Sustainable LT resources

How to build language technology resources for the next 100 years

by Sjur Nørstebø Moshagen

Introduction

An overview of the whole presentation

- why this talk - the overall goal of our LT work
- what resources do we care about - and why
- how to future-proof those resources (as far as possible)
- closing remarks

Vision

The vision for the work on lesser-resourced languages of which LT development is but a small part, can be captured in these points:

- linguistic diversity
- Language survival
- the right of each individual to their own (ie their parents' language(s))
- Possibly Language Revival

The Language Technology part of this vision is:

- To make sure small and resource-limited languages are as usable in our digital world as the top 10 languages.

Real-life limitations on working towards the vision

- limited resources
- small communities
- we can't assume language-technology knowledge (or any advanced computer knowledge)
- some services and products might be too resource-intensive to be achievable with the avilable resources
    - *... with the technology we have today*
    - *that is, with the present technology our vision won't be fully implementable for many languages*
    - *we have to rely on future technological development to come to our resque*

Approaches to work around the real-life limitations

Thus, we want to:

- minimise redoing earlier work
- reduce the number of different resources to the bare minimum, and then prioritise them
- maximise reuse
- build user and developer communities
- share resources as much as possible
- try to prepare as much as possible for future technological development

These are all good intentions - for the rest of the talk I will try to substantiate them, based on the experience we have built in the Divvun and Giellatekno teams at the University of Tromsø.

List of topics to be covered

I will substantiate the list on the previous slide by covering the following topics:

- What do we mean by LT resources?
- rules vs statistics / heuristics
- infrastructure sharing
- code-sharing
- user and developer community
- reuse
- standards

LT resources

The language technology resources can be grossly split in two:

- low-level resources:
  - *keyboards, charsets, fonts*
- linguistic resources:
  - *corpora, grammars, dictionaries*

Each of them deserve their own discussion

Low-level LT resources

To be able to use a language on a computing device, we need to have at least the following in place:

- computer-internal symbol representation - character encoding
- text input - typically keyboards
- text output/display - display/output technology (fonts, speech synthesis, etc)

Without these, no computational operations are possible on language in the form of text, and no interaction between humans and computing devices is possible.

Character endcoding

Up until the 90's character encoding was a traumatic experience for minority language speakers, and a major barrier to entry into efficient computing. But then came:

- Unicode = character encoding chaos resolved

Unicode is now established as *the* character encoding standard, in its different transformation formats.

Since around 2000 the main OS's have supported Unicode, and they are also regularly updated to include the latest Unicode standard.

Unicode also has a standardised procedure for how to add more characters when needed, and it is being updated continually.

This means that given reasonable documenation, any missing symbol from any language we may encounter in the future can and will be added, and as such there are no major future risks with the machine-level representation of language symbols.

Text input

The situation here is very different from that of character encoding:

- new computing paradigms may require new input methods (cf smart phones, tablet computers)
- it is guaranteed that new computing paradigms will enter the scene over the course of the next 100 years
- evidence so far indicates that each new computing paradigm requires lobbying for the inclusion of anything but a couple of tens of western and extremely large languages (e.g. software keyboard on tablets, speech/handwriting recognition)

Input method technologies

The different input methods can use vastly different technology:

- keyboards (software or hardware)
- handwriting recognition
- speech recognition

The resource needs are quite different for different input methods.

Presently keyboard input (by hardware or software keyboards) is the dominating technology, and it is in itself quite straightforward.

Since the keyboard technology is pretty straightforward and well known and tested over many years, the only reasonable explanation for the lack of keyboard support for many languages must be something else.

Text input hierarchy

Exemplified by iOS support:

- languages with Siri voice recognition: 4 (en (*3), fr, de, ja)
- languages with simpler voice recognition: 17 (da, en*3, fi, fr*2, el, it, ja, zh*3, ko, nl, no, pl, po*2, ru, es*2, sv, de)
- languages with keyboard: 44
  - *some languages have several keyboards*
  - *not all languages have speller support*
- all other languages: 6 875 (6 909 - 44)

Text input hierarchy (cont.)

