THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Methods and tools for automating language engineering

GRÉGOIRE DÉTREZ

UNIVERSITY OF GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology & University of Gothenburg

Göteborg, Sweden 2016

Methods and tools for automating language engineering
GRÉGOIRE DÉTREZ
ISBN 978–91–628–9854–0 (Print) 978–91–628–9855–7 (PDF)

Technical Report no. 127D
Department of Computer Science and Engineering
Chalmers University of Technology & University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone: + 46 (0)31–772 1000

Typeset with LuaLaTeX
Printed by Ineko AB
Göteborg, Sweden 2016

Methods and tools for automating language engineering
Thesis for the degree of Doctor of Philosophy in Computer Science
GRÉGOIRE DÉTREZ
Department of Computer Science and Engineering
Chalmers University of Technology & University of Gothenburg

Abstract

Language-processing software is becoming increasingly present in our society. Making such tools available to the greater number is not just a question of access to technology but also a question of language as they need to be adapted, or localized, to each linguistic community. It is thus important to make the tools necessary to the engineering of language-processing systems as accessible as possible, for instance through automation. Not so much to help the traditional software creators but more importantly to enable communities to bring their language use into the digital world on their own terms.

Smart paradigms are created in the hope that they can decrease the amount of work for the lexicographer who wishes to create or update a morphological lexicon. In the first paper, we evaluate smart paradigms implemented in GF. How good are they to guess the correct inflection tables? How much information is required? How good are they at compressing the lexicon?

In the second paper, we take some distance from the smart paradigms, although they have been used in this work, they are not the main focus of the study. Instead, we compare two rule-based machine translation systems based on different translation models and try to determine the potential of a possible hybridization.

In the third paper we come back to the smart paradigms. If they can reduce the work of the lexicographer, someone still needs to create the smart paradigms in the first place. In this paper we explore the possibility of automatically creating smart paradigms based on existing traditional paradigms using machine-learning techniques.

Finally, the last paper presents a collection of tools meant to help grammar engineering work in the Grammatical Framework community: a tokenizer; a library to embedded grammars in Java applications; a build server; a document translator and a kernel to Jupyter notebooks.

Keywords: Natural language processing, Language Engineering, Morphology, Lexicon, Complexity

Acknowledgements

I thank my supervisor—Aarne Ranta—and the members of my PhD committee—Lars Borin, Harald Hammarström, Sally McKee and Bengt Nordström. I thank my co-authors and collaborators as well as the anonymous reviewers who provided comments on the publications included in this thesis. I also thank my friends and colleagues at the University of Gothenburg and Chalmers University of Technology, with a special mention to Peter Dybjer for his kindness and Guilhem for many interesting discussions.

This work would not have been possible without the support of the Swedish National Graduate School of Language Technology, GSLT, who funded my graduate studies.

I am grateful to my parents, Éric and Isabelle, who always believed in me even when I didn't and to my brothers, family and friends for providing a much needed alternative reality.

Finally, and maybe most of all I would like to thank Leonor for her support, her patience and her understanding during all this years.

Thesis

This thesis consists of an introduction and the following appended papers:

Paper A	G. Détrez and A. Ranta (2012). "Smart paradigms and the predictability and complexity of inflectional morphology". In: *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics.* Association for Computational Linguistics, 645–653
Paper B	G. Détrez, V. M. Sánchez-Cartagena, and A. Ranta (2014). "Sharing resources between free/open-source rule-based machine translation systems: Grammatical Framework and Apertium". In: *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14).* European Language Resources Association (ELRA)
Paper C	G. Détrez. "Learning Smart Paradigms". Under journal submission.
Paper D	G. Détrez (2015). *Tools for a grammar engineering community.* Tech. rep.

Contributions

Paper A: My contribution to this paper was to organize and run all the experiments on the smart paradigms, except the compression experiments, and 50% of the writing.

Paper B: I contributed about half of the experiments and writing.

Paper C: I am the only contributor to this paper.

Paper D: I am the only contributor to the work and writing of this paper except for the JPGF library to which I contributed about two-third of the coding.

Contents

Introduction

1 On the definition and challenges of language engineering

1.1 ˈlæŋgwɪdʒ ˌen.dʒɪˈnɪə.rɪŋ

Language engineering means different things in different communities. Two existing uses of the term are particularly relevant to the work presented in this thesis: *language engineering* as the application of natural language processing research; and *language engineering* as a form of language planning.[1]

Language engineering as applied natural language processing. The Oxford English Dictionary gives the following definition for language engineering:

> The field of computing that uses tools such as machine-readable dictionaries and sentence parsers in order to process natural languages for applications such as speech synthesis and machine translation.

Similarly, about twenty years ago, Cunningham 1998 suggested the following answer to the question "What is language engineering":

> Language Engineering is the discipline or act of engineering software systems that perform tasks involving processing human language. Both the construction process and its outputs are measurable and predictable. The literature of the field relates to both application of relevant scientific results and a body of practice.

In practice language engineering may involve various tasks such as lexicon creation, grammar engineering or corpus annotation; and has multiple well-known applications like spell-checking, machine translation, question answering and text analysis.

Stretching the definition we might also include *internationalization* and *localization*, the former being the process of designing or modifying software so that it can potentially be adapted to various languages and regions whereas the later is the process of adapting (internationalized) software to a specific region[2]. Internationalization and localization are sometimes included under the umbrella of language engineering, as it is the case at the Wikimedia Foundation[3].

In the context of translation, Sager 1994 suggests that "language engineering is concerned with the design and use of tools for activities involving languages". This is a rather broad definition but the interesting difference with the previous view of language

[1] A third use of *language engineering* is the design and implementation of *programming* languages. While not directly relevant to this thesis, it is worth mentioning that a tool like Grammatical Framework, that defines a domain-specific (programming) language to write grammars for (natural) languages, is also an example of language engineering in this sense.

[2] Source: `https://en.wikipedia.org/wiki/Internationalization_and_localization>`

[3] Source: `https://www.mediawiki.org/wiki/Wikimedia_Language_engineering`.

engineering as applied natural language processing is that not only creators but also users of the tools are viewed as doing language engineering. It also does not limit the definition to systems that *process* language (where we interpret "process language" as *operate on language as data*[4]).

Indeed localization can be applied to almost any existing software product and while it is traditionally done by human translators, it may involve complex natural language processing applications (see for instance Ranta, Unger, and Hussey 2015 for the use of GF in localization).

In this context, language engineering is best seen not as just a particular case of software engineering but as a trans-disciplinary activity which, as we observe below, may be done by different communities with different skills sets.

Language engineering as a form of language planning. Language engineering is sometimes used to describe the intentional modification of the language itself though engineering practices. In this sense it has been used as an alternative term to *language planning* or *language cultivation* describing "how an existent language is standardized to meet the exigencies of the modern wold" (Ammon et al. 2006). Interestingly Sager 1994, in its glossary, also gives a second definition of language engineering as "the techniques and practices concerned with adjusting the instrument of language to a number of specified uses, usually by the development of subject or situation-specific sub-languages."

While in practice most of the work presented in this thesis is related to natural language processing and its applications, and hence would fall under the corresponding use of language engineering, its motivation lies in the realization that there is a growing intersection between the two different uses of the term presented above: as our communications, our writings and our use of language in general is increasingly enabled and shaped by software that automatically analyses, 'auto-corrects' or even censors what we say and what we write, we need to look at how the software designed to process language is also used to shape it. It is our belief that a community should be able to shape its own language and for this to be possible, language engineering need to be made as accessible as possible, in both senses of the term.

1.2 The importance of free software

"Free software" means software that respects users' freedom as defined by the Free Software Foundation[5]:

- The freedom to run the program as you wish, for any purpose (freedom 0).

[4] According to the Oxford Dictionary, "Process: (Computing) Operate on (data) by means of a program."

[5] Note that we ignore here the distinction which is sometimes made between *free software* and *open-source software* and which is technically between copyleft and non-copyleft licences. For more information, we invite the reader to look at the definition from the Free Software Foundation (`https://www.gnu.org/philosophy/free-sw.html`) and the Open Source Initiative (`https://opensource.org/faq`).

- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

The case have been made many times as to why research software, and in particular software that is the product of publicly funded research, should be made available as free software. The main arguments are that

- Publicly funded research should have as a primary goal to advance the sum of human knowledge. As such, its output should be made as widely available as possible.
- As more and more research critically depends on increasingly complex computer programs, sometimes specifically built for the task at hand, to generate and analyze data—and, as argued for instance by Hey, Tansley, and Tolle 2009, this is not anymore limited to a few fields such as computer science—, limiting access to those programs and their source code is harmful for the peer review process and for reproducibility in general.
- Research is an iterative process where new results are built on top of existing ones. As the complexity of the software systems used in published research increases it becomes more and more difficult to rebuild the foundation that allows the improvement of existing results.

In this section, we would like to bring the reader's attention to some reasons more specific to language engineering, whether the software is produced by researchers or not.

Misaligned economic incentives. Adapting software to a new language, a process often known as localization, has a certain cost. In the case of nonfree software, whether or not the software is adapted to a particular language depends only on the will of the original software creator. Naturally, for them it might seem like a simple cost-benefit economic trade-off: can the cost of localizing the software in a given language be outweighed by the expected benefits from catering to this language community?

This trade-off is not favorable to smaller languages as the cost of localization is more or less constant (although for many languages it might be difficult to find language experts which can make localization in those languages more expansive) but the expected return on investment varies with the size and economic weight of the language community.

In addition, in many places where multilingualism is the norm, it might be tempting to think that it is enough to cater for the official language or the one with the largest community, thus reinforcing the dominant position of one language over the others.

To take a recent example, an article from The Atlantic relates the sudden increase of Facebook usage in Myanmar, where prohibitive barriers in the acquisition of cell phones have recently been relaxed[6]. This article reports that, despite the popularity of the Facebook application in the country, people are forced to use it in English as the application is not available in Burmese, let alone in any of the country's regional languages and only Facebook has the possibility to adapt the interface to another language.[7]

It may be argued that those communities may be perfectly happy to use the software in the dominant language. Nonetheless we believe that it is a moral imperative that they are in a position to make this choice for themselves.

The language-planning aspect of language engineering. As argued in the previous section, language-processing software is increasingly in a position to shape our language in its everyday use.

An open-source license does not only make the source freely available for anyone to examine and adapt to their own (language) preferences but in addition the transparent governance that is common in free-software communities makes is possible for anyone to study, discuss or challenge the decisions that have been implemented, whether technical or linguistic. (For an overview of governance in free-software communities, see Fogel 2014.)

So not only, as argued above, free software allows communities to adapt and use systems in the language of their choosing but it is also a prerequisite for them to take a greater role in shaping and governing their language.

1.3 Challenges in language engineering

We see as an additional challenge in language engineering the fact that it is unlikely that software creators can themselves produce a tool which is useful for more than a handful of languages. The traditional way to solve this problem is to outsource the translation, or other tasks necessary to localize the software to other people who in addition to their own language, know the original language of the product (most likely English).

In free software, it is not uncommon to see those translators organizing themselves in communities which are transversal to the groups that develop the software. So on one hand there are groups of people dedicated to a particular piece of software, like the document reader Evince[8], and on the other hand groups of people dedicated to making free software available in a particular language and who may work on many different pieces of software. A good example are the language teams in Gnome[9].

Ideally, and in the simplest cases, this is a simple task of translating text from one language to the other, but as the complexity of the linguistic functionalities of the software increases, the task becomes more and more technical. Example of common tasks may be maintaining a dictionary for spell-checking, writing rules for a grammar checker, or localizing a controlled language.

[6] Craig Mod, The Facebook-Loving Farmers of Myanmar, January 21, 2016.

[7] Note that while we are here concentrating on the language engineering aspect of the problem, there are other potentially more concerning issues when a community, and in particular a community in a transition to a more democratic governance, is relying on a centrally censored and opaquely governed infrastructure.

[8] `https://wiki.gnome.org/Apps/Evince`

[9] `https://l10n.gnome.org/`

Many languages used today may not have in their community enough members with the necessary skills to perform the work necessary for their language to be available with the same level of functionality as those with more resources. Thus we consider that it is important to make the tools necessary for language engineers as accessible as possible. This is the motivation behind most of the work presented in this thesis.

2 On lexicons

Descriptions of natural languages are often divided in two parts: on one side there is the language *lexicon*, the list of its words, sometimes also called the language *wordstock*. On the other side there is the language *grammar* that describes how those words can be assembled together to create larger items such as sentences and paragraphs.

Natural language processing is a field at the intersection of computer science, linguistics and artificial intelligence which is focused on the manipulation of human languages by machines. Lexicons are nowadays a cornerstone of a large set of tasks in natural language processing.

2.1 Motivation: what are lexicons for?

One first example of a task which is part of natural language processing, with which the reader is undoubtedly already familiar, is *spell-checking*. In its simplest form, a spell checker is a very trivial program that marks in a document every word that does not belong to its predefined lexicon. Of course modern spell checkers do much more than just looking up words in a lexicon: they are able to automatically detect the language in use, propose alternatives to incorrect forms and learn new words, but the essence of a good spell-checker is still to build a large, high quality lexicon for a particular language.

Another familiar task of natural language processing is *information retrieval*, which most people probably know better as "searching" in a set of documents, the most common example of which would be web search. Lexicons are very useful for search engines and are used for instance to automatically expand a query to include related word forms. For example, modern search engines automatically provide you with result for both the singular and the plural form of an English keyword (when searching for *automaton*, Google returns results for both *automaton* and *automata*). This is especially important for languages where every word can have a great number of different forms. Although it would be possible to manually provide to the search engine the two forms of an English word (singular and plural, like *automaton* and *automata*) it would be much more work for a Finnish speaker who may have to write thousands of forms for a single noun!

A close parent of information retrieval is *information extraction* where the goal is not anymore to find a particular document but to extract structured data from a collection of text documents. Structured data is a term used in software engineering to refer to data that fits a predefined structure, or *model*. Note that texts written in natural language are usually considered *unstructured data* in this context although language has a lot of internal structure (defining the model underlying this structure is still a major challenge for formal linguists and natural language processing researchers).

One of the major and oldest application of natural language processing is automated translation of natural language, often called *machine translation*. Here as well, lexicons are used. They can be a central part of the translation process, providing the basis for analysing the language to be translated (the *source language*), translating (using what is often referred to as a *bilingual lexicon*) and generating text in the *target language* (the language to translate into). But a lexicon can also be used for peripheral tasks such as text-alignment, which is a process during which a text and its translation are aligned on a detailed level (usually the sentence or the word). Alignment is a very important step in statistical machine translation, where instead of designing the translation process manually, the programmer lets the computer "learn" the correct way to translate between two languages by analysing a large quantity of texts aligned with their manual translation.

Other, more technical tasks in natural language processing use lexicons as well. *Part of speech tagging* is one of them. It is the process of assigning to each word in a sentence its grammatical category, such as noun or verb. Having a lexicon can greatly reduce the work needed for such a task by taking care of most forms, for which the lexicon gives you only one possible tag, leaving only the job of disambiguating homographs and handling unknown words (which is in no way a trivial job).

There are still many other tasks that could be listed here, and I do not intend to give a comprehensive list. The last example I can mention is parsing. Parsing a sentence is trying to extract its internal structure, the way it is constructed, according to a grammar of the language. This may include identifying the main verb, its subject and complement, etc. Parsing is what GF[10] does and why it needs lexicons.

2.2 What exactly is a lexicon?

The word *lexicon* comes from the Greek λεξικός (*lexikos*, "of words"). A lexicon is often defined as a list of words. One problem with this definition is that the term *word* is not precisely and uniquely defined in linguistics. So we prefer to speak about a list of *lexemes*. From the same origin as the word lexicon itself, a lexeme is an abstraction that groups together inflected forms taken by a single word.

Let me elaborate.

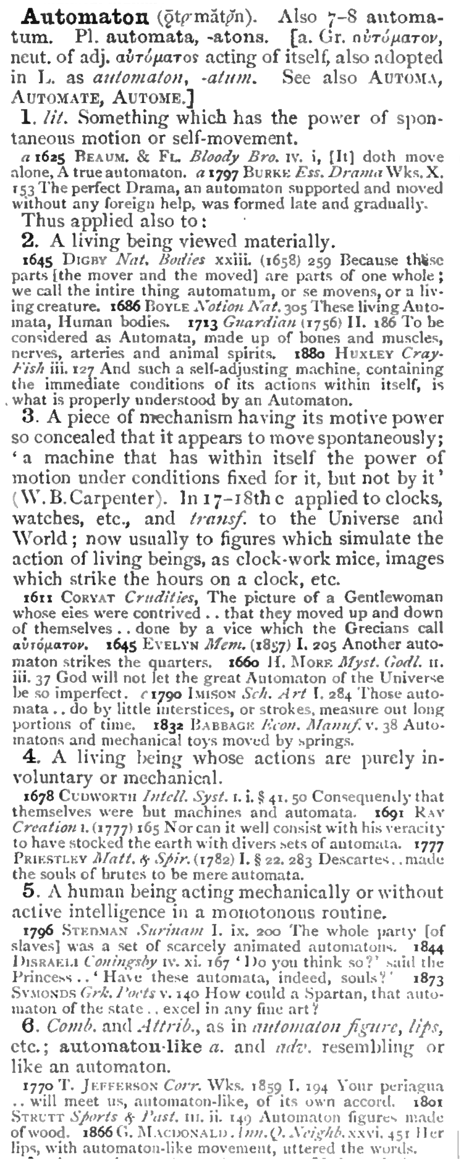
A *word form* is a unique suite of letters that appears in a sentence. For instance *woman* and *women* are two different word forms. A *lexeme* on the other hand is abstracted away from the inflection, so *woman* and *women* are said to belong to the same lexeme. A lexicon is a list of lexemes.

A lexeme being an abstract entity, we need a concrete way to represent the abstract lexemes. This is often solved by choosing a representative among the word forms of the lexeme. This representative is called a *lemma* or *dictionary form* because it generally coincides with the form which is listed in traditional dictionaries, which are themselves a form of lexicon.

Most dictionaries are written to be used by human, and are example of unstructured data or semi-structured data (as opposed to structured data which is data structured against a predefined formal data model).

[10]Grammatical Framework, or GF, is a set of tools and libraries for parsing, generation and translation of natural language using multilingual grammars and type theory.

Dictionaries are only one possible form of lexicon. The main difference between different types of lexicons is the data associated with each lemma. In a traditional dictionary, each lemma is associated with some inflection information (e.g. a plural form), its part of speech and one or several definitions.



Automaton (ǭtǫ·mătǫ̆n). Also 7–8 **automatum.** Pl. **automata, -atons.** [a. Gr. αὐτόματον, neut. of adj. αὐτόματος acting of itself, also adopted in L. as *automaton, -atum.* See also AUTOMA, AUTOMATE, AUTOME.]

1. *lit.* Something which has the power of spontaneous motion or self-movement.

a **1625** BEAUM. & FL. *Bloody Bro.* IV. i, [It] doth move alone, A true automaton. *a* **1797** BURKE *Ess. Drama* Wks. X. 153 The perfect Drama, an automaton supported and moved without any foreign help, was formed late and gradually.

Thus applied also to:

2. A living being viewed materially.

1645 DIGBY *Nat. Bodies* xxiii. (1658) 259 Because these parts [the mover and the moved] are parts of one whole; we call the intire thing automatum, or se movens, or a living creature. **1686** BOYLE *Notion Nat.* 305 These living Automata, Human bodies. **1713** *Guardian* (1756) II. 186 To be considered as Automata, made up of bones and muscles, nerves, arteries and animal spirits. **1880** HUXLEY *Cray-Fish* iii. 127 And such a self-adjusting machine, containing the immediate conditions of its actions within itself, is what is properly understood by an Automaton.

3. A piece of mechanism having its motive power so concealed that it appears to move spontaneously; 'a machine that has within itself the power of motion under conditions fixed for it, but not by it' (W. B. Carpenter). In 17–18th c applied to clocks, watches, etc., and *transf.* to the Universe and World; now usually to figures which simulate the action of living beings, as clock-work mice, images which strike the hours on a clock, etc.

