Semantic Web 14 (2023) 1–12
IOS Press

Using Wikidata Lexemes and Items to Generate Text from Abstract Representations

Mahir Morshed [*]
University of Illinois Urbana–Champaign, Urbana, IL 61801, USA
E-mail: mmorshe2@illinois.edu

Abstract. Ninai/Udiron, a living function-based natural language generation system, uses knowledge in Wikidata lexemes and items to transform abstract representations of factual statements into human-readable text. The combined system first produces syntax trees based on those abstract representations (Ninai) and then yields sentences from those syntax trees (Udiron). The system relies on information about individual lexical units and links to the concepts those units represent, as well as rules encoded in various types of functions to which users may contribute, to make decisions about words, phrases, and other morphemes to use and how to arrange them. Various system design choices work toward using the information in Wikidata lexemes and items efficiently and effectively, making different components individually contributable and extensible, and making the overall resultant outputs from the system expectable and analyzable. These targets accompany the intentions for Ninai/Udiron to ultimately power the Abstract Wikipedia project as well as be hosted on the Wikifunctions project.

Keywords: Wikidata, Abstract Wikipedia, Natural language generation, Dependency grammar

1. Introduction

The Abstract Wikipedia project [1] has the intended goal of creating a body of structured, language-independent encyclopedic content that can be transformed into natural written text in arbitrary languages when provided functionality to do so. This functionality is intended to be hosted on the Wikifunctions project; this general repository of functions, to which volunteers would contribute, would provide the necessary code for the aforementioned transformations into natural text. Both named projects were approved for creation in July 2020 [2], but development of both are ongoing as of the time of writing [3]. There have nevertheless been invitations from their development team to the community to provide both possible ideas for functions and ideas for possible output texts, including one to submit code to a beta version of Wikifunctions [4].

Ninai[1] and Udiron[2] (respectively, the Classical Tamil 'to think' and the Bengali pronunciation of the Sanskrit 'communicating, saying, speaking') aim to bring forward the vision of Abstract Wikipedia as much as possible. Broadly speaking, Ninai is designed to handle the creation of abstract content and its processing into syntax trees, while Udiron handles the manipulation of such syntax trees and their eventual conversion into linear sentences. At the start of its development in August 2021 [5], Ninai/Udiron was the first system to demonstrate the power of Wikidata's lexicographical data in generating text, not just encyclopedic in nature but potentially with respect to other declarative textual genres as well.

[*]Corresponding author. E-mail: mmorshe2@illinois.edu.
[1]https://gitlab.com/mahir256/ninai
[2]https://gitlab.com/mahir256/udiron

The rest of this paper proceeds as follows. Section 2 details several decisions made in the course of Ninai/Udiron's development that help explain certain idiosyncratic features of the system. Section 3 discusses information added to lexemes, forms, and senses on Wikidata, including various Wikidata properties on those entities used (and potentially usable) by Ninai/Udiron as well as how the resultant relationships convey certain types of information. After section 4 discusses both the core and the wider umbrella of Ninai and Udiron components, section 5 outlines the different steps of the overall abstract content rendering process. Section 6 gives concluding remarks.

2. Background

Several aspects in the development of Ninai/Udiron, whether data-driven, linguistic, or technical, do not have trivial origins, may appear to contravene existing methodology around rule-based text generation, and yet continue to serve as guiding principles for the system's future development.

2.1. Data-driven design choices

The outputs from Ninai/Udiron are intended to be composed to the maximum extent possible through combining different lexical forms, whether enumerated explicitly in Wikidata lexemes or deducible from other information present on those lexemes or connected items. This is particularly enforced through requiring that abstract content first be transformed into a syntactic representation before a flat textual representation may be formed, which does not leave room for components to inject entirely arbitrary free text into the output. This intends to drive the development of Wikidata's lexicographical data, as well as the knowledge graph of items to which the senses on Wikidata's lexemes link, so that improvements to the textual output of the system may primarily originate from improvements to the underlying lexical and conceptual elements.

The decisions that Ninai/Udiron makes with lexical elements, however, need not always be encoded inside Wikidata's data explicitly. The system should generally be tolerant enough in the absence of certain information to make potentially reasonable estimates about what processing needs to be done with a certain lexeme or lexical form or sense in a particular language, this by virtue of having certain implemented functionality to that effect. Though the degree to which such estimation needs to be done may vary per language or per lexical element, the addition of data to a certain lexical or conceptual element for completeness should in principle reduce the amount of necessary estimation.

2.2. Linguistic design choices

Syntactic trees are represented through tree nodes named *Catenae* (singular *Catena*)—so named as a reference to a type of flexible unit used in dependency grammars. These nodes and the grammars in which they are used are adopted mainly due to the presence of Universal Dependencies [6] trees for more than 100 languages, whose syntactic role sets could be mapped to Wikidata items and against which the structures of output texts from Ninai/Udiron could be easily compared. (For languages and constructs not adequately handled by Universal Dependencies, new roles and structural linkage strategies may be readily devised and used.) They have also been adopted, in part, due to their ability to be expressed more readily in Wikidata lexemes for lexical entities with multiple components: each link between such a lexeme and each of its components can be annotated with a single syntactic dependency.

