UCxn Specification v1

Nathan Schneider, Archna Bhatia, Nina Böbel, William Croft, Santiago Herrera

This document summarizes the technical implementation of a project known as UCxn, which consists of construction annotations as a layer of treebank annotation atop Universal Dependencies (UD, specifically UDv2). The effort is detailed in Weissweiler et al. (LREC-COLING 2024), which examines 5 construction families across 10 languages.

For the selected constructions, morphosyntactic rules have been written and applied to treebanks in these 10 languages. Here we describe the data format and naming conventions in the resulting data. It is worth emphasizing that both the data format and linguistic categorizations of particular constructions are **provisional**. We anticipate that refinements and additions will be necessary as more languages and constructions are brought into the framework, and new user needs/workflows are encountered.

Overview

To illustrate briefly the sort of construction we annotate in the data, consider existentials:

> There is <u>no answer</u> [English]
> Il n' y a pas de <u>reponse</u> [French]

The English construction uses an expletive 'there' and a copula alongside the **Pivot**, the underlined phrase expressing a content element whose existence is being affirmed or denied. French uses two expletives as well as a form of the verb meaning 'have'.

UCxn annotates sentences like these by marking that a construction is present and also marking its *construction elements*, the components that are realized by individual words/phrases. The annotated construction elements have semantic names, e.g. **Pivot**, which can be qualified with the construction name: e.g. **Existential.Pivot**. Here we are assuming for purposes of illustration a simple construction name, "Existential", which is defined based on meaning in a way that generalizes across languages. Construction name subtyping is also possible, with hyphen-separated parts of increasing specificity. Some of the parts can reflect *strategies* that differentiate how languages realize the meaning morphosyntactically. Below, we expand on the technicalities of the approach:

- Format Specification - how constructions are encoded in .conllu files
- Current Policy on Construction Elements
- Inventory of Nomenclature - names used in our data for the specific constructions and subtypes in our pilot study

Finally, a word about tokens vs. types: In principle, a *Constructicon* (lexicon-of-constructions) would define the meaning-bearing grammatical types of the language. UCxn takes inspiration from Constructicon resources that have been developed for several languages. However, UCxn is merely a standard for annotating instances. That is, UCxn itself does not propose a formal notation for construction type definitions; it merely assumes the list of construction types, their elements, and associated type-level properties (such as the range of kinds of phrases that can fill a given element, optionality of elements, agreement constraints, etc.) are known to the annotator. In our rule implementation, Grew patterns are used to formalize the morphosyntactic elements that are essential for identifying instances of a construction (but there may be additional aspects of constructional form that would be included in a full construction definition).

Format Specification

The UCxn layer is currently implemented in the CoNLL-U format—the standard for UD treebanks—within the final (MISC) column, which supports arbitrary key-value pairs.[1] For each construction instance, or *construct*, UCxn supports annotation in two fields:

- the **Cxn** field for the name of the construction and specification of its elements—once per construct
- the **CxnElt** field for each construct element to be listed separately, anchored on a node representing that element.

An example sentence illustrating resultative and interrogative constructions (with simplified names), showing annotation lines for all words/UD nodes (underscore = “no UCxn annotation on this node”):

```
        Who let the dogs out ?
        1       Who     …       2       nsubj   …       CxnElt=2:Interrogative.WHWord
        2       let     …       0       root    …
Cxn=Interrogative,Resultative|CxnElt=2:Interrogative.Clause,2:Resultative.Event
        3       the     …       4       det     …       _
        4       dogs    …       2       obj     …       _
        5       out     …       2       xcomp   …       CxnElt=2:Resultative.ResultState
        6       ?       …       2       punct   …       _
```

The details of the format are described below.

Construction name

A construction (or construction element) name is a string that must begin and end with alphanumeric characters. Internal characters must be alphanumerics, hyphens, and/or underscores.

[1] A standard for stand-off annotation of new layers by introducing separate files has been proposed but is not currently recognized by the central UD infrastructure. Moving UCxn to such a format would be something to explore in the future.

In principle, namespaces of constructions might be specified in the .conllu file, e.g. in a document metadata comment. This would enable formal linking to Constructicon resources such that a file could refer to constructions from different sources, and validation, for example to make sure that construction names are spelled correctly and contain valid construction element names. However, this is not part of the present spec.