1611 CORYAT *Crudities,* The picture of a Gentlewoman whose eies were contrived .. that they moved up and down of themselves .. done by a vice which the Grecians call αὐτόματον. **1645** EVELYN *Mem.* (1857) I. 205 Another automaton strikes the quarters. **1660** H. MORE *Myst. Godl.* II. iii. 37 God will not let the great Automaton of the Universe be so imperfect. *c* **1790** IMISON *Sch. Art* I. 284 Those automata .. do by little interstices, or strokes, measure out long portions of time. **1832** BABBAGE *Econ. Manuf.* v. 38 Automatons and mechanical toys moved by springs.

4. A living being whose actions are purely involuntary or mechanical.

1678 CUDWORTH *Intell. Syst.* I. i. § 41. 50 Consequently that themselves were but machines and automata. **1691** RAY *Creation* I. (1777) 165 Nor can it well consist with his veracity to have stocked the earth with divers sets of automata. **1777** PRIESTLEY *Matt. & Spir.* (1782) I. § 22. 283 Descartes .. made the souls of brutes to be mere automata.

5. A human being acting mechanically or without active intelligence in a monotonous routine.

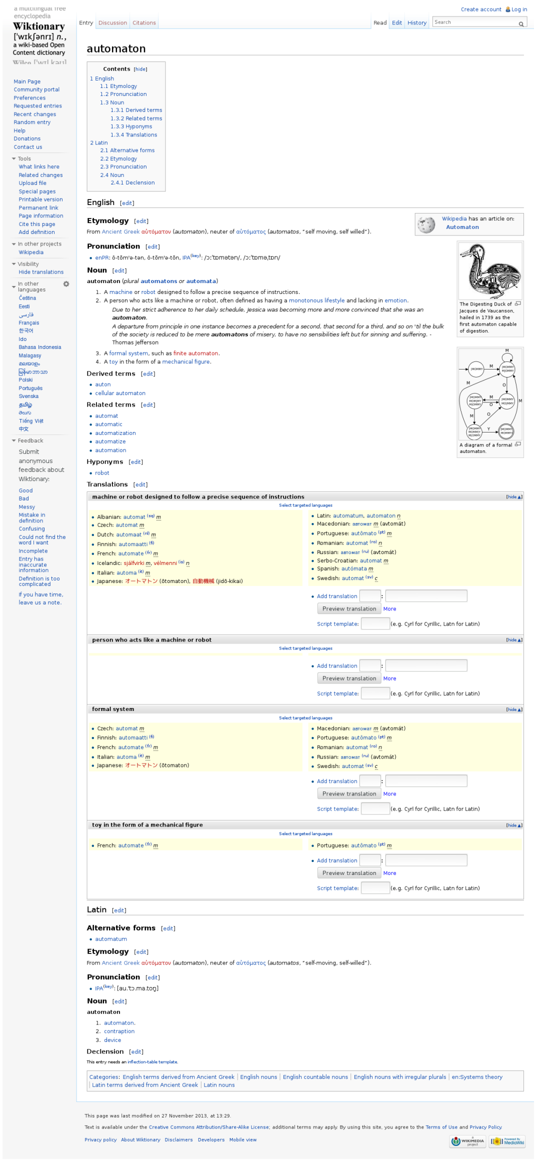
1796 STEDMAN *Surinam* I. ix. 200 The whole party [of slaves] was a set of scarcely animated automatons. **1844** DISRAELI *Coningsby* IV. xi. 167 'Do you think so?' said the Princess .. 'Have these automata, indeed, souls?' **1873** SYMONDS *Grk. Poets* v. 140 How could a Spartan, that automaton of the state .. excel in any fine art?

6. *Comb.* and *Attrib.*, as in *automaton figure, lips,* etc.; **automaton-like** *a.* and *adv.* resembling or like an automaton.

1770 T. JEFFERSON *Corr.* Wks. 1859 I. 194 Your periagua .. will meet us, automaton-like, of its own accord. **1801** STRUTT *Sports & Past.* III. ii. 149 Automaton figures made of wood. **1866** G. MACDONALD *Ann. Q. Neighb.* xxvi. 451 Her lips, with automaton-like movement, uttered the words.

Figure 1.1: *Entry for the word "Automaton" in* A New English Dictionary on Historical Principles: Founded Mainly on the Materials Collected by the Philological Society (1893), *James A. H. Murray.*

An example of dictionary entry for the word *"automaton"* is given Figure 1.1. The entry gives several definitions (six) and also provides information about pronunciation and morphology.


a multilingual free encyclopedia
Wiktionary
['wɪkʃənrɪ] *n.*,
a wiki-based Open Content dictionary
Wikɪ ['wɪl kaɪl]

Main Page
Community portal
Preferences
Requested entries
Recent changes
Random entry
Help
Donations
Contact us

Tools
What links here
Related changes
Upload file
Special pages
Printable version
Permanent link
Page information
Cite this page
Add definition

In other projects
Wikipedia

Visibility
Hide translations

In other languages
Čeština
Eesti
فارسی
Français
한국어
Ido
Bahasa Indonesia
Malagasy
മലയാളം
မြန်မာဘာသာ
Polski
Português
Svenska
தமிழ்
తెలుగు
Tiếng Việt
中文

Feedback
Submit anonymous feedback about Wiktionary:
Good
Bad
Messy
Mistake in definition
Confusing
Could not find the word I want
Incomplete
Entry has inaccurate information
Definition is too complicated
If you have time, leave us a note.

Create account Log in
Entry Discussion Citations   Read Edit History Search

automaton

Contents [hide]
1 English
  1.1 Etymology
  1.2 Pronunciation
  1.3 Noun
    1.3.1 Derived terms
    1.3.2 Related terms
    1.3.3 Hyponyms
    1.3.4 Translations
2 Latin
  2.1 Alternative forms
  2.2 Etymology
  2.3 Pronunciation
  2.4 Noun
    2.4.1 Declension

English [edit]

Etymology [edit]

From Ancient Greek αὐτόματον (*automaton*), neuter of αὐτόματος (*automatos*, "self moving, self willed").

Wikipedia has an article on: **Automaton**

Pronunciation [edit]

- enPR: ô-tŏm'ə-tən, ô-tŏm'ə-tŏn, IPA$^{(key)}$: /ɔːˈtɒmətən/, /ɔːˈtɒməˌtɒn/

Noun [edit]

automaton (*plural* **automatons** *or* **automata**)

1. A machine or robot designed to follow a precise sequence of instructions.
2. A person who acts like a machine or robot, often defined as having a monotonous lifestyle and lacking in emotion.
   Due to her strict adherence to her daily schedule, Jessica was becoming more and more convinced that she was an ***automaton***.
   A departure from principle in one instance becomes a precedent for a second, that second for a third, and so on 'til the bulk of the society is reduced to be mere ***automatons*** *of misery, to have no sensibilities left but for sinning and suffering.* - Thomas Jefferson
3. A formal system, such as finite automaton.
4. A toy in the form of a mechanical figure.

The Digesting Duck of Jacques de Vaucanson, hailed in 1739 as the first automaton capable of digestion.

A diagram of a formal automaton.

Derived terms [edit]

- auton
- cellular automaton

Related terms [edit]

- automat
- automatic
- automatization
- automatize
- automation

Hyponyms [edit]

- robot

Translations [edit]

machine or robot designed to follow a precise sequence of instructions [hide ▲]
Select targeted languages

- Albanian: automat $^{(sq)}$ *m*
- Czech: automat *m*
- Dutch: automaat $^{(nl)}$ *m*
- Finnish: automaatti $^{(fi)}$
- French: automate $^{(fr)}$ *m*
- Icelandic: sjálfvirki *m*, vélmenni $^{(is)}$ *n*
- Italian: automa $^{(it)}$ *m*
- Japanese: オートマトン (ōtomaton), 自動機械 (jidō-kikai)
- Latin: automatum, automaton *n*
- Macedonian: автомат *m* (avtomát)
- Portuguese: autômato $^{(pt)}$ *m*
- Romanian: automat $^{(ro)}$ *n*
- Russian: автомат $^{(ru)}$ (avtomát)
- Serbo-Croatian: automat *m*
- Spanish: autómata *m*
- Swedish: automat $^{(sv)}$ *c*

- Add translation :
Preview translation More
Script template: (e.g. Cyrl for Cyrillic, Latn for Latin)

person who acts like a machine or robot [hide ▲]
Select targeted languages

- Add translation :
Preview translation More
Script template: (e.g. Cyrl for Cyrillic, Latn for Latin)

formal system [hide ▲]
Select targeted languages

- Czech: automat *m*
- Finnish: automaatti $^{(fi)}$
- French: automate $^{(fr)}$ *m*
- Italian: automa $^{(it)}$ *m*
- Japanese: オートマトン (ōtomaton)
- Macedonian: автомат *m* (avtomát)
- Portuguese: autômato $^{(pt)}$ *m*
- Romanian: automat $^{(ro)}$ *n*
- Russian: автомат $^{(ru)}$ (avtomát)
- Swedish: automat $^{(sv)}$ *c*

- Add translation :
Preview translation More
Script template: (e.g. Cyrl for Cyrillic, Latn for Latin)

toy in the form of a mechanical figure [hide ▲]
Select targeted languages

- French: automate $^{(fr)}$ *m*
- Portuguese: autômato $^{(pt)}$ *m*

- Add translation :
Preview translation More
Script template: (e.g. Cyrl for Cyrillic, Latn for Latin)

Latin [edit]

Alternative forms [edit]

- automatum

Etymology [edit]

From Ancient Greek αὐτόματον (*automaton*), neuter of αὐτόματος (*automatos*, "self-moving, self-willed").

Pronunciation [edit]

- IPA$^{(key)}$: [au.ˈtɔ.ma.toŋ]

Noun [edit]

automaton

1. automaton.
2. contraption
3. device

Declension [edit]

This entry needs an inflection-table template.

Categories: English terms derived from Ancient Greek | English nouns | English countable nouns | English nouns with irregular plurals | en:Systems theory | Latin terms derived from Ancient Greek | Latin nouns

This page was last modified on 27 November 2013, at 13:29.

Text is available under the Creative Commons Attribution/Share-Alike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.

Privacy policy About Wiktionary Disclaimers Developers Mobile view

Figure 1.2: *Wiktionary entry for the word "Automaton". Retrived 2013–12–11*

Some dictionaries may give a lot more information. For instance, the automaton entry in the English Wiktionary provides etymology, pronunciation, derived terms, related terms, hyponyms and translations (Figure 1.2).

In this thesis, we focus on lexicons which are primarily targeted to be used by a computer and not read by a human. Those are instances of structured data and written in a strict, formally defined syntax which makes them easy for a computer program to handle. For instance, Listing 1 shows a small extract of a morphological lexicon encoded using a common markup language called XML.

```
<lexicalEntry id="automate_2">
  <formSet>
    <lemmatizedForm>
      <orthography>automate</orthography>
      <grammaticalCategory>commonNoun</grammaticalCategory>
      <grammaticalGender>masculine</grammaticalGender>
    </lemmatizedForm>
    <inflectedForm>
      <orthography>automate</orthography>
      <grammaticalNumber>singular</grammaticalNumber>
    </inflectedForm>
    <inflectedForm>
      <orthography>automates</orthography>
      <grammaticalNumber>plural</grammaticalNumber>
    </inflectedForm>
  </formSet>
  <originatingEntry target="Morphalou-1.0">automate commonNoun masculine</originatingEntry>
</lexicalEntry>
```

Listing 1: Morphalou extract. The format of the lexicon is not easily understood by a human but is designed to be parsed by programs.

Other lexicons associate with a lemma in one language one or more lemmas of a different language. This kind of lexicon is referred to as *bilingual lexicon*. An example of bilingual entries used in the Apertium machine translation system for the English/Spanish language pair is given Listing 2.

```
<e><p><l>autobiography<s n="n"/></l><r>autobiografía<s n="n"/><s n="f"/></r></p></e>
<e><p><l>automatism<s n="n"/></l><r>automatismo<s n="n"/><s n="m"/></r></p></e>
<e><p><l>automaton<s n="n"/></l><r>autómata<s n="n"/><s n="m"/></r></p></e>
<e><p><l>automedication<s n="n"/></l><r>automedicación<s n="n"/><s n="f"/></r></p></e>
<e><p><l>automotion<s n="n"/></l><r>automoción<s n="n"/><s n="f"/></r></p></e>
<e><p><l>autonomy<s n="n"/></l><r>autonomía<s n="n"/><s n="f"/></r></p></e>
<e><p><l>autopsy<s n="n"/></l><r>autopsia<s n="n"/><s n="f"/></r></p></e>
<e><p><l>autumn<s n="n"/></l><r>otoño<s n="n"/><s n="m"/></r></p></e>
```

Listing 2: Small extract of the Apertium bilingual dictionary for the English $\rightarrow$ Spanish translator.

The particular kind of lexicon we concentrate on in this thesis is what we refer to as a *morphological lexicon*. It is a machine readable database that lists, for each lexeme, all the possible inflected forms associated. The Morphalou snippet above is an example of a morphological lexicon, one that we use again later in this thesis. An other example, with a somewhat simpler format where we list the forms separated by a comma is given Listing 3.

```
automaatio,automaatio,automaation,automaatiota,automaationa,automaatioon,automaatioiden,…
automaatti,automaatti,automaatin,automaattia,automaattina,automaattiin,automaattien,…
automaattisuus,automaattisuus,automaattisuuden,automaattisuutta,automaattisuutena,…
automatiikka,automatiikka,automatiikan,automatiikkaa,automatiikkana,automatiikkaan,…
automatisointi,automatisointi,automatisoinnin,automatisointia,automatisointina,…
autonomia,autonomia,autonomian,autonomiaa,autonomiana,autonomiaan,autonomioiden,…
autonominen,autonominen,autonomisen,autonomista,autonomisena,autonomiseen,autonomisten,…
```

Listing 3: A few entries from a morphological lexicon in *comma separated value* format, extracted from the Finnish lexicon in GF.

2.3 Lexicon creation

Creating a morphological lexicon is a tedious task, especially for languages having a richer morphology than English and which can have tens of forms for a single lexeme. For instance, on some account, Finnish verbs are said to have more than ten thousand (10 000) forms.

Even if a lexicon already exists for a particular language, it might not be usable for a variety of reasons:

- *It may not be distributed.* Companies selling proprietary spell checking software, for instance, might create large lexicons but won't distribute them to the community.
- *It is only available at a prohibitive cost.* It is important for researchers to be able to validate each other's results. The need to buy an expansive lexicon to replicate a study is an obstacle in that direction.
- *The format is difficult to exploit.* This is often the case in digitalized paper lexicons.
- *The license may prohibit some usages.* For instance it might prevent you from distributing your changes to the lexicon (adding information or new entries, or correcting errors) or you may not be able to use it in a commercial activity.

Because of all those reasons, lexicons often have to be created not once but several times for the same language. In addition, even once the lexicon is created and made available under satisfying conditions it still needs to be regularly updated to include new words, remove deprecated ones and incorporate orthographic changes.

This explains why, despite being one of the central piece of natural language processing, lexicon creation is not, and is probably never going to be, a finished task and why it is important to make it as non work intensive as possible. This is especially true for small and under-resourced languages, which are often ignored by large companies because the market for linguistic tools in those languages is too small to be economically interesting. Many such languages rely only on their community to create those tools using open-source methodologies. In those cases, you cannot always expect the lexicographer to be a trained expert and it is critical to have tools that can help them as much as possible.

For many years, linguists have studied patterns in the formation of word forms and have written rules that can be used to correctly generate the forms of a particular lexeme from its lemma. By grouping together all the lexemes following the same rules in what we call a *paradigm*, the work needed to create a morphological lexicon can be greatly reduced: instead of having to write all the forms manually, the rules for the paradigm only have to be defined once, and then one only needs to list the lemmas of the lexemes in this paradigm.

Classical examples of paradigms are Latin declensions. Each declension encapsulates a set of rules that, when applied to a lemma, allow the construction of all forms of the word. For instance, the first declension, traditionally exemplified by the word *rosa*:

Case	Singular	Plural
nominative	rosa	rosae
genitive	rosae	rosārum
dative	rosae	rosīs
accusative	rosam	rosās
ablative	rosā	rosīs
vocative	rosa	rosae

We can refer to the same table not only to find the forms of the word *rosa* itself but any first declension noun, such as for instance *machina* by following the model:

Case	Singular	Plural
nominative	māchina	māchinae
genitive	māchinae	māchinārum
dative	māchinae	māchinīs
accusative	māchinam	māchinās
ablative	māchinā	māchinīs
vocative	māchina	māchinae

We can do this extrapolation because we are able to see the table as a set of rules instead of just word forms:

Case	Singular	Plural
nominative	##+a	##+ae
genitive	##+ae	##+ārum
dative	##+ae	##+īs
accusative	##+am	##+ās
ablative	##+ā	##+īs
vocative	##+a	##+ae

This mechanism is really useful and allows a formidable compression of the work needed to describe a lexicon. To give an idea of what this compression represents, let's take another example. One of the reference for verb conjugation in French is a book titled *La Conjugaison pour tous* (Conjugation for all) in the collection Bescherelle, often referred to as "The Bescherelle". The book first gives the full inflection tables for *model verbs*, about a hundred of them depending on the edition. It then provides a list of several thousands verbs (9600 in the 2012 edition) for which only the lemma is given, together with a pointer to the model table (the models are all given a unique number, different from the page number, which is used to identify them).

From this information only, the lemmas and the inflection tables for the model verbs, the reader is able to reconstruct the inflection table of any verb. Now, still in the 2012 edition, if you count the number of pages, you get the following: 104 pages for the model tables (each table fits on one page) and 81 pages for the list of verbs. (The book itself contains more than 185 pages, including some grammar rules but we consider those irrelevant for our current calculation.)

If the author had needed instead to give the full inflection table for each of the 9600 verbs, it would have required 9600 pages. This means that we saved about 9400 pages, or that we have a compression ratio of $185/9600 = 0.019$, about 2%: The space needed to describe the lexicon was reduced by a factor 50. We come back to this idea of lexicon compression when evaluating the Grammatical Framework *smart paradigms*.

There is a natural trade-off between the work needed to define and apply the paradigms and the creation of the lexicon: having more complex paradigms allows you to have less of them and thus makes the work of the lexicographer easier because they have less alternatives to choose from; on the other hand it requires more work to apply those rules when computing the forms in the lexicon. The idea behind the smart paradigms is that this trade-off is not optimized in traditional paradigms. More precisely, the traditional paradigms, written to be understood and applied by human readers, are not making use of the full potential of the computer which is very good at consistently applying complex rules. By creating more complex paradigms, the work of the lexicographer can be greatly reduced, ideally to listing the lemmas and letting the computer figure out the inflection automatically. (In practice we still need to help the process by sometimes giving "hints" on how the lexeme is inflected. In the case of smart paradigms, those hints are additional inflected forms.)

Paper A in this thesis describes in more details the smart paradigms as implemented in GF and proposes an evaluation of some of the existing smart paradigms.

In Paper B we present different methods for sharing data between two open-source machine translation projects. While not focused on smart paradigms, this work shows an example of their usage in reducing the manual work needed to port a lexicon from one format to an other (from Apertium to GF).

In Paper C, we attempt to automatically learn smart paradigms on top of existing "classical" paradigms using machine learning.

3 The many ways to improve language engineering

We have defined language engineering as an activity at the intersection of many disciplines from translation to software engineering. Many of those disciplines have large bodies of knowledge, tools and practices that we can draw upon. In this thesis, we have also experimented with some of those ideas, which are presented in Paper D.

Some of the tools come directly from software engineering. In particular, as they are often working on the same product, language engineers and software engineers also often share the same tools. Examples are code repositories (git, darcs, CVS, etc.) or compilers (gcc, ghs, etc. but also tools like gettext and GF). The line is even thinner in grammar engineering in GF as the grammar is written in a domain-specific language, which is a programming language specially created to write natural language grammars.

