Knowledgebase csv file formats

# Introduction

This document describes the format of the csv files for a language model or knowledgebase (KB) for iKnow. A KB consists of seven or eight csv files:

Content filename for development filename after compilation

* Abbreviations (\*) XX\_dev\_xx\_acro.csv acro.csv
* Filter (\*) XX\_dev\_xx\_filter.csv filter.csv
* Grammatical labels XX\_dev\_xx\_labels.csv labels.csv
* Lexical representations XX\_dev\_xx\_lexreps.csv lexreps.csv
* Metadata XX\_dev\_xx\_metadate.csv metadata.csv
* Pre-processor (\*) XX\_dev\_xx\_prepro.csv prepro.csv
* Rules XX\_dev\_xx\_rules.csv rules.csv
* Regular expressions XX\_dev\_xx\_regex.csv regex.csv

(optional) (\*)

(\*) *Not applicable for the Japanese model. The Japanese files are present, but they contain dummy items.*

In this text, “word” refers to a string of Latin characters surrounded by word separators (space, end-of-line, tab).

Some files have an ID number in the first field. The ID value is assigned automatically by the Delve tools. When creating new entries in a csv file, the ID field can be left blank.

Note: Originally, the KBs were stored in a database. Therefore, all files had IDs in field 1 and a version number in field 2. For compatibility reasons, the structure of the files has largely remained unchanged although the implementation has changed. That explains why some files have empty fields.

The “comment” fields can contain remarks or examples.

# Abbreviations

The abbreviations file contains words that change the default behavior regarding sentence endings. There are two types of abbreviations:

* Non-Sentence Separator. These are strings that end with a sentence-ending character, mostly a period. The engine would mistakenly interpret them as the last word of a sentence if they were not in acro.csv. Non-sentence separators are indicated by “0”.
* Sentence Seperator. These are words that do not end with a sentence-ending character but that should be treated as the end of a sentence. Sentence separators are indicated by “1”.

An item in acro.csv must be a complete word. The detection passes case-sensitive.

Syntax: <ID or empty field>;<empty field>;<entry>;<value 0/1>;<optional comments field>

Examples: ;;Mr.;0;

;;Abstract:;1;

# Filters

In a filter file, rewritings to be applied at the end of the processing can be defined. The filter works on the indexes of Concepts, Relations, PathRelevants and NonRelevants.

Beware that the filter operates on single lexreps, not merged lexreps. When, at the end of processing, lexreps are merged, they become a sequence of single lexreps.

From the TRACE information :

***ConceptFiltered****:<lexrep id=3 type=Concept value="Tratado de Não-Proliferação Nuclear." index="****tratado de não- proliferação nuclear****" labels="PTCon;PTNegation(a:Entity,Negation,);PTNPC;" />;****tratado de não-proliferação nuclear****;*

The filter entry from the Portuguese (“pt”) filter file :

*;;Cnão- ;**não-;0;to rewrite adjectives with 'não-' back together*

Is applied here, since “**tratado de não-proliferação nuclear”** is a single lexrep.

However, in the following Dutch (“nl”) example:

***MergedConcept****:<lexrep id=8 type=Concept value="niet-" index="****niet-****" labels="NLAttradj;NLNegWord(a:Entity,Negation,);NLNPcomp;Lit\_niet-;" />;<lexrep id=9 type=Concept value=" gecertificeerde" index="****gecertificeerde****" labels="NLAttradj;NLNPcomp;" />;<lexrep id=10 type=Concept value="****uitzendbureaus****" index="uitzendbureaus" labels="NLCon;NLNPcomp;" />;sum=0.000000;*

The filter entry from the Dutch (“nl”) filter file :

*;;Cniet- ;niet-;0;to attach prefix back to the concept*

Will not be applied here, since the “merged” lexrep is a sequence of 3 single lexreps : “niet-“, “gecertificeerde” and “uitzendbureaus”.

This behaviour can cause some confusion…

Syntax: <ID or empty field>;<empty field>;<search pattern>;<substitution>;<position>;<optional comments field>

Examples: ;;R -;-;0;to merge verbs with split-off personal pronouns;

;;Cno- ;no-;0;to merge adjectives with split-off no- again;

Search pattern: The engine looks for the search pattern within an entity (if the search pattern is preceded by C: a Concept, if preceded by R: a Relation, if preceded by P: a PathRelevant, if preceded by N: a NonRelevant, if none of these, the default type is Concept). The search pattern can contain spaces but no capital letter or characters that have been normalized or deleted during processing.