Or to put it in other terms:

- languages with Siri voice recognition: biggest European languages + Japanese
- languages with simpler voice recognition: mostly West European languages + Japanese, Chinese, Korean
- languages with keyboard: most (?) remaining European state languages + some former European colonies + Cherokee
- all other languages/regions

The missing languages in iOS

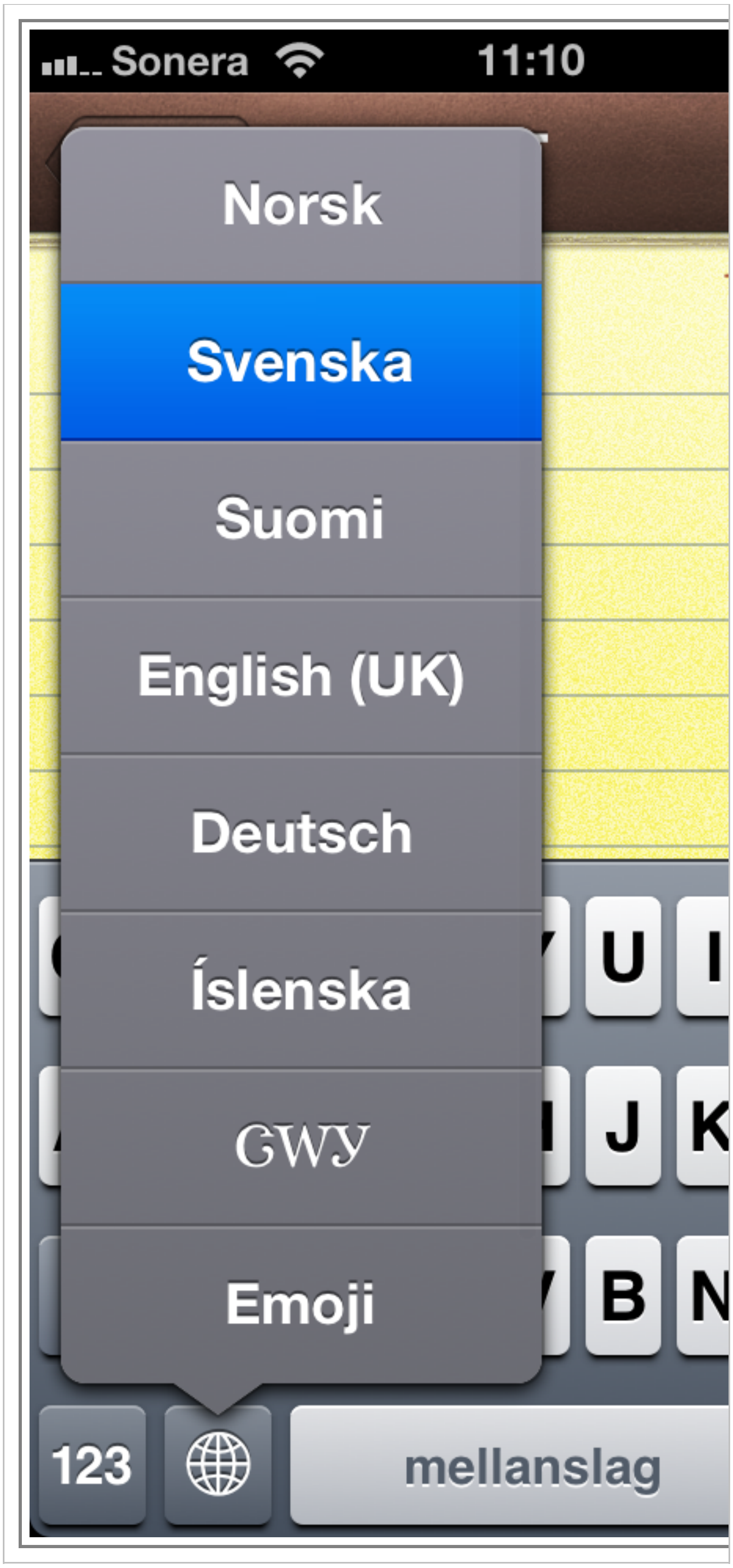
- Africa is represented by *one* keyboard (Arabic)
    - *Africa south of Sahara doesn't have a single keyboard layout in iOS!*
- South and South-East Asia is also completely absent
- there are no minority language keyboards except Cherokee
- A number of state languages with distinct alphabets or scripts have no keyboard (e.g. Bangla)