The large body of work about the development and governance of free-software communities can also teach us a lot on how language engineering can be done in an open and sustainable way. One example is the *mailing list*, which is a common tool in free-software communities, whether of software development communities (one of the most famous is certainly the LKML, the Linux kernel mailing list) or language communities (like the linuxfr mailing list or the fsfe translator mailing list).

Finally, by automating interesting research evaluations developed by computational linguists, we can use new metrics and tools to not only evaluate the linguistic quality of the software but also to make sure that projects like Grammatical Framework continue to be state-of-the art tools that researchers can use with confidence to build new results on.

We have barely scratched the surface of what each of those fields could bring to language engineering and how to improve the work process of many of those who dedicate time, often as volunteers, to improve the linguistic quality and availability of the tools we use every day.

4 Future prospects

In the first paper, we have defined and used several metrics to evaluate smart paradigms. A natural question that we plan to explore in the future is what else can we learn from those metrics? Are they only useful for grammar engineering or can they reveal something on the modeled language? Does the complexity of the smart paradigms reflect the complexity of a language's morphology or does it only reflect different programmers' styles?

Another interesting question is to see whether a correlation exists between metrics on GF code and traditional linguistic metrics such as indices of synthesis and fusion. I wish to explore the relation, if it exists, between the complexity of the model (the GF code) and the complexity of the languages in traditional linguistics.

Finally, we hope to be able to move beyond morphology and investigate syntactic complexity using techniques borrowed from software complexity measurement.

References

Ammon, U. et al., eds. (2006). *Sociolinguistics : an international handbook of the science of language and society*. Berlin; New York: Walter de Gruyter.

Cunningham, H. (1998). “A definition and short history of Language Engineering”. In: *Natural Language Engineering* 5.01.

Fogel, K. (2014). *Producing Open Source Software: How to Run a Successful Free Software Project*. 2nd ed. O’Reilly Media. URL: `http://www.producingoss.com/`.

Hey, T., S. Tansley, and K. Tolle, eds. (2009). *The Fourth Paradigm: Data-intensive Scientific Discovery*. Redmond, Washington: Microsoft Research.

Ranta, A., C. Unger, and D. V. Hussey (2015). “Grammar Engineering for a Customer: a Case Study with Five Languages”. In: DOI: `10.18653/v1/w15-3301`.

Sager, J. C. (1994). *Language Engineering and Translation*. Amsterdam, Netherlands: John Benjamins Publishing Co.

Paper A

Smart paradigms and the predictability and complexity of inflectional morphology

Abstract

Morphological lexica are often implemented on top of morphological paradigms, corresponding to different ways of building the full inflection table of a word. Computationally precise lexica may use hundreds of paradigms, and it can be hard for a lexicographer to choose among them. To automate this task, this paper introduces the notion of a **smart paradigm**. It is a meta-paradigm, which inspects the base form and tries to infer which low-level paradigm applies. If the result is uncertain, more forms are given for discrimination. The number of forms needed in average is a measure of **predictability** of an inflection system. The overall **complexity** of the system also has to take into account the code size of the paradigms definition itself. This paper evaluates the smart paradigms implemented in the open-source GF Resource Grammar Library. Predictability and complexity are estimated for four different languages: English, French, Swedish, and Finnish. The main result is that predictability does not decrease when the complexity of morphology grows, which means that smart paradigms provide an efficient tool for the manual construction and/or automatically bootstrapping of lexica.

1 Introduction

Paradigms are a cornerstone of grammars in the European tradition. A classical Latin grammar has five paradigms for nouns ("declensions") and four for verbs ("conjugations"). The modern reference on French verbs, *Bescherelle* (Bescherelle 1997), has 88 paradigms for verbs. Swedish grammars traditionally have, like Latin, five paradigms for nouns and four for verbs, but a modern computational account (Hellberg 1978), aiming for more precision, has 235 paradigms for Swedish.

Mathematically, a paradigm is a function that produces inflection tables. Its argument is a word string (either a **dictionary form** or a **stem**), and its value is an n-tuple of strings (the word forms):

$$P : \text{String} \rightarrow \text{String}^n$$

We assume that the exponent n is determined by the language and the part of speech. For instance, English verbs might have $n = 5$ (for *sing, sings, sang, sung, singing*), whereas for French verbs in *Bescherelle*, $n = 51$. We assume the tuples to be ordered, so that for instance the French second person singular present subjunctive is always found at position 17. In this way, word-paradigm pairs can be easily converted to morphogical lexica and to transducers that map form descriptions to surface forms and back. A properly designed set of paradigms permits a compact representation of a lexicon and a user-friendly way to extend it.

Different paradigm systems may have different numbers of paradigms. There are two reasons for this. One is that traditional paradigms often in fact require more arguments than one:

$$P : \text{String}^m \rightarrow \text{String}^n$$

Here $m \leq n$ and the set of arguments is a subset of the set of values. Thus the so-called fourth verb conjugation in Swedish actually needs three forms to work properly, for instance *sitta, satt, suttit* for the equivalent of *sit, sat, sat* in English. In Hellberg (1978), as in the French *Bescherelle*, each paradigm is defined to take exactly one argument, and hence each vowel alternation pattern must be a different paradigm.

The other factor that affects the number of paradigms is the nature of the string operations allowed in the function P. In Hellberg (1978), noun paradigms only permit the concatenation of suffixes to a stem. Thus the paradigms are identified with suffix sets. For instance, the inflection patterns *bil–bilar* ("car–cars") and *nyckel–nycklar* ("key–keys") are traditionally both treated as instances of the second declension, with the plural ending *ar* and the contraction of the unstressed *e* in the case of *nyckel.* But in Hellberg, the word *nyckel* has *nyck* as its "technical stem", to which the paradigm numbered 231 adds the singular ending *el* and the plural ending *lar*.

The notion of paradigm used in this paper allows multiple arguments and powerful string operations. In this way, we will be able to reduce the number of paradigms drastically: in fact, each lexical category (noun, adjective, verb), will have just *one* paradigm but with a variable number of arguments. Paradigms that follow this design will be called **smart paradigms** and are introduced in Section 2. Section 3 defines the notions of **predictability** and **complexity** of smart paradigm systems. Section 4 estimates these figures for four different languages of increasing richness in morphology: English, Swedish, French, and Finnish. We also evaluate the smart paradigms as a data compression method. Section 5 explores some uses of smart paradigms in lexicon building. Section 6 compares smart paradigms with related techniques such as morphology guessers and extraction tools. Section 7 concludes.

2 Smart paradigms

In this paper, we will assume a notion of paradigm that allows multiple arguments and arbitrary computable string operations. As argued in (Kaplan and Kay 1994) and amply demonstrated in (Beesley and Karttunen 2003), no generality is lost if the string operators are restricted to ones computable by finite-state transducers. Thus the examples of paradigms that we will show (only informally), can be converted to matching and replacements with regular expressions.

For example, a majority of French verbs can be defined by the following paradigm, which analyzes a variable-size suffix of the infinitive form and dispatches to the *Bescherelle* paradigms (identified by a number and an example verb):

mkV : String $\rightarrow$ String^{51} mkV(s) =

- conj19finir(s), if s ends *ir*
- conj53rendre(s), if s ends *re*
- conj14assiéger(s), if s ends *éger*
- conj11jeter(s), if s ends *eler* or *eter*
- conj10céder(s), if s ends *éder*
- conj07placer(s), if s ends *cer*
- conj08manger(s), if s ends *ger*
- conj16payer(s), if s ends *yer*

- conj06parler(s), if s ends *er*

Notice that the cases must be applied in the given order; for instance, the last case applies only to those verbs ending with *er* that are not matched by the earlier cases.

Also notice that the above paradigm is just like the more traditional ones, in the sense that we cannot be sure if it really applies to a given verb. For instance, the verb *partir* ends with *ir* and would hence receive the same inflection as *finir*; however, its real conjugation is number 26 in *Bescherelle*. That mkV uses 19 rather than number 26 has a good reason: a vast majority of *ir* verbs is inflected in this conjugation, and it is also the productive one, to which new *ir* verbs are added.

Even though there is no mathematical difference between the mkV paradigm and the traditional paradigms like those in *Bescherelle*, there is a reason to call mkV a **smart paradigm**. This name implies two things. First, a smart paradigm implements some “artificial intelligence” to pick the underlying “stupid” paradigm. Second, a smart paradigm uses heuristics (informed guessing) if string matching doesn’t decide the matter; the guess is informed by statistics of the distributions of different inflection classes.

One could thus say that smart paradigms are “second-order” or “meta-paradigms”, compared to more traditional ones. They implement a lot of linguistic knowledge and intelligence, and thereby enable tasks such as lexicon building to be performed with less expertise than before. For instance, instead of “07” for *foncer* and “06” for *marcher*, the lexicographer can simply write “mkV” for all verbs instead of choosing from 88 numbers.

In fact, just “V”, indicating that the word is a verb, will be enough, since the name of the paradigm depends only on the part of speech. This follows the model of many dictionaries and methods of language teaching, where **characteristic forms** are used instead of paradigm identifiers. For instance, another variant of mkV could use as its second argument the first person plural present indicative to decide whether an *ir* verb is in conjugation 19 or in 26:

mkV : $\text{String}^2 \rightarrow \text{String}^{51}$ mkV(s, t) =

- conj26partir(s), if for some x, $s = x{+}ir$ and $t = x{+}ons$
- conj19finir(s), if s ends with *ir*
- (all the other cases that can be recognized by this extra form)
- mkV(s) otherwise (fall-back to the one-argument paradigm)

In this way, a series of smart paradigms is built for each part of speech, with more and more arguments. The trick is to investigate which new forms have the best discriminating power. For ease of use, the paradigms should be displayed to the user in an easy to understand format, e.g. as a table specifying the possible argument lists:

verb	*parler*
verb	*parler, parlons*
verb	*parler, parlons, parlera, parla, parlé*
noun	*chien*
noun	*chien*, masculine
noun	*chien, chiens*, masculine

Notice that, for French nouns, the gender is listed as one of the pieces of information needed for lexicon building. In many cases, it can be inferred from the dictionary form just like the inflection; for instance, that most nouns ending *e* are feminine. A gender argument in the smart noun paradigm makes it possible to override this default behaviour.

2.1 Paradigms in GF

Smart paradigms as used in this paper have been implemented in the GF programming language (Grammatical Framework, (Ranta 2011)). GF is a functional programming language enriched with regular expressions. For instance, the following function implements a part of the one-argument French verb paradigm shown above. It uses a case expression to pattern match with the argument *s*; the pattern `_` matches anything, while `+` divides a string to two pieces, and `|` expresses alternation. The functions `conj19finir` etc. are defined elsewhere in the library. Function application is expressed without parentheses, by the juxtaposition of the function and the argument.

```
mkV : Str -> V
mkV s = case s of {
  _ + "ir" -> conj19finir s ;
  _ + ("eler"|"eter")
          -> conj11jeter s ;
  _ + "er" -> conj06parler s ;
  }
```

The GF Resource Grammar Library[11] has comprehensive smart paradigms for 18 languages: Amharic, Catalan, Danish, Dutch, English, Finnish, French, German, Hindi, Italian, Nepalese, Norwegian, Romanian, Russian, Spanish, Swedish, Turkish, and Urdu. A few other languages have complete sets of “traditional” inflection paradigms but no smart paradigms.

Six languages in the library have comprehensive morphological dictionaries: Bulgarian (53k lemmas), English (42k), Finnish (42k), French (92k), Swedish (43k), and Turkish (23k). They have been extracted from other high-quality resources via conversions to GF using the paradigm systems. In Section 4, four of them will be used for estimating the strength of the smart paradigms, that is, the predictability of each language.

3 Cost, predictability, and complexity

Given a language $\mathcal{L}$, a lexical category C, and a set P of smart paradigms for C, the **predictability** of the morphology of C in $\mathcal{L}$ by P depends inversely on the average number of arguments needed to generate the correct inflection table for a word. The lower the number, the more predictable the system.

Predictability can be estimated from a lexicon that contains such a set of tables. Formally, a smart paradigm is a family P_m of functions

[11] Source code and documentation in `http://www.grammaticalframework.org/lib`.

$$P_m : \text{String}^m \to \text{String}^n$$

where m ranges over some set of integers from 1 to n, but need not contain all those integers. A **lexicon** L is a finite set of inflection tables,

$$L = \{w_i : \text{String}^n \mid i = 1, \ldots, M_L\}$$

As the n is fixed, this is a lexicon specialized to one part of speech. A **word** is an element of the lexicon, that is, an inflection table of size n.

An **application** of a smart paradigm P_m to a word $w \in L$ is an inflection table resulting from applying P_m to the appropriate subset $\sigma_m(w)$ of the inflection table w,

$$P_m[w] = P_m(\sigma_m(w)) : \text{String}^n$$

Thus we assume that all arguments are existing word forms (rather than e.g. stems), or features such as the gender.

An application is **correct** if

$$P_m[w] = w$$

The **cost of a word** w is the minimum number of arguments needed to make the application correct:

$$\mathit{cost}(w) = \operatorname*{argmin}_m(P_m[w] = w)$$

For practical applications, it is useful to require P_m to be **monotonic**, in the sense that increasing m preserves correctness.

The **cost of a lexicon** L is the average cost for its words,

$$\text{cost}(L) = \frac{\sum_{i=1}^{M_L} \text{cost}(w_i)}{M_L}$$

where M_L is the number of words in the lexicon, as defined above.

The **predictability** of a lexicon could be defined as a quantity inversely dependent on its cost. For instance, an information-theoretic measure could be defined

$$\text{predict}(L) = \frac{1}{1 + \log \text{cost}(L)}$$

with the intuition that each added argument corresponds to a choice in a decision tree. However, we will not use this measure in this paper, but just the concrete cost.

The **complexity of a paradigm system** is defined as the size of its code in a given coding system, following the idea of Kolmogorov complexity (Solomonoff 1964a; Solomonoff 1964b). The notion assumes a coding system, which we fix to be GF source code. As the results are relative to the coding system, they are only usable for comparing definitions in the same system. However, using GF source code size rather than e.g. a finite automaton size gives in our view a better approximation of the "cognitive load" of the paradigm system, its "learnability". As a functional programming language, GF permits abstractions comparable to those available for human language learners, who don't need to learn the repetitive details of a finite automaton.

We define the **code complexity** as the size of the abstract syntax tree of the source code. This size is given as the number of nodes in the syntax tree; for instance,

- $\text{size}(f(x_1, \ldots, x_n)) = 1 + \sum_{i=1}^{n} \text{size}(x_i)$
- $\text{size}(s) = 1$, for a string literal s

Using the abstract syntax size makes it possible to ignore programmer-specific variation such as identifier size. Measurements of the GF Resource Grammar Library show that code size measured in this way is in average 20% of the size of source files in bytes. Thus a source file of 1 kB has the code complexity around 200 on the average.

Notice that code complexity is defined in a way that makes it into a straightforward generalization of the cost of a word as expressed in terms of paradigm applications in GF source code. The source code complexity of a paradigm application is

$$\text{size}(P_m[w]) = 1 + m$$

Thus the complexity for a word w is its cost plus one; the addition of one comes from the application node for the function P_m and corresponds to knowing the part of speech of the word.

4 Experimental results

We conducted experiments in four languages (English, Swedish, French and Finnish[12]), presented here in order of morphological richness. We used trusted full form lexica (i.e. lexica giving the complete inflection table of every word) to compute the predictability, as defined above, in terms of the smart paradigms in GF Resource Grammar Library.

We used a simple algorithm for computing the cost c of a lexicon L with a set P_m of smart paradigms:

```
• set $c := 0$
• for each word $w_i$ in $L$,
    – for each $m$ in growing order for which $P_m$ is defined:
      if $P_m[w] = w$, then $c := c + m$, else try with next $m$
• return c
```

The average cost is c divided by the size of L.

The procedure presupposes that it is always possible to get the correct inflection table. For this to be true, the smart paradigms must have a "worst case scenario" version that is

[12]This choice correspond to the set of language for which both comprehensive smart paradigms and morphological dictionaries were present in GF with the exception of Turkish, which was left out because of time constraints.

Table 4: Lexicon size and average cost for the nouns (N) and verbs (V) in four languages, with the percentage of words correctly inferred from one and two forma (i.e. $m = 1$ and $m \leq 2$, respectively).

Lexicon	**Forms**	**Entries**	**Cost**	$m = 1$	$m \leq 2$
Eng N	2	15,029	1.05	95%	100%
Eng V	5	5,692	1.21	84%	95%
Swe N	9	59,225	1.70	46%	92%
Swe V	20	4,789	1.13	97%	97%
Fre N	3	42,390	1.25	76%	99%
Fre V	51	6,851	1.27	92%	94%
Fin N	34	25,365	1.26	87%	97%
Fin V	102	10,355	1.09	96%	99%

able to generate all forms. In practice, this was not always the case but we checked that the number of problematic words is so small that it wouldn't be statistically significant. A typical problem word was the equivalent of the verb *be* in each language.

Another source of deviation is that a lexicon may have inflection tables with size deviating from the number n that normally defines a lexical category. Some words may be "defective", i.e. lack some forms (e.g. the singular form in "plurale tantum" words), whereas some words may have several variants for a given form (e.g. *learned* and *learnt* in English). We made no effort to predict defective words, but just ignored them. With variant forms, we treated a prediction as correct if it matched any of the variants.

The above algorithm can also be used for helping to select the optimal sets of characteristic forms; we used it in this way to select the first form of Swedish verbs and the second form of Finnish nouns.

The results are collected in Table 4. The sections below give more details of the experiment in each language.

4.1 English

As gold standard, we used the electronic version of the Oxford Advanced Learner's Dictionary of Current English[13] which contains about 40,000 root forms (about 70,000 word forms).

Nouns. We considered English nouns as having only two forms (singular and plural), excluding the genitive forms which can be considered to be clitics and are completely predictable. About one third of the nouns of the lexicon were not included in the experiment because one of the form was missing. The vast majority of the remaining 15,000 nouns are very regular, with predictable deviations such as *kiss, kisses* and *fly, flies* which can be easily predicted by the smart paradigm. With the average cost of 1.05, this was the most predictable lexicon in our experiment.

Verbs. Verbs are the most interesting category in English because they present the richest morphology. Indeed, as shown by Table 4, the cost for English verbs, 1.21, is

[13]available in electronic form at `http://www.eecs.qmul.ac.uk/~mpurver/software.html`

similar to what we got for morphologically richer languages.

4.2 Swedish

As gold standard, we used the SALDO lexicon (Borin, Forsberg, and Lönngren 2008).

Nouns. The noun inflection tables had 8 forms (singular/plural indefinite/definite nominative/genitive) plus a gender (uter/neuter). Swedish nouns are intrinsically very unpredictable, and there are many examples of homonyms falling under different paradigms (e.g. *val, val* "choice" vs. *val -valar* "whale"). The cost 1.70 is the highest of all the lexica considered. Of course, there may be room for improving the smart paradigm.

Verbs. The verbs had 20 forms, which included past participles. We ran two experiments, by choosing either the infinitive or the present indicative as the base form. In traditional Swedish grammar, the base form of the verb is considered to be the infinitive, e.g. *spela, leka* ("play" in two different senses). But this form doesn't distinguish between the "first" and the "second conjugation". However, the present indicative, here *spelar, leker*, does. Using it gives a predictive power 1.13 as opposed to 1.22 with the infinitive. Some modern dictionaries such as Lexin[14] therefore use the present indicative as the base form.

4.3 French

For French, we used the Morphalou morphological lexicon (Romary, Salmon-Alt, and Francopoulo 2004). As stated in the documentation[15] the current version of the lexicon (version 2.0) is not complete, and in particular, many entries are missing some or all inflected forms. So for those experiments we only included entries where all the necessary forms were presents.

Nouns: Nouns in French have two forms (singular and plural) and an intrinsic gender (masculine or feminine), which we also considered to be a part of the inflection table. Most of the unpredictability comes from the impossibility to guess the gender.