Substitution: The substitution must be a string of characters that can occur in index values (no capital letters). It can be empty.

Position: A rewriting can be applied

* + - anywhere in an index, indicated by position value 0
    - at the beginning of an index, indicated by position value 1
    - at the end of an index, indicated by position value 2
    - at the beginning and at the end of an index, indicated by position value 3

# Grammatical Labels

Grammatical labels are used to label lexical representations (lexreps, see below). They are also used in rules to represent lexreps.

Syntax: <ID>;<phase number(s)>;<label name>;<label type>;<comment 1>;<0/1>;<comment 2>;<path-related info(only for certain attributes)>

Examples:

1;1,20,25,26,40,50,55,60,62,70,71,75,80,81,$;ENCon;typeConcept;concept;1;;

24;3,$;ENAux;typeRelation;auxiliary verbs;0;;

Phase number: Rules are organized in blocks or phases. The phases in which a certain label is used in the search pattern of a rule have to be listed in the second field of the label file, separated by commas. The last, default phase is represented by the dollar sign. All labels on this phase will finally decide the type of the entity (Concept, Relation, PathRelevant, NonRelevant). A label not defined on the last phase will be ignored, see the description of the rules on how this final decision mechanism works.

Label name: There are a number of mandatory labels and attributes (Concept, Relation, Unknown, Punctuation, NonRelevant, CapitalAll, CapitalInitial, CapitalMixed, NonSemantic) that are built in and must not be defined in the labels file. These built-in labels are only usable in phases 1 and $ (= the last phase, 99). If these labels are needed in other phases, extra language-specific labels have to be created for that purpose.  
The labels “SBegin” (beginning of sentence), “SEnd” (end of sentence), “Join” (merge lexreps) and “-“ (split lexrep) are built in too, but they can be used in all phases. It is not necessary to define language-specific variants of these labels.  
The labels Subject, Object and PrimaryRelation have to be present in the labels file for all languages, even when they are not used. For these labels, it is possible to indicate in which phases they must be active.   
  
Per our convention the language specific labels must have names that start with the two-letter language code in capital letters, followed by a name with a capital initial, e.g. ENPrep, DEAdj, FRInf. The name of the label should reflect its meaning: a part-of-speech (e.g. ENAdv) or specific word (e.g. ENNo).

A label name cannot contain spaces. If a label expresses in its name that a lexrep is grammatically ambiguous, we use capital letters for the different parts, e.g. ENVerbCon -> verb or noun. If two functions are both present in one word, or if one part of the names specifies the other part, the different parts don’t get capitals, e.g. DEPrepart -> contraction of preposition and article in one word, ENPosspron -> possessive pronoun. However, we are aware of some inconsistencies in the actual models.

Label type: Each label must have a type. There are 11 predefined types:

* + - typeConcept
    - typeRelation
    - typePathRelevant
    - typeBeginConcept
    - typeBeginRelation
    - typeEndConcept
    - typeEndRelation
    - typeBeginEndConcept
    - typeBeginEndRelation
    - typeOther
    - typeAttribute

Apart from these types, it is possible to use other type names, e.g. “typeAmbiguous”. All self-defined type names will be treated as typeOther, i.e. as NonRelevant elements.

Types play a role in entity merging and final disambiguation. The final decision on the entity type depends on the remaining labels on the last phase of the rules processing (phase 99 or ‘$’). See “Final Entity Type Selection” in the rules description fro details.

Entity merging: consecutive Concepts and Relations are merged together. It is, however, possible to impose a ‘border’ when applying typeBeginConcept/typeEndConcept/typeBeginEndConcept labels for Concepts, and typeBeginRelation/typeEndRelation/typeBeginEndRelation labels for Relations.  
e.g. typeConcept | typeConcept => 1 Concept  
 typeConcept | typeBeginConcept => 2 Concepts.

typeAttribute: these labels express extra information that is specific for the individual lexrep apart from its role as Concept, Relation, PathRelevant or NonRelevant, e.g. is the word capitalized, is the word a negation marker, etc. There are two categories of attributes. The first category is automatically assigned by the engine, e.g. the labels for capitalized words, SBegin or SEnd. The second category consists of manually added attributes. These can either be attributes linked to a known attribute with expansion: Negation, Sentiment, Time or Measurement; or they can be any attribute you want to define for use in rules.

