FEX User Guide
Version 1.2

Chad Cumby, Wen-Tau Yih

April 21, 2003

Part 1: Basics

Fex is a 'feature extractor' utility which can process a prepared corpus and produce a file of SNoW compatible examples.
The basic command-line for invoking Fex is as follows:

```
fex [options] script-file lexicon-file corpus-file example-file
```

The 4 files must all be specified, except in "server" mode in which case only the first 2 are specified. The script file contains a set (1 or more) of feature generation specifications. The corpus file contains an input corpus prepared in a specified format from which examples will be generated. The lexicon contains the mappings from features to IDs used by SNoW. And the example file is an output file where the generated examples are written.

The script file is formatted with each feature generation specification on its own line in the file. Each specification must be a valid sentence in the feature extraction language as described below in part 2.

The feature extractor uses the script to generate a single example for each occurence of a 'target' in the corpus. The example is formed by taking the union of all features generated by the script.

Input Formats

There are two main input formats that Fex will accept. The first is a general linear format where the corpus file should contain the prepared input with a single 'sentence' per line. When generating examples, Fex never crosses line boundaries of the corpus file. So even if the window specification is 'wider' than the current line, it does not 'wrap' to the next (or previous) line. This also means that it is possible to use Fex to produce examples which span several sentences, given that they are combined into a single line. Fex could then be used to extract features for Information Retrieval tasks.

This general linear format can take one of 3 forms. The first form is simple text, each white space delimited sequence of characters is treated as a word. Tag sets are empty. No special treatment of punctuation is provided, so this must be done by some form of preprocessing if necessary.

The second form is tag/word pairs. Each pair consists of an open parenthesis, the tag, the word and a closing parenthesis.

The third form is a more general form of the second in which rather than a single tag and word there is a tag set and a word set. A semi-colon ';' delimits the end of the tag set from the start of the word set.

Since the parenthesis and semi-colon are tokens to the fex parser, if they are to be included in a word or tag they are "escaped" with the back-slash '\' character. Hence the back-slash itself must be escaped. In general any character immediately following a back-slash is treated as a literal character and not a special token.

Finally, the input may include any combination of form 1, 2 and 3 input for each index of the input sentence. For example the following is legal input to fex:

```
w1 w2 (p1 w3) w4 (p1 p2 p3; \)\;) (p4 p5; w6 w7) w8 (.  .)
```

which is parsed as follows:

index	tags	words
0	`<empty>`	w1
1	`<empty>`	w2
2	p1	w3
3	`<empty>`	w4
4	p1 p2 p3	);
5	p4 p5	w6 w7
6	`<empty>`	w8
7	.	.

The second main input format is a column based table of records, with each record corresponding to a single word or data element. Each sentence or example in the corpus is denoted by a set of records separated by blank lines. In this way it is similar to the ouput produced by the FDG and ILK parsers. The following is an example of a valid sentence in this input format:

0001	1	0	I-NP	NNP	Pierre	NP-SBJ	join	8
0001	1	1	I-NP	NNP	Vinken	NP	join	8
0001	1	2	O	COMMA	COMMA	NOFUNC	Vinken	1
0001	1	3	I-NP	CD	61	NP-SBJ	years	0
0001	1	4	I-NP	NNS	years	NP	old	5
0001	1	5	I-ADJP	JJ	old	ADJP	Vinken	1
0001	1	6	O	,	,	NOFUNC	Vinken	1
0001	1	7	I-VP	MD	will	AAA	join	8
0001	1	8	I-VP	VB	join	M-V	join	-1
0001	1	9	I-NP	DT	the	NOFUNC	board	10
0001	1	10	I-NP	NN	board	NP	join	8
0001	1	11	I-PP	IN	as	ADJ	join	8
0001	1	12	I-NP	DT	a	NOFUNC	director	14
0001	1	13	I-NP	JJ	nonexecutive	NOFUNC	director	14
0001	1	14	I-NP	NN	director	NP	as	11
0001	1	15	B-NP	NNP	Nov.	NOFUNC	29	16
0001	1	16	I-NP	CD	29	NP-TMP	join	8
0001	1	17	O	.	a=b	NOFUNC	join	8

Each column of the input corpus corresponds to an attribute of a single data element, in this case a word. As Fex is designed to process language data, each column has a set name derived from the type of data placed in that column by the ILK parser. This named field is used by the Feature Extraction Language described below to extract features. The first three fields are header fields with the first two defined as dummy fields and the third as the index of the word in the senetence. The next 7 fields are used by Fex to extract features. They are: *Phrase*, *Tag*, *Word*, *Role*, *Role*

Pointer Value, and *Role Pointer* respectively. However, in general other data could be placed in each of these columns depending on the problem domain, as long as the user realizes that the same field names must be used in the Feature Extraction Language.