This is a known story from other platforms - history repeats itself.

But the strange thing is that there could have been more languages with very little effort from Apple - it doesn't have to be like this:

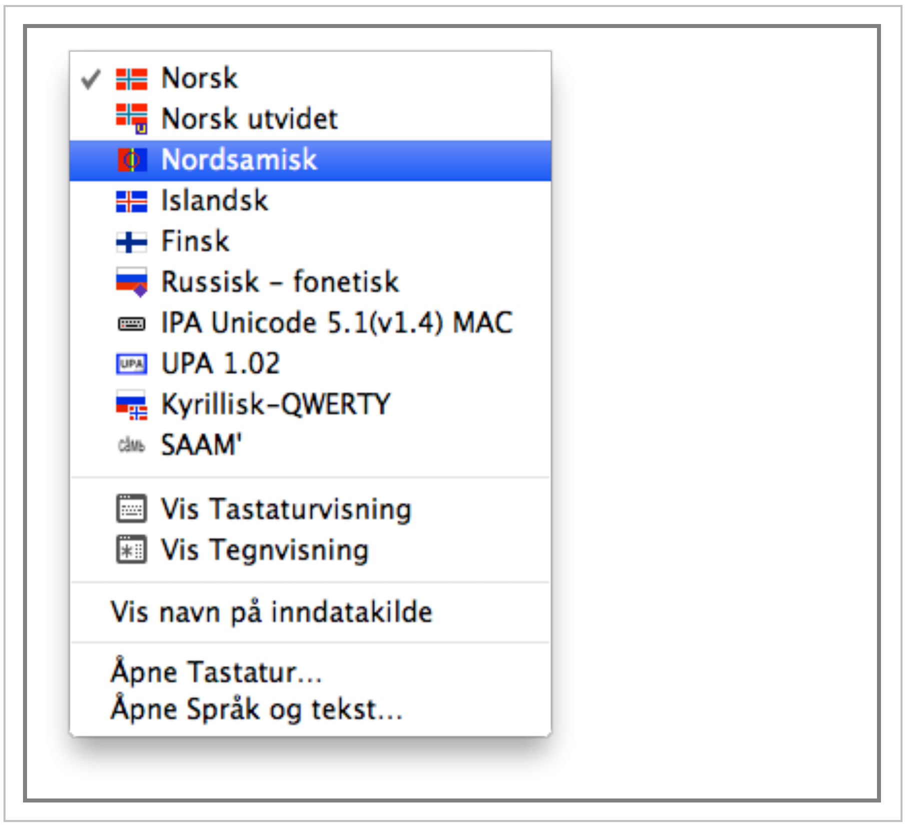
The missing languages in iOS (cont.)

Where is my Sámi keyboard?


Sonera
11:10
Norsk
Svenska
Suomi
English (UK)
Deutsch
Íslenska
ᏣᎳᎩ
Emoji
123
mellanslag

The missing languages in iOS (cont.)