0/1: One grammatical label has to be marked as the default label, i.e. the label that gets assigned to all words that are not known lexreps. This label must have value “1” here. In theory, any label can be marked as the default label, but in practice we choose the label that is used for nouns, which is typically a typeConcept label, e.g. ENCon.  
All other labels get value “0” here.

Comment 2: Extra comment field, mostly empty.

Path-related info: This field is only used for typeAttribute labels linked to a known ‘semantic’ attribute with expansion, i.e. Negation, Sentiment, Time, Frequency, Duration, Measurement and Certainty. It is not possible to just define path-related information for other semantic attributes that you might want to add.

Examples:

112;20,50,60,70,71,$;ENNegation;typeAttribute;negation marker;0;;Entity(Negation)

107;20,55,60,70,71,$;ENNegStop;typeAttribute;to end the negation span;0;;Path(End,Negation)

108;60,$;ENNegBegin;typeAttribute;to begin the negation span;0;;Path(Begin,Negation)

# Metadata

This file contains general indexing settings for the concerned language model.

Syntax: <name of setting>;<value>

Example: ConceptsToMergeMax;6  
 ModifiersOnLeft;1  
 LanguageCode;nl  
 PathConstruction;PR;

The Japanese file has more and different entries.

# Lexreps

Lexical representations, or “lexreps”, are words and word groups. The lexreps file is a list of lexreps with their labels. A lexrep can have more than one label.

There are 3 types of lexrep representations : regular expression references, token generators, and normal words.

“;;{number-moreless};;ENNumber;”

Here, “number-moreless” refers to a regular expression, that is defined in regex.csv.

“;;0(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5)(0|1|2|3|4|5|6|7|8|9) (gmt|cet|cst|ct|edt|est|et);;ENAdvTime;”

This is a token generator, all combinations in between “(“ and “)” will be generated.

“;;a few more;;ENQuantity;”

This is a normal lexical expression : BEWARE, because of the special meaning of ‘(‘ and ‘)‘, you need to escape these symbols if needed in a normal lexical expression, use the ‘\’ character :

;;a \(few\) more;;ENQuantity;

This matches the lexical expression: “a (few) more”.

There are more symbols to “escape”: the special meaning of “{“ = start of regex expression must also be escaped if the literal “{“ is needed (“\{“, and corresponding “\}”).

And last of course the escape symbol itself, if “\” is meant, use “\\”.

If a lexrep consists of more than one word, there are several labeling options:

* The words can be considered as one lexrep with one or more labels,   
  e.g. board member -> ENCon.
* The words can be considered as a sequence of lexreps, linked to a series of labels,  
  e.g. live music -> ENAdj;-;ENCon;
* The words can be grouped within the lexrep,  
  e.g. ;;a couple of;;ENArtPosspron;-;ENQuantity;Join;  
  meaning that “couple of” will become one lexrep with label ENQuantity, “a” remains a separate lexrep with label ENArtPosspron.

Basic syntax: <empty field>;<metadata or empty field>;<lexrep>;<comment>;<label 1>;(<label 2>;<label3>;…)

Examples: ;;architectural;;ENAdj;

;;archive;;ENInfCon;

;;archived;;ENPart;

;;archives;;ENVerbCon;

;;archiving;;ENPrespart;

;;archly;Adverb of manner;ENAdv;ENAdvmandeg;

An artificial lexrep is added at the head of the sentence (named “B”), and another one at the tail (named “E”). They are labeled automatically by the engine (SBegin/SEnd), and can be used in lexrep lookup, if making a positional difference is necessary. They should however keep their original label, and remain isolated (they are removed afterwards.

There can be no confusion with “normal” lexreps, since these are always normalized (=lowercase).

Examples:

;;B in effect;example of Begin lexrep;SBegin;-;ENAdvstruct;Join;

;;after all E;example of End lexrep;ENAdvstruct;Join;-;SEnd;

The first lexrep will only match “(I/i)n effect,” at the start of the sentence, the second lexrep will match “after all” at the very end of the sentence.

The second field can be used to specify metadata for the lexrep. That should be a list of key=value pairs, separated with ‘,’. A theoretical example could be “m=7,c=2,v=44”, with each pair defining a value for a different attribute. The use depends on the required functionality.

