世界のUniversal Dependenciesと係り受け解析ツール群

AUTHOR(S):

安岡, 孝一

CITATION:

安岡, 孝一. 世界のUniversal Dependenciesと係り受け解析ツール群. 第3回Universal Dependencies公開研究会 2021: 1-20

ISSUE DATE:

2021-06-22

URL:

http://hdl.handle.net/2433/265505

世界の Universal Dependencies と係り受け解析ツール群

安岡孝一*

1　はじめに

筆者が班長を務める京都大学人文科学研究所共同研究班「古典中国語のコーパスの研究」(班員: Christian Wittern、守岡知彦、池田巧、山崎直樹、二階堂善弘、鈴木慎吾、師茂樹、白須裕之、藤田一乗) では、古典中国語 (漢文) の文法解析に精力を傾注しており、その道具立ての一つとして、Universal Dependencies[1](以下「UD」) の古典中国語への適用[2]を研究してきた。依存文法解析それ自体は、Tesnière の構造的統語論[3]に源を発し、Мельчук の有向グラフ記述[4]によって、一応の完成を見た手法である。その最大の特長は、いわゆる動詞中心主義によって言語横断的な記述が可能だという点にあり、Мельчук 依存文法をコンピュータ向けに洗練した UD においても、言語に関わらない記述、という特長が前面に押し出されている。UD における文法構造記述は、句構造を考慮せず、全てを単語間のリンクとして表現する。これは、Мельчук の有向グラフ記述が、単語間のリンクという形態を取っていたからであり、そういう割り切りの結果として、言語横断的な文法構造記述を可能としているのである。

では UD は、本当に言語横断的なのだろうか。UD 可視化ツール deplacy[5]を、古典中国語 UD のみならず、他の言語 UD に適用するにあたって、筆者らが抱いた疑問がこれだった。60 以上の言語の UD に対し、様々な係り受け解析ツールを調査 (表 1) していくうちに、筆者らの疑問は確信に変わっていった。UD は、記述上は言語横断的であるものの、その内実は違う。言語ごとに異なる「クセ」があって、それが係り受け解析ツールの得手不得手となって現れている。本稿では、これら各言語 UD の「クセ」と、係り受け解析ツールの「クセ」を、ざっと見ていくことにしよう。

2　Universal Dependencies と CoNLL-U の概要

UD は、書写言語における品詞・形態素属性・依存構造 (係り受け関係) を、言語に関わらず記述する手法である。句構造を考慮せずに係り受け関係を記述することで、言語横断性を高めており、全ての文法構造を単語間のリンクで記述するのが特徴である。

*京都大学人文科学研究所附属東アジア人文情報学研究センター

[1]Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Jan Hajič, Christopher D. Manning, Sampo Pyysalo, Sebastian Schuster, Francis Tyers, Daniel Zeman: Universal Dependencies v2: An Evergrowing Multilingual Treebank Collection, Proceedings of the 12th Language Resources and Evaluation Conference (May 2020), pp.4034-4043.

[2]Koichi Yasuoka: Universal Dependencies Treebank of the Four Books in Classical Chinese, DADH2019: 10th International Conference of Digital Archives and Digital Humanities (December 2019), pp.20-28.

[3]Lucien Tesnière: Éléments de Syntaxe Structurale, Paris: C. Klincksieck (1959).

[4]Igor A. Mel'čuk: Dependency Syntax: Theory and Practice, New York: State University of New York Press (1988).

[5]安岡孝一: Universal Dependencies にもとづく多言語係り受け可視化ツール deplacy, 人文科学とコンピュータシンポジウム「じんもんこん 2020」論文集 (2020 年 12 月), pp.95-100.

表 1: deplacy に接続可能な係り受け解析ツール (2021 年 6 月 20 日現在)

アフリカーンス語	Camphr-Udify, spaCy-COMBO, UDPipe 2, Trankit, spaCy-jPTDP, Turku-neural-parser-pipeline など
アラビア語	Trankit, UDPipe 2, Turku-neural-parser-pipeline, spacy-udpipe, NLP-Cube, spaCy-COMBO など
ベラルーシ語	Camphr-Udify, Stanza, Trankit, UDPipe 2, spacy-udpipe
ブルガリア語	Trankit, Camphr-Udify, UDPipe 2, CLASSLA, NLP-Cube, Stanza, spaCy-COMBO, spaCy-jPTDP など
カタルーニャ語	Camphr-Udify, Stanza, NLP-Cube, spacy-udpipe, COMBO-pytorch, Trankit, UDPipe 2 など
コプト語	Stanza, spaCy-Coptic, spacy-udpipe, UDPipe 2
チェコ語	Camphr-Udify, spaCy-jPTDP, Trankit, UDPipe 2, spacy-udpipe, NLP-Cube, Stanza など
ウェールズ語	UDPipe 2, luigi/custom_models, Camphr-Udify, Stanza
デンマーク語	Camphr-Udify, UDPipe 2, Stanza, Trankit, Turku-neural-parser-pipeline, COMBO-pytorch など
ドイツ語	Camphr-Udify, Stanza, COMBO-pytorch, UDPipe 2, spaCy-jPTDP, NLP-Cube, spacy-udpipe など
ギリシア語	Stanza, UDPipe 2, Trankit, spacy-udpipe, Turku-neural-parser-pipeline, spaCy-jPTDP など
英語	Camphr-Udify, Stanza, COMBO-pytorch, UDPipe 2, NLP-Cube, spaCy-COMBO, Trankit など
スペイン語	spaCy, Stanza, UDPipe 2, NLP-Cube, spacy-udpipe, Trankit, Camphr-Udify, spaCy-COMBO など
エストニア語	Stanza, spacy-udpipe, EstMalt, Turku-neural-parser-pipeline, COMBO-pytorch, spaCy-jPTDP など
バスク語	spaCy-ixaKat, COMBO-pytorch, Trankit, Turku-neural-parser-pipeline, Stanza, spacy-udpipe など
ペルシア語	Stanza, spaCy-COMBO, Turku-neural-parser-pipeline, spaCy-jPTDP, spacy-udpipe, Trankit など
スオミ語	Stanza, UDPipe 2, spacy-udpipe, NLP-Cube, spacy-fi, Turku-neural-parser-pipeline など
フェロー語	Stanza, Turku-neural-parser-pipeline
フランス語	spaCy, Stanza, NLP-Cube, UDPipe 2, Trankit, spacy-udpipe, Camphr-Udify, spaCy-COMBO など
ゲール語 (アイルランド)	Stanza, UDPipe 2, COMBO-pytorch, Trankit, spacy-udpipe, spaCy-COMBO, spaCy-jPTDP など
ゲール語 (スコットランド)	GLA, Stanza, UDPipe 2, spacy-udpipe, Trankit
ガリシア語	Camphr-Udify, Stanza, spaCy-jPTDP, Turku-neural-parser-pipeline, NLP-Cube, spacy-udpipe など
古典ギリシア語	Camphr-Udify, UDPipe 2, spaCy-COMBO, spaCy-jPTDP, Stanza, Trankit, spacy-udpipe など
ヘブライ語	Trankit, HebPipe, Stanza, spaCy-COMBO, spaCy-jPTDP, Turku-neural-parser-pipeline, UDPipe 2 など
ヒンディー語	NLP-Cube, spaCy-COMBO, UDPipe 2, Trankit, Turku-neural-parser-pipeline, Stanza など
クロアチア語	NLP-Cube, UDPipe 2, Turku-neural-parser-pipeline, COMBO-pytorch, Trankit, Camphr-Udify など
ハンガリー語	Trankit, UDPipe 2, COMBO-pytorch, Camphr-Udify, NLP-Cube, Turku-neural-parser-pipeline など
アルメニア語	Stanza, Camphr-Udify, Trankit, UDPipe 2, spacy-udpipe, spaCy-COMBO, YerevaNN など
インドネシア語	Stanza, Trankit, Turku-neural-parser-pipeline, NLP-Cube, spacy-udpipe, Camphr-Udify, Malaya など
アイスランド語	COMBO-pytorch, Stanza
イタリア語	NLP-Cube, COMBO-pytorch, spaCy-COMBO, Trankit, Camphr-Udify, Stanza, spaCy など
日本語	spaCy-SynCha, spaCy-ChaPAS, UniDic-COMBO, spaCy, NLP-Cube, GiNZA, Camphr-KNP など
カザフ語	Camphr-Udify, NLP-Cube, spaCy-COMBO, Trankit, Turku-neural-parser-pipeline など
韓国語	Camphr-Udify, Stanza, UDPipe 2, Trankit, spaCy-jPTDP, Turku-neural-parser-pipeline など
ラテン語	spaCy-COMBO, Trankit, Stanza, UDPipe 2, spacy-udpipe, Camphr-Udify, NLP-Cube など
リトアニア語	spaCy, Trankit, Stanza, Camphr-Udify, UDPipe 2, spacy-udpipe
ラトビア語	Camphr-Udify, NLP-Cube, Trankit, UDPipe 2, Stanza, spaCy-COMBO, spacy-udpipe など
古典中国語	SuPar-Kanbun, UD-Kanbun, GuwenCOMBO, UD-Chinese, Trankit, spacy-udpipe, Stanza, UDPipe 2
マケドニア語	Camphr-Udify, spaCy
マルタ語	Stanza, UDPipe 2, spacy-udpipe, COMBO-pytorch, Camphr-Udify
ノルウェー語 (ブークモール)	Stanza, COMBO-pytorch, spaCy-COMBO, Camphr-Udify, UDPipe 2, Turku-neural-parser-pipeline など
ノルウェー語 (ニーノルスク)	Stanza, spacy-udpipe, spaCy-COMBO, spaCy-jPTDP, UDPipe 2, Turku-neural-parser-pipeline など
オランダ語	spaCy-Alpino, Camphr-Udify, Stanza, UDPipe 2, Trankit, spacy-udpipe, spaCy, spaCy-COMBO など
ポーランド語	spaCy, spaCyPL, Camphr-Udify, Stanza, UDPipe 2, spacy-udpipe, Trankit, COMBO-pytorch など
ポルトガル語	spaCy, Camphr-Udify, Stanza, COMBO-pytorch, NLP-Cube, spaCy-COMBO, spaCy-jPTDP など
ルーマニア語	UDPipe 2, COMBO-pytorch, Trankit, Camphr-Udify, spaCy, Stanza, NLP-Cube, spaCy-COMBO など
ロシア語	Stanza, Trankit, Camphr-Udify, UDPipe 2, Truku-neural-parser-pipeline, spaCy, COMBO-pytorch など
スロバキア語	Camphr-Udify, UDPipe 2, Trankit, spacy-udpipe, NLP-Cube, spaCy-COMBO, Stanza など
スロベニア語	CLASSLA, Turku-neural-parser-pipeline, NLP-Cube, UDPipe 2, COMBO-pytorch, spacy-udpipe など
セルビア語 (キリル)	Camphr-Udify
セルビア語 (ラテン)	CLASSLA, UDPipe 2, Trankit, Camphr-Udify, Turku-neural-parser-pipeline, NLP-Cube など
スウェーデン語	Camphr-Udify, COMBO-pytorch, Trankit, NLP-Cube, Stanza, spaCy-COMBO, UDPipe 2 など
タミル語	Camphr-Udify, Trankit, UDPipe 2, spacy-udpipe, Stanza
タイ語	spaCy-Thai, Turku-neural-parser-pipeline
タガログ語	Camphr-Udify
トルコ語	Stanza, NLP-Cube, spaCy-COMBO, spaCy-jPTDP, Camphr-Udify, UDPipe 2, spacy-udpipe など
ウクライナ語	Stanza, Camphr-Udify, spaCy-COMBO, UDPipe 2, Trankit, Turku-neural-parser-pipeline など
ベトナム語	Trankit, Stanza, Turku-neural-parser-pipeline, spaCy-jPTDP, spaCy-COMBO, NLP-Cube など
ウォロフ語	UDPipe 2, Stanza
中国語 (簡体)	Trankit, Stanza, UDPipe 2, spacy-udpipe, UD-Chinese, spaCy など
中国語 (繁体)	Stanza, UDPipe 2, spacy-udpipe, UD-Chinese, spaCy など