Verbs: The paradigms generate all of the simple (as opposed to compound) tenses given in traditional grammars such as the *Bescherelle.* Also the participles are generated. The auxiliary verb of compound tenses would be impossible to guess from morphological clues, and was left out of consideration.

4.4 Finnish

The Finnish gold standard was the KOTUS lexicon (Kotimaisten Kielten Tutkimuskeskus 2006). It has around 90,000 entries tagged with part of speech, 50 noun paradigms, and 30 verb paradigms. Some of these paradigms are rather abstract and powerful; for instance, grade alternation would multiply many of the paradigms by a factor of 10 to 20, if it was treated in a concatenative way. For instance, singular nominative-genitive pairs show alternations such as *talo–talon* ("house"), *katto–katon* ("roof"), *kanto–kannon* ("stub"),

[14]`http://lexin.nada.kth.se/lexin/`

[15]`http://www.cnrtl.fr/lexiques/morphalou/LMF-Morphalou.php`, accessed 2011–11–04

Table 5: Paradigm complexities for nouns and verbs in the four languages, computed as the syntax tree size of GF code.

language	**noun**	**verb**	**total**
English	403	837	991
Swedish	918	1039	1884
French	351	2193	2541
Finnish	4772	3343	6885

rako–raon (“crack”), and *sato–sadon* (“harvest”). All of these are treated with one and the same paradigm, which makes the KOTUS system relatively abstract.

The total number of forms of Finnish nouns and verbs is a question of definition. Koskenniemi (Koskenniemi 1983) reports 2000 for nouns and 12,000 for verbs, but most of these forms result by adding particles and possessive suffixes in an agglutinative way. The traditional number and case count for nouns gives 26, whereas for verbs the count is between 100 and 200, depending on how participles are counted. Notice that the definition of predictability used in this paper doesn’t depend on the number of forms produced (i.e. not on n but only on m); therefore we can simply ignore this question. However, the question is interesting if we think about paradigms as a data compression method (Section 4.5).

Nouns. Compound nouns are a problem for morphology prediction in Finnish, because inflection is sensitive to the vowel harmony and number of syllables, which depend on where the compound boundary goes. While many compounds are marked in KOTUS, we had to remove some compounds with unmarked boundaries. Another peculiarity was that adjectives were included in nouns; this is no problem since the inflection patterns are the same, if comparison forms are ignored. The figure 1.26 is better than the one reported in (Ranta 2008), which is 1.42; the reason is mainly that the current set of paradigms has a better coverage of three-syllable nouns.

Verbs. Even though more numerous in forms than nouns, Finnish verbs are highly predictable (1.09).

4.5 Complexity and data compression

The cost of a lexicon has an effect on learnability. For instance, even though Finnish words have ten or a hundred times more forms than English forms, these forms can be derived from roughly the same number of characteristic forms as in English. But this is of course just a part of the truth: it might still be that the paradigm system itself is much more complex in some languages than others.

Following the definitions of Section 3, we have counted the the complexity of the smart paradigm definitions for nouns and verbs in the different languages in the GF Resource Grammar Library. Notice that the total complexity of the system is lower than the sum of the parts, because many definitions (such as morphophonological transformations) are reused in different parts of speech. The results are in Table 5.

These figures suggest that Finnish indeed has a more complex morphology than French,

Table 6: Comparison between using bzip2 and paradigms+lexicon source as a compression method. Sizes in kB.

Lexicon	Fullform	bzip2	fullform/bzip2	Source	fullform/source
Eng N	264	99	2.7	135	2.0
Eng V	245	78	3.2	57	4.4
Swe N	6,243	1,380	4.5	1,207	5.3
Swe V	840	174	4.8	58	15
Fre N	952	277	3.4	450	2.2
Fre V	3,888	811	4.8	98	40
Fin N	11,295	2,165	5.2	343	34
Fin V	13,609	2,297	5.9	123	114

and English is the simplest. Of course, the paradigms were not implemented with such comparisons in mind, and it may happen that some of the differences come from different coding styles involved in the collaboratively built library. Measuring code syntax trees rather than source code text neutralizes some of this variation (Section 3).

Finally, we can estimate the power of smart paradigms as a data compression function. In a sense, a paradigm is a function designed for the very purpose of compressing a lexicon, and one can expect better compression than with generic tools such as bzip2. Table 6 shows the compression rates for the same full-form lexica as used in the predictability experiment (Table 4). The sizes are in kilobytes, where the code size for paradigms is calculated as the number of constructors multiplied by 5 (Section 3). The source lexicon size is a simple character count, similar to the full-form lexicon.

Unexpectedly, the compression rate of the paradigms improves as the number of forms in the full-form lexicon increases (see Table 4 for these numbers). For English and French nouns, bzip2 is actually better. But of course, unlike the paradigms, it also gives a global compression over all entries in the lexicon. Combining the two methods by applying bzip2 to the source code gives, for the Finnish verb lexicon, a file of 60 kB, which implies a joint compression rate of 227.

That the compression rates for the code can be higher than the numbers of forms in the full-form lexicon is explained by the fact that the generated forms are longer than the base forms. For instance, the full-form entry of the Finnish verb *uida* (“swim”) is 850 bytes, which means that the average form size is twice the size of the basic form.

5 Smart paradigms in lexicon building

Building a high-quality lexicon needs a lot of manual work. Traditionally, when one is not writing all the forms by hand (which would be almost impossible in languages with rich morphology), sets of paradigms are used that require the lexicographer to specify the base form of the word and an identifier for the paradigm to use. This has several usability problems: one has to remember all the paradigm identifiers and choose correctly from them.

Smart paradigm can make this task easier, even accessible to non-specialist, because of their ability to guess the most probable paradigm from a single base form. As shown by Table 4, this is more often correct than not, except for Swedish nouns. If this information is not enough, only a few more forms are needed, requiring only practical knowledge of the language. Usually (92% to 100% in Table 4), adding a second form ($m = 2$) is enough to cover all words. Then the best practice for lexicon writing might be always to give these two forms instead of just one.

Smart paradigms can also be used for an automatic bootstrapping of a list of base forms into a full form lexicon. As again shown by the last column of Table 4, one form alone can provide an excellent first approximation in most cases. What is more, it is often the case that uncaught words belong to a limited set of "irregular" words, such as the irregular verbs in many languages. All new words can then be safely inferred from the base form by using smart paradigms.

6 Related work

Smart paradigms were used for a study of Finnish morphology in (Ranta 2008). The present paper can be seen as a generalization of that experiment to more languages and with the notion of code complexity. Also the paradigms for Finnish are improved here (cf. Section 4.4 above).

Even though smart paradigm-like descriptions are common in language text books, there is to our knowledge no computational equivalent to the smart paradigms of GF. Finite state morphology systems often have a function called a **guesser**, which, given a word form, tries to guess either the paradigm this form belongs to or the dictionary form (or both). A typical guesser differs from a smart paradigms in that it does not make it possible to correct the result by giving more forms. Examples of guessers include (Chanod and Tapanainen 1995) for French, (Hlaváčová 2001) for Czech, and (Nakov et al. 2004) for German.

Another related domain is the unsupervised learning of morphology where machine learning is used to automatically build a language morphology from corpora (Goldsmith 2006). The main difference is that with the smart paradigms, the paradigms and the guess heuristics are implemented manually and with a high certainty; in unsupervised learning of morphology the paradigms are induced from the input forms with much lower certainty. Of particular interest are (Chan 2006) and (Dreyer and Eisner 2011), dealing with the automatic extraction of paradigms from text and investigate how good these can become. The main contrast is, again, that our work deals with hand-written paradigms that are correct by design, and we try to see how much information we can drop before losing correctness.

Once given, a set of paradigms can be used in automated lexicon extraction from raw data, as in (Forsberg, Hammarström, and Ranta 2006) and (Clément, Sagot, and Lang 2004), by a method that tries to collect a sufficient number of forms to determine that a word belongs to a certain paradigm. Smart paradigms can then give the method to actually construct the full inflection tables from the characteristic forms.

7 Conclusion

We have introduced the notion of smart paradigms, which implement the linguistic knowledge involved in inferring the inflection of words. We have used the paradigms to estimate the predictability of nouns and verbs in English, Swedish, French, and Finnish. The main result is that, with the paradigms used, less than two forms in average is always enough. In half of the languages and categories, one form is enough to predict more than 90% of forms correctly. This gives a promise for both manual lexicon building and automatic bootstrapping of lexicon from word lists.

To estimate the overall complexity of inflection systems, we have also measured the size of the source code for the paradigm systems. Unsurprisingly, Finnish is around seven times as complex as English, and around three times as complex as Swedish and French. But this cost is amortized when big lexica are built.

Finally, we looked at smart paradigms as a data compression method. With simple morphologies, such as English nouns, bzip2 gave a better compression of the lexicon than the source code using paradigms. But with Finnish verbs, the compression rate was almost 20 times higher with paradigms than with bzip2.

The general conclusion is that smart paradigms are a good investment when building morphological lexica, as they ease the task of both human lexicographers and automatic bootstrapping methods. They also suggest a method to assess the complexity and learnability of languages, related to Kolmogorov complexity. The results in the current paper are just preliminary in this respect, since they might still tell more about particular implementations of paradigms than about the languages themselves.

Acknowledgements

We are grateful to the anonymous referees for valuable remarks and questions. The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007–2013) under grant agreement no FP7-ICT-247914 (the MOLTO project).

References

Beesley, K. R. and L. Karttunen (2003). *Finite State Morphology.* CSLI Publications.

Bescherelle (1997). *La conjugaison pour tous.* Hatier.

Borin, L., M. Forsberg, and L. Lönngren (2008). "SALDO 1.0 (Svenskt associationslexikon version 2)". In: *Sprakbanken.*

Chan, E. (2006). "Learning probabilistic paradigms for morphology in a latent class model". In: *Proceedings of the Eighth Meeting of the ACL Special Interest Group on Computational Phonology and Morphology.* SIGPHON '06. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 69–78. DOI: 10.3115/1622165.1622174.

Chanod, J.-P. and P. Tapanainen (1995). "Creating a tagset, lexicon and guesser for a French tagger". In: *CoRR.*

Clément, L., B. Sagot, and B. Lang (2004). "Morphology based automatic acquisition of large-coverage lexica". In: *Proceedings of LREC-04, Lisboa, Portugal*, pp. 1841–1844.

Dreyer, M. and J. Eisner (2011). "Discovering morphological paradigms from plain text using a Dirichlet process mixture model". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. EMNLP '11. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 616–627. ISBN: 978-1-937284-11-4.

Forsberg, M., H. Hammarström, and A. Ranta (2006). "Morphological Lexicon Extraction from Raw Text Data". In: *FinTAL 2006*. Ed. by T. Salakoski. Vol. 4139. LNCS/LNAI. DOI: 10.1007/11816508_49.

Goldsmith, J. (2006). "An Algorithm for the Unsupervised Learning of Morphology". In: *Natural Langage Engineering* 12.4, pp. 353–371. ISSN: 1351-3249. DOI: 10.1017/S1351324905004055.

Hellberg, S. (1978). *The Morphology of Present-Day Swedish*. Almqvist & Wiksell.

Hlaváčová, J. (2001). "Morphological Guesser of Czech Words". In: *Text, Speech and Dialogue*. Ed. by V. Matoušek et al. Vol. 2166. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 70–75. DOI: 10.1007/3-540-44805-5_9.

Kaplan, R. and M. Kay (1994). "Regular Models of Phonological Rule Systems". In: *Computational Linguistics* 20, 331–380.

Koskenniemi, K. (1983). "Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production". PhD thesis. University of Helsinki.

Kotimaisten Kielten Tutkimuskeskus (2006). *KOTUS Wordlist*. URL: http://kaino.kotus.fi/sanat/nykysuomi.

Nakov, P. et al. (2004). "Guessing morphological classes of unknown German nouns". In: *Recent Advances in Natural Language Processing III*. John Benjamins Publishing Company, p. 347. DOI: 10.1075/cilt.260.39nak.

Ranta, A. (2008). "How predictable is Finnish morphology? An experiment on lexicon construction". In: *Resourceful Language Technology: Festschrift in Honor of Anna Sågvall Hein*. Ed. by J. Nivre, M. Dahllöf, and B. Megyesi. University of Uppsala, pp. 130–148.

— (2011). *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications. ISBN: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

Romary, L., S. Salmon-Alt, and G. Francopoulo (2004). "Standards going concrete: from LMF to Morphalou". English. In: *The 20th International Conference on Computational Linguistics - COLING 2004*. coling. Geneva, Switzerland.

Solomonoff, R. J. (1964a). "A formal theory of inductive inference: Part I". In: *Information and Control* 7 (1), pp. 1–22. DOI: 10.1016/s0019-9958(64)90223-2.

— (1964b). "A formal theory of inductive inference: Part II". In: *Information and Control* 7 (2), pp. 224–254. DOI: 10.1016/s0019-9958(64)90131-7.

Paper B

Sharing resources between free/open-source rule-based machine translation systems: Grammatical Framework and Apertium

Abstract

In this paper, we describe two methods developed for sharing linguistic data between two free and open source rule based machine translation systems: Apertium, a shallow-transfer system; and Grammatical Framework (GF), which performs a deeper syntactic transfer. In the first method, we describe the conversion of lexical data from Apertium to GF, while in the second one we automatically extract Apertium shallow-transfer rules from a GF bilingual grammar. We evaluated the resulting systems in a English-Spanish translation context, and results showed the usefulness of the resource sharing and confirmed the a-priori strong and weak points of the systems involved.

Keywords: rule-based machine translation, linguistic resource sharing, open-source linguistic resources

1 Introduction

Machine Translation (MT) can be defined as the use of software to translate content from one natural language, the source language (SL), into another, the target language (TL). Two main MT paradigms can be established according to the kind of knowledge involved in the translation process.

On the one hand, corpus-based approaches use large parallel corpora as the source of knowledge. A parallel corpus is a collection of parallel texts, that is, texts in one language together with their translation into another language. The statistical machine translation (SMT; Koehn 2010) corpus-based approach is currently the leading paradigm in MT. SMT systems can be built with little human effort, provided that a large enough parallel corpus is available.

Rule-based machine translation (RBMT; Hutchins and Somers 1992) systems on the other hand are best characterized by their use of explicit linguistic knowledge. This knowledge can take many forms, from simple monolingual dictionaries to complex semantic structures but it usually needs to be manually encoded by experts, which represents a big part of the effort needed to create such systems and has a great influence on their overall performance.

Among the different RBMT approaches, transfer-based systems are those in which the translation process can be split in the following three steps: they perform an analysis of the SL text into an SL intermediate representation; after that, the intermediate representation is transferred to the TL; and finally the translation is generated from the TL intermediate representation.

Software licensed as Open Source allows anyone to study, change and distribute the software to anyone and for any purpose. In the case of RBMT systems, this creates the possibility of reusing the linguistic knowledge encoded in one system to create, or at least bootstrap, a different system.

In this paper, we started exploring the many possible ways for sharing linguistic data between the free/open-source RBMT systems Apertium (Forcada et al. 2011) and Grammatical Framework (GF, Ranta 2011) with two new methods. Apertium is a shallow-transfer system, which means that it does not perform a full syntactic analysis to build the intermediate representation. Contrarily, GF is a multilingual grammar formalism that has been used to build MT systems, among other applications. The methods we developed allowed us to create two RBMT systems from the same resources and do an empirical comparison between Apertium and GF, analyzing their strong and weak points.

Previous strategies to share Apertium or GF linguistic resources include using Apertium data to enrich statistical machine translation (Tyers 2009; Sánchez-Cartagena, Sánchez-Martínez, and Pérez-Ortiz 2011) and example-based systems (Sánchez-Martínez, Forcada, and Way 2009), and combining SMT systems with GF (Enache et al. 2012). Resource sharing between Apertium and GF has however never been explored.

2 Integration

We have developed two sharing strategies: augmenting the GF lexicon with entries from an Apertium dictionary, and creating Apertium shallow-transfer rules from GF grammars. They are described in this section together with the main differences between GF and Apertium.

2.1 Differences between GF and Apertium

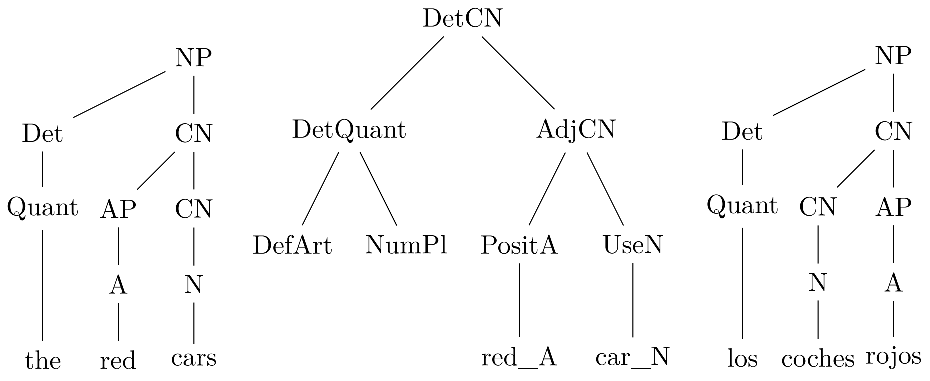
GF (Grammatical Framework, Ranta 2011) is a multilingual grammatical formalism, and the key point of its design is the separation of a language independent abstract syntax from multiple concrete syntaxes. GF also provides the Resource Grammar Library (RGL; Ranta 2009), a model of the low level structures of syntax and morphology for (at the time of this writing) 29 natural languages. Thanks to the RGL, When writing a domain-specific grammar, one needs only to concentrate on the abstract syntax for her domain, leaving the tedious linguistic details to the library. However, the linguistic information of the RGL can also be exploited to perform open-domain translation by using the common API of the RGL as a pivot (Figure 3), which is the configuration explored in this paper. In a taxonomy of MT according to the abstraction level of the intermediate representation, this last configuration can be seen as a form of syntactic transfer.

Unlike GF, the Apertium shallow-transfer RBMT platform (Forcada et al. 2011) was initially designed for open-domain translation. Apertium uses a simple, flat, intermediate representation: a sequence of *lexical forms* representing the lemma, lexical category and morphological inflection information of the words to be translated. E.g.:

the.**det**.def *red*.**adj** *car*.**n**.pl

The translation between the source-language (SL) and the target-language (TL) lexical forms is carried out by a set of shallow-transfer rules performing operations such as agreements, re-orderings, preposition changes, etc (see Figure 4). Each rule processes a chunk of lexical forms and they are applied in a greedy manner.

While GF guarantees a grammatically correct output and allows more sophisticated transformations (e.g. long-distance re-orderings), the quality of the translation drops when the sentence cannot be fully parsed because out-of-vocabulary words or an irregular grammatical structure. Although, for such sentences, GF generates some partial



(a) *English concrete tree obtained after parsing* the red cars

(b) *Abstract syntax tree*

(c) *Spanish concrete tree which is linearized to* los coches rojos

Figure 3: *Example of parse trees in GF when performing an English-Spanish open-domain translation*

subtrees (Angelov 2011), the shallow-transfer approach followed by Apertium allows for more robustness.

The 37 language pairs supported by Apertium include 17 languages not present in GF RGL (Aragonese, Asturian, Basque, Breton, Galician, Icelandic, Indonesian, Kazakh, Macedonian, Mataysian, North, Nynorsk, Occitan, Portugese, Serbo-Croatian, Slovenian, Sámi, Tatar and Welsh[16]), while the RGL contains data for 23 languages (Amahric, Chinese, Estonian, Finnish, German, Greek, Hebrew, Hindi, Japanese, Latin, Latvian, Mongolian, Nepali, Persian, Punjabi, Polish, Russian, Sundhi, Swahili, Thai, Tswana, Turkish and Urdu[17]) and more than 700 language pairs not yet present in Apertium. This shows the potential advantages of sharing resources between Apertium and GF.

