user were to select “place” as a synonym for the verb “set”. The application would call the **Inflect Word from Model** function with “set” as the model and “place” as the word to be inflected. Because the verb form “set” can be either present or past, the function would return two inflected forms for the verb “place”: “place” and “placed”. Because both “set” and “place” can also be nouns, a noun

form would have been identified as well, but since this has the same spelling as one of the verb forms, it is not listed separately in the reply area.

Chapter 12. Text-Processing Functions for Text Analysis

Simple Tokenization

Takes input text in the form of a block format data element list, groups characters into simple tokens (words), and outputs these in the form of a token list.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_rqst_type*
- *lx_func_code* — set to LX_ISOLATE_WORD
- *lx_num_cpg* — number of code page that data is in
- *lx_lang_code* — required for DBCS processing
- *lx_elements_ct*
- *lx_elements_p* — pointer to input data element list
- *lx_elt_format* — must be LX_BLOCK_FORMAT
- *lx_out_units* — maximum number of tokens that the application wishes to obtain on a single call to this function; set to zero for all
- *lx_reply_p*
- *lx_reply_size*

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc*
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_cont_reply_f*
- *lx_delivered_out_units*
- *lx_reply_p* — The Reply Area will contain a token list in which each token represents essentially a single word and its attributes.
- *lx_hFirstToken*

© Copyright IBM Corp. 2000

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_ELE_TYPE — invalid element type
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_STR_ADDR — invalid or NULL string address provided
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_FUNC_CODE_CHG — change of function code not allowed
LX_REPLY_FULL — reply area is full

Application Programming Guidelines

For single-byte (SBCS) input, each token corresponds to a word, as defined in “Definition of a Word in SBCS Text” on page 80. Because Korean uses blanks (white space) as word delimiters, this function also returns a token for each word in Korean text. In Chinese and Japanese mixed DBCS/SBCS input, Linguistic Tools will separate the SBCS text from the DBCS text and analyze the SBCS portion into word tokens as defined in “Definition of a Word in SBCS Text” on page 80. Contiguous DBCS text generally will be grouped together in a single token. Only DBCS punctuation is put into separate tokens. **Lexical Analysis** does language-specific processing for Chinese and Japanese.

Note that a simple token is not always exactly the same thing as a word. For example, the hyphenated word *mother-in-law* is divided into three simple tokens: *mother in law*. Similarly, forms such as *PS/2* are broken at the slash. Setting *lx_out_units* to the maximum number of tokens that the application wishes to obtain at one time will force Linguistic Tools to return only that number of tokens, regardless of the amount of text that has been processed. Issue a continue reply to obtain the next group of tokens.

Text Segment Identification

Identifies sentences and sentence fragments. Marks boundaries by placing new-sentence tokens in the output token list. See "Text Segment Identification" on page 33 for additional information.

Language-specific processing may be performed. See Appendix D, "Language-Specific Processing" on page 281 for details.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_rqst_type*
- *lx_func_code* — set to LX_ISOLATE_SEGMENT
- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the Initialize function
- *lx_num_cpg* — number of code page that data is in
- *lx_elt_format* — must be LX_BLOCK_FORMAT
- *lx_elements_ct*
- *lx_elements_p* — pointer to input data element list
- *lx_lang_code* — required
- *lx_out_units* — maximum number of sentences requested; set to 0 for all
- *lx_reply_p*
- *lx_reply_size*
- *lx_dict_tkns_ct* — number of dictionaries in dictionary list
- *lx_dict_tkns* — dictionary list
- *lx_abbr_tkns_ct* — number of dictionaries in Abbreviation Addenda Dictionary list
- *lx_abbr_tkns* — Abbreviation Addenda Dictionary list

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_delivered_out_units*
- *lx_reply_p* — The reply area will contain a token list, with a new-sentence token (type LX_NSEN) inserted before each text segment. This output may be accessed using the Linguistic Tools token list utility functions.
- *lx_hFirstToken*

- *lx_next_elemt_p*
- *lx_next_char_p*
- *lx_cont_reply_f*

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_ELE_TYPE — invalid element type
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_LANG_CODE — invalid language code
LX_BAD_TOKEN — invalid pointer to a token
LX_CPG_NOT_SUPPORTED — code page not supported
LX_END_OF_INPUT — end of input reached
LX_END_OF_STORAGE — end of system storage reached
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_INVALID_DBCS_INPUT — an incorrect sequence of SO/SI characters was encountered.
LX_INVALID_SBCS_INPUT — code points which do not exist in the code page were encountered.
LX_OUT_GOAL — output goal reached before end of input
LX_REPLY_2_SMALL — reply area is too small to allow any output
LX_REPLY_FULL — reply area is full; output may be continued

Application Programming Guidelines

The language code field is required for this function. For Linguistic Tools to handle DBCS terminal punctuation correctly, the correct language code ***must*** be specified. For SBCS text, the language code is used to access the appropriate abbreviation list. A value of zero is valid if no language-specific processing is desired.

Setting *lx_out_units* to the maximum number of sentences that the application wishes to obtain at one time will force Linguistic Tools to return only that number of sentences, regardless of the amount of text has been processed. Request a continue-reply to obtain the next group of sentences.

The following fields may be changed by the application prior to requesting a continue-reply:

- the number of output units (lx_out_units)
- the reply area size (lx_reply_size)
- the reply area pointer (lx_reply_p)

Lexical Analysis

This function analyzes and, if necessary, changes the structure of simple tokens containing internal punctuation such as apostrophes&mdashfor example, English contractions (he'll), French prefixed articles (l'enfant). See Appendix D, "Language-Specific Processing" on page 281 for details of processing in various languages.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_rqst_type*
- *lx_func_code* — set to LX_LEXICAL_ANALYSIS
- *lx_num_cpg* — number of code page that data is in
- *lx_elt_format* — must be LX_BLOCK_FORMAT
- *lx_elements_ct*
- *lx_elements_p*
- *lx_lang_code* — required
- *lx_out_units* — maxmimum number of tokens that the application wishes to have returned on one function call; set to zero for all
- *lx_reply_p*
- *lx_reply_size*
- *lx_dict_tkns_ct* — number of dictionaries in dictionary list
- *lx_dict_tkns* — dictionary list
- *lx_abbr_tkns_ct* — number of dictionaries in Abbreviation Addenda Dictionary list
- *lx_abbr_tkns* — Abbreviation Addenda Dictionary list

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_delivered_out_units*
- *lx_reply_p*
- *lx_hFirstToken*
- *lx_next_elemt_p*
- *lx_next_char_p*
- *lx_cont_reply_f*

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_ELE_TYPE — invalid element type
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_FIRST — invalid first token; lx_first_tkn_p must point to a token
LX_BAD_LANG_CODE — invalid language code
LX_BAD_LAST — invalid last token; lx_next_tkn_p must point to a token
LX_BAD_TOKEN — invalid token discovered in token list
LX_CPG_NOT_SUPPORTED — code page not supported
LX_END_OF_INPUT — end of input reached; processing successfully completed

Note: This function returns LX_END_OF_INPUT instead of LX_RC_OK.
LX_END_OF_STORAGE — end of system storage reached
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_FUNC_CODE_CHG — change of function code not allowed
LX_NO_AID_AVAIL — system dictionary not available for Japanese processing
LX_NO_CPX — no lexical analysis for this language; only default processing was performed

Note: This return code is returned only if Lexical Analysis completed the default processing successfully; otherwise, Lexical Analysis returns the appropriate error condition.
LX_OUT_GOAL — output goal reached before end of input
LX_REPLY_2_SMALL — reply area is too small to allow any output
LX_REPLY_FULL — reply area is full

Application Programming Guidelines

The language code is required for this function. For Chinese and Japanese processing, the correct language code ***must*** be specified. A value of zero is valid if no language-specific processing is desired. For Japanese only, the system dictionary ***must*** be active for Lexical Analysis. If the system dictionary is not found, this function will return LX_NO_AID_AVAIL.

Setting *lx_out_units* to the maximum number of tokens that the application wishes to obtain at one time will force Linguistic Tools to return only that number of tokens, regardless of the amount of text that the function has processed. Issue a continue reply to obtain the next group of tokens.

The following control block fields may be changed on a continue reply:

- the number of output units (lx_out_units)
- the reply area size and pointer (lx_reply_size and lx_reply_p).

Alternate Terms (LX_SUBTERM Property)

When processing Chinese text, the Lexical Analysis function creates tokens that correspond to words. Although multiple combinations of words may be determined from the same input text, a single parse is represented in the main token list. The combination of words that spans the entire sentence with the fewest word breaks is output in the token list. This tends to cause the Lexical Analysis function to output longer terms.

Because many smaller terms are not represented in the token list produced by Lexical Analysis, the LX_SUBTERM property is added to a token whenever the word corresponding to that token may contain a separate word. For example, if the term "software" is represented by a token, it could have two LX_SUBTERM properties associated with it. Because the words "soft" and "ware" are also valid words, they would each be indicated as terms by individual LX_SUBTERM properties attached to the token that corresponds to the word "software."

The LX_SUBTERM property has a value that indicates both the offset where the word begins and the length of the word. The offset where the word begins is in reference to the current token; an offset of zero is the first byte of that token. The length of the word is the length in bytes of the word that starts at the specified offset. Both of these items are stored in the "two_vals" union of the LX_PVAL data structure. The offset is stored in the *ushort1* field of the union, and the length is stored in the *ushort2* field.

It should be noted that all substrings that are separate terms are not represented via LX_SUBTERM property. Only words that combine with others to span an entire sentence are considered valid terms. For example, the word "develop" is present in the term "development." Since the string "ment" is not a known word, the word "develop" would not be marked with the LX_SUBTERM property.

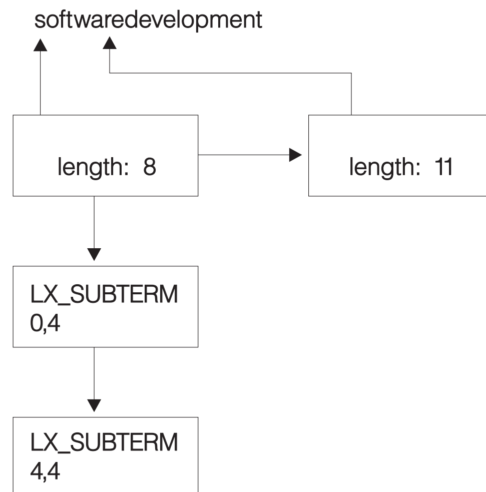


Figure 10. LX_SUBTERM example for "softwaredevelopment"

To provide an example of sample output, English text with no spaces will be shown as a substitute for Chinese characters. Figure 10 shows the sample output for input of "softwaredevelopment". Two tokens are output that correspond to "software" and "development". The first token has two LX_SUBTERM properties attached to it. The first property has an offset of zero and a length of four; it represents the term "soft". The second property has an offset of four and a length of four; it represents the term "ware".

Figure 11 on page 184 shows a complex example in which word boundaries overlap. The text “therealways” can be interpreted as “there always” and “the real ways”. Because “there always” is two words rather than three, it is represented on the main token list. The first token has two LX_SUBTERM properties attached to it. The first property represents the word “the”. The second property represents the word “real”. Because the sum of the second property's offset and length is greater than the token’s length ((3 + 4) > 5), the application must obtain the rest of the word from the beginning of the following token. The second token has a single LX_SUBTERM property that represents the word “ways”. Although situations in which word boundaries overlap are rare, the application should be ready to handle the output of Lexical Analysis when such situations occur.

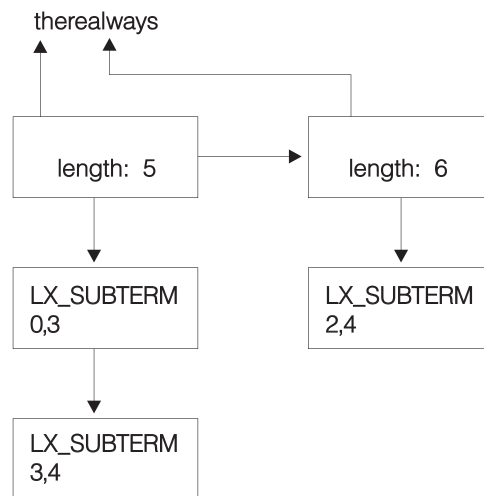


Figure 11. LX_SUBTERM example for “therealways”

Chapter 13. Text-Processing Functions for Text Extraction

Extract Keywords

Extracts keywords from input text for use in text indexing. Processing options include:

- **Part of speech filtering** (*lx_pos_f*)

  Return only specified parts of speech. (This is the only option available for Japanese.)

- **Stopword filtering** (*lx_stopw_f*)

  Ignore words listed in the active stopword addenda dictionary(ies).

- **Word reduction option** (*lx_wordre_f*)

  Return base form of the keyword.

- **Word decomposition option** (*lx_wordde_f*)

  For Germanic languages only, return all possible components of compound keywords.

- **Word modification option** (*lx_word_mod_f*)

  Return keyword and base form in a normalized form.

- **Phonetic key option** (*lx_phonetic_f*)

  Return a phonetic key for the keyword.

- **Start of sentence processing option** (*lx_sent_begin_f*)

  Perform special processing when looking up the first word in a sentence.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_EXTRACT_KEYWORD
- *lx_rqst_type*
- *lx_num_cpg* — number of code page that data is in
- *lx_elt_format* — must be LX_BLOCK_FORMAT
- *lx_elements_ct*
- *lx_elements_p* — pointer to the input data element list
- *lx_reply_p*
- *lx_reply_size*
- *lx_lang_code* — required
- *lx_pos_f*
  - LX_XTRACT_POS_ALL — return all parts of speech

© Copyright IBM Corp. 2000

- LX_XTRACT_POS_NOUNS_ONLY — return nouns only
- LX_XTRACT_POS_NOUNS_ADJS — return nouns and adjectives only
- LX_XTRACT_POS_NOUNS_VERBS — return nouns and verbs only
- LX_XTRACT_POS_NOUNS_VERBS_ADJS — return nouns, verbs, and adjectives only
- LX_XTRACT_JAIRS — JAIRS part of speech filtering (Japanese only — see "JAIRS Part of Speech Filtering" on page 324)

- *lx_word_mod_f* — specify whether keywords and base forms (lemmas) are to be returned in normalized form
- *lx_stopw_f* — specify whether stopwords are to be filtered out
- *lx_wordde_f* — specify whether compound word decomposition is desired
- *lx_wordre_f* — specify whether base forms (lemmas) are to be returned for keywords (if not, only the input form is returned)
- *lx_phonetic_f* — specify whether phonetic keys should be returned
- *lx_sentence_f* — set to LX_TRUE to get text segment identification (sentence separation); otherwise, only word isolation is done
- *lx_sent_begin_f* — specify whether special processing should be done for the first word in a sentence
- *lx_decompnd_further_f* — Only used for compound words. Set to zero to prevent further breaking down of a valid decomposition.
- *lx_dict_tkns_ct* — number of dictionary handles in dictionary list
- *lx_dict_tkns* — dictionary list
- *lx_abbr_tkns_ct* — number of dictionary handles in Abbreviation Addenda Dictionary list
- *lx_abbr_tkns* — Abbreviation Addenda Dictionary list
- *lx_stopw_tkns_ct* — number of dictionary handles in Stopword Addenda Dictionary list
- *lx_stopw_tkns* — Stopword Addenda Dictionary list

Output

The updated Linguistic Tools control block. The following output fields or the areas they point to are updated:

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_reply_p* — logical structure of the output data produced by Extract Keywords will be implemented using the Token List data structure. This output can be accessed using the Token List Utility functions.
- *lx_reply_used*
- *lx_cont_reply_f*
- *lx_hFirstToken*
- *lx_next_elemt_p*

- *lx_next_char_p*

A token consists of:

- Token type:
  - LX_NSEN — new sentence
  - LX_NLIN — new line
  - LX_PARA — new paragraph
  - LX_UDAT — user data
  - LX_IGNORE — user data
  - LX_SOSI — shift in/out characters
  - LX_SRCHTERM — keyword/query term
- String — text content

  For host DBCS output, all strings containing DBCS text will be proper "mixed" strings with DBCS components enclosed in SO and SI characters.
- Properties — one or more properties for each keyword

Keyword property types include:

LX_CSEN — sentence
LX_CKWD — keyword
LX_CLEM — base form
LX_PHON — phonetic key

For each of these properties Linguistic Tools returns a pointer to a structure of the appropriate type:

LX_EKW_CSEN_PVAL
LX_EKW_CKWD_PVAL
LX_EKW_CLEM_PVAL
LX_PHONETIC_PVAL.

These structures are defined in one of the include files shipped as part of the Linguistic Tools.

LX_CSEN contains the number of the sentence in which the keyword was found.

Note: Sentence numbering begins with zero.

LX_CKWD contains information about the keyword such as:

- Keyword type:
  - LX_KEYWORD — regular keyword
  - LX_KWORD_HAS_HYPHEN — keyword contains hyphen(s)
  - LX_KWORD_HAS_SLASH — keyword contains slash(es)
  - LX_KWORD_IN_HYPHEN_FORM — keyword is part of a hyphenated form (broken at the hyphens)
  - LX_KWORD_IN_SLASH_FORM — keyword is part of a slash form (broken at the slashes)
  - LX_COMPOUND_KEYWORD — keyword is a Germanic compound (returned only if *lx_wordde_f* is set to LX_TRUE)
  - LX_COMPONENT_KEYWORD — keyword is a component of a Germanic compound (returned only if *lx_wordde_f* is set to LX_TRUE)
  - LX_KWORD_NOT_IN_DICT — keyword not found in dictionary
- Part of speech:

- LX_EKW_NOUN
  - LX_EKW_VERB
  - LX_EKW_ADJECTIVE
  - LX_EKW_ADVERB
  - LX_EKW_PRONOUN
  - LX_EKW_INTERJECTION
  - LX_EKW_CONJUNCTION
  - LX_EKW_PREPOSITION (Japanese particles are mapped to this part of speech)
  - LX_EKW_UNKNOWN

- Normalized keyword length
- Normalized keyword address
- Keyword order number
- Slash component number (if keyword is a slash word)
- Hyphen component number (if keyword is a hyphen word)
- Decomposition number (if keyword is a compound word)
- Component number of the decomposition (if keyword is a compound word)

Keyword order number and processing of compound, hyphenated, and slash keywords are explained under "Extract Keywords/Extract Query Terms" on page 41.

LX_CLEM contains information about the base form of the keyword. If the word reduction (return base form) and compound word decomposition options are selected (set to LX_TRUE), a compound lemma will be returned if Linguistic Tools cannot find the input word in the dictionary, but is able to analyze it into components. The string returned for the LX_CLEM of a compound word which has been decomposed by Linguistic Tools (that is, was not found in the dictionary) is not guaranteed to be a valid word. The form of the last component in a compound lemma will be a lemma, if one can be found for that component; otherwise, it will be the component form.

The form of the compound lemma depends on whether normalization is selected. If normalization is not selected (that is, *lx_word_mod_f* is set to LX_FALSE), German compound noun casing will be incorrect. Each noun component will begin with a capital letter because German nouns normally begin with a capital letter. Linguistic Tools does not lowercase each component following the first as it concatenates them. Thus, for an input word "Alpenabschnitt", Linguistic Tools returns "AlpenAbschnitt."

If the normalization option is selected, Linguistic Tools will return the string normalized according to the language-specific rules specified by *lx_lang_code* in the function call. For the German word "Alpenabschnitt" the normalized string would be "alpenabschnitt."

LX_PHON contains a phonetically respelled representation of the keyword, generated by the same algorithm which produces phonetic representations for Spell Aid.

For Japanese and Chinese processing, the keyword property types returned are LX_CSEN and LX_CKWD. Dictionary information is not available to provide either base (lemma) forms or phonetic keys for Japanese or Chinese words.

Japanese words have JPOS codes, Chinese words have CPOS codes, and Korean words have Double-Byte Parse Codes (DBPCs) to indicate their parts of speech. Each of these codes maps to a condensed part of speech corresponding to the parts of speech listed above under LX_CKWD. (See "Japanese Part of Speech (POS) Codes" on page 324, "Korean Double-Byte Parsing Codes (DBPCs)" on page 331, and "Chinese Double-Byte Parsing Codes (DBPCs)" on page 287 for details.) When *lx_pos_f* is set to any option other than LX_XTRCT_JAIRS, part of speech filtering will use the condensed part of speech to determine whether or not a word should be returned as a keyword. For example, if *lx_pos_f* is set to LX_XTRCT_NOUNS, and a Japanese word has a JPOS value of 19, which corresponds to a condensed part of speech code of LX_EKW_NOUN, the word will be returned as a keyword.

When *lx_pos_f* = LX_XTRCT_JAIRS, filtering will be based on the JPOS code; this is referred to as JAIRS filtering. If the JPOS code of the word is one of those listed for JAIRS filtering, a keyword will be returned for that word. For example, if *lx_pos_f* is set to LX_XTRCT_JAIRS and a word has a JPOS value of 13, it will be returned as a keyword; if the word has a JPOS value of 86, it will not be returned as a keyword.

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

- LX_BAD_FUNCT_CODE — invalid function code
- LX_BAD_RQST_TYPE — invalid request type
- LX_RC_OK — processing successfully completed
- LX_UNEXPECTED_RC — unexpected or unknown condition
- LX_BAD_SERV_HANDLE — invalid Service Area Handle
- LX_BAD_DICT_TKN — invalid dictionary token
- LX_BAD_DICT_TKN_CNT — invalid dictionary token count
- LX_BAD_ELE_FORMAT — invalid data element format
- LX_BAD_ELE_TYPE — invalid element type
- LX_BAD_ELEMENTS_CT — invalid element count
- LX_BAD_POS_FILTER_PARM — invalid part of speech filter input—invalid value for *lx_pos_f* field in control block
- LX_BAD_SENT_PARM — invalid sentence determination option input
- LX_BAD_STOP_WORD_PARM — invalid stopword option input (*lx_stopw_f*)
- LX_BAD_WORD_DECOMP_PARM — invalid word decomposition option input (*lx_wordde_f*)
- LX_BAD_WORD_MOD_PARM — invalid word modification option input (*lx_word_mod_f*)
- LX_BAD_WORD_REDUCE_PARM — invalid word reduction option input (*lx_wordre_f*)
- LX_CPG_NOT_SUPPORTED — code page not supported
- LX_ELE_P — missing element list
- LX_ELT_FORMAT_CHG — change of data element format not allowed
- LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library

- LX_EXTRACT_PARMS_CHG — parameters were changed on EXTRACT continue reply
- LX_FUNC_CODE_CHG — change of function code not allowed
- LX_MEM_ALLOC_ERR — memory allocation error
- LX_NO_REPLY_P — missing reply area
- LX_REPLY_FULL — reply area is full
- LX_REPLY_2_SMALL — reply area is too small to allow any output

Language-Specific Processing

For information about how this function breaks the input text into words for a specific language, see “Definition of a Word in SBCS Text” on page 80 and Appendix D, “Language-Specific Processing” on page 281.

For a language which is not supported by the Linguistic Tools, **Extract Keywords** will do either word isolation alone or word isolation and sentence determination, depending on what the application requests. The application must know that without a dictionary available, sentence determination will work on a best-guess basis only. For example, the text “Mr. Dumas has a M.S. degree in Physics.” could be analyzed as four sentences:

1. Mr
2. Dumas has a M
3. S
4. degree in Physics

The keyword normalization schemes for the various languages are described in “Word Normalization” on page 44.

Application Programming Guidelines

The language code field is required for this function. For DBCS text, the correct language code ***must*** be specified. For SBCS text, normalization processing will be determined by the language code. A value of zero is valid if no language-specific processing is desired.

If the reply area is not big enough to hold all of the output, **Extract Keywords** will return a part of the output and a return code LX_REPLY_FULL. The application should get the output from the reply area and call **Extract Keywords** again with lx_rqst_type set to LX_CONT_REPLY.

If special start-of-sentence processing is requested (*lx_sent_begin_f* set to LX_TRUE), Linguistic Tools will do extra processing during dictionary lookup for the first word in an input sentence:

- If the input word is only found in the dictionary with an initial capital letter, that is the form that Linguistic Tools will use.
- If the input word is only found in the dictionary in all-lowercase, Linguistic Tools will use only that form.
- If the input word is found both with initial capital and in all-lowercase, Linguistic Tools will carry both forms into the next stage of its processing.
- If the input word is not found in the dictionary, it will be maintained as-is.

This processing is done before any of the other options (part of speech filtering, stopword filtering, word decomposition, and so on) apply. It does not modify the processing of these other options. Forms found through start of sentence processing may be filtered out by some of the other options. If such filtering does not occur, Linguistic Tools will return additional forms for some sentence-initial words.

To support the lx_sent_begin_f processing when no morphological output is requested, the following processing occurs:

When lx_sent_begin_f is on and all morphology-related flags (*lx_pos_f, lx_wordex_f, lx_wordre_f*, and *lx_synonym_f*) are off, a fast path will be taken which may create multiple LX_SRCHTERM tokens for the first word of the sentence, that is, one for each case-insensitive matching word in a dictionary. Note that, if stopword filtering is on, each of these multiple words will have stopword matching applied; thus some candidates may be eliminated, and not all dictionary matches may actually be returned. The token will have an LX_CKWD property with POS = 0. For the first word of a sentence, multiple tokens with word number 0 may thus be created. For words not at the beginning of a sentence, only one such token will be created for a normal keyword.

Flags not related to morphology (*lx_phonetic_f, lx_word_mod_f, lx_wordde_f, lx_stopw_f*) may be either on or off. Only the last two (Germanic compound word component isolation and stopword filtering) will affect how many LX_SRCHTERM tokens are returned.

See "Case Matching between Input Words and Dictionary Words" on page 82 for information on how this function generally handles different case patterns in input text.

Example

NOTE: Input simulates a Germanic text. "Bookstore" is used to show how compound words are processed in Germanic languages; this processing is not supported for English.

```
                        I WON A GIFT FROM THAT BOOKSTORE

Word order number       0  1  2   3   4    5      6
```

Assume that the words I, A, FROM and THAT are deleted because they were in the Stopword Addenda Dictionary.

Output can be described logically as follows (data structures are defined in one of the include files shipped as part of the Linguistic Tools):

```
Token type       LX_NSEN
String           contains no data
Property
  LX_CSEN – Sentence
     0                    – sentence number 0

Token type      LX_SRCHTERM
String (WON)
Properties
```

```
  LX_CKWD – keyword
     LX_KEYWORD            – keyword type
     LX_EKW_VERB           – part of speech
     3                     – normalized keyword length
     ----> 'WON'           – address of 'won'
     1                     – keyword order number
     0                     – slash component number
     0                     – hyphen component number
     0                     – decomposition number
     0                     – component number
  LX_CLEM – base form
     LX_EKW_VERB           – part of speech
     3                     – base form length
     ----> 'WIN'           – address of 'WIN'

Token type      LX_SRCHTERM
String (GIFT)
Properties
  LX_CKWD – keyword
     LX_KEYWORD            – keyword type
     LX_EKW_NOUN           – parts of speech
        and LX_EKW_VERB
     4                     – normalized keyword length
     ----> 'GIFT'          – address of 'gift'
     3                     – keyword order number
     0                     – slash component number
     0                     – hyphen component number
     0                     – decomposition number
     0                     – component number
  LX_CLEM – base form
     LX_EKW_NOUN           – part of speech
     4                     – base form length
     ----> 'GIFT'          – address of 'GIFT'

  LX_CLEM – base form
     LX_EKW_VERB           – part of speech
     4                     – base form length
     ----> 'GIFT'          – address of 'GIFT'

Token type      LX_SRCHTERM
String (BOOKSTORE)
Properties

  LX_CKWD – keyword
     LX_COMPOUND_KEYWORD   – keyword type
     LX_EKW_UNKNOWN        – part of speech
     10                    – normalized keyword length
     ----> 'BOOKSTORE'     – address of 'bookstore'
     6                     – keyword order number
     0                     – slash component number
     0                     – hyphen component number
     0                     – decomposition number
     0                     – component number

  LX_CLEM – base form
     LX_EKW_UNKNOWN        – part of speech
     9                     – base form length
     ----> 'BOOKSTORE'     – address of 'BOOKSTORE'
```

```
Token type      LX_SRCHTERM
String (BOOK)
Properties

  LX_CKWD – keyword
     LX_COMPONENT_KEYWORD  – keyword type
     LX_EKW_NOUN           – part of speech
     4                     – normalized keyword length
     ----> 'BOOK'          – address of 'book'
     6                     – keyword order number
     0                     – slash component number
     0                     – hyphen component number
     0                     – decomposition number
     0                     – component number

  LX_CLEM – base form
     LX_EKW_NOUN           – part of speech
     4                     – base form length
     ----> 'BOOK'          – address of 'BOOK'

Token type      LX_SRCHTERM
String (STORE)
Properties

  LX_CKWD – keyword
     LX_COMPONENT_KEYWORD  – keyword type
     LX_EKW_NOUN           – part of speech
     5                     – normalized keyword length
     ----> 'STORE'         – address of 'store'
     6                     – keyword order number
     0                     – slash component number
     0                     – hyphen component number
     0                     – decomposition number
     1                     – component number

  LX_CLEM – base form
     LX_EKW_NOUN           – part of speech
     5                     – base form length
     ----> 'STORE'         – address of 'STORE'
```

Extract Query Terms

Extracts words from input text for use as seach terms in queries. Processing options include:

- **Part of speech filtering** (*lx_pos_f*)

  Return only specified parts of speech. (This is the only option available for Japanese.)

- **Stopword filtering** (*lx_stopw_f*)

  Ignore words listed in the active stopword addenda dictionary(ies).

- **Word reduction option** (*lx_wordre_f*)

  Return base form of the query term.

- **Word decomposition option** (*lx_wordde_f*)

  For Germanic languages only, return all possible components of compound query terms.

- **Word modification option** (*lx_word_mod_f*)

  Return query term, base form, inflected forms, and synonyms in a normalized form.

- **Word expansion option** (*lx_wordex_f*)

  Return inflected forms of a query term.

- **Synonym option** (*lx_synonym_f*)

  Return synonyms of the query term.

- **Phonetic key option** (*lx_phonetic_f*)

  Return a phonetic key for the query term.

- **Start of sentence processing option** (*lx_sent_begin_f*)

  Perform special processing when looking up the first word in a sentence.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_func_code* — set to LX_EXTRACT_QRY_TERM
- *lx_rqst_type*
- *lx_dict_tkns_ct* — number of dictionary handles in the input dictionary list
- *lx_dict_tkns* — input dictionary list
- *lx_num_cpg* — number of code page that data is in
- *lx_elements_ct*
- *lx_elements_p* — pointer to the input data element list
- *lx_reply_p*
- *lx_reply_size*

- *lx_elt_format* — must be LX_BLOCK_FORMAT
- *lx_lang_code* — required
- *lx_pos_f*
  - LX_XTRACT_POS_ALL — return all parts of speech
  - LX_XTRACT_POS_NOUNS_ONLY — return nouns only
  - LX_XTRACT_POS_NOUNS_ADJS — return nouns and adjectives only
  - LX_XTRACT_POS_NOUNS_VERBS — return nouns and verbs only
  - LX_XTRACT_POS_NOUNS_VERBS_ADJS — return nouns, verbs, and adjectives only
  - LX_XTRACT_JAIRS — JAIRS part of speech filtering (Japanese only — see "JAIRS Part of Speech Filtering" on page 324)
- *lx_word_mod_f* — specify whether query terms should be returned in normalized form (note that *lx_lang_code* must be set correctly for language-specific processing to be performed)
- *lx_stopw_f* — specify whether stopwords should be filtered out
- *lx_wordde_f* — specify whether compound word decomposition should be performed (for Germanic languages only)
- *lx_wordre_f* — specify whether base forms (lemmas) should be returned for query terms (if not, then only the input form is returned)
- *lx_wordex_f* — specify whether inflected forms should be returned for each query term
- *lx_synonym_f* — specify whether synonyms are desired for each query term
- *lx_phonetic_f* — specify whether phonetic keys should be returned
- *lx_decompnd_further_f* — Only used for compound words. Set to zero to prevent further breaking down of a valid decomposition.
- *lx_stopw_tkns*
- *lx_stopw_tkns_ct*
- *lx_sent_begin_f* — specify whether special processing should be done when looking up the first word of a sentence

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated:

- *lx_rc* — return code
- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_reply_p* — The logical structure of the output data produced by Extract Query Terms will be implemented using the Token List data structure. This output can be accessed using the Token List Utility functions.
- *lx_reply_used*
- *lx_cont_reply_f*
- *lx_hFirstToken*
- *lx_next_elemt_p*
- *lx_next_char_p*

A token consists of

- Token type:
  - LX_NSEN — new sentence
  - LX_NLIN — new line
  - LX_PARA — new paragraph
  - LX_UDAT — user data
  - LX_IGNORE — user data
  - LX_SOSI — shift in/out characters
  - LX_SRCHTERM — keyword/query term
- String — text content

  For host DBCS output, all strings containing DBCS text will be proper “mixed” strings with DBCS components enclosed in SO and SI characters.
- properties — one or more properties for each query term

Query term property types include:

- LX_CKWD — query term
- LX_CLEM — base form
- LX_PHON — phonetic key
- LX_CSYN — synonym
- LX_CINF — inflected form

For each of these properties Linguistic Tools returns a pointer to a structure of the appropriate type:

- LX_EQT_CKWD_PVAL
- LX_EQT_CLEM_PVAL
- LX_PHONETIC_PVAL.
- LX_EQT_CSYN_PVAL
- LX_EQT_CINF_PVAL

These structures are defined in one of the include files shipped as part of the Linguistic Tools.

LX_CKWD contains information about the query term such as:

- Query term type:
  - LX_KEYWORD — regular query term
  - LX_KWORD_HAS_HYPHEN — query term contains hyphen(s)
  - LX_KWORD_HAS_SLASH — query term contains slash(es)
  - LX_KWORD_IN_HYPHEN_FORM — query term is part of a hyphenated form (broken at the hyphens)
  - LX_KWORD_IN_SLASH_FORM — query term is part of a slash form (broken at the slashes)
  - LX_COMPOUND_KEYWORD — query term is a Germanic compound (returned only if *lx_wordde_f* is set to LX_TRUE)
  - LX_COMPONENT_KEYWORD — query term is a component of a Germanic compound (returned only if *lx_wordde_f* is set to LX_TRUE)
  - LX_KWORD_NOT_IN_DICT — keyword not found in dictionary
- Part of speech:

- LX_EKW_NOUN
  - LX_EKW_VERB
  - LX_EKW_ADJECTIVE
  - LX_EKW_ADVERB
  - LX_EKW_PRONOUN
  - LX_EKW_INTERJECTION
  - LX_EKW_CONJUNCTION
  - LX_EKW_PREPOSITION (Japanese particles are mapped to this part of speech)
  - LX_EKW_UNKNOWN

- Normalized query term length
- Normalized query term address
- Query term order number
- Slash component number (if query term is a slash word)
- Hyphen component number (if query term is a hyphen word)
- Decomposition number (if query term is a compound word)
- Component number of the decomposition (if query term is a compound word)

Query term order number and processing of compound, hyphenated, and slash query terms are explained under "Extract Keywords/Extract Query Terms" on page 41.