Here is my Sámi keyboard:


✓ Norsk
Norsk utvidet
Nordsamisk
Islandsk
Finsk
Russisk – fonetisk
IPA Unicode 5.1(v1.4) MAC
UPA 1.02
Kyrillisk-QWERTY
SAAM'
Vis Tastaturvisning
Vis Tegnvisning
Vis navn på inndatakilde
Åpne Tastatur...
Åpne Språk og tekst...

Text input hierarchy on other platforms

The situation is slightly better on other platforms:

- Latest Debian has keyboard layouts for ~85 locales
- Latest MacOSX has ~150 keyboard layouts
    - *some languages have multiple layouts*
- in some cases several languages can share one layout

But the pattern still remains:

- there is a huge divide between a handful languages (speaker independent, continuous speech recognition) and most of the languages of the world (no input method at all)
- then there are some languages somewhere in between
- as we saw above, even for the languages that are "in" to some degree, some fights must be fought over and over again it seems

Future proofing keyboard input

Since keyboard layouts are so simple to make (it is all software), a large number of languages are served by community initiatives and more or less open source solutions.

This does solve the immediate problem for each language, but is not a very good solution in the longer run:

- there is no systematic quality assurance of the keyboard layouts, and thus the quality varies
- people must install and configure their keyboards separately, instead of it being included, tested and ready for use when they turn on the computing device the first time
- there is no pressure to make sure existing keyboards for one platform is included when new computing platforms enter the market

I have no ready receipt for future-proofing keyboard input.

But making keyboard layouts for as many languages as possible available on open-source platforms will at least put some pressure on closed-source vendors.

Other input technologies: recognition

- handwriting recognition
- speech recognition

Even though the basic technology is language independent, automatic recognition means a lot of work for each language.

Handwriting recognition:

- not widespread, not required by any widespread computing platform
- thus no motivation to improve it, or expand the language coverage
- unless there comes a new computing device on which hand-writing recognition is essential, nothing will happen with the support for most languages of the world

Speech recognition

- has been a niche technology until know
- will the new smart-phone use of the technology give it a breakthrough?
- very resource-demanding with today's technology, both for development and during use

The present technologies for working speech recognition are closed-source

It also needs more resources than are typically available for the languages we work on

My personal hope goes to future development of a rule and grammar based speech recognition system

Cf what has happened in the MT field, with Grammarsoft producing grammar-based translations of the same quality as the best statistical translators:

Rule-based MT example output

Original Danish:

"NATO’s generalsekretær Anders Fogh Rasmussen understreger efter et stormøde med samtlige ISAF-partnere, at den nye operation, som endnu er navnløs, ikke bare vil være en fortsættelse af den hidtidige operation:"

Google Translate:

"NATO Secretary General Anders Fogh Rasmussen stresses after a public meeting with all ISAF partners that the new operation, which is still nameless, will not only be a continuation of the previous operation:"

Grammarsoft translation:

"NATO's secretary general Anders Fogh Rasmussen emphasizes after a grand meeting with all ISAF partners that the new operation, which is still nameless, not just will be a continuation of the previous [[operation]]:"

That is: rule and grammar-based MT is fully possible, and thus within reach of lesser-resourced languages, for which statistical approaches would be completely useless.

Unless the same happens with speech recognition, it will not be available to these language communities.

Text output

- Printing and displaying text is *basically* working
- There are Unicode fonts with pretty good coverage, even open-source

But there are some issues:

- automatic placement of combining diacritics is not working reliably
- most fonts don't cover the less frequent letters

Both of these issues are hitting hard on the lesser-resourced languages (and is at the same time completely invisible to the biggest languages' users)

- e.g. Kildin Sámi and many West African Languages must rely on automatic placement of diacritics
  - *but this doesn't work in all applications or all OS's - you can't rely on it*
- no *new* precomposed characters are allowed in Unicode
  - *... and since all the big languages are covered, this will only be an issue for the late newcomers, ie all small or economically uninteresting language communities*

I see no easy way out of this - text rendering as we know it now isn't very future proof for many of our languages