UD 依存構造コーパスの交換用フォーマットとして、CoNLL-U と呼ばれるタブ区切りテキスト (文字コードは UTF-8) が規定されている。CoNLL-U の各行は各単語に対応しており、以下に示す 10 個のタブ区切りフィールドで構成される。

1. ID: 単語ごとに付与されたインデックスで、文ごとに 1 から始まる整数。
2. FORM: 語、または、句読記号。
3. LEMMA: 基底形、語幹。
4. UPOS: UD で規定された言語普遍な品詞タグ[6]。
5. XPOS: 言語固有の品詞タグ。
6. FEATS: UD で規定された言語普遍な形態素属性のリスト。言語固有の拡張も可。
7. HEAD: 当該の単語の係り受け元 ID。係り受け元が無い場合は 0 とする。
8. DEPREL: UD で規定された言語普遍な係り受けタグ (表 2)。HEAD が 0 の場合は root とする。言語固有の拡張も可。
9. DEPS: 複数の係り受け元を持つ場合、全ての HEAD:DEPREL ペア。
10. MISC: その他のアノテーション。

ID・FORM・LEMMA は、単語そのものに関するフィールドである。UPOS・XPOS・FEATS は、単語の品詞と形態素属性に関するフィールドである。HEAD・DEPREL・DEPS は、単語の係り受けに関するフィールドである。

UD における係り受け関係は、単語間の有向グラフを HEAD と DEPREL で記述する。HEAD は、その単語に入る有向枝のリンク元 ID を示しており、DEPREL は、その有向枝における係り受けタグである。ただし、HEAD が 0 の場合、その枝に入るリンク元は存在しない。リンクの本数は単語の個数に等しく、各リンクのリンク先は、全て互いに異なっている。すなわち、各単語から出るリンクは複数有り得るが、各単語に入るリンクは 1 つだけである。なお、リンクはループしない。

表 2: UD 係り受けタグ (DEPREL)

	Nominals	**Clauses**	**Modifier Words**	**Function Words**
Core arguments	nsubj 主語 obj 目的語 iobj 間接目的語	csubj 節主語 ccomp 節目的語 xcomp 節補語		
Non-core dependents	obl 斜格補語 vocative 呼称語 expl 形式語 dislocated 外置語	advcl 連用修飾節	advmod 連用修飾語 discourse 談話要素	aux 動詞補助成分 cop 繋辞 mark 標識
Nominal dependents	nmod 体言による連体修飾語 appos 同格 nummod 数量による修飾語	acl 連体修飾節	amod 用言による連体修飾語	det 決定詞 clf 類別詞 case 格表示
Coordination	**MWE**	**Loose**	**Special**	**Other**
conj 接続 cc 接続詞	fixed 固着 flat 並列 compound 複合	list 細目 parataxis 隣接表現	orphan 親なし goeswith 泣き別れ reparandum 言い損じ	punct 句読点 root 親 dep 未定義

[6] ADJ・ADP・ADV・AUX・CCONJ・DET・INTJ・NOUN・NUM・PART・PRON・PROPN・PUNCT・SCONJ・SYM・VERB・X の 17 種類。

UD の係り受けリンクは、Мельчук 依存文法の後裔であり、いわゆる動詞中心主義である。動詞をリンク元として、主語や目的語へとリンクする。修飾関係においては、被修飾語から修飾語へとリンクする。ただし、側置詞 (前置詞や後置詞) を体言の修飾語だとみなす点[7]が、Мельчук とは異なっている。ちなみに、コピュラ文においては、補語をリンク元として、主語へとリンクする。

なお、本稿の図版に用いている UD 可視化ツール deplacy は、LEMMA・FEATS・DEPS を可視化対象としていない。これらの議論をおこなう際は、CoNLL-U データそれ自体を見てほしい。

3 Universal Dependencies における単語の扱い

UD における基本単位は、単語である。CoNLL-U の各行も、各単語に対応している。しかしながら、単語という考え方は、全ての言語に普遍なものではない。

3.1 単語間に区切りがない言語の UD

特に、単語と単語の間に区切りがない言語においては、単語の長さにすら、必ずしも合意がない。古典中国語 UD では、とりあえず 1 文字＝ 1 語とした (図 1) 上で、固有名詞や数詞に限って 2 文字以上の単語を許しているが、これは筆者らの製作過程を合理化するためのものであり、一般に合意が取れているわけではない。中国語 (簡体) UD においては、筆者と斉鵬 (Stanza[8]製作グループの一人) の間で、単語長について合意を試みた

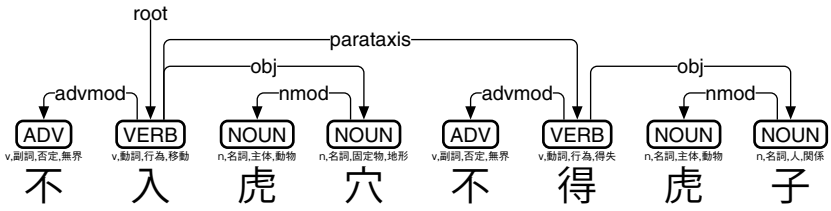


図 1: 古典中国語 UD「不入虎穴不得虎子」(UD-Kanbun)

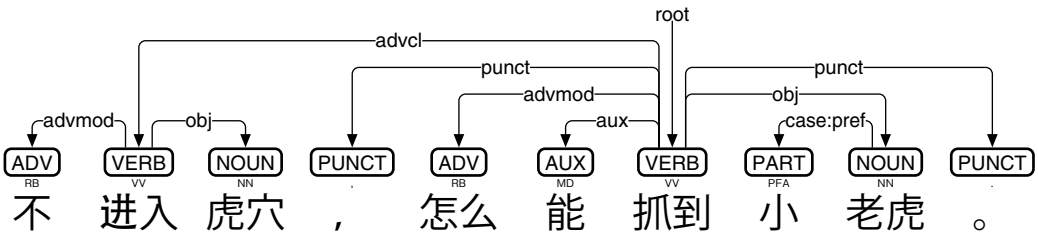


図 2: 中国語 (簡体) UD「不进入虎穴，怎么能抓到小老虎。」(Stanza)

[7]Joakim Nivre: Towards a Universal Grammar for Natural Language Processing, CICLing 2015: 16th International Conference on Intelligent Text Processing and Computational Linguistics (April 2015), pp.3-16.