Similarly, LX_CLEM, LX_CSYN, and LX_CINF contain information about the base form, synonyms, and inflected forms of the query term.

If the word reduction (return base form) and compound word decomposition options are selected (set to LX_TRUE), a compound lemma will be returned if Linguistic Tools cannot find the input word in the dictionary, but is able to analyze it into components. The string returned for the LX_CLEM of a compound word which has been decomposed by Linguistic Tools (that is, was not found in the dictionary) is not guaranteed to be a valid word. The form of the last component in a compound lemma will be a lemma, if one can be found for that component; otherwise, it will be the component form.

The form of the compound lemma depends on whether normalization is selected. If normalization is not selected (that is, *lx_word_mod_f* is set to LX_FALSE), German compound noun casing will be incorrect. Each noun component will begin with a capital letter because German nouns normally begin with a capital letter. Linguistic Tools does not lowercase each component following the first as it concatenates them. Thus, for an input word "Alpenabschnitt", Linguistic Tools returns "AlpenAbschnitt."

If the normalization option is selected, Linguistic Tools will return the string normalized according to the language-specific rules specified by *lx_lang_code* in the function call. For the German word "Alpenabschnitt" the normalized string would be "alpenabschnitt."

LX_PHON contains a phonetically respelled representation of the query term, generated by the same algorithm which produces phonetic representations for Spell Aid.

For Japanese and Chinese processing, the query term property types returned are LX_CSEN and LX_CKWD. Dictionary information is not available to provide either base (lemma) forms or phonetic keys for Japanese or Chinese words.

Japanese words have JPOS codes, Chinese words have CPOS codes, and Korean words have Double-Byte Parse Codes (DBPCs) to indicate their parts of speech. Each of these codes maps to a condensed part of speech corresponding to the parts of speech listed above under LX_CKWD. (See “Japanese Part of Speech (POS) Codes” on page 324, “Korean Double-Byte Parsing Codes (DBPCs)” on page 331, and “Chinese Double-Byte Parsing Codes (DBPCs)” on page 287 for details.) When *lx_pos_f* is set to any option other than LX_XTRCT_JAIRS, part of speech filtering will use the condensed part of speech to determine whether or not a word should be returned as a query term. For example, if *lx_pos_f* is set to LX_XTRCT_NOUNS, and a Japanese word has a JPOS value of 19, which corresponds to a condensed part of speech code of LX_EKW_NOUN, the word will be returned as a query term.

When *lx_pos_f* = LX_XTRCT_JAIRS, filtering will be based on the JPOS code; this is referred to as JAIRS filtering. If the JPOS code of the word is one of those listed for JAIRS filtering, a query term will be returned for that word. For example, if *lx_pos_f* is set to LX_XTRCT_JAIRS and a word has a JPOS value of 13, it will be returned as a query term; if the word has a JPOS value of 86, it will not be returned as a query term.

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_ELE_TYPE — invalid element type
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_POS_FILTER_PARM — invalid part of speech filter input—invalid value for *lx_pos_f* field in control block
LX_BAD_SENT_PARM — invalid sentence determination option input
LX_BAD_SERV_HANDLE — invalid Service Area Handle input
LX_BAD_STOP_WORD_PARM — invalid stopword option input (*lx_stopw_f*)
LX_BAD_SYN_PARM — invalid synonym option input (*lx_synonym_f*)
LX_BAD_WORD_DECOMP_PARM — invalid word decomposition option input (*lx_wordde_f*)
LX_BAD_WORD_EXPAND_PARM — invalid word expansion option input (*lx_wordex_f*)
LX_BAD_WORD_MOD_PARM — invalid word modification option input (*lx_word_mod_f*)
LX_BAD_WORD_REDUCE_PARM — invalid word reduction option input (*lx_wordre_f*)
LX_CPG_NOT_SUPPORTED — code page not supported

LX_ELE_P — missing element list
LX_ELT_FORMAT_CHG — change of data element format not allowed
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_EXTRACT_PARMS_CHG — parameters were changed on EXTRACT continue reply
LX_FUNC_CODE_CHG — change of function code not allowed
LX_MEM_ALLOC_ERR — memory allocation error
LX_NO_REPLY_P — missing reply area
LX_REPLY_FULL — reply area is full
LX_REPLY_2_SMALL — reply area is too small to allow any output

Language-Specific Processing

For information about how this function breaks the input text into words for a specific language, see "Definition of a Word in SBCS Text" on page 80 and Appendix D, "Language-Specific Processing" on page 281.

The query term normalization schemes for the different languages are described in "Word Normalization" on page 44.

Application Programming Guidelines

The language code field is required. For DBCS text, the correct language code ***must*** be specified. For SBCS text, the language code determines what kind of normalization processing is performed. A value of zero is valid if no language-specific processing is desired.

If the reply area is not big enough to hold all of the output, **Extract Query Terms** will return a part of the output and a return code LX_REPLY_FULL. The application should get the output from the reply area and call **Extract Query Terms** again with lx_rqst_type set to LX_CONT_REPLY.

See the application programming guidelines for the **Extract Keywords** function for an explanation of the special processing done for the first word in a sentence. Note that since the **Extract Query Terms** function expects to get only one sentence in its input, it will assume that the first word in its input is a possible start of sentence and will process it as such if the start of sentence option is selected (*lx_sent_begin_f* set to LX_TRUE).

See "Case Matching between Input Words and Dictionary Words" on page 82 for information on how this function handles different case patterns in input text.

Output example

NOTE: Input simulates a Germanic text. "Bookstore" is used to show how compound words are processed in Germanic languages; this processing is not supported for English.

```
                          I WON A GIFT FROM THAT BOOKSTORE

  Word order number       0  1  2   3   4    5     6
```

Assume that the words I, A, FROM and THAT are deleted because they were on the stopword list.

Output can be described logically as follows:

```
Token type      LX_SRCHTERM
String (WON)
Properties
  LX_CKWD – query term
     LX_KEYWORD           – query term type
     LX_EKW_VERB          – part of speech
     3                    – Normalized query term length
     ----> 'WON'          – address of 'won'
     1                    – query term order number
     0                    – slash component number
     0                    – hyphen component number
     0                    – decomposition number
     0                    – component number
     20                   – query term weight factor

  LX_CLEM – base form
     LX_EKW_VERB          – part of speech
     3                    – base form length
     ----> 'WIN'          – address of 'WIN'
     15                   – base form weight factor

  LX_CINF – Inflected form
     7                    – inflected form length
     ----> 'WINNING'      – address of 'WINNING'
     10                   – inflected form weight factor

  LX_CINF – Inflected form
     3                    – inflected form length
     ----> 'WINS'         – address of 'WINS'
     10                   – inflected form weight factor

  LX_CINF – Inflected form
     3                    – inflected form length
     ----> 'WON'          – address of 'WON'
     10                   – inflected form weight factor

  LX_CSYN – Synonym
     LX_EKW_VERB          – part of speech
     7                    – synonym length
     ----> 'CAPTURE'      – address of 'CAPTURE'
     6                    – synonym weight factor

  LX_CSYN – Synonym
     LX_EKW_VERB          – part of speech
     3                    – synonym length
     ----> 'GET'          – address of 'GET'
     6                    – synonym weight factor

  LX_CSYN – Synonym
     LX_EKW_VERB          – part of speech
     4                    – synonym length
     ----> 'TAKE'         – address of 'TAKE'
     6                    – synonym weight factor

Token type      LX_SRCHTERM
String (GIFT)
Properties
  LX_CKWD – query term
```

```
    LX_KEYWORD              – query term type
    LX_EKW_NOUN             – parts of speech
      and LX_EKW_VERB
    4                       – Normalized query term length
    ----> 'GIFT'            – address of 'gift'
    3                       – query term order number
    0                       – slash component number
    0                       – hyphen component number
    0                       – decomposition number
    0                       – component number
    20                      – query term weight factor

  LX_CLEM – base form
    LX_EKW_NOUN             – part of speech
    4                       – base form length
    ----> 'GIFT'            – address of 'GIFT'
    15                      – base form weight factor

  LX_CINF – Inflected form
    5                       – inflected form length
    ----> 'GIFTS'           – address of 'GIFTS'
    10                      – inflected form weight factor

  LX_CLEM – base form
    LX_EKW_VERB             – part of speech
    4                       – base form length
    ----> 'GIFT'            – address of 'GIFT'
    15                      – base form weight factor

  LX_CINF – Inflected form
    7                       – inflected form length
    ----> 'GIFTING'         – address of 'GIFTING'
    10                      – inflected form weight factor

  LX_CINF – Inflected form
    6                       – inflected form length
    ----> 'GIFTED'          – address of 'GIFTED'
    10                      – inflected form weight factor

  LX_CINF – Inflected form
    6                       – inflected form length
    ----> 'GIFTS'           – address of 'GIFTS'
    10                      – inflected form weight factor

  LX_CSYN – Synonym
    LX_EKW_VERB             – part of speech
    4                       – synonym length
    ----> 'GIVE'            – address of 'GIVE'
    6                       – synonym weight factor

  LX_CSYN – Synonym
    LX_EKW_VERB             – part of speech
    7                       – synonym length
    ----> 'PRESENT'         – address of 'PRESENT'
    6                       – synonym weight factor

Token type      LX_SRCHTERM
String (BOOKSTORE)
```

```
Properties

  LX_CKWD – query term
     LX_COMPOUND_KEYWORD   – query term type
     LX_EKW_UNKNOWN        – part of speech
     9                     – Normalized query term length
     ----> 'BOOKSTORE'     – address of 'bookstore'
     6                     – query term order number
     0                     – slash component number
     0                     – hyphen component number
     0                     – decomposition number
     0                     – component number
     20                    – query term weight factor

  LX_CLEM – base form
     LX_EKW_UNKNOWN        – part of speech
     9                     – base form length
     ----> 'BOOKSTORE'     – address of 'BOOKSTORE'
     15                    – base form weight factor


Token type      LX_SRCHTERM
String (BOOK)
Properties

  LX_CKWD – query term
     LX_COMPONENT_KEYWORD  – query term type
     LX_EKW_NOUN           – part of speech
     4                     – Normalized query term length
     ----> 'BOOK'          – address of 'bookstore'
     6                     – query term order number
     0                     – slash component number
     0                     – hyphen component number
     0                     – decomposition number
     0                     – component number
     10                    – query term weight factor

  LX_CLEM – base form
     LX_EKW_NOUN           – part of speech
     4                     – base form length
     ----> 'BOOK'          – address of 'BOOK'
     6                     – base form weight factor

  LX_CINF – Inflected form
     5                     – inflected form length
     ----> 'BOOKS'         – address of 'BOOKS'
     4                     – inflected form weight factor

  LX_CSYN – Synonym
     LX_EKW_NOUN           – part of speech
     6                     – synonym length
     ----> 'VOLUME'        – address of 'VOLUME'
     2                     – synonym weight factor

  LX_CSYN – Synonym
     LX_EKW_NOUN           – part of speech
     4                     – synonym length
     ----> 'TOME'          – address of 'TOME'
```

```
    2                      – synonym weight factor

  LX_CLEM – base form
    LX_EKW_VERB            – part of speech
    4                      – base form length
    ----> 'BOOK'           – address of 'BOOK'
    6                      – base form weight factor

  LX_CINF – Inflected form
    7                      – inflected form length
    ----> 'BOOKING'        – address of 'BOOKING'
    4                      – inflected form weight factor

  LX_CINF – Inflected form
    6                      – inflected form length
    ----> 'BOOKED'         – address of 'BOOKED'
    4                      – inflected form weight factor

  LX_CINF – Inflected form
    6                      – inflected form length
    ----> 'BOOKED'         – address of 'BOOKED'
    4                      – inflected form weight factor

  LX_CSYN – Synonym
    LX_EKW_VERB            – part of speech
    4                      – synonym length
    ----> 'LIST'           – address of 'LIST'
    2                      – synonym weight factor

Token type      LX_SRCHTERM
String (STORE)
Properties

  LX_CKWD – query term
    LX_COMPONENT_KEYWORD  – query term type
    LX_EKW_NOUN            – part of speech
    5                      – Normalized query term length
    ----> 'STORE'          – address of 'bookstore'
    6                      – query term order number
    0                      – slash component number
    0                      – hyphen component number
    0                      – decomposition number
    1                      – decomposition component number
    10                     – query term weight factor
```

Chapter 14. Text-Processing Functions for Other

Single-Word Data Element Creation

Takes input text in the form of a block format data element list, groups characters into simple tokens (words), and associates with each word a single-word format data element pointing to it. Output is a single-word format data element list. See "Use of Single-Word Format Data Elements" on page 71 for details.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_rqst_type*
- *lx_func_code* — set to LX_DATA_ELE_LIST
- *lx_num_cpg* — number of code page that data is in
- *lx_elements_ct*
- *lx_elements_p* — pointer to input data element list
- *lx_elt_format* — must be LX_BLOCK_FORMAT
- *lx_reply_p*
- *lx_reply_size*

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated:

- *lx_rc* — return code
- lx_serv_area_p — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_cont_reply_f*
- *lx_reply_used*
- *lx_delivered_out_units*
- *lx_reply_p* — Reply Area will be filled with single-word data elements, each pointing to a single word from the input text. The fields used within the output elements are the following:
  - *lx_data_len* — length of the single word pointed to by this element
  - *lx_data_p* — set to point directly into the input text, to a single word from that text
  - *lx_trail* — the first input text character following the word referenced by this element
  - *lx_input_f*

    The LX_ELMT_JOIN flag will be set to indicate that this element is to be joined to the next element. A value of LX_ELMT_REMOVED indicates that punctuation preceding the word represented by the current data element has been filtered out and is not represented in the output data element list. Other possible values of this field are listed in Table 16 on page 75.

© Copyright IBM Corp. 2000

- *lx_rc* — initialized to 0
- *lx_databyte* — initialized to 0

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_ELE_TYPE — invalid element type
LX_BAD_ELEMENTS_CT — invalid element count
LX_BAD_STR_ADDR — invalid or NULL string address provided
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ELT_FORMAT_CHG — change of element format not allowed
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_FUNC_CODE_CHG — change of function code not allowed
LX_REPLY_FULL — reply area is full

Application Programming Guidelines

The data pointers within the data element list passed as input to this function should point directly into the text to be divided into words, not to strings containing copies of text from the document. This function will not alter the input text in any way.

From an input data element list containing block format elements, the data element creation function will build a single-word format data element list describing the input text, and put it into the Reply Area. If the same text is to be passed as input to a Linguistic Tools function later, the single-word format list provided by the service should be passed to avoid unnecessary processing.

Chapter 15. Text-Processing Functions for Language-Specific Support

Compound Word Component Isolation

This function presents all possible decompositions for an input compound word.

This function is supported only for the Germanic languages: Danish, Dutch, German (National and Swiss), Icelandic, Norwegian (Bokmål and Nynorsk), and Swedish. Compound words in Japanese are processed by Lexical Analysis.

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

- *lx_serv_area_p* — Linguistic Tools Service Area Handle returned from the **Initialize** function
- *lx_rqst_type* — must be set to LX_NEW_REQ
- *lx_func_code* — set to LX_COMPOUND_WD_ISOL
- *lx_dict_tkns_ct*
- *lx_dict_tkns*
- *lx_num_cpg* — number of code page that data is in
- *lx_elements_ct*
- *lx_elements_p* — pointer to input data element list
- *lx_elt_format* — must be LX_SINGLE_WORD_FMT
- *lx_reply_p*
- *lx_reply_size*
- *lx_decompnd_further_f* — Null prevents the function from further breaking down an earlier decomposition. When set to ON, the function will break down previous decomposition even further to create new decompositions.

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- *lx_rc*
- lx_serv_area_p — Linguistic Tools Service Handle Area returned from the **Initialize** function
- *lx_reply_used*
- *reply_p* — The Linguistic Tools reply area will be filled with the following output:
  - Number of decompositions found (type LX_USHORT)
  - For each decomposition:
    - Language of the decomposition (type LX_USHORT)

© Copyright IBM Corp. 2000

- Number of components in decomposition (type LX_UCHAR)
- For each component in decomposition:
  - Position within compound word of 1st character of the component (type LX_UCHAR); this is zero for the first component.
  - Code indicating where component may appear in a compound word (type LX_UCHAR). The codes are as follows:
    - LX_BACK_STANDALONE — component may be the back portion of a compound, and is also a word by itself
    - LX_NONBACK_ONLY — component may only be any non-back portion of a compound
    - LX_ANYWHERE_STANDALONE — component may appear anywhere in a compound, and is also a word by itself
    - LX_NONBACK_STANDALONE — component may be any non-back portion of a compound, and is also a word by itself
    - LX_BACK_ONLY — component may only be the back portion of a compound
    - LX_NONFRONT_ONLY — component may only be any non-front portion of a compound
    - LX_NONFRONT_STANDALONE — component may be any non-front portion of a compound, and is also a word by itself
  - Flag indicating whether elision occurred at the junction between this component and the next one (type LX_BOOLBYTE); set to LX_TRUE if elision occurred, LX_FALSE otherwise.
  - Length of component (type LX_UCHAR)
  - Component (string of characters, each of type LX_UCHAR)

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The list that follows describes the possible values of the return codes:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_DICT_TKN — invalid dictionary token
LX_BAD_DICT_TKN_CNT — invalid dictionary token count
LX_BAD_ELE_FORMAT — invalid data element format
LX_BAD_WORD — invalid word
LX_CPG_NOT_SUPPORTED — code page not supported
LX_ERROR_LOAD_MODULE — error loading Dynamic Link Library
LX_IO_ERROR — file input/output error
LX_MEM_ALLOC_ERR — memory allocation error
LX_NO_DECOMP — no decompositions found
LX_REPLY_FULL — reply area is full
LX_UNEXPECTED_RC — unknown or unexpected return-code received

Application Programming Guidelines

See “Use of Single-Word Format Data Elements” on page 71 for a description of how input words are to be passed to this function by a single-word format data element list.

Chapter 16. General Utility Functions

NlpFindDicts

This function deals with the dictionaries installed on a system and how they are located. There are two types of processing available, based on the contents of the path_f parameter:

- Find the dictionaries available on the system and return information for each dictionary found.
- Find the dictionaries available on the system and return a path specification to indicate the locations where the dictionaries were found.

```
LX_USHORT LX_CALLMODE
NlpFindDicts(LX_HNLPSERV hNlpService,
             LX_UCHAR    PathFlag,
             LX_USHORT   NumCpg,       (reserved for future use)
             LX_PUCHAR   pReply,
             LX_UINT     ReplySize,
             LX_PUINT    pReplyUsed,
             LX_PULONG   pOutUnits);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

PathFlag may contain one of the following values:

- LX_FALSE indicates that full dictionary information is desired
- LX_TRUE indicates that a path is desired which contains the locations where dictionaries were found

NumCpg Number of code page that flat file addenda dictionaries are in

pReply Address of the reply area in which the list of available dictionaries will be returned

ReplySize Size of the reply area

pReplyUsed Address of the place where the number of bytes in the reply area actually used on output will be returned

pOutUnits Address of the place where the number of dictionary entries placed in the reply area will be returned

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pReply the output in this area depends on the type of processing requested:

If PathFlag was set to LX_FALSE, this area contains the list of

© Copyright IBM Corp. 2000

available dictionaries. The following information is given in the structure pointed to by LX_AVL_REPLY_P for each dictionary:

- Dictionary type field — indicates the type of dictionary found (type LX_UCHAR)
- Dictionary content field — describes the information contained in the dictionary (type LX_UCHAR)
- Dictionary version number — identifies the level of a system dictionary (type LX_UCHAR)
- Dictionary language code — specifies the language of the dictionary (type LX_USHORT)
- Dictionary name — NULL-terminated variable-length string containing the fully-qualified name of a dictionary (string of type LX_UCHAR)

If PathFlag was set to LX_TRUE, this area contains a string of type LX_UCHAR. The string contains the locations on the system where dictionaries were found in the same format as the dictionary search path, as described in "Dictionary Search Path" on page 108.

pReplyUsed The number of bytes in the reply area actually used on output

pOutUnits If PathFlag was set to LX_FALSE, this value will indicate the number of dictionary entries placed in the Reply Area. If PathFlag was set to LX_TRUE, the number of output units will be set to 1 to indicate that a single path was successfully placed in the Reply Area.

rc return code

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

Return codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

- LX_BAD_FUNCT_CODE — invalid function code
- LX_BAD_RQST_TYPE — invalid request type
- LX_RC_OK — processing successfully completed
- LX_UNEXPECTED_RC — unexpected or unknown condition
- LX_BAD_SERV_HANDLE — invalid Service Area Handle
- LX_BAD_PARM — missing or illegal input value for parameter
- LX_BAD_SERV_HANDLE — invalid Service Area Handle input
- LX_END_OF_STORAGE — end of system storage reached
- LX_IO_ERROR — file input/output error
- LX_NO_REPLY_P — missing reply area
- LX_REPLY_2_SMALL — reply area is too small to allow any output

Application Programming Guidelines

One suggested use of this function is for an application to be able to present to the user a list of dictionaries available on his or her system. From this list, the user should be able to select a dictionary, which would cause the application to pass the fully-qualified name of the selected dictionary to the Activate Dictionary function. See the description of Activate Dictionary for more details.

Another suggested use of this function is for an application to identify in advance where all dictionaries are located on a user's system, using the return path option (by setting PathFlag to LX_TRUE). Then the application may manipulate this path by adding or deleting search locations before returning it to the service to be committed to memory (by calling **NlpSetSearchPath()**). This will enable the application to limit later dictionary searches to only those locations where dictionaries have been found or which were specifically requested by the application.

If the application has used **NlpSetSearchPath()** to set a default dictionary search path, **NlpFindDicts()** will follow that path. If the application has not set its own search path, Linguistic Tools will observe the defaults described in "Dictionary Search Path" on page 108.

If PathFlag was set to LX_FALSE, the dictionaries returned in the Reply Area will have been checked to verify that they belong to the Linguistic Tools. This check consists of reading the dictionary header for dictionaries with recognized extensions and looking for identifying characteristics. The default extensions are DIC, LEG, MED, STW, STB, ABR, ABB, ADD, and ADB. If the application has substituted its own default extensions by calling **NlpSetDictDef()**, Linguistic Tools will recognize these instead.

If an attempt is made to read the header of a dictionary that is on a LAN server, and the file cannot be opened (because it is currently locked by another application, or some other reason), then that dictionary will ***not*** be listed in the reply area with the available dictionaries.

In the current release of the Linguistic Tools, this function does not allow a code page number to be input. By default, it assumes that the header of a flat file addenda or stopword dictionary will be readable in whatever code page was placed in the service area by the last Linguistic Tools function that accepted an input code page number. If the application calls **NlpFindDicts()** before calling any Linguistic Tools function that accepts an input code page number, by default Linguistic Tools will assume that the code page is 850 on the PC and 500 on the host.

All supported PC (ASCII) code pages listed in Table 17 on page 79 are compatible with each other in this respect, as are all supported host (EBCDIC) code pages listed in this table. On the PC, assuming a default code page of 850, when Linguistic Tools tries to read the header of a Korean flat file addenda in code page 944, it will succeed and identify it as a dictionary that the Linguistic Tools can use. However, it will be unable to read the header of a Korean flat file addenda dictionary in code page 933 (the host code page) and will not identify it as a dictionary that the service can use. On the host, assuming a default code page of 500, the reverse will be true. The flat file addenda in code page 933 will be recognized, but not the one in code page 944.

If the application needs to be able to locate host code page flat file addenda or stopword dictionaries on the PC, or vice versa, it must first call an Linguistic Tools function such as **Spell Verify** with the appropriate input code page before calling **NlpFindDicts()**.

Note: If there is no real word needing to be verified, a dummy word such as “xxx” can be used (it does not even need to be in the correct code page), as long as the results are discarded.

Regardless of whatever code page information may have been left in the service area by a previous call to Linguistic Tools, **NlpFindDicts()** will identify correctly any system dictionary or binary addenda dictionary supplied with or created by this release of Linguistic Tools. Earlier versions (1.7 and earlier) of the system dictionaries may or may not be identified correctly. Binary addenda or stopword dictionaries created by earlier releases of the Linguistic Tools will not be recognized.

For lists of the possible values of the type, content, and language code fields in the reply area, see the Output description for Activate Dictionary. Note that the last field in the LX_AVL_REPLY_P structure contains a variable-length field for the dictionary name. The application must use pointer arithmetic to move to the next entry.

NlpIsDelimiter

Determines whether a character should be treated as a delimiter in the current context.

```
LX_USHORT LX_CALLMODE
NlpIsDelimiter(LX_HNLPSERV hNlpService,
               LX_UINT     PrevChar,
               LX_UINT     CurrChar,
               LX_UINT     NextChar,
               LX_UINT     Reserved,
               LX_PBOOL    pfWordBegin);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

PrevChar Text character that immediately precedes the possible delimiter character.

CurrChar Text character to be tested as a possible delimiter

NextChar Text character that immediately follows the possible delimiter character

Reserved Reserved field; should always be set to zero

pfWordBegin Flag indicating the beginning of a word

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pfWordBegin Flag indicating the beginning of a word

rc Return Code

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_CPG_NOT_SUPPORTED — code page not supported
LX_DELIMITER — target input character is a delimiter
LX_JOIN_CHAR — character should produce a join flag
LX_NOT_DELIMITER — target input character is not a delimiter

Application Programming Guidelines

The characters passed to this function are assumed to be in the same code page as the last function call to Linguistic Tools. The data type of the character parameters is LX_UINT rather than LX_UCHAR to allow for a later extension of this function to operate in a limited manner on double-byte text.

The beginning and ending of text are handled in a special manner. If the application is passing the first character of its text as the **CurrChar** parameter, then the **PrevChar** parameter should equal zero. If the application is passing the last character of its text as the **CurrChar** parameter, then the **LastChar** parameter should equal zero.

The **pfWordBegin** parameter is both an input and an output parameter. If the application is calling **NlpIsDelimiter** repetitively to find the next delimiter character in the text, it should set this field to LX_TRUE when passing the first character of a word in the **CurrChar** parameter. The application should not alter the value of the field at any other time. The returned value should be passed on the next call of the function unless a **LX_DELIMITER** return code is produced. The **pfWordBegin** parameter is required to correctly handle repeated punctuation characters.

Whether or not a given repeated punctuation character is a delimiter depends not only on the surrounding characters, but also on whether that character is the first character of a word. For example, given the text “a...b”, the first period is a delimiter of the first word “a”. However, that same character is not a delimiter when it is classified as the first character of the three character series. The **pfWordBegin** parameter allows the application to indicate this difference to Linguistic Tools.

A return code of LX_JOIN_CHAR indicates that although the current character should be treated as a word delimiter based on the definition in section “Definition of a Word in SBCS Text” on page 80, because the current word may also be part of a larger form, the application may wish to process this character in a special manner. For example, in the form “mother-in-law,” each of the hyphens would produce a LX_JOIN_CHAR return code because, although the characters delimit words, a larger form is also present.

NlpMixedMode

Allows the caller to change or restore the default behavior of the Linguistic Tools 2.7 service as it relates to the processing of mixed case words.

```
LX_USHORT LX_CALLMODE
NlpMixedMode(LX_HNLPSERV  hNlpService,
             LX_BOOL      bMixedMode)
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

bMixedMode Set to LX_TRUE to allow case matching between an all-uppercase input word and a mixed case dictionary word; set to LX_FALSE to restore Linguistic Tools' default behavior (described in "Case Matching between Input Words and Dictionary Words" on page 82).

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

rc Return Code

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle

Application Programming Guidelines

The default mode is to treat mixed case words in the dictionary as special, in that only an exact match with an input word is allowed. For example, the input word &PHD" will not match the dictionary word "PhD", nor would the input word "PCS" match the dictionary word "PCs." Any functions which rely on matching against the dictionary (such as Spell Verify, Synonym Aid, and Morphological Identification) would report that the word was not found. "Case Matching between Input Words and Dictionary Words" on page 82 provides a detailed description of how each Linguistic Tools text-processing function handles different case patterns in input text.

By setting this mode, the default behavior is changed, but only for all-uppercase words. With the mode set, the input word "PHD" will match the dictionary word

“PhD”, and the input word “PCS” will match the dictionary word “PCs”. However, an input word “phd” still would not match the dictionary word “PhD”.

Note that the effect of allowing an input string “PHD” to match the dictionary word “PhD” means only that “PHD” will verify as correctly spelled if the MixedMode flag is set. Spell Aid will continue to return the dictionary case pattern, regardless of the MixedMode flag setting. That is because “PhD” and a few other words like it (such as German “GmbH”) should always appear in mixed case, according to the spelling rules of the language.

Should an application change the default behavior or leave it as it is? That question can only be answered by the application itself, and only after evaluating its use of the Linguistic Tools. If, for instance, the user were sending text that was made up exclusively of book titles, paragraph headings, and other such data, the default behavior might cause problems. All uppercase names are also a potential problem, such as “MACDONALD” and “XYQUEST”. In the latter case, however, you may not want a match to be allowed in order to force the mixed case for “MacDonald” and “XyQuest”.

The application can call this function as often as it wants, but it was designed to be called once, immediately following an Initialize Service call. If you choose to call it more often than this, there are side-effects that you must know about. First, each dictionary maintains a cache of accessed words and data. If the mode is changed, it may not be reflected in the function calls that follow, as data will be pulled from the cache and not from a dictionary match. Second, you may confuse your user as to what a valid word is in any instance of your application. And finally, any index or list built with the default mode would not match an index or list built with the mode set. For these reasons, it may be best to use the function once for each instance of the Linguistic Tools.

NlpQryDelimTable

Returns the current table used to determine word boundaries in SBCS text

```
LX_USHORT LX_CALLMODE
NlpQryDelimTable(LX_HNLPSERV  hNlpService,
                 LX_PUCHAR    pDelimTable,
                 LX_UINT      TableSize);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

pDelimTable Pointer to area owned by the application to hold current delimiter table

TableSize Size (in bytes) of area to hold current delimiter table (This parameter currently should always be 256)

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pDelimTable A copy of the current delimiter table is placed in the buffer indicated by the pDelimTable parameter

rc Return Code

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_ADDRESS — invalid address

Application Programming Guidelines

This function may be used in conjunction with the **NlpSetDelimTable()** function to make minor changes to Linguistic Tools' definition of a word. Linguistic Tools' default behavior is described in "Definition of a Word in SBCS Text" on page 80.

Because **NlpQryDelimTable()** returns the table currently used to determine word boundaries, its output may be modified and used as input to **NlpSetDelimTable()** to make minor changes to the default delimiter table. Figure 12 on page 225 shows how this function may be used with **NlpSetDelimTable()** to change the way in which a hyphen is treated as a delimiter.

NlpQrySearchPath

Returns the current search path used for dictionaries

```
LX_USHORT LX_CALLMODE
NlpQrySearchPath(LX_HNLPSERV  hNlpService,
                 LX_PPUCHAR   ppSearchPath);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

ppSearchPath Pointer to search path

rc return code

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle

Application Programming Guidelines

This function will return the same string passed to **NlpSetSearchPath()**. If the user and/or application has not set a search path on the workstation, this function will return the default path described in "Dictionary Search Path" on page 108. On the host, if the user and/or application has not set a search path, the function will return NULL.

The pointer returned in the **ppSearchPath** parameter is not guaranteed to be valid after another call to Linguistic Tools is made.

NlpRegisterCodePage

Provides Linguistic Tools with the necessary translate table to process text in a given 256-byte code page

```
LX_USHORT LX_CALLMODE
NlpRegisterCodePage(LX_HNLPSERV    hNlpService,
                    LX_CODE_PAGE   CodePage,
                    LX_PUSHORT     pTransTable);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

This is obtained from a call to Linguistic Tools initialization

CodePage Number of code page to associated with the translate table

pTransTable Translate table to be used to convert the application'code points to Linguistic Tools internal format

This should be a table of 256 LX_USHORT values. Each element in the array should contain the Linguistic Tools internal code point for the code point in the code page (see Appendix B, "Internal Character Codes" on page 253 for a list of internal codes). Any code point in the caller's code page that does not have a corresponding character in the Linguistic Tools internal character set should be mapped to zero (0x0000). For example, since the ASCII code point for an upper-case "A" is 64, the 64th element of pTransTable would equal 0x0081 (0x0081 is the Linguistic Tools internal code point for the letter "A").

Note that only a single table is input. Linguistic Tools will generate the internal to external, DAP external to internal, and simple token tables based on the one table. Therefore, **one external character must not map to the same internal character as another external character**.

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

rc Return code

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed

LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle

Application Programming Guidelines

The application may register more than one code page by making repeated calls to this function. The user indicates to Linguistic Tools the code page of the input text via the lx_num_cpg field of the control-block. If Linguistic Tools receives a value in the lx_num_cpg field that it does not recognize (that is, the code page is not one of the default supported code pages or has not been successfully registered with **NlpRegisterCodePage()**), Linguistic Tools will return a return code of LX_CPG_NOT_SUPPORTED to the caller. To have Linguistic Tools accept input text in an additional code page, the caller must call **NlpRegisterCodePage()** before passing the input text to Linguistic Tools

The application may free space pointed at by the **pTransTable** parameter after completion of this call. One KB of internal Linguistic Tools data area is required for each code page that is registered.