Abstract content is represented through nested objects named *Constructors*—so named as a reference to the content units described in prior writings about Abstract Wikipedia, but consistently capitalized here due to a number of structural differences, also described in Section 4, from those previously described units. Constructors can represent any number of semantically meaningful entities, such as concepts, phrases, clauses, sentences, containers for all four of those, or simply signals for any of these. The intent is to ensure that differences in how they are handled are dealt with purely by functions at different stages in the overall rendering process, rather than prematurely restricting the potential components of an abstract content element differently.

2.3. Technical design choices

Although Ninai/Udiron is envisioned for deployment on Wikifunctions and use on Abstract Wikipedia, and although the development of Ninai/Udiron is currently taking place without both of those projects having been launched, specific considerations are being taken into account in the system's code.

Many of these considerations are intended to make the current implementation consistent with how code must ultimately run on Wikifunctions [7]. To this end the following are ensured throughout the codebase:

- All of the code is written in Python (one of the two programming languages available at Wikifunctions' launch).
- All code relies only on each other and the Python Standard Library.
- All newly defined types are immutable container types (achieved by only directly subclassing `NamedTuple`).
- All objects provided to functions are assumed to be immutable.
- All code is statically typed (achieved by annotating for compliance with the most recent version of `mypy`).

Note that functionality to retrieve/use/manipulate Wikidata lexemes and items[3] and functionality to retrieve a lexeme sense given a Wikidata item (see subsection 4.4) are currently exempt from the second point above, since the nature of network accesses to Wikidata and its Query Service on Wikifunctions has not been specified. These functionalities are also exempt from the third point above, in addition to aliases for standard library types provided in Python's `typing` module.

A few other changes are meant to align with the language-neutral nature of both Wikidata's data and Wikifunctions's code implementations:

- Wikidata item IDs are used as much as possible–whether as keys or values for mappings or as names only ever propagated as strings within the code.
- Functions' parameters and roles are separately and thoroughly documented.

For convenience when writing this implementation, identifiers for actual Python objects are exempt from the first point above. The second point above has been adopted since the names of functions must be converted to ZIDs (e.g. `Z12345`) and the names of their arguments to extensions of ZIDs (e.g. `Z12345K2` for the second argument to `Z12345`) when migrating code implementations to Wikifunctions, eliminating any readable information that might be in their names and requiring that information to be moved to separate Wikifunctions object fields.

Ninai and Udiron also aim to satisfy a number of desiderata for Abstract Wikipedia that were elaborated on soon after its development began [8]. How much the current Python implementation satisfies these is outlined below:

- *The set of constructors has to be extensible by the community*: any Constructor may be newly defined through invoking a single function (see subsubsection 4.1.1).
- *Renderers have to be written by the community*: various function types (including renderers and their components) may be written and then registered through a single decorator (see subsubsection 4.3).
- *Lexical knowledge must be easy to contribute*: achieved through contributing to Wikidata lexemes[4].
- *Graceful degradation*: The current system is a *living system*; different languages currently handle different phenomena better than others, and it is possible to specify that, in the absence of a particular function for a particular language, a function for another language may be used.

Omitted above are that *content must be easy to contribute* and that *content has to be editable in any language*: at the moment all Constructors have Python identifiers with English names for convenience and uniformity in development, but the resultant textual form of abstract content is intended to be sufficiently clear on a surface reading as well as directly invocable as Python code. An interface to automatically convert Wikifunctions ZIDs to labels in one's desired language and display them in an editor would help address both of these problems.

[3] https://gitlab.wikimedia.org/toolforge-repos/twofivesixlex/
[4] https://www.wikidata.org/wiki/Wikidata:Lexicographical_data

predicate	object
wdt:P5137	wd:Q571
wdt:P5972	wd:L619325-S1
wdt:P5973	wd:L1012261-S1
wdt:P5973	wd:L480664-S1
wdt:P6191	wd:Q104597585

(a) L1012254-S1 (the sense "book" on the Bengali "pustak").

predicate	object
wdt:P9970	wd:Q199657
wdt:P9971	wd:Q392648
wdt:P9971	wd:Q170212
wdt:P5972	wd:L1759-S1
wdt:P5972	wd:L37926-S1
wdt:P6593	wd:L261809-S1

(b) L38412-S1 (the sense "to read" on the Swedish "läsa").