Above, the first record in the corpus contains information about the word with index 0. The Phrase field contains the value "I-NP" to designate that it is part of a noun-phrase. The Tag field contains the value "NNP" to designate that the word is a proper noun. The Word field contains the value "Pierre" as that is the word the record refers to. The Role field contains the value "NP-SBJ" to designate that the word is the subject of the sentence. The Role Pointer Value field holds the value "join" and at this point is not actually used, because the Role Pointer field holds the same basic information. The Role Pointer field contains the value 8 because as the subject of the sentence, in a verb-rooted parse tree, it would point to the main verb of the sentence, which is index 8.

Each column of the table must be filled for each record if the input is to be parsed correctly. Thus a special null-token is defined. This null-token can be indicated for any column with the word "NOFUNC"

Part 2: Extraction Language

Fex operates on a representation of an *observation*. Currently, an observation is a list of *active* records, where each record contains k fields. Each of the fields is a *set* of a variable number of values. The possible fields are called: *Word*, *Tag*, *Phrase*, *Role*, and *Role-Pointer* respectively.

Given an input of this form, Fex evaluates Feature Functions. Namely, given an observation and a collection of relational definitions that define *types* of features, Fex computes the instantiation of the Relation Generation Functions that are active in the observation, indexes them, and re-writes the observation as a list of active features.

The purpose of this section is to define the language used in the Relation Generation Functions, its interpretation, and a set of relational definitions that is currently used.

The basic syntax of a sentence in the feature extraction language is as follows, with words in square brackets (`[]`) optional:

```
targ [inc] [loc]:  RGF [[left-offset, right-offset]]
```

`targ` - Target index or word (potentially a relation). If `-1` is specified, fex will treat each record in the sentence as a target and extract features for that record.

`inc` - Use the actual target (word or tag) instead of the generic place-holder ('*'), so that features match only for the exact target.

`loc` - Include location of feature relative to target.

`RGF` - A valid Relation Generation Function defined in terms of a Basic RGF or a Complex RGF composed of Basic RGFs. (described below)

`left-offset` - Left edge of window for generating features. Negative values are left of target, positive to the right. If the left and right offsets are omitted in a feature definition, Fex will extract features from each record in the entire sentence with respect to the target.

`right-offset` - Same as left-offset except specifies the right edge.

Basic Relation Generation Functions

The basic building block for a RGF is a unary *Sensor*. A Sensor operates on a single record with respect to a specific location, and evaluates to either True of False. When it evaluates to True (namely, the feature type is active in the observation) it outputs the instantiation of the feature type in the observation. Examples of Sensors:

Type Definition	Mnemonic	Interpretation	Output
Word	w	always active	`w[`*word in current position*`]`
Tag	t	active if Tag is supplied	`t[`*tag in current position*`]`
Role	r	same	`r[`*role in current position*`]`
Vowel	v	active if the Word in the current record starts with a vowel	`v[]`
Prefix	pre	active if the Word in the current record starts with one of the patterns in a given list.	`pre[`*The active prefix*`]`
Suffix	suf	As above	`suf[`*the active suffix*`]`
Base_tag	base	active as the baseline tag of the Word in the current record. (requires a file that has a list of baseline tags indexed by words)	`base[` *active base_tag*`]`
Lemma	lem	active as the lemma of the Word in the current record	`lem[`*active lemma*`]`
Phrase	phr	produces active features for each Phrase attribute present in the sentence	`phr[`*active phrase]*

In addition, the user has the ability to designate a specific instantiation of the Sensor as a parameter, which will only generate features when that instance is present in the current record.

