

## The GiellaLT infrastructure: A multilingual infrastructure for rule-based NLP<sup>1</sup>

Sjur Nørstebø Moshagen, Flammie Pirinen, Lene Antonsen, Børre Gaup, Inga Mikkelsen, Trond Trosterud, Linda Wiechetek, Katri Hiovain-Asikainen

Department of Language and Culture  
NO-9019 UiT The Arctic University of Norway, Norway  
*sjur.n.moshagen@uit.no*

### Abstract

This article gives an overview of the GiellaLT infrastructure, the main parts of it, and how it has been and can be used to support a large number of indigenous and minority languages, from keyboards to speech technology and advanced proofing tools. A special focus is given to languages with few or non-existing digital resources, and it is shown that many tools useful to the daily digital life of language communities can be created with reasonable effort, even when you start from nothing. A time estimate is given to reach alpha, beta and final status for various tools, as a guide to interested language communities.

**Keywords:** *Infrastructure, spelling checkers, keyboards, rule-based, machine translation, finite state transducers, constraint grammar*

### 1. Introduction












You know your language, you are technically quite confident, you want to support your community by making language tools. But where do you start? How do you go about it? How do you make your tools work in Word or Google Docs? The GiellaLT infrastructure is meant to be a possible answer to these and similar questions: it is a tool chest to take you from initial steps all the way to the final products delivered on computers or mobile phones in your language community. In this respect, the GiellaLT infrastructure is world leading in its broad support for many languages and tools, and in making them possible to develop irrespective of the size of your language community.

As an example, Inari Sami with 450 speakers is one of the over 130 languages in the GiellaLT. Within the GiellaLT infrastructure Inari Sami linguists have been able to support the Inari Sami language community with high quality tools like keyboards, a spellchecker, MT, a smart dictionary and are now developing an Inari Sami grammar checker. All these tools are available for Inari Sami speakers in the app stores of various operating systems, and through the GiellaLT distribution system. In a revitalization perspective these tools are crucial, but without an infrastructure supporting and reusing resources this would have been impossible for the language community to solve on its own.








































---

<sup>1</sup> **Ref:** Moshagen, Sjur Nørstebø, Flammie Pirinen, Lene Antonsen, Børre Gaup, Inga Mikkelsen, Trond Trosterud, Linda Wiechetek, Katri Hiovain-Asikainen. 2023. The GiellaLT infrastructure: A multilingual infrastructure for rule-based NLP . In: Arvi Hurskainen, Kimmo Koskenniemi, and Tommi Pirinen (eds.), Rule-Based Language Technology. NEALT Monograph Series, 2:70-94. <https://dspace.ut.ee/handle/10062/89595>

The GiellaLT infrastructure supports development of various tools in several operating systems, as shown in Table 1 — the tools are further described later on. Infrastructure setup, existing resources and tools are available on Github and documented on <https://giellaLT.github.io>.

 = Windows,  = macOS,  = Linux,  = iOS/iPadOS,  
 = Android,  = ChromeOS,  = browser,  
 = MS Word,  = Google Docs,  = LibreOffice,  = REST/GraphQL API

systems, as shown in Table 1 — the tools are further described later on. Infrastructure setup, existing resources and tools are available on Github and documented on <https://giellaLT.github.io>.

Tools	Supported systems	Further info
<b>Keyboards</b>	     	
<b>Spelling checkers</b>	        	
<b>Hyphenators</b>		
<b>Text tokenisation and analysis</b>	   	
<b>Grammar checking</b>	   	Includes speller, LO only Linux
<b>Machine translation</b>		
<b>Text-to-speech</b>	    	planned, not ready
<b>Dictionaries</b>	   	
<b>Language learning</b>		
<b>Automatic installation and updating</b>	   	

When adding a new language to the GiellaLT infrastructure, the system builds a toy model and sets it up with all language independent build files needed in order to make the tools shown in Table 1. The main advantage of GiellaLT is thus that the quite substantial work that has gone into writing the files needed for compiling, testing and integrating the language models into the various tools. The language models differ from tool to tool in systematic ways, in some cases a descriptive model (accepting de facto usage) is needed, in other cases a normative one (accepting only forms according to the norm) is called for. Part of the language model, e.g. the treatment of Arabic numerals and non-linguistic symbols is also re-used from language to language. A large part of any practical language technology project is the integration of the tools in user programs in different operative systems. For the GiellaLT infrastructure this has been done for a wide range of cases, as shown in Table 1. This makes it possible to make language technology tools for new languages without spending several man-years on building a general infrastructure.


GiellaLT is developed by two research groups at UiT. At present, it includes support and tools for 130 different languages, most of them extremely low resource like Inari Sámi and Plains Cree. Such languages have few or no text resources, where the few that exist typically are too noisy to be able to represent a norm. These and similar languages must thus rely on mostly rule-based methods like the ones described in this article. Another point to consider is that the smaller the language community, the larger the need of tools to

support using the language in writing. Having such an infrastructure is thus of crucial importance for the future of many of the world's languages.

Not only does the GiellaLT infrastructure offer a pipeline for building tools, it also supports the process in the other end: when the language work is approaching production quality, there exists a delivery and update ecosystem, making it easy to distribute the tools to the user community. The infrastructure also contains tools to develop the linguistic data and verify its integrity and quality automatically.

What is needed to utilise GiellaLT is a linguistic description of the language and a skilled linguist, a native speaker to fill in the gaps in the description, and a programmer to help with configuring the infrastructure for a new language and support the linguist in the daily work. A language project will also benefit from activists who need the tools: they will tell what they need, test the tools, and thus ensure the linguistic quality of the output. In practice every new language added to the infrastructure for which practical tools are developed will also require adaptations and additions in GiellaLT, thereby contribute to the strength of the GiellaLT infrastructure.

Throughout this chapter of the book, we try to give an estimate of the expected amount of work needed to reach a certain maturity level for tools and resources. The information is given in side-bar info boxes, as seen below. The symbols and abbreviations used are as follows:


$\alpha$ : ¼ mm
$\beta$ : 1 mm
1.0: 3-4 mm

- $\alpha$  – **alpha** version, first useful but clearly unfinished version<sup>2</sup>
- $\beta$  – **beta** version, getting ready, but some polish still needed<sup>2</sup>
- 1.0 – **final** version, released to the language community<sup>2</sup>
- mm – **man month**, one month's work

The estimates given are based on our experience, but a number of external factors will influence the actual time a given project takes. We still believe that having an indication of expected work estimates can be useful when planning language technology projects. It should be emphasized that all estimates assumes that the GiellaLT infrastructure and support system is being used.

## 2. An overview of the infrastructure

The GiellaLT infrastructure contains all the necessary pieces to go from linguistic source code to install-ready products. Most of the steps are automated as part of a continuous integration & continuous delivery (CI/CD) system, and language-independent parts are included from or taken from various independent repositories<sup>3</sup>.

The infrastructure is in practice a way of organising all the linguistic data and scripts in a manner that is easily maintainable by humans who work on various aspects of the text and then can be systematically built into ready-to-use software products by automated tools. The way we approach this in practice changes from time to time as the software engineering ecosystem develops; however, the organisation of the system aims to keep the linguistic data more constant, as the linguistics do not change at the pace software engineering tools. The crux of the infrastructure in the large scale is though having the right

---

<sup>2</sup> A more precise definition of what these labels mean within the GiellaLT infrastructure is available at <https://giellalt.github.io/MaturityClassification.html>

<sup>3</sup> <https://github.com/divvun>

files of linguistic data in the right place. That is, standardisation of the folder and filenames and standardisation of the analysis tags are main features of the infrastructure.

The data is stored in the version control system `git`<sup>4</sup>, hosted by GitHub<sup>5</sup>. In GitHub the data is organised in repositories. Each repository is a unit of mostly self-standing tools and source code.

There are a few different kinds of linguistic repositories in our infrastructure, mainly ones for keyboard development (keyboard repositories) and ones for development of morphological dictionaries, grammars, and all other linguistic data (language repositories); these repositories are language specific, i.e., there's one repository for each language<sup>6</sup>. The different repository types and the content they contain, along with its structure, is described in the following sections.

## 2.1. Keyboard repositories

Keyboards enable us to enter text into digital devices. Without keyboards, no text. This is a very real obstacle for the majority of the languages of the world, the ones with no keyboards. It thus needs to be easy to create and maintain keyboard definitions and make them available to users.