2.2 Augmenting the GF lexicon with Apertium data

Lexica in RBMT contain the analysis of each word the system is able to translate or generate, and mappings between analyzed forms in different languages. In Apertium and GF, inflection paradigms are used to efficiently encode them.

The first system in our comparison was based on the GF Resource Grammar Library, in which lexicon entries from open lexical categories have been replaced with the information from the Apertium dictionaries. Although the GF Resource Grammar Library already contained a huge English lexicon, for the purposes of this work we only included in the resulting system the entries from closed lexical categories. Regarding Spanish, the GF lexicon contains all the words from closed lexical categories but only a few words from

[16]List of stable language pairs. Retrieved October 21, 2013, from `http://wiki.apertium.org/wiki/Main_Page`.

[17]The Status of the GF Resource Grammar Library. Retrieved October 21, 2013, from `http://www.grammaticalframework.org/lib/doc/status.html`.

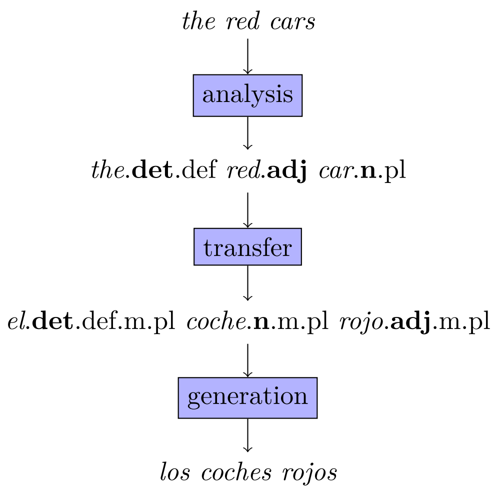

the red cars
analysis
the.det.def red.adj car.n.pl
transfer
el.det.def.m.pl coche.n.m.pl rojo.adj.m.pl
generation
los coches rojos

Figure 4: *Lexical forms produced by the Apertium engine when translating the English phrase the red cars into Spanish. Lemmas are shown in italics and lexical categories in bold. det stands for determiner, adj means adjective and n means noun. The determiner is definite (def), the gender is masculine (m) and the noun is plural (pl).*

open categories. The latter were removed too and replaced with the ones from Apertium.

For many lexicon entries, porting them from Apertium to GF simply meant dealing with the different encoding details of both systems. First, we expanded the Apertium SL monolingual dictionary entries (i.e., to apply the corresponding paradigm to the stem to generate all word forms) and created a new entry in the GF SL lexicon for each of them by providing to a smart paradigm (Détrez and Ranta 2012) all the expanded forms.

For instance, the following was the entry for the English noun *car* in the Apertium monolingual dictionary.

```
<e><i>car</i><par n="house__n"/></e>
```

The entry indicates that the noun *car* is inflected in the same way *house* is (the plural form is built by adding -s, and when adding the genitive marker to it, the suffix becomes -s'.) In that case, the result of the expension was:

```
car:car.n.sg
cars:car.n.pl
car’s:car.n.sg.gen
cars’:car.n.pl.gen
```

And the resulting GF entry, using the smart paradigm mkN for nouns:

```
lin car_N = mkN "car" "cars" "car's" "cars’'" ;
```

The same process was repeated for the target language. In our example, the entry for the translated lexeme in the target language was:

```
<e><i>coche</i><par n="abismo__n"/></e>
```

Which was converted to:

```
lin car_N = mkN "coche" "coches" masculine ;
```

Note that in both cases the GF function is named *car_N* instead of *coche_N* in the target language. This was necessary to map the entry to its source language equivalent and it was achieved by looking up the lemma in Apertium's bilingual dictionary.

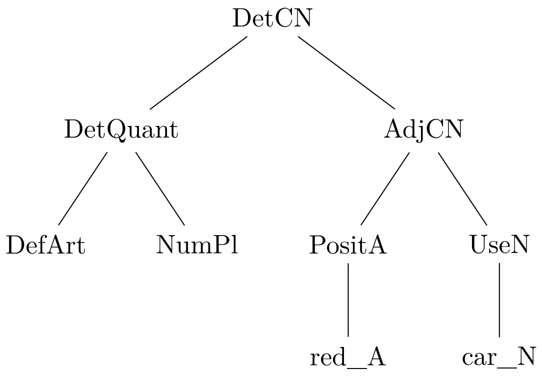
This simple startegy was made possible by the design of GF's smart paradigms (Détrez and Ranta 2012) which allowed the creation of a valid lexicon entry giving only partial information, the missing forms and parameters being infered using the language morphology. For instance, adjective entries in the English GF lexicon had their adverbial form attached, while Apertium had separated entries for adjectives and adverbs. When porting English adjectives from Apertium to GF, we let the GF smart paradigms infer the adverbial form of the adjective. An other example, still regarding adjectives: Spanish adjectives are usually placed after the noun they modify, but a few of them, called prepositive adjectives, are placed after the noun. This feature was needed in the GF lexicon, but was not present in the Apertium one. As a consequence, when including Spanish adjectives from Apertium in the GF lexicon, we could not provide any information about this to the smart paradigms. When there was not enough information, the smart paradigms chose the most common option: in this case, that the adjective was not prepositive. Finally, English nouns contain a humanity feature in GF, which is not encoded in Apertium and in this case, all Apertium nouns were imported as non-human (the most common value).

In addition, since GF uses a deeper intermediate representation, some additional linguistic information was required when inserting certain entries in the GF lexicon. In particular, in the case of verbs, the GF lexicon contains valency information used in parsing. For instance, the valency V indicates an intransitive verb (for example: *run*), V2 a transitive verb (*hit*), VA states that a verb is complemented by an adjective (*become*) and so on. Since it was not possible to infer the verb valencies we imported them from the existing GF English lexicon[18] and used them for both English and Spanish verbs (in the RGL API, the valency is encoded in the language-independent abstract syntax, so it is necessarily the same for linearization of the same abstract function.)

2.3 Generating Apertium shallow-transfer rules from GF data

The second system used the Apertium engine and lexicon but we extracted structural transfer rules from the GF resource grammar library. The Apertium shallow-transfer rules process fixed-length chunks of lexical forms and perform agreements, re-orderings, preposition changes and other grammatical transformations. Naturally, since Apertium does not perform a full parsing, we could not simply re-encode the gf grammars into Apertium rules. Instead, we developed a method based on flattening abstract syntax trees.

[18]The GF RGL originally contains over 60.000 entries in the English lexicon, but only a few dozens in the Spanish one.



(a) *GF abstract syntax tree*

the.**det**.def *red*.**adj** *car*.**n**.pl → *el*.**det**.def.m.pl *coche*.**n**.m.pl *rojo*.**adj**.m.pl

(b) *Its linearization into English (left) and Spanish (right) when replacing the linearization of terminal symbols with Apertium lexical forms*

the.**det**.def **adj** **n**.pl|m.pl → *el*.**det**.def.m.pl **n**.m.pl **adj**.m.pl

(c) *Same example with nouns and adjectives replaced by word classes.*

the.**det**.def **adj** **n**.pl|m.pl → *el*.**det**.def.m.pl \$3.**n**.m.pl \$2.**adj**.m.pl

(d) *Apertium rule extracted from it. The rule matches the definite determiner* the*, followed by any adjective and a plural noun whose gender after being looked up in the bilingual lexicon is masculine, and its number is plural.The expression \$i means that the lemma is obtained by looking up in the bilingual lexicon the i-th matching SL lexical form*

Figure 5: *Steps carried out obtain an Apertium shallow transfer rule from a GF abstract syntax tree.*

In a nutshell, our strategy involved generating, for each GF abstract syntax function from the Resource Grammar Library, all the possible abstract trees which could be built (up to a certain depth). Each tree was then linearized in both SL and TL to obtain a bilingual phrase (the GF engine provides the word-by-word alignment), and an Apertium shallow-transfer rule was extracted from the pair of linearizations using the algorithm developed by Sánchez-Martínez and Forcada (Sánchez-Martínez and Forcada 2009).

The depth limit was important to obtain a finite and manageable set of abstract syntax trees. In order to avoid the generation of an unmanageable set of bilingual phrases, we also took advantage of the fact that in GF the grammar rules are not influenced by a word form but only by the features (e.g. two masculine nouns will appear in exactly the same set of trees) Thus, for each open lexical category, only one for each combination of SL and TL features was included in the GF lexicon used to generate bilingual phrases. For instance, with regard to common nouns when translating from English to Spanish, the only relevant feature for translation was the gender in Spanish. Consequently, we only needed to include in the lexicon a noun which is masculine in Spanish (such as *car*), and another one which is feminine (for instance, *house*). In addition, other modifications were carried out to ensure that Apertium shallow-transfer rules could be obtained from the pair of linearizations. First, the linearization of each GF lexical function was replaced by its corresponding Apertium lexical form. In this way, pairs of lexical form sequences were obtained when linearizing the abstract function trees. These pairs could then be directly converted into Apertium shallow-transfer rules.

However, the approach which has just been described would generate a vast amount of rules, since a rule for each combination of lexical entries would be obtained. Since it was desirable to obtain a smaller set of rules, we performed a further modification: the introduction of word classes. We modified again the GF lexicon, in which previously the original entries have been replaced with Apertium lexical forms, and replaced lexical forms from open lexical categories with word classes. The word class of an SL lexical form is defined as the concatenation of its lexical category, morphological inflection information and morphological inflection information obtained when looking it up in the Apertium bilingual dictionary. Word classes group together words which behave in the same way when being translated. For example, the word class of lexical forms such as *car*.**n**.pl, *phone*.**n**.pl, or *day*.**n**.pl is **n**.pl|**m**.pl, since all of them are plural nouns in English which are translated as masculine plural nouns in Spanish. Word classes of TL lexical forms only contain the lexical category and TL morphological inflection information (the bilingual dictionary is not involved). Figure 5c shows the pair of linearizations from the tree presented in figure 5a, in which lexical forms have been replaced by word classes.

Although the pair of lexical form sequences just shown is more similar to an actual Apertium rule than the previous examples, one detail remain to be fixed: alignments were needed in order match SL and TL word classes and allow the Apertium engine to collect the lemmas of the TL word classes by looking up in the bilingual dictionary the corresponding SL words. Fortunately, the GF engine provides them. The final Apertium rule obtained is depicted in figure 5d. This rule matches the definite determiner *the*, followed by an adjective and a plural noun which is masculine and plural in Spanish, and generates, in Spanish, a definite, masculine, plural determiner; a masculine plural noun whose lemma is obtained by looking up in the bilingual dictionary the lemma of the

Corpus	System	BLEU	METEOR	TER
newstest2011A	*sharedGF*	0.027	0.181	0.847
	sharedApertium	<u>**0.138**</u>	<u>**0.390**</u>	<u>**0.678**</u>
	Apertium word-for-word	0.111	0.368	0.703
	Apertium	0.200	0.443	0.617
newstest2011B	*sharedGF*	<u>0.152</u>	<u>0.388</u>	<u>0.703</u>
	sharedApertium	<u>0.148</u>	<u>0.391</u>	<u>0.691</u>
	Apertium word-for-word	0.106	0.361	0.713
	Apertium	0.212	0.451	0.620

Table 7: Values of the evaluation metrics obtained by the different English-Spanish MT systems. A score in bold for *sharedApertium* means that it outperforms *sharedGF* by a statistically significant margin computed by paired bootstrap resampling (Koehn 2004) with $p = 0.05$. An underlined score indicate that the system outperforms Apertium word-by-word translation, according to the same criterion.

English noun, and a masculine plural adjective whose lemma is obtained by looking up in the bilingual dictionary the lemma of the English adjective.

3 Evaluation

We used the methods described above to build two English-Spanish MT systems stemming from the same resources: the Apertium lexicon and the GF RGL. *sharedApertium* is an Apertium-based system containing the original Apertium lexicon and a set of shallow-transfer rules created from the GF RGL, while *sharedGF* is a GF-based system in which the lexicon has been ported from Apertium.

We performed an automatic evaluation using two subsets of the *newstest2011*[19] set. We computed BLEU (Papineni et al. 2002), METEOR (Lavie and Agarwal 2005) and TER (Snover et al. 2006) scores for the aforementioned systems, along with out-of-the-box Apertium and a word-for-word translation with the Apertium lexicon. The subset *newstest2011A* (1896 sentences) contains the parallel sentences from newstest2011 which can be parsed (either fully or partially) by GF in a reasonable time, while *newstest2011B* (130 sentences) contains only those fully parsed by GF. Results are shown in Table 7.

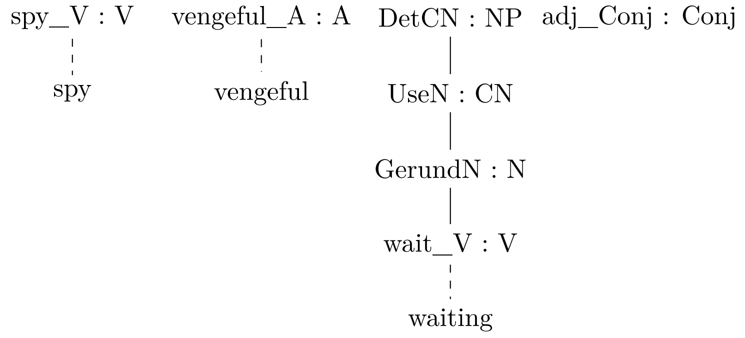
The most remarkable conclusion that can be drawn from the results is that our resource sharing strategies eased the development of new RBMT systems: an Apertium-based system which outperformed word-for-word translation has been created without manually writing a single shallow-transfer rule; and a GF-based system, which also outperformed Apertium word-for-word translation on the smallest corpus, has been built despite that the GF lexicon only contained originally a few entries in the Spanish side.

Regarding the differences between *sharedGF* and *sharedApertium*, GF performed poorly on the bigger, *newstest2011A* corpus, mainly due to out-of-vocabulary words and

[19]Distributed as part of the WMT 2011 shared translation task: `http://www.statmt.org/wmt11/translation-task.html`

out-of-grammar constructions. These are less of an issue for Apertium, which simply translates word-by-word when no rule matches the input chunks. An example of this situation is presented in Figure 6.

source: Vengeful **hackers** and spies **are** waiting



sharedGF:	vengativo espiar
sharedApertium:	Vengativo **hackers** y espías **son** esperando

Figure 6: *SL sentence from the newstest2011A evaluation corpus, partial parse trees obtained by the GF parser of the sharedGF system, their translation, and the translation of the same SL sentence by sharedApertium. Observe that the out-of-vocabulary word hackers prevents GF from fully parsing the sentence.*

sharedGF catches up with *sharedApertium* on the smaller, fully-parsed, evaluation corpus. As pointed out previously, analyzing the whole sentences allowed GF to perform more accurate translations than Apertium for some constructions, as in the example presented in figure 7. The GF parser was also able to automatically detect named entities, which lead to the correct translation of Saxon genitives even when the proper noun was not in the lexicon. See Figure 8 for an example.

source:	But the man remains (V) skeptic
sharedGF:	Pero el hombre queda (V) escéptico
sharedApertium:	Pero el hombre restos (N) escéptico
source:	The matter examines the type of damage (N)
sharedGF:	El asunto examina el tipo de daño (N)
sharedApertium:	El asunto examina el tipo de averiar (V)

Figure 7: *SL sentences from the newstest2011A evaluation corpus and their translation with the systems being evaluated. Lexical category is shown in parentheses.*

However, even when the sentence was fully parsed, the GF-based system had some drawbacks when compared to Apertium. For instance, the Apertium analyzer correctly handled most multi-word expressions encoded in the lexicon because it always tries to match the longest possible segments, but the GF parser relies on statistics to choose among

source:	This is Dan Brown's success mechanism
sharedGF:	Éste es mecanismo de éxito de Dan Brown
sharedApertium:	Esto es Dan Brown mecanismo de éxito

Figure 8: *SL sentence from the newstest2011B evaluation corpus and its translation with the systems being evaluated.*

the possible parse trees, which could lead to situations such as the one shown in Figure 9. One could remedy to this problem by tuning the probability in the GF grammar—either manually or using treebank data—so that the idiosyncratic interpretation is chosen over the compositional one. On the other hand, GF could also analyze discontinuous multiword expressions which cannot be encoded in Apertium's lexicon.

source:	An extra portion of sugar is needed for lemon ice cream
sharedGF:	Una porción extra de azúcar está necesitada para crema de hielo de limón
sharedApertium:	Una porción extra de azúcar es necesitar para limón helado

Figure 9: *SL sentence from the newstest2011B evaluation corpus and its translation with the systems being evaluated. The right translation into Spanish of the multi-word expression ice cream is helado, while crema de hielo is the literal translation.*

4 Conclusions and future work

We have presented two strategies for sharing linguistic resources between Apertium and GF and used them to create two RBMT systems stemming from the same linguistic resources. Our experiments showed the usefulness of the resource sharing and confirmed the a-priori strong and weak points of the systems involved.

Possible future works include exploring other ways to share Apertium and GF's resources. For instance, porting the GF lexicon to Apertium or using GF smart paradigms (Détrez and Ranta 2012) to ease the creation of the Apertium lexicon. A deeper integration of Apertium and GF could also be achieved by combining them at runtime, following a approach similar to the strategy designed to integrate GF and SMT(Enache et al. 2012).

5 Acknowledgements

Work partially supported by the Spanish government through project TIN2012–32615 and by Generalitat Valenciana through grant ACIF/2010/174 and Swedish Research Council through financial support under grant nr. 2012–5746 (Reliable Multilingual Digital Communication: Methods and Applications).

References

Angelov, K. (2011). “The Mechanics of the Grammatical Framework”. PhD thesis. ISBN: 978-91-7385-605-8.

Détrez, G. and A. Ranta (2012). “Smart Paradigms and the Predictability and Complexity of Inflectional Morphology”. In: *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, pp. 645–653.

Enache, R. et al. (2012). “A Hybrid System for Patent Translation”. In: *The 16th Annual Conference of the European Association for Machine Translation*. Trento, Italy, pp. 269–276.

Forcada, M. L. et al. (2011). “Apertium: a free/open-source platform for rule-based machine translation”. In: *Machine Translation* 25.2. Special Issue: Free/Open-Source Machine Translation, pp. 127–144. DOI: `10.1007/s10590-011-9090-0`.

Hutchins, W. and H. Somers (1992). *An introduction to machine translation*. Vol. 362. Academic Press New York.

Koehn, P. (2004). “Statistical significance tests for machine translation evaluation”. In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Vol. 4, pp. 388–395.

Koehn, P. (2010). *Statistical Machine Translation*. 1st. New York, NY, USA: Cambridge University Press. ISBN: 0521874157, 9780521874151.

Lavie, A. and A. Agarwal (2005). “METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments”. In: *Proceedings of the Second Workshop on Statistical Machine Translation - StatMT ’07*. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 65–72. DOI: `10.3115/1626355.1626389`.

Papineni, K. et al. (2002). “BLEU: a Method for Automatic Evaluation of Machine Translation”. In: *ACL ’02: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 311–318. DOI: `10.3115/1073083.1073135`.

Ranta, A. (2009). “The GF Resource Grammar Library”. In: *Linguistic Issues in Language Technology* 2.2.

— (2011). *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications. ISBN: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

Sánchez-Cartagena, V. M., F. Sánchez-Martínez, and J. A. Pérez-Ortiz (2011). “Integrating shallow-transfer rules into phrase-based statistical machine translation”. In: *Proceedings of the XIII Machine Translation Summit*. Xiamen, China, pp. 562–569.

Sánchez-Martínez, F. and M. L. Forcada (2009). “Inferring shallow-transfer machine translation rules from small parallel corpora”. In: *Journal of Artificial Intelligence Research* 34.1, pp. 605–635.

Sánchez-Martínez, F., M. L. Forcada, and A. Way (2009). “Hybrid rule-based – example-based MT: Feeding Apertium with sub-sentential translation units”. In: *Proceedings of the 3rd Workshop on Example-Based Machine Translation*. Ed. by M. L. Forcada and A. Way. Dublin, Ireland, pp. 11–18.