Example

```
/* Sample table for fictitious code page 999               */
/* In this table, letters A-Z are at positions 0x41 to 0x5A. */
/* Letters a-z are at positions 0x61 to 0x7A.              */
LX_USHORT p999Table[] =
{
/*     -0   -1   -2   -3   -4   -5   -6   -7   -8   -9   -A   -B   -C   -D   -E   -F  */
/*0-*/0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
/*1-*/0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
/*2-*/0xBF,0xFB,0x7F,0x71,0x72,0x73,0x80,0x1B,0x4B,0x9C,0x76,0x4D,0x9B,0x1C,0x3F,0xBE,
/*3-*/0x40,0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4F,0x50,0x4A,0xED,0x4C,0x75,
/*4-*/0x00,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8A,0x8B,0x8C,0x8D,0x8E,0x8F,
/*5-*/0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9A,0x74,0x00,0x4E,0x00,0x00,
/*6-*/0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,
/*7-*/0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x00,0x00,0x00,0xE3,0x00,
/*8-*/0xB3,0x31,0x1E,0x28,0x2D,0x23,0x37,0x33,0x29,0x2E,0x24,0x2F,0x2A,0x25,0xAD,0xB7,
/*9-*/0x9E,0x38,0xB8,0x2B,0x30,0x26,0x2C,0x27,0x32,0xB0,0xB1,0x3B,0x00,0xBB,0x00,0x00,
/*A-*/0x1D,0x1F,0x20,0x21,0x35,0xB5,0x00,0x00,0xBA,0x00,0x00,0x00,0x00,0xB9,0x00,0x00,
/*B-*/0x00,0x00,0x00,0x00,0x00,0x9D,0xA8,0xA3,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
/*C-*/0x00,0x00,0x00,0x00,0x00,0x00,0x34,0xB4,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
/*D-*/0x3C,0xBC,0xA9,0xAE,0xA4,0x39,0x9F,0xAA,0xAF,0x00,0x00,0x00,0x00,0x00,0xA5,0x00,
/*E-*/0xA0,0x3A,0xAB,0xA6,0x36,0xB6,0x00,0x3D,0xBD,0xA1,0xAC,0xA7,0x22,0xA2,0x00,0x00,
/*F-*/0x70,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x3E,0x00,0x00,0x00,0x00,0x00
};

LX_CB_P   CB_p;      /* Control block */
LX_USHORT rc;


/* Register code page 999 with the NLP Service */
rc = NlpRegisterCodePage(CB_p->lx_serv_area_p, 999, p999Table);

if (rc == LX_RC_OK)
  {
  /* Set up control block field to call NLP Service */
  CB_p->lx_func_code   = LX_SPELL_VERIFY;
  CB_p->lx_elements_p  = PtrToUsersElements;
  CB_p->lx_elements_ct = UsersElementsCt;
  CB_p->lx_num_cpg = 999;

  /* Call the NLP Service */
  NlpEntry(CB_p);
  }
```

NlpSetDelimTable

Changes the table used to determine word boundaries in SBCS text

```
LX_USHORT LX_CALLMODE
NlpSetDelimTable(LX_HNLPSERV  hNlpService,
                 LX_PUCHAR    pDelimTable,
                 LX_UINT      TableSize)
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

pDelimTable Pointer to table describing character delimiters

TableSize Size (in bytes) of the input current delimiter table (This parameter currently should always be 256)

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

rc Return Code

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_ADDRESS — invalid address

Application Programming Guidelines

This function is useful for allowing the application to change Linguistic Tools' definition of a word (described in "Definition of a Word in SBCS Text" on page 80). Because the delimiter table indicates which characters act as word delimiters, changing the default table will also change Linguistic Tools' default definition of a word.

You should use great caution when changing the default delimiter table. Many Linguistic Tools functions make basic assumptions concerning the default definition of a word. Only functions whose primary purpose is to determine word boundaries (that is, Data Element Creation and Word Isolation) should be called after changing the default delimiter table. The output of all other high-level block-input functions (Block spell verify, lexical analysis, extract key word, and so on) will be undefined if the default delimiter table is not used.

Changing the default delimiter table is useful when an application wishes Linguistic Tools to perform word isolation differently than it would by the Linguistic Tools default delimiter table. For example, assume that an application wishes to count the number of words in the input text, but it wants hyphenated forms to be counted as one word. Because this requires the hyphen character to be treated differently as a delimiter than Linguistic Tools' default behavior, the default delimiter table must be modified.

The delimiter table specifies how characters should be treated as word delimiters. The table applies only to the first 256 characters in a given code page. The table is indexed by the Linguistic Tools internal code point for each character. The classification for each character must be one of the following constants:

LX_LOWERCASE	Lower case alphabetic
LX_UPPERCASE	Upper case alphabetic
LX_NUMERIC	Numeric digit
LX_DELIM	Punctuation character always treated as a delimiter
LX_TRAILCHAR	Trailing character delimiter (that is, space)
LX_STKNECLASS	Punctuation characters not treated as a delimiter when surrounded by alphabetics
LX_STKNFCLASS	Punctuation characters not treated as a delimiter when surrounded by numerics
LX_STKNGCLASS	Punctuation characters not treated as a delimiter when preceded by a punctuation and followed by a numeric
LX_STKNHCLASS	Punctuation characters not treated as a delimiter when immediately preceded by an identical character
LX_STKNICLASS	Punctuation characters not treated as a delimiter when surrounded by numerics or if preceded by an identical character
LX_STKNKCLASS	Punctuation characters not treated as a delimiter when surrounded by numerics; or preceded by a punctuation and followed by a numeric; or if preceded by an identical character
LX_STKNDBBEG	A character that marks the beginning of a double-byte character
LX_STKNSHIFT	Host shift-out or shift-in character

An application would change a character's behavior as a delimiter by modifying the position in the delimiter table indicated by that character's Linguistic Tools internal code. For example, because the Linguistic Tools internal code point for the hyphen character is 0x1C, changing the value contained at offset 0x1C would change how a hyphen character acts as a delimiter. (For a list of internal Linguistic Tools characters, see Appendix B, "Internal Character Codes" on page 253.)

When an override table is input via **NlpSetDelimTable()**, Linguistic Tools will allocate an internal buffer to hold a copy of the application's table. Thus, the application does not need to maintain an area to hold the delimiter table.

Example

```
LX_HNLPSERV hNlpService;      /* NLP Service handle */
LX_UCHAR    DelimTable[256];  /* Working Delimiter table */
LX_USHORT   rc;               /* Return Code */

rc = NlpQryDelimTable(hNlpService, DelimTable, 256);

if (rc == LX_RC_OK)
  {
  DelimTable[0x1C]  = LX_STKNJCLASS;     /* Change hyphen delimiter */
                                         /* classification          */
  rc = NlpSetDelimTable(hNlpService, DelimTable, 256);
  }
```

Figure 12. Sample code to change treatment of a hyphen as a word delimiter

NlpSetDictDef

Provides Linguistic Tools with the application's default extensions/file types/qualifiers for different types of dictionary file names. This permits an application to substitute its file naming conventions for the Linguistic Tools defaults listed below, without hindering the **NlpFindDicts()** function's ability to identify different types of dictionaries.

```
LX_USHORT LX_CALLMODE
NlpSetDictDef(LX_HNLPSERV       hNlpService,
              LX_PDCT_EXT       pDictExts);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

pDictExts pointer to a structure of type LX_DCT_EXT, containing pointers to dictionary extensions/file types/qualifiers.

```
pD15Dict    – system dictionary v1.5
pD17Dict    – system dictionary v1.7
pLegDict    – system legal dictionary
pMedDict    – system medical dictionary
pCptDict    – system computing terms dictionary
pBinAddDict – binary addenda dictionary
pFltAddDict – flat-file addenda dictionary
pBinStwDict – binary stopword addenda
pFltStwDict – flat-file stopword addenda
pBinAbbDict – binary abbreviation addenda
pFltAbbDict – flat-file abbreviation addenda
```

The dictionary extensions should be given as null-terminated strings, not exceeding LX_EXT_LEN in length.

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

rc return code

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

- LX_BAD_FUNCT_CODE — invalid function code
- LX_BAD_RQST_TYPE — invalid request type
- LX_RC_OK — processing successfully completed
- LX_UNEXPECTED_RC — unexpected or unknown condition
- LX_BAD_SERV_HANDLE — invalid Service Area Handle
- LX_BAD_PARM — missing or illegal input value for parameter

Application Programming Guidelines

Typically an application wishing to use extensions/file types/qualifiers other than the Linguistic Tools defaults will call this function once immediately after initializing the service. The new default extensions/file types/qualifiers supplied by the application will be remembered and used from that point until Linguistic Tools is terminated. An application may place a NULL character in any field for which the Linguistic Tools default should be maintained.

"File Names" on page 106 gives more information about Linguistic Tools file naming conventions.

NlpSetSearchPath

Changes the locations to be used in searching for dictionaries.

```
LX_USHORT LX_CALLMODE
NlpSetSearchPath(LX_HNLPSERV  hNlpService,
                 LX_PUCHAR    SearchPath);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Handle Area returned from the **Initialize** function

SearchPath String describing search path for dictionaries

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

- none

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle

Application Programming Guidelines

A search path may be used to find dictionaries. If the application does not specify a customized path, Linguistic Tools will use the default search path described in “Dictionary Search Path” on page 108.

“Dictionary Search Path” on page 108 gives more information about specifying a search path.

The processing of **Activate Dictionary** and **NlpFindDicts()** are both affected by the search path. An application may improve the performance of these functions by setting the search path to include only those locations which actually contain dictionaries. One way to do this is to call **NlpFindDicts()** to filter out of the search path those locations not containing dictionaries and then to send the shortened path to **NlpSetSearchPath()**.

Chapter 17. Token List Utility Functions

NlpGetBeginType

Returns the character-type of the first character associated with a token.

```
LX_USHORT LX_CALLMODE
NlpGetBeginType(LX_HNLPSERV  hNlpService,
                LX_HTOKEN    hToken,
                LX_PUINT     pBeginType)
```

Input

The address of the Linguistic Tools control block is the only parameter passed to Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle of the current token

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pBeginType Type of the first character in the input

LX_UPPERCASE
LX_LOWERCASE
LX_NUMERIC
LX_PUNC
LX_STKNDCLASS — DBCS

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token

© Copyright IBM Corp. 2000

Application Programming Guidelines

The output parameter *pBeginType* contains a bit-mask that describes the types of text characters associated with a token. This value may be evaluated against the constants LX_UPPERCASE, LX_LOWERCASE, LX_NUMERIC, and LX_PUNC to determine whether the first character of the token is uppercase, lowercase, numeric, or punctuation.

DBCS kanji text is indicated by a type of LX_STKNDCLASS. The shift-out and shift-in control characters are ignored by this function.

NlpGetComponent

Returns the handle of a token's component.

```
LX_USHORT LX_CALLMODE
NlpGetComponent(LX_HNLPSERV  hNlpService,
                LX_HTOKEN    hToken,
                LX_UINT      NumComponent,
                LX_PHTOKEN   phComponent);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle of the current token

NumComponent Desired component number

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

phComponent Handle of the token's component

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token
LX_NO_COMP - Token component not found

Application Programming Guidelines

This function provides the application with the token handles necessary to obtain detailed information about the components of a joined token. For basic (non-joined) tokens, the function will return the same token handle that was input. Before calling this function, the application should have called **NlpGetNumComponent()** to find out how many components are associated with a given token.

To obtain the first component of a joined token a value of one should be passed in the *NumComponent* parameter.

NlpGetContent

Returns a description of a token's contents.

```
LX_USHORT LX_CALLMODE
NlpGetContent(LX_HNLPSERV      hNlpService,
              LX_HTOKEN        hToken,
              LX_PTOKCONTENT   pContent);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle of a token

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pContent Bit mask indicating types of characters contained in token string

LX_UPPERCASE
LX_LOWERCASE
LX_NUMERIC
LX_PUNC

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token
LX_NO_MORE_TOKENS

Application Programming Guidelines

The output parameter *pContent* contains a bit-mask that describes the types of text characters associated with a token. This value may be evaluated against the constants LX_UPPERCASE, LX_LOWERCASE, LX_NUMERIC, and LX_PUNC to determine if text contains upper-case, lower-case, numeric and/or punctuation characters, respectively.

When applied against a token from a token list generated via the Linguistic Tools Extract Keyword or Extract Query Terms functions, the returned content is based on the *normalized* form of that token.

This function is undefined for DBCS text.

NlpGetFirstProperty

Returns the first property associated with a token.

```
LX_USHORT LX_CALLMODE
NlpGetFirstProperty(LX_HNLPSERV hNlpService,
                    LX_HTOKEN   hToken,
                    LX_PROPID   RqstPropID,
                    LX_PHPROP   phFirstProp);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle to token

RqstPropID Property identifier for a specific property

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

phFirstProp Handle of the first property associated with a token

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token
LX_END_OF_PROPS — end of property list reached

Application Programming Guidelines

This function will get only the first property associated with a token. In order to get other properties the application should call **NlpGetNextProperty()**. Linguistic Tools properties are defined in the include file EFZLEXPR.H shipped with the Linguistic Tools.

The application may use the *RqstPropID* parameter to request the first property with a specific identifier. If the *RqstPropID* parameter is set to the constant LX_ANYPROP, the first property will be returned to the caller, regardless of its identifier. If the caller places any other value in this parameter, Linguistic Tools will search all of the properties associated with the token to find a property whose

identifier matches the input parameter. **NlpGetFirstProperty()** will return LX_END_OF_PROPS if no properties match the *RqstPropID* parameter.

To obtain the value of a property, the application should call **NlpGetPval()** using the property handle returned by **NlpGetFirstProperty()**.

NlpGetNextProperty

Returns a handle to the property that follows the current property.

```
LX_USHORT LX_CALLMODE
NlpGetNextProperty(LX_HNLPSERV hNlpService,
                   LX_HPROP    hProperty,
                   LX_PROPID   RqstPropID,
                   LX_PHPROP   phNextProp);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hProperty Handle to a token's property

RqstPropID Property identifier for a specific property

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

phNextProp Handle of the property that follows hProperty

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_PROP_HANDLE
LX_END_OF_PROPS — end of property list reached

Application Programming Guidelines

To get only the first property associated with a token, the application should call **NlpGetFirstProperty()**. Linguistic Tools properties are defined in the include file EFZLEXPR.H shipped with the Linguistic Tools.

As with **NlpGetFirstProperty()** the application may use the *RqstPropID* parameter to request a property with a specific identifier. If the *RqstPropID* parameter is set to the constant LX_ANYPROP, the next property associated with the token will be returned to the caller, regardless of its identifier. If the caller places any other value in this parameter, Linguistic Tools will search all properties following the one pointed to by *hProperty* to find one whose identifier matches the input parameter.

NlpGetNextProperty() will return LX_END_OF_PROPS if no properties follow the one pointed to by *hProperty* that match the *RqstPropID* parameter.

To obtain the value of a property, the application should call **NlpGetPval()** using the property handle returned by **NlpGetFirstProperty()**.

NlpGetNextToken

Returns next token handle present in the token list.

```
LX_USHORT LX_CALLMODE
NlpGetNextToken(LX_HNLPSERV  hNlpService,
                LX_HTOKEN    hToken,
                LX_PHTOKEN   phNextTok);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle of the current token

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

phNextTok Handle of next token in token list

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token
LX_END_OF_INPUT — end of input reached

NlpGetNumComponent

Returns the number of components in a joined token.

```
LX_USHORT LX_CALLMODE
NlpGetNumComponent(LX_HNLPSERV  hNlpService,
                   LX_HTOKEN    hToken,
                   LX_PUINT     pNumComponent);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle of a token

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pNumComponent Number of components in a joined token

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token

Application Programming Guidelines

If this token is not joined, a value of zero is returned in the *pNumComponent* parameter.

NlpGetNumTrail

Returns the number of bytes of basic delimiters that trail a given token.

```
LX_USHORT LX_CALLMODE
NlpGetNumTrail(LX_HNLPSERV hNlpService,
               LX_HTOKEN   hToken,
               LX_PUSHORT  pNumTrail);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle of a token

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pNumTrail The number of bytes of basic delimiters that follow a given token

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token

Application Programming Guidelines

By default the basic delimiter is defined as a blank. Therefore, if a token in the input text is immediately followed by a non-blank delimiter (e.g. a comma), the returned number is zero, regardless of whether that non-blank delimiter is followed by any blanks.

NlpGetPrevToken

Returns the handle of previous token in the token list.

```
LX_USHORT LX_CALLMODE
NlpGetPrevToken(LX_HNLPSERV hNlpService,
                LX_HTOKEN   hToken,
                LX_PTOKEN   phPrevToken);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Service Area Handle return from the **Initialize** function

hToken Handle of a token

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

phPrevToken Handle of the previous token

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token
LX_END_OF_INPUT — end of input reached

NlpGetPval

Returns the value of a property.

```
LX_USHORT LX_CALLMODE
NlpGetPval(LX_HNLPSERV hNlpService,
           LX_HPROP    hProperty,
           LX_PPROPID  pPropID,
           LX_PPVOID   pPval);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returneded from the **Initialize** function

hProperty Handle of a property

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pPropID Property identifier of the requested property

pPval Value of the requested property (NULL if the property had no value)

rc Return code

hNlpService Linguistic Tools Service Area Handle returneded from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_PROPERTY
LX_BAD_SERV_HANDLE — invalid Service Area Handle input

Application Programming Guidelines

Linguistic Tools properties are defined in the file EFZLEXPR.H, which is included under EFZLNLPS.H.

NlpGetString

Places the string associated with a token in a buffer.

```
LX_USHORT LX_CALLMODE
NlpGetString(LX_HNLPSERV  hNlpService,
             LX_HTOKEN    hToken,
             LX_PUCHAR    Buffer,
             LX_USHORT    BufferLen,
             LX_TOKTYPE   TokenType);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle of a token

Buffer Pointer to string destination buffer

BufferLen Length of string destination buffer

TokenType Type of string that should be output

LX_TEXT Only normal text characters (no control characters)

LX_ALL_TEXT All text associated with this token (include control characters)

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

This function places the string associated with a token into the buffer designated by the *Buffer* parameter.

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_ELE_TYPE — invalid element type
LX_BAD_TOKEN — invalid pointer to a token
LX_BUFFER_2_SMALL - String did not fit in buffer
LX_STRING_BUFFER - String buffer needs to be allocated

Application Programming Guidelines

This function will return single-byte, double-byte, or mixed strings. Any leading, medial, or trailing blanks in the token or its components will not be returned.

The TokenType parameter typically will be set to LX_TEXT. If a type of LX_TEXT is requested, any double-byte host shift-in or shift-out characters will be included in the output buffer.

The application should call **NlpGetTokenLen()** to determine how large the output buffer should be. If the length of the output buffer passed to this routine is larger than necessary, this routine will place a NULL character delimiter at the end of the string.

When using this utility to obtain strings for tokens that may be pointing to host (EBCDIC) double-byte text, the application should add an additional two bytes to the input buffer. These additional bytes will allow Linguistic Tools to insert host shift-in and shift-out characters at the beginning and/or end of the string, when appropriate. When the text represented by a token is in a host double-byte code page, Linguistic Tools will add a shift-out character at the beginning of the string if the string begins with a double-byte character. Likewise, a shift-in character will be added to the end of the buffer if the string ends with a double-byte character (This processing is in addition to including any shift-in or shift-out characters contained within the token's string). Note that if the application provides a buffer that is always three bytes greater (one each for shift-out and shift-in in host DBCS text, plus one for an optional null at the end of a null-terminated string on any platform) than the value returned from **NlpGetTokenLen()**, the application may call strlen() to determine how many extra shift-in or shift-out characters (if any) have been added to a token's string.

NlpGetStringAddr

Returns the location of a token's string in terms of the input text.

```
LX_USHORT LX_CALLMODE
NlpGetStringAddr(LX_HNLPSERV hNlpService,
                 LX_HTOKEN   hToken,
                 LX_PPUCHAR  ppString);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle of a token

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

ppString Pointer to the portion of the input associated with the token.

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_SERV_HANDLE — invalid Service Area Handle input
LX_BAD_TOKEN — invalid pointer to a token

NlpGetStringFlag

Returns string flags of a token, indicating

- whether characters in the string part of the token are represented in single- or double-byte format
- whether the token is basic or joined

```
LX_USHORT LX_CALLMODE
NlpGetStringFlag(LX_HNLPSERV hNlpService,
                 LX_HTOKEN   hToken,
                 LX_PSTRFLAG pStrFlag);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle of a token

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pStrFlag Bit-mask containing encoded information about the token's string

LX_JOIN - joined token
LX_DBCS - token string contains DBCS characters

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token

Application Programming Guidelines

The caller should perform a bitwise AND using the constants LX_JOIN and LX_DBCS with the value returned in the *pStrFlag* parameter to determine string information.

NlpGetTokenLen

Returns the length of the string associated with a token

```
LX_USHORT LX_CALLMODE
NlpGetTokenLen(LX_HNLPSERV hNlpService,
               LX_HTOKEN   hToken,
               LX_TOKTYPE  TokenType,
               LX_PUSHORT  pTokenLen);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService
: Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken
: Handle of a token

TokenType
: Type of text desired

  LX_TEXT Only normal text characters (for host DBCS text this includes shift-out and shift-in characters)

  LX_ALL_TEXT All text associated with this token (including non-text elements, such as user data, as well as host DBCS shift-out/shift-in)

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pTokenLen
: Length of the string associated with a token (not including trailing characters)

rc
: Return code

hNlpService
: Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token

Application Programming Guidelines

This utility will return the length, in bytes, of the text represented by the token. The application may use this value to determine the buffer size required by **NlpGetString()**. If the application is dealing with host double-byte text, it should add two bytes to the value returned by this utility to allow for the insertion of shift-in and/or shift-out characters (see “NlpGetString” on page 243).

For a token with components which contain trailing blanks, the length of the string does not include any such blanks, regardless of their position within the (possibly joined) token. The string returned by **NlpGetString()** will not contain any blanks. This consideration applies in the case where a hyphenated portion of a word is followed by a new-line, blanks, and then the rest of the word. Only the length of the non-blank characters will be returned, since only the non-blank characters will be returned by **NlpGetString()**.

The *TokenType* parameter is ignored when the input token is not a joined token. Otherwise, this parameter indicates that only components of the specified type should be included in the length returned. If the *TokenType* parameter equals the constant LX_ALL_TEXT, the length of all components is returned. If a type of LX_TEXT is requested, any double-byte host shift-in or shift-out characters will be included in the output length, but no user data or other non-text elements.

NlpGetTokenType

Returns the type of a given token

```
LX_USHORT LX_CALLMODE
NlpGetTokenType(LX_HNLPSERV hNlpService,
                LX_HTOKEN   hToken,
                LX_PTOKTYPE pTokenType);
```

Input

The address of the Linguistic Tools control block is the only parameter passed to the Linguistic Tools. The following input fields are used:

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

hToken Handle of a token

Output

The updated Linguistic Tools control block. The following output fields, or the areas that they point to, are updated.

pTokenType Type of token

LX_NLIN — new-line character
LX_NSEN — new-sentence marker
LX_PARA — new-paragraph marker
LX_SOSI — shift-out/shift-in character
LX_TEXT — text
LX_UDAT — user data
LX_IGNORE— user data

rc Return code

hNlpService Linguistic Tools Service Area Handle returned from the **Initialize** function

Return Codes

A return code that indicates the outcome of processing is passed back to the application in *lx_rc* field of the Linguistic Tools control block. The return code can have the following values:

LX_BAD_FUNCT_CODE — invalid function code
LX_BAD_RQST_TYPE — invalid request type
LX_RC_OK — processing successfully completed
LX_UNEXPECTED_RC — unexpected or unknown condition
LX_BAD_SERV_HANDLE — invalid Service Area Handle
LX_BAD_TOKEN — invalid pointer to a token

Application Programming Guidelines

If the input token is a joined token, this function returns the type of the first component of the token.

Appendix A. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

© Copyright IBM Corp. 2000

Trademarks

The following are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX	OS/390
IBM	OS/400
the IBM logo	SUN
OS/2	UNIX

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

Appendix B. Internal Character Codes

To provide consistency for applications which may assume one of many possible codes for alphanumeric characters (EBCDIC, ASCII, and so forth), the words in this dictionary, when decoded, are assigned internal codes used only in Linguistic Tools. This internal coding scheme allows Linguistic Tools to have the same programs process the dictionaries regardless of system character codes. The system character codes are translated to and from the Linguistic Tools internal codes as needed.

Constant Codes

A large number of internal codes are constant for all languages and alphabets. These codes follow:

Table 23 (Page 1 of 4). Linguistic Tools Internal Codes: Constant Codes

Internal Code	Char.	Description	Internal Code	Char.	Description
x'00'		word boundary	x'80'	&	ampersand
x'01'	a	Lower case a	x'81'	A	Upper case A
x'02'	b	Lower case b	x'82'	B	Upper case B
x'03'	c	Lower case c	x'83'	C	Upper case C
x'04'	d	Lower case d	x'84'	D	Upper case D
x'05'	e	Lower case e	x'85'	E	Upper case E
x'06'	f	Lower case f	x'86'	F	Upper case F
x'07'	g	Lower case g	x'87'	G	Upper case G
x'08'	h	Lower case h	x'88'	H	Upper case H
x'09'	i	Lower case i	x'89'	I	Upper case I
x'0A'	j	Lower case j	x'8A'	J	Upper case J
x'0B'	k	Lower case k	x'8B'	K	Upper case K
x'0C'	l	Lower case l	x'8C'	L	Upper case L
x'0D'	m	Lower case m	x'8D'	M	Upper case M
x'0E'	n	Lower case n	x'8E'	N	Upper case N
x'0F'	o	Lower case o	x'8F'	O	Upper case O
x'10'	p	Lower case p	x'90'	P	Upper case P
x'11'	q	Lower case q	x'91'	Q	Upper case Q
x'12'	r	Lower case r	x'92'	R	Upper case R
x'13'	s	Lower case s	x'93'	S	Upper case S
x'14'	t	Lower case t	x'94'	T	Upper case T

© Copyright IBM Corp. 2000

Table 23 (Page 2 of 4). Linguistic Tools Internal Codes: Constant Codes

Internal Code	Char.	Description	Internal Code	Char.	Description
x'15'	u	Lower case u	x'95'	U	Upper case U
x'16'	v	Lower case v	x'96'	V	Upper case V
x'17'	w	Lower case w	x'97'	W	Upper case W
x'18'	x	Lower case x	x'98'	X	Upper case X
x'19'	y	Lower case y	x'99'	Y	Upper case Y
x'1A'	z	Lower case z	x'9A'	Z	Upper case Z
x'1B'	'	apostrophe	x'9B'	,	comma
x'1C'	-	hyphen	x'9C'	)	right parenthesis
x'1D'	á	a-ACUTE	x'9D'	Á	A-ACUTE
x'1E'	é	e-ACUTE	x'9E'	É	E-ACUTE
x'1F'	í	i-ACUTE	x'9F'	Í	I-ACUTE
x'20'	ó	o-ACUTE	x'A0'	Ó	O-ACUTE
x'21'	ú	u-ACUTE	x'A1'	Ú	U-ACUTE
x'22'	ý	y-ACUTE	x'A2'	Ý	Y-ACUTE
x'23'	à	a-GRAVE	x'A3'	À	A-GRAVE
x'24'	è	e-GRAVE	x'A4'	È	E-GRAVE
x'25'	ì	i-GRAVE	x'A5'	Ì	I-GRAVE
x'26'	ò	o-GRAVE	x'A6'	Ò	O-GRAVE
x'27'	ù	u-GRAVE	x'A7'	Ù	U-GRAVE
x'28'	â	a-CIRCUMFLEX	x'A8'	Â	A-CIRCUMFLEX
x'29'	ê	e-CIRCUMFLEX	x'A9'	Ê	E-CIRCUMFLEX
x'2A'	î	i-CIRCUMFLEX	x'AA'	Î	I-CIRCUMFLEX
x'2B'	ô	o-CIRCUMFLEX	x'AB'	Ô	O-CIRCUMFLEX
x'2C'	û	u-CIRCUMFLEX	x'AC'	Û	U-CIRCUMFLEX
x'2D'	ä	a-DIAERESIS	x'AD'	Ä	A-DIAERESIS
x'2E'	ë	e-DIAERESIS	x'AE'	Ë	E-DIAERESIS
x'2F'	ï	i-DIAERESIS	x'AF'	Ï	I-DIAERESIS
x'30'	ö	o-DIAERESIS	x'B0'	Ö	O-DIAERESIS
x'31'	ü	u-DIAERESIS	x'B1'	Ü	U-DIAERESIS
x'32'	ÿ	y-DIAERESIS	x'B2'	Ÿ	Y-DIAERESIS
x'33'	ç	c-CEDILLA	x'B3'	Ç	C-CEDILLA

Table 23 (Page 3 of 4). Linguistic Tools Internal Codes: Constant Codes

Internal Code	Char.	Description	Internal Code	Char.	Description
x'34'	ã	a-TILDE	x'B4'	Ã	A-TILDE
x'35'	ñ	n-TILDE	x'B5'	Ñ	N-TILDE
x'36'	õ	o-TILDE	x'B6'	Õ	O-TILDE
x'37'	å	a-OVERCIRCLE	x'B7'	Å	A-OVERCIRCLE
x'38'	æ	ae-DIPHTHONG	x'B8'	Æ	AE-DIPHTHONG
x'39'	ı	i-DOTLESS	x'B9'	¡	upside-down exclamation
x'3A'	ß	Eszett	x'BA'	¿	upside-down question
x'3B'	ø	o-SLASH	x'BB'	Ø	O-SLASH
x'3C'	ð	eth/d-ICELANDIC	x'BC'	Đ	D-STROKE
x'3D'	þ	thorn-SMALL	x'BD'	Þ	THORN-CAPITAL
x'3E'	·	high-dot	x'BE'	/	slash right
x'3F'	.	period	x'BF'		blank
x'40'	0	zero	x'C0'		language-specific numeric
x'41'	1	one	x'C1'		language-specific numeric
x'42'	2	two	x'C2'		language-specific numeric
x'43'	3	three	x'C3'		language-specific numeric
x'44'	4	four	x'C4'		language-specific numeric
x'45'	5	five	x'C5'		language-specific numeric
x'46'	6	six	x'C6'		language-specific numeric
x'47'	7	seven	x'C7'		language-specific numeric
x'48'	8	eight	x'C8'		language-specific numeric
x'49'	9	nine	x'C9'		language-specific numeric
x'4A'	<	less-than sign	x'CA'		language-specific comma
x'4B'	(	left-parenthesis	x'CB'		Reserved
x'4C'	>	greater-than sign	x'CC'	\	slash left
x'4D'	+	plus sign	x'CD'	˜	tilde
x'4E'	]	right bracket	x'CE'	=	equals
x'4F'	:	colon	x'CF'	!	exclamation point
x'50'	;	semicolon	x'D0'		language-specific semicolon
x'51'-x'6F'		lower case language-specific	x'D1'-x'EF'		upper case language-specific
x'70'		syllable hyphen	x'F0'		language-specific

Table 23 (Page 4 of 4). Linguistic Tools Internal Codes: Constant Codes

Internal Code	Char.	Description	Internal Code	Char.	Description
x'71'	#	pound sign	x'F1'		language-specific
x'72'	$	dollar sign	x'F2'		language-specific
x'73'	%	percent sign	x'F3'		reserved
x'74'	[	left bracket	x'F4'		reserved
x'75'	?	question mark	x'F5'		language-specific question mark
x'76'	*	asterisk	x'F6'		reserved
x'77'		at sign	x'F7'		reserved
x'78'-x'7C'		lower case language-specific	x'F8'-x'FC'		upper case language-specific
x'7D'		language-specific	x'FD'		escape character
x'7E'		reserved	x'FE'		wild card character 1
x'7F'	"	quote	x'FF'		wild card character 2

An input word will be converted to this internal form before accessing the dictionary or any of the functions which use the dictionary. Similarly, any words returned from the dictionary will be converted back from this internal form. Except for characters that have no corresponding upper or lower case, the code for the upper case differs by X'80' from the lower case character.

Greek-Specific Codes

The Greek alphabet contains characters which are not found in other languages' alphabets. The character set used also allows Roman characters. Therefore, it is required that the general internal Linguistic Tools codes be used along with Greek-specific internal codes. These Greek-specific codes will represent different characters or will simply not be used in different language dictionaries.

On the workstation, code page 869 is supported for Greek. On the host, code page 875 is supported for Greek. In order to assist the application in identifying the character associated with each of the Linguistic Tools internal code points for Greek, the following table relates the Greek characters to internal Linguistic Tools codes, code page 869 codes, and code page 875 codes:

Table 24 (Page 1 of 2). Linguistic Tools Internal Codes: Workstation and Host Greek Codes

Internal Code	Char.	Description	CP869 Code	CP875 Code	Internal Code	Char.	Description	CP869 Code	CP875 Code
x'51'	α	alpha	x'D6'	x'8A'	x'D1'	Α	ALPHA	x'A4'	x'41'
x'52'	β	beta	x'D7'	x'8B'	x'D2'	Β	BETA	x'A5'	x'42'
x'53'	γ	gamma	x'D8'	x'8C'	x'D3'	Γ	GAMMA	x'A6'	x'43'
x'54'	δ	delta	x'DD'	x'8D'	x'D4'	Δ	DELTA	x'A7'	x'44'
x'55'	ε	epsilon	x'DE'	x'8E'	x'D5'	Ε	EPSILON	x'A8'	x'45'
x'56'	ζ	zeta	x'E0'	x'8F'	x'D6'	Ζ	ZETA	x'A9'	x'46'
x'57'	η	eta	x'E1'	x'9A'	x'D7'	Η	ETA	x'AA'	x'47'
x'58'	θ	theta	x'E2'	x'9B'	x'D8'	Θ	THETA	x'AC'	x'48'
x'59'	ι	iota	x'E3'	x'9C'	x'D9'	Ι	IOTA	x'AD'	x'49'
x'5A'	κ	kappa	x'E4'	x'9D'	x'DA'	Κ	KAPPA	x'B5'	x'51'
x'5B'	λ	lambda	x'E5'	x'9E'	x'DB'	Λ	LAMBDA	x'B6'	x'52'
x'5C'	μ	mu	x'E6'	x'9F'	x'DC'	Μ	MU	x'B7'	x'53'
x'5D'	ν	nu	x'E7'	x'AA'	x'DD'	Ν	NU	x'B8'	x'54'
x'5E'	ξ	xi	x'E8'	x'AB'	x'DE'	Ξ	XI	x'BD'	x'55'
x'5F'	ο	omicron	x'E9'	x'AC'	x'DF'	Ο	OMICRON	x'BE'	x'56'
x'60'	π	pi	x'EA'	x'AD'	x'E0'	Π	PI	x'C6'	x'57'
x'61'	ρ	rho	x'EB'	x'AE'	x'E1'	Ρ	RHO	x'C7'	x'58'
x'62'	σ	sigma	x'EC'	x'AF'	x'E2'	Σ	SIGMA	x'CF'	x'59'
x'63'	ς	sigma final	x'ED'	x'BA'	x'E3'		reserved		
x'64'	τ	tau	x'EE'	x'BB'	x'E4'	Τ	TAU	x'D0'	x'62'
x'65'	υ	upsilon	x'F2'	x'BC'	x'E5'	ϒ	UPSILON	x'D1'	x'63'
x'66'	φ	phi	x'F3'	x'BD'	x'E6'	Φ	PHI	x'D2'	x'64'

Table 24 (Page 2 of 2). Linguistic Tools Internal Codes: Workstation and Host Greek Codes

Internal Code	Char.	Description	CP869 Code	CP875 Code	Internal Code	Char.	Description	CP869 Code	CP875 Code
x'67'	χ	chi	x'F4'	x'BE'	x'E7'	Χ	CHI	x'D3'	x'65'
x'68'	ψ	psi	x'F6'	x'BF'	x'E8'	Ψ	PSI	x'D4'	x'66'
x'69'	ω	omega	x'FA'	x'CB'	x'E9'	Ω	OMEGA	x'D5'	x'67'
x'6A'		alpha acute	x'9B'	x'B1'	x'EA'		ALPHA ACUTE	x'86'	x'71'
x'6B'		epsilon acute	x'9D'	x'B2'	x'EB'		EPSILON ACUTE	x'8D'	x'72'
x'6C'		eta acute	x'9E'	x'B3'	x'EC'		ETA ACUTE	x'8F'	x'73'
x'6D'		iota diaer.	x'A0'	x'B4'	x'ED'		IOTA DIAER.	x'91'	x'68'
x'6E'		iota acute	x'9F'	x'B5'	x'EE'		IOTA ACUTE	x'90'	x'75'
x'6F'		omicron acute	x'A2'	x'B6'	x'EF'		OMICRON ACUTE	x'92'	x'76'
x'78'		iota diaer. acute		x'CC'	x'F8'		IOTA DIAER. ACUTE		
x'79'		upsilon diaer. acute		x'CD'	x'F9'		UPSILON DIAER. ACUTE		
x'7A'		upsilon acute	x'A3'	x'B7'	x'FA'		UPSILON ACUTE	x'95'	x'77'
x'7B'		upsilon diaer.	x'FB'	x'B8'	x'FB'		UPSILON DIAER.	x'96'	x'69'
x'7C'		omega acute	x'FD'	x'B9'	x'FC'		OMEGA ACUTE	x'98'	x'78'

Russian-Specific Codes

The Russian character codes overlap the Greek character codes.