Locus of annotation

As a matter of convention, we need to establish where in the UD tree—i.e., on which node—a construct is anchored. The name of the construction will be listed on this node in the **Cxn** field.

The simplest convention for determining the anchor is to use the lowest node in the tree which (i) is part of the construct, and (ii) dominates any other word that is in some part of the construct—content as well as functional elements.[2] There is no requirement that the anchor dominate *only* words that belong to some part of the construct—in particular, additional modifiers or conjuncts (if a first element of a coordination) may be present as dependents.[3]

If multiple constructs share the same anchor, they are sorted alphabetically by construction name and separated by commas. This will be quite common: for one thing, some of the constructs may represent modification constructions (of which there are an unbounded number); and moreover, constructions may overlap—e.g., a conditional and an existential ("If there are any good ideas…"), or an interrogative and a resultative ("Who let the dogs out?").

Units

As UD does not provide a theory of phrase structure, and all dependents of a construct anchor are not necessarily part of the construct, we cannot necessarily rely on full subtrees of a given node to express units of a construct. E.g., when annotating an existential construction:

According$_1$ to the teacher , there$_6$ is a$_8$ right answer$_{10}$ and a wrong answer$_{14}$, but I am not sure which is which .

It would be reasonable to consider just the highlighted portion to be the span of the existential construct as a whole, and the underlined portion as the span of the Pivot element. However, "answer$_{10}$" is the root of the full sentence, so simply anchoring the Existential construct (and its Pivot element) on node 10 is not very specific.

Token spans (or multispans for discontinuous expressions) would offer the most flexibility to express units. However, toolchains (such as the Grew system in which we implemented rules) may focus on graph relations rather than token spans, and there may be substantial ambiguity

[2] The functional elements are ordinarily dependents of the content elements anyway given UD guidelines.
[3] There is theoretically a possibility that no node meeting criteria (i) and (ii) exists, i.e., the construct is disjointed in the tree. We have not encountered this situation and defer a decision on how to handle it until examples can be collected.

resolution necessary to determine the precise span (e.g., including the nominal conjunct “and a wrong answer” but excluding the clausal conjunct “but I am not sure which is which”).

For maximum flexibility, at the format level, 4 kinds of units are available:

- A **span** (or multispan), which is specified in terms of UD word indices, and must contain a hyphen. (“13-13” is an example of a single-word span.)
- A **node**, without specifying what (if any) dependents may be involved in the unit.
- A **full subtree** (in the basic UD tree), which we notate as a node index followed by “f”, e.g. “13f”
- A **partial subtree**, which we notate as a node index followed by “p”, e.g. “13p”. This stands in contrast with a full subtree, indicating that not all dependents belong in the unit, but does not specify precisely which.

In the existential example, the rule for identifying the construction might specify “10p” as the Pivot element, whereas a more precise annotation might specify “8-14”.

Multiple constructs of the same construction and anchor

It may be necessary to disambiguate these: e.g., in a clause with multiple protasis modifiers, there could be multiple Conditional labels on the same anchor:

If you try$_3$ to sound like an opera singer , it 'll just **come**$_{14}$ to you if you have$_{19}$ the ear

These require disambiguation, which is indicated with an index after “#”, where the order follows the order of the construct elements in the sentence (in this case, the Protasis elements):

```
14	come	Cxn=Conditional#1,Conditional#2
```

Comparing construct element anchor tuples (3,14) and (14,19), 3 < 14, so the “if you try” construct is annotated first. If the full construction name only occurs once on an anchor, “#” indexations are not provided.

Construct elements

Elements of a construct are anchored on a node and linked via the **CxnElt** to the construct anchor. The unit information for the element, if specified, goes at the end following an “@” sign.