Snover, M. et al. (2006). “A study of translation edit rate with targeted human annotation”. In: *In Proceedings of Association for Machine Translation in the Americas*, pp. 223–231.

Tyers, F. M. (2009). “Rule-based augmentation of training data in Breton-French statistical machine translation”. In: *Proceedings of the 13th Annual Conference of the European Association for Machine Translation.* Ed. by L. Màrquez and H. Somers, pp. 213–217.

Paper C

Learning Smart Paradigms

Abstract

Lexicons are an important component of many applications upon which we have come to rely, from word processors to machine translation systems. The lexicon builder traditionally creates a lexicon by defining a set of morphological paradigms and then associates each lemma in the language with one of those paradigms. This task can be accomplished more efficiently via constructs such as smart paradigms; functions that take lemmas of a particular part of speech and produce correct inflection tables according to the rules of the target language. This not only speeds the work of the lexicon builder, who can simply list lemmas with their respective part of speech without having to manually assign a paradigm to each lemma, but it also makes lexicon building accessible to those lacking detailed knowledge of the underlying morphology. Here we demonstrate an approach to learning smart paradigms from existing sets of paradigms. Our results show that learned smart paradigms perform similarly to manually written equivalents. Further we show that the method provides benefits even if the existing lexicon is small, indicating that the method can be useful from an early point in the lexicon building process.

1 Introduction

From simple spell-checking applications to state-of-the-art machine translation systems, lexicons are an essential ingredient of many natural language processing tasks, but their creation and maintenance remain labor-intensive and costly. Even though large lexicons may exist for the major languages of the world, they often need to be adapted to new usages or even re-created for legal or technical reasons (e.g., a lexicon distributed under a restrictive license cannot be used in a free/libre spell checker). Finally, many communities using less privileged languages cannot afford the creation of those resources, which prevents those languages from competing on an equal ground in an increasingly digitalized world. Which may even accelerate language disappearance.

Many solutions have been proposed (see Section 5). Here we concentrate on one, in particular, the GF smart paradigms (Ranta 2009; Ranta 2011). A smart paradigm is a function that infers full inflection tables from one or a few forms, thus easing the creation and maintenance of morphological lexicons by hiding the details of the underlying morphology. But smart paradigms have until now been built by hand, which requires skills both in linguistics and computer programming.

We explore a way to automatically create smart paradigms from existing lexicon data using machine learning techniques, and we evaluate their accuracy and usefulness on four different languages: Finnish, French, German and Maltese.

More precisely, the contributions of this paper are three fold:

- we introduce a method for constructing a smart paradigm from existing lexicon entries;
- we evaluate our approach on different lexicons and show that we can achieve similar results as manually created smart paradigms; and

- we evaluate the usefulness of this approach in lexicon building by showing that even with a small amount of data the learned smart paradigm can provide some assistance to the lexicon builder.

2 Background

We first define important concepts used in this paper: morphological lexicon (§ 2.1), paradigm (§ 2.2) and smart paradigm (§ 2.3).

2.1 Morphological lexicon

A morphological lexicon is a structure in which each word, or more precisely each lexeme, is associated with all possible inflectional forms that it can take. For instance, a morphological lexicon for English verbs gives seven forms for most entries: infinitive, imperfect, present, third person singular present, past, past participle and present participle. Lexemes in a morphological lexicon are identified by a *lemma*, which is most often one of the inflected forms that has been chosen to represent the whole lexeme.

The word forms of a lexeme may be classified according to shared inflection categories and arranged into tables (see for instance the Latin forms for LŪDUS 'game' given in Table 8). We build upon this tradition by defining an *inflection table* as the n-tuple $String^n$ of word forms for a lexeme, where each slot in the tuple corresponds to one or more combinations of values for each inflectional category. The choice of the relevant inflectional categories and their possible values (and thus the size n of the tuple) depend on the syntactic rules of the language and somewhat on the tradition of the lexicographer. Note that because the inflection table is a tuple, not a set, it is possible for the same word form to appear in different slots. For instance, a possible inflection table for the English lexeme GO[20] is ⟨*go, go, go, goes, went, gone, going*⟩ (respectively infinitive, imperfect, present, third person singular present, past, past participle, present participle).

Intensional and extensional lexicon. There are two ways to structure a morphological lexicon. An *intensional lexicon* lists the lexemes of the language and associates with each lexeme its inflection table (and potentially other information related to the lexicon's purpose). Example 1 shows simplified intensional lexicon entries for the English verbs EAT and PLAY and the noun PLAY. Note that syncretism between the third person singular simple present of the verb and plural of the noun *plays* or between the simple past and past participle form *played* is handled naturally. A more common way to represent an intensional lexicon is to replace the inflection table by a lemma and an identifier for the inflection class. Where the identifier references rules to construct the inflectional forms from the lemma. See Section 2.2.

On the other hand, an *extensional lexicon* lists all inflected forms individually. As Example 2 shows each form can be associated with a lexeme identifier and a morpho-

[20]**Typographic conventions** In this paper, we use the following conventions to distinguish lexemes from inflectional forms: we use small caps to write lexemes (represented by their lemma) and italics to write inflectional forms. E.g. *man* is one inflectional form of the lexeme MAN.

	Singular	Plural
nominative	*lūdus*	*lūdī*
genitive	*lūdī*	*lūdōrum*
dative	*lūdō*	*lūdīs*
accusative	*lūdum*	*lūdōs*
ablative	*lūdō*	*lūdīs*
vocative	*lūde*	*lūdī*

Table 8: Example of how the word forms of a lexeme may be arranged into a table, here the Latin forms for LŪDUS 'game'. The word forms are classified according to shared inflectional categories: all the forms in a column share the same number whereas forms in a row share the same grammatical case.

EAT	V	⟨*eat, eat, eat, eats, ate, eaten, eating*⟩
PLAY	V	⟨*play, play, play, plays, played, played, playing*⟩
PLAY	N	⟨*play, plays, play's, plays'*⟩

Example 1: This example of an intensional lexicon shows entries for the English verbs EAT and PLAY and the noun PLAY. Note that the second and third entries share the same lemma, although they have different inflection tables. We can also see that syncretism between the third person singular simple present of the verb and plural of the noun *plays* or between the simple past and past participle form *played* is naturally represented.

syntactic analysis. The amount and nature of associated information depends on the task for which the lexicon is built. For machine translation, for instance, the form might be associated with a lemma, morpho-syntactic analysis and semantic information to help choose the appropriate translation. A simple spell checker, on the other hand, needs no extra information, and it suffices to list the accepted forms. Example 2 also shows that alternative forms (such as the alternative forms *learnt* and *learned* for the simple past and past participle of LEARN) are easy to add to an extensional lexicon by creating a new entry (although not represented in this example, if a form is missing there is no entry). In cases of syncretism though, entries share the same form, thus creating ambiguity in analysis.

In practice, intensional lexicons are easier to create and maintain thanks to their more hierarchical structure, whereas extensional lexicons lend themselves better to analysis tasks and it is common for the lexicographer to work on the former, using paradigm functions that we define in the next section to generate the inflection tables and use automated tools to convert the intensional lexicon to an extensional form.

2.2 Paradigms

In this paper, a paradigm is a function $P : String \rightarrow String^n$ producing an inflection table from a *word string* (either a lemma or a stem)[21].

[21]Here we choose to use the term *paradigm* as a formal description of an inflection pattern, and we use the term *inflection table* for the list of inflected forms of a lexeme, which is also sometimes called its

learn	V	LEARN	imperative
learn	V	LEARN	infinitive
learn	V	LEARN	simple present
learned	V	LEARN	simple past
learned	V	LEARN	past participle
learning	V	LEARN	present participle
learns	V	LEARN	third-person singular simple present
learnt	V	LEARN	simple past
learnt	V	LEARN	past participle
play	N	PLAY	singular
play	V	PLAY	simple present
play	V	PLAY	infinitive
play	V	PLAY	imperative
played	V	PLAY	simple past
played	V	PLAY	past participle
playing	V	PLAY	present participle
plays	N	PLAY	plural
plays	V	PLAY	third-person singular simple present

Example 2: This example shows an extensional lexicon in which each form is associated with a part-of-speech, lemma and a morpho-syntactic analysis. Alternative forms (*learnt*/*learned* for the simple past and past participle of LEARN) are represented by creating a new entry. In case of syncretism, two or more entries entries share the same form.

The same paradigm function can be used for different lexemes. Each paradigm function defines an *inflectional class*, which is the set of all lexemes for which the paradigm produces the expected inflection table. For instance LŪDUS above belongs to the same inflectional class as DOMINUS and FILIUS, the second Latin declension. In the rest of the paper we use the terms *paradigm* and *inflectional class* interchangeably: saying that a lexeme belongs to a paradigm implies that it belongs to its inflectional class.

Inflection classes are a way to make lexicons more compact. For instance, the French Bescherelle, a reference book for French conjugation (Arrivé 2012), allows one to check the conjugation of several thousand French verbs (about 9600 in the 2012 edition). Given that the complete inflection table of a single verb takes one full page, it would take 9600 pages to print the complete inflection tables of all verbs. Using inflectional classes allows the authors to compress this to 184 pages (103 full inflection tables for the model verbs plus eighty-one pages of index giving a reference to the corresponding model for each of the 9600 verbs). In this particular case, the space needed to describe the lexicon was reduced by a factor 50.

This compression effect is also useful when learning a new language: in Latin, for instance, it is common to learn first a full inflection table of an example lexeme for each inflection class and then, when learning a new word, the pupil needs only remember to which inflectional class the word belongs to be able to reconstruct any form required.

2.3 Smart Paradigms

Different paradigm systems for the same language can have different numbers of paradigms. To continue our example above, the Bescherelle defines about eighty-two inflection classes for French verbs, whereas traditional French grammar has only three (*premier, deuxième et troisième groupes*).

One factor that determines the number of paradigms needed to cover the morphology of a lexicon is the set of string operations a paradigm is allowed to use. If the paradigms are limited to simple operations like concatenating a suffix to a stem, more paradigms are required. For example, if one were to define a paradigm to inflect English nouns, most could be inflected with the following function:

$$\texttt{noun}(l) = \langle l, l + \text{“s”} \rangle$$

e.g.:

(1) $\texttt{noun}(\textit{game}) = \langle \textit{game, games} \rangle$

But this would fail to produce the expected result for a noun like BABY:

(2) $\texttt{noun}(\textit{baby}) = \langle \textit{baby, babys} \rangle$

If we assume the only operation allowed in our paradigms is suffixing, we have no other choice but to define a new paradigm to handle nouns ending with *-y*:

$$\texttt{ynoun}(l) = \langle l + \text{“y”}, l + \text{“ies”} \rangle$$

This then produces the expected forms:

inflectional paradigm.

(3) $\texttt{ynouns}(bab) = \langle \textit{baby, babies} \rangle$

Now not only do we need to know that we should use a different paradigm to handle nouns ending with *-y*, but we also need to remember that the argument is the singular form without the *-y* suffix. By contrast, if we have access to a more powerful set of operations to define our paradigms, we can extend the original function:

$$\texttt{noun}(l) = \langle l, \text{if } l \text{ matches } x + \text{“y” then } x + \text{“ies” else } l + \text{“s”} \rangle$$

The number of arguments also affects the minimum number of paradigms required. For example, verbs ending in *-ir* in French can belong either to the second group (FINIR, ‘finish’) or third group (VENIR, ‘come’), and it is impossible to guess to which group it belongs based on the lemma alone. As a result, no matter how powerful our paradigms, we need at least two different ones to handle those verbs. If, on the other hand, we have access to other forms of the verbs, we can use this additional information. In our example, the third person singular indicative present is enough to differentiate between second and third groups (*nous finissons* versus *nous venons*).

If we allow paradigms with with arbitrarily complex string operation and multiple arguments, we need a smaller number of them.In GF, for some languages there is only one paradigm for each part of speech, but that paradigm has variable number of arguments. This unique paradigm is called a *smart paradigm*.

A common way to implement smart paradigms is to first implement a set of simpler paradigm functions, as in traditional paradigm systems, and then to create an “intelligent” function implementing a heuristic to pick one of the “dumb” paradigms. For instance, the function in Example 3 can correctly inflect a majority of French verbs.

```
mkV : String → String[51]
mkV(s) =
  conj19finir(s), if s ends with "ir"
  conj53rendre(s), if s ends with "re"
  conj14assiéger(s), if s ends with "éger"
  conj11jeter(s), if s ends with "eler" or "eter"
  conj10céder(s), if s ends with "éder"
  conj07placer(s), if s ends with "cer"
  conj08manger(s), if s ends with "ger"
  conj16payer(s), if s ends with "yer"
  conj06parler(s), if s ends "er"
```

Example 3: This example shows a smart paradigm for French verbs. Here simpler paradigms are available as the `conj*` functions modeled after the conjugations in the Bescherelle (like `conj19finir`). The smart paradigm chooses which one to use based on the ending of the lemma passed as argument (the infinitive form of the verb in this case).

In Table 9, we reproduce the evaluation of some of the manually written GF smart paradigms from Détrez and Ranta 2012. Most of the smart paradigms frequently generate

Lexicon	**Accuracy**	
	One argument	Two arguments
Eng N	95%	100%
Eng V	84%	95%
Swe N	46%	92%
Swe V	97%	97%
Fre N	76%	99%
Fre V	92%	94%
Fin N	87%	97%
Fin V	96%	99%

Table 9: GF smart-paradigm accuracy, as measured by Détrez and Ranta 2012. It reports the accuracy of the one-argument smart paradigms (e.g. Fre. `mkV "manger"`) and two-arguments smart paradigms (e.g. Fre. `mkV "jeter" "jette"`). While most smart paradigms are already good at generating the expected inflection table based on the lemma only (one argument), adding a second argument improves the results to more than 90% for all lexicons in this evaluation.

the expected inflection table based on the lemma only (one argument), but adding a second argument improves the results to more than 90% for all tested smart paradigms.[22]

Creating those heuristics manually is difficult and requires an intimate knowledge not only of the underlying paradigms but also of their distribution in the language (so that in case of ambiguities, the most frequent paradigm is picked). As a result, many languages in the GF resource grammar do not have smart paradigms.

Assuming the underlying "dumb" paradigms exist, the function of a smart paradigm can be seen as a classification task: given a new lemma, return its most probable inflection class considering only the string of characters: that is, the smart paradigms do not make use of external knowledge—such as a corpus—or feedback loops—like generating the forms and checking that the lemma is correctly re-generated. We thus hypothesize that smart paradigms can be created by training classifiers on existing lemma-paradigm pairs.

3 Experiments

First we describe the data we use in our experiments; these data come from a variety of lexicon projects (§ 3.1). Then we describe the machine learning techniques we use (§ 3.2). The purpose of our first experiment is to refine our approach to building smart paradigms (§ 3.3) and the purpose of the next is to evaluate its usefulness in the context of lexicon building (§ 3.4).

[22]If you are familiar with French, it might seem strange that the smart paradigm for nouns with two arguments does not reach 100% as French nouns have two forms (singular and plural). This is due to the fact that the creator of the grammar choose to use the noun's *gender* as the second argument to the smart paradigm as it was thought to be more informative because less predictable (so, one would write `mkN "maison" feminine`).

3.1 Lexicons

We describe our chosen lexicons below. We use a variety of lexicons from different sources during our experiments. First we want to apply our method on different languages to show the linguistic applicability of the method. We also want to show that the technique is not bound to a particular project (e.g., GF).

Note that we require data in a specific format: we use an intensional lexicon that gives for each lexeme the lemma (base form) and an identifier to the inflection class this lexeme belongs to. This is a standard way of representing lexicons but not necessarily the way they are distributed as it might be more practical for other usages to have the full form lexicon (where inflection tables are fully expanded and the paradigm identifiers are not necessarily retained)[23].

Finnish. For Finnish we use the KOTUS wordlist, published by the Institute for the Languages of Finland (Kotimaisten Kielten Tutkimuskeskus 2006). In this lexicon, each inflection class is represented by an integer (1–50 are nouns paradigms and 52–79 are verbs paradigms) sometimes accompanied by a letter indicating a consonant gradation (see Example 4). We combined those labels to create a new set of inflection classes where for instance, e.g. 1 and 1B are considered separate inflection classes.

```
<st><s>leikki</s><t><tn>5</tn><av>A</av></t></st>
<st><s>leikkiä</s><t><tn>61</tn><av>A</av></t></st>
```

Example 4: This example shows two entries in the Finnish lexicon: the noun LEIKKI 'game' and the verb LEIKKIÄ 'play'. Each element `st` is one entry where `st/s` is the lemma, `st/t/tn` the inflection class and `st/t/av` the optional consonant gradation.

French. For French, we use nouns and verbs from the Lefff (*Lexique des Formes Fléchies du Français*—Lexicon of French inflected forms, Sagot 2010). The Lefff is distributed in two different forms. The "source" format is the form that we are most interested in in this paper, where each entry corresponds to a different lexeme with a lemma, paradigm identifier and morpho-syntactic information (intentional lexicon). This is also the form used by the lexicographer to make changes to the lexicon (insertion, deletion or editing of entries) so it is where smart paradigms can be most useful. The second format lists each inflected form as a separate entry associated with all its possible morpho-syntactic analyses and syntactic information (extensional lexicon). This second format is automatically generated from the first and it is the format which most natural language processing tools use for text analysis. Example of entries from each format are given in Example 5.

Although, as described above, the Lefff contains morpho-syntactic information about the lexemes, we have limited ourself to predicting the inflection classes and ignored the syntactic information in our experiments. The main reason is that we wanted to have

[23]A full-form lexicon is a lexicon that lists all possible word forms of its lexeme. Extensional lexicon are full-form lexicon (e.g. Example 2) but intensional lexicons can also be full-form lexicons as it is the case in Example 1.

```
joueur  nc-eur \
  100;Lemma;nc;<Objde:(de-sinf|de-sn),Objà:(à-sinf)>;cat=nc;%default
```

```
joueur  100 nc \
  [pred="joueur_____1<Objde:(de-sinf|de-sn),Objà:(à-sinf)>",cat=nc,@ms] \
  joueur_____1  Default ms  %default  nc-eur
joueurs 100 nc \
  [pred="joueur_____1<Objde:(de-sinf|de-sn),Objà:(à-sinf)>",cat=nc,@mp] \
  joueur_____1  Default mp  %default  nc-eur
joueuse 100 nc \
  [pred="joueur_____1<Objde:(de-sinf|de-sn),Objà:(à-sinf)>",cat=nc,@fs] \
  joueur_____1  Default fs  %default  nc-eur
joueuses 100 nc \
  [pred="joueur_____1<Objde:(de-sinf|de-sn),Objà:(à-sinf)>",cat=nc,@fp] \
  joueur_____1  Default fp  %default  nc-eur
```

Example 5: Examples of entries for the lexeme JOUEUR (‘player/gamer’) from the Lefff. Top: intentional lexicon with one entry per lexeme. The first column is the lemma (JOUEUR), then comes the paradigm (`nc-eur`) and the last part are the syntactic information. Bottom: extensional version where each inflected form gets its own entry. From left to right are the word form, a weight (can be used to favor some entries, for instance if the weight of a multi-word expression is greater than the sum of the weight of its constituents), then the syntactic and morphological information. (A \ indicate that the line had to be broken to fit the paper but in the original data it continues on the same line.)

comparable experiments across languages and not all of the other lexicons have such detailed morpho-syntactic information. In addition, where the possibility of predicting the paradigms from the lemma (with a reasonable confidence) has been shown by the GF smart paradigm (Détrez and Ranta 2012), the relation between syntactic properties of a lexeme and the form of the lemma is not as clear and, to us, not obvious. It is though an interesting question that we would like to investigate in the future.

German. From the same author as the Lefff, we use the German lexicon DeLex (Sagot 2014). We also applied the method on both nouns and verbs. The comments made above about the Lefff apply here as well.