Fig. 1. Predicates and objects for two lexeme senses. All objects presented are actual Wikidata entities.

predicate	object
pq:P1545	"4"
pq:P5548	wd:L620962-F3
pq:P5980	wd:L620962-S1
pq:P9764	"5"
pq:P9763	wd:Q3685154

(a) The statement (L622496 "combines" L620962.)

predicate	object
wdt:P5713	wd:Q1233197
wdt:P3831	wd:Q1094061

(b) The statement (L38440-S1 "has thematic relation" Q20820253.)

Fig. 2. RDF predicates and objects qualifying two statements. All objects presented (save for the two string values of pq:P1545) are actual Wikidata entities.

3. Preparation

Before Wikidata may be used as a source for the retrieval of words in text generation, the entities within it, particularly senses on lexemes, should have certain pieces of information in the form of statements, not just to orient them in relation to other lexeme senses and to Wikidata items representing entities, but also to control the behavior of Ninai/Udiron functions that can take advantage of such information. The suite of Wikidata properties that make such information possible to add continues to grow as more types of linguistic data become conceivable to import or necessary for the completeness of a particular set of items or lexemes; this section goes through only some of those that exist at the time of writing, including those which are actually used somewhere in the implementation at the time of writing (introduced here with italicized names).

(Throughout this section, entities are generally referred to with their identifiers as exported for Wikidata's Query Service, be these items (e.g. Q12345), properties (e.g. P1234), lexemes (e.g. L12345), forms on lexemes (e.g. L12345-F1), or senses on lexemes (e.g. L12-S2): property links representing direct statements applied to lexemes, forms, or senses are referred to with the wdt: RDF prefix[5]; properties serving as qualifiers to other statements are referred to with the pq: RDF prefix[6]; and Wikidata entity targets are referred to with the wd: RDF prefix[7].)

3.1. Common properties

A number of properties used by Ninai/Udiron were actually introduced soon after lexicographical data support was added to Wikidata, including some of the most frequently used properties across all types of lexemes, forms, and senses. Their specific potential use in text generation applications has been previously described more generally [9], but the principles behind how they are used have since evolved and been clarified in different areas.

[5]http://www.wikidata.org/prop/direct/
[6]http://www.wikidata.org/prop/qualifier/
[7]http://www.wikidata.org/entity/

- Correspondences between different lexeme senses may be marked directly with the "*translation*" (`wdt:P5972`) and "*synonym*" (`wdt:P5973`) properties, depending on whether or not the two senses are part of lexemes belonging to the same language.
- Senses that encompass other senses may be linked using "hyperonym" (`wdt:P6593`).
- The contexts in which a sense is used can be specified by one of "*language style*" (`wdt:P6191`), "*location of sense usage*" (`wdt:P6084`), and "field of usage" (`wdt:P9488`) depending on the type of context being specified.
- It may be necessary to specify on a lexeme, form, or sense that it "*requires [a specific] grammatical feature*" (`wdt:P5713`) somewhere in its context when it is used.

Examples of the values of most of these properties on two lexeme senses are given in Figure 1.

3.2. Syntactic properties

Some properties end up being particularly useful to model syntactic phenomena. They are specifically used to qualify the "*combines lexemes*" (`wdt:P5238`) property [10]:

- The position of a component relative to other components may be specified using the "*series ordinal*" (`pq:P1545`) qualifier (typically as a string representing an integer).
- The specific form and sense a component takes on in the parent lexeme may be specified using "object form" (`pq:P5548`) and "*object sense*" (`pq:P5980`).
- Due to the lack of a string-item tuple data type in Wikidata for properties, the link between a dependent and a head in a syntactic relationship is expressed with two qualifying properties:
  - "*syntactic dependency head position*" (`pq:P9764`) specifies the "series ordinal" of the head of the relationship of which a component is the dependent.
  - "*syntactic dependency head relationship*" (`pq:P9763`) specifies the type of relationship between a component and the head indicated by `pq:P9764`.

An example of the qualifiers to such a "combines" statement is shown in Figure 2a.

3.3. Preparing nominals

Certain properties are particularly relevant to noun-like lexemes (be these pronouns, nouns, or noun phrases):

- As an indirect counterpart to the "translation" and "synonym" properties, and appearing far more frequently than those, the "*item for this sense*" (`wdt:P5137`) property links senses on lexemes to a Wikidata item describing the concepts those senses represent. A lexeme sense linked via this property to "book" (`wd:Q571`) thus eliminates the need for "translation" or "synonym" statements pointing from that sense to any other lexeme sense linked via "item for this sense" to "book".
- A noun (or, more typically, a pronoun) with an inherent grammatical gender, number, or person indication may have these specified with "*grammatical gender*" (`wdt:P5185`), "grammatical number" (`wdt:P11054`) or "grammatical person" (`wdt:P11053`) as appropriate on the lexeme.
- Languages using classifiers may specify them on senses with "classifier" (`wdt:P5978`).
- Where the gender of a entity to which a sense refers differs from the lexeme's grammatical gender, the referred entity's gender may be specified with "semantic gender" (`wdt:P10339`).
- Specific places whose inhabitants are referred to by a sense may be specified with "*demonym of*" (`wdt:P6271`).