In the GiellaLT infrastructure, keyboard definitions have their own repositories (using the repository name pattern `keyboard-*`) that contain the linguistic data defining the layout of the keyboards of the languages, and all metadata to build the final installation packages. The repositories are organised as a bundle, which is consumed by the tool `kbdgen`.<sup>7</sup> The bundle structure is as follows:

```
sma.kbdgen
├── layouts          # actual layout definitions
│   ├── sma-NO.yaml # desktop layout for Norway
│   ├── sma-SE.yaml # ditto for Sweden - they are different
│   └── sma.yaml    # mobile keyboard, identical for SE/NO
├── project.yaml    # top-level metadata
├── resources       # platform specific resources
│   └── mac
│       ├── icon.sma-NO.png
│       └── icon.sma-SE.png -> icon.sma-NO.png
└── targets         # metadata for various platforms
    ├── android.yaml
    ├── ios.yaml
    ├── mac.yaml
    └── win.yaml
```

The layout definitions are described in the next chapter.

Based on the layouts and metadata, `kbdgen` builds installers, packages, or suitable target files for the following systems: macOS, Windows, Linux (X11, IBus m17n), Android, iOS/iPadOS, and ChromeOS. For macOS and Windows, installers are readily available via Divvun's package manager *Páhkat*, further described towards the end of this chapter. For iOS and Android, layouts are included in one or both of two keyboard apps: *Divvun Keyboards*, and *Divvun Dev Keyboards*. *Divvun Dev Keyboards* functions as a testing and

---

<sup>4</sup> <https://git-scm.com>

<sup>5</sup> <https://github.com>

<sup>6</sup> languages are defined by ISO-639-3 codes, this is a requirement imposed on us by the manufacturers of operating system etc., we can only make keyboards work if they have an ISO code atm.

<sup>7</sup> <https://github.com/divvun/kbdgen>

development ground, whereas production ready keyboards go into the *Divvun Keyboards* app. All of this is done automatically or semi-automatically using CI and CD servers<sup>8</sup>.

The `kbdgen` tool also supports generating SVG files for debugging and documentation, as well as exporting the layouts as xml files suitable for upload to CLDR. And finally, it can also generate finite state error models for keyboard-based typing errors, giving suitable penalties to neighbouring letters based on the layout.

The Windows installer includes a tool to register unknown languages, so that even languages never seen on a Windows computer will be properly registered, and thus making Windows ready to support proofing tools and other language processing tools for those languages.

## 2.2. Language repositories

The language repositories contain lexical data, grammar rules, morphophonology, phonetics, etc., anything linguistic and specific to the language or even specific to the tools is built from the language data.

The language repositories, using a repository name pattern of `lang-*`, contain the whole dictionaries of the languages, laid out in a format that can be compiled into the NLP tools we provide. To achieve this, the lexical data has to be rich enough to achieve inflecting dictionaries, that is, the words have to be added some information of their inflectional patterns for example. In practice, there is an unlimited amount of information that can be recorded per dictionary word that can be interesting. So in practice, this central part of the language repository becomes like a lexical database of linguistic data. On top of that we need different kinds of rules governing morphographemics, phonology, syntax, semantics and so forth.

In practice, writing a finite state (see 2.2.1) standardised language model in `src/fst/` will provide the user with the basis for all the NLP tools we can build. To draw a parallel on how this works, if one is familiar with java programming for example, this is akin putting your maven-based project into `src/java/` or rust-based configurations in `Cargo.toml` etc. would be a software engineering interpretation of what an infrastructure is.

We also have some standards as to how to tag specific linguistic phenomena, as well as other lexical information. The linguistic software we write is in part based on that similar phenomena are marked in same manner in all languages. This ensures that components that are language-independent work the best. If specific languages deviate from some standards, it practically can mean that for those languages specific exceptions need to be written for every application. This is especially clear when working with such grammar-based machine translation, even a small mismatch in marking the same structures makes the translation fail whereas systematic use of standard annotations makes everything work automatically.


The language repositories follow a specific template, structure, and practice to make building everything easier:

```
lang-sme
├── docs
├── src
│   ├── cg3
│   ├── filters
│   └── fst
├── ...
└── tools
    ├── spell-checkers
    ├── grammarcheckers
    └── mt
```

---

<sup>8</sup> CI = continuous integration, CD = continuous delivery (Shahin et al, 2017)

### 2.2.1. Morphological analysis

  
 $\alpha$ : 1-2 mm  
 $\beta$ : 6 mm  
1.0: 12 mm

The underlying format for the linguistic models in the GiellaLT infrastructure is based on finite state morphology (FST), combining the *lexc* and *twolc* programming languages (Koskenniemi 1983, Beesley and Karttunen 2003). This is no accident: These programming languages as well as the Constraint Grammar formalism presented in the next subsection were all developed for Finnish, a language with a complex grammar and many skilled computational linguists. The contribution of the persons behind GiellaLT has been to port these compilers into open formats and set them up in an integrated infrastructure. Most GiellaLT languages are of no interest to commercial language technology companies, and the infrastructure thus contains pipelines for all aspects of language technology.

The morphology is written as sets of *entries*, where each entry contains two parts (divided by space and terminated by semicolon). The first part contains a pair of two symbol strings (to the left and the right of the colon, called the upper and lower level). The part after the space is a pointer to a *lexicon* containing a set of entries, so that the content of all entries in this lexicon is concatenated to the content of all entries pointing to it. The symbol # has special status, denoting the end of the string. In the text box to the right, the entry *kana:kana* is directed to the lexicon *n1*. This lexicon contains 3 entries, each pointing to the lexicon *clitics*. These 3 lexica will then give rise to  $3*3*4=36$  distinct forms. The upper level of the entries contains lemmas and morphosyntactic tags, whereas the lower level contains stems and affixes. It may also contain archiphonemes, such as  $\wedge A$  representing front *ä* and back *a*, and triggers for morphophonological processes. In this example  $\wedge WG$  triggers the weak grade of the consonant gradation process, a process which in this example assimilates *nt* into *nn* in certain grammatical contexts (here: genitive and inessive singular).

An excerpt from the *lexc* file for Kven nouns (the file *src/fst/stems/nouns.lexc* in the infrastructure):

```
LEXICON Nouns
kana:kana n1 "hen" ;
kynä:kynä n1 "pen" ;
hinta:hinta n1 "price" ;

LEXICON n1
+N+Sg+Nom: clitics ;
+N+Sg+Gen: $\wedge WG$ >n clitics ;
+N+Sg+Ine: $\wedge WG$ >ss $\wedge A$  clitics ;
...

LEXICON clitics
# ;
+Qst: $\%>k^O$  # ;
+Foc/han: $\%>h^An$  # ;
+Foc/ken: $\%>kin$  # ;
```

The morphographemics is taken care of in a separate finite state transducer, written in a separate language, *twolc*, in a separate file (*src/fst/phonology.twolc*):

```
Alphabet
a b c ... ä ä ö  $\%U$ :y  $\%A$ :ä  $\%O$ :ö ;

Sets
Vow = a e i o u y ä ö ä  $\%U$   $\%A$   $\%O$  ;

Rules
"vowel harmony"
 $\%A$  <=> a [a|o|u] \[ä|ö|y]*  $\%>$  (\[ä|ö|y]*  $\%>$ ) \[ä|ö|y]* ;

"Consonant Gradation nt:nn"
t:n <=> n _ Vow:  $\%WG$ : ;
```

*Twolc* defines the alphabet and sets of the model. Also, this transducer has an upper and lower level, so that the upper level of this transducer is identical to the lower level of the morphological (*lexc*) transducer. The rule format *A:B <=> L \_ R ;* denotes that there is a relation between upper level *A* and lower level *B* when occurring between the contexts *L* and *R*. The result is that non-concatenative processes such as vowel harmony and consonant gradation are done in a morphographemic transducer separate from the morphological one.

These two transducers are then compiled into one transducer, containing string pairs like for example `hinta+N+Sg+Ine:hinnassa`, where the intermediate representation `hinta^WG>ss^A` is not visible in the resulting transducer. The result is a model containing the pairs of all and only the grammatical wordforms in the language and their corresponding lemma and grammatical analyses.

The actual amount of work needed to get to a reasonable quality will vary depending on the complexity of the language, available electronic resources, existing documentation in the form of grammars and dictionaries, and experience, but based on previous projects a reasonable first version with decent coverage can be made in about six months. For good coverage, one should estimate at least a year of work.

### 2.2.2. Morphosyntactic disambiguation

Most wordforms are grammatically ambiguous, such as the English verb and noun *walks*. The correct analysis is in most cases clear from the context. The form *walks* may e.g. occur after determiners (and be a noun) or after personal pronouns (and be a verb). More complex grammars typically contain more homonymy, such as the South Saami *leah* ‘they/you are’ - has four different analyses with the same lemma, three of them finite verb analyses and one of them a non-finite analysis, the con-negative verb reading:

```
"<leah>"
  "lea" V IV Ind Prs Pl3 <W:0.0>
  "lea" V IV ConNeg <W:0.0>
  "lea" V IV Imprt Sg2 <W:0.0>
  "lea" V IV Ind Prs Sg2 <W:0.0>
```

Within the GiellaLT infrastructure, disambiguation and further analysis of text is made with constraint grammar (Karlsson 1990) compiled with the free open-source implementation VISLCG-3 (Bick 2015). The morphological output of the transducer feeds into a chain of CG-modules that do disambiguation, parsing, and dependency analysis. The analysis may be sent to applied CG modules such as e.g., grammar checking.