On the workstation, code page 866 is supported for Russian. On the host, code page 1025, which is Cyrillic multilingual is supported for Russian. Code page 1025 includes many characters which do not exist in Russian, and these are not used in Linguistic Tools. The following table relates the Russian characters to internal Linguistic Tools codes, and to both of these code pages:

Table 25 (Page 1 of 2). Linguistic Tools Internal Codes: Workstation and Host Russian Codes

Internal Code	Char.	Description	CP866 Code	CP1025 Code	Internal Code	Char.	Description	CP866 Code	CP1025 Code
x'51'		a	x'A0'	x'77'	x'D1'		A	x'80'	x'B9'
x'52'		b	x'A1'	x'78'	x'D2'		B	x'81'	x'BA'
x'53'		v	x'A2'	x'AF'	x'D3'		V	x'82'	x'ED'
x'54'		g	x'A3'	x'8D'	x'D4'		G	x'83'	x'BF'
x'55'		d	x'A4'	x'8A'	x'D5'		D	x'84'	x'BC'
x'56'		ye	x'A5'	x'8B'	x'D6'		YE	x'85'	x'BD'
x'57'		yo	x'F1'	x'44'	x'D7'		YO	x'F0'	x'63'
x'58'		zh	x'A6'	x'AE'	x'D8'		ZH	x'86'	x'EC'
x'59'		z	x'A7'	x'B2'	x'D9'		Z	x'87'	x'FA'
x'5A'		i	x'A8'	x'8F'	x'DA'		I	x'88'	x'CB'
x'5B'		i-short	x'A9'	x'90'	x'DB'		I-SHORT	x'89'	x'CC'
x'5C'		k	x'AA'	x'9A'	x'DC'		K	x'8A'	x'CD'
x'5D'		l	x'AB'	x'9B'	x'DD'		L	x'8B'	x'CE'
x'5E'		m	x'AC'	x'9C'	x'DE'		M	x'8C'	x'CF'
x'5F'		n	x'AD'	x'9D'	x'DF'		N	x'8D'	x'DA'
x'60'		o	x'AE'	x'9E'	x'E0'		O	x'8E'	x'DB'
x'61'		p	x'AF'	x'9F'	x'E1'		P	x'8F'	x'DC'
x'62'		r	x'E0'	x'AA'	x'E2'		R	x'90'	x'DE'
x'63'		s	x'E1'	x'AB'	x'E3'		S	x'91'	x'DF'
x'64'		t	x'E2'	x'AC'	x'E4'		T	x'92'	x'EA'
x'65'		u	x'E3'	x'AD'	x'E5'		U	x'93'	x'EB'
x'66'		f	x'E4'	x'8C'	x'E6'		F	x'94'	x'BE'
x'67'		x	x'E5'	x'8E'	x'E7'		X	x'95'	x'CA'
x'68'		ts	x'E6'	x'80'	x'E8'		TS	x'96'	x'BB'
x'69'		ch	x'E7'	x'B6'	x'E9'		CH	x'97'	x'FE'

Table 25 (Page 2 of 2). Linguistic Tools Internal Codes: Workstation and Host Russian Codes

Internal Code	Char.	Description	CP866 Code	CP1025 Code	Internal Code	Char.	Description	CP866 Code	CP1025 Code
x'6A'		sh	x'E8'	x'B3'	x'EA'		SH	x'98'	x'FB'
x'6B'		shch	x'E9'	x'B5'	x'EB'		SHCH	x'99'	x'FD'
x'6C'		hard sign	x'EA'	x'B7'	x'EC'		HARD SIGN	x'9A'	x'57'
x'6D'		y	x'EB'	x'B1'	x'ED'		Y	x'9B'	x'EF'
x'6E'		soft sign	x'EC'	x'B0'	x'EE'		SOFT SIGN	x'9C'	x'EE'
x'6F'		e	x'ED'	x'B4'	x'EF'		E	x'9D'	x'FC'
x'7A'		yu	x'EE'	x'76'	x'FA'		YU	x'9E'	x'B8'
x'7B'		ya	x'EF'	x'A0'	x'FB'		YA	x'9F'	x'DD'

Latin II Specific Codes

The Latin II character codes overlap the Greek and Russian character codes.

Latin II characters support several Easter European languages such as Polish, Czech, and Hungarian. Workstation code page 852 and host code page 870 are supported byLinguistic Tools. The following table relates the Latin II characters to internal Linguistic Tools codes, and to both of these code pages.

Description of diacritic marks:

The term "hacek" indicates a diacritical mark shaped like a check mark. The "quote" diacritic is like a double acute accent. The "cedilla" looks like a comma and is placed under a character. The "nasal" mark is placed under a character like a cedilla, but it curves in the opposite direction. The term "slash" indicates that a slash crosses the letter, whereas the term "dot" indicates that a dot is placed over the letter. Note that the code point for the s-cedilla differs from that for the Turkish code page.

Table 26 (Page 1 of 2). Linguistic Tools Internal Codes: Workstation and Host Latin II Codes

Internal Code	Char.	Description	Code Page 852 Code	Code Page 870 Code	Internal Code	Char.	Description	Code Page 852 Code	Code Page 870 Code
x'51'		a-hacek	x'C7'	x'46'	x'D1'		A-hacek	x'C6'	x'66'
x'52'		a-nasal	x'A5'	x'A0'	x'D2'		A-nasal	x'A4'	x'B1'
x'53'		c-hacek	x'9F'	x'47'	x'D3'		C-hacek	x'AC'	x'67'
x'54'		c-acute	x'86'	x'49'	x'D4'		C-acute	x'8F'	x'69'
x'55'		d-hacek	x'D4'	x'EA'	x'D5'		D-hacek	x'D2'	x'FA'
x'56'		e-nasal	x'A9'	x'52'	x'D6'		E-nasal	x'A8'	x'72'
x'57'		e-hacek	x'D8'	x'DF'	x'D7'		E-hacek	x'B7'	x'DA'
x'58'		l-hacek	x'96'	x'57'	x'D8'		L-hacek	x'95'	x'77'
x'59'		l-acute	x'92'	x'58'	x'D9'		L-acute	x'91'	x'78'
x'5A'		l-slash	x'88'	x'9A'	x'DA'		L-slash	x'9D'	x'BA'
x'5B'		n-hacek	x'E5'	x'8B'	x'DB'		N-hacek	x'D5'	x'AB'
x'5C'		n-acute	x'E4'	x'9B'	x'DC'		N-acute	x'E3'	x'BB'
x'5D'		o-quote	x'8B'	x'CF'	x'DD'		O-quote	x'8A'	x'EF'
x'5E'		r-hacek	x'FD'	x'8E'	x'DE'		R-hacek	x'FC'	x'AE'
x'5F'		r-acute	x'EA'	x'CD'	x'DF'		R-acute	x'EB'	x'ED'
x'60'		s-acute	x'98'	x'8A'	x'E0'		S-acute	x'97'	x'AA'
x'61'		s-cedilla	x'AD'	x'8F'	x'E1'		S-cedilla	x'B8'	x'AF'
x'62'		s-hacek	x'E7'	x'9C'	x'E2'		S-hacek	x'E6'	x'BC'

Table 26 (Page 2 of 2). Linguistic Tools Internal Codes: Workstation and Host Latin II Codes

Internal Code	Char.	Description	Code Page 852 Code	Code Page 870 Code	Internal Code	Char.	Description	Code Page 852 Code	Code Page 870 Code
x'63'		t-cedilla	x'EE'	x'44'	x'E3'		T-cedilla	x'DD'	x'B3'
x'64'		t-hacek	x'9C'	x'DD'	x'E4'		T-hacek	x'9B'	x'FD'
x'65'		u-overcircle	x'85'	x'54'	x'E5'		U-overcircle	x'DE'	x'74'
x'66'		u-quote	x'FB'	x'DB'	x'E6'		U-quote	x'EB'	x'FB'
x'67'		z-dot	x'BE'	x'B2'	x'E7'		Z-DOT	x'BD'	x'B4'
x'68'		z-hacek	x'A7'	x'B6'	x'E8'		Z-hacek	x'A6'	x'B8'
x'69'		z-acute	x'AB'	x'B7'	x'E9'		Z-acute	x'8D'	x'B9'

Turkish-Specific Codes

The Turkish character codes overlap the Greek, Russian, and Latin II character codes.

The Turkish alphabet is similar to the Latin alphabet, but it has additional characters with diacritics.

Turkish code page 857 is supported on the workstation, and code page 1026 is supported on the host. The following table relates the unique Turkish characters to internal Linguistic Tools codes, and to both of these code pages. Notice, in particular, that the lowercase "i" with a dot transforms to an uppercase "I" with a dot when capitalized, whereas the dotless lowercase "i" changes to the dotless uppercase "I". For this reason, the dotless-i (Code point X'39') of the constant codes is not used for Turkish, since it does not have a corresponding upper case form. Similarly, the lower case "i" (Code point X'09') and the corresponding upper case Code point "I" (X'89') of the constant codes are not used because they have the wrong upper/lower case relationship for Turkish. The Turkish "yumushak ge" is called here a "g-hacek" because it is written as a "g" with a check mark on top.

Table 27. Linguistic Tools Internal Codes: Workstation and Host Turkish Codes

Internal Code	Char.	Description	Code Page 857 Code	Code Page 1026 Code	Internal Code	Char.	Description	Code Page 920 Code	Code Page 1026 Code
x'51'		g-hacek	x'A7'	x'D0'	x'D1'		G-hacek	x'A6'	x'5A'
x'52'		dotless-i	x'8D'	x'79'	x'D2'		dotless-I	x'49'	x'C9'
x'53'		dotted-i	x'69'	x'89'	x'D3'		dotted-I	x'98'	x'5B'
x'54'		s-cedilla	x'9F'	x'6A'	x'D4'		S-cedilla	x'9E'	x'7C'

Arabic-Specific Codes

The Arabic character codes overlap the Greek, Russian, Latin II, and Turkish character codes.

On the workstation, code page 864 is supported for Arabic. On the host, code page 420 is supported for Arabic. Arabic has no upper case. The following table relates the Arabic characters to internal Linguistic Tools codes, and to both of these code pages. Note: Only the base Arabic shapes are supported. It is the application's responsibility to convert all Arabic characters to their base shapes.

Table 28 (Page 1 of 2). Linguistic Tools Internal Codes: Workstation and Host Arabic Codes

Internal Code	Char.	Description	Code Page 864 Code	Code Page 420 Code
x'51'		hamza	x'C1'	x'46'
x'52'		alif madda	x'C2'	x'47'
x'53'		alif hamza	x'C3'	x'49'
x'54'		waw hamza	x'C4'	x'52'
x'55'		hamza ala nabra	x'C6'	x'55'
x'56'		alif	x'C7'	x'56'
x'57'		ba	x'C8'	x'59'
x'58'		teh marbuta	x'C9'	x'62'
x'59'		ta	x'CA'	x'64'
x'5A'		tha	x'CB'	x'66'
x'5B'		jim	x'CC'	x'68'
x'5C'		comma ha	x'CD'	x'70'
x'5D'		kha	x'CE'	x'72'
x'5E'		dal	x'CF'	x'73'
x'5F'		dhal	x'D0'	x'74'
x'60'		ra	x'D1'	x'75'
x'61'		zay	x'D2'	x'76'
x'62'		seen	x'D3'	x'78'
x'63'		sheen	x'D4'	x'8A'
x'64'		sad	x'D5'	x'8C'
x'65'		dad	x'D6'	x'8E'
x'66'		comma ta	x'D7'	x'8F'
x'67'		comma za	x'D8'	x'90'
x'68'		ayn	x'D9'	x'9C'
x'69'		ghayn	x'DA'	x'A0'
x'6A'		fa	x'E1'	x'AC'
x'6B'		qaf	x'E2'	x'AE'
x'6C'		caf	x'E3'	x'B0'
x'6D'		lam	x'E4'	x'BA'
x'6E'		mim	x'E5'	x'BC'
x'6F'		noon	x'E6'	x'BE'
x'7A'		heh	x'E7'	x'CB'
x'7B'		waow	x'E8'	x'CF'
x'7C'		alif maqsura	x'E9'	x'DA'
x'7D'		yeh	x'EA'	x'DE'
x'7E'		shadda	x'F1'	x'42'
x'C0'		arabic 0	x'B0'	x'DF'

Table 28 (Page 2 of 2). Linguistic Tools Internal Codes: Workstation and Host Arabic Codes

Internal Code	Char.	Description	Code Page 864 Code	Code Page 420 Code
x'C1'		arabic 1	x'B1'	x'EA'
x'C2'		arabic 2	x'B2'	x'EB'
x'C3'		arabic 3	x'B3'	x'ED'
x'C4'		arabic 4	x'B4'	x'EE'
x'C5'		arabic 5	x'B5'	x'EF'
x'C6'		arabic 6	x'B6'	x'FB'
x'C7'		arabic 7	x'B7'	x'FC'
x'C8'		arabic 8	x'B8'	x'FD'
x'C9'		arabic 9	x'B9'	x'FE'
x'D0'		arabic semicolon	x'BB'	x'C0'
x'F5'		arabic question mark	x'BF'	x'D0'
x'CA'		arabic comma	x'AC'	x'79'

Hebrew-Specific Codes

The Hebrew character codes overlap the Greek, Russian, Latin II, Arabic, and Turkish character codes.

On the workstation, code page 862 is supported for Hebrew. On the host, code page 424 is supported for Hebrew. Hebrew has no upper case. The following table relates the Hebrew characters to internal Linguistic Tools codes, and to both of these code pages.

Table 29. Linguistic Tools Internal Codes: Workstation and Host Hebrew Codes

Internal Code	Char.	Description	Code Page 862 Code	Code Page 424 Code
x'51'		alef	x'80'	x'41'
x'52'		bet	x'81'	x'42'
x'53'		gimel	x'82'	x'43'
x'54'		dalet	x'83'	x'44'
x'55'		he	x'84'	x'45'
x'56'		waw	x'85'	x'46'
x'57'		zayin	x'86'	x'47'
x'58'		het	x'87'	x'48'
x'59'		tet	x'88'	x'49'
x'5A'		yod	x'89'	x'51'
x'5B'		kaf final	x'8A'	x'52'
x'5C'		kaf	x'8B'	x'53'
x'5D'		lamed	x'8C'	x'54'
x'5E'		mem final	x'8D'	x'55'
x'5F'		mem	x'8E'	x'56'
x'60'		nun final	x'8F'	x'57'
x'61'		nun	x'90'	x'58'
x'62'		samech	x'91'	x'59'
x'63'		ayin	x'92'	x'62'
x'64'		pe final	x'93'	x'63'
x'65'		pe	x'94'	x'64'
x'66'		zadi final	x'95'	x'65'
x'67'		zadi	x'96'	x'66'
x'68'		qof	x'97'	x'67'
x'69'		resh	x'98'	x'68'
x'6A'		shin	x'99'	x'69'
x'6B'		tav	x'9A'	x'71'

Thai-Specific Codes

The Thai character codes overlap the Greek, Russian, Latin II, Arabic, Hebrew, and Turkish character codes.

On the workstation, code page 874 is supported for Thai. On the host, code page 838 is supported for Thai. Thai has no upper case. The following table relates the Thai characters to internal Linguistic Tools codes, and to both of these code pages.

Table 30 (Page 1 of 2). Linguistic Tools Internal Codes: Workstation and Host Hebrew Codes

Internal Code	Char.	Description	Code Page 874 Code	Code Page 838 Code
x'51'		ko kai	x'A1'	x'42'
x'52'		kho khai	x'A2'	x'43'
x'53'		kho khuat	x'A3'	x'44'
x'54'		kho khwai	x'A4'	x'45'
x'55'		kho khon	x'A5'	x'46'
x'56'		kho rakhang	x'A6'	x'47'
x'57'		ngo ngu	x'A7'	x'48'
x'58'		cho chan	x'A8'	x'52'
x'59'		cho ching	x'A9'	x'53'
x'5A'		cho chang	x'AA'	x'54'
x'5B'		so so	x'AB'	x'55'
x'5C'		cho choe	x'AC'	x'56'
x'5D'		yo ying	x'AD'	x'57'
x'5E'		do chada	x'AE'	x'58'
x'5F'		to patak	x'AF'	x'62'
x'60'		tho than	x'B0'	x'63'
x'61'		tho nangmontho	x'B1'	x'64'
x'62'		tho phuthao	x'B2'	x'65'
x'63'		no nen	x'B3'	x'66'
x'64'		do dek	x'B4'	x'67'
x'65'		to tao	x'B5'	x'68'
x'66'		tho thung	x'B6'	x'72'
x'67'		tho thahan	x'B7'	x'73'
x'68'		tho thong	x'B8'	x'74'
x'69'		no nu	x'B9'	x'75'
x'6A'		bo baimai	x'BA'	x'76'
x'6B'		po pla	x'BB'	x'77'
x'6C'		pho phung	x'BC'	x'78'
x'6D'		fo fa	x'BD'	x'8A'
x'6E'		pho phan	x'BE'	x'8B'
x'6F'		fo fan	x'BF'	x'8C'
x'78'		pho samphao	x'C0'	x'8D'
x'79'		mo ma	x'C1'	x'8E'
x'7A'		yo yak	x'C2'	x'8F'
x'7B'		ro rua	x'C3'	x'9A'
x'7C'		ru	x'C4'	x'9B'
x'7D'		lo ling	x'C5'	x'9C'
x'D1'		thai lu	x'C6'	x'9D'
x'D2'		wo waen	x'C7'	x'9E'

Table 30 (Page 2 of 2). Linguistic Tools Internal Codes: Workstation and Host Hebrew Codes

Internal Code	Char.	Description	Code Page 874 Code	Code Page 838 Code
x'D3'		so sala	x'C8'	x'9F'
x'D4'		so rusi	x'C9'	x'AA'
x'D5'		so sua	x'CA'	x'AB'
x'D6'		ho hip	x'CB'	x'AC'
x'D7'		lo chula	x'CC'	x'AD'
x'D8'		o ang	x'CD'	x'AE'
x'D9'		ho nokhuk	x'CE'	x'AF'
x'DA'		paiyannoi	x'CF'	x'BA'
x'DB'		sara a	x'D0'	x'BB'
x'DC'		mai han akat	x'D1'	x'BC'
x'DD'		sara aa	x'D2'	x'BD'
x'DE'		sara am	x'D3'	x'BE'
x'DF'		sara i	x'D4'	x'BF'
x'E0'		sara ii	x'D5'	x'CB'
x'E1'		sara ue	x'D6'	x'CC'
x'E2'		sara uee	x'D7'	x'CD'
x'E3'		sara u	x'D8'	x'CE'
x'E4'		sara uu	x'D9'	x'CF'
x'E5'		phinthu	x'DA'	x'DA'
x'E6'		sara e	x'E0'	x'DB'
x'E7'		sara ae	x'E1'	x'DC'
x'E8'		sara o	x'E2'	x'DD'
x'E9'		sara ai maimuan	x'E3'	x'DE'
x'EA'		sara ai maimalai	x'E4'	x'DF'
x'EB'		lakkhangyao	x'E5'	x'EA'
x'EC'		maiyamok	x'E6'	x'EB'
x'ED'		maitaikhu	x'E7'	x'EC'
x'EE'		mai ek	x'E8'	x'ED'
x'EF'		mai tho	x'E9'	x'EE'
x'F8'		mai tri	x'EA'	x'EF'
x'F9'		mai chattawa	x'EB'	x'FA'
x'FA'		thanthakhat	x'EC'	x'FB'
x'FB'		nikkhahit	x'ED'	x'FC'
x'FC'		yamakkan	x'EE'	x'71'
x'F0'		fongman	x'EF'	x'80'
x'F1'		angkhankhu	x'FA'	x'90'
x'F2'		khomut	x'FB'	x'A0'
x'C0'		thai 0	x'F0'	x'B0'
x'C1'		thai 1	x'F1'	x'B1'
x'C2'		thai 2	x'F2'	x'B2'
x'C3'		thai 3	x'F3'	x'B3'
x'C4'		thai 4	x'F4'	x'B4'
x'C5'		thai 5	x'F5'	x'B5'
x'C6'		thai 6	x'F6'	x'B6'
x'C7'		thai 7	x'F7'	x'B7'
x'C8'		thai 8	x'F8'	x'B8'
x'C9'		thai 9	x'F9'	x'B9'

Collating Sequences for Linguistic Tools Internal Code Pages

The following tables illustrate the collating sequences which are used for Linguistic Tools internal codepage for various languages. Characters appearing on the same line will be grouped together for the purpose of establishing equivalence. Otherwise, the collating sequence will be from left to right and top to bottom.

These collating sequences are used by the **List Addenda Words** function to order the words in the addenda dictionary for display and by the **Save Addenda** function when saving an addenda dictionary in flat file format.

Greek

0
1
2
3
4
5
6
7
8
9
Α-acute α-acute Α α
Β β
Γ γ
Δ δ
Ε-acute ε-acute Ε ε
Ζ ζ
Η-acute η-acute Η η
Θ θ
Ι-acute ι-acute Ι-diaeresis ι-diaeresis Ι ι
Κ κ
Λ Λ
Μ μ
Ν ν
Ξ ξ
Ο-acute ο-acute Ο ο
Π π
Ρ ρ
Σ σ ς
Τ τ
ϒ-acute υ-acute ϒ-diaeresis υ-diaeresis ϒ υ
Φ φ
Χ χ
Ψ ψ
Ω-acute ω-acute Ω ω

Figure 13 (Part 1 of 2). Greek Collating Sequence for Linguistic Tools Internal Code Page

A a
B b
C c
D d
E e
F f
G g
H h
I i
J j
K k
L l
M m
N n
O o
P p
Q q
R r
S s
T t
U u
V v
W w
X x
Y y
Z z

Figure 13 (Part 2 of 2). Greek Collating Sequence for Linguistic Tools Internal Code Page

Russian

0
1
2
3
4
5
6
7
8
9
rus-A rus-a A a
rus-B rus-b
rus-V rus-v B b V v W w
rus-G rus-g G g
rus-D rus-d D d
rus-YE rus-ye E e rus-YO rus-yo
rus-ZH rus-zh
rus-Z rus-z Z z
rus-I rus-i rus-I-SHORT rus-i-short I i J j
rus-K rus-k K k Q q
rus-L rus-l L l
rus-M rus-m M m
rus-N rus-n H h N n
rus-O rus-o O o
rus-P rus-p
rus-R rus-r P p R r
rus-S rus-s C c S s
rus-T rus-t T t
rus-U rus-u Y y U u
rus-F rus-f F f
rus-X rus-x X x
rus-TS rus-ts
rus-CH rus-ch
rus-SH rus-sh
rus-SHCH rus-shch
rus-HARD SIGN rus-hard sign
rus-Y rus-y
rus-SOFT SIGN rus-soft sign
rus-E rus-e
rus-YU rus-yu
rus-YA rus-ya

Figure 14. Russian Collating Sequence for Linguistic Tools Internal Code Page

Turkish

0
1
2
3
4
5
6
7
8
9
a A á Á â Â à À ã Ã ä Ä
b B
c C
ç Ç
d D
e E é É ê Ê è È ë Ë
f F
g G
g-hacek G-hacek
h H
ı I
i I-dotted í Í î Î ì Ì ï Ï
j J
k K
l L
m M
n N ñ Ñ
o O
ö Ö ó Ó ô Ô ò Ò õ Õ
p P
q Q
r R
s S
s-cedilla S-cedilla
t T
u U
ü Ü ú Ú û Û ù Ù
v V
w W
x X
y Y ý Ý ÿ Ÿ
z Z

Figure 15. Turkish Collating Sequence for Linguistic Tools Internal Code Page

Icelandic:

0
1
2
3
4
5
6
7
8
9
a A â Â à À ã Ã ä Ä
á Á
b B
c C ç Ç
d D
ð Đ
e E ê Ê è È ë Ë
é É
f F
g G
h H
i I î Î ì Ì ï Ï
í Í
j J
k K
l L
m M
n N
ñ Ñ
o O ô Ô ò Ò õ Õ
ó Ó
p P
q Q
r R
s ß S
t T
u U û Û ù Ù ü Ü
ú Ú
v V
w W
x X
y Y ÿ Ÿ
ý Ý
z Z
þ Þ
æ Æ
ø Ø
å Å
ö Ö

Figure 16. Icelandic Collating Sequence for Linguistic Tools Internal Code Page

Swedish and Finnish

0
1
2
3
4
5
6
7
8
9
a A á Á â Â à À ã Ã
b B
c C ç Ç
d D
ð Đ
e E é É ê Ê è È ë Ë
f F
g G
h H
i I í Í î Î ì Ì ï Ï
j J
k K
l L
m M
n N
ñ Ñ
o O ó Ó ô Ô ò Ò õ Õ
p P
q Q
r R
s ß S
t T
u U ú Ú û Û ù Ù ü Ü
v V
w W
x X
y Y ý Ý ÿ Ÿ
z Z
þ Þ
æ Æ
ø Ø
å Å
ä Ä
ö Ö

Figure 17. Swedish and Finnish Collating Sequence for Linguistic Tools Internal Code Page.

Latin II languages (Polish, Hungarian, Czech).

0
1
2
3
4
5
6
7
8
9
a A á Á â Â a-hacek A-hacek a-nasal A-nasal ä Ä
b B
c C ç Ç c-hacek C-hacek
d D d-hacek D-hacek
ð Đ
e E é É ê e-nasal E-nasal Ê e-hacek E-hacek ë Ë
f F
g G
h H
i I í Í î Î
j J
k K
l L l-acute L-acute l-hacek L-hacek l-slash L-slash
m M
n N n-acute N-acute n-hacek N-hacek ñ Ñ
o O ó Ó ô Ô o-quote O-quote ö Ö
p P
q Q
r R r-acute R-acute r-hacek R-hacek
s S s-acute S-acute s-hacek S-hacek s-cedilla S-cedilla ß
t T t-hacek T-hacek t-cedilla T-cedilla
u U ú Ú u-overcircle U-overcircle u-quote U-quote ü Ü
v V
w W
x X
y Y ý Ý
z Z z-acute Z-acute z-hacek Z-hacek z-dot Z-dot
þ Þ

Figure 18. Latin II Collating Sequence for Linguistic Tools Internal Code Page.

All Other Languages

0
1
2
3
4
5
6
7
8
9
a A á Á â Â à À ã Ã ä Ä
b B
c C ç Ç
† ch Ch CH
d D
ð Đ
e E é É ê Ê è È ë Ë
f F
g G
h H
i I í Í î Î ì Ì ï Ï
j J
k K
l L
† ll Ll LL
m M
n N
ñ Ñ
o O ó Ó ô Ô ò Ò õ Õ ö Ö
p P
q Q
r R
† rr Rr RR
s ß S
t T
u U ú Ú û Û ù Ù ü Ü
v V
w W
x X
y Y ý Ý ÿ Ÿ
z Z
þ Þ
æ Æ
ø Ø
å Å

Figure 19. Default (Other Languages) Collating Sequence for Linguistic Tools Internal Code Page.

† This group only applies when the language is Spanish.

Appendix C. Return Codes

LX_ALIAS_NO_MATCH Mismatch between alias and new word forms—the **Add Morphology to Addenda** function expects the same number of forms in both input reply areas

LX_ART_CHECKER_F_CHG Flag change not allowed on continue reply

LX_BAD_ADDRESS Invalid address

LX_BAD_CHAR Null character is not acceptable

LX_BAD_DICT Bad dictionary—no recognizable header in file submitted as a dictionary

LX_BAD_DICT_LEVEL Dictionary version is incompatible with the code

LX_BAD_DICT_NME Invalid dictionary name

LX_BAD_DICT_TKN Invalid dictionary token

LX_BAD_DICT_TKN_CNT Invalid dictionary token count

LX_BAD_ELE_FORMAT Invalid data element format

LX_BAD_ELE_TYPE Invalid element type

LX_BAD_ELEMENTS_CT Invalid element count

LX_BAD_FIRST Invalid first token

LX_BAD_FUNCT_CODE Invalid function code

LX_BAD_LANG_CODE Invalid language code

LX_BAD_LAST Invalid last token

LX_BAD_PARM Missing or illegal input value for parameter

LX_BAD_POS_FILTER_PARM Invalid part of speech filter input—invalid value for *lx_pos_f* field in control block

LX_BAD_RQST_TYPE Invalid request type

LX_BAD_SENT_PARM Invalid sentence determination option input

LX_BAD_SERV_HANDLE Invalid Service Area Handle input

LX_BAD_STOP_WORD_PARM
Invalid stop word option input (*lx_stopw_f*)

LX_BAD_STR_ADDR Invalid or NULL string address provided

LX_BAD_SYN_PARM Invalid synonym option input (*lx_synonym_f*)

LX_BAD_TOKEN Invalid pointer to a token

LX_BAD_WORD Invalid word

LX_BAD_WORD_DECOMP_PARM
Invalid word decomposition option input (*lx_wordde_f*)

LX_BAD_WORD_EXPAND_PARM
Invalid word expansion option input (*lx_wordex_f*)

LX_BAD_WORD_MOD_PARM Invalid word modification option input (*lx_word_mod_f*)

LX_BAD_WORD_REDUCE_PARM
Invalid word reduction option input (*lx_wordre_f*)

© Copyright IBM Corp. 2000

LX_CPG_NOT_SUPPORTED Code page not supported—if the application wishes to use a code page not in the Linguistic Tools default set listed in Table 17 on page 79, it must supply the necessary tables through the **NlpRegisterCodePage()** function.

LX_DELIMITER Target input character is a delimiter

LX_DICT_NO_UPDT Non-updateable dictionary cannot be used with this function—the function is intended to be used only with addenda and other updateable dictionaries (even though the function itself actually may not change the dictionary). This return code indicates that the function was called with a non-updateable dictionary (that is, a system dictionary).

LX_DUP_DICT Duplicate dictionary

LX_DUP_WORD Duplicate word added to addenda dictionary

LX_ELE_P Missing element list

LX_ELT_FORMAT_CHG Change of element format not allowed

LX_END_OF_INPUT End of input reached

LX_END_OF_PROPS End of property list reached

LX_END_OF_STORAGE End of system storage reached

LX_ERROR_LOAD_MODULE Error loading Dynamic Link Library

LX_EXTRACT_PARMS_CHG Parameters were changed on EXTRACT continue reply

LX_FL_ACC_DEND File access denied

LX_FL_NO_FND File not found

LX_FL_OP_FAILD File open failed

LX_FL_SHR_VIOLTN File sharing violation

LX_FOUND_SDLM A text-segment delimiter has been found

LX_FUNC_CODE_CHG Change of function code not allowed

LX_IO_ERROR File input/output error

LX_JOIN_CHAR Character should produce a join flag

LX_MEM_ACC_SHR Access given for existing shared memory

LX_MEM_ALLOC_ERR Memory allocation error

LX_NO_AID_AVAIL No aid available—this function is supported for this language, but no dictionary information for this function was found for the input word. If the input word was not found in the dictionary at all, Linguistic Tools returns LX_WORD_NOT_FOUND.

LX_NO_BIT_MASK_ENTRY No bit mask entry for this PCODE

LX_NO_BIT_MASK_TABLE No bit mask table in the dictionary

LX_NO_CPX No Lexical Analysis for this language

LX_NO_DATA_AVAIL No data available—no dictionary information is available for this function; the function is not supported for this language.

LX_NO_DATA_SAT No data satisfies input parameter request

LX_NO_DECOMP No decompositions found

LX_NO_INFLECT	Synonym inflection not supported
LX_NO_REPLY_P	Missing reply area
LX_NOT_DELIMITER	Target input character is ***not*** a delimiter
LX_OUT_GOAL	Output goal reached before end of input
LX_PATH_TOO_LONG	Linguistic Tools path length exceeds limit
LX_RC_OK	Processing successfully completed
LX_RC_OK_NC	Processing successfully completed; input unchanged
LX_RC_OK_NHS	Processing successfully completed; Non-Hangeul stem returned by **Morphological Identification** for Korean.
LX_REPLY_FULL	Reply area is full—application should process output present in the reply area, then call the function again with a continue reply to process the rest of the input.
LX_REPLY_2_SMALL	Reply area is too small to allow any output—application should increase the size of the reply area, then call the function again.
LX_SHARED_MEM_NAME_ERROR	Shared memory name is not appropiate
LX_SPEL_CHNG	(De)hyphenation changed spelling
LX_SPELL_VER_F_CHG	Spell verify flag change not allowed on continue reply
LX_SPLIT_ERROR	Unable to split token
LX_SYN_OK_NO_MID	Morphology error in synonym aid; output not inflected—the **Morphological Identification** call issued by the **Synonym Aid** function failed, so no morphological information has been returned for the input word. If the input word is a lemma form for which synonym data are available, those synonyms will be returned. If the input word is an inflected form which does not coincide with any lemma form, no synonyms will be returned. This does not cause a hard error.
LX_UNEXPECTED_RC	Unknown or unexpected condition
LX_WORD_NOT_FOUND	Word not found in dictionary—the word was not found in any of the dictionaries present in the current dictionary list.
LX_ZERO_SIZE	Invalid zero size
LX_2_MANY_DICTS	Too many dictionaries open

Appendix D. Language-Specific Processing

Arabic Language-Specific Processing

Morphological Identification

Because Arabic morphology is so different from that of the European languages, an external morphological analyzer is used rather than the sort of dictionary coding provided for the European languages.

© Copyright IBM Corp. 2000

Catalan Language-Specific Processing

Spell Verify

Prefix processing

Catalan allows the combination of prefixes with some words by joining them with an apostrophe. The rules enforced by Linguistic Tools governing this formation are:

- The prefix must be found in the list of valid prefixes shown below
- The word following the prefix and apostrophe must begin with the letter “a”, “e”, “i”, “o”, “u”, “h”, or some accented form of these letters
- The word following the prefix must not be found in the list of exception trailing words shown below
- The prefix must not be mixed case
- The word following the prefix must be verified as correct

If any of these conditions are not met, the prefixed word is considered incorrect.

Valid prefixes are

```
d'
l'
m'
n'
s'
t'
```

Figure 20. Catalan Prefixes

At present there are no exception trailing words for Catalan.