[8]Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, Christopher D. Manning: Stanza: A Python Natural Language Processing Toolkit for Many Human Languages, 58th Annual Meeting of the Association for Computational Linguistics: Proceedings of the System Demonstration (July 2020), pp.101-108.

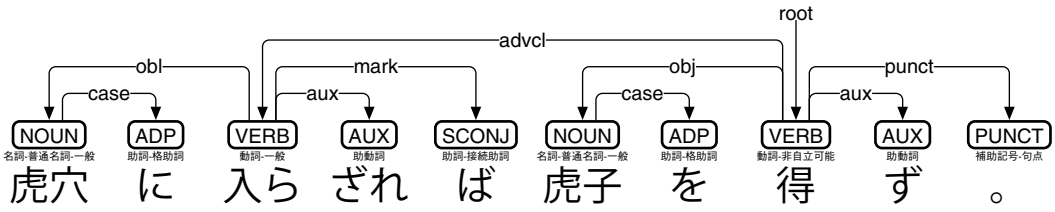


図 3: 日本語 UD「虎穴に入らざれば虎子を得ず。」(GiNZA)

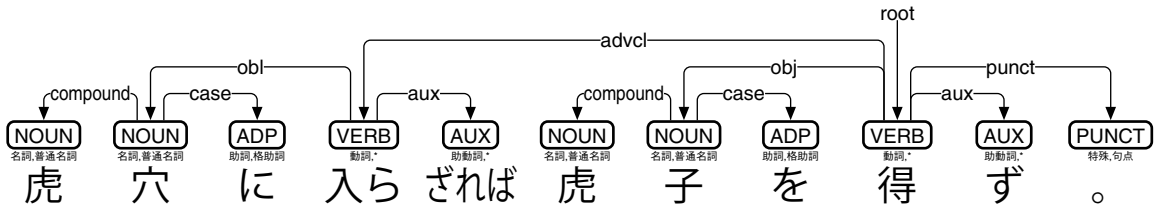


図 4: 日本語 UD「虎穴に入らざれば虎子を得ず。」(Camphr-KNP)

が、正直なところ妥協の産物 (図 2) である。結果として「虎穴」に対し、これを 2 語とみなすか 1 語とみなすか、古典中国語と現代中国語の間で異なる扱いになっている。

日本語 UD[9]は、国語研短単位[10]を 1 語とみなしている。この結果、GiNZA[11]など多くの日本語係り受け解析ツールは、国語研短単位を単語として用いている (図 3)。ただし、Camphr-KNP[12]は、JUMAN[13]品詞体系にもとづいて製作[14]されており、単語長も独特である (図 4)。特に、動詞活用「入らざれば」を、3 語とみなすか 2 語とみなすかについては、注意が必要である。

タイ語 UD に関しては、現状、あまりうまくいっていない。spaCy-Thai[15]も、Turku-neural-parser-pipeline[16]も、筆者の目標 (図 5) にはほど遠い。筆者としては、「**ถ้าเสือ**」や「**ลูกเสือ**」を 2 語に分けたいのだが、PyThaiNLP[17]のグループと合意が取れない[18]のだ。前身の ORCHID[19]における単語長が、コーパス全体の確率分布に基づくものであるため、筆者としては非常に扱いにくい。単語長を外形的に決められないと、実際の作業を進めることができず、なかなか難しいところなのである。

---

[9]浅原正幸, 金山博, 宮尾祐介, 田中貴秋, 大村舞, 村脇有吾, 松本裕治: Universal Dependencies 日本語コーパス, 自然言語処理, Vol.26, No.1 (2019 年 3 月), pp.3-36.

[10]近藤明日子: 近代文語 UniDic 短単位規程集, Ver.1.1, 立川: 国立国語研究所コーパス開発センター (2016 年 3 月).

[11]松田寛: GiNZA - Universal Dependencies による実用的日本語解析, 自然言語処理, Vol.27, No.3 (2020 年 9 月), pp.695-701.

[12]`https://camphr.readthedocs.io/en/latest/notes/knp.html`

[13]日本語形態素解析システム JUMAN version 7.0, 京都: 京都大学大学院情報学研究科黒橋・河原研究室 (2012 年 1 月).

[14]安岡孝一: 形態素解析部の付け替えによる近代日本語 (旧字旧仮名) の係り受け解析, 情報処理学会研究報告, Vol.2020-CH-124 (2020 年 9 月), No.3, pp.1-8.

[15]`https://github.com/KoichiYasuoka/spaCy-Thai`

[16]Jenna Kanerva, Filip Ginter, Niko Miekka, Akseli Leino, Tapio Salakoski: Turku Neural Parser Pipeline: An End-to-End System for the CoNLL 2018 Shared Task, Proceedings of the CoNLL 2018 Shared Task (October 2018), pp.133-142.

[17]`https://github.com/PyThaiNLP/pythainlp`

[18]現時点の PyThaiNLP は、「**ถ้าเสือ**」を 2 語とみなす一方、「**ลูกเสือ**」を 1 語とみなしている。

[19]Virach Sornlertlamvanich, Naoto Takahashi, and Hitoshi Isahara: Building a Thai part-of-speech tagged corpus (ORCHID), Journal of the Acoustical Society of Japan (E), Vol.20, No.3 (May 1999), pp.189-198.

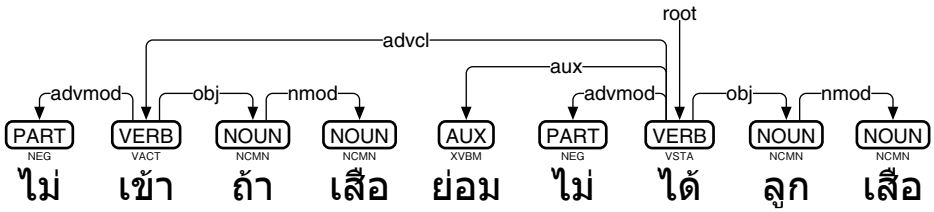


図 5: タイ語 UD「ไม่เข้าถ้าเสือ ย่อมไม่ได้ลูกเสือ」(筆者の目標)

3.2 縮約冠詞など縮約語に対する UD

単語間に空白を有する言語においても、単語認定において注意すべき点は、いくつかある。その一つが縮約冠詞である。

フランス語 UD においては、縮約冠詞を前置詞と冠詞に分解した上で、品詞付与や係り受け解析をおこなう。たとえば「aux」は、「à」と「les」に分解した上で、その後の解析をおこなう。Stanza や Trankit[20]は、この仕様を忠実に守っており、実際の係り受け解析においても、自動的に縮約冠詞の分解をおこなう (図 6)。一方、spaCy[21]や NLP-Cube[22]や COMBO-pytorch[23]は、縮約冠詞の分解を怠っている (図 7)。イタリア語 UD・ポルトガル語 UD・ガリシア語 UD・ドイツ語 UD においても同様に、縮約冠詞の分解が規定されているが、spaCy や NLP-Cube や COMBO-pytorch は仕様に従っていない。

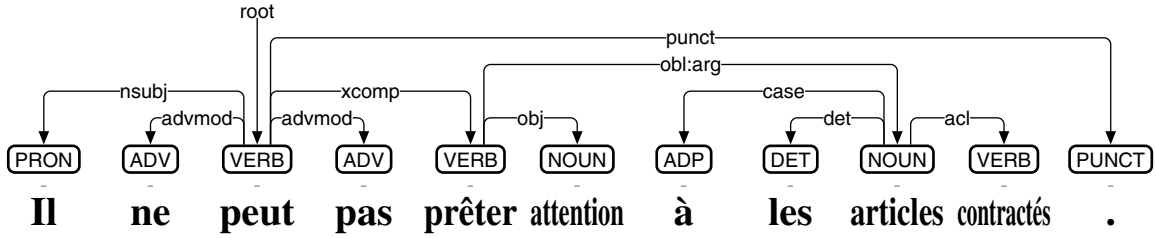


図 6: フランス語 UD「Il ne peut pas prêter attention aux articles contractés.」(Trankit)

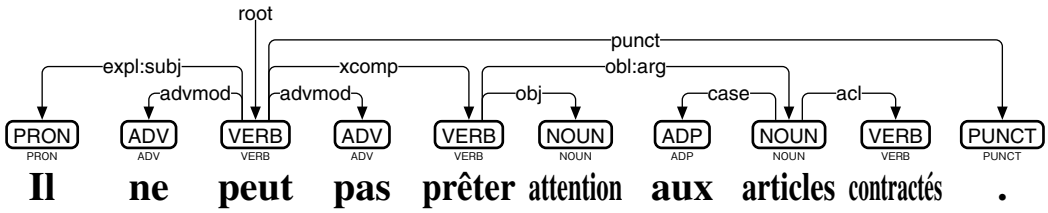


図 7: フランス語 UD「Il ne peut pas prêter attention aux articles contractés.」(spaCy)

[20]Minh Van Nguyen, Viet Dac Lai, Amir Pouran Ben Veyseh and Thien Huu Nguyen: Trankit: A Light-Weight Transformer-based Toolkit for Multilingual Natural Language Processing, EACL 2021: 16th Conference of the European Chapter of the Association for Computational Linguistics (April 2021), System Demonstrations, pp.80-90.