For example, the feature `w(boat)` will be active if the current record contains the word *boat*. Such a parameter can be supplied to any of the Basic RGFs like so:

```
-1 :  w(x=boat)
```

The **lab** RGF is a special function that takes any RGF as an argument and produces a feature with a `lab` tag appended to the feature produced by the argument RGF. This feature will have a lower feature ID (1-1000), and can be used to distinguish a target feature from the normal feature if multiple targets appear in the same sentence.

There are potentially many other unary Sensors that can be defined on an observation.

Complex RGF's

In addition to the Basic Relation Generation Functions, four Complex RGF's have been defined as compositions of the unary feature functions.
They are:
the conjunction operator '&', the disjunction operator '|', *coloc*, and *scoloc*.

`&` - Used to specify a feature which is active only in a record only when two or more Basic Relations are active. For example, the relation

  `w&t`
  is active as the combination of a record's Word and Tag and is written in the lexicon as:

  `w[`*word active in record*`]&t[`*tag active in record*`]`

| - Used to specify the union of the sets of features which are active when any of two or more Basic Relations are active within the record. For example, the relation

  `w|t`
  is active as the both the record's Word and the record's Tag and *two* features are written into the lexicon as:

  `w[`*word active in record*`]`
  `t[`*tag active in record*`]`

`coloc` - Specifies a consecutive collocation of RGF's active over two or more records. This RGF takes a parameter in the form of a list of DNF expressions defined using the & and | operators and other RGF's. For example:

  `coloc` $((R_1|R_2),\ R_1)$

  outputs consecutive conjunction of size 2, of the form R_1 R_1 or R_2 R_1 (that is, in the i-th word R_1 is active and in the i+1 word R_1 is active, or in the i-th word R_2 is active and in the i+1 word R_1 is active).

`scoloc` - Sparse collocation of RGF's active over two or more records. Parameter same as above.

Examples

In this section several different script specifications will be shown along with the features generated by each specification. For all examples features will be extracted from the following sentence:

```
(t0 the) (t1 old) (t2 man) (t3 went) (t4 to) (t5 the) (t6 store)
```

<u>Example 1</u>
```
2:  w [-2,2]
```
features:
`w[the], w[old], w[went], w[to]`

<u>Example 2</u>
```
2:  w&t [-2,2]
```
features:
`w[the]&t[t0], w[old]&t[t1], w[went]&t[t3], w[to]&t[t4]`

Example 3

```
the:  w|v [-2,2]
```

features:

```
w[old], w[man], w[went], w[to], w[store], v[+]
```

Example 4

```
2 loc:  coloc(w,t) [-2,2]
```

features:

```
w[the_*]-t[t1*], w[old*]-t[*], w[*]-t[*t3], w[*went]-t[*_t4]
```

Example 5

```
2 inc loc:  coloc(w,t) [-2,2]
```

features:

```
w[the_*man*]-t[t1*t2*], w[old*man*]-t[*t2*], w[*man*]-t[*t2*t3],
w[*man*went]-t[*t2*_t4]
```

Example 6

```
2 inc:  scoloc(w,t) [-2,2]
```

features:

```
w[the]-t[t1], w[the]-t[t2], w[the]-t[t3], w[the]-t[t4], w[old]-t[t2],
w[old]-t[t3], w[old]-t[t4], w[man]-t[t3], w[man]-t[t4], w[went]-t[4]
```

Part 3: Command-Line Options

`-h histogram file` : Produce Histogram file with counts for each word in the data.

`-i historgram file` : Produce Histogram file with counts by feature.

`-j statistics file` : Produce Statistics file with Chi^2 values for each feature.

`-p` : Column input format. If set, the input corpus will be parsed as the second column-based input format detailed in section 1.

`-P length` : Activate Phrase Extension. See Appendix A.

`-r` : Read-only Lexicon. Only features already present in the Lexicon will be written to examples, and the Lexicon will not expand to include new features.

`-s port` : Server Mode. Fex will run in server mode accepting socket connections on the given port number. *Do not* provide a corpus filename when running in this mode. When connecting to the Fex server, provide input sentence by sentence in this format:

`-S stopwords file` : Stop words. use this option with a stopword file to designate a list of words which Fex will ignore when creating features.