The planned certainty attributes can have a “certainty level”, ranging from “0” to “9”. That level is specified in the lexreps file, an example:

;c=6;almost certainly;;ENAdvmodal;ENCertainty;

The assigned metadata is visible in the trace info:

(c=6)Lexrep("almost certainly")=ENAdvmodal+ENCertainty(A:Entity|Certainty)

If the metadata field is empty, no level will be assigned, but the attribute will be generated.

# Pre-processor

The pre-processor file contains a list of rewritings again. These rewritings are executed before the normalization of the input. This means that the pattern detection is case sensitive.

Syntax: <ID>;<empty field>;<search pattern>;<substitution>;<empty field>;<comment>

Examples: 1;;“;";;replaces curly quotes by straight ones

14;;You'll;You 'll;;

35;;\it's\;it 's;;

Search pattern: Patterns cannot include word boundaries, but their scope can be smaller than one word. By default, the rewriting is executed whenever the pattern occurs. If a rewriting should only be applied at the beginning or at the end of a word, this can be indicated by a backslash before or after the pattern. Backslashes before and after a pattern mean that the pattern has to match exactly one word.

Substitution: The substitution can be empty, to delete the search pattern. It can also contain spaces, to split the search pattern into smaller words.

Note that – since no normalization is done yet – punctuation will be considered to be part of a word at this point. The pre-processor can be used to rewrite punctuation.

The order of the lines determines in which order the rewritings will be executed.

# Rules

Rules work on lexrep labels. They are used for

* disambiguation of grammatically ambiguous lexreps,
* to assign attributes (including scope) and
* to insert extra boundaries between successive Concepts or Relations.

Rules have an input **label pattern**, that will be matched against the lexrep labels in the sentence, and an output label pattern, that will define how the existing lexrep labels need to be modified.

The input label pattern can span several consecutive lexreps, up to 8. The minimum (obviously) is 1. The label pattern for each lexrep must be separated with the ‘|’ symbol.

If the rule input pattern matches, we say the rule fires, the existing labels need to be modified, based on the rule output pattern specification. The sentence is scanned from **left to right** to examine the input label pattern. If the rule fires, the lexrep labels are rewritten, and scanning continues on the next lexrep in the sentence (and NOT on the next lexrep after the rule input pattern span).

Rules are organized in **phases.** The phase number will decide which labels are taken into account for input pattern selection and rewriting. As a rule of thumb, only the labels that are defined on the rule’s phase can be used for pattern matching.

Rules are applied from the first phase to the last, and within the phase in the sequence of appearance in the rules.csv file.

A rule has an ID, but this is only for documenting purposes. The engine does not use this information. The ID is assigned automatically during compilation of the csv model.

A rule also has a comment field. That information appears in the trace output.

The maximum length of a rule (not including comments) is 256 characters.

Syntax: <ID>;<phase number>;<pattern of labels>;<substitution>;<empty field>;<comment>

Examples: 484;40;ENAdj|ENVerbCon;\*|ENCon;;

/\* 484. The speech was broadcasted on all American channels.

33;40;ENAuxCon;ENmAux;;

/\* 33. John will.

1148;60;ENPunctuation|ENWhich;ENPunctuation|+ENNegStop;;

/\* 1148. That didn't reassure markets, which were looking beyond a successful year.

Input pattern: Is a sequence of selection labels that can span several lexreps, then the labels for each consecutive lexrep are separated by vertical bars: |. See previous examples: ID:484 covers 2 lexreps, the second needs to have label ENCon, the previous will remain untouched (output operator ‘\*’ means ‘no operation’). The rules’ phase is 40, so both labels need to have 40 as a label phase. ID:33 only selects 1 lexrep having label ENAuxCon. The basic selection criterion is “having label”, but there are other cases:

**^** -> not this label, e.g. ^ENCon -> any label but ENCon

**+** -> AND-operator, e.g. ENAdj+ENInf, denotes that several labels have to be present on a lexrep (ENAdj and ENInf). Up to 8 labels can be combined on a lexrep.

**=** -> there is only one lexrep label: e.g. =ENNot -> the lexrep must have only the label ENNot. This is phase independent.

**~** -> there is only one lexrep label on the rule’s phase, e.g. ~ENDate -> the lexrep can have more labels, but on this phase, there is only one.

**:** -> OR-operator, e.g. ENCon:ENAdj -> one lexrep with the label ENCon or the label ENAdj (or both). The maximum number of labels in an OR-selector is 8.