3.4. Preparing predicates

Just as certain properties are relevant to noun-like lexemes, certain properties are particularly relevant to verb-like lexemes (be these verbs, verb phrases, or other predicates):

- Just as "item for this sense" serves as an indirect counterpart to "translation" and "synonym" for nominals, "*predicate for*" (`wdt:P9970`) serves as a similar counterpart for predicate-like lexemes. A lexeme sense linked via this property to "reading" (`wd:Q199657`) thus eliminates the need for "translation" or "synonym" statements pointing from that sense to any other lexeme sense linked via "predicate for" to "reading".
- Although there are properties such as "transitivity" (`wdt:P9295`) and "valency" (`wdt:P5526`) that provide some predicate argument information, the arguments themselves may be specified with the property currently named "*has thematic relation*" (`wdt:P9971`). In particular, the specific syntactic role an argument takes may currently be specified with "*object has role*" (`wdt:P3831` and any inflections that must be applied to that argument with `wdt:P5713`. An example of the qualifiers to a `wdt:P9971` statement is shown in Figure 2b.
- Verbs inherently expressing particular aspects may specify these with "grammatical aspect" (`wdt:P7486`).
- Verbs using particular helping verbs in periphrastic constructions may specify these with "auxiliary verb" (`wdt:P5401`).

4. Architecture

Ninai/Udiron's actual software components consist of a number of core elements and several areas which are intended to be developed by the community. A diagram of these components is given in Figure 3.

4.1. Ninai

Ninai, broadly speaking, handles semantic representations as well as pragmatic and contextual decisions arising therefrom. It only deals with syntax in a surface-level fashion because of Ninai renderers' use of Udiron's syntax tree editors.

4.1.1. Constructors

The Constructor is the primary unit of abstract content definable in Ninai. It consists of an indication of the type of abstract content it represents (whether concept, phrase, clause, sentence, container, signal, or something else), a unique identifier for that Constructor usable by other Constructors, and three different sets of arguments: 1) *core* arguments, all of which are Constructors themselves, which the Constructor's abstract content type requires be present (by assigning names to them), 2) *scope*, or non-core, arguments, also all Constructors themselves, which are not specifically required by the abstract content type in question, and 3) all arguments that are not Constructors themselves.

Some examples of Constructor types provided by the current implementation include "Speaker" (yielding a nominal representing the abstract content's author, typically a first-person singular pronoun), "FutureTense" (a signal to another Constructor that something occurs after the present), "Cause" (manipulating another Constructor to indicate that it is the cause of another Constructor), and "Existence" (manipulating another Constructor to yield a clause or sentence stating that said other Constructor exists). These types, and especially the names currently given to them, are *not* by any means fixed or mandated by Ninai; they were introduced to showcase the wide variety of possible Constructors, and it is possible that a community may decide to use an entirely different set of Constructor types for abstract content.

Abstract content in Ninai is ultimately assembled through the nested composition of different types of Constructors in a particular arrangement; an example of such an arrangement of Constructors is shown in Figure 4a. How exactly a Constructor is created, and how it and its arguments are processed when rendering a Constructor, are discussed in subsection 5.1.

4.1.2. Other data types

A number of other data structures are provided by Ninai for ease of information management throughout the Constructor rendering process (for which see Section 5):

- "Contexts" are lists representing where and how deep into a tree of Constructors a particular rendering step is occurring (effectively functioning like a call stack).