`-t target file` : Use target file instead of script specification to determine targets and target indices to use in extracting features from each sentence. A target file must consist of a list of target words each on its own line of the file.

`-T target file` : Modified target file option. Like above, except that each target in the targetfile will be used in only one example corresponding to that target's position in the sequence.

`-u` : Set raw input option. If set, the input corpus will be parsed using the linear input format detailed in Section 1, but each string or token will be parsed as a word.

`-v verbosity` : Set verbosity. options are:

- `max` - maximum verbosity: script structure, target indices, features extracted, and examples constructed will all be displayed to standard out along with the basic version info.
- `med` - medium verbosity: script structure, and examples constructed will be displayed to standard out along with the basic version info.
- `min` - minimum verbosity: only the basic version info will be displayed.
- `off` - no information or output will be displayed to standard out.

Part 4: Tutorial

This tutorial is designed to guide the user through a practical application of the Fex feature extractor and the corresponding feature extraction language. It will include an explanation of how to set up training and test corpora, creation of a script, and the output written to the lexicon and example files - all from the perspective of a part-of-speech tagger. Through this demonstration, the concepts behind the specific constructions used in the feature extraction language should become clear.

POS Tagging

Part-of-speech tagging is the process of assigning each word in an input sentence its correct part-of-speech in context. The task of Fex in this process is to convert the initial word-based representation of the sentence into a feature-based representation recognizable by SNoW. Two sets of examples must be generated - one for training and one for testing. Thus, the first step is to produce the example file

```
traindata.feat
```

To train a POS predictor, it is necessary to produce a separate example for each word in the tagged training corpus, using the existing tag of each target word as the label for each example. To do this we create a new script file named

```
pos.scr
```

and there insert the specification line

```
-1:  lab(t)
```

The *-1* portion of the specification signifies that an example is to be created for each word in the input sentence. The *lab* portion is a RGF keyword specifying that a feature corresponding to a target label should be generated for each example. The *(t)* portion is an argument to that RGF specifying that the label generated should be the POS tag of the target word.

The next step in creating the training file is to choose the types of features that would be most useful in trying to learn the correct part-of-speech for a given target in a given sentence. For this, we look to (Roth & Zelenko, 1998) as a guide. In this study, the features used were described thusly:

1. The preceding word is tagged *c*.
2. The following word is tagged *c*.
3. The word two before is tagged *c*.
4. The word two after is tagged *c*.
5. The preceding word is tagged *c* and the following word is tagged *t*.
6. The preceding word is tagged *c* and the word two before is tagged *t*.
7. The following word is tagged *c* and the word two after is tagged *t*.
8. The current word is *w*.
9. The most probable part of speech for the current word is *c*.

All of these feature definitions can be easily represented in the feature extraction language of Fex.

To generate those features defined by rules 1-4 above, we add this line to the already existing script file:

```
-1 loc:  t [-2,2]
```

This specification indicates that Fex should generate features treating every word in the sentence as a target, by the *-1* token. Additionally, the Sensor RGF*t* indicates that the features generated should consist of the POS tags from the tagged training corpus. The range portion *[-2,2]* signifies that the POS tags extracted will be the tags two before, one before, one after, and two after the target position. The tag of the target position *does not* generate a feature since the *inc* flag is not included in the specification.

Rules 5-7 defined above requires its own line in the script:

```
-1:  coloc(t,t,t) [-2,2]
```

With this line, we designate that three term colocations of POS tags should be generated within a *(-2,2)* interval around the target position. The target position within each colocation will be represented by the "*" placeholder. For example, if the input sentence were:

```
(t1 The) (t2 car) (t3 drove) (t4 very) (t5 quickly) (.  .)
```

Then the features generated from this specification would be:

```
t[t1]-t[t2]-*
t[t2]-*-t[t3]
-t[t2]-t[t3]
```

For the eighth definition, we add this line: `-1 inc:  w [0,0]`

As before, the *-1* indicates that this feature is to be generated treating every word in the sentence as a target. The addition of the *inc* flag before the colon indicates that the target word will be used in the example instead of the general target-placeholder "*". The range of words extracted by the *w* Sensor RGFis limited to only the word in the target position by the specification of the *[0,0]* range.