In the above example, this would look like

```
3	try	CxnElt=14:Conditional#1.Protasis@f

14	come	CxnElt=14:Conditional#1.Apodosis@p,14:Conditional#2.Apodosis@p

19	have	CxnElt=14:Conditional#2.Protasis@f
```

Rejected ideas

Redundant specification of construction elements

In addition to marking a node within the element itself with CxnElt, we considered offering a “bird’s-eye view” of the construct by listing its elements on the construct anchor:

```
Who let the dogs out ?
1    Who    …    2    nsubj  …    CxnElt=2:Interrogative.WHWord
2    let    …    0    root   …    Cxn=Interrogative(1:WH,2:Clause);Resultative(2:Event,5:ResultState)|CxnElt=2:Interrogative.Clause;2:Resultative.Event
3    the    …    4    det    …    _
4    dogs   …    2    obj    …    _
5    out    …    2    xcomp  …    CxnElt=2:Resultative.ResultState
6    ?      …    2    punct  …    _
```

It ’ll just come$_4$ to you if you have$_9$ the ear .

```
4    come   …    0    root   …    Cxn=Conditional(_p:Apodosis,9f:Protasis)|CxnElt=_p:Conditional.Apodosis

9    have   …    4    advcl  …    CxnElt=4f:Conditional.Protasis
```

(In the second example, “_” is shorthand for “this node”, i.e. “4”.) The highlighted portions are redundant with the CxnElt information, and a script to produce one from the other. The thinking was that having the information in both places would make it easier to perform regex searches for things like order of elements of a construct, in the event that tools don’t process CxnElt information as graph edges. But we are optimistic that Grew can be extended with such processing along the lines of PARSEME annotations, and thus favor the less verbose solution.

Assigning a unique index or variable to every construct in a tree

Instead of identifying constructs by anchor node + construction name (+ index if further disambiguation is needed), an alternative would be to assign a unique variable or index to every construct. This would remove the need for notation specific to a (rare) special case where multiple instances of the same construction share the same anchor. But it would make the common case more verbose/complex to read, with a new numbering scheme or variable system distinct from the UD node numbers. And with a numbering scheme not based on UD node IDs, adding new annotations in a sentence could mean renumbering old ones, which gets tedious.

More elaborate notation for construction names

Beyond the hyphenation of construction names, it was discussed whether to use additional special characters to indicate the construction/strategy distinction and hierarchical vs. orthogonal categorizations. We have elected to keep the notation as simple as possible for now, and especially to avoid characters like `.` `,` `:` `=` `|` `@` # which have other meanings in the format, but additional special characters might be added in the future.

Discussion: Format Considerations

- Kinds of searches we contemplated vis-a-vis CxnElts:
    - Checking the morphosyntax (deprel, morphology, functional dependents) associated with a construction element name—can be done shallowly with a regex: `"CxnElt=[^=]*Conditional\.Protasis"`
    - Checking the relative order of a pair of elements—to do this precisely (in case a sentence has multiple constructs of the same construction), it is necessary for the query tool to extract the graph.
- Annotation interface (for manual annotation)
    - With pre/postprocessing, it may be possible to use FrameNet-style annotation tools. These typically are over spans, so the postprocessing would need to identify the head within each span.

Customization of query language for specifying rules

The Grew system for expressing and applying patterns required a slight extension for this project. Grew itself is not expressive enough to support construct-append operations (where multiple constructs occur on the same anchor, and multiple constructs of the same construction require disambiguation) or linking of elements to their construct anchors. We therefore designate a special variable name for anchors and apply postprocessing to populate the `Cxn` and `CxnElt` fields. Here is a Swedish example:

```
rule r3a { % Conditionals with 'om'
  pattern {
    _anchor_ -["advcl"|"advmod"]-> P;
    P-[mark]->M;
    M[lemma="om"|"ifall"];
  }
  commands {
    _anchor_.Cxn="Conditional-Marker";
    P.CxnElt="Protasis";
    _anchor_.CxnElt="Apodosis";
  }
}
```

In our rules, the special name `_anchor_` is used for the variable matching the construct anchor node. In the `commands` block, the `_anchor_` is always assigned a construction name in its `Cxn` property, and then nodes may be identified as elements (understood to belong to the same construct). The full string values for the MISC column (with node indices and comma separators and so forth) are constructed under the hood. Note that a rule can match multiple times per sentence and a node may serve as the anchor for multiple constructs of the same or different constructions. However, only one construct can be annotated per application of a rule.