Maltese. Finally we used the list of Maltese verbs from Camilleri 2013. This lexicon was created as part of the Maltese resource grammar library for GF. Maltese is a Semitic language but thorough its history has been heavily influenced by Romance languages (mostly Sicilian and Italian) and, more recently, English. Thus, words in Maltese can exhibit either a concatenative or a root-and-pattern morphology (Spagnol 2011). This lexicon lists the verbs belonging to the second category, those exhibiting a root-and-pattern inflection.

The Maltese verb list is written in the GF programming language (Ranta 2011). As it is implemented, the lexicographer needs to manually specify the root of the lexeme to get the inflection table (in Example 6, the root would be *l-għ-b*). Although it means that we wouldn't be able to reconstruct the full inflection table from the lemma by simply guessing the paradigm without also having a way to automatically extract the root, we believe the predicted information might still be useful to the lexicographer. In addition we were curious to see how the method would perform on a non-concatenative morphology.

3.2 Sub-sequences and string kernels

Sub-sequences for a string work in the following way: given a sub-sequence length k and a decay factor λ, every, non-necessarily continuous, sub-sequence of k characters present in the input string is a feature with value λ^i where i is the span of the subsequence in the instance.

For instance, the strings *linguistics* and *immunology* both include the sub-sequence *i-n-g*. In the former, the sub-sequence spans 3 characters (l**ing**uistics) so the corresponding feature is attributed the value λ^3, whereas in the later it spans 9 characters (**i**mmu**n**olo**g**y) so the value is λ^9. As the decay factor λ is chosen between 0 and 1, the higher the exponent the lower the value.

Table 10 gives an example of feature values for the strings *dire* and *lire*. A method for efficiently computing the kernel matrix for sub-sequences is given in Lodhi et al. 2002.

We chose to use sub-sequences as features inspired by the way smart paradigms have been constructed in GF. As shown in example 3, the GF paradigm for French verbs works mostly by inspecting the lemma's suffix to select the best paradigm. The smart paradigm for Maltese verbs, on the other hand, uses the root of the verb to select the paradigm (in the example 6, the root was the disjoint sub-string *l-għ-b*). To make our method as general as possible, we decided to use all sub-sequences as input features.

```
{ s = table {
    VPerf (AgP1 Sg)     => lgħabt   ;
    VPerf (AgP1 Pl)     => lgħabna  ;
    VPerf (AgP2 Sg)     => lgħabt   ;
    VPerf (AgP2 Pl)     => lgħabtu  ;
    VPerf (AgP3Sg Masc) => lagħab   ;
    VPerf (AgP3Sg Fem)  => lagħbet  ;
    VPerf AgP3Pl        => lagħbu   ;
    VImpf (AgP1 Sg)     => nilgħob  ;
    VImpf (AgP1 Pl)     => nilogħbu ;
    VImpf (AgP2 Sg)     => tilgħob  ;
    VImpf (AgP2 Pl)     => tilogħbu ;
    VImpf (AgP3Sg Masc) => jilgħob  ;
    VImpf (AgP3Sg Fem)  => tilgħob  ;
    VImpf AgP3Pl        => jilogħbu ;
    VImp Sg             => ilgħob   ;
    VImp Pl             => ilogħbu  ;
  };
  i = {
    class = Strong LiquidMedial;
    form = FormI;
    imp = "ilgħob";
    root = {C1 = "l"; C2 = "għ"; C3 = "b"; C4 = []};
    vseq = {V1 = "a"; V2 = "a"}
  };
}
```

Example 6: Example of entry in the GF Maltese lexicon for the verb LAGĦAB '(he) plays' (simplified for readability). This is an example of intensional lexicon: here the inflection table is represented as an associative array where the keys are the morpho-syntactic analyses and the values are the forms. In the case of Maltese, the GF smart paradigm for verbs need to be given the consonantal root, e.g. `mkV "lagħab" (mkRoot "l-għ-b")`.

	d-i-e	d-i-r	d-r-e	i-r-e	l-i-e	l-i-r	l-r-e
dire	λ^4	λ^3	λ^4	λ^3	0	0	0
lire	0	0	0	λ^3	λ^4	λ^3	λ^4

Table 10: Example of feature space for the forms *dire* and *lire*. The features are all possible sequences of letters of length k (here $k = 3$) and their values is either 0 if the sequence does not appear in the word form (e.g., the sequence d-i-e is not a sub-sequence of the form *lire*) and λ^l otherwise, where l is of the span of the subsequence in the form (e.g., the sequence d-i-e is a subsequence of *dire* spanning four characters).

3.3 Experiment 1

In our first experiment, we evaluate our method on the different lexicons. As the machine learning technique we have chosen requires some parameters to be set, the first part of the experiment tries to experimentally select the best values for those parameters.

The first parameter, the soft margin parameter C controls the trade-off between correctly classifying more instances (in the training data) or maximizing the margin of the classifier. Lower values of C produce a classifier that fits the training data better, but may not generalize as well, whereas higher values of C produce classifiers that may not classify the training data as well but might generalize better.

The other two parameters are k and λ, related to the sub-sequence features: k represents the length of the extracted subsequences and λ, the decay factor (see Section 3.2).

There is, to our knowledge, no theory-based method to choose the parameter C for a given classification task. As for k and λ, as we want our method to work without linguistic knowledge from the user, we want to have them selected automatically as well.

We used a classic grid search method to select the parameters where we checked all combination of parameters with $C \in \{2^{-5}, 2^{-3}, \ldots, 2^{9}\}$, $k \in \{3, 4, 5\}$ and $\lambda \in \{0.1, 0.3, \ldots, 0.9\}$.

Train, tune and test data. To make the results more comparable between languages, we first randomly extracted 3000 verbs and 3000 nouns from each lexicon. We then divided each of those lists into three subsets. The biggest part, 1800 entries, is used to train the classifiers with different values of C, k and λ. Half of the remaining 1200 entries are used to evaluate those classifiers a during the grid-search algorithm. Finally we use the final part of our data, (the last 600 entries), for the evaluation of the final classifier using the best parameters from the grid search. It is important here that we do not use the same set we used during the grid search to do the final evaluation in order to avoid over-fitting.

We did this separately for each lexicon. The result of the grid search and accuracies of the resulting classifiers are presented Section 4.1.

3.4 Experiment 2

One natural question at this point is the following: We may be able to build a smart paradigm by training a classifier on an existing lexicon, but if one already has the lexicon, why would one be interested in building smart paradigms? We believe there is two parts to answering this question. First, training a smart paradigm on a large existing lexicon can be used to extend and otherwise maintain it. No lexicon is ever complete as languages evolve as new words are introduced. Furthermore, an existing lexicon might need to be adapted to a new domain and enriched with a specific terminology. Smart paradigms can make this work easier and cheaper, especially if they can be obtained automatically.

With this in mind, we can push this idea of extending the lexicon further and see lexicon development as an iterative process where lexemes are added one by one and consider each addition an extension of the existing lexicon. Then the question becomes at which point in the lexicon creation—from how many existing entries—does it become interesting to train a smart paradigm from the existing lexicon to help add the rest of the lexemes?

To get an estimation for this, we ran the following experiment: we divided the lexicon into slices $S_1, \ldots, S_N$ and, we train $N-1$ classifiers $C_1, \ldots, C_{N-1}$ where classifier C_i is trained on slices $S_1, \ldots, S_i$ and evaluated on slice S_{i+1}. This should simulate a process of iteratively training classifiers and using them to add more words. In this experiment we use the same lexicon subsets as in Experiment 1, divided into thirty slices of one hundred lemmas each.

The results of those experiments are presented in Section 4.2.

4 Results

This section presents the results of the experiments described in Section 3.

4.1 Experiment 1

To evaluate our smart paradigms, we use different metrics corresponding to slightly different use cases:

First-best accuracy. When applied to an unseen lemma, the learned smart paradigms can return an ordered list of possible paradigms where the first one is the most likely guess and so on. We define first-best accuracy as the ratio of lemmas for which the correct paradigm is the first one returned by the classifier. The intention here is to measure the ability of the method to work entirely automatically, without manual correction.

Recall. We define recall as the ratio of lemmas for which the correct paradigm is one of the returned candidates. (we followed Lindén 2009 in limiting the number of candidates to 6 for computing precision and recall). Together with the average precision, this metric aims at evaluating the usefulness of the method as an aid to lexicon creation (where the best candidates are given as suggestion to the lexicographer).

Average precision. We define average precision on a single lemma as $1/r$ where r is the rank of the correct paradigm in the candidate list or 0 if the correct paradigm is not in the list of predictions. We then average this number over the whole lexicon.

We compare the results to a simple baseline that returns the most frequent paradigms over the whole lexicon and, when available, to the accuracy of the manually written GF smart paradigms as measured in Détrez and Ranta 2012 (note however the results for French were computed on a different lexicon were the inflection classes might not be exactly the same).

Lexicon		Smart paradigm						Baseline			GF
		k	*λ*	*C*	*acc.*	*rec.*	*prec.*	*acc.*	*rec.*	*prec.*	*acc.*
Finnish	N	3	0.5	8	0.73	0.84	0.78	0.14	0.55	0.27	0.87
Finnish	V	3	0.5	2	0.90	0.94	0.92	0.21	0.55	0.31	0.96
French	N	4	0.5	2	0.75	0.96	0.84	0.44	0.94	0.66	0.76
French	V	4	0.5	32	0.95	0.96	0.95	0.85	0.93	0.88	0.92
German	N	4	0.7	2	0.26	0.47	0.33	0.30	0.65	0.42	-
German	V	3	0.7	0.5	0.49	0.89	0.65	0.15	0.62	0.32	-
Maltese	V	4	0.3	128	0.94	1.00	0.96	0.29	0.94	0.50	-

Table 11: Results of experiment 1, for each lexicon. k, λ and C are the parameters that were established during the grid search. Accuracy (*acc.*), recall (*rec.*) and precision (*prec.*) are calculated as defined in Section 4.1. Results for both the learned smart paradigms and the baseline are presented here. Where available, we also reproduced the accuracy of the GF smart paradigm for comparison.

As can be seen in Table 11, the learned smart paradigms beat the baseline for all but one lexicon for every metric, showing that the method might indeed be useful both as a way to automatically extend a lexicon and also as an aid to a human lexicographer.

Compared to the manually created smart paradigms from GF, for the languages that are common to this experiment and Détrez and Ranta 2012 we can observe that the learned smart paradigms obtain a comparable accuracy, although generally a bit lower.

4.2 Experiment 2

Results from experiment 2 are presented in Figure 10 where the x axes represents the size of the lexicon and the y axes the classifier accuracy when evaluated on the next 100 lemmas. This aims to be a simplified simulation of an hypothetical lexicon creation task where a lexicographer provides 100 entries with their paradigms, train a classifier on those and use it to classify the next 100 entries. Then they can fix the errors in those 100 entries and train a new classifier for the next batch, etc.

On the graph for each lexicon, we have also represented the accuracy of the smart paradigm and baseline from the first experiments as a dark (resp.: light) horizontal line.

As can be seen from the graph, in almost every case the smart paradigm, even when trained on a small amount of data, provide an improvement compared to the baseline. In most cases, it also performs well (as good as can be expected in light of the results of the previous experiment) with only about 1000 training examples. These results indicate that, in different languages, this technique could be of real help to a lexicon builder.

5 Related work

Nakov et al. 2003 introduce MorphoClass, a system designed to analyze unknown nominal word forms in German texts, group them into possible inflection tables and suggest a

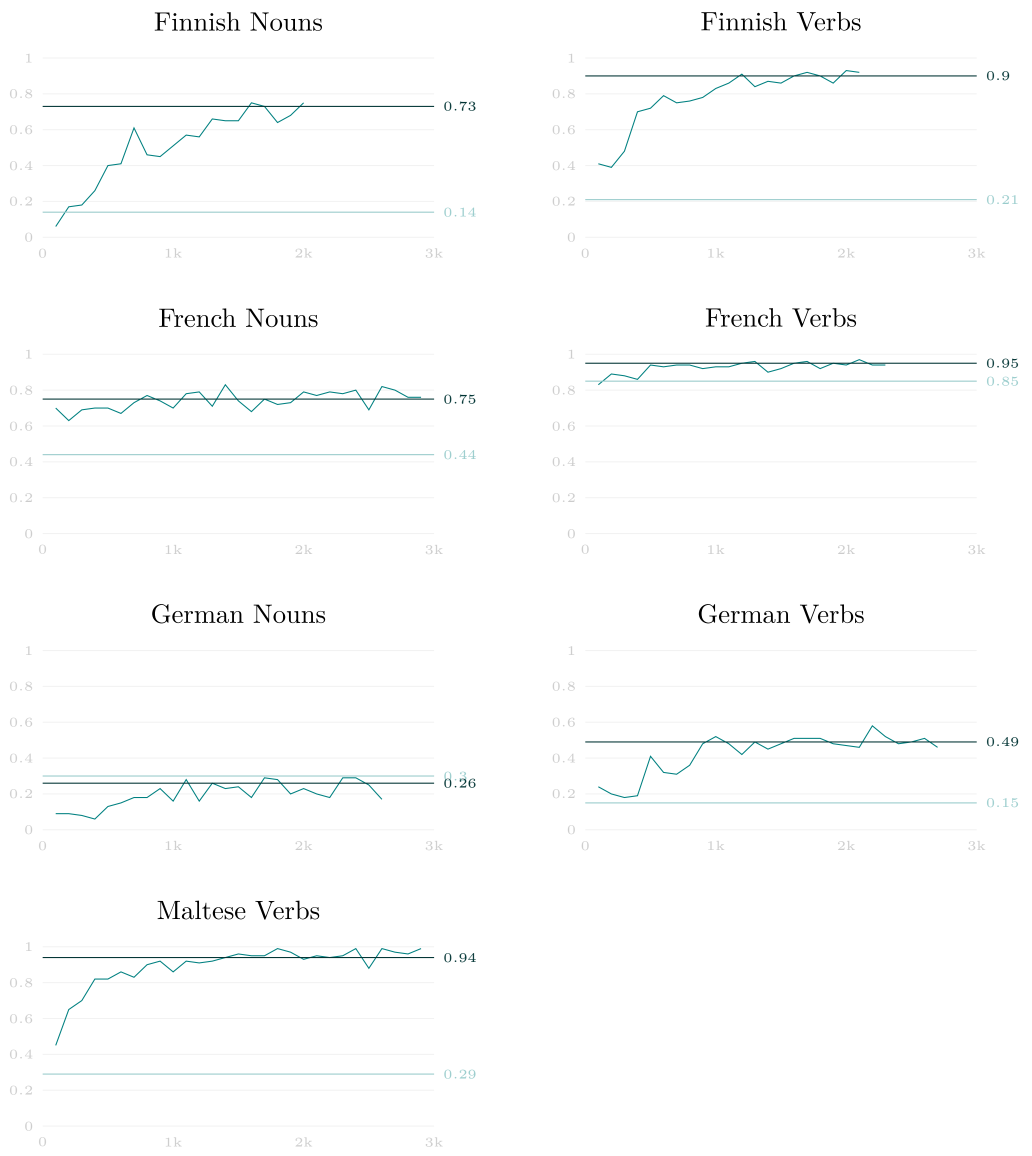


Figure 10: *Incremental accuracies of the learned smart paradigms. On the X axis is the size of the lexicon and on the Y axis, the accuracy of the smart paradigm when evaluated on the next 100 lemmas. The two horizontal lines represent the baseline* — *(bottom, except for German nouns) and the accuracy of the smart paradigm from experiment 1* — *(top, except for German nouns).*

stem and an inflection class. For the last part, the assignment of an inflection class, a short list of paradigms compatible with the grouped forms is ranked using an heuristic based on the stem ending. This ranking is then refined using articles, prepositions and pronouns preceding the considered word forms in the corpus (so for instance, as nouns in German have an intrinsic gender, if one of the form is preceded by a feminine article, then paradigms that apply to feminine nouns should be preferred).

Lindén 2009 proposes an way to generate lexicon entries from unknown word forms using analogies. Two methods are proposed, and eventually combined to solve the problem. One is corpus based, where two models are trained separately: the first guesses a base form from an inflected form and the second assigns a paradigm to the base form. The corpus material required are a list of inflected forms annotated with their corresponding base forms and a list of base forms annotated with their corresponding paradigms. The second method uses the implementation of a finite state transducer for the language morphology, that relates base forms to inflected forms, to create an entry generator that directly predict a lexicon entry (base form + paradigm identifier) from an inflected form. On Finnish, they obtain 0.89 recall, 0.81 precision and 0.85 f-score using the combined model. The first model which does not use the transducer achieves 0.80 recall, 0.69 precision and 0.74 f-score.

Šnajder 2013 proposes to use a binary classifier to predict whether a lemma-paradigm pair is correct. This differs from most other work in this domain (including the work described here) that consider the task as a multi-label classification problem (i.e. given a lemma, predict the paradigm label). They use a handcrafted set of linguistically motivated features as input to the machine learning algorithm, both string-based features (i.e. properties of the lemma or word form) and corpus based features (attested word-form count and distribution). They report results of 0.37 precision, 0.92 recall and 0.52 f-score for nouns and 0.33 precision, 0.83 recall and 0.47 f-score for verbs.

Ahlberg, Forsberg, and Hulden 2014; Ahlberg, Forsberg, and Hulden 2015 describe a method to extend a full-form lexicon by first automatically extracting paradigms from the inflection tables using the longest common subsequence (e.g. *swim, swam, swum* $\rightarrow$ *sw[iau]m* which is then generalized to $x_1[iau]x_2$). Then they try to automatically assign those paradigms to unseen word forms. They first exclude the paradigms that are incompatible with the word form (if a paradigm cannot possibly generate this form, it is excluded). Then, two different methods are proposed to rank the remaining paradigms. One uses a corpus to count how many of the generated forms are attested and paradigms generating more attested forms are ranked higher. The second method uses suffixes and prefixes as feature for a multi-label classification task using machine learning.

Novák 2015 proposes a supervised machine learning algorithm to predict the correct paradigm for a new lexicon entry using longest matching suffixes and some morpho-syntactic features such as part of speech, gender, etc. In addition, some paradigms have an associated suffix which limits their applicability, which means that it is known in advance that this paradigm can only be applied if the given suffix is present in the lemma.

We believe that the technique proposed in this paper differs from previous work in that it has fewer requirements and thus should be more generally applicable. In particular: there are no constraints on the way the paradigms are implemented (e.g. using finite-state transducers); it does not make use of linguistically motivated features or heuristics which are often language dependent and it does not require the availability of a corpus.

With regard to making the task of inserting new entries in a lexicon easier, Esplà-Gomis et al. 2014 is interesting. The objective of their method is to be usable by people with only speaker-level knowledge of the language, not trained lexicographers. To do this, they present the user with generated word forms and ask if those are valid forms of the word to be inserted. Machine learning is used to minimize the number of questions to ask the user but the possible paradigms need to be ranked first, for which they use a large corpus. Our technique could easily be used in place of the corpus based ranking if no large corpus is available for a given language.

A different but related stream of research is the unsupervised learning of morphology where the input is raw (unannotated) text and the goal is to extract some kind of morphological analysis in an unsupervised way. The level of analysis varies and can go from simple segmentation of the text to a complete morphological analysis of the language, See Hammarström and Borin 2011 for a recent review.

6 Future work

Our most immediate plan is to extend this research to more languages and more language families.

An other interesting direction is to evaluate the usefulness of the method in an actual lexicon creation task. This includes figuring out the best way to present the output of the smart paradigms to the user. One could imagine that different users may require different interfaces: for instance a linguist familiar with the set of traditional inflection classes could be presented with a list sorted according to the smart paradigm preferences, whereas a native speaker without specific linguistic training could be shown the generated inflection table and asked to decide if the forms are correct. A more sophisticated version of the later is presented by Esplà-Gomis et al. 2014 and could be adapted to work with our smart paradigms.