In considering the ninth and final feature definition from *Roth & Zelenko* it becomes necessary to define the meaning of Baseline tags. The Baseline tag of a target word is simply the most common part-of-speech associated with that word. An external file must be prepared by statistical analysis from a tagged corpus, containing a list of words with their Baseline tags. This list should be of the format:

word tag

with each word-tag pair on a new line. Once such a file is prepared, it can be accessed by using the *base* Feature Function described in Part 2. The last feature definition then becomes:

```
-1 inc:  base [0,0]
```

This will produce the most probable POS tag for the target word, centered in the range *[0,0]*. The *inc* flag must be set because the feature is being generated for the target position.

Appendix A: Phrase Extension

This extension is added to provide support for extracting phrase-type features. The extension only accepts COLUMN format input. In addition, the first column is used to store phrase labels. The second column is used to store named entity tags. Both these two use BIO format.

Usage: `-P` *length*

-P takes one argument, which stands for the maximum length of the candidate phrases. For example, `fex -P 4` will generate examples for every phase of length 1, 2 ,3 and 4 from the corpus file. If length is equal to 0, then only positive examples will be generated.

Window Range

The meaning of offsets in window has been changed a little. Let me use the following example to explain it.

Example - corpus:
w1 w2 w3 W4 W5 W6 w7 w8 w9

Suppose that **W4 W5 W6** is the target phrase. When the window is before the target (both left offset and right offset are negative), it's the same as usual. In other words, location -1 is **w3**, location -2 is **w2**, and location -3 is **w1**. The positive location means the words after the target, so location +1 is **w7**, +2 is **w8**, and +3 is **w9**. If the window is [0,0], then it means words from the target phrase. For example,

```
-1:  w[0,0]
```

gives you word **W4**, **W5**, and **W6**.

When the window is outside the target phrase, the whole target is treated as a single location. Therefore,

```
-1 loc:  w[-2,-1]
-1 loc:  w[1, 2]
```

will give you:

```
w[w2_*], w[w3*], w[*w7], w[*_w8]
```

If the window is `[0,0]` and location information is included, then it will treat one word before the targer phrase as the target and put an asterik after the generated output to denote it's actually from the target phrase. For example:

```
-1 loc:  w[0,0]
```

will generate

```
w[*W4]*, w[*_W5]*, w[*__W6]*
```

Windows that overlap the target phrase such as `[-5,0]`, `[0,4]`, `[-2,2]` are a little bit tricky. In these cases, only the first word of the phrase is treated as the target location. If you want to use this type of windows, make sure the output is really what you need.

Changes in RGFs

The syntax of `lab` has been modified a little. The parameters of `lab` will be ignored. In addition, `lab` can have no parameter at all. That is, `lab(w)`, `lab(t)`, and `lab` are all the same.

Target files are not supported in this extension. Also, you can only use "-1" (all target) as the target indication in each RGF.

Two phrase type sensors *phLen* and *phNE* are added. *phLen* returns the length and *phNE* returns the named entity tag of the target phrase. Basically, the window for these two new sensors should always be `[0,0]`. For example, `-1 phLen[0,0]` returns 3 for the above corpus file, since **W4 W5 W6** contains 3 words.

A special complex RGF operator called *conjunct* is created. It takes any number of FULL RGF statements, seperated by semicolons, as parameters. The output is the results of each subRGF connected by "–". For example,

`conjunct(-1:w[-2,-1]; -1:phLen[0,0]; -1:w[1,2])` generates

```
w[w2]--phLen[3]--w[7], w[w2]--phLen[3]--w[8],
w[w3]--phLen[3]--w[7], w[w3]--phLen[3]--w[8]
```

Examples

corpus file, corp1:

B-subj	B-Peop	0	NOFUNC	NNP	I	NOFUNC	x	O
O	O	1	NOFUNC	NNP	am	NOFUNC	x	O
O	O	2	NOFUNC	NNP	dealing	NOFUNC	x	O
O	O	3	NOFUNC	POS	with	NOFUNC	x	O
B-term	B-Term	4	B-NP	NNP	Information	NOFUNC	x	O
I-term	I-Term	5	I-NP	NN	Extraction	NOFUNC	x	O
O	O	6	NOFUNC	VBD	problems	NOFUNC	x	O
O	O	7	NOFUNC	.	.	NOFUNC	x	O

corpus file, corp1:

B-subj	B-Peop	0	NOFUNC	NNP	I	NOFUNC	x	O
O	O	1	NOFUNC	NNP	am	NOFUNC	x	O
O	O	2	NOFUNC	NNP	dealing	NOFUNC	x	O
O	O	3	NOFUNC	POS	with	NOFUNC	x	O
B-term	B-Term	4	B-NP	NNP	Information	NOFUNC	x	O
I-term	I-Term	5	I-NP	NN	Extraction	NOFUNC	x	O
O	O	6	NOFUNC	VBD	problems	NOFUNC	x	O
O	O	7	NOFUNC	.	.	NOFUNC	x	O

script file, scr1:

```
-1:  lab
-1 loc:  w[1,2]
-1 loc:  w[-2,-1]
-1 loc:  w[0,0]
conjunct(-1:w[-2,-1]; -1:phLen[0,0]; -1:w[1,2])

fex -P 0 scr1 lex1 corp1 ex1
```

lexicon file, lex1:

```
1 label[subj]
1001 w[*I]*
1002 w[*_dealing]
1003 w[*am]
2 label[term]
1004 w[dealing]--phLen[2]--w[.]
1005 w[dealing]--phLen[2]--w[problems]
1006 w[with]--phLen[2]--w[.]
1007 w[with]--phLen[2]--w[problems]
1008 w[*Information]*
1009 w[*_Extraction]*
1010 w[dealing_*]
1011 w[with*]
1012 w[*_.]
1013 w[*problems]
```

example file, ex1:

```
1, 1001, 1002, 1003:
2, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011, 1012, 1013:

fex -P 2 scr1 lex2 corp1 ex2
```

lexicon file, lex2:

```
1 label[subj]
1001 w[*I]*
1002 w[*_dealing]
1003 w[*am]
2 label[IRRELEVANT]
1004 w[*_am]*
1005 w[*_with]
1006 w[*dealing]
1007 w[I]--phLen[1]--w[dealing]
1008 w[I]--phLen[1]--w[with]
1009 w[*am]*
1010 w[I*]
1011 w[I]--phLen[2]--w[Information]
1012 w[I]--phLen[2]--w[with]
1013 w[*_dealing]*
1014 w[*_Information]
1015 w[*with]
1016 w[I]--phLen[1]--w[Information]
1017 w[am]--phLen[1]--w[Information]
1018 w[am]--phLen[1]--w[with]
1019 w[*dealing]*
1020 w[I_*]
1021 w[am*]
1022 w[I]--phLen[2]--w[Extraction]
1023 w[am]--phLen[2]--w[Extraction]
1024 w[am]--phLen[2]--w[Information]
1025 w[*_with]*
1026 w[*Information]
1027 w[*_Extraction]
1028 w[am]--phLen[1]--w[Extraction]
1029 w[dealing]--phLen[1]--w[Extraction]
1030 w[dealing]--phLen[1]--w[Information]
1031 w[*with]*
1032 w[am_*]
1033 w[dealing*]
1034 w[am]--phLen[2]--w[problems]
1035 w[dealing]--phLen[2]--w[Extraction]
1036 w[dealing]--phLen[2]--w[problems]
1037 w[*_Information]*
1038 w[*Extraction]
```

```
1039 w[*_problems]
1040 w[dealing]--phLen[1]--w[problems]
1041 w[with]--phLen[1]--w[Extraction]
1042 w[with]--phLen[1]--w[problems]
1043 w[*Information]*
1044 w[dealing_*]
1045 w[with*]
3 label[term]
1046 w[dealing]--phLen[2]--w[.]
1047 w[with]--phLen[2]--w[.]
1048 w[with]--phLen[2]--w[problems]
1049 w[*_Extraction]*
1050 w[*_.]
1051 w[*problems]
1052 w[Information]--phLen[1]--w[.]
1053 w[Information]--phLen[1]--w[problems]
1054 w[with]--phLen[1]--w[.]
1055 w[*Extraction]*
1056 w[Information*]
1057 w[with_*]
1058 w[Information]--phLen[2]--w[.]
1059 w[*_problems]*
1060 w[*.]
1061 w[Extraction]--phLen[1]--w[.]
1062 w[*problems]*
1063 w[Extraction*]
1064 w[Information_*]
1065 w[*_.]*
1066 w[*.]*
1067 w[Extraction_*]
1068 w[problems*]
```