Text rendering compared with Unicode

- The good thing about Unicode was that it turned out as an all-or-nothing option
- If an OS maker added support for Unicode, **all** languages were automatically included
- There was no option "take the rich and big ones, and leave the rest out"
- But this is what has happened in the text rendering department.
- In addition, the different operating systems have partly implemented different technologies
- There are also third-party initiatives like the SIL Graphite technology with the aim of solving these problem
- But as long as there is no automatic, all-encompasing solution based on open standards, some languages will always be left in the cold

Other text output methods

There are other ways of outputting texts:

- braille
- speech synthesis

Speech synthesis is — in contrast to speech recognition — not that demanding, and can be built using rule and grammar-based solutions to a large extent.

The main hindrance is not one of access to large corpora, but rather lack of detailed phonology grammars and phonetic descrioptions, and lack of money - it is quite labor-intensive.

But given the knowledge, expertise and work force, speech synthesis should be within reach for most language communities

Recent and future development will likely also help in reducing the amount of work required to produce new syntheses.

Low-level LT Resources summary

- character encoding is solved or solvable for any future needs
- text input is solved in one respect (the technology is there), but at the same time surprisingly unsolved in another respect (a very large number of written languages do not have OS support)
- text output is not working well for many of the less-resourced languages, and I see no easy way of solving that for the future

Rule-Based Technology

I have already hinted at my view that the future is rule-based, at least for the languages we are discussing here.

In the following discussion, I will just assume that the main bulk of the code is rule-based:

- lexicon
- morphology
- (morpho)phonology
- disambiguation
- syntax
- etc.

In the 100 year perspective in this presentation there are a couple of reasons for this:

Resource-Limited Languages

- For these languages statistical methods are just not feasable or workable - there just is not enough text material to produce useful models
- On the other hand - what is always available as long as there are speakers is inherent grammatical and lexical knowledge. Rule-based methods basically tries to make that embodied knowledge explicit, and codify it according to whatever formalism is in use.
- Even though the methods and technologies might change (and for sure will change in 100 years), the value of an explicitly encoded grammar, with a full and productive lexicon, is indispensible.

Statistics - the black box

Statistics vs rules - summary

- Even though you can inspect a statistical model to a certain degree, it doesn't tell you anything about the language.
- The generated rules are random
    - *some might happen to cover true generalisations over the language*
    - *… but some might just as well be accidental and invalid generalisations*
- … and you can't know which is which without yourself having that knowledge.
- Which leads us back to the need for humans with the language competence.

Methods will evolve

- For sure a lot will happen in a hundred years, both with rulebased and more heuristic approaches to LT
- i am not dismissing the idea that statistical and heuristic methods can't be useful in the future even to languages with very limited resources
- my point is merely that by starting out rule-based, you will have a solid foundation for the future, whatever the future is
- starting out with heuristics or statistics is much more like gambling, which we can't afford in the languages we are working with

Grammar and rule based formalisms and technologies are the only viable solution in the long run for lesser-resourced languages.

With that discussion asside, let's continue with other topics

Infrastructure Sharing

- The biggest cost aside from the pure linguistic development is infrastructure development and maintenance.
- In the long run the only sensible solution is to build and share one or a few infrastructure(s) for many languages
- this is presently done in e.g. Apertium and in Giellatekno/Divvun
- These two probably aren't the last word said on scalable infrastructure for language technology for multiple languages
- But they are nevertheless real attempts at building a working infrastructure and a good starting point for evaluation and discussion

Benefits

- "write once, run everywhere" - heard it before?
  - *but really - the goal is to have an infrastructure that will scale gracefully with the number of languages*
  - *... such that one can write one infrastructure, and it will work for all languages sharing it*
- new products or feataures can be made automatically available to all languages
  - *... but any required linguistic work must of course be done separately*
- having a flexible and extendable infrastructure will remove the burden from the language maintainers, and will be a cornerstone for being "future-proof"
- since infrastructure maintenance is costly in itself, having a larger user community to build on will spread the cost
- one often under-valued exercise is systematic testing; with an infrastructure that provides ready-made test templates and automatic testing as soon as you fill in the linguistic details, chances are that testing will be done more frequently