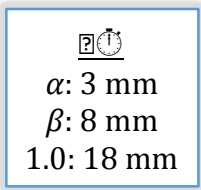
Syntactic rules of the parser disambiguate the morphological ambiguity and can add syntactic function tags. In the following sentence the context shows that *leah* should be con-negative based on the negation verb *ij* to the left.

*Im leah.*  
*NEG.1sg.prs be.con-negative*  
*"I am not./No."*

```
"<Im>"
  "ij" V IV Neg Ind Prs Sg1 <W:0.0> @+FAUXV
:
"<leah>"
  "lea" V IV ConNeg <W:0.0> @-FMAINV
  "lea" V IV ConNeg <W:0.0> @-FMAINV
;
  "lea" V IV Imprt Sg2 <W:0.0>
;
  "lea" V IV Ind Prs Pl3 <W:0.0>
;
  "lea" V IV Ind Prs Sg2 <W:0.0>
```

```
IFF ConNeg (*-1 Neg BARRIER CC OR COMMA OR ConNeg);
```

The rule that is responsible for removing all the other readings of *leah* and only picking the con-negative (ConNeg) reading is an IFF rule that refers to the lemma *lea* “be” in its Con-Neg form and a negation verb to the left, without any conjunction (CC) or comma or



another ConNeg form in between. The rule here is simplified, there are more constraints for special cases. The `IFF` operator either selects the ConNeg reading if the constraints are true, or removes it if the constraints are not true.

As the analysis is moved further and further away from the language-specific morphology, the rules become increasingly language independent. Antonsen et al (2010) have shown that whereas disambiguation should be done by language-specific grammars, closely related languages may share the same function grammars (assigning roles like *subject*, *object*). The dependency grammar was shown to be largely language independent.

### 2.3. Other repository types

The GiellaLT infrastructure contains several other repository types, although not as structured as the keyboard and language repositories. **Corpus** repositories (repository name pattern `corpus-*`) contain corpus texts, mostly in original format, with metadata and conversion instructions in an accompanying `xml` file. This is done to make it easy to rerun conversions as needed. The corpus tools and processing are further described later on.

There are a few repositories with **shared** linguistic data (repository name pattern `shared-*`). Typically, they contain proper names shared among many language repositories, definition of punctuation symbols, numbers, etc. The shared data is included in the regular language repositories by reference.

Both keyboard and language repositories are structurally set up and maintained using a templating system, for which we have two **template** repositories (repository name pattern `template-*`). Updates to the build system, basic directory structure, and other shared properties are propagated from the templates to all repositories using the tool `gut`,<sup>9</sup> and allows all supported repositories to grow in tandem when new features and technologies are introduced. This ensures relatively low-cost scaling in terms of features and abilities for each language, so that a new feature or tool with general usability easily can be introduced to all languages.

There are separate repositories for **speech** technology projects (repository name pattern `speech-*`). As speech technology is a quite recent addition to the GiellaLT, these repositories are not standardised in their structure yet, and have no templates to support setup and updates. As we gain more experience in this area, we expect to develop our support for speech technologies.

## 3. Linguistic tools and software

The main point of the *GiellaLT* infrastructure is to provide tools for language communities. In this chapter we present the tools that are most prominent and have been most used and useful for our users: keyboards, grammar and spell checking and correction, dictionaries, linguistic analysis, and machine translation between or from the minority languages. Also, language learning and speech synthesis is covered, and finally we show how to distribute the tools to the language communities and to ensure that the tools stay up to date.

We also show what constitutes the starting point for building a software tool for your language as well as the prerequisites needed for getting that far. Finally, we show some


---

<sup>9</sup> <https://github.com/divvun/gut>



ideas that are under construction or showing promising results, ideas for which we cannot yet provide an exact recipe on how to build working systems.

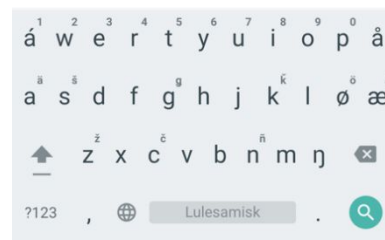
### 3.1. Keyboards

  
 $\alpha$ : 1/20 mm  
 $\beta$ : 1/4 mm  
 1.0: 1 mm

To be able to type and write a language, you need a keyboard. Using the tool `kbdgen`, one can easily specify a keyboard layout in a YAML file, mimicking the actual layout of the keyboard. The listing below shows the definition of the Android mobile keyboard layout for Lule Sámi. The `kbdgen` tool takes this definition and a bit of metadata, combines it with code for an Android keyboard app, compiles everything, signs the built artefact and uploads it to the Google Play Store, ready for testing.

```
modes:
  android:
    default: |
      á w e r t y u i o p å
      a s d f g h j k l ø æ
      z x c v b n m η
```

⇒ `kbdgen` ⇒



Additional metadata is for example language name in the native language, names for some of the keys, icons for easy recognition of the active keyboard, and references to speller files, if available.


The YAML files can contain definitions for dead key sequences for desktop keyboards, as well as long-press popup definitions for touch-screen keyboards. The newest version of the `kbdgen` tool supports various physical layouts for desktop keyboards, to allow for non-ISO keyboard definitions. Further details for the layout specification can be found in the `kbdgen` documentation<sup>10</sup>.

The overall goal is to make it as easy as possible for linguists to write a keyboard definition and get it into the hands of the language community. A first draft can be created in less than a day, and a good keyboard layout for most operating systems will take about a week to develop, with a couple of weeks more for testing, feedback, and adjustments.

The keyboard infra in GiellaLT works well for any alphabetic and syllabary-based writing system, essentially everything except iconographic and similar systems.

Because of technical limitations by Apple and Google, it is not possible to create keyboard definitions for external, physical keyboards for Android tablets and iPads. Our on-screen keyboards work as they should even when a physical keyboard is attached to such tablets.

### 3.2. Spell-checking and correction

  
 $\alpha$ : 1-3 mm  
 $\beta$ : 6 mm  
 1.0: 12 mm

In GiellaLT, spell-checking and correction are built on a language model (in the form of a finite-state transducer) for the language in question. A spell-checker is a mechanism that recognises wordforms that do not belong to a dictionary of all acceptable wordforms in the language and tries to suggest the most likely wordforms for the unknown ones. The

<sup>10</sup> <https://divvun.github.io/kbdgen>

GiellaLT spellcheckers differ from this approach in not containing a list of wordforms, but rather a list of stems with a pointer to concatenative morphology (see also the section 2.2.2 on language model repositories for specifics on what this looks like and is built) as well as a separate morphophonological component. The resulting language model should then recognise all and only the correct wordforms of the language in question. Since the language model is dynamic it is also able to recognise wordforms resulting from productive but unlexicalized morphological processes, such as dynamic compounding or derivation. In languages like German or Finnish, one can freely combine for example words like *kissa* ‘cat’, *bussi* ‘bus’ and *kauppias* ‘salesman’ into *kissabussikauppias* that will be acceptable for language users and thus should be supported by the spell-checker. The main challenge when building a spell-checker and corrector is to have a good coverage in the lemma list, and in our experience, several months of work in dictionary building or around 10,000 well-selected words will suffice for a good entry-level spell-checker. Since the FST also will be used for analysing texts (chapter 2.2), it will be necessary to compile a normative version of the FST, which excludes non-normative forms. They are excluded by means of tags.

The mechanism for correcting wrongly spelled words into correct ones is called *error correction*. The most basic error correction model is based upon the so-called Levenshtein distance (Levenshtein 1965): For an unknown wordform, suggest known wordforms resulting from one of the following operations: delete character, add character, exchange character with another one or swap the order of two characters. The benefit of this baseline model is language independent, so we can use it for all languages as a starting point.

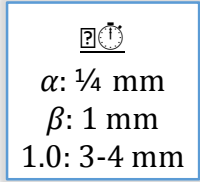
Spell-checking and correction can be improved on a per language basis drawing from the knowledge of the language and its users. One of the most basic ways of improving the quality of spelling corrections, is improving the error modelling. If we know what kind of errors users make and why, we can ensure that the relevant corrections are suggested more commonly. Segment length, especially consonant length, may be hard to perceive and hence likely to be misspelled. Doubling and omitting identical letters are thus built into most error models. The same are parallel diphthongs such as North Saami *uo/oa*, *ie/ea*, where L2 writers may be likely to pick the wrong one. By making the diphthong pairs part of the error model the errors can easily be corrected. The position in the word may also be relevant: For a language where final stop devoicing is not shown in the written language, exchanging *b*, *d*, *g* with *p*, *t*, *k* may be given a penalty number lower than the default value.