example file, ex2:

```
1, 1001, 1002, 1003:
2, 1001, 1004, 1005, 1006:
2, 1005, 1006, 1007, 1008, 1009, 1010:
2, 1009, 1010, 1011, 1012, 1013, 1014, 1015:
2, 1008, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021:
2, 1011, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027:
2, 1017, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033:
2, 1023, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039:
2, 1029, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045:
3, 1036, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051:
2, 1042, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057:
2, 1047, 1055, 1056, 1057, 1058, 1059, 1060:
2, 1052, 1060, 1061, 1062, 1063, 1064:
2, 1062, 1063, 1064, 1065:
```

`2, 1066, 1067, 1068:`

Appendix B: Relation Extension

This extension is used to handle relations between entities of which the boundaries are konwn. The data format it accepts is similar to the COLUMN format only with some slight changes. We introduce the data format, changes in scripts, and some examples in the following subsections.

Data Format

Suppose we are given this sentence:

> In 1904, bandleader Jimmy Dorsey was born in Shenandoah, Penn.

From this sentence, we know that the *birthplace* of a *person* ("Jimmy Dorsey") is a *place* ("Shenandoah, Penn."). In other words, we have the relation *birthplace*("Jimmy Dorsey", "Shenandoah, Penn."), and the entities *person*("Jimmy Dorsey") and *person*("Shenandoah, Penn.").

We use the following data format to represent this sentence.

corpus file, corp2:

O	O	0	O	IN	In	NOFUNC	x	O
O	O	1	NP	CD	1904	NOFUNC	x	O
O	O	2	O	,	,	NOFUNC	x	O
O	O	3	NP	NN	bandleader	NOFUNC	x	O
O	Peop	4	NP	NNP/NNP	Jimmy/Dorsey	NOFUNC	x	O
O	O	5	O	VBD	was	NOFUNC	x	O
O	O	6	O	VBN	born	NOFUNC	x	O
O	O	7	O	IN	in	NOFUNC	x	O
O	Loc	8	NP	NNP/,/NNP/.	Shenandoah/,/Penn/.	NOFUNC	x	O

4	8	birthplace

This representation contains four units:

- a sentence in column format (but the words and pos tags belonging to the same entity appear in one row)
- empty line
- relation descriptors (may be empty or more than one line)
- empty line

In the COLUMN format here, only four columns are meaningful:

- col-2: Entity class label (e.g. Peop, Loc)
- col-3: Element order number (starting from 0)
- col-5: Part-of-speech tags
- col-6: Words

Other columns can simply be ignored.

Note that the pos tags and words of an entity are put in the same row separated by '/' (e.g. "NNP/NNP" in row 4 and "Shenandoah/,/Penn/." in row 8). In other words, the boundary of each entity (either in training or testing data) are assumed predefined.

A relation descriptor has three fileds.

- 1st field : the element number of the first argument.
- 2nd field : the element number of the second argument.
- 3rd field : the name of the relation (e.g. birthplace).

Extract features for entities

The parameter for extracting features from entities in a relation/entity corpus is "-R e".

To extract features with respect to an entity, the syntactic change in scripts is to use "lab(ent)" intead of "lab", and the semantic change is obviously the respresentation of an entity. An entity is considers as a set of words. In other words, the order of the words in the entity is not preserved. It may be clear after seeing an example.

Suppose the script file is:

script file, scr2:

```
-1:  lab(ent)
-1 loc:  w[1, 0]
```

Applying `scr2` on `corp2`, Fex will generate the following lexicon and example files:

lexicon file, lex2:

```
1 label[ent[Peop]]
1001 w[Jimmy]
1002 w[Dorsey]
2 label[ent[Loc]]
1003 w[Shenandoah]
1004 w[,]
1005 w[Penn]
1006 w[.]
```

example file, ex2:

```
1, 1001, 1002:
2, 1003, 1004, 1005, 1006:
```

However, this also means that it is not possible (without inventing new sensors) to generate co-locations on words inside the entity. For example, RGF "-1 inc: coloc(w,w)[0,0]" will not return any (active) features.