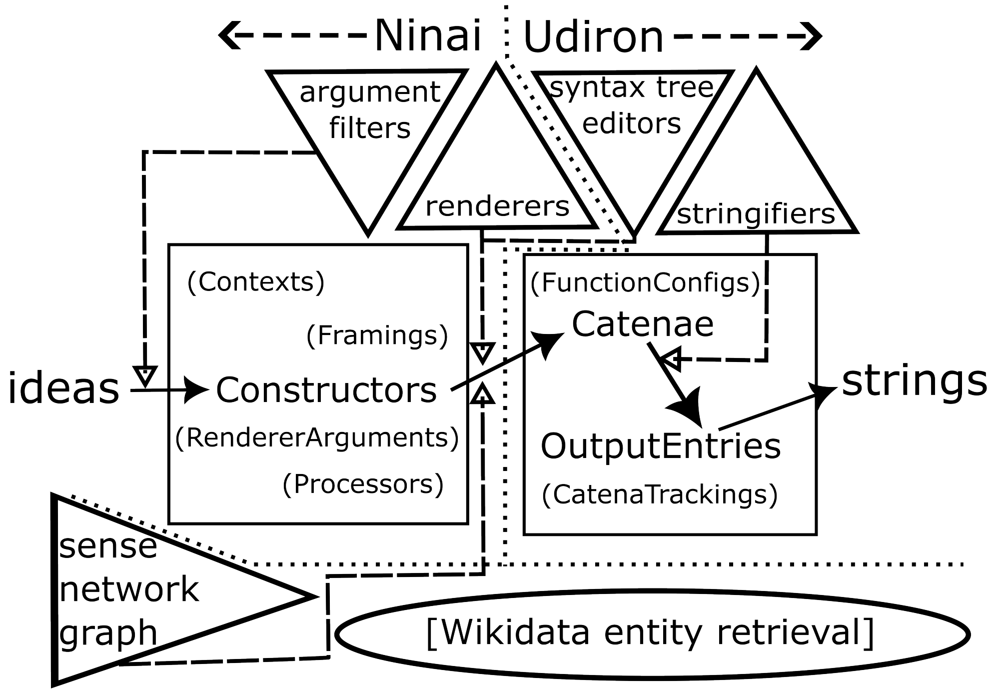


Fig. 3. Overall diagram of the components of Ninai/Udiron. The more vertical dotted line separates Ninai to the left and Udiron to the right, and the more horizontal dotted line separates components that are part of either from those that are part of neither. The squares represent the base portions of Ninai and Udiron, where the names within refer to different types defined in those portions (parenthesized types being accessories to non-parenthesized types); the triangles represent portions involving the contributions of community members (lexicographical data for the sense network graph, different types of functions for the others). The solid arrows indicate the various transformations from ideas to strings (e.g. Constructors get transformed into Catenae); the dashed arrows with open arrowheads indicate where certain functionality outside the base system comes into play within the base system (e.g. stringifiers are used in the Catena to OutputEntry transformation). Although “Wikidata entity retrieval” is not directly connected to anything, it is used in all other elements of this diagram.

- “Framings” hold information provided at the start of the Constructor rendering process to control the behavior of named individual Constructor types, named individual Constructors, individual languages, or some combination of all three.
- “RendererArguments” objects collect and separate outputs from the different argument types mentioned in the previous subsubsection so that they may be properly referred to in the main step of the Constructor rendering process.
- “Processors” are multiple function object types that perform different steps of the Constructor rendering process, that can be paused if certain required information is absent, and that can be resumed when said information is ultimately provided.

4.2. Udiron

Udiron, broadly speaking, handles syntactic manipulations as well as phonological and morphological transformations that may be needed throughout the abstract content rendering process.

4.2.1. *Catenae and OutputEntries*

The Catena is the primary unit of syntactic trees that may be generated by Udiron. It consists of a lexeme, the language to be used when stringifying the Catena (typically the same as the lexeme's language), the sense being expressed by the lexeme, a set of inflections–that is, Wikidata item IDs representing grammatical feature items– to be applied when stringifying, a configuration container, and links to both logical left- and right-hand syntactic dependents (which are pairs of Catenae and Wikidata item IDs representing syntactic relationships).

Distinct from but related to Catenae are "OutputEntries", which are the main outputs from rendering Catenae. (They are a more recent development, relative to the time of writing, prompted by an interest in maintaining some ability to examine the origins of parts of the output.). In languages that use word separators such as spaces or other characters, they represent distinct words, which may in fact represent multiple underlying lexical components (an OutputEntry for German 'im' would have two components representing 'in' and 'dem'). The components of these OutputEntries have much of the same information as Catenae, save for the links to dependencies and configuration, while the OutputEntries themselves provide, in addition to the components, the surface representation yielded by the combination of the components and a configuration container similar to that used by Catenae. An overall output string from Catena rendering is generally formed by concatenating the OutputEntries' surface representations in order.

Textual content in Udiron is ultimately assembled through the arrangement of Catenae by generating them at different stages of the Constructor rendering process and attaching them together at other stages into a syntactic tree; an example of such an arrangement of Catenae is shown in Figure 4b. How exactly a Catena tree is transformed into OutputEntries, or "stringified"[8], is discussed in subsection 5.1.

4.2.2. *Other data types*

A couple of other data structures are provided by Udiron for ease of information management throughout the Catena rendering process (for which see Section 5):

- "FunctionConfigs" hold information used to control the behavior of syntax tree editors.
- "CatenaTrackings" indicate the origins of individual Catenae (that is, the Constructors and their place in the abstract content tree that directly led to their generation).