Drawbacks

- building a scalable, all-encompassing and tentatively future-proof infrastructure is considerably more demanding than making a small, DIY-for-your-own-needs infrastructure
- it isn't always easy to see where to draw the line between fixed, shared features, and language-specific variation

Code Sharing

Code sharing? Human languages are not like programming languages, so why?

It turns out that there actually is a lot to share:

- tags and feature names
- higher-order analysis
- code structure
- perhaps proper nouns

Tags And Feature Names

When designing the linguistic analysis, there are several things to consider:

- established grammatical tradition
- the same name for the same phenomenon across languages
- a sensible analysis for the language & phenomenon at hand

These things do not necessarily go in the same direction, and choosing which one to follow when they conflict is not easy.

- Following the tradition will make newcomers feel at home, ease introduction and lower the time it takes to get up to speed
- ... but if the grammatical tradition of one language is very different from other languages, the above principle might give us very different analyses for similar or identical phenomenons

Consequences of differences

Different analyses for similar phenomenon has practical consequences in both the short and long run:

- applications like MT translation will have to add one extra layer of conversion between languages
- cooperation across language teams will be harder
- it is one more source for possible bugs and inconsistencies

There is no clear winner - tradition or cross-language consistency - but this illustrates that the choice of seemingly trivial things such as tags and feature names will have far-reaching consequenses. It can be very hard to change things after just one year - imagine how it is after 10, 20 or more years.

One example

- Does North Sámi have six or seven cases? Ie, should acc/gen be analysed as one case (forget the name of it for the moment), or as two? Except for a few pronouns, the cases are identical everywhere.
- But in most other Sámi languages accusative is distinct from genitive at least in plural. In this case tradition is following cross-linguistic analysis, so even though you might argue that a more sensible linguistic analysis would be to just have one case, we have followed tradition and the other Sámi languages - and have two different analysis for what is one and the same word form.

Another example

- We have had different tags and analyses for past tense negative verbals in some of the Sámi languages, really by accident and uncoordinated work, not because of any major linguistic differences between the languages.
- This has led to issues when reading, testing and disambiguating the morphological analyses.
- This again leads to inconsistencies, bugs and wasted time.
- We have now finally corrected this, and we have consistent analyses of negation across all the Sámi languages

Higher-Order Analysis

- For the Sámi languages, we use one common dependency grammar, which takes as input the syntactically disambiguated sentence.
- This has worked wurprisingly well, so well that we have also tested it with faroese input - with quite acceptable results.
- The more abstract the linguistic representation, the more you'll find similarities across languages - and thus more code to share.
- Exactly what and how to share will of course vary from formalism to formalism, but the important thing is to look for similarities and repeated code.

Code Structure

- One of the big time savers in our work has been shared layout and structure of the code.
- If the overall organisation of the grammar follows the same pattern from language to language, it makes it very easy for a linguist working on one language to jump in and help with another language, even though the details are different
- You can spend time discussing exactly the linguistic details that are important, instead of explaining the organisation of your code - because the other linguists can see the trees immediately, not only the forrest.
- The structure with the best long-term organisation is not always obvious, and it takes some experience and iterations to work it out.
- Structured code with a shared organisation also helps debugging the code, and makes it easier to spot errors.
- In this respect, linguistic work is just like software development and coding practices.

Examples

- organisation of compounding code for compounding languages
- overall organisation of morphological and morphophonological processes
- We are still in the learning phase in the Giellatekno & Divvun teams for several aspects of our code.
- But for the cases where the structure has been worked out, the benefits are very clear and tangible.
- Having a clear and readable code structure is part of making your code future-proof.