Hyphenation/Dehyphenation

During hyphenation special processing is done for the “high-dot” character (represented by · in the following examples). When the letter group “l·l” occurs and a hyphenation point occurs after the high-dot, then the high-dot character is deleted during hyphenation. For example:

```
novel·la  -->  ...... novel-  (line ends in syllable hyphen)
               la ......      (next line begins with "la")
```

This deletion of the high-dot character during hyphenation must be reversed during dehyphenation. Specifically, when a word ends a text line with “...l-” (“l” plus syllable hyphen), and the next text line begins with an “l...”, dehyphenation must restore the high-dot character is to the word. For example:

```
..... novel-  (line ends in syllable hyphen)  -->  novel·la
la ......     (next line begins with "la")
```

The use of the high-dot outside of the “l·l” group does not occur in Catalan. “ll” without high-dot occurs, but the hyphenation must occur prior to the first of the two “l”s.

Lexical Analysis for Catalan

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see “Hyphenated Forms” on page 296). This processing includes dehyphenation. Lexical Analysis for Catalan supports spelling changes that occur in dehyphenation.

Note: A Catalan dictionary must be the first dictionary activated for spelling changes during dehyphenation to be made properly.

Numbers and Numeric Ranges

The processing of numbers and numeric ranges uses a method analogous to the one used for English (see “Numbers” on page 300), except that the group delimiter is the period and the radix point is the comma.

Prefixes

Simple tokens that begin with one of the prefixes in Figure 20 on page 282 are split into two tokens. The first token contains the prefix, including the final apostrophe.

Note: Lexical Analysis does not check that the letters following a prefix are valid; that is the responsibility of the caller. The property LX_PREFIX is added to this token. The second token contains the rest of the original token.

Property Inventory

This is an inventory of properties assigned to tokens by the Lexical Analysis function:

LX_PREFIX A prefix ending in “'”.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Catalan the following language-specific processing is performed:

- All accents are removed.

Chinese Language-Specific Processing

Linguistic Overview

The Chinese language is written completely in ideographic characters known in Chinese as *hanzi*. Because of the number of symbols used, these must be represented in the computer as a Double-Byte Character Set (DBCS). The Traditional Chinese character set includes only those characters used in the Republic of China (Taiwan) and Hong Kong. The Simplified Chinese character set includes only those characters used in the People's Republic of China and Singapore.

Text data entry in Chinese is done in one of two basic ways: on the basis of pronunciation (phonetically) or on the basis of the way in which the character is written. Phonetically-based input involves typing in a phonetic representation - either romanized (Latin alphabet) or using the *chuyinfuhao* (for Traditional Chinese) and *pinyin* (for Simplified Chinese) symbols—and converting to *hanzi*. Because of the extremely high degree of homonymy (different words with the same pronunciation) in Chinese, the same phonetic string may represent many *hanzi*. The *hanzi* conversion function may make use of frequency, context, stored user preferences, or other kinds of information to decide which of the possible *hanzi* to offer first. Writing (stroke)-based input involves dividing up the character into components, entering these using the *cang jie* system (for Traditional Chinese), and converting the result to *hanzi*. Because the input does not specify every single stroke in order and its exact relationship with other strokes, some ambiguity is possible, but much less than with phonetically-based input.

Normal Chinese text consists of a string of DBCS characters without any special delimiter between words. Phrases and sentences may be delimited by either DBCS or SBCS punctuation characters. A mixed text may include SBCS strings such as non-Chinese names or terms, Western-style numbers, etc. Word isolation strategies based on European-language writing conventions will not work for such text.

The Chinese language does not have inflected forms, so morphological identification and generation of inflected forms are not needed for Chinese. Because synonym data are not included in the Linguistic Tools Chinese dictionary, no Chinese synonym aid is available in Linguistic Tools.

Spell Verify and Spell Aid are not supported for Chinese because of the difficulty of implementing either.

No hyphenation/dehyphenation or abbreviation processing is done for Chinese because hyphenation and abbreviations (in the European-language sense) do not occur in Chinese.

Supported Functions

Only the following functions are supported for Chinese:

- Initialize Linguistic Service
- Terminate Linguistic Service
- Activate Dictionary
- Deactivate Dictionary
- Simple Tokenization

- Text Segment Identification
- Lexical Analysis
- Morphological Identification
- Extract Keyword
- Extract Query Term
- Create Addenda Dictionary
- Add Word with Data
- Remove Word from Addenda
- List Addenda Words
- Save Addenda
- Look up Word
- Poe utilities
  – NlpSetDictDef
  – NlpQrySearchPath
  – NlpSetSearchPath
  – NlpFindDicts
  – Poe Token List Utilities

No other functions are supported for Chinese.

Simple Tokenization for Chinese

Linguistic Tools' **Simple Tokenization** function will separate the DBCS and SBCS portions of a mixed text from each other and will do normal European-style word isolation for the SBCS portion, but will do no further processing on the DBCS portion of the text. **Lexical Analysis** must be invoked to perform word isolation on the DBCS text.

Lexical Analysis for Chinese

The **Lexical Analysis** function for Chinese uses dictionary information to break the text into words. Words are represented as tokens and valid substrings are represented as LX_SUBTERM properties. See "Lexical Analysis" on page 181 for details.

Addenda Dictionary Support

Currently a Chinese addenda dictionary may only be used to store words and any associated user data.

Morphological Identification for Chinese

Morphological Identification for Chinese will produce the output described in "Morphological Identification" on page 167 except for the following deviations:

- The input word passed can be a mixed string and must be in either the host Chinese combined code pages (935 or 937) or PC combined code page (948, 950, or 1381). All shift-in and shift-out characters found in the input will be removed. All other single byte characters, in both host and PC code pages, will be converted into double byte characters prior to searching the dictionary, since the dictionary does not contain any single byte characters.
- The input word returned in the reply area is an all DBCS string to show exactly how the word was looked up in the dictionary. Note that for host code pages the DBCS string will be surrounded by a single set of SBCS shift-out/shift-in characters.

- The POS returned will be the condensed POS (same as for Latin languages) and will be derived from the last EBCDIC value of the PCODE.
- The DBPC field will contain a non-zero value. Possible values are described in “Chinese Double-Byte Parsing Codes (DBPCs)” on page 287.
- No alias processing is performed.
- No inflective classification (bit mask) information is returned.
- Output cannot be used to generate inflected forms because the the Generate Inflected Forms function is not supported for Chinese.
- The Morphological Identification function will search all dictionaries in the input dictionary list until it finds a word that can qualify as the lemma of the input word. The lx_dict_found_in flag will return the token of the first dictionary in which such a word was found.
- The entire input word will be returned as the lemma with a return code, LX_RC_OK.

The double-byte parse code (DBPC), which is a numeric representation of the PCODE information, will be returned to the calling application. These codes are documented in “Chinese Double-Byte Parsing Codes (DBPCs)” on page 287.

Keyword and Query Term Extraction for Chinese

The **Extract Keywords** and **Extract Query Terms** functions use the output of **Morphological Identification** to return part of speech information for keywords from a block of text. For the text extraction functions, the application must set a flag indicating the part(s) of speech desired. Only words which have been tagged by **Morphological Identification** as having the part(s) of speech specified by the filtering options will be returned. Part of speech information is returned in terms of the condensed POS, rather than the CPOS (see “Chinese Double-Byte Parsing Codes (DBPCs)” on page 287 for details). Note that “unknown” words (those which cannot be found in the dictionary) are returned with part of speech LX_EKW_UNKNOWN. For part of speech filtering, this category is included with the nouns.

While **Lexical Analysis** on the host returns shift-out and shift-in as separate tokens from the DBCS text string

```
token 1 — SO
token 2 — DBCS_text
token 3 — SI
```

the text extraction functions return the DBCS text string enclosed by shift-out and shift-in:

```
token 1 — SO_DBCS_text_SI
```

Thus the host DBCS token is two bytes longer than the corresponding DBCS token on the PC, but the number of tokens returned by the text extraction functions is the same on host and PC.

Chinese Double-Byte Parsing Codes (DBPCs)

The part of speech codes for DBCS languages differ from those for the European languages. Technically, they should be referred to as DBCS parsing codes (DBPC) rather than part of speech codes, because they are used by the Lexical Analysis function to determine base forms (lemmas) of words in Chinese text, but they do not correspond exactly to the parts of speech used to categorize words in the European languages.

DBPC values used in the Chinese dictionary are listed in Table 31 These values may be assigned to words in an addenda dictionary. For convenience, the table of Chinese parse codes has been given in order according to DBPC number. The condensed POS code, returned by the Morphological Identification (MID) function, have been defined as external constants in EFZLEXTY.H. Others may be defined by the application, if needed.

Table 31 (Page 1 of 2). Chinese Double-Byte Parse Codes (DBPCs)

DBPC	PCODE	Description
1	N0nN	Nouns, may also include the following subcategories:
2	NC0nN*	Common nouns, for example, tian1, ren2, mian4tiao2, mu4ban3
3	NF0nN*	Family names, for example, Zhao4, Qian2, Sun1, Li3
4	NL0nN*	Locative nouns, for example, shang4, xia4, qian2mian4, zuo3bian1
5	NP0nN*	Proper nouns, for example, Zhong1guo2, Mei3guo2, Huang2he2, Sun1 Zhong1shan1
6	MW0nN*	Measure words, may also include the following subcategories:
7	MWN0nN	Nominal measure words, for example, “ben3” in yi1 ben3 shu1 (a book)
8	MWV0nN	Verbal measure words, for example, “xia4” in kan4 yi1 xia4 (have a look)
9	SUF0nN*	Suffixes, for example, hua4, jia1, xing4, zhe3
10	PN0nP*	Pronouns, may also include the following subcategories:
11	PNC0nP*	Common pronouns, for example, wo3, ta1, zhe4, zi4ji3
12	PNI0nP	Interrogative pronouns, for example, shui2, shen2me5, na3li3, duo1shao3
13	INT0nP*	Interrogatives, for example, ma5, ba5, a5, ne5
14	V0nV*	Verbs, may also include the following subcategories:
15	VI0nV	Intranstive verbs, for example, song4xing2, xiu1xi5, zhao2xiang3, zhi3zheng4
16	VT0nV	Transtive verbs, may also include the following subcaterories:
17	VAUX0nV*	Auxiliary verbs, for example, neng2, hui4, yuan4, hui4
18	VTA0nV	Action verbs, for example, shuo1, kan4, zou3, da3
19	VTD0nV	Directional verbs, for example, shang4, xia4, qi3lai2, chu1qu4
20	VTE0nV	Existancial verbs, for example, you3, zai4, cun2zai4, ju4bei4
21	VTF0nV	Figurative verbs, for example, xiang4, si4, ru2, fang3fu2
22	VTI0nV	Imperative verbs, for example, shi3, jiao4, ming4ling4, po4shi3
23	VTL0nV	Linking verbs, for example, shi4, suan4, deng3yu2, jiao4zuo4
24	A0nJ*	Adjectives, may also include the following subcategories:

Table 31 (Page 2 of 2). Chinese Double-Byte Parse Codes (DBPCs)

DBPC	PCODE	Description
25	AC0nJ	Common adjectives, for example, hao3, xiao3, mei3li4, you1xiu4
26	ANP0nJ	Non-predicate adjectives, for example, xiao3xing2, shang4deng3, e4xing4, zhen1zheng4
27	NUM0nJ*	Numbers, may also include the following subcategories:
28	NUMC0nJ	Cardinal numbers, for example, yi1, shi2, bai3, qian1, wan4, yi4
29	NUMO0nJ	Ordinal numbers, for example, “di4” in di4 yi1 (first), “tou2” in tou2 yi1 hui2 (first time)
30	PREF0nJ*	Prefixes, for example, lao3, fan3, ban4, hou4
31	ADV0nA*	Adverbs, may also include the following subcategories:
32	ADVA0nA	Approximative adverbs, for example, da4gai4, ye3xu3, hao3xiang4, si4hu1
33	ADVD0nA	Degree adverbs, for example, hen3, zui4, fei1chang2, te4bie2
34	ADVF0nA	Frequentative adverbs, for example, you4, zai4, duo1ci4, zai4shan1
35	ADVL0nA	Linking adverbs, for example, ji4, you4, ye3, jiu4
36	ADVM0nA	Mood adverbs, for example, ke3, dao3, jian3zhi2, di2que4
37	ADVN0nA	Negative adverbs, for example, bu4, mo4, mei2you3, bu2yong4
38	ADVT0nA	Time adverbs, for example, zheng4, zai4, ma3shang5, li4ke4
39	ONOM0nA	Onomatopoeia, for example, peng1, ding1dong1, hua1la1la1, wang1wang1wang1
40	CJ0nC*	Conjunctions, may also include the following subcategories:
41	CJS0nC	Subordinate conjunctions, for example, su1ran2, yin1wei2, zhi3yao4, yu2shi4
42	CJC0nC	Coordinate conjunctions, for example, gen1, tong2, bing4qie3, huo4zhe3
43	INTJ0nI*	Interjections, for example, a1, ai1, ai1ya1, ha1ha1
44	P0nR*	Prepositions, for example, cong2, xiang4, wei4le5, dui4yu5
45	AW0nI	Auxiliary words, may also include the following subcategories:
46	AWS0nI*	Structure auxiliary words, for example, de5, di5, d5
47	AWT0nI*	Tense auxiliary words, for example, zhe5, le5, guo5
48	S0nI*	Sentences, for example, Ren2yan2 ke3 wei4 (Gossip is a fearful thing).

Note that:

- PCODE has a maximum length of 7 letters, the first four letters are for Chinese POS, followed by two reserved digits, 0n, and the last letter indicates the condensed POS code. For example, NC02N, NP03N, PREF05J, and NUM04J are all legal.
- POSs with an asterisk (*) indicate that they are currently in the existing Traditional and Simplified Chinese dictionaries.
- The condensed POS code is:
  - LX_NOUN if the last letter of PCODE is N

- LX_VERB if the last letter of PCODE is V
- LX_ADJECTIVE if the last letter of PCODE is J
- LX_ADVERB if the last letter of PCODE is A
- LX_PRONOUN if the last letter of PCODE is R
- LX_INTERJECTION if the last letter of PCODE is I
- LX_CONJUNCTION if the last letter of PCODE is C
- LX_PREPOSITION if the last letter of PCODE is P

Danish Language-Specific Processing

Spell Verify

Danish is a language which allows compound words. Its dictionary therefore has BOFA flags and the compound word processing is performed, as needed, when verifying words. Only words five characters or longer are processed as potential compounds.

Elliptic enumeration

Elliptic enumeration allows words coded with BOFA “F” (which can only occur in the front or middle of a compound) to verify if they are followed by a hyphen.

Lexical Analysis for Danish

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see “Hyphenated Forms” on page 296). This processing includes dehyphenation.

Numbers and Numeric Ranges

The processing of numbers and numeric ranges uses a method analogous to the one used for English (see “Numbers” on page 300) except that the group delimiter is the period and the radix point is the comma.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Danish the following language-specific processing is performed:

- A-overcircle (å), ae-ligature (æ), and o-slash (ø) are rewritten as “aa”, “ae”, and “oe”, respectively.
- All other accents will be removed.

Dutch Language-Specific Processing

Spell Verify

In response to the new spelling which was implemented in the Netherlands in September 1996, IBM Nederland decided to update and improve the contents of the dictionaries. This project was carried out in cooperation with Van Dale Lexicografie B.V., the main provider of lexicographic products and services in the Netherlands. As a result, these new dictionaries not only comply with the new spelling rules, but are also updated with a large number of new words which have found their way into everyday speech.

As explained in the brochure "Van Dale en de nieuwe spelling" (available on the Internet at www.vandale.nl), there is some difference of opinion on the exact spelling of a small number of words between the compilers of the "Woordenlijst Nederlandse Taal" (the Groene Boekje) on the one hand and the compilers of the Van Dale dictionaries (as well as Wolters and Prisma). These spelling differences are divided into two categories: "acceptable differences" (see paragraph 3.3 in the brochure) and "genuine differences" (see paragraph 3.3). The first category relates to different spelling versions which are the result of different interpretations of particular spelling rules by Van Dale and the "Nederlandse Taalunie" (the organisation which compiled the "Groene Boekje"). For this category, both resulting spelling versions have been approved by the Taalunie. The second category relates to differences which exist because Van Dale—for linguistic reasons—applies a different spelling rule than the Taalunie to some words.

For this reason, IBM Nederland decided to put together two dictionaries: NEDERLND.DIC and NEDPLUS.DIC. Whereas NEDERLND.DIC is solely based on the Groene Boekje, NEDPLUS.DIC contains both the spelling versions offered by the Groene Boekje and those offered by Van Dale, Wolters and Prisma.

Comprehensive information on both the new spelling rules and the various spelling versions can be found in the brochure "Van Dale en de nieuwe spelling" referred to above.

Compound Words

Dutch is a language which allows compound words. Its dictionary therefore has BOFA flags and the compound word processing is performed, as needed, when verifying words. Only words five characters or longer are processed as potential compounds.

Elliptic enumeration

As in other languages employing the BOFA system for verifying compound words, elliptic enumeration is supported in Dutch. This allows words coded with BOFA "F" (which can only occur in the front or middle of a compound) to verify if they are followed by a hyphen.

Stress accents

In Dutch, the user is permitted to add emphasis to a word by adding accents to vowel letters. Such accents can also occur in some words in the Dutch dictionary. Therefore, special processing is done for Dutch words which contain accented letters. If a Dutch word is not found in a dictionary, and the word contains one or more letters with acute or grave accents, the accents are removed from all letters in the word and the dictionaries are searched again. This search includes invoking the compound word processing. It is recognized that if a word which normally contains an accent has another accent added by the user, this processing will not verify the word as being correctly spelled.

Hyphenation

When a word is hyphenated preceding a vowel with a diaeresis, the diaeresis is removed. The word “geïsoleerd”, when hyphenated after “ge” loses the diaeresis, becomes “ge-/isoleerd”. Words such as this are marked in the source dictionary with a comma at that point - “ge,ïsoleerd”, since the character with diaeresis is translated to the character without the diaeresis in the corresponding upper or lower case.

Another special case is the elimination of a double letter, for example, “chocolaatje” hyphenates as “chocola-/tje”. The source entry for this word is marked as “chocola,atje”. Words with identical vowels on either side of the special hyphenation marker lose the second vowel upon hyphenation except when the vowel is “e”. For the cases with “e,e”, the following exceptions, we use the following rule: “e,etje” → “é-/tje”, as in the following words:

```
cafeetje      -->   café-tje
clicheetje    -->   cliché-tje
comiteetje    -->   comité-tje
```

There are 3 exceptions to this rule (6 if one counts the plural forms); these must be handled differently than above:

```
dejeuneetje    -->  dejeuner-tje
dineetje       -->  diner-tje
soupeetje      -->  souper-tje
```

Dehyphenation of these specially-processed words requires that the access code check for the following cases and apply the appropriate transformation rules:

hyphen context	transformation	example
a-a	aä	naäpen
a-e	aë	pentaëder
a-i	aï	maïzena
a-ui	aui	
a-u	aü	Kafarnaüm
e-e	eë	geërfd
e-ij	eij	beijverd
e-i	eï	beïnvloeden
e-ui	eui	geuit
e-u	eü	geürineerd
i-ee	iee	officieel
i-eu	ieu	officieus
i-e	ië	officiële
o-e	oë	zoëven
o-i	oï	colloïdaal
o-o	oö	zoölogie
u-i	uï	fluïdum
u-u	uü	vacuüm
a-tje	aatje	chocolaatje
é-tje	eetje	employeetje
o-tje	ootje	autootje
u-tje	uutje	menuutje
dejeuner-tje	dejeuneetje	dejeuneetjes
diner-tje	dineetje	dineetjes
souper-tje	soupeetje	soupeetjes

Lexical Analysis for Dutch

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see "Hyphenated Forms" on page 296). This processing includes dehyphenation.

Numbers and Numeric Ranges

The processing of numbers and numeric ranges uses a method analogous to the one used for English (see "Numbers" on page 300) except that the group delimiter is the period and the radix point is the comma.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Dutch the following language-specific processing is performed:

- All accents, including tremas, will be removed.

English (US, UK, and Australian) Language-Specific Processing

Spell Aid

English allows possessives to be formed by adding an apostrophe-s ('s) to singular words. Special processing is done when computing the Spell Aid candidate quality-ranking for words ending in apostrophe-s to increase the likelihood that the aid candidate words will also end in apostrophe-s.

Lexical Analysis for English

Apostrophe Forms

Contracted Auxiliary Verbs: The contracted form of the auxiliary is separated from the remaining portion of the simple token. The property LX_CNTRAUX is added to each new token that represents a contracted auxiliary form.

Contracted Form	Full form(s)
'd	had, would
'll	will
'm	am
're	are
's	has, is
've	have

Contracted Negatives with Auxiliary Verbs: Most contracted negatives are included in the dictionary, so no special processing by lexical analysis is required. An inventory of these forms appears below. The property LX_CNTRNEG is added for any word ending in *n't*. Less common forms that currently do not appear in the dictionary are marked with an asterisk.

```
-------------------
Contracted Negative
   (Simple Token)
-------------------
      don't
      doesn't
      didn't

      haven't
      hasn't
      hadn't

     *ain't
      isn't
      aren't
      wasn't
      weren't

      can't
      couldn't
     *mayn't
      mightn't
      shan't
      shouldn't
      won't
      wouldn't
      mustn't
     *oughtn't
     *usedn't
     *needn't
     *daren't
```

Abbreviation and Acronym Plurals: As an indication of plurality, either "'s" or "s" may be appended to abbreviations (without the period) or acronyms.

Some examples:

```
He got three RBI's.
Tokens and features are converted into GRCB's.

He got three RBIs.
Tokens and features are converted into GRCBs.
```

Single letter Plurals: Single letters of the alphabet also can be pluralized with "'s":

```
Don't forget to dot your i's and cross your t's.
There were a lot of B's, and only a few A's and C's.
```

Recognition of apostrophe forms is supported currently in U.S., U.K. and Australian English.

Hyphenated Forms

When it occurs between alphabetic characters, the hyphen "-" is treated as a token delimiter by the Simple Tokenization function. Thus if the input text contains the string "mother-in-law", Simple Tokenization would yield five text tokens: "mother", "-", "in", "-", and "law". Similarly, the text string "machine-readable" would become three text tokens: "machine", "-", and "readable". Since many text processing applications need to have hyphenated forms represented as a single token, Lexical Analysis joins the elements of a hyphenated form into a single token. After Lexical Analysis, then, these forms would be represented as single tokens: "mother-in-law" and "machine-readable".

At present, a simplified treatment of hyphenated forms is used. For hyphenated forms not found in the dictionary, the rightmost element is isolated and treated as the "head" of the complete form. For purposes of syntactic analysis, the complete hyphenated form is treated as if it had the same syntactic characteristics as the head element.

A property with the name LX_HYPHEN is added to the token. The value of this property is the position in the string representing the word of the rightmost hyphen preceding the rightmost element of the hyphenated form. The property with name LX_CPXHEAD has as value a pointer to a string that represents the head element of the hyphenated form. This string is used as the basis for dictionary lookup.

This approach is adequate for many hyphenated forms, for example:

```
machine-readable
natural-language
```

For other forms, however, this results in the assignment of inappropriate properties to the hyphenated form. This is the case, for example, with forms like *run-on*. In the sentence:

```
Run-on's are considered stylistically poor.
```

it is impossible to produce a correct syntactic analysis of the sentence if the hyphenated form is treated syntactically as a preposition.

Dehyphenation: After Simple Tokenization, a hyphenated form like "con-taining" in (1.a), below, would be represented by the four tokens in (1.b). In Lexical Analysis, these four tokens are reduced to a single token to correctly represent the word "containing." Note that cases such as this, in which the hyphen is eliminated, must be distinguished from cases like "mother-in-law" in (2.a), and "machine-readable" in (3.a) where the hyphen is retained. This can be done accurately only by consulting a dictionary to (a) identify lexicalized hyphenated forms like "mother-in-law", (b) determine if a form like "con-taining" would be a valid lexical item if the hyphen was removed, or (c) determine if the elements of a form like "machine-readable" are valid lexical items. The resulting tokens would appear as in (1.c), (2.c), and (3.c).

```
1. a.  You can create a file called MYTEKEY.DEF con-
       taining the appropriate MYTEKEY statements.

   b.  "con"   "-"   (NEWLINE)   "taining"

   c.  "containing"
```

```
2. a.  In December of 1986, I had dinner with my mother-
       in-law at a Japanese diner in Manhattan.

   b.  "mother"   "-"   (NEWLINE)   "in"   "-"   "law"

   c.  "mother-in-law"

3. a.  Nowadays, there is no shortage of machine-
       readable text.

   b.  "machine"   "-"   (NEWLINE)   "readable"

   c.  "machine-readable"
```

The current implementation of dehyphenation uses the Dictionary Access Processor function Spell Verify to distinguish ambiguous, end-of-line hyphens as either syllable hyphens, used to break a word at a syllable boundary for formatting purposes, as in (1.a), or required hyphens, used to separate the elements of the complete hyphenated word, as in (2.a) and (3.a).

It begins by assuming that all end-of-line hard hyphens (the ambiguous hyphens) are syllable hyphens, and attempts to find in the dictionary the complete word with the current configuration of hyphens. If it cannot, it methodically postulates that some of the ambiguous hyphens are required hyphens and others are syllable hyphens, checking all possible unique configurations of hyphens against the dictionary in hopes of resolving all ambiguity. The routine finishes when either all of the ambiguous hyphens have been resolved or the routine has exhausted all possible hyphen configurations. If all possible hyphen configurations have been tried but all ambiguity could not be resolved (which means that one of the parts of the hyphenated word could not be found in the dictionary), the routine settles on the the configuration where all remaining ambiguous hyphens remain hard hyphens.

In the dehyphenation process, the routine may find that a hard hyphen appearing at the end of a line should really be a syllable hyphen. It has reasoned this by consulting the dictionary. Since the original string that contains the hard hyphen does not represent the word accurately, the routine creates a new string, put in the property LX_CPXWORD, which represents the string correctly, i.e., treats the hard hyphen as a syllable hyphen. In English texts, it has the effect of removing the hyphen altogether. In non-English texts, the presence of the syllable hyphen may effect some spelling changes.

Note: If dehyphenation determines that the word contains all syllable hyphens, it adds neither the property LX_CPXHEAD nor LX_HYPHEN. For example (1.a), dehyphenation would only add the property LX_CPXWORD, whose text string would be (1.c).

End-of-line Syllable Hyphens: It should be noted that only hard hyphens at the end of a line are considered ambiguous. The routine knows that end-of-line syllable hyphens are unambiguous, thus respecting an application's use of them.

End-of-line Hyphens Followed by Blanks: Some applications routinely append blanks to the ends of lines. For this reason, the dehyphenation routine must recognize that an end-of-line hyphen is acceptable even when followed by blanks. To accurately represent the text string, though, the routine must use an

LX_CPXWORD property, since the original text string would contain embedded blanks. For example, if the hyphenated word were:

```
machine-
readable
```

and the end-of-line hyphen was followed by one blank, the original text string would be *"machine- readable"*. The assigned LX_CPXWORD string would be the correct form, i.e. *"machine-readable"*.

Recognition of hyphenated words is supported currently in U.S., U.K. and Australian English.

Alternatives with "/"

Forms containing "/" to indicate alternatives, such as:

```
and/or,   is/are,   mission/responsibilites
```

are represented by three tokens each after tokenization. For purposes of syntactic analysis, it is best to combine all three elements into a single token. The rightmost element is then used for dictionary access, so the entire complex form will have the syntactic characteristics of the rightmost element.

A property with name LX_SLASH is added to the resulting token's property list. The value of this property is the position of the rightmost slash in the string representing the word. A property named LX_CPXHEAD has as its value a token handle that represents the rightmost element of the complete form.

Since it is possible that a hyphenated word appears within a slash form, the routine is aware that property interactions may arise. The routine adds a property with the name LX_CPXWORD when the original text does not represent the word correctly. For example, if *input/output* were split across a line as in

```
The machine is capable of sophisticated input/out-
put.
```

the original text *input/out-put* would not properly represent the word. The routine thus creates an LX_CPXWORD property to hold the *input/output* representation of the string.

It is hoped that the performance of the routine agrees with the intuitions of the caller. For the example *manual/machine-readable*, the routine would assign the LX_CPXHEAD property to the word *readable* since that word is the LX_CPXHEAD of *machine-readable*. If the word were split across a line, as in:

```
Regardless of the manual/machine-read-
able form of the data...
```

then the LX_CPXHEAD property would still contain the word *readable* and the token would have an LX_CPXWORD property which contained *manual/machine-readable*.

The LX_CPXWORD property will appear in a form containing a slash when

1. One or more of the parts in a slash form have an LX_CPXWORD property; or,

2. A slash appearing at the end of the line is followed (before the newline) by blanks. This would occur if blanks are appended to and not stripped from the end of the line. For example, if the form *input/output* appeared as:

```
input/
output
```

and the slash were followed by a blank, the original text string would be

```
input/ output
```

with the explicit blank intervening. In this example, the LX_CPXWORD property would be *input/output*.

Forms containing more than one slash will be treated in the same way.

In addition, names such as:

```
System/3X,   DisplayWrite/370,   DW/370
```

must be dealt with. Though the rightmost element will not be found in the dictionary, such forms can be treated in the same fashion as “and/or”. The default syntactic category used for “3X” or “370”, will be used for the whole complex form.

Recognition of words containing slashes is supported currently in U.S., U.K. and Australian English.

Dates

Various forms of dates are recognized and identified by adding LX_DATE as a property to the token. Date formats to be recognized include:

```
 10/1/87     10-1-87
10/31/87    10-31-87
 2/22/48     2-22-48
    9/89        9-89
    12/2        12-2

In general: MsDsY, MsD, and MsY, where
            M represents the month, D the day,
              and Y the year, and
            M = mm or m
            D = dd or d
            Y = yy
            s = / or -
```

The value of each element of a date can be examined to verify that it is in a range appropriate for dates.

Several additional properties are added that indicate the internal structure of the date. Properties with names LX_MONTH, LX_DAY, and LX_YEAR have as values the positions of the first character of the part of the token string that represents the month, day, and year in the date. Properties with names LX_MONTHLEN, LX_DAYLEN, and LX_YEARLEN have values that specify the lengths of the strings representing each component. Properties LX_MONTHERR, LX_DAYERR, and LX_YEARERR indicate that an inappropriate value for a date component has been identified. Presently, the expected ranges for the components are: 1 <= month <= 12, 1 <= day <= 31, and 1 <= year <= 9999.

The property LX_NOLOOKUP will be attached to date forms because dates should not be looked up in the dictionary. This property is added during default (language-independent) processing based on the string type of the token.

Since the slash character is not considered a delimiter when surrounded by numeric characters, a date represented in this format is recognized as a single token by the Lexical Analysis function. When the date recognition routine sees the use of the hyphen to delimit the fields, it should not decide arbitrarily that the form is necessarily a date; it might be a numeric range that fits the criteria of a date form. To note this ambiguity, the routine marks the token as both a date *and* a numeric range.

Recognition of date expressions is currently supported for U.S. English only.

Times

Various forms of times must be recognized and identified by adding a property with name LX_TIME to the token. Time formats that represent hours of the day and night include:

```
2:30
10:25:31
17:15
```

A similar format can be used to specify a span of time. For example:

```
72:48:15.347
```

Properties with names LX_HOUR, LX_MIN, LX_SEC, and LX_FRAC have values that specify the position in the token string of the components that represent the hour, minute, second and fractional element of the time. Properties LX_HOURLEN, LX_MINLEN, LX_SECLEN, and LX_FRACLEN have values that specify the lengths of the respective strings. Using some simplifiying assumptions, an attempt is made to identify components that have inappropriate values. These are indicated by properties with names LX_HOURERR, LX_MINERR, and LX_SECERR.

The property LX_NOLOOKUP is added to time forms because they should not be looked up in the dictionary. This property is added during default (language-independent) processing based on the string type of the token.

Recognition of time expressions is supported currently in U.S., U.K. and Australian English.

Numbers

U.S., U.K. and Australian English use the period as the radix point and the comma as the group separator. Numbers containing commas and/or decimal points are identified and given a property with name LX_NUM. The value of property LX_NUMDEC is the position of the decimal point in the token string. The presence of certain errors in the form of a number is indicated by a property with name LX_NUMERR. This property is assigned if more than one decimal point is found, or if the placement of commas is not correct.

Every number is assigned an LX_NUMGROUP property by default. This indicates how many valid groupings of numbers are present in the number string. The comma and period act as group separators. For example, the string “1” has one grouping of numbers, “1,000” has two, “1,000.00” has three and so on.

The property LX_NOLOOKUP is added to numeric forms because they should not be looked up in the dictionary. This property is added during default (language-independent) processing base on the string type of the token.

Numeric Ranges

Any group of numbers separated by a hyphen and not recognized as a phone number is considered a range of numbers and is indicated by the property LX_NUMRANGE. The numeric portions of the token are analyzed as explained in “Numbers” on page 300.

Currently, when a token has the LX_NUMRANGE property it also has all the properties associated with the first number of the range. Since all numbers have the LX_NUM and LX_NUMGROUP properties, a token with the LX_NUMRANGE property will have them as well. For the properties which have relevant values, such as LX_NUMGROUP and LX_NUMDEC, the values will correspond to the first number of the range only. For example, for the number range

```
1,200.64-12,000,000,123,233
```

the LX_NUMDEC property would have the value 6 and the LX_NUMGROUP property would have the value 3 corresponding to the first number, even though the second number has five groups.

The property LX_NOLOOKUP is added to numeric range forms because they should not be looked up in the dictionary. This property is added during default (language-independent) processing, based on the string type of the token.

Recognition of numbers and number ranges is supported currently in U.S., U.K. and Australian English.

Roman Numerals

A token which represents a valid roman numeral is marked with the property LX_ROMNUM. Only Roman Numerals that are all in uppercase letters are identified.

Money

A token with an initial “$”, and containing numeric but not alphabetic characters is examined to determine if its structure is appropriate for a numeric monetary amount. If so, properties with names LX_MONEY and LX_USDOLLARS are assigned to the token. The numeric portion of the token is analyzed as explained in “Numbers” on page 300, so these forms may also have any of the properties associated with numbers.

Tokens representing cents are treated similarly, and are assigned properties LX_MONEY and LX_USCENTS. Processing of cents differs from dollars, however, in that a cent sign (“¢s.”) is identified by Simple Tokenization as a token separate from the associated number. Therefore, the two tokens have to be joined.

The property LX_CENTSERR is added if the number portion of the amount is separated from the cents sign by any blanks.

The property LX_NOLOOKUP is added to monetary forms because they should not be looked up in the dictionary. This property is added during default (language-independent) processing, based on the string type of the token.

Recognition of money expressions is supported currently in U.S. English only.

Telephone Numbers

Telephone numbers are recognized and identified by adding a property with name LX_PHONE. Examples of the forms recognized are:

```
794-8951
301/794-8951
301-794-8951
(301)794-8951
(301) 794-8951
```

If an area code is present, a property with name LX_PHONEAC is also added. The forms that contain parentheses require that four tokens be joined, since the tokenization process constructs four tokens for these forms: "(", "301", ")", and "794-8951."