[21]`https://spacy.io`

[22]Tiberiu Boros, Stefan Daniel Dumitrescu, Ruxandra Burtica: NLP-Cube: End-to-end raw text processing with neural networks, Proceedings of the CoNLL 2018 Shared Task (October 2018), pp.171-179.

[23]`https://gitlab.clarin-pl.eu/syntactic-tools/combo`

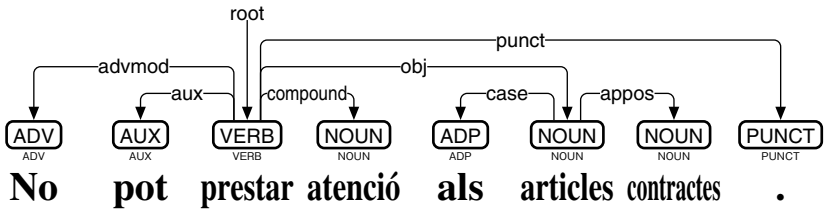


図 8: カタルーニャ語 UD「No pot prestar atenció als articles contractes.」(Trankit)

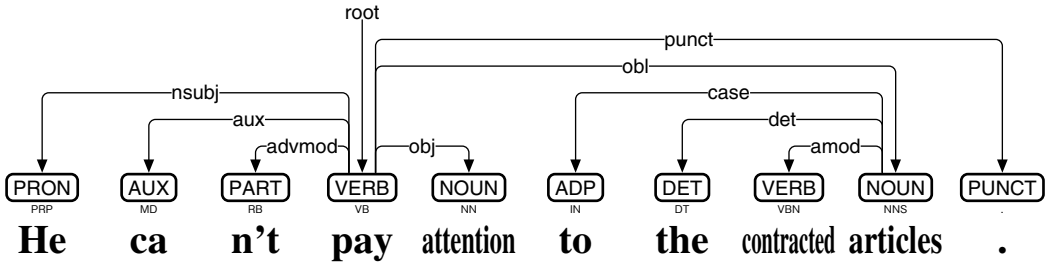


図 9: 英語 UD「He can't pay attention to the contracted articles.」(Stanza)

カタルーニャ語 UD は、UD2.7[24]以前は縮約冠詞を分解していなかったが、UD2.8[25]で縮約冠詞の分解を規定した。たとえば「als」は、「a」と「els」に分解することになったが、Trankit は、まだ UD2.8 に追いついていない (図 8)。スペイン語 UD も同様に、UD2.8 で縮約冠詞の分解を規定したが、これをサポートしているのは、本稿執筆時点では Stanza だけである。

英語 UD は、縮約語の分解を徹底的におこなっている。ただし、分解時に語形を変形しない、というポリシーがあるらしく、たとえば「can't」は「ca」と「n't」に分解する (図 9)。結果として、英語 UD における単語の区切りは、空白以外でも起こりうることになる。

日本語 UD は、縮約語の分解をおこなっていない。国語研短単位に、縮約語の分解という考え方がないのである。たとえば図 10 では、「とく」を「て」「おく」に分解していない。筆者としては、日本語 UD において縮約語の分解を導入すべきではなく、現状のままの方がいいと考えているが、今後、議論となる可能性はあるだろう。

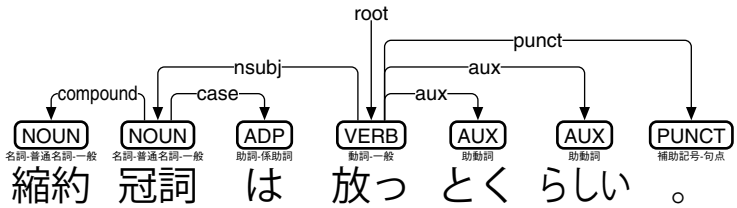


図 10: 日本語 UD「縮約冠詞は放っとくらしい。」(GiNZA)

[24]http://hdl.handle.net/11234/1-3424
[25]http://hdl.handle.net/11234/1-3687

3.3 空白が単語の区切りではない言語の UD

ベトナム語の空白は、音節を区切るものであり、単語を区切るものではない。2 音節以上の単語は、ベトナム語 UD において、単語中に空白を含むことになる。ただ、図 11 の例において「mũi tên」が 1 語なら、「con chim」も 1 語であるように筆者には思えるのだが、Trankit は「con chim」を 2 語とみなしている。ベトナム語 UD は、この点に関する規準を示しておらず、何とも悩ましい。

韓国語の空白は、語節を区切るものであり、単語を区切るものではない。ところが韓国語 UD は、1 語節＝1 語とみなす一方、各語節の XPOS では、複数の単語の品詞が＋で繋がれている (図 12)。世宗コーパス[26]以来の伝統らしいが、たとえば、韓国人の姓名を姓と名に分けずに 1 語とみなす、という点で、筆者には得心がいかない。

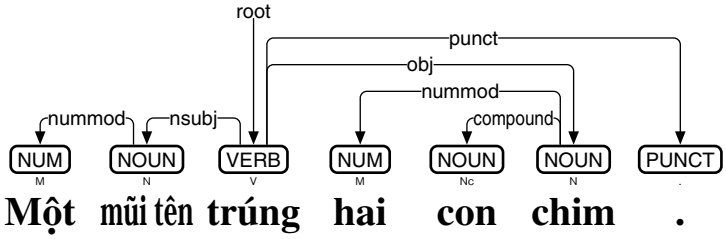


図 11: ベトナム語 UD「Một mũi tên trúng hai con chim.」(Trankit)

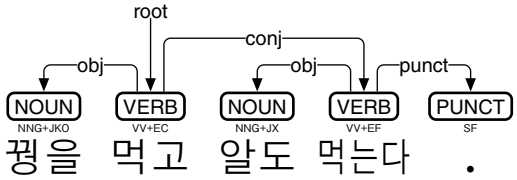


図 12: 韓国語 UD「꿩을 먹고 알도 먹는다.」(Trankit)

4 Universal Dependencies における節の扱い

UD 依存構造は、Мельчук 依存文法における係り受けタグを、表 2 の 37 種類に限定する野心的な試み、として捉えることができる。ただ、UD の前身である Stanford Typed Dependencies[27]が、初期には Bresnan 語彙機能文法[28]からスタートしており、その後に依存文法へとシフトしたことから、いくつか不思議な概念が紛れ込んでしまっている。その一つが、節 (clause) である。

節という概念を、Мельчук は依存文法から削除したが、これを UD は再導入している。表 2 の nsubj と csubj は、いずれも主語を表す UD 依存構造タグであり、リンク先が節

[26]Hansaem Kim: Korean National Corpus in the 21st Century Sejong Project, 第 13 回国立国語研究所国際シンポジウム『言語コーパスの構築と活用』(2006 年 3 月), pp.49-54.
[27]Marie-Catherine de Marneffe and Christopher D. Manning: The Stanford Typed Dependencies Representation, Coling 2008: Proceedings of the Workshop on Cross-Framework and Cross-Domain Parser Evaluation (August 2008), pp.1-8.
[28]Joan Bresnan: Lexical-Functional Syntax, Malden: Blackwell (2001).

である時は csubj を、そうでない時は nsubj を使う。obj と ccomp についても同様だし、advmod と advcl についても同様だし、amod と acl についても同様である。case と mark については、リンク元が節である時は mark を、そうでない時は case を使う[29]。

ただ、ある単語 (あるいは単語の列) が節を構成しているかどうかは、なかなかに判断が難しい。たとえば、図 6 でフランス語 UD は、屈折した動詞「contractés」が節であるとみなして、acl を用いている。一方、図 9 で英語 UD は、屈折した動詞「contracted」が節であるとみなさず、amod を用いている。これらを見る限り、節という概念を言語横断的に規定するのは、非現実的であるようにすら筆者には思える。

また、UD におけるコピュラ文の扱いは、補語が節である場合、まずいことになる。たとえばバスク語は、主語も補語もコピュラ節であるようなコピュラ文を簡単に書けるが、これを UD で表すと、かなり異様な形となる (図 13)。「Euskaldun izatea」が節主語、「lan extra bat izatea」が節補語で、それらを繋辞「da」が繋いでいるはずなのだが、そう筆者には見えない。節補語の中にある繋辞「izatea」と、外にある繋辞「da」が、混ざって見えてしまう。この問題を回避すべく、spaCy-ixaKat[30]は、「izan」(およびその屈折) を繋辞とみなさず、一般の動詞と同じように処理 (図 14) している。しかし、これはこれで UD の規定に違反しており、筆者としては悩ましい。