For co-locations between the words in the target and other words, each word in the target will conjunct with words outside. For instnace, "-1 inc: coloc(w,w)[-1,0]" generates the following features.

```
w[badlander]-w[Jimmy]
w[badlander]-w[Dorsey]
w[in]-w[Shenandoah]
w[in]-w[,]
w[in]-w[Penn]
w[in]-w[.]
```

Extract features for relations

The parameter for extracting features from relations in a relation/entity corpus is "-R r".

Extrating features for (binary) relations is a little tricky. Let's first take a look at possible relation targets. A (binary) relation is defined by its two entity arguments. In `corp2`, there are two entities ("Jimmy Dorsey" and "Shenandoah,Penn."), which define two relations:

- R_1("Jimmy Dorsey", "Shenandoah,Penn.")
- R_2("Shenandoah,Penn.", "Jimmy Dorsey")

From the relation descriptor of `corp2`, we also know that the label of R_1 is "birthplace" and R_2 is "irrelevant".

The order of the relation targets is defined as follows. Suppose there are n entities in a sentence, and the entities are $E_1, E_2, \cdots, E_n$, where E_i appears before E_j if $i < j$. If we use R_{ij} to represent realtion (E_i, E_j). Then, the order of the relation targets is:

$$R_{12}, R_{21}, R_{13}, R_{31}, \cdots, R_{1n}, R_{n1}, \cdots, R_{(n-2)(n-1)}, R_{(n-1)(n-2)}, R_{(n-2)n}, R_{n(n-2)}, R_{(n-1)n}, R_{n(n-1)}$$

In order to fully specify positions with respect to these two argument entities, the index system is changed in the following way.

- 100 : the position of the first argument
- 200 : the position of the second argument
- -1ww : the ww-th position to the left of the first argument
- 1xx : the xx-th position to the right of the first argument
- -2yy : the yy-th position to the left of the second argument
- 2zz : the zz-th position to the right of the second argument

Take `corp2` as example. When the target relations are R_1 and R_2, Table 1 and 2 show some mappings between the location numbers and words.

A window that is defined by two numbers specifies the focus of the RGF. However, if the actual position of the left window index is to right of the actual position of the right window index, then no feature will be extracted.

For example, script "w[-101,201]" extracts words from one word before the first argument to one word after the second argument. If the target relation is R_1, then

location	words
100	Jimmy Dorsey
-102	,
101	was
200	Shenandoah,Penn.
-201	in
201	**not exist**

Table 1: R_1("Jimmy Dorsey", "Shenandoah,Penn.")

location	words
100	Shenandoah,Penn.
-102	born
101	**not exist**
200	Jimmy Dorsey
-201	bandleader
201	was

Table 2: R_2("Shenandoah,Penn.", "Jimmy Dorsey")

w[bandleader], w[Jimmy], w[Dorsey], w[was], w[born], w[in], w[Shenandoah], w[,], w[Penn], w[.]

will be extracted. However, if the target relation is R_2, then no feature will be extracted since the first word before the first argument (i.e. *in*) is actually after the first word after the second argument (i.e. *was*).

If an RGF has no window, the focus is then the whole sentence. For example, script "-1: Verb&w" extracts all the verbs in this sentence.

Another modification of the script is the use of "mark". If you put *mark* before the colon, then "12" will be added to the lexicon of the feature when the first argument appears before the second argument and "21" for the other case. For example, "-1 mark inc: t[-101,201]" extracts the following features for R_1 (Note that it can't extract any feature for R_2 because of the abovementioned reason.)

t[NN]12, t[NNP]12, t[VBD]12, t[VBN]12, t[NNP]12, t[,]12, t[.]12

and "-1 mark inc: t[-201,101]" extracts the following features for R_2

t[NN]21, t[NNP]21, t[VBD]21, t[VBN]21, t[NNP]21, t[,]21, t[.]21