When the dictionary reaches sufficiently high coverage the problems caused by suggesting relatively rare words become more apparent, one way of dealing with this problem is codifying the rare words on the lexical level. If there are corpora of correctly written texts available, it is also possible to use statistical approaches to make sure that very rare words are not suggested as corrections unless we are sure they are the most likely ones.

It is also possible to list common misspellings of whole words. The South Saami error model contains the pair *uvre:eevre* (for *eevre* "just, precisely"), where word forms like *muvre* and *duvre* otherwise would have had a shorter Levenshtein distance.

Seen from a minority language perspective, the main point is that the GiellaLT infrastructure offers a ready-made way of building not only language models but also speller error models and a possibility to integrate them into a wide range of word processors. And having a mechanism for automatically separating normative from descriptive forms means that the same source code and language model can be used and reused in many different contexts.

### 3.3. Automatic hyphenation

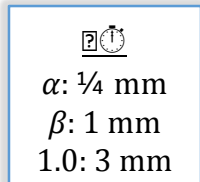


The rules for proper hyphenation vary from language to language, but usually it is based on the syllable structure of the words. Depending on the language, morphology may also play a role, especially word boundaries in languages with compounding.

The GiellaLT infrastructure supports hyphenation using several mechanisms. The core component is an FST rule component defining the syllable structure. Exceptions can be specified in the lexicon, and both lexicalised and dynamic compounds will be used to override the rule-based hyphenation. The result is high-quality hyphenation, but it requires good coverage by the lexicon. The lexical component is based on the morphological analyser described in earlier sections.

Given that the morphological analyser is already done, adding hyphenation rules does not take a lot of work. The most time-consuming work is testing and ensuring that the result is as it should be. Getting or building hyphenated test data can be time consuming.

### 3.4. Dictionaries



The GiellaLT infrastructure includes a setup for combining dictionaries and language models. Most GiellaLT languages possess a rich morphology, with tens or even hundreds of inflected forms for each lemma, often including complex stem-alternation processes, prefixing or dynamic compounding. Looking up unknown words may be a tedious endeavour, since few of the instances of a lemma in running text may be the lemma form itself.

The GiellaLT dictionaries combine the dictionary with an FST-based lookup model that finds the lemma form, sends it to the dictionary and presents the translation to the user. This may be done via a click-in-text-function, as in *Figure 1*.

*Figure 1: The dictionary set-up. The word, which is clicked for in this image, is both inflected and a compound. The analyser finds the base form for both parts, gives them to the dictionary, and the translation is then sent to the user.*



The FST may then also be used to generate paradigms of different sizes to the user, as well as facilitate example extraction from corpora. The tags in the analysis can also be used as triggers for additional information about the word, e.g. derivation, which can be linked to information in an online-grammar (see Johnson et al 2013 for a presentation).

Dictionary source files are written in a simple XML format. If one has access to a bilingual machine-readable dictionary and the lemmas in the dictionary have the same form as the lemmas in the FST, the dictionary may easily be turned into a morphology-enriched dictionary in the GiellaLT infrastructure. Less structured dictionaries will require far more

work. Homonymous lemmas with different inflection should be distinguished by getting different tags, both in *lexc* and in the XML schema. In this way we can ensure that the words are presented with the correct inflectional paradigm to the dictionary user.

### 3.5. Grammar-checking and correction



$\alpha$ : 3 mm  
 $\beta$ : 8 mm  
1.0: 12 mm

*Everybody \*make errors* – both grammatical errors and typos. Expectations are higher as to what grammar checkers should do compared to a spellchecker, even if we do not think about it consciously. Whereas to find that *\*teh* is a typo for *the* we only need to check the word itself, to find that *\*their* is a typo for *there* we need to actually look at the whole sentence. So even for typos we need a more powerful tool that understands grammar.

These rather trivial errors make up a big part in English grammar checking (e.g., the *Grammarly* program<sup>11</sup>). Most minority languages, especially the circumpolar ones in the GiellaLT infrastructure, are morphologically much more complex. They are rich in inflections and derivations, with complex wordforms that bear a lot of potential for errors. When wordforms are pronounced close to each other or endings are omitted in speech they are also frequently misspelled in writing. Typical errors are those that concern agreement between subject and verb or determiner and noun (cf. the North Sámi example), as well as case marking errors in different parts of the sentence.

*Mii sámit maid \*áigot gullot. > áigut*  
*We Sámi.Nom.Pl want.3Pl listen > want.1Pl*  
*We Sámi also (they) want to listen > we want to*

When writing a rule-based grammar checker we first identify the morphological and syntactic structure of a sentence similar to parsing. Each word is associated with a lemma and a number of morphological tags. If the word is homonymous either in its form or already based on different lexemes (like *address* – 1. noun 2. verb infinitive 3. finite verb) it is associated with more than one possible analysis. Disambiguation is not the same as in parsing as the goal of it is another one. Instead of aiming for one remaining analysis, we only want to remove as much ambiguity as is necessary to find a potential error. But since we expect the sentence to contain errors, therefore being a bit more unreliable, we are a bit more relaxed on disambiguation. At the same time, we do need reliable information as to what the context of our error is. In case of *address*, we would need to identify it as a noun before looking for an agreement error with a subsequent finite verb.

The analysis must be robust enough to recognise the grammatical pattern in question even when the grammar is wrong. Constraint grammar differs from other rule-based grammar formalisms in being bottom-up, taking the words and their morphological analysis as a starting point, removing irrelevant analyses, and adding grammatical functions. For robust rules we make use of all the potential of rule-based methods. We have access to semantic information (e.g. human, time, vehicle), valency information (case and semantic role of the arguments of a verb), pronunciation related traits which makes the form bound to certain misspellings. We can also add dependencies and map semantic roles to their respective verbs in order to find valency-based case or pre-/postposition errors.

---

<sup>11</sup> <https://grammarly.com>

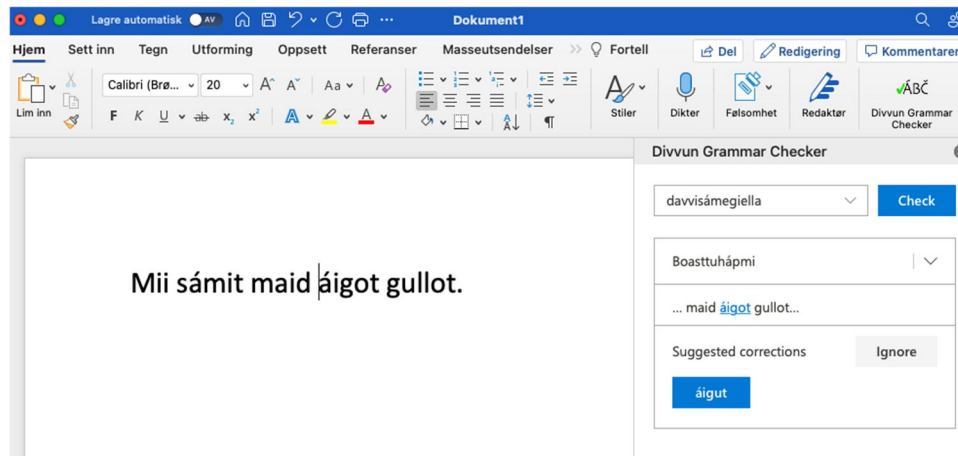


Figure 2 North Sámi Divvun grammar checker in MS Word in the right sidebar instead of the blue underline

GiellaLT's first grammar checker was made for North Sámi and work started in 2011. (Wiechetek 2012; Wiechetek 2017). Since 2019 grammar checkers are supported by GiellaLT for any language. Its setup includes various modules to ensure availability of the required information to find a grammatical error and generate suggestions and user feedback.

As Microsoft Word unfortunately does not open for integrating third-party solutions for low-resource languages, we are forced to using its web-based plugin interface instead of the usual blue line below the text. The grammar checker interface, detected errors and suggested corrections are presented in a sidebar to the right of the text, as seen in the screen shot of a version of the North Sámi grammar checker in *Figure 2*.

When making a grammar checker from scratch without any previous study of which grammatical errors are frequent for either L1 or L2 users, building a grammar checker becomes a study of errors at the same time as the tool is made. A well-functioning grammar checker requires hand-written rules that are validated by a set of example sentences from the corpus (newspaper texts, fiction, etc.) so exceptions to a certain grammatical construction are well covered. This set of examples should be included in a daily testing routine to see if rules break when they are modified and to follow development of precision and recall. One should start with a set of at least 50 examples including both positive and negative examples of a certain error so that both precision and recall can be tested.

Error detection and correction rules that add error tags to a specific token and exchange incorrect lemmata or morphological tag combinations with correct ones. The most complex of these rules include very specific context conditions and numerous negative conditions for exceptions of the rules. The rule in the following example detects an agreement error in a word form that is expected to be third person plural and fails to be so.