The property LX_NOLOOKUP is added to telephone numbers because they should not be looked up in the dictionary. This property is added during default (language-independent) processing based on the string type of the token.

Recognition of telephone numbers is supported currently in U.S. English only.

Ordinal Numbers

Abbreviated forms of ordinal numbers are recognized and identified by adding LX_ORD to the feature list. The correct forms are:

```
x1st, x2nd, x2d, x3rd, x3d, yzth
```

where x is a (possibly empty) string of digits, not ending in 1, and z is any digit other than 1, 2, or 3. (For example, "12th" is correct, but "22th" is not, and "221st" is correct, but "211st" is not.) Observing these restrictions, the routine recognizes any number accepted by the number routine, including numeric ranges as described in "Numeric Ranges" on page 301.

The ordinal routine processes numeric ranges specially. Given a numeric range in which the second part does not have a ten's digit, the routine assumes that the ten's digit is the same as the ten's digit of the first part of the numeric range, if one exists. For example, the routine would assume that *21-3rd* means *the twenty-first through the twenty-third*, for the purposes of verifying the ordinal abbreviation. Contrast this with *11-3rd*, which would be marked as erroneous because the correct form is *11-3th* (*the eleventh through the thirteenth*).

Violation of these restrictions results in a property LX_ORDERR being added to the token. Since the numeric portion of an ordinal is usually an integer, the routine also adds the LX_ORDERR property when the numeric portion is not an integer.

The property LX_ORDD indicates that one of the less common forms such as "2d" or "3d" was used instead of "2nd" or "3rd".

The property LX_NOLOOKUP is added to ordinal forms because they should not be looked up in the dictionary. This property is added during default (language-independent) processing, based on the string type of the token.

Recognition of ordinal expressions is currently supported in U.S., U.K. and Australian English.

Abbreviations

Currently, no abbreviation processing is done as part of Lexical Analysis, beyond what is performed in Text Segment Identification.

Related discussions of abbreviations: Abbreviations are discussed in several sections other than this one. In order to have a complete view of how abbreviations are processed in Lexical Analysis, refer also to:

- Section “Contracted Auxiliary Verbs” on page 294
- Section “Alternatives with “/”” on page 298
- Section “Abbreviation Processing” on page 35

Property Inventory

The following properties may assigned to tokens by the Lexical Analysis function:

LX_CENTSERR Marked if the number portion of the amount is immediately followed (no intervening blanks) by the cents symbol.

LX_CNTRAUX A contracted auxiliary verb, for example, “’ll” (from “I’ll”).

LX_CNTRNEG A contracted negative, for example, “can’t”.

LX_CPXHEAD Pointer to a string that represents the rightmost element of a hyphenated or slashed form.

LX_CPXWORD Pointer to a string that represents the disambiguated form of a word which is not represented by the original text string (see “Hyphenated Forms” on page 296 and “Alternatives with “/”” on page 298).

LX_DATE A date, for example, 2/6/52, 11/16, 2/1

LX_DAY Position of the first character in a date that represents the day.

LX_DAYERR Present if the value representing the day in a date is invalid.

LX_DAYLEN Length of the string representing the day in a date.

LX_FRAC Position of the first character of the portion of a time that represents the fractional part of the time.

LX_FRACLEN Length of the fractional portion of a time.

LX_HOUR Position of the first character of the portion of a time that represents the hour.

LX_HOURERR Present if the value representing the hour is invalid.

LX_HOURLEN Length of the string representing the hour portion of a time.

LX_HYPHEN Present in forms containing a hyphen. Its value is the index position of the last hyphen found in the correct hyphenated word (whether it be the token string itself or the value of the CPXWORD property).

LX_INITCAP Present if the first letter of the word is uppercase and the rest of the word is lowercase. Added by default, if applicable, by Lexical Analysis.

LX_MIN Position of the first character of the portion of a time that represents the minute.

LX_MINERR Present if the value representing the minute is invalid.

LX_MINLEN Length of the string representing the minute portion of a time.

LX_MONEY Present on tokens that represent monetary amounts.

LX_MONTH Position in a date where the string representing the month begins.

LX_MONTHERR Present if the value of the month portion of a date is invalid.

LX_MONTHLEN Length of the string representing the month in a date.

LX_NOLOOKUP Present on tokens that should not be looked up in the dictionary. Added during default (language-independent) processing. To facilitate checking for this attribute, Lexical Analysis sets the appropriate bit in the string flags indicating that the token is not likely in the dictionary.

LX_NUM Present on numeric forms, including those containing decimal fractions and commas.

LX_NUMDEC Position of the decimal point in a numeric string.

LX_NUMERR Present on a numeric form if there is a problem with its form, for example, more than one decimal point, improper placement of commas.

LX_NUMGROUP Added to all number strings, this property indicates how many groups of numbers are distinguished in the string.

LX_NUMRANGE Present if two or more numbers are separated by a hyphen and the string has not been recognized as a phone number or date.

LX_ORD Present on ordinal numbers, for example, 1st, 10th, 3d, 3rd.

LX_ORDD Present if one of the less common ordinal forms (2d, 3d) is used instead of the more common forms (2nd, 3rd).

LX_ORDERR Error in form of an ordinal number.

LX_PHONE A telephone number.

LX_PHONEAC Present on a telephone number that includes an area code.

LX_ROMNUM Present if the string represents a valid roman numeral.

LX_SEC Position of the first character in a time that represents seconds.

LX_SECERR Present if the value representing seconds is invalid.

LX_SECLEN Length of the string representing seconds in a time.

LX_SLASH A form containing a slash, like “and/or” or “System/370”. Its value is the index position of the last slash appearing in the form.

LX_TIME A time expression, for example, 2:30, 72:14:03.789.

LX_USCENTS A monetary amount in U.S. cents.

LX_USDOLLARS A monetary amount in U.S. dollars.

LX_YEAR Position in a date where the string representing the year begins.

LX_YEARERR Present if the value of the year portion of a date is invalid.

LX_YEARLEN Length of the string representing the year in a date.

Finnish Language-Specific Processing

Hyphenation

Only algorithmic hyphenation is available in Finnish. Linguistic Tools allows words to be placed into a user dictionary to indicate the word's hyphenation points. If no user dictionary is active or the word is not found in the user dictionary, an algorithm is used to determine the hyphenation points for the word.

Lexical Analysis for Finnish

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see "Hyphenated Forms" on page 296). This processing includes dehyphenation.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Finnish the following language-specific processing is performed:

- No accent stripping is performed.

French (National and Canadian) Language-Specific Processing

Spell Verify

Prefix Processing

French allows the combination of prefixes with some words by joining them with an apostrophe. The rules enforced by Linguistic Tools governing this formation are:

- The prefix must be found in the list of valid prefixes shown below
- The word following the prefix and apostrophe must begin with the character "a", "e", "i", "o", "u", "y", "h", or some accented form of these letters
- The word following the prefix must not be found in the list of exception trailing words shown below
- The prefix must not be mixed case
- The word following the prefix must be verified as correct

If any of these conditions are not met, the prefixed word is considered incorrect.

The list of valid prefixes follows:

```
c
d
j
jusqu
l
lorsqu
m
n
puisqu
qu
quoiqu
s
t
```

Figure 21. French prefixes

The list of exception trailing words follows:

```
huit
huitain
huitains
huitaine
huitaines
huitième
huitièmes
onze
onzième
onzièmes
oui
yacht
yachts
yak
yaks
yole
yoles
yucca
yuccas
```

Figure 22. French exception trailing words

Any combination of all-lower-case, first-letter-only-upper-case, or all-upper-case prefix and trailing word are allowed. Mixed case prefixes are always considered incorrect. Mixed case trailing words are valid if found in a dictionary.

Accent Removal

National French allows the user to optionally remove accents from upper case letters. This removal can be done on all letters in an all upper case word, or on the first letter of a capitalized word. Linguistic Tools functions, therefore, have special processing for these cases.

For National French, each input word which is all upper case is compared as is to words from the dictionary. If that match fails, the accents are removed from the dictionary word and the comparison is done again. Each input word which is first letter only upper case is compared as is to words in the National French dictionary. If that match fails, the accent is removed from the first letter of the dictionary word and the comparison is done again. This processing allows accents to be retained or removed from all upper case letters by the user.

No accent removal processing is done for Canadian French.

Spell Aid

In spell aid when the aid candidate words are converted to upper case to match the input word's case, all accents are removed from the letters.

Lexical Analysis for French

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see "Hyphenated Forms" on page 296). This processing includes dehyphenation.

Prefixes

Simple tokens that begin with one of the prefixes in Figure 21 on page 306 are split into two tokens. The first token contains the prefix, including the final apostrophe. The property LX_PREFIX is attached to this token. The second token contains the rest of the original token.

Note: Lexical Analysis does not check that the letters following a prefix are valid; that is the responsibility of the caller.

Property Inventory

This is an inventory of properties assigned to tokens by the Lexical Analysis function:

LX_PREFIX A prefix ending in "".

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For French the following language-specific processing is performed:

- All accents are removed.

National German Language-Specific Processing

Spell Verify

Compound words

German is a language which allows compound words. Its dictionary, therefore, has BOFA flags and the compound word processing is performed, as needed, when verifying words. Only words five characters or longer are processed as potential compounds.

German words which verify by means of the decompounding mechanism have an additional restriction that other Germanic languages do not have: if the last component of the compound word is in the dictionary in upper case, the compound word must begin with an upper- case letter, for example the word “bundes” is in the dictionary in lower case and the word “Republik” is in the dictionary in upper case; therefore, the word “Bundesrepublik” should verify, but “bundesrepublik” should not.

Elliptic enumeration

Elliptic enumeration allows words coded with BOFA “F” (which can only occur in the front or middle of a compound) to verify if they are followed by a hyphen.

Umlaut Translation

In words with an upper case first letter, or words which are all upper case, umlauted letters may optionally change in spelling. This spelling change involves changing the Ä, Ö or Ü to its upper case, non-umlauted form (A, O, or U) and inserting an “e” as the next letter. The inserted “e” will be lower case if the word is only first letter upper case, or will be upper case if the entire word is upper case. For example, “Übersetzung” may be spelled as is or may be changed to “Uebersetzung.” “ÜBERSETZUNG” may be spelled as is or may be changed to “UEBERSETZUNG.” The change in spelling must be made on all or no umlauted letters in all upper case words. The change is never allowed on lower case letters. It is recognized that in the cases in which such a letter group occurs more than once and not all of the letter groups were changed, then the word will not be found either in the dictionary nor as a compound.

Eszett processing

The *Eszett (ß)* character is a lower-case character only. When a word that contains an *Eszett* is written with all upper-case letters, the *Eszett* is normally written as “SS.” Therefore, the access code must perform special processing for Spell Verify and Spell Aid.

In Spell Verify, a word in all upper-case letters that contains an *Eszett* should be highlighted as being misspelled. Additionally, a word that normally contains an *Eszett* but that is written with all upper-case letters with “SS” in place of the *Eszett* should verify as being correctly spelled.

Spell Aid

Case-sensitive processing

Special case-sensitive processing has been added to Spell Aid for German. If the input word is capitalized (i.e. begins with an upper-case letter), only candidates that are capitalized in the dictionary will be returned as candidates from Spell Aid, unless the "quality-ranking" (method of determining "closeness" and therefore suitability as a candidate) of an all-lowercase word is such that the word is a very close match.

If the misspelled word begins with a lower-case letter, only words that begin with a lower-case letter in the dictionary will be offered in Spell Aid, unless a word beginning with an upper-case letter has a very high "quality-ranking."

Eszett processing

Spell aid does not return any candidates in all upper-case letters that also contain an *Eszett (ß)*. Instead, it replaces the *Eszett* with "SS."

Hyphenation/Dehyphenation

The hyphenation of German words may result in spelling changes. The two cases of this are "ck" changed to "kk", and double consonants in compound words changed to triple consonants. If a word is hyphenated such that the break point is between a letter "c" and a letter "k", then the spelling is changed to "kk". This results in the first part of the word ending a line with "k-" and the next line having the next part of the word beginning with a "k". This "ck" to "kk" spelling change must be removed through dehyphenation before the word can be passed to any Linguistic Tools function, and when the hyphenation break point is removed from the word. Note that this spelling change must only be performed within a word, but never at component junctions of a compound word.

The other case of spelling change during hyphenation involves contiguous consonants in compound words. When two components of a compound word are joined, and the last two letters of the preceding component are identical consonants, and the first letter of the following component is the same consonant, and the first letter consonant is followed by a vowel or the first letter consonant is a "t" followed by an "h", then one of the three identical consonants is dropped. For example, when "Schiff" and "Fahrt" are joined to form a compound word, then one "f" is dropped and the final spelling is "Schiffahrt". However, when such a word is hyphenated between these components, then the dropped consonant must be returned to the word. This results in the first part of the word ending a text line with two identical consonants and a syllable hyphen, and the the next text line having the next part of the word beginning with a the same consonant. For example, "Schiffahrt" when hyphenated between the word components is spelled "Schiff-" with "fahrt" on the next line. This spelling change must be removed through dehyphenation before the word can be passed to any Linguistic Tools function, and when the hyphenation break point is removed from the word.

Lexical Analysis for German

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see “Hyphenated Forms” on page 296). This processing includes dehyphenation. Lexical Analysis for German supports spelling changes that occur in dehyphenation.

Note: A German dictionary must be the first dictionary activated for spelling changes during dehyphenation to be made properly.

Numbers and Numeric Ranges

The processing of numbers and numeric ranges uses a method analogous to the one used for English (see “Numbers” on page 300) except that the group delimiter is the period and the radix point is the comma. Number processing currently only applies to National German, not to Swiss German.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For National German the following language-specific processing is performed:

- *Eszett (ß)* is replaced by “ss”. Note that Eszett rendered as “sz” will not be replaced by “ss”.
- The umlauted letters ä, ö, and ü will be rewritten as “ae”, “oe”, and “ue”, respectively.
- All other accents will be removed.

Swiss German Language-Specific Processing

Swiss German is similar to national German: its dictionary has BOFA flags, and the compound word processing is performed, as needed, when verifying words. Also, **elliptic enumeration** is allowed. The languages are processed identically, except for the following:

Umlaut Translation

In **Spell Aid**, the candidates with uppercase umlauts will be presented with the "correct" spelling (i.e., no changes to remove the umlauts). During Spell Aid, a check is made to determine if an umlauted letter change in spelling may have occurred on the first letter of the word. If it may have occurred, then Spell Aid is obtained on the word using an umlauted upper case letter as the first letter and deleting the inserted "e". This change is not made with words beginning with "Aero" because those letters are a common prefix used in Swiss German.

Eszett processing

The *Eszett* character is not used in Swiss German, but is replaced with "ss." This change has been made in the dictionary and does not affect any processing code or logic.

Triple-s elision

In compound words, the triple letter elision to two letters does not apply to the consonant s. Therefore, three "s"s followed by any letter, including a vowel, are never reduced to two.

Lexical Analysis in Swiss German

Numbers and Numeric Ranges

Number processing currently is not available for Swiss German.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Swiss German the following language-specific processing is performed:

- The umlauted letters ä, ö, and ü are rewritten as "ae", "oe", and "ue", respectively.
- All other accents are removed.

Greek Language-Specific Processing

Greek Character Set

The Greek character set (host code page 875, PC code page 869) includes Roman letters, Greek letters, a few accented Roman letters, numerals, and punctuation symbols. See “Greek-Specific Codes” on page 257 for details.

Spell Verify

Accent Removal

In order to match an uppercase word without accents, which is a common occurrence in Greek, the following procedure is used:

1. Each input word in all upper case is compared as is to words in the dictionary.
2. If that match fails, the accents are removed from the dictionary word and the comparison is repeated.

This processing allows accents to be removed from all upper case letters by the user.

Double Accent Words

According to the monotonic accent system, Greek words of more than one syllable typically bear only one written accent appearing in the syllable which is stressed in pronouncing the word. In certain contexts, however, words which normally have the accent on the antepenultimate (third syllable from the end) may have a second accent added to the final syllable.

In order to verify these words, Linguistic Tools first removes the final-syllable accent. It then attempts to match the remaining string. For single-word format input, no further checking is done; if the word (minus the second accent) matches a dictionary word, the double-accent form is assumed to be correct. With block format input, Linguistic Tools will examine the word following the double accent word; for the double-accent word to verify, the following word must be a member of the closed set of enclitic forms which condition the occurrence of the double accent.

Because the Spell Aid function accepts only single-word format input, it is impossible to check whether the occurrence of a second accent on an input word is contextually correct. Linguistic Tools assumes that it is, and returns aid candidates with double accents.

Sigma Final Processing

Greek has a sigma final letter (ς) that can only validly appear as the last letter in a word. However, there is no upper case sigma final letter in the character set. When a sigma final letter should appear as the last letter in a word and the last letter is upper case, an upper case sigma letter is used.

If a Greek word is input to Linguistic Tools with its last letter an upper case sigma, Linguistic Tools recognizes that a lower case sigma final letter must be used when

comparing to lower case dictionary words. Also, when Spell Aid candidates are converted to upper case to match the input word's case, sigma final letters are converted to upper case sigma letters.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Greek, the following language-specific processing is performed:

- Last-letter uppercase sigma is converted to (lowercase) sigma final.
- All accents are removed.

Hungarian Language-Specific Processing

Hyphenation/Dehyphenation

Only algorithmic hyphenation is available in Hungarian. Linguistic Tools allows words to be placed into a user dictionary to indicate the word's hyphenation points. If no user dictionary is active or the word is not found in the user dictionary, an algorithm is used to determine the hyphenation points for the word.

The hyphenation of Hungarian words may result in spelling changes. These transformations include:

hyphen context	transformation
ccs	cs-cs
ddzs	dzs-dzs
ddz	dz-dz
ggy	gy-gy
lly	ly-ly
nny	ny-ny
ssz	sz-sz
tty	ty-ty
zzs	zs-zs

Dehyphenation reverses these transformations.

Lexical Analysis for Hungarian

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see "Hyphenated Forms" on page 296). This processing includes dehyphenation.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Hungarian the following language-specific processing is performed:

- No accent stripping is performed.

Icelandic Language-Specific Processing

Spell Verify

Compound Words

Icelandic is a language which allows compound words. Its dictionary therefore has BOFA flags and the compound word processing is performed, as needed, when verifying words. Only words five characters or longer are processed as potential compounds.

Icelandic allows a single one letter prefix to be attached to any word or any component of a compound word. This letter is “o” acute, “ó”. The compound word processing handles this prefix letter. Therefore, only words five characters or longer which are processed by the compound word processing will be verified with such a prefix.

Elliptic enumeration

Elliptic enumeration allows words coded with BOFA “F” (which can only occur in the front or middle of a compound) to verify if they are followed by a hyphen.

Lexical Analysis for Icelandic

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see “Hyphenated Forms” on page 296). This processing includes dehyphenation.

Numbers and Numeric Ranges

The processing of numbers and numeric ranges uses a method analogous to the one used for English (see “Numbers” on page 300) except that the group delimiter is the period and the radix point is the comma.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Icelandic the following language-specific processing is performed:

- No accent stripping is performed.

Italian Language-Specific Processing

Spell Verify

Prefix Processing

Italian allows the combination of prefixes with some words by joining them with an apostrophe. The rules enforced by Linguistic Tools governing this formation are:

- The prefix must be found in the list of valid prefixes shown below
- The word following the prefix and apostrophe must begin with the letter “a”, “e”, “i”, “o”, “u”, or “h”. The trailing word’s first letter must be an “i” for some prefixes.
- The word following the prefix must not be found in the list of exception trailing words shown below
- The prefix must not be mixed case
- The word following the prefix must be verified as correct

If any of these conditions are not met then the prefixed word is considered incorrect.

The list of valid prefixes follows. Some valid prefixes may only be followed by trailing words that begin with an “i”. These prefixes are marked as “Only I”.

```
agl       Only I
alcun
all
anch      Only I
bell
buon
c         (currently not restricted, eventually Only I and E)
ch        Only I
cogl      Only I
coll
com
cos
d
dagl      Only I
dall
degl      Only I
dell
gl        Only I
grand
l
m
n
negl      Only I
nell
neanch    Only I
nessun
nient
pover
qual
qualcos
qualcun
quand
quant
quegl     Only I
quell
quest
s
sant
senz
sugl      Only I
sull
t
tant
tutt
un
v
```

Figure 23. Italian Prefixes

At present, Italian has no exception trailing words.

Any combination of all-lower-case, first-letter-only-upper-case, or all-upper-case prefix and trailing word are allowed. Mixed case prefixes are always considered incorrect. Mixed case trailing words are valid if found in a dictionary.

Trailing Apostrophe Processing

In common usage of Italian, the user is allowed to replace an accented last letter of a word with the unaccented form of the letter followed by an apostrophe. Since only the last letters of Italian words are accented, this allows the user to key accented letters without having the accented letters available on the keyboard. This usage of the trailing apostrophe causes special processing in Linguistic Tools.

Input words are compared to dictionary words as is for Italian. If this match fails, and the last character of the input word is lower case, then the trailing character is checked. If this character is an apostrophe, any accent is removed from the last letter of the dictionary word and the words are compared again. This allows the input word's unaccented last letter to match the dictionary word's accented last letter. If the trailing character is not an apostrophe, or the dictionary word's last letter does not have an accent, or the dictionary word's unaccented letter does not match the input word's last letter, the words do not match.

Lexical Analysis for Italian

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for National Portuguese (see "Hyphenated Forms" on page 340). This processing includes dehyphenation.

Numbers and Numeric Ranges

The processing of numbers and numeric ranges uses a method analogous to the one used for English (see "Numbers" on page 300) except that the group delimiter is the period and the radix point is the comma.

Prefixes

Simple tokens that begin with one of the prefixes in Figure 23 on page 318 are split into two tokens. The first token contains the prefix, including the final apostrophe. The property LX_PREFIX is attached to this token. The second token contains the rest of the original token.

Note: Lexical Analysis does not check that the letters following a prefix are valid; that is the responsibility of the caller.

Property Inventory

This is an inventory of properties assigned to tokens by the Lexical Analysis function:

LX_PREFIX A prefix ending in "".

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Italian the following language-specific processing is performed:

- All accents are removed.

Japanese Language-Specific Processing

Linguistic Overview

Written Japanese includes three types of characters:

- Kanji
- Hiragana
- Katakana

The two last are known collectively as *kana*.

Kanji are ideographic characters originally borrowed from Chinese (the name means literally “Chinese characters”). These are used to write such things as personal and place names, noun stems, verb stems, etc. Because of the number of *kanji* used in Japanese, these must be represented as a double-byte character set (DBCS). Even though many of the same characters are used in both Chinese and Japanese, they are represented differently in the Chinese and Japanese code pages.

Hiragana are syllabic characters used to write verb endings, grammatical particles, and native Japanese words for which there is no *kanji* representation or for which the writer does not know or wish to use the *kanji*. The number of *hiragana* is much smaller than the number of *kanji*. *Hiragana* can be represented as single-byte or double-byte characters.

Text data entry in Japanese is generally done with the user typing in *hiragana* - some systems permit a user to type in *romaji* (Roman or Latin alphabet) - and converting to *kanji* combined with *hiragana*, the way in which normal Japanese text is written. Because of the relatively high degree of homonymy (different words with the same pronunciation) in Japanese, it is possible for the same *hiragana* sequence to correspond to several *kanji/hiragana* combinations. The *kana*-to-*kanji* conversion function may make use of frequency, context, stored user preferences, or other kinds of information to decide which of the possible *kanji* to offer first.

Katakana are another set of syllabic characters used to write non-Japanese words in a more or less phonetic representation. There is a single-byte character set (SBCS) for *katakana*; there is also a double-byte set.

A Japanese text may contain a mixture of double-byte and single-byte strings. Care must be taken in specifying string lengths; in Linguistic Tools all lengths are given in bytes, not characters.

Normal Japanese text consists of a string of DBCS characters without any special delimiter between words. Word isolation strategies based on European-language writing conventions will not work for Japanese. Linguistic Tools does special processing to isolate words in Japanese text. Japanese addenda dictionaries can be used to help in this processing for supported functions. Details are described below.

Currently, morphological identification and generation of inflected forms are not supported in Japanese, and only a limited number of words are available in a separate Japanese synonym dictionary.

Because Linguistic Tools does not have a Japanese phonetic algorithm, the text extraction functions will not return phonetic keys for Japanese.

Spell Verify and Spell Aid are not (yet) supported for Japanese, because of lack of agreement among Japanese computational linguists as to what is required.

No hyphenation/dehyphenation processing is done for Japanese, because hyphenation does not occur in Japanese.

Supported Functions

Only the following functions are supported for Japanese:

- Activate Dictionary
- Deactivate Dictionary
- Simple Tokenization
- Text Segment Identification
- Lexical Analysis
- Extract Keyword
- Extract Query Term
- Synonym aid support
- Create Addenda Dictionary
- Add Word with Data
- Remove Word from Addenda
- List Addenda Words
- Save Addenda
- Look up Word
- Poe Utilities
  - NlpSetDictDef
  - NlpQrySearchPath
  - NlpSetSearchPath
  - NlpFindDicts
  - Poe Token List Utilities

All other functions are not supported for Japanese. However, Japanese code pages can be used to process SBCS Latin I characters.

Simple Tokenization for Japanese

Because Linguistic Tools' **Simple Tokenization** function determines word boundaries via simple character examination, it is unable to determine word boundaries in double-byte text which does not use blanks to separate words. However, simple tokenization is useful for separating single-byte text from double-byte text in Japanese. If more accurate word boundaries are required, the the application should call the **Lexical Analysis** function.

When **Simple Tokenization** is given Japanese text, it will separate single-byte text from double-byte text, determine word boundaries in the single byte text (as described in "Definition of a Word in SBCS Text" on page 80), isolate shift/in and shift/out characters in host DBCS text and the double-byte punctuation characters listed in Table 2 on page 33 and Table 3 on page 34. New lines will not necessarily cause a break; text will be returned as joined tokens if no delimiters are found.

For example, the input string “Tokenizing!!?) a string” would generate six tokens:

```
"Tokenizing"
"!!"
"?"
")"
"a"
"string"
```

In this example the input string contains only SBCS characters. If, instead, the first 5 letters of “Tokenizing” and the second exclamation point were DBCS characters, the string would be analyzed as 8 tokens:

```
"Token"
"izing"
"!"
"!"
"?"
")"
"a"
"string"
```

Lexical Analysis for Japanese

Lexical Analysis uses the output of **Text Segment Identification** to isolate words and return their part(s) of speech; information from both Japanese system and addenda dictionaries may be used.

Given a token list of a sentence as input, **Lexical Analysis** will isolate words within the sentence and add a part of speech property. Characters will be passed in the code page of the caller. The input token list will be joined and split where **Lexical Analysis** identifies a word boundary. **Lexical Analysis** will also add at least one LX_JPOS property to each token. LX_JPOS property values are listed in table Table 32 on page 325. LX_JPOS property values have the type LX_PVAL_USHORT.

Note that the shift-out and shift-in characters in host DBCS text are treated as separate tokens distinct from the text strings which they delimit. PC DBCS text does not contain shift-out or shift-in characters. Thus, the number and types of tokens returned for the same DBCS text will differ from the host to the PC.

Furthermore, if a line of host DBCS text begins with blanks (i.e., shift-out followed by DBCS blanks), these will be returned by **Lexical Analysis** as a separate token indicating just the number of bytes occupied by these DBCS blanks. For block input on the PC, this information will be returned as a component of a joined token, the first component of which will be the last text or punctuation token from the previous line of the input text.

Synonym Aid Support

A list of synonyms is returned for the specified headword. The headword must be in the root form, and no morphological support is provided for Japanese.

Keyword and Query Term Extraction for Japanese

The **Extract Keywords** and **Extract Query Terms** functions use the output of **Lexical Analysis** to isolate words from a block of text. For the text extraction functions, the application must set a flag indicating the part(s) of speech desired. Since **Lexical Analysis** for Japanese returns part of speech information, only words which have been tagged by **Lexical Analysis** as having the part(s) of speech specified by the filtering options will be returned. Part of speech information is returned in terms of the condensed POS, rather than the JPOS (see "Japanese Part of Speech (POS) Codes" for details). Note that "unknown" words (those that cannot be found in the dictionary) are returned with part of speech LX_EKW_UNKNOWN. For part of speech filtering, this category is included with the nouns.

While **Lexical Analysis** returns shift-out and shift-in on the host as separate tokens from the DBCS text string

```
token 1 – SO
token 2 – DBCS_text
token 3 – SI
```

the text extraction functions return the DBCS text string enclosed by shift-out and shift-in

```
token 1 – SO_DBCS_text_SI
```

Thus the host DBCS token is two bytes longer than the corresponding DBCS token on the PC, but the number of tokens returned by the text extraction functions is the same on host and PC.

JAIRS Part of Speech Filtering

For Japanese only, there is an additional part of speech filtering option in the text extraction functions. This is to return only the parts of speech identified for JAIRS - words whose JPOS value is one of the following: 13, 18, 19, 23, 24, 97, 102, 103, 104, 106, 107.

Japanese Part of Speech (POS) Codes

There are three kinds of Part of Speech (POS) codes used by Japanese functions for the Linguistic Tools:

- The **JPOS Code** is used to specify the most detailed POS in terms of internal state transition in Japanese functions. Words in the Function Word Dictionary are specified by this code. This code is also used to return words from Japanese Lexical Analysis (use token list utility **NlpGetFirstProperty()** or **GetNextProperty()**).
- The DBPC (Double-Byte Parsing Code), also known as the **Macro POS Code**, is used to specify the part of speech of independent words (stem) in the system dictionary and in user-created addenda dictionaries.
- The **Condensed POS Code** is used for words returned from the **Extract Keywords** and **Extract Query Terms** functions.

The following table shows how the three kinds of Japanese POS Codes are related to each other:

Table 32 (Page 1 of 3). Japanese Part of Speech Codes (JPOS, POS, DBPC/Macro POS).			
JPOS Code	**Part of Speech Description**	**DBPC/Macro POS Code**	**Condensed POS Code**
1	Stem of KA-row-U-dropping Conjugation Verbs (KA-gyou-5-dan-katsuyou Doushi Gokan)	1	X'20'
2	Stem of KA-row-U-dropping Conjugation Verbs Special (KA-gyou-5-dan-katsuyou Doushi Gokan)	2	X'20'
3	Stem of GA-row-U-dropping Conjugation Verbs (GA-gyou-5-dan-katsuyou Doushi Gokan)	3	X'20'
4	Stem of SA-row-U-dropping Conjugation Verbs (SA-gyou-5-dan-katsuyou Doushi Gokan)	4	X'20'
5	Stem of TA-row-U-dropping Conjugation Verbs (TA-gyou-5-dan-katsuyou Doushi Gokan)	5	X'20'
6	Stem of NA-row-U-dropping Conjugation Verbs (NA-gyou-5-dan-katsuyou Doushi Gokan)	6	X'20'
7	Stem of BA-row-U-dropping Conjugation Verbs (BA-gyou-5-dan-katsuyou Doushi Gokan)	7	X'20'
8	Stem of MA-row-U-dropping Conjugation Verbs (MA-gyou-5-dan-katsuyou Doushi Gokan)	8	X'20'
9	Stem of RA-row-U-dropping Conjugation Verbs (RA-gyou-5-dan-katsuyou Doushi Gokan)	9	X'20'
10	Stem of WAA-row-U-dropping Conjugation Verbs (WAA-gyou-5-dan-katsuyou Doushi Gokan)	10	X'20'
11	Stem of RU-dropping Conjugation Verbs (1-dan-katsuyou Doushi) which become substantives	11	X'20'
12	Stem of RU-dropping Conjugation Verbs (1-dan-katsuyou Doushi) which do not become substantives	12	X'20'
13	Stem of Noun-type SA-row-irregular Conjugation Verbs (Meishi-kei-SA-hen Doushi Gokan)	13	X'20'
14	Stem of SURU-type SA-row-irregular Conjugation Verbs (SURU-gata-SA-hen Doushi Gokan)	14	X'20'
15	Stem of ZURU-type SA-row irregular Conjugation Verbs (ZURU-gata-SA-hen Doushi Gokan)	15	X'20'
16	Kanji part of KA-row irregular Conjugation Verbs (KA-hen Doushi Kanji-bu)	16	X'20'
17	Stem of Adjectives (Keiyoushi Gokan)	17	X'10'
18	Stem of Pseudo Adjectives (Keiyou Doushi)	18	X'10'
19	Nouns (Meishi)	19	X'40'
20	Pseudo Adjectives (Rentaishi)	20	X'10'
21	Adverbs (Fukushi)	21	X'08'
22	Conjunctions (Setsuzokushi), Interjections (Kantoushi)	22	X'03'
23	Prefixes (Settouji)	23	X'40'
24	Suffixes (Setsubiji)	24	X'40'
25	(Flag which indicates possibility of the start of a phrase)		
26	-NAI type Negative inflection of U-dropping Conjugation Verbs		X'80'
27	-U type Negative inflection of U-dropping Conjugation Verbs		X'80'
28	-MASU type Conjunctive inflection of U-dropping Conjugation Verbs		X'80'
29	-TA type Conjunctive inflection of U-dropping Conjugation Verbs		X'80'
30	-DA type Conjunctive inflection of U-dropping Conjugation Verbs		X'80'
31	Present/Past inflection of U-dropping Conjugation Verbs		X'80'
32	Conditional/Command inflection of U-dropping Conjugation Verbs		X'80'
33	Conditional inflection of RU-dropping Conjugation Verbs		X'80'
34	Command inflection of RU-dropping Conjugation Verbs		X'80'
35	-SERU type Negative inflection of SA-row-irregular Conjugation Verbs		X'80'
36	-ZU type Negative inflection of SA-row-irregular Conjugation Verbs		X'80'
37	-NAI type Negative/Conjunctive inflection of SA-row-irregular Conjugation Verbs		X'80'
38	Special type inflection of SA-row-irregular Conjugation Verbs (“SUBEKI”, “ZUBEKI”)		X'80'
39	Negative inflection of KA-row-irregular Conjugation Verbs		X'80'
40	Conjunctive inflection of KA-row-irregular Conjugation Verbs		X'80'
41	-TA type Conjunctive inflection of Adjectives		X'80'
42	-NAI type Conjunctive inflection of Adjectives		X'80'
43	Present/Past inflection of Adjectives		X'80'
44	Conditional inflection of Adjectives		X'80'
45	-TA type Conjunctive inflection of Pseudo Adjectives		X'80'
46	-NAI type Conjunctive inflection of Pseudo Adjectives		X'80'
47	-NARU type Conjunctive inflection of Pseudo Adjectives		X'80'

Table 32 (Page 2 of 3). Japanese Part of Speech Codes (JPOS, POS, DBPC/Macro POS).