Proper Nouns

Our experience is that proper nouns basically fall in two categories:

- language-specific names, typically native names
- Other names:
  - *majority language names*
  - *international nammes on products, organisations, well-known politicians etc*

Even though the morphology for the names will vary from language to langauge, the set of relevant names tend to be the same for all languages.

Sharing a common namebase for the second category (ie non-native names) will at least give some consistency across languages.

Caveat: doesn't work well across scripts, so you need one list for each script, and possibly also for each country in cases where the language in question is spoken in several countries.

Reuse

Design your linguistic components such that they can be reused as easily as possible

- you make transducers for morphological analysis, but you should also plan them for:
  - *... morphological generation*
  - *... spelling*
  - *... hyphenation*
  - *... use in dictionaries, etc*
- Not perfect in our infrastructure yet, but we are working our way towards a modell where we start out with an all-encompassing transducer
- this transducer should have unique symbols and tags for everything we can imagine
- it is in a sense an abstract transducer
- then we remove symbols, or subsets of the language, or change it based on tags and symbols
- in the end we should be able to generate all transducers we need from this single starting point, using just fst manipulations.

Reuse (cont.)

- This way we reuse one component in many contexts, and the reuse is easily extendable in a programmatic way
- New uses might require new tags, which equals new info in the lexicon. That is always a substantial amount of work, but will just make the lexicon richer (and usually more correct) for each iteration
- We have been going through our lexicons many times already.

Standards

Standards evolve with technology, and as such reflect the present (or close past) of the technological development.

- But some standards are really icebreakers and revolutions for lesser-resourced languages.
- **Unicode** is such a standard. Without it most of the world's languages would have been without a realistic chance to become usable in the digital society.
- **XML** is another such standard (but not all the document schemas written in XML).
- At least Unicode have a chance to be important for text processing in 50-100 years.
- XML might have a chance, but even if it is dead technology then, structured texts are so important that it won't be replaced with something with less structural capabilities.
- (Then again, text might not be very relevant in 100 years.)

Open source

- Open source as a principle is essential to the future of LT work for lesser-resourced languages.
- The risks and costs are just too high to rely on closed-source
- This is true for the linguistic resources (lexicons, grammars, etc.) (imagine these resources being controlled by a third party)
- ... as well as for the technology used both for development and in end-user products

Illustrative case

- The Divvun proofing tools have been tied to a commercial partner and their technology since the beginning of the project
- At that time there was no viable open-source alternative
- During the last project they went bankrupt
- When you have invested many man-years in a working system, and a whole society is waiting for updates, this is not what you were hoping for
- Even though we were aware of such risks, and thought we had made reasonable mitigations we nevertheless were caught off-guard.
- In a large time-scale perspective, this will happen over and over again, even so that major applications and operating systems will disappear and new ones will come.
- The only reasonable answer to this is to control the full development stack.
- And for small communities the only practical way of doing this is through open source, as part of a larger

community of users and developers for many languages.

The Divvun/GT development stack

At present we have a development stack based on open-source technologies for the following components:

- morphological analysis and generation (HFST + (earlier only) Xerox FST tools)
- disambiguation and syntactic parsing, even higher-order analysis (HFST / Xerox tools + VISLCG3)
- electronic dictionaries (XML)
- spellers (HFST / Xerox tools)
- hyphenators (HFST / Xerox tools)
- Machine Translation (HFST + VISLCG3 + Apertium)
- language learning (HFST + VISLCG3 + MySQL + ...)

We're working on the following:

- grammar checker (VISLCG3)
- speech synthesis (HFST + VISLCG3 + ??? - no open-source synthesis engine yet, but that will be a natural and logical extension)

We're still using Xerox a lot in our daily operations, but over time all reliance on closed source should be at least completely optional.