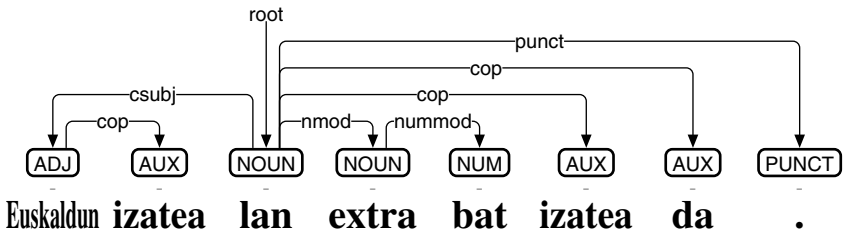


図 13: バスク語 UD「Euskaldun izatea lan extra bat izatea da.」(Turku-neural-parser-pipeline)

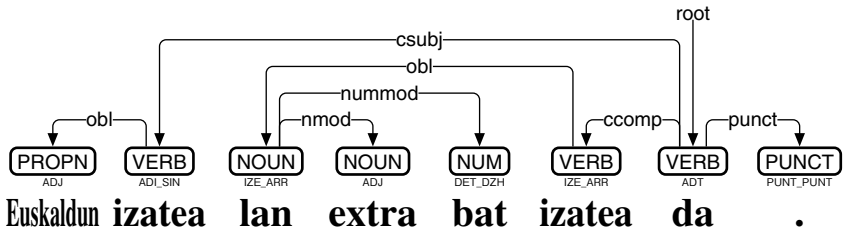


図 14: バスク語 UD「Euskaldun izatea lan extra bat izatea da.」(spaCy-ixaKat)

[29]Marie-Catherine de Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, Christopher D. Manning: Universal Stanford Dependencies: A Cross-Linguistic Typology, Proceedings of the 9th International Conference on Language Resources and Evaluation (May 2014), pp.4585-4592.

[30]`https://github.com/KoichiYasuoka/spaCy-ixaKat`

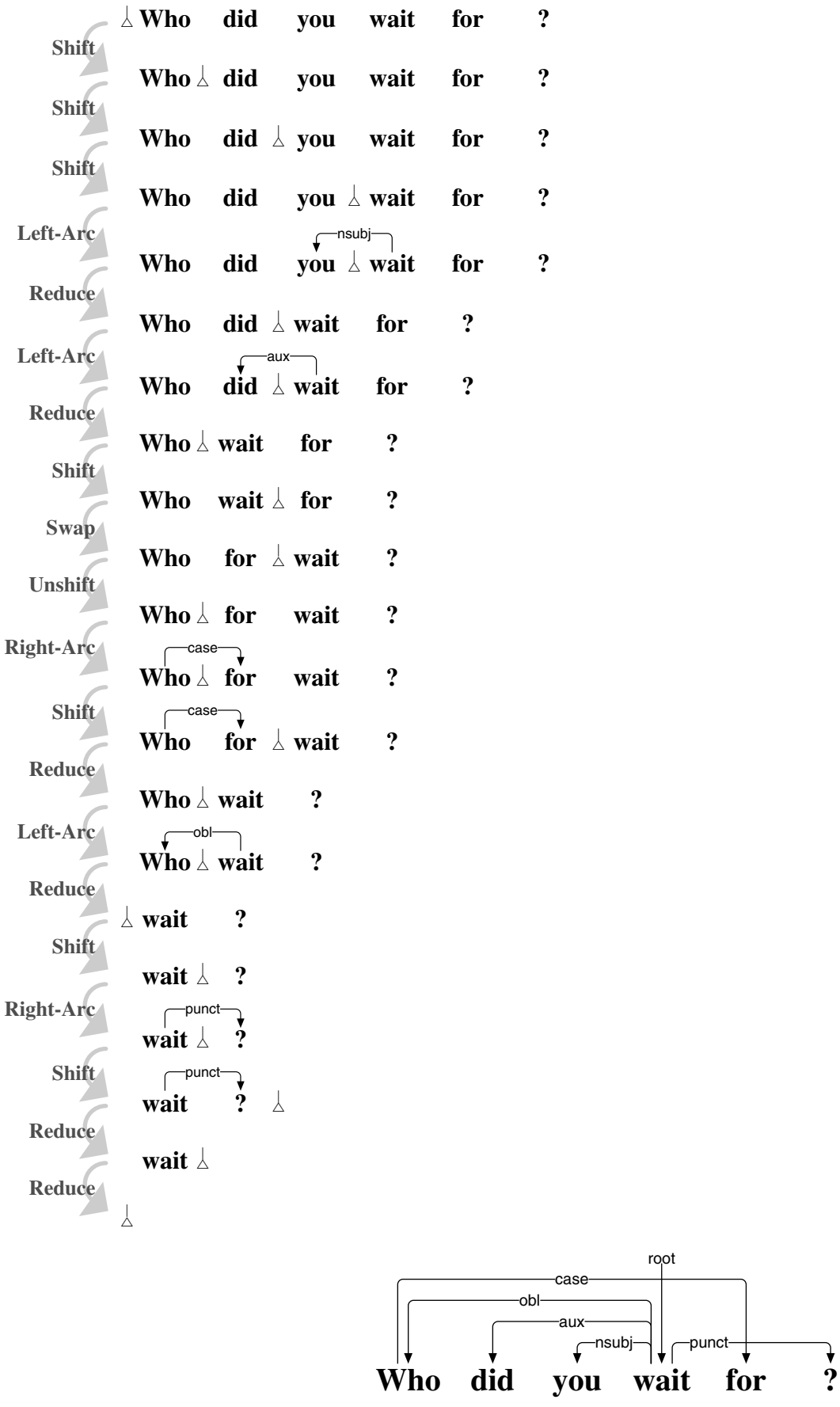


図 15: 状態遷移型アルゴリズムによる「Who did you wait for?」の係り受け解析 (概念図)

5 Universal Dependencies における係り受け解析

係り受け解析は、UD による言語処理の「キモ」であり、多くの解析アルゴリズムが乱立している。本稿では、これらの解析アルゴリズムを、状態遷移型・隣接行列型・系列ラベリング型の 3 つに分けて、概説する。

5.1 状態遷移型アルゴリズム

arc-swap[31]に代表される状態遷移型アルゴリズムは、単語列の先頭から末尾に向かって「垣根」(stack-buffer boundary) を移動していく、というイメージで処理をおこなう。「垣根」がおこなう遷移は、以下の 6 種類に定式化される。

- **Shift** 「垣根」を右に 1 単語分、移動する。
- **Reduce** 「垣根」のすぐ左の単語を除去して、解析結果へ移す。
- **Left-Arc** 「垣根」のすぐ右の単語から、すぐ左の単語へリンクを繋ぐ。
- **Right-Arc** 「垣根」のすぐ左の単語から、すぐ右の単語へリンクを繋ぐ。
- **Unshift** 「垣根」を左に 1 単語分、移動する。
- **Swap** 「垣根」のすぐ右の単語と、すぐ左の単語を入れ替える。

単語が全て **Reduce** されて、「垣根」がポツンと取り残された時点で、アルゴリズムは終了である。状態遷移型アルゴリズムによる解析例を、図 15 に示す。解析結果において、入るリンクがない単語に対しては、通常は root を割り当てる。

spacy-udpipe[32]・UD-Kanbun[2]・UniDic2UD[14]は、土台となっている UDPipe[33]が状態遷移型アルゴリズムに **Unshift**・**Swap** を実装しておらず、結果として、リンクに交差がある UD を扱えない。これに対し、spaCy は **Unshift** を実装[34]しており、どうやら **Swap** も実装しているようである。なお、現実の実装においては、**Left-Arc** の直後には **Reduce** を、**Right-Arc** の直後には **Shift** と **Reduce** を、それぞれまとめて遷移させる手法が主流である。

状態遷移型アルゴリズムは、BERT[35]など事前学習モデルによる精度向上が期待できる[36]はずだが、これは実装がかなり難しい。残念ながら、手間暇がかかる割りに、隣接行列型アルゴリズムの精度に追いつけていない、というのが、本稿執筆時点での筆者の見解である。

---

[31]Joakim Nivre: Non-Projective Dependency Parsing in Expected Linear Time, Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP (August 2009), pp.351-359.

[32]`https://github.com/TakeLab/spacy-udpipe`

[33]Milan Straka and Jana Straková: Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe, Proceedings of the CoNLL 2017 Shared Task (August 2017), pp.88-99.

[34]Matthew Honnibal, Mark Johnson: An Improved Non-monotonic Transition System for Dependency Parsing, Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (September 2015), pp.1373-1378.

[35]Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova: BERT: Pre-training of Deep Bidirectional Transformers for Language, Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (June 2019), Human Language Technologies, Vol.1, pp.4171-4186.

[36]Alireza Mohammadshahi, James Henderson: Graph-to-Graph Transformer for Transition-based Dependency Parsing, Findings of the Association for Computational Linguistics: EMNLP 2020 (November 2020), pp.3278-3289.