The expectation is based on a preceding pronominal subject in third person plural or a nominal subject in nominative plural. \*-1 specifies its position to the left. The BARRIER operator restricts the distance to the subjects by specifying word (forms) that may or may not appear between the subject and its verb. In this case only adverbs or particles may appear between them. The target of error detection is a verb in present or past tense.

The exceptions are specified in separate condition specifications afterwards. Some regard the target and exclude common homonyms like illative case forms or idiosyncratic adverbs, common spelling mistakes of nouns (that then get a verbal analysis), homonymous non-finite verbforms. The rule also excludes 1st person plural present tense forms except for those ending in *-at* (here specified by a regex), and infinitives that are preceded by

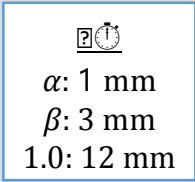
lemmata with infinitive valency tags. The exceptions specify also coordinated noun-phrases, dual forms with coordinated human subjects or coordinations that involve first or second person pronouns, to name some of them.

```
ADD (&syn-number_congruence-subj-verb) TARGET (V Ind Prs) - ConNeg OR (V Ind Prt)
  IF
    (*-1 (Pron Sem/Hum Pers Pl3 Nom) OR (N Pl Nom) BARRIER NOT-ADV-PCLE)
    (NEGATE 0 (Sg Ill))
    (NEGATE 0 N + Err/Orth-any)
    (NEGATE 0 Prs Pl1) - ("<.*at>r")
    (NEGATE 0 Inf LINK -1 <TH-Inf> OR <*Inf>)
    (NEGATE 0 Du3 LINK *1 Sem/Human BARRIER NOT-ADV-PCLE LINK 1 CC LINK 1 Sem/Human)
    (NEGATE 0 Pl1 LINK -1 Sem/Human + (Nom Pl) LINK -1 CC LINK -1 ("mun" Nom));
```

ADD rules cooccur with COPY rules which replace the incorrect tags (in this case any person number that is not Pl3) with the correct one, i.e., Pl3 with the error tag added by the accompanying ADD rule.

```
COPY (Pl3 &SUGGEST) EXCEPT Sg1 OR Sg2 OR Sg3 OR Du1 OR Du2 OR Du3 OR Pl1 OR Pl2
  TARGET (V Ind Prs &syn-number_congruence-subj-verb) ;
```

### 3.6. Machine translation



Machine translation (MT) in GiellaLT is handled by making our morphological analysers, and syntactic parsers available to the Apertium MT infrastructure (see Khanna et al 2021). This is basically performed by a series of conversions in order to convert the output of the GiellaLT language models to adhere to Apertium conventions as well as by adding some new components. The basic building block is the morphological and syntactic analyser for the source language (see section 2.2.2) and a bilingual dictionary. This analyser must have a good coverage, which also includes non-normative forms. If the source language and the target language are not closely related, the syntactic tags in the analysis of the source language are very useful.

The output in the target language is generated by a morphological analyser which at least covers the lemmas in the dictionary. In addition, one needs a grammatical description and phrases where the grammars of source and target languages do not meet. This grammar handles all mismatches between source and target language, whatever they may be: dropping of pronouns, introduction of articles, gendering of pronouns, idioms and MWE's, etc. We do the bilingual lexicography like this:

```
<e><p><l>bivdi<s n="n"/></l><r>jeger<s n="n"/><s n="m"/></r></p></e>
<e><p><l>bivdi<s n="n"/></l><r>fisker<s n="n"/><s n="m"/></r></p></e>
<e><p><l>bivdin<s n="n"/></l><r>jakt<s n="n"/><s n="f"/></r></p></e>
```

Inside the *e* (entry) and *p* (pair) node there are two parts, *l* (left) and *r* (right), with the two languages as node content. Each word is followed by a set of *s* nodes, containing grammatical information (here: POS (*n* = noun) and gender (*m*, *f*, *nt* = masculine, feminine and neuter). This is a basic XML format used with Apertium. The logic is simple, and it benefits from the tags being systematic. If the dictionary contains two or more word pairs with identic lemma on the left side, one must consider which word pair to choose, or one can make lexical selection rules, based on context in the sentence. These rules make a lexical selection module, which can be made as XML, based on position and lemma and tags in the context, or can be made with CG rules, see 3.2.2.

The system supports multi-words in both directions and idioms. The primary goal is to get a good coverage of singleton words. Syntactic transfer is done as follows:

```
S -> VP NP { 1 _  
  * (maybe_adp) [case=2.case]  
  * (maybe_art) [number=2.number,  
  case=2.case,gender=2.gender  
  ,def=ind] 2 } ;  
V -> %vblex {1[person =  
(if (1.tense = imp) "" else 1.person),  
  number = (if (1.number = du)  
  pl else 1.number)] } ;
```

In this simplified example from a real-world grammar for syntactic transfer from North-Saami to a Germanic language we show that the machine translation syntax is quite like the notation conventions from e.g., the Standard Theory (Chomsky 1965). Rules may thus operate on either word or phrase level. Here we handle distribution of case, number and gender for the generated adpositions and articles, which are based on the case and the position in the sentence, and translating the singular-dual-plural system of North Sámi into the Germanic singular-plural system (Pirinen and Wiecheteck 2022). After chunking words together in the first two modules, the following modules change the word order, to the extent it is necessary.

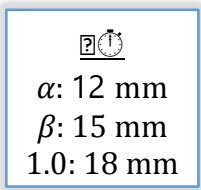
In our experience the systems start to be usable for understanding the idea of the text in translation when containing 5,000 well-selected translation pairs, if it is possible also to transfer compounding and/or derivations from source language to target language. This makes a few months of work. If the language pair requires large re-ordering of the grammar, it will demand more work.

Although there is a request for MT programs from the majority language to the minority language, we have chosen to make systems the other way, from the minority language to the majority language. These systems give the freedom to write in the minority language. A possible bad quality of the output in a majority language has no bad impact on the language itself. We have also built MT systems for translating between minority languages, which are related to each other, both to avoid using the majority language as a lingua franca, and to make it possible to use the bigger minority language as a pivot language for text production (see Antonsen et al 2017).

### 3.7. Text-to-speech

Developing TTS for an indigenous language with few resources available can be challenging. Resources such as grammars, language learning books or phonetic descriptions are important in designing the project, building, and checking the corpora and evaluating the TTS output phonetically. With using only a phonetic/phonological description and utilizing the knowledge of native speakers of a language, a simple and "old-fashioned" but still usable TTS application, such as the Espeak formant synthesis (Kastrati et al., 2014; Pronk et al., 2013) can be built from scratch. The downside of this is that while it might be a working speech synthesizer, the sound quality is very machine-like and the users' expectations for the quality of a TTS system are very high due to the realistic and very human-like examples from well-resourced languages such as English.

We are now working on a modern, open-source TTS system that could be openly available for anyone who wants to develop speech technology for any low-resource language. The system will make all language-independent parts integrated into the larger GiellaLT infrastructure, ensuring that maintenance and updates are done regularly. When finished, it will also ensure that all voices will be available on all supported platforms, and that new platforms will be available to all existing voices. It will utilise the rule-based text-



processing technologies already present in the GiellaLT infrastructure, using machine learning technologies only for the soundwave synthesis step.

The development of a TTS system requires multidisciplinary input from fields like natural language processing (NLP), phonetics and phonology, machine learning (ML) and digital signal processing (DSP). Tasks connected to NLP are important in developing the text front-end for the TTS – these are, for example, automatically converting numbers and abbreviations to full words in a correct way. Phonetics and phonology are essential in corpus design, writing text-to-IPA rules and evaluating the TTS output. Importantly, by using phonetic annotations of the texts, it is possible to address phenomena that are not visible in the orthographic texts. The importance of ML is growing in the field of speech technology as neural networks are used to model the acoustic features of human speech, allowing for realistic and natural-sounding TTS. Procedures related to DSP are important in (pre)processing the audio data: these include filtering, resampling, and normalizing the corpus for suitable audio quality.

The first step in developing a TTS system is designing and building a text corpus, considering the requirements of the speech technology. In the next section, we describe our working process in developing a TTS voice for the Lule Sámi language.

### **3.7.1. Building and processing a speech corpus**

Building a speech corpus starts by collecting a suitable multi-domain text corpus which corresponds to at least 10 hours of recorded read speech, that has been shown to be enough to achieve an end-user suitable TTS system for North Sámi (Makashova, 2021). Before the speech recordings could be done, a suitable text corpus had to be collected. In the Lule Sámi project we reused part of a Lule Sámi gold corpus developed in 2013 within the GiellaLT community and collected additional texts of various text styles we knew to be well written. The resulting Lule Sámi text corpus for TTS consists of text styles such as news, educational, parliament etc. with altogether over 74,000 words. We also checked that our corpus covers all important phonological contrasts, sound combinations, and consonant gradation patterns according to a grammatical description (Spiik, 1989), and in the case of missing or very scarce gradation patterns, we added additional sentences to cover for these and to balance the occurrences of various patterns. All texts will be open source, as well as the recordings and the rest of the resources needed for the synthesis.