+ and : can be combined, making the longest selector look like:

"LA:LB:LC:LD:LE:LF:LG:LH+LI:LJ:LK:LL:LM:LN:LO:LP+LQ:LR:LS:LT  
:LU:LV:LW:LX+LY:LZ:L1:L2:L3:L4:L5:L6+L7:L8:L9:L10:L11:L12:L13:L14+L15:L16:L17:L18:L19:L20:L21:L22+L23:L24:L25:L26:L27:L28:L29:L30+L31:L32:L33:L34:L35:L36:L37:L38"

^ can also be combined with + and :. Some examples:

^LA+^LB : not LA and not LB  
^LA:LB : not (LA or LB) = not LA and not LB. Both are logically equivalent.  
^LA+LC:LD+^LE:LF : not LA and (LC or LD) and not LE and not LF

typeConcept, typeRelation, typeBeginConcept, typeEndConcept, typeBeginEndConcept,

typeBeginRelation, typeEndRelation, typeBeginEndRelation, typeOther, typePathRelevant: these label type selectors can be used in rules to refer to “any label of typeX”.   
The type selectors can be combined with the AND-operator (+) and OR-operator (:). You can combine labels and label types with the AND-operator, but **not** with the OR-operator:  
LA+typeAmbiguous : valid  
LA:LB+typeRelation:typeEndRelation : valid   
LA:LB:typeAttribute+typeRelation:typeEndRelation : NOT VALID, labels and label types in OR-pattern.  
Negative selection can also be applied on type selectors:  
^typeConcept:typeBeginConcept:typeBeginEndConcept:typeEndConcept : no label of any of the types specified.

**NOTE:** Be aware that only labels that are defined in the rule’s phase are taken into   
 consideration!

**Literal labels**: any word can be used as a selecting label without defining it in the lexreps file, just by placing it between quotation marks in the input pattern, e.g. “only”.   
This must be used with caution. If the rules define a literal that is not part of the lexrep dictionary, then it is silently added. This can result in unexpected lexreps if the rule does not fire, and the literal remains in the lexrep list.  
To enable the rules engine to work on literal labels, behind the scene a new label is created, named "Lit\_" plus the literal itself (in this example, a new label "Lit\_only" will be created). This name will show up in the trace output. It is important to note that literal labels are of typeConcept. The label typeConcept in the rules input pattern will select all present literal labels.  
Since literal labels are treated by the iKnow Engine as normal labels, they can be combined with the OR-operator (:) and AND-operator (+), with the usual limitations.   
A literal is of typeConcept. If it remains in the resulting lexreps list, it is converted into a concept.

It is possible to select a variable number of lexreps with the same input label selection, using the **operator ‘\*’**. Without ‘\*’ the label pattern is fixed size, with the bars (‘|’) separating the consecutive lexreps: “LabelA|LabelB|LabelC” selects 3 consecutive lexreps, the first having label “LabelA”, the second “LabelB”, and the third “LabelC”. With the operator ‘\*’, a variable number of lexreps can be selected: “LabelA|\*LabelB|LabelC”. The first lexrep must be labeled “LabelA”, the following lexreps must be labeled “LabelB”, and the next lexrep must be labeled “LabelC”. That will make the rule fire. Variable means “at least one”, in a “LabelA|LabelB|LabelC” situation, the rule will fire, but not if LabelC directly follows LabelA.

The corresponding output pattern will be repeated for every input pattern selected lexrep, example: “\*ENCon|ENNot;+ENNegBegin|\*;;“. Consecutive lexreps labeled “ENCon” will be selected by this input pattern, if they precede a lexrep that is labeled “ENNot”. If the rule fires, all lexreps labeled ENCon will receive an additional label “+ENNegBegin”, the last lexrep labeled “ENNot” will remain untouched (output operator ‘\*’ means ‘no operation’).

Another variable lexrep selector **operator** is **‘.’**. It behaves the same as the previous ‘\*’, except that it also supports zero matches (operator ‘\*’ means “one or more” matches, operator ‘.’ means “zero or more” matches). Example: ENmAux|.ENArtPosspron|\*ENCon|ENInf;\*|\*|\*|+ENVerb. This rule fires whether there is an article in the second position of the pattern or not.