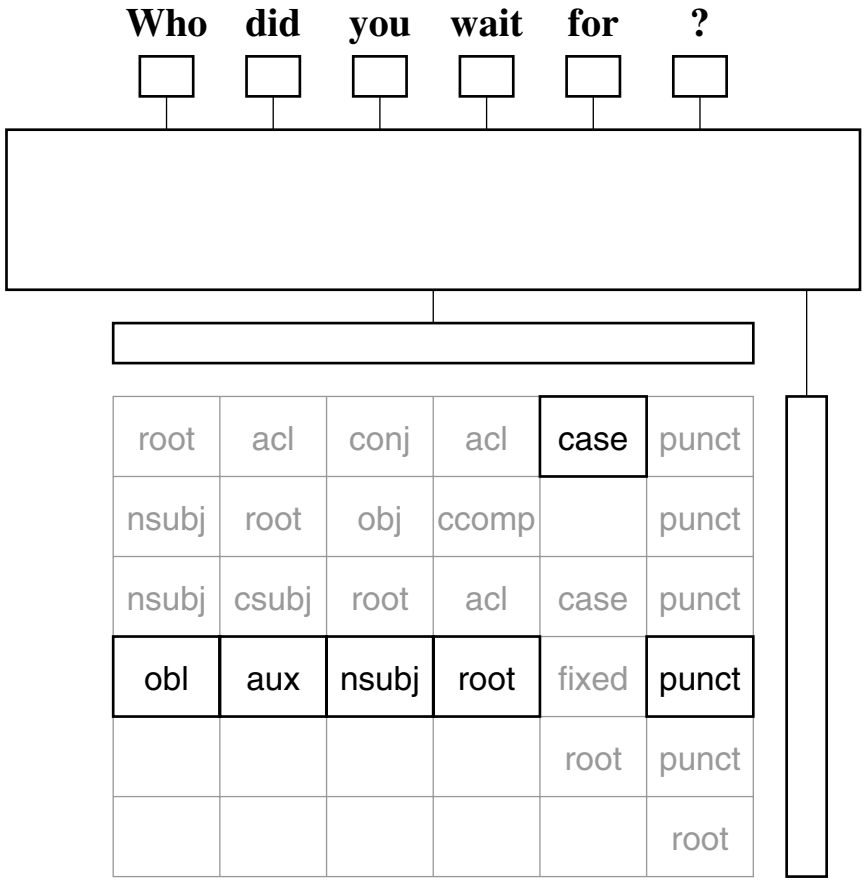


図 16: 隣接行列型アルゴリズムによる「Who did you wait for?」の係り受け解析 (模式図)

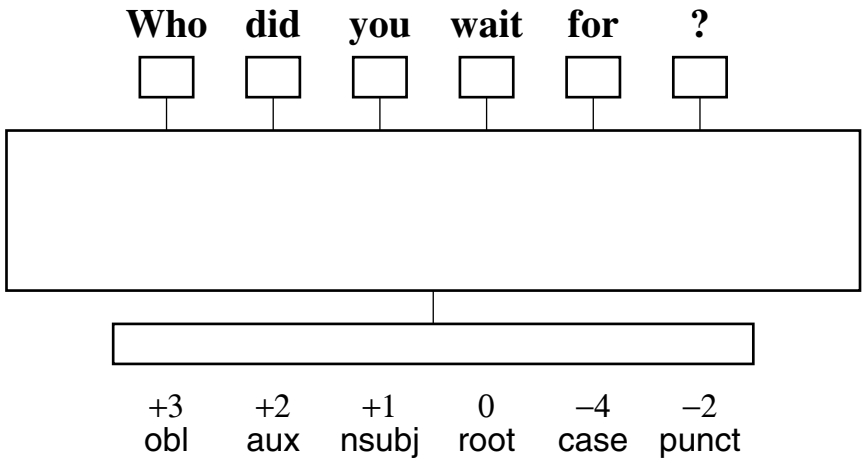
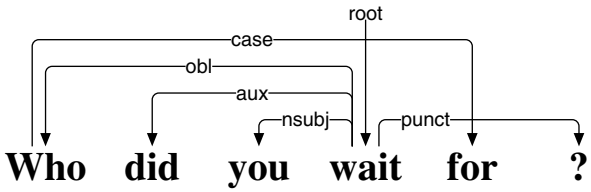


図 17: 系列ラベリングによる「Who did you wait for?」の係り受け解析 (模式図)

5.2 隣接行列型アルゴリズム

Biaffine[37]に代表される隣接行列型アルゴリズムは、UD 依存構造を有向グラフとみなし、その隣接行列を求める。たとえば



という有向グラフであれば

$$\begin{bmatrix} - & - & - & - & \text{case} & - \\ - & - & - & - & - & - \\ - & - & - & - & - & - \\ \text{obl} & \text{aux} & \text{nsubj} & \text{root} & - & \text{punct} \\ - & - & - & - & - & - \\ - & - & - & - & - & - \end{bmatrix}$$

という隣接行列を求めることになる (図 16)。具体的には、隣接行列の各列ごとに要素を 1 つ選びつつ、対角成分上の root から 1 つを選ぶ、というのを確率的におこなう。ただし、それだけでは、ループを必ずしも防げないことから、Chu-Liu-Edmonds 等[38]を適宜、併用する。

隣接行列型アルゴリズムは、BERT など事前学習モデルとの相性が良く、実際、Camphr-Udify[39]は bert-base-cased[35]を、Trankit は XLM-RoBERTa[40]を、それぞれ事前学習モデルとして用いている。筆者らの SuPar-Kanbun[41]は 6 種類の事前学習モデル (古典中国語) を、SuPar-UniDic[42]は 14 種類の事前学習モデル (日本語) を、それぞれ繋ぎ替えて楽しめる。一方、Stanza や NLP-Cube は、隣接行列型アルゴリズムに事前学習モデルを用いていない。

5.3 系列ラベリング型アルゴリズム

系列ラベリング型アルゴリズムによる係り受け解析は、隣接行列を行ベクトルに圧縮し、各リンク長をオフセットとして求める形で、定式化できる (図 17)。ID と HEAD の差を、DEPREL と同時に求めるアルゴリズム、だと考えることもできる。理論的には、隣

[37]Timothy Dozat, Christopher D. Manning: Deep Biaffine Attention for Neural Dependency Parsing, 5th International Conference on Learning Representations (April 2017), C25.
[38]H. N. Gabow, Z. Galil, T. Spencer and R. E. Tarjan: Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs, Combinatorica, Vol.6, No.2 (June 1986), pp.109-122.
[39]`https://camphr.readthedocs.io/en/latest/notes/udify.html`
[40]Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, Veselin Stoyanov: Unsupervised Cross-lingual Representation Learning at Scale, Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (July 2020), pp.8440-8451.
[41]`https://github.com/KoichiYasuoka/SuPar-Kanbun`
[42]`https://github.com/KoichiYasuoka/SuPar-UniDic`

接行列型アルゴリズムと同等の精度で解析可能なはずだが、現実には、なかなか解析精度を上げにくい。

系列ラベリング型アルゴリズムが品詞付与にも適用可能なことから、COMBO-pytorch は、HEAD･DEPREL に加え UPOS･XPOS･FEATS をも同時に導出している。また、BERT など事前学習モデルとの相性も良く、筆者らの UniDic-COMBO は bert-base-japanese-whole-word-masking[43]を、GuwenCOMBO は GuwenBERT[44]を、それぞれ事前学習モデルとして用いている。

6　日本語 UD を用いた係り受け解析器の自作

ここまで述べてきた内容をもとに、日本語係り受け解析器を自作してみよう。自作と言っても一から作るわけではなく、既存データと既存ツールの組み合わせで、何とかすることを考える。まずは、日本語 UD の既存データをダウンロードしよう。

- `https://github.com/UniversalDependencies/UD_Japanese-GSD`
- `https://github.com/UniversalDependencies/UD_Japanese-PUD`
- `https://github.com/UniversalDependencies/UD_Japanese-Modern`
- `https://github.com/UniversalDependencies/UD_Japanese-BCCWJ`
- `https://github.com/UniversalDependencies/UD_Japanese-KTC`

`GSD` は、日本語 Wikipedia をもとにした現代日本語 UD であり、train(7050 文 168333 語)･dev(507 文 12287 語)･test(543 文 13034 語) の 3 つの CoNLL-U データからなる。`PUD` は、英語から翻訳された日本語文の UD であり、test データ (1000 文 28787 語) のみである。`Modern` は、『明六雑誌』をもとにした近代日本語 UD であり、test データ (822 文 14494 語) のみである。これらはいずれも、単語長に国語研短単位を採用[45]している。`BCCWJ` と `KTC` は、本文が別売 (上記データ上は「墨塗り」) である。

使い勝手が良さそうなのは `GSD` だ。`GSD` の train データに、目的に応じて `PUD` なり `Modern` なりを加えてやるのが、良さそうである。

6.1　UDPipe を用いる場合

UDPipe は状態遷移型アルゴリズムを採用しており、C++版[46]と python 版[47]がある。C++版を用いた解析器の自作は、コマンドラインから

```
udpipe --train my.udpipe train.conllu
```