When using machine-learning methods to build up a speech model for TTS, the quality of the recordings must be excellent, i.e., room reverberation or background noise must be avoided in the recordings, because the noise would be modelled as well and audible in the TTS output. Thus, the recordings must be done in a sound-treated room with professional microphones and recording set-up. The minimal requirement for the audio recording is so-called CD quality (44.1 kHz sample, 16-bit). In spring 2022, we have recorded our first TTS corpus of 10 hours with a male speech talent. Ultimately, our plan is to build both male and female voices (each representing different areal/dialectal varieties of Lule Sámi) and thus altogether 20 hours of speech is going to be recorded, processed, and made publicly available for future projects and for the language community to use.

After recording the speech corpus, a lot of effort must be put into cleaning the audio from unnecessary noise (coughing, clearing one's throat, background sounds if any) and editing the original corpus texts so that they correspond to the audio as accurately as possible. Even mistakes, repetitions and hesitations are transcribed if they are not completely cut out from the audio. In our case, we did not cut these because it would have led to great loss of data. Subsequently, after editing the texts the audio and corresponding texts are automatically time-aligned (i.e., force-aligned) using an online tool called WebMAUS



Basic, maintained by the University of Munich<sup>12</sup>. By using this tool together with manual inspection and correction afterwards, finding the word and sentence boundaries are streamlined for effective processing of the data: in this way, the audio can be split into sentence-long files very quickly, as most of the modern machine-learning based TTS frameworks require the training data to be in this format.

### 3.7.2. Experiments with different frameworks

As described in Makashova (2021), a North Sámi TTS voice was trained with a female voice, with a data set consisting of 3500 training sentences, corresponding to approximately 3 hours of speech. The TTS model consisted of four components: Tacotron, Forward-Tacotron, Tacotron2 and WaveGlow, the two latter ones from the official Nvidia repository<sup>13</sup>. The training of this successful and good quality Tacotron model and the WaveGlow model took one month, and for the ForwardTacotron three days, on a single graphical processing unit (GPU). Access to high-performance computing and storage services can significantly reduce the training time needed for the models.

To reduce the environmental costs of model training, one could consider adapting existing speech models by training the models further with additional data and pre-trained models from a “neighbouring” language. This so-called transfer learning (Tu et al., 2019; Debnath et al., 2020) allows for utilising smaller data sets for training, making it possible, for example, to use the North Sámi TTS model as the starting point for the Lule Sámi TTS.

We actually tried this on a TTS model using transfer learning between North and Lule Sámi. With a miniature data set (approx. one hour of speech data recorded with a cell phone), we were able to train a Lule Sámi voice, but the quality of the output showed that this corpus did not cover all necessary phonemes of the language and thus there were phonological inaccuracies. Moreover, the North and Lule Sámi orthographies are somewhat different: for example, the alveolar fricative sound written in English as *sh*, is written as <š> in North Sámi, and as <sj> in Lule Sámi. This creates unnecessary differences in the training data. By converting both North and Lule Sámi texts to an approximate IPA (International Phonetic Alphabet) transcription, these differences could be resolved, and the transfer learning would presumably be more successful.

## 3.8. Corpus analysis and processing

The GiellaLT infrastructure contains text corpora as well as a set of tools for treating them. Given its rule-based approach, GiellaLT mainly use corpora for testing of tools, as well as for linguistic research.

### 3.8.1. Corpus infrastructure and tools

The basic principle is to store all incoming material in a triple set of files: The original file (untouched), an xsl file containing metadata (including information on languages found in the document as well as special conversion challenges) and a derived XML file containing the metadata and the content of the original file divided into paragraphs annotated for language. The process deriving the XML file can be repeated as the metadata is improved. There are also tools for extracting different parts out of the corpus (only this or that

---

<sup>12</sup> <https://clarin.phonetik.uni-muenchen.de/BASWebServices/interface/WebMAUSBasic>

<sup>13</sup> <https://github.com/NVIDIA>

language, only list items, etc.). The metadata files also keep track of parallel language versions of a given file. Corpus collection, annotation and conversion is done by CorpusTools, a software package we have made in order to add and administrate corpus content, create and keep track of metadata files and convert files for internal and external use. The tools also keep track of translated files.

The corpora are then morphologically annotated and given syntactic functions and for some languages also dependency relations by the FST and CG tools presented above, with a very high degree of coverage. The corpora are presented for linguists and dictionary users via the corpus interface Korp<sup>14</sup>. At present, the GiellaLT instances of Korp present annotated corpora for 18 different languages, ranging from 300,000 to 65 million tokens. Parallel text is presented as such in Korp and also available for download as translation memories.

### 3.8.2. Annotated corpora for special purposes

To aid the development of spelling and grammar checkers, we have developed a language independent markup of errors<sup>15</sup>. We then manually mark errors in parts of our corpus files, and use CorpusTools to extract error/correction pairs to test our tools. Here is an examples from error markup of Lule Sámi:

*{Valla aj la ájnas}¥{wo|Valla l aj ájnas} dáđjadit dajt hásstalusájt {mij}£{congr.pron|ma} varresvuodan ja máhtsastimen {lulu}£{congr.cond|lulun} gá ulmusj la sábme, tjielggi Trond Bliksvær.*

*Rahpamin lij aj {áttják}§{vowlat,á-a|áttjak} álggám áđá sámeássjij stáhttatjálle Ragnhild Vassvik Kalstad.*

In 2013 we made a Lule Sámi gold corpus with this markup. The purpose was to test the existing Lule Sámi spellchecker developed in 2007 against this gold corpus. In order to be able to use the gold corpus to test the spellchecker, we collected texts that had not been proofread before and with a range of different authors. We aimed for a gold corpus with at least 1,000 typos (non-words) in a corpus of at least 20,000 words of continuous text. Over a period of 9 months one Lule Sámi linguist used 390 hours to mark and correct a gold corpus of 29,527 words with 1,505 non-word errors and additional 1,322 morpho-syntactic, syntactic and lexical errors. Even though the purpose in 2013 was to test the Lule Sámi spellchecker, it was important to mark all types of errors so that we could reuse the gold corpus. We are now developing a Lule Sámi grammarchecker and have been testing the grammarchecker on that same gold corpus (Mikkelsen et al. 2022). During the linguistic work with building a Lule Sámi gold corpus, it was the markup format that was most challenging. It is important that the markup is consistent, the same type of error should always be marked in the same way. In addition, the marking of errors should follow the same pattern for all languages.

### 3.9. Computer-assisted language-learning tools

Computer-assisted language learning (CALL) is typically used towards vocabulary learning. Programs are enriched with various techniques for remembering new words, ranging



$\alpha$ : 1 mm  
 $\beta$ : 3 mm  
1.0: 6 mm

<sup>14</sup> SIKOR. UiT Norgga árktalaš universitehta ja Norgga Sámedikki sámi teakstačoakkáldat, Veršuvdna 01.12.2021, URL: <http://gtweb.uit.no/korp/>. Også lenkje til Göteborg-korp + referanse til standardartikkel om Korp (?)

<sup>15</sup> <https://giellalt.uit.no/proof/spelling/testdoc/error-markup.html>

from encouraging association rules to following the learning process on a word-for-word basis.

Learners of morphology-rich languages face more fundamental challenges. In addition to learning the words, the pupils must also learn both their inflection patterns and in what context each inflection form should be used. Within GiellaLT, the *Oahpa learning platform* provides just that. It offers automatically generated exercises for practicing morphology, with or without context. We have also been experimenting with a system-governed dialogue mode, where the system asks questions based upon input from the pupil and analyses and comments the answers provided by the pupil. The morphological exercises are created by means of finite state transducers, and the dialogue games are created by means of constraint grammar (see Antonsen et al 2009a, 2009b for a presentation).

A further extension to this was the *Konteksta* program, based on the WERTi system (Meurers et al 2010). The idea is to offer language teachers a possibility to give the pupils authentic online texts and have the system creating exercises based upon an analysis of the text in question. The exercise may be to find e.g. the finite verbs in a text, and then to do multiple choice tests or type in the wordform, for these finite verbs. The philosophy is partly that the pupils may be more motivated when the texts are interesting to them and partly that the teachers may want to use the program if it really is able to offer the teacher a time-saving way of making grammatical exercises. When adapting the program to North Sámi, we had to find solutions for challenges in authentic texts, typical for a minority language: variations in orthography, and large proportion of non-normative forms (Antonsen and Argese 2018).

It is necessary to involve language teachers in the design of CALL programs. The amount of work depends entirely on the design of the program.