When adding **‘<’** to ‘\*’ or ‘.’ the lexrep selector operator will search for a pattern in a narrow way, so that \*LabelA doesn’t cover the maximum number of occurrences, but the minimal number.

Example:

lexreps:

xxx: LabelA

yyy: LabelB

zzz: LabelA + LabelB

input:

1) xxx yyy

2) xxx xxx yyy

3) xxx zzz yyy

4) xxx xxx zzz

rule input pattern maximal selector:

\*LabelA|LabelB

The pattern covers 1), 2) and 3). It does not cover 4): zzz also has LabelA and is covered by \*LabelA, so no following lexrep with LabelB is found.

rule input pattern minimal selector:

\*<LabelA|LabelB

The pattern covers 1), 2), and 4). The rule pattern does not cover the entire input pattern of 3). It will only affect xxx zzz, and not yyy, because zzz also has LabelB, which makes it the last lexrep covered by the rule pattern.

The variable rule selector ‘\*’ can also have parameters for a **minimum match**, and a **maximum match** specification: LabelA|\*{2,7}LabelB|LabelC. The condition for the rule to fire is that LabelB should be present over “at least” 2 consecutive lexreps, but no more than 7. If this condition is not met, the rule does not fire. A single parameter can also be used to express the exact matching lexreps: LabelA|\*{3}LabelB|LabelC. LabelB should be present over exactly 3 consecutive lexreps.

There is the possibility of optional extra **Key=Value parameters**, the first one specifies the length of the lexrep: if label selection is fine, an extra condition is examined, the lexrep should be exactly of the specified length. Optional parameters are listed between brackets, after label selection specification. An example: 887;1;CapitalAll(len=4);^ENCapitalAll;;

The lexrep must have the CapitalAll label, **and** it must be of length 4, otherwise the rule will not fire. The Key in this case is “len”, the Value is 4. The Value must be a single number between 1 and 9.

**SBegin/SEnd** labels: these “typeAttribute” labels behave like normal labels, with one exception: they cannot be removed. Every sentence receives an artificial “head” lexrep, labeled “SBegin”, and an artificial “tail” lexrep, labeled “SEnd”. These dummy lexreps can be used in rules processing, and are removed afterwards. The SBegin/SEnd labels are added automatically by the engine, but they should also be assigned when used in lexrep lookup (they are referred to as “B” for the head, and “E” for the tail). They can also be assigned by rules processing, but this functionality should, however, be used with great caution (since we assume SBegin to be the first lexrep in the sencence, and SEnd to be the last).

The operators ‘=’ (only one label on the matching lexrep), and ‘~’ (only one label on the rules’ phase of the matching lexrep) can be negated. The syntax is “^=LabelA”: if LabelA is \*not\* the only label on the matching lexrep, and “~=LabelA”: if LabelA is \*not\* the only label on the rules’ phase of the matching lexrep.

Substitution: The output pattern must have the same number of lexrep elements, separated by |-symbols, as the input pattern. By default, all (phase) labels on the selected lexrep are replaced by the labels in the output pattern. If a label is defined on several phases, and it is removed as a result of this label rewrite, it is no longer available on the other phases.

Note that in the substitution, there is no restriction with regard to phase definitions: labels from all phases can be used.

**Operators and functions**:

**^** -> replace the input selection label(s) (even Literal labels!) with the new label, but

keep all other labels

**+** -> add the new label to the existing lexrep labels, don’t delete any labels

**-**  -> remove this label, but keep all other labels

**\***  -> No Operation, keeping the existing lexrep labels

**“Join”** -> merge the lexrep with the previous one. All labels from all phases (from the “Join” lexrep) are removed, except the attribute labels from the rules’ phase and the last phase ($). If the previous lexrep has a literal label, it is removed:

Rule = LabelA|LabelB;LabelC|Join

Example: “lr1” (LabelA) “lr2” (LabelB) => “lr1 lr2” (LabelC)

If “lr1” has a literal label, it is removed, since it does not represent “lr1 lr2” anymore. typeAttribute labels on “lr2” remain.

**“JoinReverse”** -> merge the lexrep with the next one. All labels from all phases (from the “JoinReverse” lexrep) are removed, except the attribute labels from the rules’ phase and the last phase ($). If the next lexrep has a literal label, it is removed:

Rule = LabelA|LabelB;JoinReverse|LabelC

Example: “lr1” (LabelA) “lr2” (LabelB) => “lr1 lr2” (LabelC)

If “lr2” has a literal label, it is removed, since it does not represent “lr1 lr2” anymore. typeAttribute labels on “lr2” remain.