Current Policy on Construction Elements

A construction may contain two kinds of construction elements: **content elements** (the constituent phrases carrying semantic weight) and **functional elements** (grammatical markers associated with the strategy that offer little independent meaning[4]). Reflecting our meaning-based definitions of constructions as well as UD's principle of favoring relations between content words, **only content elements are explicitly marked in our current implementation**. Thus, for example, the word "if" would not be individually marked as an element in an if-conditional (though its parent, the predicate serving as protasis, would). Neither would the tense of the predicate be marked as an element of the construction, even if it is part of the rule for detecting that construction. Finally, at present, **we only mark *overt* content elements**, never empty/omitted/implicit elements. Functional or empty construction elements may, however, be added in a future version.

Inventory of Nomenclature

The full construction names identified in the pilot rules/data are listed below. These were determined post hoc, after first writing language-specific rules to cover as many variants of the construction family as possible, and then assigning subcategories to those rules. While we aim for naming consistency across languages, the way language-specific rules have carved up a construction into strategies can differ, and the strategies in our data for a given construction do not reflect the full range of strategies attested typologically. Future work should develop a more systematic taxonomy.

The nomenclature we use in the data—particularly for form/strategy subtypes—should be regarded as convenient to help disambiguate which rule (or group of closely related rules) produced which annotation. As such, the names reflect *intra*linguistic variety in the morphosyntactic patterns that fall under a construction family. For the full morphosyntactic details it is necessary to consult the rule itself.

[4] In general, these are words bearing a dependency relation listed in the "function words" column of this page—aux, cop, mark, case, clf, det—plus cc and expl. But there are exceptions, e.g. adpositions in NPN constructions are considered content elements as they contribute meaning.

Each name begins with the construction family name from the paper (“Conditional”, “Existential”, “Interrogative”, “Resultative”, or “NPN”; note that NPN is a form-based category as described in the paper, whereas the others are meaning-based constructions). Additional subtypes are provided in hyphenated parts, with more general/cross-linguistic/meaning-based ones preceding more specific/form-based ones.

At this point we do not impose a rigorous structure on construction names in the data. A mapping to more elaborate names (based on *Morphosyntax*; Croft 2022) clarifies which parts of the name correspond to different kinds of concepts, like construction vs. semantics vs. strategy. In future efforts with more constructions, it may become necessary to disambiguate terms in the data itself, e.g. “LocativeCxn” vs. “LocativeStr”.

Cxn Family: Conditional	**Content CxnElts:** Protasis, Apodosis	
Full Name in Data	*Languages*	*Details*
`Conditional-Interrogative`	hi, en	“What if” (Apodosis=“what”, Protasis=“if…”)
`Conditional-NegativeEpistemic`	cop, es, fr, he, hi, pt, zh	Negative epistemic stance a.k.a. *counterfactual* (“If you had arrived on time, we would have finished by now”: conditional event stipulated to have not occurred)
`Conditional-NegativeEpistemic-NoInversion`	en	Subject is present (nonreduced) and precedes the verb
`Conditional-NegativeEpistemic-Reduced`	en	
`Conditional-NegativeEpistemic-SubjVerbInversion`	en	
`Conditional-NeutralEpistemic`	cop, es, fr, he, hi, pt, zh	Neutral epistemic stance (“If you touch it, it will explode": conditional event may or may not occur)
`Conditional-UnspecifiedEpistemic`	pt	Epistemic stance cannot be determined by the query alone
`Conditional-UnspecifiedEpistemic-NoInversion`	en	
`Conditional-UnspecifiedEpistemic-Reduced`	en	
`Conditional-UnspecifiedEpistemic-SubjVerbInversion`	en	

`Conditional-Marker`	sv	
`Conditional-Marker-Complex`	de	
`Conditional-Marker-Simple`	de	
`Conditional-Marker-Subjunctive`	hi	
`Conditional-Reduced`	fr, de	*Reduced* means no subject, e.g. "if possible": Conditional_cxn-Deranked_str
`Conditional-SubjVerbInversion`	sv, de	

Remarks: For conditionals we were able to write rules for a range of strategies, some of which signal epistemic stance (of *negative* = counterfactual or *neutral* = possible; *positive* epistemic stance, indicating that something did or will happen, is incompatible with conditionality). For English, queries identify certain negative epistemic conditionals, but other conditionals are not so easy to label as negative vs. neutral based on their morphosyntax, and so are listed as *unspecified* for epistemic stance. The 7 full names used in English follow the regex

```
Conditional-(Interrogative|
   ((Negative|Unspecified)Epistemic-(Reduced|NoInversion|SubjVerbInversion))).
```

The Swedish and German queries do not attempt to identify epistemic stance, but are assigned subtypes based on formal criteria.