の 1 行である。これで、訓練用データ `train.conllu` から `my.udpipe` が作られる。出来上がった `my.udpipe` で係り受け解析をおこなうには、コマンドラインから

[43]`https://huggingface.co/cl-tohoku/bert-base-japanese-whole-word-masking`
[44]閻覃, 遅澤聞: 基于継続訓練的古漢語語言模型, 第 19 届中国計算語言学大会 “古聯杯” 古籍文献命名実体識別 (2020 年 10 月 31 日).
[45]松田寛, 若狭絢, 山下華代, 大村舞, 浅原正幸: UD Japanese GSD の再整備と固有表現情報付与, 言語処理学会第 26 回年次大会発表論文集 (2020 年 3 月), pp.133-136.
[46]`https://ufal.mff.cuni.cz/udpipe/1/install`
[47]`https://pypi.org/project/ufal.udpipe`

```
echo 虎穴に入らざれば虎穴を得ず。 | udpipe --tokenize --tag --parse my.udpipe
```

の 1 行である。python 版の場合は、以下のプログラムで解析をおこなう。

```
import ufal.udpipe
mdl = ufal.udpipe.Model.load("my.udpipe")
nlp = ufal.udpipe.Pipeline(mdl, "tokenize", "", "", "").process
doc = nlp("虎穴に入らざれば虎子を得ず。")
print(doc)
```

ただし、UDPipe の解析精度は、やや低い。また、リンクに交差がある UD を扱えない点にも注意されたい。

6.2 SuPar と bert-large-japanese と fugashi を用いる場合

SuPar[48]は、隣接行列型アルゴリズムや系列ラベリング型アルゴリズムを、多数サポートしており、python3.7 以上で動作する。Biaffine 解析器の自作は、コマンドラインから以下の 1 行 (複数行に分かれているが 1 行に書いてほしい) を実行する。

```
biaffine-dep train -b -c biaffine-dep-en -p my.supar -f bert
  --bert cl-tohoku/bert-large-japanese --embed=
  --train train.conllu --dev dev.conllu --test test.conllu
```

GPU が使える場合は「`-d 0`」オプションも追加する。これで、訓練用 `train.conllu`・評価用 `dev.conllu`・テスト用 `test.conllu` の 3 データから、bert-large-japanese[49] (日本語 Wikipedia をもとにしており、単語長は国語研短単位) を事前学習モデルに使いつつ、`my.supar` が作られる。

ただし、SuPar は、単語切りや品詞付与はおこなわない。出来上がった `my.supar` も、おこなえるのは係り受け解析 (HEAD・DEPREL の付与) だけであり、以下の python プログラムのように、解析したい文を、国語研短単位で切って渡さねばならない。

```
import supar
prs = supar.Parser.load("my.supar")
nlp = lambda x: prs.predict([x], lang=None).sentences[0]
doc = nlp(["虎穴","に","入ら","ざれ","ば","虎子","を","得","ず","。"])
print(doc)
```

このあたり、もう少し何とかしたい。アイデアの一つは、Transformers[50]のトークナイザを借りて、bert-large-japanese に合わせた単語切りをおこなう方法である。

[48]Yu Zhang, Zhenghua Li, Min Zhang: Efficient Second-Order TreeCRF for Neural Dependency Parsing, Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (July 2020), pp.3295-3305.

[49]`https://huggingface.co/cl-tohoku/bert-large-japanese`

[50]Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, Alexander M. Rush: Transformers: State-of-the-Art Natural Language Processing, Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (October 2020), System Demonstrations, pp.38-45.

```
import supar
from transformers import AutoTokenizer
prs = supar.Parser.load("my.supar")
brt = "cl-tohoku/bert-large-japanese"
tkz = AutoTokenizer.from_pretrained(brt, do_subword_tokenize=False)
nlp = lambda x: prs.predict([tkz.tokenize(x)], lang=None).sentences[0]
doc = nlp("虎穴に入らざれば虎子を得ず。")
print(doc)
```

この AutoTokenizer の実体は、fugashi[51] と unidic-lite[52] なので、置き換えてみよう。

```
import supar, fugashi, unidic_lite
prs = supar.Parser.load("my.supar")
tag = fugashi.Tagger("-d " + unidic_lite.DICDIR)
def nlp(sentence):
  s = tag(sentence)
  d = prs.predict([[t.surface for t in s]], lang=None).sentences[0]
  return d
doc = nlp("虎穴に入らざれば虎子を得ず。")
print(doc)
```

さらに LEMMA と XPOS も、fugashi の出力を流用しよう。また、日本語 UD では、単語間に空白を含まない場合、MISC に「SpaceAfter=No」を入れていることから、これも fugashi の出力を流用しよう。

```
import supar, fugashi, unidic_lite
prs = supar.Parser.load("my.supar")
tag = fugashi.Tagger("-d " + unidic_lite.DICDIR)
def nlp(sentence):
  s = tag(sentence)
  d = prs.predict([[t.surface for t in s]], lang=None).sentences[0]
  d.values[2] = [t.feature.lemma for t in s]
  d.values[4] = [t.pos.replace(",*", "").replace(",", "-") for t in s]
  d.values[9] = ["_" if t.white_space else "SpaceAfter=No" for t in s]
  return d
doc = nlp("虎穴に入らざれば虎子を得ず。")
print(doc)
```

UPOS を XPOS と DEPREL から生成すれば (次ページ)、SuPar と bert-large-japanese と fugashi を用いた日本語係り受け解析器の完成である。なお、SuPar-UniDic[42] では、他の UniDic[53] や、他の事前学習モデルとの組み合わせにも、筆者なりに挑戦している。事前学習モデルの単語長が国語研短単位と異なる場合は、図 18 で示すような工夫[54] が必要となるので、注意されたい。

[51] Paul McCann: fugashi, a Tool for Tokenizing Japanese in Python, Proceedings of Second Workshop for NLP Open Source Software (November 2020), pp.44-51.
[52] https://github.com/polm/unidic-lite
[53] https://unidic.ninjal.ac.jp
[54] SuPar 1.0.1a1 以降、char モードに内部実装してもらった。

```
import supar, fugashi, unidic_lite
prs = supar.Parser.load("my.supar")
tag = fugashi.Tagger("-d " + unidic_lite.DICDIR)
def nlp(sentence):
  s = tag(sentence)
  d = prs.predict([[t.surface for t in s]], lang=None).sentences[0]
  d.values[2] = [t.feature.lemma for t in s]
  x = {"名詞":"NOUN", "代名詞":"PRON", "動詞":"VERB", "助動詞":"AUX",
    "形容詞":"ADJ", "形状詞":"ADJ", "連体詞":"DET", "副詞":"ADV",
    "助詞":"ADP", "接続詞":"CCONJ", "接頭辞":"NOUN", "接尾辞":"PART",
    "感動詞":"INTJ", "補助記号":"PUNCT", "記号":"SYM", "空白":"SYM"}
  y = {"助動詞語幹":"AUX", "固有名詞":"PROPN", "数詞":"NUM",
    "終助詞":"PART", "接続助詞":"SCONJ", "名詞的":"NOUN"}
  z = {"aux":"AUX", "cop":"AUX", "advmod":"ADV", "amod":"ADJ"}
  u = []
  for i,t in enumerate(s):
    f,g = t.feature,d.values[7][i]
    u.append(z[g] if g in z else y[f.pos2] if f.pos2 in y else x[f.pos1])
    if g == "aux" and int(d.values[0][i]) - int(d.values[6][i]) == 1:
      h = s[i-1].feature.pos3
      if h.find("形状詞可能") >= 0 and f.lemma in ["だ", "なり"]:
        u[i-1] = "ADJ"
      elif h.startswith("サ変"):
        u[i-1] = "VERB"
  d.values[3] = u
  d.values[4] = [t.pos.replace(",*", "").replace(",", "-") for t in s]
  d.values[9] = ["_" if t.white_space else "SpaceAfter=No" for t in s]
  return d
doc = nlp("虎穴に入らざれば虎子を得ず。")
print(doc)
```

6.3 Transformers と bert-large-japanese-char-extended を用いる場合

Transformers の系列ラベリングを使って、日本語係り受け解析器を作ってみよう。事前学習モデルには、国語研短単位と異なる単語長 (1 文字＝ 1 語) の bert-large-japanese-char-extended[55]を、使ってみることにする。工夫としては goeswith を用いて (図 18)、訓練用データ `train.conllu` を変形し、単語長を bert-large-japanese-char-extended に合わせる。