JPOS Code	Part of Speech Description	DBPC/Macro POS Code	Condensed POS Code
48	Present inflection of Pseudo Adjectives		X'80'
49	Past inflection of Pseudo Adjectives		X'80'
50	Conditional inflection of Pseudo Adjectives		X'80'
51	Stem of Negative Auxiliary Verbs “NAI” and Wishing Auxiliary Verbs “TAI”		X'80'
52	Present/Past form of Presumptive Auxiliary Verbs “RASHII”		X'80'
53	Stem of Aspect Auxiliary Verbs “SOUDA”		X'80'
54	Stem of Hearsay Auxiliary Verbs “SOUDA”		X'80'
55	Stem of Simile Auxiliary Verbs “YOUDA”		X'80'
56	-NAI type Conjunctive inflection of Conclusion Auxiliary Verbs “DA”		X'80'
57	Past inflection of Conclusion Auxiliary Verbs “DA”		X'80'
58	Negative form of Polite Auxiliary Verbs “MASU”		X'80'
59	Conjunctive form of Polite Auxiliary Verbs “MASU”		X'80'
60	Present/Past form of Polite Auxiliary Verbs “MASU”		X'80'
61	Conjunctive form of Polite Conclusion Auxiliary Verbs “DESU”		X'80'
62	Present/Past form of Polite Conclusion Auxiliary Verbs “DESU”		X'80'
63	Present/Past form of Past/Perfect Tense Auxiliary Verbs “TA”/“DA”		X'80'
64	Conjunctive form of Negative Auxiliary Verbs “NU”		X'80'
65	Present/Past form of Negative Auxiliary Verbs “NU”		X'80'
66	Supposition/Will Auxiliary Verbs “U“/“YOU” and their Negative Auxiliary Verbs “MAI”		X'80'
67	Conjunctive form of Classical Auxiliary Verbs “BESHI”		X'80'
68	Past form of Classical Auxiliary Verbs “BESHI”		X'80'
69	Conjunctional Particles “TE”/“DE” followed by “HOSHII”		X'80'
70	Conjunctional Particles “TE” not followed by “HOSHII”		X'80'
71	“RI“ part of Conjunctional Particles “TARI”/“DARI”		X'80'
72	Conjunctional Particles “NAGARA”/“TSUTSU”		X'80'
73	Conjunctional Particles “BA”/“GA”/“TO”		X'80'
74	Conjunctional Particles “SHI”		X'80'
75	Case Particles “GA”		X'80'
76	Case Particles “NO”		X'80'
77	Case Particles “WO”		X'80'
78	Case Particles “NI”		X'80'
79	Case Particles “NI” (for example, means assumption)		X'80'
80	Case Particles “HE”		X'80'
81	Case Particles “TO”		X'80'
82	Case Particles “TO” (quotation)		X'80'
83	Case Particles “YORI”		X'80'
84	Case Particles “KARA”		X'80'
85	Relation Particles “HA”/“SAE”/“SURA”/“SHIKA”		X'80'
86	Relation Particles “MO”/“KOSO”		X'80'
87	Sub Particles “DAKE”/“NOMI”/“BAKARI”/“GURAI”/“MADE”/“NADO”		X'80'
88	Parallel/Final Particles “KA”		X'80'
89	Parallel Particles “YARA”/“DANO”/“NARI”		X'80'
90	Final Particles “NA”		X'80'
91	Conjunctional Particles “KARA”		X'80'
92	Semi-Substantive Particles “NO” and “NO” part of Conjunctional Particles “NONI” etc.		X'80'
93	“GOZAI” part of special expression “GOZAIMASU”		X'80'
94	Formal Noun, Pronoun		X'04'
95	Special Command Form “NASAI”/“IRASSHAI”/“KUDASAI”		X'80'
97	Unknown words	33	X'FF'

JPOS Code	Part of Speech Description	DBPC/Macro POS Code	Condensed POS Code
98	Quotations	34	X'80'
99	Starting of a phrase		
100	Words which always end phrases		X'80'
101	Case Particles "DE"		X'80'
102	Nouns which do not form compound words	25	X'40'
103	Numerals	35	X'FF'
104	Proper Nouns (Koyuu Meishi)	26	X'40'
105	Pronouns (Daimeishi)	27	X'04'
106	Prefixes of numerals	28	X'40'
107	Suffixes of numerals	29	X'40'
108	Adverbs (Fukushi) which become predicates followed by "DA"	30	X'08'
109	Adverbs (Fukushi) followed by Particles "NO" and "HA"	31	X'08'

Table 32 (Page 3 of 3). Japanese Part of Speech Codes (JPOS, POS, DBPC/Macro POS).

Addenda Dictionary Support

A Japanese addenda dictionary may be used to alter the parsing of Japanese words by adding words and a language-specific DBCS parsing code (DBPC). All parsing codes are listed as Macro POS Codes in "Japanese Part of Speech (POS) Codes" on page 324.

Japanese addenda dictionaries may be used to supplement or override the information in the Japanese system dictionary by adding the following:

1. Words not found in the Japanese system dictionary and their parsing codes
2. Words found in the Japanese system dictionary with additional parsing codes
3. Words that are found in the Japanese system dictionary with replacement parsing codes.

In addition to altering the parsing of words, Japanese addenda may also be used to store user data associated with the dictionary words.

Japanese addenda dictionaries have the same basic format as the addenda dictionaries for all other languages. However, because the synonym and morphology functions are not supported for Japanese, these fields in the addenda dictionary format are not used by Linguistic Tools for Japanese processing.

Korean Language-Specific Processing

The native Korean writing system uses component symbols called *jamo* to compose syllabic *hangeul* symbols, which are represented in the computer by a double-byte character set (DBCS). Historically Korean also used Chinese characters (*hanja*) in writing, but these are rare in contemporary text. When they do occur, it is generally as proper names.

Unlike Chinese and Japanese, Korean uses blanks as word separators, so that word isolation can be done in the same fashion (taking into account the DBCS punctuation) as in European languages. Hyphenation is not used in Korean.

Morphological information is available in the Korean dictionary, so the **Morphological Identification** function is supported for Korean. However, the results will be slightly different from those for a European language (see “Morphological Identification for Korean” on page 329 for details). The **Generate Inflected Forms** function has not (yet) been implemented for this language. Neither have the spell checking functions.

Synonym data are not (yet) available for Korean.

Only the following functions are supported for Korean:

- Initialize Linguistic Service
- Terminate Linguistic Service
- Activate Dictionary
- Deactivate Dictionary
- Simple Tokenization
- Text Segment Identification
- Lexical Analysis
- Morphological Identification
- Create Addenda Dictionary
- Add Word with Data
- Remove Word from Addenda
- List Addenda Words
- Save Addenda
- Look up Word
- Extract Keyword
- Extract Query Term
- Poe utilities
  - NlpSetDictDef
  - NlpQrySearchPath
  - NlpSetSearchPath
  - NlpFindDicts
  - Poe Token List Utilities

All other functions are not supported for Korean.

Morphological Identification for Korean

Morphological Identification for Korean will produce the output described in "Morphological Identification" on page 167 except for the following deviations:

- The input word passed can be a mixed string and must be in the host Korean combined code page 933 or PC combined code page 944. All shift-in and shift-out characters found in the input will be removed. All other single byte characters, in both host and PC code pages, will be converted into double byte characters prior to searching the dictionary, since the dictionary does not contain any single byte characters.
- Lemmas (base forms) will be ranked by length, with the longest returned first. A 'K' in the PCODE for a particular lemma indicates that this was arrived at by (educated) guesswork. These lemmas will be ranked below all lemmas found in the dictionary, regardless of length. Within the set of lemmas with 'K' PCODES, ranking will be by length.
- The input word returned in the reply area is an all DBCS string to show exactly how the word was looked up in the dictionary. Note that for host code pages the DBCS string will be surrounded by a single set of SBCS shift-out/shift-in characters.
- The POS returned will be the condensed POS (same as for Latin languages) and will be derived from the last EBCDIC value of the PCODE.
- The DBPC field will contain a non-zero value. Possible values are described in "Korean Double-Byte Parsing Codes (DBPCs)" on page 331.
- No alias processing is performed.
- No inflective classification (bit mask) information is returned.
- Output cannot be used to generate inflected forms because the the Generate Inflected Forms function is not supported for Korean.
- The **Morphological Identification** function will search all dictionaries in the input dictionary list until it finds a word that can qualify as the lemma of the input word. The lx_dict_found_in flag will return the token of the first dictionary in which such a word was found.
- For Korean words that contain non-hangeul characters, the lemma will be assumed to be the characters up to the first hangeul character. If no hangeul characters are present, the entire word will be returned as the lemma. In these cases a special return code, LX_RC_OK_NHS (Non-Hangeul Stem) will be passed back to the application. The DBPC, PCODE, and part of speech fields will be zero. For applications which are only looking for the lemma information, this return code can be treated the same as LX_RC_OK.

In Korean, no hyphenation marks are used to specify whether or not a word at the end of a line continues onto the following line. To accomodate this, the application may put the character string found at the end of a line in one data element and the character string at the beginning of the following line in the next data element. The lx_input_f field on the first data element should be marked with the LX_ELMT_JOIN and LX_ELMT_NO_HYPHEN flags. If identified in this manner, the service will attempt to determine if the characters are to be joined as one word or if the characters represent two separate words. The Linguistic Tools will try to find the lemma of the combined characters first. If found, the appropriate return code will be put in the lx_rc field of both elements. If the lemma could not be determined, the last element will be marked with LX_WORD_NOT_FOUND. Next, the first

element will be used to determine the lemma and its return code will be put into its lx_rc field. All lx_rc fields will be filled in. The word that was used can be determined by examining the return codes in the lx_rc fields. If needed, more than two data elements may be used in this manner by marking all but the last element with LX_ELMT_JOIN and LX_ELMT_NO_HYPHEN flags. Once a lemma is successfully determined, no additional processing is done.

The double-byte parse code (DBPC), which is a numeric representation of the PCODE information, will be returned to the calling application. These codes are documented in "Korean Double-Byte Parsing Codes (DBPCs)" on page 331.

Korean Addenda Dictionary Support

A Korean addenda dictionary may be used to change the lemma information returned by the **Morphological Identification** function by including words and a language-specific DBCS parsing code (DBPC). The Korean parsing codes are documented in "Korean Double-Byte Parsing Codes (DBPCs)" on page 331.

In addition, a Korean addenda dictionary may be used to store user data with a word.

The system dictionary information may be overridden by placing the addenda token before the system dictionary token in the dictionary token list. Assigning zero as the only DBPC for an addenda word effectively deletes it from the dictionary. If the word is in the system dictionary and the addenda precedes the system dictionary in the input dictionary list, the word will not be found in the system dictionary.

Assigning any other valid values for the DBPC field for an addenda word will make those values available to the MID function.

No addenda alias processing is supported for Korean.

Keyword and Query Term Extraction for Korean

The **Extract Keywords** and **Extract Query Terms** functions use the output of **Morphological Identification** to return part of speech information for keywords from a block of text. For the text extraction functions, the application must set a flag indicating the part(s) of speech desired. Only words which have been tagged by **Morphological Identification** as having the part(s) of speech specified by the filtering options will be returned. Part of speech information is returned in terms of the condensed POS, rather than the DBPC (see "Korean Double-Byte Parsing Codes (DBPCs)" on page 331 for details). Note that "unknown" words (those which cannot be found in the dictionary) are returned with part of speech LX_EKW_UNKNOWN. For part of speech filtering, this category is included with the nouns.

While **Lexical Analysis** on the host returns shift-out and shift-in as separate tokens from the DBCS text string

```
token 1 – SO
token 2 – DBCS_text
token 3 – SI
```

the text extraction functions return the DBCS text string enclosed by shift-out and shift-in:

```
token 1 – SO_DBCS_text_SI
```

Thus the host DBCS token is two bytes longer than the corresponding DBCS token on the PC, but the number of tokens returned by the text extraction functions is the same on host and PC.

Korean Double-Byte Parsing Codes (DBPCs)

The part of speech codes for DBCS languages differ from those for the European languages. Technically, they should be referred to as DBCS parsing codes (DBPC) rather than part of speech codes, because they are used by the Lexical Analysis function to determine base forms (lemmas) of words in Korean text, but they do not correspond exactly to the parts of speech used to categorize words in the European languages.

DBPC values used in the Korean dictionary are listed in Table 33 and Table 34 on page 332. These values may be assigned to words in an addenda dictionary. For convenience, the table of Korean parse codes has been given twice, once in order according to DBPC number, the second time ordered by Korean PCODE. Some of these have been defined as external constants in EFZLEXTY.H. These are listed in Table 35 on page 334. Others may be defined by the application, if needed.

There are some additional DBPCs that may be returned by the Morphological Identification (MID) function. These are the result of MID processing and may not be assigned to words in an addenda dictionary. They are listed in Table 36 on page 334.

Table 33 (Page 1 of 2). Korean Double-Byte Parse Codes (DBPCs) arranged by Number

DBPC	PCODE	Description
1	NN	noun
2	DIV	suffix “-ha” : intransitive verb
3	DTV	suffix “-ha” : transitive verb
4	AVA	adverb
5	VIV	intransitive verb
6	DDV	suffix “-ha” : adjective
7	VTV	transitive verb
8	GEV	suffix “-doe”
9	AJJ	adjective
10	DOV	suffix “-ha” : intransitive and transitive verb
11	ETJ	pre-noun (determiner)
12	NMN	numeral
13	AJ_RBJ	b-irregular adjective
14	XLI	exclamation (interjection)
15	SEV	suffix “-sreob”
16	VPV	passive verb
17	VCV	causative verb
18	PP	pronoun
19	VITV	intransitive and transitive verb

Table 33 (Page 2 of 2). Korean Double-Byte Parse Codes (DBPCs) arranged by Number

DBPC	PCODE	Description
20	VT_RLV	transitive and l-irregular
21	NXN	auxiliary noun
22	AJ_RHJ	h-irregular adjective
23	DKV	suffix “-ha” : intransitive and adjective
24	AJ_RLJ	l-irregular adjective
25	VI_RLV	intransitive and l-irregular
26	VI_RGV	intransitive and g-irregular
27	VXV	auxiliary verb
28	VI_RNV	intransitive and n-irregular
29	VT_RSV	transitive and s-irregular
30	VT_RDV	transitive and d-irregular
31	DJV	suffix “-ha” : transitive verb and adjective
32	VT_RGV	transitive and g-irregular
33	VI_RSV	intransitive and s-irregular
34	VT_RNV	transitive and n-irregular
35	JXJ	auxiliary adjective
36	VI_RDV	intransitive and d-irregular
37	VT_RBV	transitive and b-irregular
38	AJ_RRJ	r-irregular adjective
39	VI_RBV	intransitive and b-irregular
40	VCIV	incomplete intransitive verb
41	DVV	suffix “-ha” : intransitive, transitive, and adjective
42	VCTV	incomplete transitive verb
43	VI_RRV	intransitive and r-irregular
44	VT_RUV	transitive and u-irregular
45	VXCV	auxiliary verb

Table 34 (Page 1 of 2). Korean Double-Byte Parse Codes (DBPCs) arranged by PCODE

PCODE	DBPC	Description
AJ_RBJ	13	b-irregular adjective
AJ_RHJ	22	h-irregular adjective
AJ_RLJ	24	l-irregular adjective
AJ_RRJ	38	r-irregular adjective
AJJ	9	adjective
AVA	4	adverb
DDV	6	suffix “-ha” : adjective
DIV	2	suffix “-ha” : intransitive verb

Table 34 (Page 2 of 2). Korean Double-Byte Parse Codes (DBPCs) arranged by PCODE

PCODE	DBPC	Description
DJV	31	suffix “-ha” : transitive verb and adjective
DKV	23	suffix “-ha” : intransitive and adjective
DOV	10	suffix “-ha” : intransitive and transitive verb
DTV	3	suffix “-ha” : transitive verb
DVV	41	suffix “-ha” : intransitive, transitive, and adjective
ETJ	11	pre-noun (determiner)
GEV	8	suffix “-doe”
JXJ	35	auxiliary adjective
NMN	12	numeral
NN	1	noun
NXN	21	auxiliary noun
PP	18	pronoun
SEV	15	suffix “-sreob”
VCIV	40	incomplete intransitive verb
VCTV	42	incomplete transitive verb
VCV	17	causative verb
VI_RBV	39	intransitive and b-irregular
VI_RDV	36	intransitive and d-irregular
VI_RGV	26	intransitive and g-irregular
VI_RLV	25	intransitive and l-irregular
VI_RNV	28	intransitive and n-irregular
VI_RRV	43	intransitive and r-irregular
VI_RSV	33	intransitive and s-irregular
VITV	19	intransitive and transitive verb
VIV	5	intransitive verb
VPV	16	passive verb
VT_RBV	37	transitive and b-irregular
VT_RDV	30	transitive and d-irregular
VT_RGV	32	transitive and g-irregular
VT_RLV	20	transitive and l-irregular
VT_RNV	34	transitive and n-irregular
VT_RSV	29	transitive and s-irregular
VT_RUV	44	transitive and u-irregular
VTV	7	transitive verb
VXCV	45	auxiliary verb
VXV	27	auxiliary verb
XLI	14	exclamation(interjection)

Table 35. Defined Constants for Korean Double-Byte Parse Codes (DBPCs)

Constant	DBPC	Description
LX_KMA_NOUN	1	noun
LX_KMA_ADV	4	adverb
LX_KMA_ADJ	9	adjective
LX_KMA_DET	11	determiner
LX_KMA_NUMERAL	12	numeral
LX_KMA_EXCL	14	exclamation
LX_KMA_PNOUN	18	pronoun
LX_KMA_XNOUN	21	auxiliary noun
LX_KMA_XVERB	27	auxiliary verb
LX_KMA_XADJ	35	auxiliary adj.

Table 36. Additional Korean Double-Byte Parse Codes (DBPCs) Returned by MID

PCODE	DBPC	Description
LX_KMA_VERB	101	verb - analyzed as X + eomi --> verb
LX_KMA_ALPHANUM	102	Alphanumerics - word consisted entirely of alphanumeric characters
LX_KMA_NVERBK	104	noun + verb - analyzed as X + ending that could be either eomi or josa --> could be either noun (if ending is josa) or verb (if ending is eomi)
LX_KMA_CNOUNK	105	compound noun - analyzed as noun + noun
LX_KMA_UNKNOWN	106	guessed as noun - analyzed as X + josa --> X probably a noun
LX_KMA_HANJA	107	Hanja lemma

Norwegian Language-Specific Processing

Spell Verify

Compound words

Norwegian is a language which allows compound words. Its dictionary therefore has BOFA flags and the compound word processing is performed, as needed, when verifying words. Only words five characters or longer are processed as potential compounds.

Elliptic enumeration

Elliptic enumeration allows words coded with BOFA “F” (which can only occur in the front or middle of a compound) to verify if they are followed by a hyphen.

Hyphenation/Dehyphenation

Elision

The hyphenation of some compound words causes spelling changes in Norwegian. When two components of a compound word are joined, and the last two letters of the preceding component and the first letter of the following component are the same consonant, then one of the three identical consonants is dropped. However, when such a word is hyphenated between these components, the dropped consonant must be returned to the word. This results in the first part of the word ending a text line with two identical consonants and a syllable hyphen, and the next text line having the next part of the word beginning with a the same consonant.

This spelling change must be removed before the word can be passed to any Linguistic Tools function, and when the hyphenation break point is removed from the word. The application can do this by recognizing the triple consonants separated by a syllable hyphen and removing the syllable hyphen and third consonant before calling any Linguistic Tools function.

Lexical Analysis for Norwegian

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see “Hyphenated Forms” on page 296). This processing includes dehyphenation. Lexical Analysis for Norwegian supports spelling changes that occur in dehyphenation.

Note: A Norwegian dictionary must be the first dictionary activated for spelling changes during dehyphenation to be made properly.

Numbers and Numeric Ranges

The processing of numbers and numeric ranges uses a method analogous to the one used for English (see “Numbers” on page 300) except that the group delimiter is the period and the radix point is the comma.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Norwegian the following language-specific processing is performed:

- A-overcircle (å), ae-ligature (æ), and o-slash (ø) are rewritten as “aa”, “ae”, and “oe”, respectively.
- All other accents will be removed.

Portuguese Language-Specific Processing

Enclitic Processing

Portuguese enclitic pronouns, unlike Spanish or Italian enclitics, may be imbedded within verb forms. The following paragraphs describe the rules for forming these enclitics. This information was used in the design of an algorithm to remove the enclitics from a verb form in order to generate the original form of the verb to which the enclitic pronouns were added. Enclitic processing is done for the **Spell Verify, Morphological Identification, Generate Inflected Forms**, and **Synonym Aid** functions in Portuguese.

Categories of pronouns and contractions: The enclitic pronouns and contracted forms are presented in Table 37 through Table 42.

Table 37. Portuguese Reflexive Pronoun (RP)

Enclitic	Person
-se	3

Table 38. Portuguese Personal Pronouns (PP), Accusative and Dative Case

Enclitic	Person	Number
-me	1	S
-te	2	S
-nos -no	1	P
-vos -vo	2	P

Table 39. Portuguese Impersonal Pronouns (IP), Accusative Case

Enclitics	Person	Number	Gender
-o -lo -no	3	S	M
-os -los -nos	3	P	M
-a -la -na	3	S	F
-as -las -nas	3	P	F

Table 40. Portuguese Indirect Object Pronouns (IO), Dative Case

Enclitic	Person	Number
-lhe	3	S
-lhes	3	P

Table 41 (Page 1 of 2). Portuguese PP/IP Contractions (PPIPC), Dative + Accusative Case

me + IP	te + IP
-m'o -mo	-t'o -to
-m'os -mos	-t'os -tos
-m'a -ma	-t'a -ta
-m'as -mas	-t'as -tas

Table 42. Portuguese IO/IP contractions (IOIPC), Dative + Accusative Case

lhe + IP
-lh'o -lho
-lh'os -lhos
-lh'a -lha
-lh'as -lhas

Brazilian Portuguese can use the contractions with apostrophes and also the special contraction -lh' instead of -lhe- when the enclitic is imbedded by itself in the future or conditional form of the verb, for example, dar-lh'emos.

General enclitic formation rules: Any verb form may have from one to three enclitics. Each enclitic is appended to or imbedded in the verb form separated by a hyphen. If one enclitic is used, it can be either RP, PP, IO, IP, PPIPC or IOIPC. If two enclitics are used, they can be PP+IP, RP+PP, RP+IO, RP+PPIPC or RP+IOIPC. The combination RP+IP is never used. If three enclitics are used only RP+PP+IP is valid, where PP is "nos" or "vos" subject to the transformation rules.

Each enclitic pronoun is separated from the verb form or from the previous pronoun by a hyphen. Contractions, except -lh' by itself, are considered to be two pronouns and are used in the combinations stated above. The IP forms starting with "l" or "n" are used only when the transformation rules given below apply.

Embedding rules: Future and conditional verb forms are decomposed into stem and ending before the enclitics are imbedded, then the ending is added following the enclitics and separated from them by a hyphen. The verb stem or the enclitics themselves may undergo transformations according to the rules given below. The future endings are: -ei, -ás, -á, -emos, -eis, -ão. The conditional endings are: -ia, -ias, -ia, -íamos, -íeis, -iam. (Examples: dar-lhe-emos, dar-lho-ia)

The future and conditional of the verbs "fazer", "dizer", and "trazer" are irregular in that they are derived from the short infinitive of Latin "far(e)", "dir(e)", "trar(e)", but the rules for imbedding are the same as above and also follow the transformation rules, for example, farei + o = fá-lo-ei.

Transformation rules: The IP forms -lo, -los, -la, and -las exist only as transformations of the forms -o, -os, -a, and -as under the following two conditions:

1. When an infinitive verb form (or a future or a conditional which consists of the infinitive plus an ending) needs to take the enclitics -o, -os, -a, or -as, the “r” of the infinitive stem is dropped and the enclitic is transformed to -lo, -los, -la, or -las, respectively. If the vowel preceding the “r” is “a”, it changes to “á”, if it is “e” but not “õe”, it changes to “ê”, and if “o” it changes to “ô”.
2. When forms ending in “z” of the verbs “trazer”, “fazer”, “dizer” and their derivatives such as “afazer”, “satisfazer”, “bendizer”, and so on, need to take the enclitics -o, -os, -a, or -as, the “z” is dropped and the enclitic is transformed to -lo, -los, -la, or -las, respectively. If the vowel preceding the “z” is “a”, it changes to “á” and if it is “e” it changes to “ê”.
3. When a verb form ending in “s” needs to take the enclitics -o, os, -a, or -as, the “s” is dropped and the enclitic is transformed to -lo, -los, -la, or -las, respectively.
4. The final “s” of a first person plural verb form ending in “mos” is dropped when followed by the enclitic “-nos” to generate “mo-nos”. This rule does not apply to the future and conditional forms which imbed the enclitics.
5. When the pronouns “nos” and “vos” are to be followed by -o, -os, a or -as, the “s” of “nos” or “vos” is dropped and the following enclitic is transformed to -lo, -los, -la, or -las, respectively.

These rules apply even when the enclitic endings are imbedded. Some examples are given in Table 43.

Table 43. Sample Application of Portuguese Enclitic Transformation Rules (a)

dar + o	=	dá-lo
traz + o	=	trá-lo
pões + o	=	põe-lo
darei + o	=	dá-lo-ei
daria + as	=	dá-las-ia
viveriam + o	=	viv*-lo-iam
trazes + nos + o	=	trazes-no-lo
trazem + vos + o	=	trazem-vo-lo
dispor + o	=	dispJ-lo

The IP forms -no, -nos, -na, and -nas are transformations of the forms o, -os, -a, and -as when they occur after a verbal form ending with the letter “m” or after the nasal vowel combinations “ão” and “õe”. The fact that the ending -nos is also a personal pronoun is a potential ambiguity (see Table 44).

Table 44. Sample Application of Portuguese Enclitic Transformation Rules (b)

lavavam + os	=	lavavam-nos
trazem + o	=	trazem-no

Hyphenation

National Portuguese has special processing for words containing required hyphens. Hyphenation break points should be placed before required hyphens. This will result in the first part of the word being placed on one text line followed by a syllable hyphen, and the other part of the word will be placed on the next text line preceded by the required hyphen.

This processing will not be done for Brazilian Portuguese.

Lexical Analysis for Portuguese

Hyphenated Forms

Orthographic conventions make the hyphen unambiguous in National Portuguese. A word split across a line is represented differently, depending on the character at which the split occurs. If a word containing a hyphen is split across the line at the hyphen character, Portuguese marks this situation distinctly by placing two hyphens: one at the end of the line and the other at the beginning of the following line. If the word is split at any other position, Portuguese places only one hyphen: at the end of the line.

For example, the word ***guarda-chuva*** (umbrella) could appear in any one of the following three forms:

```
(1)
guar-
da-chuva
(2)
guarda-
—chuva
(3)
guarda-chu-
va
```

In example (2), the line end occurs exactly at the hyphen character, so it is marked explicitly by putting a second hyphen on the following line. In examples (1) and (3), the line end occurs within the parts of the hyphenated word. The single hyphen indicates that a hyphen is not part of the word itself.

Hyphenated forms in Brazilian Portuguese are treated using a method analogous to the one used for English (see “Hyphenated Forms” on page 296). This processing includes dehyphenation.

Numbers and Numeric Ranges

The processing of numbers and numeric ranges uses a method analogous to the one used for English (see “Numbers” on page 300) except that the group delimiter is the period and the radix point is the comma.

Morphological Identification

In order to determine the lemma of a word in either National or Brazilian Portuguese, enclitics are removed from the input word, as described above.

Generate Inflected Forms

In order to generate the inflected forms of a word in either National or Brazilian Portuguese, enclitics are removed from the input word, as described above.

Synonym Aid

In order to determine the synonyms for a word in either National or Brazilian Portuguese, enclitics are removed from the input word, as described above.

Word Normalization

The text extraction functions (**Extract Keywords** and **Extract Query Terms**) may optionally return a normalized form of the word. For both National and Brazilian Portuguese the following language-specific processing is performed:

- All accents are removed.

Spanish Language-Specific Processing

Lexical Analysis for Spanish

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see "Hyphenated Forms" on page 296). This processing includes dehyphenation.

Numbers and Numeric Ranges

The processing of numbers and numeric ranges uses a method analogous to the one used for English (see "Numbers" on page 300) except that the group delimiter is the period and the radix point is the comma.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Spanish, the following language-specific processing is performed:

- All accents are removed.

Russian Language-Specific Processing

Spell Verify

Russian processing does not differ from English except for the handling of the letter YO (ë). Russian input text may be written with the letter YE (e) in place of the letter YO. Thus, the verification process must map an input YO to a YE when the Russian dictionary does not have the letter YO. However, for a dictionary that has words with YO, the verification process must map input words with YE to dictionary words that have YO in the corresponding position.

Swedish Language-Specific Processing

Spell Verify

Compound Words

Swedish is a language which allows compound words. Its dictionary therefore has BOFA flags and compound word processing is performed, as needed, when verifying words. Only words five characters or longer are processed as potential compounds.

Elliptic enumeration

Elliptic enumeration allows words coded with BOFA "F" (which can only occur in the front or middle of a compound) to verify if they are followed by a hyphen.

Hyphenation/Dehyphenation

The hyphenation of some compound words causes spelling changes in Swedish. When two components of a compound word are joined, and the last two letters of the preceding component are identical consonants, and the first letter of the following component is the same consonant, then one of the three identical consonants is dropped. For example, when "adress" and "stadga" are joined to form a compound word, then one "s" is dropped and the final spelling is "adresstadga". However, when such a word is hyphenated between these components, then the dropped consonant must be returned to the word. This results in the first part of the word ending a text line with two identical consonants and a syllable hyphen, and the the next text line having the next part of the word beginning with a the same consonant. For example, "adresstadga" when hyphenated between the word components is spelled "adress-" with "stadga" on the next line.

This spelling change must be removed before the word can be passed to any Linguistic Tools function, and when the hyphenation break point is removed from the word. This removal can be done by the application by recognizing the triple consonants separated by a syllable hyphen and removing the syllable hyphen and third consonant before calling any Linguistic Tools function.

Lexical Analysis for Swedish

Hyphenated Forms

Hyphenated forms are treated using a method analogous to the one used for English (see "Hyphenated Forms" on page 296). This processing includes dehyphenation. Lexical Analysis for Swedish supports spelling changes that occur in dehyphenation.

Note: A Swedish dictionary must be the first dictionary activated for spelling changes during dehyphenation to be made properly.

Word Normalization

The text extraction functions (**Extract Keyword** and **Extract Query Term**) may optionally return a normalized form of the word. For Swedish the following language-specific processing is performed:

- A-diaeresis (ä), o-diaeresis (ö), and a-overcircle (å) are rewritten as “ae”, “oe”, and “aa”, respectively.
- All other accents are removed.

Appendix E. Morphology Grammar Masks

Morphology grammar masks are 32 bits long. Any bits not enumerated in the following lists are not used. All unused bits must be set to zero in order to match dictionary data when calling the **Generate Inflected Forms** function. The bits are numbered with 1 being the most significant bit.