4.3. *Community-contributed functions*

The previous two subsections discussed the base Ninai/Udiron system, but this system simply cannot operate without community contributions. The first of two major parts into which these contributions may be divided consists of writing various types of functions to be invoked by the system. Both Ninai and Udiron specify these types and their ability to be chosen and run based on different parameters, usually the language and the Constructor type being dealt with. The first two function types below may be defined for Ninai, a the second two for Udiron:

- When Constructors are created (e.g. by executing `Possession(...)`), *argument filters* defined for them are run, performing the separation of provided parameters into core and scope arguments. The results from these filters are the actual Constructor objects described in previous subsections that can be rendered into different languages.
- When a Constructor is rendered into Catenae, several types of functions are run at different stages of the process, for which see subsection 5.1. Individual Constructor types (typically more generic or container-like Constructors) may define renderers that call out to sub-renderers, which are themselves chosen and run based on the same or different parameters as those involved in selecting the original renderers.
- *Syntax tree editors* spawn and edit Catenae in various ways, potentially taking various types of parameters but still returning changed syntax trees in the process. Various sub-functions and helper functions, sometimes shared across languages, are often defined alongside these.
- Different types of *stringifiers* control the selection of forms for a particular type of lexeme and inflection set in a given language, as well as handle any outward changes that must be applied after all forms have initially been selected.

[8] A more ideal term might be "linearization", but this already has different meanings in other text generation contexts.

Except for the argument filters (which are not executed with respect to a particular output language), language fallbacks for the other function types may be defined. If, for example, a renderer for a Constructor in a particular dialect of a language doesn't exist, then the renderer for the language in general might be used; if that doesn't exist, then the one defined for 'multiple languages' ('mul', `Q20923490`) might be used instead. Constructors for signals, containers, and some other more generic types typically only have functions defined for 'multiple languages', with all other languages falling back to it.

4.4. Sense graph

The second of two major parts into which community contributions may be divided consists of improvements to Wikidata's lexicographical data, specifically with regard to the construction of a dedicated sense graph used by Ninai to retrieve specific lexemes given a particular concept. (This is treated separately here, rather than considered a part of the base Ninai/Udiron system, because appropriate graph database software that can perform the necessary searches is not currently powering the Wikidata Query Service, and it is desired that such software be used and be accessible for *all* possible natural language generation systems that could be hosted on Wikifunctions, not just this system.)

The current implementation of Ninai/Udiron assembles using NetworkX [11] a graph of relationships involving certain properties (currently "translation", "synonym", "antonym", "hyperonym", "pertainym", "item for this sense", "predicate for", and "demonym of"), retrieving these relationships using simple Wikidata queries. It then produces views of these graphs containing some of these relationships (e.g. with only predicates, or with only substantives) that may be breadth-first searched from a given origin sense or item to yield all other senses connected to that origin, no matter how far they may be.

As an example, using `Q199657` as an origin, and using the predicate-only view of the sense graph, a breadth-first search would yield not only those verb senses linked directly to that item, but also senses linked to those senses via a chain of "translation" or "synonym" statements. Similarly, using `Q571` and the substantive-only view of the sense graph would yield both direct links from noun senses to that item as well as noun senses that are another five "translation" statements away.

5. Rendering

Rendering in Ninai/Udiron consists of the entire set of transformations from a Constructor into a string. When "rendering" is spoken of unqualified (or qualified with "overall"), all parts of this process are meant. This process, however, is ultimately divided into two main parts which are important to distinguish by the types of objects involved: 1) the transformation from Ninai Constructors as an input to Udiron Catenae as an output, and 2) the transformation from Udiron Catenae as an input to Udiron OutputEntries.

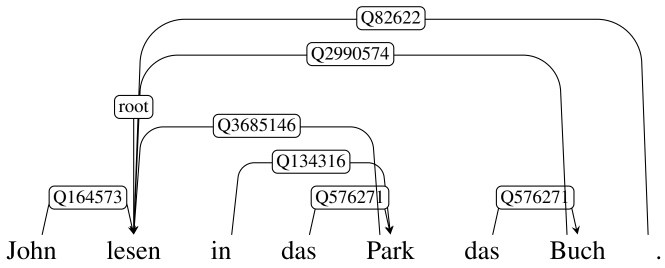
5.1. Abstract content rendering

The rendering process for a Constructor typically yields a single Catena for a syntax tree (as well as configuration information to be passed on elsewhere, if the associated Context is not empty). It can be split into five main steps, some of which may be skipped if they are not applicable to a particular Constructor, whether due to a lack of appropriate function definition or due to a lack of arguments in question:

- The *pre-hook* function (if it is defined) performs some initial modifications to the Constructor itself, returning as output a Constructor. If no modifications end up being performed (that is, if the output is the same as the input), then the step is over; otherwise, the pre-hook associated with that Constructor's type is run, and a similar check is performed on the result of that pre-hook, and so on. ("Meta-Constructors" could be set up in this way by defining appropriate pre-hooks that yield Constructors of entirely different types.)
- For each scope argument in the result from the pre-hook, this entire Constructor rendering process is run, and the output Catena and configuration from that Constructor rendering process is stored.

```
Action(
  "Q199657",
  Agent(
    Concept("Q4925477")),
  Patient(Instance(
    Concept("Q571"),
    Definite())),
  Locative(
    Instance(
      Concept("Q22698"),
      Definite())),
  PastTense()))
```

(a) Some Constructors for a sentence.