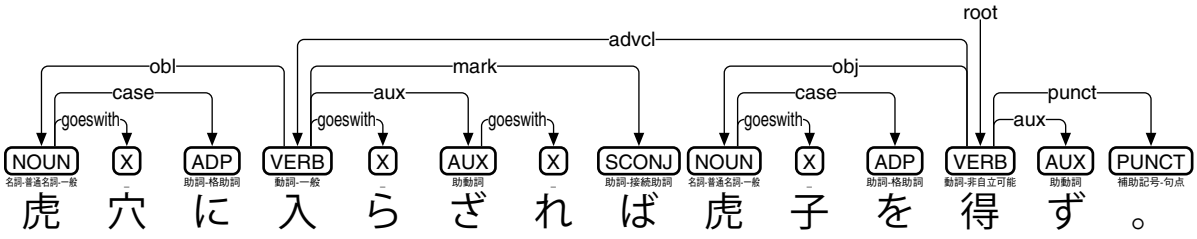


図 18: 単語長を 1 文字＝ 1 語に変形した日本語 UD

[55]https://huggingface.co/KoichiYasuoka/bert-large-japanese-char-extended

具体的には、以下の python プログラムで、train.conllu から train-c.conllu を作成する。

```
with open("train.conllu", "r", encoding="utf-8") as f:
  r = f.read()
with open("train-c.conllu", "w", encoding="utf-8") as f:
  for u in r.strip().split("\n\n"):
    w = [t for t in u.split("\n") if t.startswith("# text = ")][0]
    s = [t.split("\t") for t in u.split("\n") if not t.startswith("#")]
    for i,v in enumerate([v for v in w[9:] if not v.isspace()]):
      t = s[i]
      x,t[0],t[1],t[2],t[8] = t[1],str(i+1),v,"_","_"
      t[9] = "_" if t[9].find("SpaceAfter=No") < 0 else "SpaceAfter=No"
      if v != x:
        s.insert(i+1, [str(i+2),x[1:],"_","X","_","_",t[6]
           if t[7] == "goeswith" else str(i+1),"goeswith","_",t[9]])
        t[9] = "SpaceAfter=No"
        for t in [t for t in s if int(t[6]) > i+1]:
          t[6] = str(int(t[6])+1)
    print(w, "\n".join("\t".join(t) for t in s), "", sep="\n", file=f)
```

解析器の自作は、以下の python プログラムで、Transformers の系列ラベリング型モデルを作成する。ここでは、UPOS・XPOS・FEATS・ID と HEAD の差・DEPREL の 5 つを、合わせてラベリングしている。

```
from transformers import (AutoTokenizer, AutoConfig,
  AutoModelForTokenClassification, DataCollatorForTokenClassification,
  TrainingArguments, Trainer)
from datasets.arrow_dataset import Dataset
brt = "KoichiYasuoka/bert-large-japanese-char-extended"
with open("train-c.conllu", "r", encoding="utf-8") as f:
  tok,tag = [],[]
  for s in f.read().strip().split("\n\n"):
    v = [t.split("\t") for t in s.split("\n") if not t.startswith("#")]
    tok.append([t[1] for t in v])
    tag.append(["\t".join(t[3:6]+["{:+}".format(int(t[6])-int(t[0]))
      if int(t[6]) else "0", t[7]]) for t in v])
lid = {l:i for i,l in enumerate(set(sum(tag, [])))}
tkz = AutoTokenizer.from_pretrained(brt)
dts = Dataset.from_dict({"tokens": tok, "tags": tag,
  "input_ids": [tkz.convert_tokens_to_ids(s) for s in tok],
  "labels": [[lid[t] for t in s] for s in tag]})
cfg = AutoConfig.from_pretrained(brt, num_labels=len(lid), label2id=lid,
  id2label={i:l for l,i in lid.items()})
mdl = AutoModelForTokenClassification.from_pretrained(brt, config=cfg)
dcl = DataCollatorForTokenClassification(tokenizer=tkz)
arg = TrainingArguments(output_dir="/tmp", overwrite_output_dir=True,
  per_device_train_batch_size=4, save_total_limit=2)
trn = Trainer(model=mdl, args=arg, data_collator=dcl, train_dataset=dts)
trn.train()
trn.save_model("my.trans")
tkz.save_pretrained("my.trans")
```

per_device_train_batch_size は、使える GPU の大きさによって、適宜、変更した方がいいだろう。これで、train-c.conllu から、bert-large-japanese-char-extended を事前学習モデルに使いつつ、my.trans 配下に系列ラベリング型モデルが作られる。出来上がった my.trans で係り受け解析をおこなうには、以下の python プログラムを使う。

```
import torch
from transformers import AutoTokenizer, AutoModelForTokenClassification
tkz = AutoTokenizer.from_pretrained("my.trans")
mdl = AutoModelForTokenClassification.from_pretrained("my.trans")
def nlp(sentence):
  s = [t for t in sentence if not t.isspace()]
  t = [i for i,t in enumerate(sentence) if t.isspace()]
  m = [i-j-1 for j,i in enumerate(t)]
  e = tkz.encode(s, return_tensors="pt", add_special_tokens=False)
  for i,q in enumerate(torch.argmax(mdl(e)[0], dim=2)[0].tolist()):
    t = mdl.config.id2label[q].split("\t")
    t[3] = str(int(t[3])+i+1) if int(t[3]) else "0"
    s[i] = [s[i],"_"]+t+["_","_" if i in m else "SpaceAfter=No"]
  return "\n".join("\t".join([str(i+1)]+t) for i,t in enumerate(s))+"\n\n"
doc=nlp("虎穴に入らざれば虎子を得ず。")
print(doc)
```

ただし、これだと結果に goeswith が残ってしまうので、直前の文字とくっつける形で goeswith を削り取ろう。

```
import torch
from transformers import AutoTokenizer, AutoModelForTokenClassification
tkz = AutoTokenizer.from_pretrained("my.trans")
mdl = AutoModelForTokenClassification.from_pretrained("my.trans")
def nlp(sentence):
  s = [t for t in sentence if not t.isspace()]
  t = [i for i,t in enumerate(sentence) if t.isspace()]
  m = [i-j-1 for j,i in enumerate(t)]
  e = tkz.encode(s, return_tensors="pt", add_special_tokens=False)
  for i,q in enumerate(torch.argmax(mdl(e)[0], dim=2)[0].tolist()):
    t = mdl.config.id2label[q].split("\t")
    t[3] = str(int(t[3])+i+1) if int(t[3]) else "0"
    s[i] = [s[i],"_"]+t+["_","_" if i in m else "SpaceAfter=No"]
  for i in [i for i in range(len(s)-1, 0, -1) if s[i][6] == "goeswith"]:
    t = s.pop(i)
    s[i-1][0],s[i-1][8] = s[i-1][0]+t[0],t[8]
    for t in [t for t in s if int(t[5]) > i]:
      t[5] = str(int(t[5])-1)
  return "\n".join("\t".join([str(i+1)]+t) for i,t in enumerate(s))+"\n\n"
doc=nlp("虎穴に入らざれば虎子を得ず。")
print(doc)
```

これで、Transformers と bert-large-japanese-char-extended を用いた日本語係り受け解析器の完成である。なお、他の事前学習モデルを使う際でも、サブワード等に合わせた訓練データの変形が必須なので、注意されたい。

7 おわりに

本稿では、各言語 UD の「クセ」と、それにまつわる係り受け解析ツールの「クセ」を、かいつまんで報告した。ただ、筆者らの研究が、古典中国語の文法解析から発していることもあって、東アジアの言語にやや偏った報告となってしまった。それでも、各言語 UD ごとに異なる「クセ」がある、という点は、納得してもらえたと思う。

では、複数の言語にまたがる UD を、どういう形で作るべきか。これが、現在の筆者の興味であり、課題である。というのも、書写言語というのは、単独の言語で書かれているとは限らず、複数の言語が容易に混ざってしまうからだ (図 19)。このような言語現象を、果たして UD は捉えることができるのか。読者諸氏も、ぜひ考えてみてほしい。

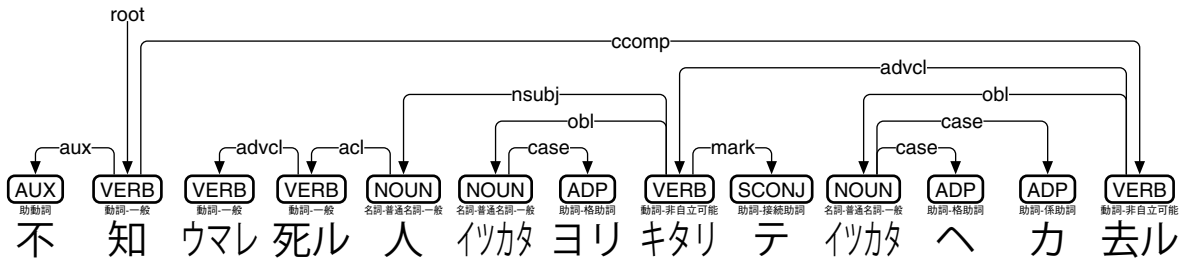


図 19: 方丈記 UD「不知ウマレ死ル人イツカタヨリキタリテイツカタヘカ去ル」(案)

なお、表 1 に示した各言語の係り受け解析ツールの実行環境と、第 6 章で示した日本語係り受け解析器の自作環境を、Google Colaboratory に準備した。Chrome・Edge・Safari などのブラウザと、Google ID があれば動かせるので、以下の URL から試してほしい。

- 各言語の係り受け解析ツールの実行環境
  `https://koichiyasuoka.github.io/deplacy#templates-for-google-colaboratory`
- 日本語係り受け解析器の自作環境
  `https://koichiyasuoka.github.io/deplacy/demo/2021-06-22`