Both operators can be combined:

Rule = LabelA|LabelB|LabelC ;JoinReverse|LabelB|Join

Example: “lr1” (LabelA) “lr2” (LabelB) “lr3” (LabelC) => “lr1 lr2 lr3” (LabelB)

All literal labels (if any) will be removed, but attribute labels (on the rule’s phase and ‘$’ phase) will be added to the merged lexrep.

**“+Join” and “+JoinReverse”**  
To understand the difference between Join(Reverse) and +Join(Reverse), one must realize that a rule iterates from left to right, shifting one lexrep per iteration, even if the rule input pattern comprises several lexreps.

Suppose the rule says: “LabelA|LabelB;LabelC|**Join**”

We have sentence: “lr1” (LabelA) “lr2” (LabelA+LabelB) “lr3” (LabelB)

First iteration gives: “lr1” (LabelC) “lr2” (**Join**) “lr3” (LabelB) (start position is “lr1”, rule fires)

Next iteration (starting at “lr2”): rule does not fire, “Join” does not match “LabelA” selecting label.

Rule processing ends, followed by Join processing:

Result = “lr1 lr2” (LabelC) “lr3” (LabelB).

Now suppose the rule says: “LabelA|LabelB;LabelC**|+Join**”

We have sentence: “lr1” (LabelA) “lr2” (LabelA+LabelB) “lr3” (LabelB)

First iteration gives: “lr1” (LabelC) “lr2” (**LabelA+LabelB+Join**) “lr3” (LabelB) (start position is “lr1”, rule fires)

Next iteration (starting at “lr2”): “lr1” (LabelC) “lr2” (LabelC) “lr3” (LabelB+Join) (start position is “lr2”, rule fires)

Rule processing ends, followed by Join processing:

Result = “lr1” (LabelC) “lr2 lr3” (LabelC).

Comment: Can contain remarks or extra information about the rule.

**NOTE:** Be aware that only labels that are active in the rule’s phase are deleted or replaced (except for the Join operations). However, any label can be used in the substitution. In other words, it is possible to add a label, replace an input label by a label or even remove a label that is not active in the phase of the rule.

**NOTE:** Be careful with “+Join/+JoinReverse”. The Join operation only takes place after the rule has been completely processed. Since a rule iteratively scans the sentence, an unexpected pattern match can be the result.

Also note that if you do want to use “(+)Join/(+)JoinReverse”, it has to be the first/last and only label in the output pattern.

**Final Entity Type Selection**: At the end of the rules processing, the iKnow engine assign the output roles (Concept, Relation, PathRelevant, NonRelevant) to the lexreps. The final role is based on the types of the remaining labels in the last phase (99 or ‘$’). TypeAttribute labels are ignored for the role definition. Conflicting labels (e.g. a typeConcept label + a typeRelation label, or a typeConcept label + a typeBeginConcept label) result in a NonRelevant lexrep. A lexreps that ends up with only a literal label or only typeAttribute labels becomes (part of) a Concept.

**Final merge of entities:** After defining the roles, consecutive lexreps of the same type are merged into entities, translating “Begin” and “End” in the type definitions into entity boundaries.   
E.g. typeConcept | typeBeginConcept | typeConcept -> Concept1 | Concept2+3  
 typeConcept | typeEndConcept | typeConcept -> Concept1+2 | Concept3  
 typeConcept | typeBeginEndConcept | typeConcept -> Concept1 | Concept2 | Concept3

# Regular expressions

The regular expressions file (xx\_dev\_xx\_regex.csv) is another means to define lexreps. Only words that are not found in the lexreps file are checked against the regex file.

Basic syntax: <name>;<regular expression>

Example: number-big1;\d{1,3}[\,]\d\d\d

Name: The name must be placed in the lexreps file to assign a label to the expression. To distinguish regex names from default lexreps, the regex name must be written between curly brackets.   
Lexrep example: ;;{number-big1};4-6 digits with comma separator;ENNumber;

Regular expression: The regular expression must follow the ICU specification,

see: <http://userguide.icu-project.org/strings/regexp>

As mentioned earlier, first the engine will look up the real lexreps from the lexreps file. In a second phase, the unmatched lexreps will be the subject of regex matching.