Catalan

1. Infinitive
2. Present
3. Imperfect
4. Future
5. Conditional
6. Simple past
7. Gerund
8. Past participle
9. Indicative
10. Subjunctive
11. Imperative
12. Finite verb form (has person/number/tense)
13. Non-finite verb form (lacks person/number/tense)
14. Perfective aspect
15. Imperfective aspect
16. First person
17. Second person
18. Third person
19. Singular
20. Plural
21. Masculine
22. Feminine
23. Neuter

Czech

1. First person
2. Second person
3. Third person
4. A
5. A4
6. AC
7. AG
8. AN
9. Ab
10. Ac
11. Adverb/Adjective
12. Aff
13. AfterPrep
14. Agr
15. Card
16. Cause

© Copyright IBM Corp. 2000

17. Cd
18. Comp
19. Cond
20. Ctd
21. D3
22. Dem
23. Du
24. El
25. Ela
26. Eli
27. Elo
28. Ely
29. En
30. Ena
31. Encirc
32. Eni

Danish

1. Imperative
2. Infinitive
3. Present
4. Past
5. Past participle
6. Past participle (gender-marked forms)
7. Present participle
8. Nominative
9. Accusative
10. Genitive
11. Active voice
12. Passive voice
13. First person
14. Second person
15. Third person
16. Singular
17. Plural
18. Common gender
19. Neuter
20. Masculine
21. Feminine
22. Positive
23. Comparative
24. Superlative
25. Superlative definite
26. Definite
27. Indefinite

Dutch

1. Infinitive
2. Imperative
3. Present
4. Past
5. Present participle
6. Past participle
7. Singular
8. Plural
9. First person
10. Second person
11. Third person
12. Positive
13. Comparative
14. Diminutive
15. Inflected form
16. Cardinal number
17. Ordinal number

English

1. Infinitive
2. Present
3. Past
4. Present participle
5. Past participle
6. First person
7. Second person
8. Third person
9. Singular
10. Plural
11. Positive
12. Comparative
13. Superlative
14. Possessive

Finnish

1. Second infinitive
2. First infinitive
3. Comparative
4. Superlative
5. First person plural
6. Second person plural
7. Third person plural
8. Nominative
9. Genitive
10. Inessive
11. Elative
12. Illative
13. Adessive
14. Ablative

15. Allative
16. Essive
17. Abessive
18. Instructive
19. Comitative
20. Paritive
21. Transitive
22. Man
23. Singular
24. Plural
25. Active
26. Passive
27. Present
28. Past
29. Negation
30. First person singular
31. Second person singular
32. Third person singular

French (National)

1. Infinitive
2. Present indicative
3. Imperfect
4. Historical past (passé simple)
5. Future
6. Present subjunctive
7. Imperfect subjunctive
8. Conditional
9. Imperative
10. Present participle
11. Past participle
12. Meaning depends on other bits with which it is combined
    - Finite verb form (has person/number/tense)
    - Proper noun
    - Masculine adjective with alternate forms (for example, *beau*)
    - Function word (article/pronoun/demonstrative) which undergoes elision
13. First person singular
14. Second person singular
15. Third person singular
16. First person plural
17. Second person plural
18. Third person plural
19. Singular
20. Plural
21. Masculine
22. Feminine

Canadian French

1. Infinitive
2. Present indicative
3. Subjunctive
4. Imperfect
5. Future
6. Conditional
7. Historical past (passé simple)
8. Imperative
9. Present participle
10. Past participle
11. First person
12. Second person
13. Third person
14. Singular
15. Plural
16. Masculine
17. Feminine

German

1. Infinitive
2. Zu-infinitive
3. Present tense
4. Past tense
5. Present subjunctive
6. Past subjunctive
7. Present participle
8. Past participle
9. Imperative
10. Stem
11. Nominative
12. Genitive
13. Dative
14. Accusative
15. First person
16. Second person
17. Third person
18. Singular
19. Plural
20. Masculine
21. Feminine
22. Neuter
23. Positive
24. Comparative
25. Superlative
26. Predicative /adverbial use
27. Determined (weak/strong)
28. Bound morpheme
29. Uninflected adjective
30. Fugen-s-noun

Greek

1. Active voice
2. (Medio-)passive voice
3. Present tense
4. Future tense
5. Past tense
6. Indicative
7. Subjunctive
8. Imperative
9. Imperfective aspect
10. Perfective aspect
11. First person
12. Second person
13. Third person
14. Singular
15. Plural
16. Non-finite verb form (lacks person/number/tense)
17. Present participle
18. Nominative case
19. Genitive case
20. Accusative case
21. Vocative case
22. Masculine
23. Feminine
24. Neuter
25. Comparative degree
26. Superlative degree
27. Unemphatic (clitic) pronoun
28. Possessive pronoun
29. Preposition fused with definite article

Italian

1. Indicative
2. Subjunctive
3. Conditional
4. Imperative
5. Participle
6. Gerund
7. Infinitive
8. Present
9. Past
10. Imperfect
11. Future
12. First person singular
13. Second person singular
14. Third person singular
15. First person plural
16. Second person plural
17. Third person plural
18. Singular
19. Plural

20. Masculine
21. Feminine

Norwegian, Bokmål

1. Infinitive
2. Mediopassive
3. Present
4. Preterite (simple past)
5. Gerund
6. Past participle
7. Imperative
8. Masculine
9. Feminine
10. Neuter
11. Indefinite
12. Definite
13. Invariant form (non-finite verb, non gender-marked adjective)
14. Singular
15. Plural
16. Nominative
17. Accusative
18. Genitive
19. Positive
20. Comparative
21. Superlative
22. Article
23. Auxiliary verb
24. Proper noun

Norwegian, Nynorsk

1. Infinitive
2. Mediopassive
3. Present
4. Preterite (simple past)
5. Gerund
6. Past participle
7. Imperative
8. Masculine
9. Feminine
10. Neuter
11. Indefinite
12. Definite
13. Invariant form (non-finite verb, non gender-marked adjective)
14. Singular
15. Plural
16. Nominative
17. Accusative
18. Genitive
19. Positive
20. Comparative
21. Superlative

22. Article
23. Auxiliary verb
24. Proper noun

Portuguese, National

1. Infinitive
2. Present
3. Imperfect
4. Future
5. Conditional
6. Preterite (simple past)
7. Gerund
8. Past participle
9. Indicative
10. Subjunctive
11. Imperative
12. Finite verb form (has person/number/tense)
13. Non-finite verb form (lacks person/number/tense)
14. Perfective aspect
15. Imperfective aspect
16. First person
17. Second person
18. Third person
19. Singular
20. Plural
21. Masculine
22. Feminine
23. Neuter
24. Nominative (subject)
25. Accusative (direct object)
26. Dative (indirect object)
27. Possessive
28. Reflexive

Brazilian Portuguese

1. Infinitive
2. Present
3. Imperfect
4. Future
5. Conditional
6. Preterite (simple past)
7. Gerund
8. Past participle
9. Indicative
10. Subjunctive
11. Imperative
12. Finite verb form (has person/number/tense)
13. Non-finite verb form (lacks person/number/tense)
14. Perfective aspect
15. Imperfective aspect
16. First person

17. Second person
18. Third person
19. Singular
20. Plural
21. Masculine
22. Feminine
23. Neuter
24. Nominative (subject)
25. Accusative (direct object)
26. Dative (indirect object)
27. Possessive
28. Reflexive
29. Meaning depends on other bits with which it is combined
    - Indefinite pronoun
    - Relative pronoun (with bit 31)
30. Meaning depends on other bits with which it is combined
    - Possessive pronoun
    - Interrogative pronoun (with bit 31)
31. Meaning depends on other bits with which it is combined
    - Demonstrative pronoun
    - Relative pronoun (with bit 29)
    - Interrogative pronoun (with bit 30)
32. Proper noun

Russian

1. Nominative
2. Genitive
3. Dative
4. Acusative
5. Instrumental
6. Prepositional
7. first person
8. second person
9. third person
10. Singular
11. Plural
12. Masculine
13. Feminine
14. Neuter
15. Infinitive
16. Present
17. Past
18. Past Participle
19. Adverbial Participle
20. Imperative
21. Active
22. Passive
23. Animate
24. Inanimate
25. Comparative

Spanish

1. Infinitive
2. Present
3. Imperfect
4. Future
5. Conditional
6. Preterite (simple past)
7. Gerund
8. Past participle
9. Indicative
10. Subjunctive
11. Imperative
12. Finite verb form (has person/number/tense)
13. Non-finite verb form (lacks person/number/tense)
14. Perfective aspect
15. Imperfective aspect
16. First person
17. Second person
18. Third person
19. Singular
20. Plural
21. Masculine
22. Feminine
23. Neuter
24. Nominative
25. Accusative
26. Dative
27. Reflexive
28. Prepositional pronoun

Swedish

1. Infinitive
2. Present tense
3. Past tense
4. Supine
5. Past participle
6. Imperative
7. Conjunctive
8. S-form of verbs
9. Singular
10. Plural
11. First person singular
12. Second person singular
13. Third person singular
14. First person plural
15. Second person plural
16. Third person plural
17. Positive
18. Comparative
19. Superlative
20. Masculine

21. Feminine
22. Real
23. Neuter
24. Utrum (masculine, feminine, real)
25. Definite
26. Indefinite
27. Genitive
28. Nominative
29. Subject form
30. Object form

Turkish

1. Proper
2. Nominal Agreement 1 Singular
3. Nominal Agreement 2 Singular
4. Nominal Agreement 3 Singular
5. Nominal Agreement 1 Plural
6. Nominal Agreement 3 Plural
7. Nominal Agreement 3 Plural
8. Nominal Possessive 1 Singular
9. Nominal Possessive 2 Singular
10. Nominal Possessive 3 Singular
11. Nominal Possessive 1 Plural
12. Nominal Possessive 2 Plural
13. Nominal Possessive 3 Plural
14. Nominal Possessive Pnon
15. Nominal Case Nominative
16. Nominal Case Accusative
17. Nominal Case Dative
18. Nominal Case Ablative
19. Nominal Case Cloc
20. Nominal Case Genitive
21. Nominal Case Cins
22. Nominal Case Cequ
23. Verbal Agreement 1 Singular
24. Verbal Agreement 2 Singular
25. Verbal Agreement 3 Singular
26. Verbal Agreement 1 Plural
27. Verbal Agreement 2 Plural
28. Verbal Agreement 3 Plural
29. Verbal Tense Aspect Mood 1 narr
30. Verbal Tense Aspect Mood 1 futr
31. Verbal Tense Aspect Mood 1 aor
32. Verbal Tense Aspect Mood 1 prog

Glossary

A

addenda dictionary. Word list which is separate from any Linguistic Tools supplied dictionary. The form of the word list is application-dependent; it may be an in-storage list, a disk-based list, a Dictionary Access Processor structured dictionary, or a modified Dictionary Access Processor structured dictionary. The use of addenda dictionaries allows the user to add word information for Linguistic Tools processing.

algorithmic hyphenation. Hyphenation points for each word are determined by a set of language-specific rules. This assures that at least one hyphenation point will be returned for each word that is long enough to be a candidate for hyphenation. See also **dictionary-based hyphenation**.

alias (morphological). A term which the user provides to the Linguistic Tools morphology functions as a model for a word being added to a user-created dictionary. Linguistic Tools will inflect the new word according the pattern of the alias word.

application. The program, system, or environment which uses and interfaces to the Linguistic Tools functions.

article. One of a small closed set of words that typically precede nouns as an indication of specificity of reference. In English *a* or *an* indicates indefinite reference, while *the* usually indicates reference to a specific known entity.

aspect (grammatical). A grammatical category which reflects the way in which verb action is regarded or experienced with respect to time (Randolph Quirk, Sidney Greenbaum, Geoffrey Leech, and Jan Svartvik. A COMPREHENSIVE GRAMMAR OF THE ENGLISH LANGUAGE. New York: Longman, 1985, p. 188). English marks *perfective* aspect by the use of the auxiliary *have* + the past participle —for example, *Has* he *finished* that task?— and *progressive* aspect by the use of the auxiliary *be* + the present participle—for example, *Is* he still *working* on that task?

auxiliary (verb). One of a small closed set of verbs that may occur together with other verbs as a "helping" verb. In English these are *be, have* and *do*, plus the modals *can, may, will, shall, could, might, would, should,* and *must*. E.g.,

```
He has completed his assignment.
When did he complete that assignment?
Would you look at these results?
```

B

base form. See **lemma**.

BOFA (flags). A system of flag indicators to designate the positions within a a compound which a given component word may occupy: front, middle, back, or some combination thereof. There is also a flag to indicate whether a particular component word may occur simplex, that is, as a stand-alone word.

byte. Eight binary digits grouped together internally by the computer. A byte can have the values 0-255, x'00'-x'FF'.

C

cang jie. A system for input of Chinese characters based on the strokes used in writing them. Each character is analyzed into components, which are entered via the keyboard, then the result is converted to *hanzi*.

case (grammatical). A grammatical category indicating the role of a noun in a phrase or sentence: subject, object, and so on. English marks case only on pronouns—for example, *He* went to the meeting. I saw *him* there. Other languages may use a larger set of cases and markers, including different spellings of the noun and/or other words used with the noun.

chuyinfuhao. A set of symbols developed for phonetic representation of Chinese, used in the Republic of China (Taiwan).

cí. The Chinese term for a word. A **cí** may consist of one or more written characters (**zì**). See also **hanzi**.

closing punctuation. Punctuation character that is the closing member of a pair, such as parentheses or quotes. Linguistic Tools recognizes the characters listed in Table 3 on page 34.

code page. A set of assignments mapping graphic characters and/or control function meanings to individual code points. For an 8-bit code there is a maximum of 256 code points.

code page tables. A set of tables needed by the Linguistic Tools to process text sent to it. These include (1) an external-to-internal (*cpg_to_dap*) table to map external character codes to the Linguistic Tools internal codes, (2) an internal-to-external (*dap_to_cpg*) table to map the Linguistic Tools internal codes to

external character codes, (3) a simple token table which indicates for each code point in the external code page which category it belongs in (always delimiter, never delimiter, sometimes delimiter) for the purpose of defining simple tokens, (4) a character name table which specifies how each code point in a particular code page should be mapped to a standard Graphic Character Global Identifier (GCGID) value. See Appendix B, "Internal Character Codes" on page 253 and "NlpRegisterCodePage" on page 221 for details.

code point. A 1-byte (8-bit) code representing one of 256 potential characters. See also **code page**.

complex token analysis. See **lexical analysis**.

complex tokenization. See **lexical analysis**.

component (of a compound word). Part of a compound word which may also occur as a dictionary entry. Must have the proper BOFA flags for the position in which it occurs within a word, in order to be considered a valid component of that compound.

component (of a joined token). A token—may be simple or itself joined— which is joined to another token to span a larger portion of the input data.

compound word. A word formed by putting together two or more other words. In any given language there may be restrictions on the position in which certain components may occur within a compound. Such restrictions are indicated by BOFA flags. In the Linguistic Tools this term refers specifically to a word which can be verified by parsing it into its component dictionary parts while obeying BOFA flag rules. See also **simple word**.

contextual search. A process which uses phrases as well as single words in searching documents for information.

contracted auxiliary. A shortened form of an auxiliary verb which is usually written as part of the preceding word, but marked off by an apostrophe. E.g.,

```
I'm starting a new project next week.
Where'd he go?
They'd like to see this.
```

See **auxiliary (verb)**.

D

data element. A data structure used by an application to pass input data to the Linguistic Tools. May be single-word or block format. A single-word format data element points to a single word or fragment thereof. A block format data element either points to a contiguous block of text or specifies a special character such as a newline. See "Use of Single-Word Format Data Elements" on page 71 for details of single-word format data elements.

DBCS. Double-Byte Character Set. A character set in which each character is represented not by one byte, but two. Used for languages with larger character inventories than can be accommodated within the limits set by 8 binary digits, such as Chinese, Japanese, and Korean. See also **SBCS**.

DBPC. DBCS Parsing Code. A two-byte language-specific hex value which contains part-of-speech-like information. This value is to parse words or determine lemmas for words in the double-byte languages for which these codes are supported (Japanese and Korean). See "Japanese Part of Speech (POS) Codes" on page 324, and "Korean Double-Byte Parsing Codes (DBPCs)" on page 331 for details.

delimiter character. A character which can be used to identify the boundaries of words and other units in the input data. The basic word delimiter in many languages is a blank space. Other delimiters include various punctuation marks, newline characters, and the like.

derivation (morphological). Grammatical process by which words are formed on the basis of other words, typically involving a change in part of speech and/or meaning. For example, the English noun *derivation* is itself based on the verb *derive*, while the negative adjective *unhappy* is based on the positive adjective *happy*.

dictionary-based hyphenation. Hyphenation points for a word are determined by looking for hyphenation data in the dictionary. This may provide a "better" (set of) hyphenation point(s) than could be obtained from algorithmic hyphenation, but only if the word is found in a dictionary which contains hyphenation data. See also **algorithmic hyphenation**.

dictionary entry. A word in a Linguistic Tools dictionary.

E

elide. See **elision**.

elision. Suppression of a letter during formation of a compound word. E.g., (German) Schiff + Fahrt = Schiffahrt. ***Note:*** Elision also occurs in the formation of what in the Linguistic Tools are treated as prefixed forms—for example, (French) le + enfant = l'enfant.

enclitic pronoun. A pronominal form that can be attached to or embedded in a verb to indicate, for example, the direct or indirect object of the verb.

F

form (of a word). Any member of a set of spellings which can be considered to be the "same word". These include the **base form**, or **lemma**, as well as spellings indicating grammatical information such as person, number, tense, aspect, gender, case, and so on. See also **lemma**, **inflected form**.

form id. Identifies the inflection of a word in a paradigm

G

GCGID. Graphic Character Global Identifier. Uniquely identifies a graphic character. Consists of two letters and six decimal digits.

gender (grammatical). A grammatical category, part of a system for subcategorizing nouns, pronouns, and other words in a language. English explicitly recognizes gender only in the third person singular pronouns *he, she,* and *it*. Many languages assign grammatical gender to all nouns—for example, in French *la table* ('the table') is feminine, while *le livre* ('the book') is masculine. French adjectives typically must agree in gender with the noun which they modify. In some languages a verb must agree in gender with the noun which is its subject.

H

hangeul. The native Korean writing system. Each hangeul unit represents a syllable consisting of several **jamo**. Most words are polysyllabic. Also spelled **hangul**. Some Korean text also includes Chinese characters, which are known as **hanja** in Korean.

hanja. The Korean name for the ideographic characters (originally borrowed from Chinese, hence the name which means literally "Chinese characters") traditionally used as part of the Korean writing system. In contemporary Korean text these characters are used primarily for proper names, but even those are often written in **hangeul** rather than **hanja**.

hanzi. The Chinese term for Chinese characters. Each **zì** represents a syllable, but most words (**cí**) in modern Chinese are made up of two or three syllables (i.e., written as two or three **zì**).

hard hyphen. A hyphen character which has been entered in the text by means of the hyphen key rather than by some hyphenation algorithm. See also **required hyphen**, **soft hyphen**, **syllable hyphen**.

headword. A dictionary term, for which there may be associated data.

high-frequency dictionary. A relatively small set of the more frequently used words in the language. May cover as much as 85% of the words in ordinary business text. Used by Linguistic Tools to speed up spell checking and other functions. See also **ultra-high frequency dictionary**.

hiragana. A set of syllabic characters used to write the Japanese language. Each character represents a syllable—for example, ma, de, ki, no, tsu. In standard Japanese text these characters may be used to write either stems (when no *kanji* are available) or, more frequently, the various grammatical suffixes and markers which may occur with a stem. See also **kanji**, **kana**, and **katakana**.

hot zone. Area at the right margin of a text line which the user has indicated should be at least partially filled with characters. Its normal use is to indicate the area in which words should be hyphenated if necessary to place at least some characters into the zone. This is also sometimes called the **hyphenation zone**.

hyphenation point. Position in a word where a hyphen may be used. Different languages have different rules governing placement of hyphens.

I

inflected form. A word altered from its base form by the addition of grammatical information concerning person, number, gender, case, tense, aspect, and so on.

inflection. A category of grammatical information. Includes person, number, tense, aspect, gender, case, and so on. Different languages use different categories and indicate them in different ways.

J

JAIRS. A Japanese information retrieval system, similar to STAIRS.

jamo. The smallest units in the native Korean writing system. Each jamo represents a consonant or a vowel. Several jamo are combined to form a **hangeul** syllabic unit.

JPOS. Japanese Part of Speech. A system of grammatical categories used solely for the analysis of

Japanese, which can be reduced to the Linguistic Tools default parts of speech by a set of equates. See "Japanese Part of Speech (POS) Codes" on page 324 for details.

K

kana. Collective term for the two sets of syllabic characters (*hiragana* and *katakana*) used in writing the Japanese language. Each symbol represents a syllable—for example, ka-na.

kanji. The Japanese name for the ideographic characters (originally borrowed from Chinese, hence the name which means literally "Chinese characters") used in writing the Japanese language.

katakana. A set of syllabic characters used especially for writing foreign words in Japanese text. Each symbol represents a syllable—for example, bei-su-bo-ru.

keyword. Word in a text which can be used as a search key in information retrieval. See also **query term, stop word**.

keyword extraction. Process by which important words in a text are identified for use in infomation retrieval.

L

lemma. The form of a word which is not marked by any inflectional information such as person, number, tense, and so on. Also called **base form**.

lexical analysis. Special processing to re-analyze certain types of forms in the input data before assigning them to objects (**tokens**) which can be manipulated and output by Linguistic Tools. Processing varies for different languages. Language-specific details are given in Appendix D, "Language-Specific Processing" on page 281. In an earlier version of the Linguistic Tools this function was called **complex tokenization** or **complex token analysis**. See also **simple tokenization**.

M

mixed case. A form or word two characters or longer in which a non-initial character is an upper case letter and some other character in the word is a lower case letter.

morphology. The study of word formation, including inflection and derivation. Specifically in the Linguistic Tools, the system which relates and identifies the inflected forms of words.

morphology mask. A 32-bit field which is used as output to identify the inflective classification of the input word. A word may satisfy many inflections. For example, the word "run" can be 1st or 2nd person singular present tense. One or more masks will be used to indicate all of the possible inflections. See Appendix E, "Morphology Grammar Masks" on page 347. for language-specific details.

N

nibble. Half of a byte. Either the lower or upper four binary digits of a byte. A nibble can have the values 0-15, x'0'-x'F'.

number (grammatical). Grammatical category indicating how many entities a word refers to. In English nouns, pronouns, and verbs may be singular or plural. In other languages adjectives may also be marked for number.

O

P

paradigm (grammatical). A pattern of inflectional changes in the spelling of a word. For nouns, this includes the singular, plural, and various case-marked forms. For verbs this includes all permissible combinations of person, number, tense, aspect, and other grammatical information that may be marked on the verb.

paradigm number. Number used by the Linguistic Tools dictionary to designate a set of inflectional spelling changes for a word. These numbers are assigned during the dictionary build process.

parser. Internal Linguistic Tools function which attempts to decompound a compound word into valid components. (Note: This differs from a syntactic parser which tries to analyze the structure of sentences or parts thereof.)

part of speech (POS). A class of words that can fill the same role within a grammatical construction. Linguistic Tools recognizes the following parts of speech: nouns, verbs, adjectives, adverbs, pronouns, articles, conjunctions, and prepositions.

PCODE. A seven byte code which is used as an index to dictionary morphology tables. Each PCODE serves as a cover term for a group of paradigms. For

example, all English verbs which may serve as the main verb in a sentence have the same PCODE. The modal verbs (*can, may, must, might, shall, will, should, would, could*) follow a different pattern of grammatical behavior, so they have a different PCODE. Each language has a different set of PCODEs.

person (grammatical). Grammatical category indicating whether the entity to which a word refers is the speaker (first person), the addressee/hearer (second person), or someone/something else (third person). Most (all?) languages have different pronouns for each person—for example, English *I/we, you, he/she/it/they*. In many languages verbs must agree in person with their subjects.

phonetic key. A respelling of a keyword used for matching on the basis of similarity in pronunciation.

phonetic spell aid. Spell aid candidates are produced based on similarity in pronunciation, even though actual spelling may be dissimilar—for example, when *night* is produced as a candidate for the input form *nite*.

POS. See **part of speech**.

preferred hyphenation point. The point which should be selected first— if it is within a specified distance from the right margin of a text— for dividing a word. In languages such as German it is considered better to hyphenate a compound word at the boundary between components than to hyphenate within a component. This information is provided in some Linguistic Tools system dictionaries, but may also be provided by the user in an addenda dictionary. See also **ranked hyphenation**.

prefix. A lexical unit which may be attached to the beginning of a word to form another word. In the Linguistic Tools this term refers specifically to the prefixed function words in Catalan, French, and Italian. In general linguistic usage the term refers to something which is more tightly bound to the rest of the word—for example, English *dis*agree, *de*activate, *re*open.

prefixed function word. A contracted word which is joined to a trailing word with an apostrophe. Typically these are function words such as articles and prepositions. For example, in the French construction *l'enfant* ("the child") *l* is the prefix (abbreviated from the article *le*) and *enfant* is the trailing word.

pinyin. A system for transcribing Chinese phonetically using the Latin alphabet, developed and used in the People's Republic of China (PRC).

preposition. One of a small closed set of words typically indicating location, direction, or duration in time or space—for example, *in, on, at, to, from*.

property (of a token). An individual piece of information associated with a token. A token may have multiple properties associated with it. The token list utility functions may be used to traverse the properties and get their values.

Q

Q-Value. A quantification value used in spell aid to indicate the closeness of match of an aid candidate word to an input word. The spell aid algorithm described in this document uses Q1 and Q2 values.

query term. Word extracted from a query for use as a search key in information retrieval. See also **keyword**.

R

ranked hyphenation. Information concerning the order in which various possible points within a word are considered to be desirable points for hyphenation. E.g., in German it is considered better to divide a compound word at the boundary between components than to divide between syllables of the same component. See also **preferred hyphenation point**.

reply area. A buffer used by the Linguistic Tools to hold the data returned by one of its functions.

required hyphen. A character that represents a hyphen occurring naturally in a word or used as punctuation. It is sometimes called a strong or hard hyphen. See also **hard hyphen**, **soft hyphen**, **syllable hyphen**.

romaji. The Japanese term for the Latin alphabet, specifically as it is used to represent the pronunciation of Japanese words. See **romanization**.

romanization. The use of the Latin alphabet to transcribe phonetically words in a language which does not use the Latin alphabet for its written representation—for example, Chinese, Japanese, and Korean.

S

SBCS. Single Byte Character Set. A character set in which each character is represented by a one-byte code. Used for languages whose writing systems use a relatively small set of symbols. See also **DBCS**.

search substring. Substring of a compound word starting at any letter of the input compound word except the last, and ending in the last letter of the same. Used

for dictionary searches for possible components. The first letter of a search substring is always set to upper case. Must have at least two letters.

service area. An area of memory used by the Linguistic Tools to store information which it needs to remember between calls.

simple tokenization. The initial assignment of portions of the input data to tokens according to the placement of delimiter characters in the input data. In general, a simple token will correspond to a word in the input text. See also **lexical analysis**.

simple word. A word in a specified language which can not be decompounded in that language.

simplex word. A word in an Linguistic Tools dictionary whose BOFA flag defines it as a valid word in itself. Also called standalone.

soft hyphen. A hyphen character (ASCII 240) which is inserted in a word solely for the purpose of fitting part of the word into the remaining spaces in a line of text. Soft hyphens do not occur naturally within a word. Also called **syllable hyphen**. See also **hard hyphen**, **required hyphen**, **syllable hyphen**.

standalone. See **simplex word**.

stem (in Linguistic Tools dictionaries). The set of characters common to the beginning portion of every term in a set of morphologically related words. Suffixes may be added to it to create related forms, or variants. Within an Linguistic Tools dictionary each stem has a number assigned to it for use in relating the words in the parallel tables to the base dictionary.

stop word. A word, typically very frequently used, which cannot be used to identify text content for information retrieval. This category includes all of the so-called "function words" such as pronouns, articles, conjunctions, and prepositions, as well as often semantically-empty words such as *be, have, do, make, item, thing,*.

syllable hyphen. A character that represents the point where a word has been hyphenated for line justification purposes. It may or may not be displayed, depending on its context and on the application. It is sometimes called a weak or soft hyphen. The GCGID for the syllable hyphen is SP320000 and the values of the character in some common codepages are:

- Latin I 500, ver. 1 - hex CA
- Latin I 850 - hex F0
- Greek 875 - not in codepage
- Greek 851 - hex F0

See also **soft hyphen**, **hard hyphen**, **required hyphen**.

synonym. A word which has the "same" meaning as the input word.

T

and so on.

tense (grammatical). Grammatical category indicating time of an event relative to time of speech. English distinguishes past (event precedes speech), present (event and speech occurring at the same time), and future (event has not yet occurred at time of speech). Other languages may or may not make the same distinctions. Many, but not all, languages change the spelling of the verb to mark tense.

terminating punctuation. Punctuation character that generally marks the end of a sentence, such as a period or question mark. Linguistic Tools recognizes the characters listed in Table 2 on page 33.

text segment. A sentence or fragment thereof that can be identified as a unit in the input text.

text segment identification. The process of identifying sentences or other textual units on the basis of punctuation and other information in the input data.

thread. An asynchronous unit of execution within a process.

token. A unit of output data used by Linguistic Tools functions. Each token contains part of the information about the text that was processed. In general, a token will represent a word in the input text. Tokens may be linked together in the form of a **token list**.

token list. An ordered collection of **tokens**. The token list can be interrogated using the token list utility functions.

tokenization. The process of assigning portions of the input data to objects (**tokens**) which can be processed and output by Linguistic Tools. See **simple tokenization**, **complex tokenization**.

trailing word. A word which follows a prefix and is joined to that prefix with an apostrophe. Example: in *l'enfant* "l" is the prefix and "enfant" is the trailing word.

U

ultra-high frequency dictionary. A very small set of the most frequently-occurring words in the language. On average approximately 50% of the words in ordinary business text are to be found in the ultra-high frequency dictionary. Used by Linguistic Tools to speed up spell checking. See also **high-frequency dictionary**.

user. The person who uses the application and to whom the application interfaces.

user dictionary. See **addenda dictionary**.

V

valid compound word. A word in a given language which is formed from dictionary entries in dictionaries of that language in accordance with both the BOFA rules and the orthographic rules of that language.

variant (in Linguistic Tools dictionaries). A form of a word made by adding a suffix to a stem word. Within an Linguistic Tools dictionary each variant has a number assigned to it which is used along with the number of its stem in relating the words in the parallel tables to the base dictionary.

W

word. A contiguous string of **non-delimiter** characters. Characters that may not appear in words (and therefore delimit them) are called **delimiter** characters. The simple tokenization table is used to determine whether a character (byte) is a delimiter.

word expansion option. Option to return inflected forms of the input word. (Used by text extraction functions.)

word isolation. The process of identifying words as textual units on the basis of delimiter characters in the input data. See also **simple tokenization**.

word order number. Number indicating the position of a word within a sentence. (Used by text extraction functions.)

word reduction option. Option to return the base form of the input word. (Used by text extraction functions.)

X

Y

Z

zì. The Chinese term for a single character in written Chinese. A word (**cí**) may consist of one or more **zì**. See also **hanzi**.

Index

A

abbreviations 33, 35
activating dictionaries 13, 19, 115
addenda dictionaries 17, 26, 27, 29, 49, 100
  add morphology to addenda 19, 23, 128
  add word to addenda 19, 124
  binary addenda 19
  create addenda dictionary 18, 121
  flat file format 18, 20
  header 18
  list addenda words 25, 132
  look up word 25, 134
  portability 26
  remove word from addenda 25, 130
  save addenda 26, 137
alias 22, 24, 31, 32, 126, 129
alphabetization 269
apostrophe 21, 40
article checking 27

B

base functions 13
BOFA 56
build 104

C

cang jie 284
case matching 23, 82, 141, 217
character codes 253
chuyinfuhao 284
code page 21, 78, 113, 127, 221
  code page-related tables 78
code points 253
collating sequence 269
compiling 104
complex token analysis
  See lexical analysis
compound word processing 45, 207
  Germanic compounds 45, 47
  Japanese compounds 46
constant definitions 104
continue reply 86
Control Block 60—69
CPOS (Chinese part of speech) codes 136, 189, 198

D

data elements 45, 69—76, 77, 205
  block format 70, 74
  single-word format 70, 71
data elements *(continued)*
  structure of 74
data types 60, 104
date format 299
DBCS 23, 47, 78, 96, 244, 248, 284, 321, 328
DBPC (DBCS parsing codes) 31, 48, 56, 124, 136, 286, 287, 324, 327, 330, 331
DBPC (Double-Byte Parse Codes) 189, 198
DBPC (double-byte parsing codes) 22
deactivating dictionaries 14, 19, 118
dehyphenation 29, 154
delimiter characters 34, 90, 91, 215, 219, 223
  sentence delimiters 34
  word delimiters 80
deliverables 103
dictionaries 55, 100
  See also addenda dictionaries
dictionary file names
  default extensions 108
dictionary files 211
dictionary handle 13, 100
double byte character set
  See DBCS

E

elision 46
enclitics 313, 337
entry points 59

F

file names 106—108, 226
  fully-qualified file names 107
fuzzy spelling aid 27

G

generating inflected forms 31, 171
Germanic languages 45
grade level 147
grade level analysis 28

H

hangeul 328
hanja 328
hanzi 284
high-frequency dictionary 55
hiragana 321
hyphenated forms 39, 43, 296
hyphenation 22, 28, 126, 151
  algorithmic 28

© Copyright IBM Corp. 2000

hyphenation *(continued)*
  dictionary-based 28
  preferred hyphenation points 28
  ranked hyphenation 22, 28
hyphens 21
  hard hyphen 21, 40, 43
  soft hyphen 22
  syllable hyphen 22, 40

I

include files 104
inflected forms 31, 32, 41, 174
  See also generating inflected forms
inflected synonyms 32, 174
  See also inflected forms
inflection 31
initializing Linguistic Tools 13, 113
input data 60
input dictionary list 13, 14, 100
input field names 60
input text 69
input text format
  See data elements

J

JAIRS part of speech filtering 189, 198, 324
jamo 328
JPOS (Japanese part of speech) codes 31, 136, 189, 198, 324

K

kana 321
kanji 321
katakana 321
keyword extraction 41, 185, 286, 324, 330
  phonetic key 41
keyword order number 42

L

language-specific processing 19, 43, 47, 281—345
  Arabic 264, 281
  Catalan 282
  Chinese 189, 198
  Chinese (Simplified) 47
  Chinese (Traditional) 47, 284
  Danish 290
  Dutch 291
  English 294
  Finnish 305
  French (National and Canadian) 306
  German (National) 309
  Greek 257, 313
  Hebrew 266
language-specific processing *(continued)*
  Hungarian 315
  Icelandic 316
  Italian 317
  Japanese 42, 47, 189, 198, 321
  Korean 47, 328
  Latin II (Latin 2) 261
  Norwegian 335
  Portuguese (National and Brazilian) 337
  Russian 259, 343
  Spanish 342
  Swedish 344
  Swiss German 312
  Thai 267
  Turkish 263
languages supported 10, 49
lemma 30
lexical analysis 37—38, 181

M

macro POS code 324
memory management 109
mixed case 82, 217
morphological identification 30, 167
morphology 23, 30, 31
  morphology mask 31, 347

N

NlpFindDicts 211
NlpGetComponent 231
NlpGetContent 232
NlpGetFirstProperty 234
NlpGetNextProperty 236
NlpGetNextToken 238
NlpGetNumComponent 239
NlpGetNumTrail 240
NlpGetPrevToken 241
NlpGetPval 242
NlpGetString 243
NlpGetStringAddr 245
NlpGetStringFlag 246
NlpGetTokenLen 247
NlpGetTokenType 249
NlpIsDelimiter 215
NlpMixedMode 217
NlpQryDelimTable 219
NlpQrySearchPath 220
NlpRegisterCodePage 221
NlpSetDelimTable 223
NlpSetDictDef 226
NlpSetSearchPath 228
normalization 44

O

operating systems supported 9
  AIX (IBM UNIX) 9
  HP-UX (Hewlett-Packard UNIX) 9
  Macintosh 9
  OS/2 Warp 9
  OS/390 OpenEdition (OS/390 UNIX System Services) 9
  OS/400 9
  Sun Solaris (Sun Microsystems UNIX) 9
  Windows 95 9
  Windows 98 9
  Windows NT 9
output data 60, 77
output field names 60

P

paradigm 30
PCODE 30
period 21, 35
pinyin 284
program build 104
properties 38, 95
punctuation 33
  closing punctuation 34
  terminating punctuation 34

Q

query term extraction 41, 194

R

reentrancy 106
reply area 76, 77, 86
  size of 86
request type 86
return codes 104, 277
romanization 284

S

SAA 59
search path 108, 220, 228
sentence separation 33
  See also text segment identification
service area 13, 69, 77
Service Area Handle 13, 69, 77
shift-out/shift-in characters 96, 98
simple tokenization 33, 80, 177
single-word data element creation 205
slash 21, 40, 298
sort sequence 269
spelling aid 27, 142
spelling verification 27, 139
stopwords 41
structures 104
synonym aid 29, 157
synonyms 22, 29

T

terminating the service 13, 114
text extraction
  See keyword extraction
text segment identification 33, 179
token list 89
token list utility functions 91
tokens 89
  basic tokens 90
  characteristics of 89
  components of 90
  joined tokens 90
  properties of 95

U

ultra-high frequency dictionary 55
uppercase 217
user data 74, 91
utility functions (general) 14

W

word (definition of) 80
word isolation
  See simple tokenization

Tell Us What You Think!

IBM Dictionary and Linguistic Tools
Application Programming Interface Description
Version 2.7

Publication No. SC30-4039-00

We hope you find this publication useful, readable, and technically accurate, but only you can tell us! Your comments and suggestions will help us improve our technical publications. Please take a few minutes to let us know what you think by completing this form. If you are in the USA, you can mail this form postage free or fax it to us at 1-800-253-3520. Elsewhere, your local IBM branch office or representative will forward your comments or you may mail them directly to us.

Overall, how satisfied are you with the information in this book?	Satisfied	Dissatisfied
	☐	☐

How satisfied are you that the information in this book is:	Satisfied	Dissatisfied
Accurate	☐	☐
Complete	☐	☐
Easy to find	☐	☐
Easy to understand	☐	☐
Well organized	☐	☐
Applicable to your task	☐	☐

Specific comments or problems:

Please tell us how we can improve this book:

Thank you for your comments. If you would like a reply, provide the necessary information below.

Name

Address

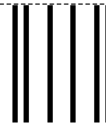
Company or Organization

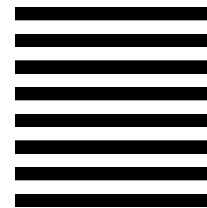
Phone No.

Tell Us What You Think!
SC30-4039-00

Cut or Fold Along Line

Fold and Tape | **Please do not staple** | Fold and Tape





NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Design & Information Development
IBM Corporation
Software Reengineering
Department G71A/ Bldg 503
P.O. Box 12195
Research Triangle Park, NC 27709-9990

Fold and Tape | **Please do not staple** | Fold and Tape

SC30-4039-00

Cut or Fold Along Line

Printed in U.S.A.