(b) The resulting syntactic dependency tree from a German rendering.

Fig. 4. The input and output from rendering a sentence about someone named John reading a particular book in a particular park in the past. Note that in Figure 4b, the Wikidata QIDs given represent the types of dependency relationships between the lexemes, the words given at the bottom are lexeme lemmata (i.e. they are *not* inflected forms), and that inflectional signals applied to each lexeme are not shown, all for brevity's sake.

- For each core argument in the result from the pre-hook, the following is run and the output Catena and configuration therefrom are stored:
  - The *argument pre-hook* (if it is defined for that argument) makes adjustments to, or extracts information from, an argument Constructor before it is rendered;
  - This entire Constructor rendering process is run for that argument; and
  - The *argument post-hook* (if it is defined for that argument) typically performs some clean-up related to the actions of the argument pre-hook.
- The *main renderer* must be defined for a particular Constructor type in a particular language before a Constructor of that type can be rendered in that language. It takes in the set of Catenae and the combination of configurations output from rendering core and scope arguments, and it produces a single Catena and potentially modified configuration. It is here that most calls to Udiron syntax tree editors originate, and thus where most Catenae are generated, combined, and modified in the Constructor rendering process.
- The post-hook (if it is defined) performs some post-processing steps to the Catena returned by the main renderer.

The Constructor in Figure 4a contains two `Instance` Constructors where, due to the core argument specification of that Constructor type, and due to the order of rendering steps, the `Definite` signals, being scope arguments, are rendered before the `Concept` core arguments. The top-level `Action` constructor, being one of the most generic Constructors, defines no core arguments and instead treats all Constructor arguments as scope arguments, examining them all at the end; those scope arguments may be re-ordered as a result. In order to distinguish objects filling certain thematic roles (e.g. the reader and what is being read), however, such flexibility in ordering is dealt with using the Constructors for the thematic relations `Agent` and `Patient`. The output from rendering Figure 4a in German is shown as a syntactic dependency graph in Figure 4b.

5.2. *Syntax tree rendering*

After obtaining a Catena from rendering a Constructor, that Catena's rendering process yields a list of OutputEntries whose surface representations, when concatenated, yield a readable output string in a natural language. The process can be split into four main steps, functions for which must be defined for each language:

- The first step, "selecting forms", consists of performing a generalized in-order traversal of the syntax tree and, for each Catena, typically checking whether the Catena's inflection set is empty *and* there is exactly one form in the Catena's lexeme that has the inflections in the inflection set. If both are the case, then nothing else

Table 1
The OutputEntries from stringifying the syntactic tree in Figure 4b; each OutputEntry consists of at least one component, specified with an appropriate sub-index relative to the OutputEntry of which it is a part

surface form	surface index	component form	component sub-index	lexeme	sense	inflections	originator
John	0	John	0	L408566	L408566-S1	Q131105	`Concept("Q4925477")`
las	1	las	0	L1759	L1759-S1	Q442485, Q1317831, Q110786, Q51929074	`Action(...)`
im	2	in	0	L6748	L6748-S2	(none)	`Locative(...)`
		dem	1	L59500	L59500-S1	Q499327, Q145599, Q110786	`Instance(...)`
Park	3	Park	0	L409006	L409006-S1	Q145599, Q110786	`Concept("Q22698")`
das	4	das	0	L59500	L59500-S1	Q1775461, Q146078, Q110786	`Instance(...)`
Buch.	5	Buch	0	L7895	L7895-S1	Q146078, Q110786	`Concept("Q571")`
		.	1	n/a	n/a	(none)	`Action(...)`

is done; otherwise, it tries to *develop* a form based on the information in the Catena and registers it in the Catena's configuration and inflection set. (This development typically involves multiple sub-functions for a particular language tailored to particular parts of speech, and may involve exploring the rest of the syntax tree if necessary.)

– The second step, "collecting forms", performs the same traversal of the syntax tree but instead for each Catena produces an OutputEntry (or OutputEntries) using just the information in that Catena–no syntax tree exploration is permissible at this stage. The decisions to be made in this step are thus likely to be less complex than those in the previous step, and if there is any information about the Catena that is not reflected in the form representation chosen and absolutely must be carried over to later steps, then it must be supplied in the OutputEntry (or OutputEntries) being produced. This includes information about surface transformations needed in the fourth step of the Catena rendering process.
– The name of the third step, "surface joining", might suggest that it "joins" the forms obtained from the previous step. Prior to the introduction of OutputEntries, this was in fact the case, as the output from this step was a single string. Since the entries now must remain distinguishable at the end of the Catena rendering process, however, the effect is now limited (for those languages with explicit word separators) to introducing word separators to the surface forms of the OutputEntries.
– The fourth step, "surface transformation", performs transformations on the list of OutputEntries output by the previous step. These transformations can originate from morphological, phonological, or simply orthographic considerations, but typically in each case lead to the merging of adjacent OutputEntries into one with multiple components. The information about transformations from the "collecting forms" step is applied here.