### 3.10. Software management, distribution, and updates

Experience has shown that users have various problems installing and making use of the tools offered by the GiellaLT infrastructure. In addition, when a tool is first installed, it is rarely updated, leading to people being stuck with old versions and non-working software.

To avoid this, we have developed a package manager named *Páhkat*<sup>16</sup> for the tools offered by the GiellaLT infrastructure. Users download this manager, install the tools for the language(s) they want, and from there on, the package manager makes sure all dependencies are installed, and that all software is kept up to date. There are stand-alone app front ends for Windows and macOS, whereas client libraries are built into mobile apps. This means that the keyboard apps automatically keep the spellers up to date for users.

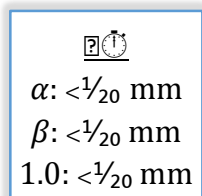
We use GitHub and continuous integration and deployment to ensure that the linguistic resources at the centre of our tools such as spelling checkers are always up to date. The system uses centralised build instructions in combination with Taskcluster<sup>17</sup>, and stores the generated artifacts in a separate repository accessible by the *Páhkat* clients. All artifacts are signed with developer certificates, and all communication is encrypted to ensure the integrity of the artifacts and the package manager system.

The work involved for a single language is minimal, a few initial configurations, and tagged commits to get a new speller or keyboard out the door. Other tools still need to be included in the automatic distribution, but as the system is developed more tools will be added, and for each tool it will apply for all languages at once.

---

<sup>16</sup> <https://github.com/divvun?q=pahkat>

<sup>17</sup> <https://taskcluster.net>



## 4. The future

The GiellaLT infrastructure is constantly being developed and enhanced. In this section we will look into possible future paths for the infrastructure. It is split in three parts: first, we will look at further development of rule-based tools, then have a peek at possible hybrid systems, and finally what other technology development one could imagine.

### 4.1. Rule based development

No work has been done related to **indexing and searching**. While there has not been enough texts to warrant a major effort in this direction, the North Sámi text production within some institutions is growing all the time, thus the need will soon be there. The existing, rule-based tools are already fit for stemming and lemmatisation, and should be easy to integrate with existing systems, but the work needs to be done. There also need to be a fall-back system for unknown words.

Another area where not much work has been done is **localisation**. While we have worked quite a lot on machine translation (see above 3.6), and we have done some work on translation memory and general translation support, no effort has been made to set up a system for streamlining software localisation. As the digital world becomes ubiquitous, there will be an increased need for this type of service, especially for the education system.

A third area for future development is to **expand the writing support tool set**. Adding synonym dictionaries, making more advanced use of our sentence parsing tools to offer simple rewrites of sentences (cf e.g. Piotrowski and Mahlow 2009), or to help writers choose more native constructions as opposed to using sentence structures borrowed from the majority language. There are many possibilities, and we have just started.

One popular use of language technology is for **named entity recognition (NER)** (see Ruokolainen 2020, Luoma et al 2020). Although nothing has been done explicitly to deliver such a tool, it would be very easy to make one. All morphological analyses include explicit tagging of names, and it is thus trivial to turn that into a basic NER tool. A bit of work to cover unknown names should be added, but also that should not be too hard, and can probably be done in a largely language independent manner.

### 4.2. Hybrid systems

As described earlier, hybrid systems combine machine learning technologies with rule-based systems. In this section we look at automatic speech recognition, dialog systems and text summarisation.

#### 4.2.1. Automatic speech recognition (ASR)

In addition to TTS, we are working towards developing a tool for automatic speech recognition (ASR) for Sámi. This section describes materials and experiments only for North Sámi, but in the future, we hope to expand our work to Lule Sámi ASR as well. In Makashova (2021), TTS and ASR models were trained simultaneously in a dual transformation loop, using the same read speech data set, corresponding to only six hours of speech from two speakers, three hours each. The ASR model in this work was based on the Way2Vec2

model which is a part of the HuggingFace<sup>18</sup> library. The model was trained for 30 000 steps, reaching a WER (Word-Error-Rate) of 41% and 0.5 loss. The most common error types in the ASR predictions seem to be in word boundaries (\*earáláhkai – eará láhkái) and in lengths of some sounds (\*rinškit – rinškkít). However, these kinds of errors would be easy to correct using Divvun’s spell checking software (described above in Section 3.2).

One of the most important differences between training the TTS and ASR models would be that for TTS, the training material needs to be very clean in terms of sound quality and there needs to be as many recordings from a single speaker as possible. For ASR, on the other hand, the recorded materials can be of poorer sound quality and preferably from multiple speakers and from different areal varieties of a language as long as there are good and accurate transcriptions of the speech.

State-of-the-art ASR frameworks normally require up to 10,000 hours of multi-speaker data for training reliable and universal models that can generalise to any unseen speaker (see, e. g., Hannun et al., 2014). As collecting these amounts of data from small minority languages is not a realistic goal, alternatives such as utilising existing archive materials can be considered for developing speech technology for Sámi. These are provided by, e.g., *The language bank of Finland* and *The language bank of Norway*. These archive materials contain spontaneous, transcribed spoken materials from various dialects and dozens of North Sámi speakers. Although extremely valuable, archive data may not be enough. Thus online data sourcing campaigns, such as the ongoing *Lahjoita puhetta*<sup>19</sup> (“Donate your speech”) project for developing ASR for Finnish, should be considered.

#### 4.2.2. Dialogue systems

The resulting TTS and ASR models under development can be used in building more advanced speech technology frameworks, such as dialogue systems (see, e.g., Jokinen et al. (2017; Wilcock et al. (2017; Trong et al. (2019)) and various kinds of mobile applications for, e.g., language learning. Dialog systems can also be used as front ends for public sector services, and in parts of the health services.

As the society becomes more and more digitised, it is more and more crucial that also minority languages can be used in all digital contexts. Eventually it should be possible to get a North Sámi spoken dialog plugin for your electric car, or a South Sámi instance of digital assistants in mobile phones.

#### 4.2.3. Text prediction

People on mobile platforms nowadays expect their keyboards to offer suggestions on how to complete the word they are writing, as well as also suggestions for the next word. This is available for a number of majority languages. We have experimented a bit with this, but nothing has entered production quality. One challenge is the scarcity of relevant text corpora, and further work needs to investigate what possibilities there are to alleviate this shortcoming using various hybrid approaches.

---

<sup>18</sup> <https://huggingface.co>

<sup>19</sup> <https://lahjoitapuhetta.fi>

### 4.3. Other technology development

The software technology develops much faster than linguistics and lexicography, this means that in comparison to the linguistic resources that are long-term projects for decades and centuries, the usage of those resources in computational linguistics change in shape and form every few years. A system made for usage of our infrastructure and resource ten years ago is probably outdated for most of the contemporary programmers.

There exists a number of various NLP platforms for various tasks, such as UralicNLP<sup>20</sup> (Hämäläinen 2019) and NLTK<sup>21</sup> (Bird et al 2009). It would be very useful to make the GiellaLT platform and tools available as free-standing components that can be easily integrated with other tools such as the two mentioned, or preferably with premade integration, so that the GiellaLT components can be fetched and used just like any other component, independent of technology. Contrary to these platforms, GiellaLT puts an emphasis on the language models, what is missing is routines for integrating the language models into modular systems.

For users it is important that the technology is available where the users are. That means a never-ending story of making sure language technology tools get access to the platforms users are using, adding new platforms and systems as needed. It also means we need to work with technology producers to make them give access to their platforms as needed for our users.

## 5. Conclusion

The GiellaLT infrastructure and most resources are available under open licences in GitHub, free for anyone to use. Setting up all the surrounding infrastructure components, CI/CD, build servers, distribution systems and so on is a non-trivial task, and we encourage interested communities to use the existing GiellaLT infrastructure. Using it, one can build a complete NLP system for a new language in a matter of months and have keyboards, spelling checkers and other applications derived from it. This can be achieved virtually with no pre-existing digital resources. Combined with a technology that fits very well for morphologically or phonologically complex languages, it ensures that tools for the modern, digital society can be developed for any language.

GiellaLT has permanent funding from a Norwegian ministry due to the needs of the Sámi people of Norway, and in combination with open-source code everywhere possible, this guarantees that the infrastructure and its resources will continue to live on and be maintained independent of other changes in the world.

New technologies will be added as needed, as shown above with speech technology in a hybrid setup. Whatever the direction of the technological development, it is our goal to ensure that the GiellaLT infrastructure stays updated and is developed in concert with the needs of the society, combining existing and new technologies for new purposes.