Cxn Family+Subfamily: Interrogative-Alternative	**Content CxnElts:** Clause, Choice1, Choice2, …	
Full Name in Data	*Languages*	*Details*
`Interrogative-Alternative`	es, fr, pt	AlternativeQuestion_cxn

Cxn Family+Subfamily: Interrogative-WHInfo[5]	**Content CxnElts:** Clause, WHWord[6]	
Full Name in Data	*Languages*	*Details*
`Interrogative-WHInfo`	cop	

[5] The term "WHInfo" acknowledges both the term "WH question" which is commonplace in computational linguistics, and the term "information question" which is one of the accepted terms in typology (along with "content question" and some others).

[6] The term "WHWord" makes clear that a word/lexical item is being targeted, not necessarily a phrase. Other terms in use for this concept might cause confusion: "question word" might be taken to include polarity markers, and "interrogative pronoun" might be taken to exclude words analyzed as determiners in UD.

`Interrogative-WHInfo-Direct`	de, en, es, fr, hi, he, pt, sv, zh	InformationQuestion_cxn
`Interrogative-WHInfo-Indirect`	de, en, es, fr, hi, he, pt, sv	InterrogativeComplement_cxn-InformationQuestion_inf

Cxn Family+Subfamily: Interrogative-Polar	**Content CxnElts:** Clause	
Full Name in Data	*Languages*	*Details*
`Interrogative-Polar`	cop	
`Interrogative-Polar-Direct`	de, en, es, fr, hi, he, pt, sv, zh	PolarityQuestion_cxn
`Interrogative-Polar-Indirect`	de, en, es, fr, he, hi, pt, sv	InterrogativeComplement_cxn-PolarityQuestion_inf

Cxn Family+Subfamily: Interrogative-Reduced	**Content CxnElts:** Clause	
Full Name in Data	*Languages*	*Details*
`Interrogative-Reduced`	zh	Contains a question mark but doesn’t match other queries; context-dependent interpretation

Cxn Family: Existential[7]	**Content CxnElts:** Pivot, (sometimes) Coda[8]	
Full Name in Data	*Languages*	*Details*
`Existential-CopPred`	hi, he	Hebrew past & future
`Existential-CopPred-HereExpl`	en	Technically this “be” is tagged as a VERB in English, but we can think of it as recruited from the copula “be”.
`Existential-CopPred-ThereExpl`	de, en	

[7] Note that the scope of what we call “existential” constructions includes instances that may better be described as presentational.

[8] “Coda” is the term used in the paper (Weissweiler et al., LREC-COLING 2024) for the location if specified alongside the pivot. It is only annotated if explicit, and even then, only by some languages/rules at present.

`Existential-ExistPred`	cop, sv, pt	
`Existential-ExistPred-FullVerb`	he	קיים
`Existential-ExistPred-NoExpl`	en	a path to victory exists” (in contrast with“ Existential-ExistPred-ThereExpl “There exists a path to victory”; cf. (Existential-ExistPred-FullVerb in Hebrew
`Existential-ExistPred-ThereExpl`	en	“There exist” (unattested in EN data; but would be a case of overlap between current ThereExpl and ExistPred rules)
`Existential-ExistPred-VblPart`	he	Verb-like particle (tagged as VERB in UD יש :(but not a full verb
`Existential-GivePred-ItExpl`	de	
`Existential-HavePred`	es, pt, zh	
`Existential-HavePred-ItExpl-ThereExpl`	fr	“Il y a”
`Existential-MannerPred-ThereExpl`	en	“There stretched…new vistas of trees and paths…”
`Existential-NotExistPred`	cop	
`Existential-NotExistPred-VblPart`	he	אין

Cxn Family: Resultative	**Content CxnElts:** Event, ResultState	
Full Name in Data	*Languages*	*Details*
`Resultative`	zh	

Cxn Family: NPN	**Content CxnElts:** N1, N2, P	
Full Name in Data	*Languages*	*Details*
`NPN`	cop, de, en, es, fr, (hi), he, pt, sv,	

Remarks: Hindi defines a rule for NPN, but no instances are attested in the corpus.