End-user tools

- End users use what they use - open-source or not
- we need to deliver to the users what they want
- what is needed is then open-source in the development stack
- ... but with support for both open and closed source in the user stack

Based on experience it is important that the top layer of closed source for integration with closed-source systems is as thin as possible.

Isolation and minimisation of closed source

Take proofing tools as an example. What we should have had to easily handle the bankruptcy of our subcontractor should have been:

- open-source for both development tools and end-user tools
- ... but a very thin layer of integration code on top, to make it work e.g. with MS Office
- Then the cost of replacing this thin layer with another would have been low
- if you have a large user base, the costs could be spread across several languages.
- As far as possible this strategy should be used everywhere for all products and services to future proof as much as possible of the core technologies.

User And Developer Community

- Some time in the future somebody will have to inherit our code
- a strong user and developer community is essential for long-time viability of the code
- ... this is the **real** issue when it comes to building LT resources for the next 100 years

New generations of developers

- A strong community is important to foster education and new generations
- it should be the basis for knowledge transfer from the old guys to the next generation
- the new developers should preferably come from the user community
- this has some practical consequences:

Inviting newcomers

- The barrier to entry should be as low as possible
    - *it should be easy to check out/download the code, do some changes, compile, test, install, and see that the changes are reflected in the installed speller, dictionary or whatever*
- the initial setup of the development evnironment should be fully automatic
- the introduction to the source code should follow a learning path
- the development environment needs to include systematic and automated testing of all components of the LT building
- ... and be fully integrated in the development process, so that it can tell newcomers whenever their additions and changes have caused something to break

End-user community

- In the long (and not-so-long) run, no LT project or LT development can exist in isolation.
- But building a real and functioning relationship between end users and developers isn't easy

Goals for end user community building

What should the end user community provide?

Why do we need one?

- a thriving end user community is the ultimate proof of success
- a thriving end user community is producing texts in their own language - hopefully with the help of the tools we provide
- devoted users give feedback and suggestions for improvements
- they may even provide (some of) their texts for inclusion in a corpus
- new corpus texts can be the basis for:
  - *improved tools*
  - *(suggestions for) new terminology*
  - *new dictionary entries and examples*
  - *... generally new data for all LT purposes*

Goals for end user community building (cont.)

In the best of cases this can turn into a positive circle, where all parts has the feeling that they both contribute and gets something in return - a win-win situation.

- This is what we really want, even more so because:
- Out of such a user community new developer generations can grow.
- New developer generations are the ultimate target for Sustainable LT Resources, as:
- in the end the human resources are the most valuable LT resource for lesser-resourced languages
- planning for sustainable LT resources means making oneself replaceable, and prepare for ones own nonexistence

Building user communities

There is no simple answer, but here's a list of things we have learned (or are trying to learn):

- be present where the users are
  - *it won't work to create arenas to which users are invited (we have tried)*
  - *=> it is better to join existing forums*
- be active and show that you care in responses to the users
- release often with updates from the users
- in cases where you can't follow user suggestions, be polite and clear when explaining why

Building user communities (cont.)

- visit user groups, such as:
  - *language centras*
  - *schools*
  - *municipalities*
  - *other important events and places for the language community*
- use modern technology to build a distributed developer organisation, even in cases where there is one common employer
- learn from the open source community :)

Summary

I have tried to show a few steps one could take to build Sustainable LT Resources for lesser-resourced languages:

- shared infrastructure
- shared code
- look-ahead planning
- flexible infrastructure
- use open source everywhere possible
- build user and developer communities, preferably for many languages
- *foster* the user community to produce new generations of developers

Conclusion

- For the types of languages we're concentrating on here, the most important resource is the speaker communities and the human resources
- That is, the most important language technology resources for the next 100 years are the speaker communities.
- If we invest in them, they will give back.
- The rest is details
- ... sort of :-D

The End

More info at **divvun.no** and **giellatekno.uit.no**

Thanks for listening!
sjur.n.moshagen@uit.no