---

<sup>20</sup> <https://uralicnlp.com>

<sup>21</sup> <https://www.nltk.org>

## References

- Antonsen, Lene and Chiara Argese, 2018:  
Using authentic texts for grammar exercises for a minority language. In *Proceedings of the 7th workshop on NLP for Computer Assisted Language Learning (NLP4CALL 2018)*. Linköping Electronic Conference Proceedings 152, p. 1–9.
- Antonsen, Lene, Ciprian Gerstenberger, Maja Kappfjell, Sandra Nystø Rahka, Marja-Liisa Oltuis, Trond Trosterud and Francis M. Tyers, 2017:  
Machine translation with North Saami as a pivot language. In *Proceedings of the 21st Nordic Conference on Computational Linguistics*, NoDaLiDa, 22–24 May 2017, Gothenburg, Sweden. Number 131 in Linköping Electronic Conference Proceedings. Linköping University Electronic Press, Linköpings universitet. p. 123–131.
- Antonsen, Lene, Saara Huhmarniemi, Trond Trosterud, 2009a:  
Interactive pedagogical programs based on constraint grammar. In *Proceedings of the 17th Nordic Conference of Computational Linguistics*. Nealt Proceedings Series 2009S. Volume 8.
- Antonsen, Lene, Saara Huhmarniemi, Trond Trosterud, 2009b:  
*Constraint Grammar in Dialogue Systems*. NEALT Proceedings Series 2009. Volum 8. s. 13–21.
- Antonsen, Lene, Linda Wiechetek and Trond Trosterud, 2010:  
Reusing Grammatical Resources for New Languages. In *Proceedings of the International conference on Language Resources and Evaluation LREC 2010*. Stroudsburg: The Association for Computational Linguistics. p. 2782–2789.
- Beesley, Kenneth. R., and Lauri Karttunen, 2003:  
*Finite-state morphology: Xerox tools and techniques*. CSLI, Stanford.
- Bick, Eckhard and Tino Didriksen, 2015:  
Cg-3—beyond classical constraint grammar. In *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)* (pp. 31-39).
- Bird, S., Klein, E., & Loper, E., 2009:  
*Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, Inc.
- Chomsky, Noam, 1965:  
*Aspects of the theory of syntax*. M.I.T. Press.
- Cooper, E., 2019:  
*Text-to-speech synthesis using found data for low-resource languages*. Columbia University.
- Debnath, A., Patil, S. S., Nadiger, G., and Ganesan, R. A., 2020:  
Low-resource end-to-end Sanskrit TTS using Tacotron2, WaveGlow and transfer learning. In *2020 IEEE 17th India Council International Conference (INDICON)*, pages 1–5. IEEE.
- Jokinen, Kristiina, Katri Hiovain, Niclas Laxström, Ilona Rauhala, and G. Wilcock, 2017:  
Digisami and digital natives: Interaction technology for the North Sami language. In *Dialogues with social robots*, pages 3–19. Springer.
- Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., et al., 2014:  
*Deep speech: Scaling up end-to-end speech recognition*. ArXiv preprint arXiv:1412.5567.
- Hämäläinen, M., 2019:  
UralicNLP: An NLP Library for Uralic Languages. In *Journal of open source software*, 4(37), [1345]
- Khanna, T., J.N. Washington, F.M. Tyers, S. Bayatl, D. Swanson, T.A. Pirinen, I. Tang og H. Aløs, 2021:  
Recent advances in Apertium, a free/open-source rule-based machine translation platform for low-resource languages. In *Machine Translation* 35, 475–502.  
<https://doi.org/10.1007/s10590-021-09260-6>

- Johnson, Ryan, Lene Antonsen, Trond Trosterud, 2013: Using finite state transducers for making efficient reading comprehension dictionaries. *Proceedings of the 19th Nordic Conference of Computational Linguistics* (NoDaLiDa 2013), May 22–24, 2013, Oslo University, Norway. *NEALT Proceedings Series* 16: 59–71.
- Karlsson, F., 1990:  
Constraint grammar as a framework for parsing running text. In *COLING 1990 Volume 3: Papers presented to the 13th International Conference on Computational Linguistics*.
- Kastrati, R., Hamiti, M., and Abazi, L., 2014:  
The opportunity of using *espeak* as text-to-speech synthesizer for Albanian language. In *Proceedings of the 15th International Conference on Computer Systems and Technologies*, pages 179–186.
- Koskeniemi, Kimmo, 1983:  
*Two-level Morphology. A General Computational Model for Word-Form Recognition and Production*. Department of General Linguistics. University of Helsinki.
- Tanmai Khanna, Jonathan N. Washington, Francis M. Tyers, Sevilay Bayatlı, Daniel G. Swanson, Tommi A. Pirinen, Irene Tang & Hector Alòs i Font, 2021:  
Recent advances in Apertium, a free/open-source rule-based machine translation platform for low-resource languages. In *Machine Translation* (2021).  
<https://doi.org/10.1007/s10590-021-09260-6>
- Levenshtein, Vladimir I. 1965 = В. И. Левенштейн, 1965:  
Двоичные коды с исправлением выпадений, вставок и замещений символов. *Доклады Академии Наук СССР*. 163 (4): 845–848. In English: "Binary codes capable of correcting deletions, insertions, and re-versals". *Soviet Physics Doklady*. 10 (8): 707–710.
- Luoma, Jouni, Miika Oinonen, Maria Pyykönen, Veronika Laippala, Sampo Pyysalo, 2020:  
A Broad-coverage Corpus for Finnish Named Entity Recognition. In *Proceedings of The 12th Language Resources and Evaluation Conference (LREC'2020)*
- Makashova, L., 2021:  
*Speech synthesis and recognition for a low-resource language: Connecting TTS and ASR for mutual benefit*. Master's thesis, University of Gothenburg.
- Meurers Detmar, Ramon Ziai, Luiz Amaral, Adriane Boyd, Aleksandar Dimitros, Vanessa Metcalf, and Niels Ott, 2011.  
Enhancing authentic web pages for language learners. In *Proceedings for the 5th Workshop on Innovative Use of NLP for Building Educational Applications (BEA-5) at NAACL-HLT 2010*, pages 10–18, Los Angeles.
- Mikkelsen, I., Wiechetek, L., Pirinen, F., 2022:  
Reusing a Multi-lingual Setup to Bootstrap a Grammar Checker for a Very Low Resource Language without Data. In *Proceedings of the Fifth Workshop on the Use of Computational Methods in the Study of Endangered Languages*, pages 149–158, Dublin, Ireland. Association for Computational Linguistics.
- Piotrowski, Michael and Cerstin Mahlow, 2009:  
Linguistic editing support. In *Proceedings of the 9th ACM symposium on Document engineering (DocEng '09)*. Association for Computing Machinery, New York, NY, USA, 214–217. <https://doi.org/10.1145/1600193.1600240>
- Pirinen, Flammie and Linda Wiechetek, 2022:  
Building an Extremely Low Resource Language to High Resource Language Machine Translation System from Scratch. In *Proceedings of the 18th Conference on Natural Language Processing (KONVENS 2022)*, pages 150–155, Potsdam, Germany. KONVENS 2022 Organizers.
- Pronk, R., Intelligentie, B. O. K., and Weenink, D. D., 2013:  
*Adding Japanese language synthesis support to the Espeak system*. University of Amsterdam.
- Ruokolainen, T., Kauppinen, P., Silfverberg, M. *et al.* A Finnish news corpus for named entity recognition. *Lang Resources & Evaluation* **54**, 247–272 (2020).  
<https://doi.org/10.1007/s10579-019-09471-7>



- Săracu, G. and Stan, A., 2021:  
An analysis of the data efficiency in Tacotron2 speech synthesis system. In *2021 International Conference on Speech Technology and Human-Computer Dialogue (SpeD)*, pages 172–176. IEEE.
- Shahin, Mojtaba, Muhammad Ali Babar, and Liming Zhu, 2017:  
Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. In *IEEE Access* 5 (2017): 3909-3943.
- Spiik, N. E., 1989:  
*Lulesamisk grammatik*. Sameskolstyrelsen.
- Trong, T. N., Jokinen, K., and Hautamäki, V., 2019:  
Enabling spoken dialogue systems for low-resourced languages—end-to-end dialect recognition for North Sami. In *9th International Workshop on Spoken Dialogue System Technology*, pages 221–235. Springer.
- Tu, T., Chen, Y.-J., Yeh, C.-c., and Lee, H.-Y. 2019:  
*End-to-end text-to-speech for low-resource languages by cross-lingual transfer learning*. ArXiv preprint arXiv:1904.06508.
- Wiechetek, Linda, 2012:  
Constraint grammar based correction of grammatical errors for North Sámi. *SaLTMil 8 – AfLaT2012*, pages 35–40.
- Wiechetek, Linda, 2017:  
*When grammar can't be trusted - Valency and semantic categories in North Sámi syntactic analysis and error detection*. PhD dissertation, UiT The Arctic university of Norway.
- Wilcock, G., Laxström, N., Leinonen, J., Smit, P., Kurimo, M., and Jokinen, K. 2017:  
Towards Samitalk: a Sami-speaking robot linked to Sami wikipedia. In *Dialogues with Social Robots*, pages 343–351. Springer