Treebanks are welcome to annotate constructions beyond the families discussed here, and to provide separate documentation.

Example Rules

Below are examples of rules, selected to illustrate a range of construction families. The full suite of rules is hosted in the repository at https://github.com/LeonieWeissweiler/UCxn.

- A Conditional rule in Swedish is discussed above. One in Spanish:

```
rule Conditional-NegativeEpistemic {
  pattern {
    _anchor_-[advcl]->P;
    P->A;
    A[Mood=Sub, Tense=Imp, lemma=haber];
    P-[mark]->S1;
    S1[upos=SCONJ,lemma=si]
  }
  without {
    P-[mark]->S2;
    S2[upos=SCONJ,lemma=como]
  }
  commands {
    _anchor_.Cxn = "Conditional-NegativeEpistemic";
    _anchor_.CxnElt = "Apodosis@p";
    P.CxnElt = "Protasis@f"
  }
}
```

- Existential rules in Coptic and French, respectively:

```
rule ExistentialNegative {
  pattern {
    _anchor_ [lemma="ⲙⲛ",xpos=EXIST];
    _anchor_ -[nsubj]-> SBJ;
  }
  commands {
     _anchor_.Cxn="Existential-NotExistPred";
     SBJ.CxnElt="Pivot";
  }
}
```

```
rule Existential-HavePred-ItExpl-ThereExpl {
  pattern {
    _anchor_ [lemma="avoir"]; N1[lemma="y"];
    _anchor_-[expl:comp]->N1;
    _anchor_ -[obj]->N2;
    N2[upos=NOUN]
  }
  commands {
    _anchor_.Cxn = "Existential-HavePred-ItExpl-ThereExpl";
    N2.CxnElt = "Pivot@f"
  }
}
```

- Interrogative rules in Hebrew, Hindi, and Portuguese, respectively:

```
rule r1a { %  Main clause polar with particle
  pattern {
     PRT [lemma = "האם"|"שמא"];
     _anchor_-[mark:q]->PRT ;
     Q [form="?"];
  }
  commands {
     _anchor_.Cxn="Interrogative-Polar-Direct";
     _anchor_.CxnElt="Clause";
  }
}

rule int-direct-info { % Interrogatives - Direct information questions
  pattern {
    W [lemma="क्या"|"कौन"|"कहाँ"|"कहां"|"कब"|"कैसे"|"कितना"|"किस"];
    _anchor_ -[^root]-> W;
  }
  without { SC [form="कि"]; _anchor_ -[mark]-> SC }
  without { V1 [upos=VERB]; V1 -[advcl]-> _anchor_; }
  commands {
    _anchor_.Cxn="Interrogative-WHInfo-Direct";
    _anchor_.CxnElt="Clause";
    W.CxnElt="WHWord"
  }
}
```

```
rule InterrogativeAlternativeRule1 {
  pattern {
    OR [lemma="ou"];
    IN [lemma="?"];
    _anchor_-[^root]->CH1;
    CH1-[conj]->CH2;
    CH2-[cc]->OR;
    CH1->IN;
  }
  without { CH1-[conj]->CH3; }
  commands {
    _anchor_.Cxn = "Interrogative-Alternative";
    _anchor_.CxnElt = "Clause";
    CH1.CxnElt = "Choice1";
    CH2.CxnElt = "Choice2";
  }
}
```

- An NPN rule in German:

```
rule NPN {
  pattern {
    _anchor_[upos=NOUN];
    N2[upos=NOUN];
    _anchor_.form = N2.form;
    P[upos=ADP];
    _anchor_ < P;
    P < N2
  }
  without {
    A[upos=ADP];
    A < _anchor_
  }
  commands {
    _anchor_.Cxn="NPN";
    _anchor_.CxnElt="N1";
    P.CxnElt="P";
    N2.CxnElt="N2"
  }
}
```

- A Resultative rule in Chinese:

```
rule r1 {
  pattern {
      _anchor_ -[compound:vv]-> RES;
  }
  commands {
    _anchor_.Cxn=Resultative;
    _anchor_.CxnElt=Event;
    RES.CxnElt=ResultState;
  }
}
```