The result from the first two steps applied to the Catena in Figure 4b is visible as the OutputEntry components in Table 1. The sentence after the third step is initially "John las in dem Park das Buch .", but information provided alongside the Catenae for "in" and "." trigger transformations in the fourth step that contract "in" with "dem" to form "im" (whence the component with surface index "2") and remove the space between "Buch" and "." to yield "Buch." (whence the component with surface index "5"), yielding at the end "John las im Park das Buch".

6. Conclusions

Ninai and Udiron form the two main parts of a natural language generation system, highly function-based in the spirit of Wikifunctions, and highly operant with lexicographical data in the spirit of Wikidata. The system is continually being designed and revised to be general enough so that any handling of linguistic phenomena has a place in which it may reside, whether in community-edited Wikidata entities or in community-contributed functions, and to be accessible enough to end users that operations common to multiple groups of functions need to be recreated as little as possible. The system's inputs are intended to be clear enough, and its outputs similarly transparent enough, that any problems or sources of improvement within the overall rendering pipeline can be identified and addressed as clearly as possible.

In addition to general code improvements and optimizations, the future development of Ninai/Udiron could be separated into two equally productive avenues, each requiring input from different types of communities. One such avenue could consist of adding and expanding support for rendering Constructors in more typologically diverse languages, especially as their lexicographical data becomes sufficiently enriched that sentences may be reliably produced using lexemes in those languages. Another such avenue could consist of introducing and expanding more general facilities within and after the overall rendering process, in order to provide more practical and powerful capabilities for users with interests in specific facets of abstract content inputs (e.g. the pausing/resuming facilities of Ninai Processors) and of textual outputs (e.g. the use of OutputEntries). Both are likely to require flexible and performant visual (and possibly aural) user interfaces for community members to suggest, edit, and add code and data once Abstract Wikipedia and Wikifunctions eventually launch.

Acknowledgements

Special thanks to Cory Massaro, Kutz Arrieta, and Maria Keet for their feedback regarding Ninai's and Udiron's code, and to (among other active contributors to Wikidata's lexicographical data) Jan Ainali, User:Nikki, and Nicolas Vigneron for their feedback regarding some of the texts output by Ninai and Udiron.

References

[1] D. Vrandečić, Architecture for a multilingual Wikipedia, *CoRR* **abs/2004.04733** (2020). https://arxiv.org/abs/2004.04733.
[2] K. Maher, Announcing a new wiki project! Welcome, Abstract Wikipedia, 2020, [Online; accessed 21 January 2023]. https://meta.wikimedia.org/w/index.php?title=Abstract_Wikipedia/July_2020_announcement&oldid=22534460.
[3] D. Vrandečić et al., State of Wikifunctions January 2023, 2023, [Online; accessed 10 February 2023]. https://meta.wikimedia.org/w/index.php?title=Abstract_Wikipedia/Updates/2023-01-19&oldid=24432310.
[4] D. Vrandečić et al., Wikifunctions Beta, 2022, [Online; accessed 10 February 2023]. https://meta.wikimedia.org/w/index.php?title=Abstract_Wikipedia/Updates/2022-08-09&oldid=23688862.
[5] M. Morshed, Ninai and Udiron: text generation with Wikidata items and lexemes, YouTube. https://www.youtube.com/watch?v=i9kvaflh4ww&t=4127.
[6] M.-C. De Marneffe, C.D. Manning, J. Nivre and D. Zeman, Universal dependencies, *Computational linguistics* **47**(2) (2021), 255–308.
[7] D. Vrandečić et al., Requirements for code in Wikifunctions, 2022, [Online; accessed 6 February 2023]. https://meta.wikimedia.org/w/index.php?title=Abstract_Wikipedia/Updates/2022-04-28&oldid=23252697.
[8] D. Vrandečić, Building a Multilingual Wikipedia, *Commun. ACM* **64**(4) (2021), 3841–. doi:10.1145/3425778.
[9] M. Morshed, Preparing languages for natural language generation using Wikidata lexicographical data, *Septentrio Conference Series* (2021). doi:10.7557/5.5949. https://septentrio.uit.no/index.php/SCS/article/view/5949.
[10] M. Morshed, Modeling Syntactic Dependency Relationships in Wikidata Lexicographical Data., in: *Wikidata@ ISWC*, 2021.
[11] A.A. Hagberg, D.A. Schult and P.J. Swart, Exploring Network Structure, Dynamics, and Function using NetworkX, in: *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught and J. Millman, eds, Pasadena, CA USA, 2008, pp. 11–15.