Finally, one major difference between the work presented here and the GF smart paradigms that inspired it is the fact that, in cases where the generated inflection table is incorrect, with the GF smart paradigm the grammar developer has the possibility to give more forms as arguments to guide the smart paradigm toward the correct inflection table. We plan to extend the classifiers presented here to include features from multiple forms to reproduce this behavior. Note that this present an extra challenge because, while the citation form used as the first argument of the smart paradigms is often dictated by the linguistics tradition of a language, the choice of what form to give for subsequent arguments is generally no as clear so as an extra step we need to automatically determine the most discriminative forms.

7 Conclusion

In this paper, we have seen that we can use machine learning together with substring features to learn the association between lemmas and paradigms to create what we refer to as a smart paradigm and evaluated this technique on different languages (Finnish, French, German and Maltese) in two different scenarios.

While providing comparable accuracy as heuristics manually crafted by experts, this technique has two main advantages: It does not require linguistic resources (such as a corpus) or advanced skills (e.g. the creation of linguistically motivated features). Additionally, it can be applied no matter which formalism has been chosen to implement the language's morphology. The technique seems to also be applicable to a variety of languages, and notably to languages that do not exhibit a concatenative morphology. Finally, we show that the technique gives good result with a modest amount of labeled data (around 1000 lexical items in our experiments) and that it can thus be useful during the creation of a new lexicon.

Some lexicons, as argued by Détrez and Ranta 2012, may have a lower intrinsic predictability, and our technique won't work in those cases, but neither do the manually created heuristics and it might be necessary to use external resources such as a corpus—another possibility is to change the form selected as lemmas, as different forms have different predicting power.

Finally, the accuracy of a smart paradigm can be used as a measure of the predictability of the lexicon (Détrez and Ranta 2012). We believe that this applies to the automatically created smart paradigms as well.

We believe that techniques like the one we presented, that are able to accelerate the creation of linguistic resources, are more and more crucial to develop since, in the words of Allwood 2006, p. 8: "the number of endangered languages is high, time is short, and economic resources are limited."

References

Ahlberg, M., M. Forsberg, and M. Hulden (2014). "Semi-supervised learning of morphological paradigms and lexicons". In: *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics.* Association for Computational Linguistics, pp. 569–578. DOI: `10.3115/v1/e14-1060`.

— (2015). "Paradigm classification in supervised learning of morphology". In: *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.* Association for Computational Linguistics. DOI: `10.3115/v1/n15-1107`.

Allwood, J. (2006). "Language survival kits". In: *Lesser-Known Languages of South Asia.* Trends in Linguistics. Studies and Monographs. Walter de Gruyter GmbH. DOI: `10.1515/9783110197785.3.279`.

Arrivé, M., ed. (2012). *Bescherelle – La conjugaison pour tous.* Hatier.

Camilleri, J. J. (2013). "A Computational Grammar and Lexicon for Maltese". Master thesis. Chalmers University of Technology.

Détrez, G. and A. Ranta (2012). "Smart Paradigms and the Predictability and Complexity of Inflectional Morphology". In: *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics.* Avignon, France: Association for Computational Linguistics, pp. 645–653.

Esplà-Gomis, M. et al. (2014). "An efficient method to assist non-expert users in extending dictionaries by assigning stems and inflectional paradigms to unknown words". In: *Proceedings of the 17th Annual Conference of the European Association for Machine Translation*. Dubrovnik, Croatia, pp. 19–26.

Hammarström, H. and L. Borin (2011). "Unsupervised learning of morphology". In: *Computational Linguistics* 37.2, pp. 309–350. DOI: `10.1162/COLI_a_00050`.

Kotimaisten Kielten Tutkimuskeskus (2006). *KOTUS Wordlist*. URL: `http://kaino.kotus.fi/sanat/nykysuomi`.

Lindén, K. (2009). "Entry Generation by Analogy – Encoding New Words for Morphological Lexicons". In: *Northern European Journal of Language Technology* 1, pp. 1–25. DOI: `10.3384/nejlt.2000-1533.09111`.

Lodhi, H. et al. (2002). "Text Classification Using String Kernels". In: *Journal of Machine Learning Research* 2, pp. 419–444.

Nakov, P. et al. (2003). "Guessing morphological classes of unknown German nouns". In: *Recent Advances in Natural Language Processing III, Selected Papers from RANLP 2003, Borovets, Bulgaria*, pp. 347–356. DOI: `10.1075/cilt.260.39nak`.

Novák, A. (2015). "Making Morphologies the "Easy" Way". In: *Computational Linguistics and Intelligent Text Processing*. Ed. by A. Gelbukh. Vol. 9041. Lecture Notes in Computer Science. Cairo, Egypt: Springer International Publishing, pp. 127–138. DOI: `10.1007/978-3-319-18111-0_10`.

Ranta, A. (2009). "The GF Resource Grammar Library". In: *Linguistic Issues in Language Technology* 2.2.

— (2011). *Grammatical Framework: Programming with Multilingual Grammars*. 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth). Stanford: CSLI Publications.

Sagot, B. (2010). "The Lefff, a freely available and large-coverage morphological and syntactic lexicon for French". In: *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10)*. Valletta, Malta: European Languages Resources Association.

— (2014). "DeLex, a freely-avaible, large-scale and linguistically grounded morphological lexicon for German". In: *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC-2014)*. Reykjavik, Iceland: European Languages Resources Association.

Spagnol, M. (2011). "A Tale of Two Morphologies: Verb structure and argument alternations in Maltese". PhD Thesis. University of Konstanz.

Šnajder, J. (2013). "Models for predicting the inflectional paradigm of croatian words". In: *Slovenšcina* 2, pp. 1–34.

Paper D

Tools for a grammar engineering community

Chapter 1

A GF tokenizer

1.1 Introduction

Tokenization is the operation that consists in taking a string of characters and slicing it into "chunks" called *tokens*. Natural language grammars are often written on the token level, which means that the described language is a sequence of tokens and not a sequence of characters. So tokenization is very often a pre-requisite for parsing.

Token or words? The concept of 'token' might be difficult to distinguish from the concept of 'word'. The notion of token does not totally overlap with our idea of a word, for instance punctuation marks are often considered separate tokens, or some words might be split in several tokens (e.g. Compound words in Swedish such as *sjöodjur* 'sea-monster'). In addition, the notion of token is more "technical" in the sense that the precise definition of what constitute a token might depend on the application of the tokenizer. For instance, there is a variant of the GF grammar for Finnish which, to reduce the number of different tokens the grammar has to consider, treat some morphological suffixes as separate tokens. We refer the reader to He and Kayaalp 2006 for a comparison of 13 off-the-shelve tokenizers for English and their differences.

1.2 Description of the algorithm

In this section we describe the algorithm used to tokenize a string based on the list of terminals recognized by the grammar.

We use as an example a very simple language with only three terminals: `foo`, `bar` and `barfoo`. Those tokens may be separated by a space character but not necessarily. For example, the following productions are considered valid in this language: `barfoo␣bar`, `foo␣barbar` but not `foo␣baz`.

The output of the tokenizer, which is used as an input for the parsing algorithm, should consist of a string of tokens separated by a single space character and including the special terminal symbol `&+` to indicate that the surrounding tokens where concatenated

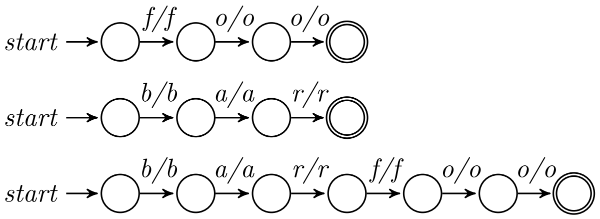


Figure 1.1: *This figure shows individual transducers for each of the valid tokens in our toy language that only accept the given token as input and print it back exactly the same way on the output.*

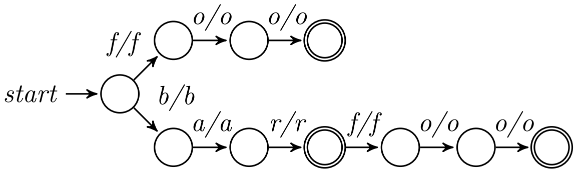


Figure 1.2: *This figure shows the union of the transducers from Figure 1.1. This unique transducer can accept any of token from the language but can still only accept a single token.*

in the original input string. For instance, tokenizing `foo␣bar` should return `foo␣bar` whereas `foobar` should return `foo␣&+␣bar`.

We first build a simple finite state transducer for each of the valid token. As shown in figure 1.1, those transducers only accept the given token and print it back on the output.

Then, we take the union of those transducers to get a single transducer that accepts (and print back) any of the valid tokens. See Figure 1.2.

Finally, for each final state in our union transducer, we create two extra transitions to the initial state, one accepting a space character and printing it back and one accepting an empty string (ϵ) and printing a binding terminal (`&+`). Figure 1.3 shows the complete transducer, with the extra transitions, for our running example. This means that two tokens separated by a space are printed as is but if two tokens are concatenated spaces are inserted together with the spacial terminal `&+`[1].

Note that the resulting transducer can be ambiguous. For instance in our toy example, the string `barfoo` has two possible tokenization: `bar␣foo` on `bar␣&+␣foo`.

1.3 Usage

Once an application grammar has been loaded, the tokenizer is available in the GF shell as the command `t` or `tokenize`. Example 7 shows an example of usage with a German

[1] In this example, we used a space as the separator because this is what GF does. This might not be suitable for every application though, as space might be a valid character in a token in other applications (for example if a multi-word expression is to be interpreted as a single token). In those cases, one may replace the output character in those extra transitions with the chosen separator.

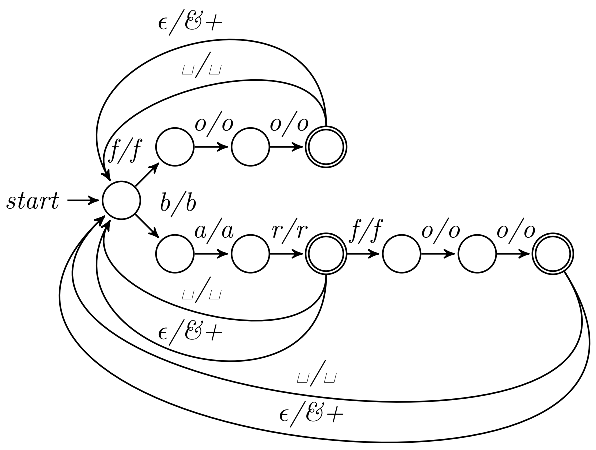


Figure 1.3: *This figure shows the final transducer used to do tokenization. By adding transitions from the final states back to the initial state, we can parse arbitrary long sequences of tokens. If the tokens are separated by a space, it is printed unchanged but if the tokens are glued the transducer inserts the special sequence **&+** to indicate to the parser that the two tokens are concatenated in the original string.*

example.

```
ExampleAbs> t "rhabarberbarbarabarbar"
rhabarber &+ barbara &+ bar &+ bar

3 msec
```

Example 7: Example of using the tokenizer in the GF shell.

Note that the transducer described in Section 1.2 is constructed and compiled when the command is first used, which means that the first invocation of the command takes longer to execute but subsequent invocations should be fairly quick.

Example 8 shows what happens when the input string can be tokenized in different ways: the tokenizer return every possible combination of tokens satisfying the input string.

```
ExampleAbs> t "rhabarberbarbarabarbarbarenbartbarbierbier"
rhabarber &+ barbara &+ bar &+ barbaren &+ bart &+ bar &+ bier &+ bier
rhabarber &+ barbara &+ bar &+ barbaren &+ bart &+ barbier &+ bier

3 msec
```

Example 8: Example of tokenizing an ambiguous string: all possible tokenizations are printed.

Finally, the output of the tokenizer command can be send to the parser using the standard piping mechanism in the GF shell.

The help page for this command is accessible from the shell with the command `help tokenize` (also given in Example 9). The code for the tokenizer is available in Appendix A.

```
ExampleAbs> help tokenize
t, tokenize
Tokenize string using the vocabulary

flags:
 -lang  The name of the concrete to use


0 msec
```

Example 9: The help page for the tokenizer command.

1.4 Current status

The work described here has now been superseded by a parser enhancement that is now capable of doing tokenization on the fly. For more information about this work see Angelov 2015.

Chapter 2

A Java Interpreter for PGF

2.1 Introduction

This chapter describes our work in implementing the basic GF runtime system in Java and using it for building applications on the Android platform.

We describe the implementation and usage of a PGF runtime library implemented for the JVM. PGF is a binary format used by GF to efficiently store a set of grammars sharing the same abstract syntax. A PGF runtime is what allow a program to do parsing and linearization (and hence translation) using those grammars.

There are many motivations to make linguistic applications for handled devices. One can think of automatic translation, tools for languages learner or for travelers and help for impaired people. Many existing services in those categories requires a live connection to the Internet, which is not always available, especially when one is traveling abroad.

One of the advantage of GF is its extensive and growing resource library, with formal grammar and basic vocabulary for over 37 languages (Ranta 2009). The library provides the linguistic background for developing domain-specific grammars and other language applications.

And finally, we choose the Android platform to experiment on because of its openness and growing adoption.

In this chapter, § 2.2 describes the JPGF library; § 2.3 reproduces the text version of a tutorial that was given at the GF summer school of 2013; § 2.4 presents PhraseDroid, an application written using JPGF for android devices; § 2.5 summarize related work; and § 2.6 concludes.

2.2 JPGF

2.2.1 Overview

JPGF is a PGF runtime implementation for the JVM[1]. It was written as part of an effort to make GF-based applications easier to write, as more developers are familiar with

Java than Haskell. In addition, compiling Haskell code to run on android devices and to communicate with the Android framework provides significant difficulties whereas code written for the JVM can easily be adapted to run in the android runtime environment[2].

The main operations that can be performed on a GF grammar are *parsing*—from natural language to the abstract syntax tree representing the underlying concept—and *linearization*—generating natural language constructions in a certain language from an abstract syntax tree. By combining this two operations one obtains a *translation* between any two concrete grammars. This approach has the advantage that the translation is always syntactically correct, due to the fact that the linearization in a certain grammar, uses the implementation of the concrete syntax module.

In addition to this, GF provides a portable runtime format, PGF (Angelov, Bringert, and Ranta 2010) which can be used to embed the libraries further on in applications written in programming languages that provide a suitable interpreter. This way, other projects can use GF modules, as normal software libraries for the development of other projects. When we started this work, up-to-date PGF interpreters existed for Haskell and JavaScript, and our work resulted in a Java version of the interpreter.

2.2.2 Implementation

To use the PGF grammars in android applications we needed to re-implement the runtime system from scratch for two reasons: first, the existing runtime system is written in Haskell and second, since the algorithms for parsing and linearization are specific to GF, we couldn't use other existing parsing libraries.

The library is composed of four somewhat independent modules: *A PGF reader* that reads the PGF binary into memory; *a parser* that analyzes strings into abstract-syntax trees (ASTs); *a linearizer* that turns ASTs into concrete strings (the opposite of the parser) and *a random generator* that randomly generates valid ASTs according to the current grammar.

The parsing algorithm is described by Angelov 2009, and the linearization algorithm by Angelov and Ranta 2010.

2.2.3 Source code

The source code for the library and the demo application is available under the terms of the GNU lesser general public license at `https://github.com/GrammaticalFramework`.

In Table 2.1 we give an overview of the size of both the JPGF and the example application PhraseDroid (presented in § 2.4).

[2] While the JVM is mostly known to run Java programs, many new languages (and compiler for old languages) have been created that target the same bytecode. One of those newer languages, which is becoming increasingly popular, is Scala. Part of the JPGF library (the parsing algorithm) have been written in Scala, although form a developer point of view using the library, it does not make a difference.

[2] Until 2013, Davlik was used as the principal runtime environment for android apps. It has then been progressively replaced by ART (Android Runtime) to provide better performances. Both use `.dex` files as input that can be easily obtain from Java `.class` files.

Language	Files	Code	Comment	Comment %	Blank	Total
JPGF						
java	91	4279	1844	30.1%	777	6900
scala	5	275	210	43.3%	71	556
Phrasedroid						
java	12	1123	372	24.9%	232	1727

Table 2.1: Table showing the size of the JPGF library and the example application PhraseDroid, measured in lines of code by ohcount.

	Parsing	Lineariza-tion	Random generation	Dependent types	Logical framework
Haskell	🟩	🟩	🟩	🟩	🟩
Java	🟩	🟩	🟩		

Figure 2.1: *This figure shows a comparison of the implemented runtime features in the reference Haskell implementation and the JPGF library. Missing features in JPGF are due to the time constraints for the implementation and also to the limited computing power of the targeted hardware (mobile phones). Although we do not expect this situation to last given the current rate at which embedded hardware is improving. We hope to be able to address those missing feature in future work.*

2.2.4 Evaluation

This section compares the new JVM library with the preexisting Haskell implementation. First, we compare the set of implemented feature and then we present a small benchmark of the parsing algorithm.

Implemented features

In this project, the focus was on implementing and optimizing parsing and linearization. The main reason is that the limited computing power of the targeted devices would make difficult to implement the full GF runtime system. Figure 2.2 gives an overview of the implemented features.

Those components are already enough to build interesting applications using natural language (see § 2.4). Moreover, for complex grammars, we quickly reach the limits of the devices computing power.

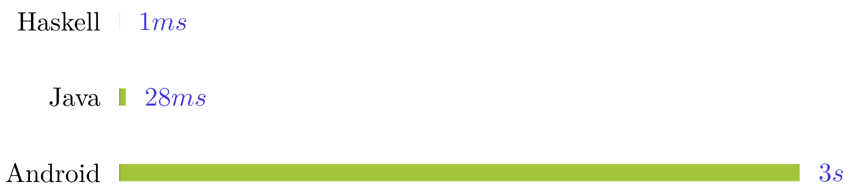


Figure 2.2: *This figure shows the results of a benchmark of the JPGF library: average translation time per sentence. The first line is the performance of the reference Haskell implementation, the second the new library evaluated on the same hardware (desktop PC) and the last one is the result of the same benchmark on Android hardware (Nexus One).*

Parsing benchmark

A small benchmarking experiment was conducted where the sames sentence was parsed multiple times (to average on multiple runs) in different settings: 1. using the Haskell runtime on a desktop computer; 2. using the JPGF runtime on the same desktop using the JVM computer and 3. using the JPGF runtime on an android device using Davlik (Nexus One). Figure 2.2 shows the results.

The poor parsing performances on the android device are expected given the lower processing power available. The gap between the two runtime on the desktop computer requires more investigation. One possible explanation is that comparing benchmarks across programming language is not trivial, even if the task is the same. But the main reason explaining the difference is that JPGF is at its first version and the focus of the developers was mainly on the correctness of the algorithm, leaving optimization for a later phase whereas the Haskell implementation has already been heavily optimized.

2.3 Tutorial

This section reproduces the text of a tutorial that was given at the GF summer school of 2013[3].

2.3.1 Introduction

In this tutorial we go through the necessary steps to build a small translation application using a PGF grammar on android. The specification of the application are quite simple:

- The interface should display a text box, a “Translate!” button and a space to show translations.
- The grammar is loaded when the application is started.

[3] The GF summer school is a bi-annual event where users, developers and researchers meet to present their work, discuss ideas and learn from each other, focusing on GF and the Resource Grammar Library. More information about the 2013 summer school can be found at `http://school.grammaticalframework.org/2013/`

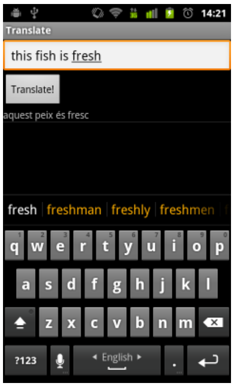


Figure 2.3: *Screen capture of the interface of the complete application.*

- When the user enters a sentence in the text box and clicks the button, the sentence is split into tokens and the grammar is used to retrieve translations
- The translations are then displayed in a list on the screen.

The final application should look like Figure 2.3.

2.3.2 Start the android application

This section is similar to the android "Hello world" tutorial[4]. Note that the exact syntax of the command may change in the future version of the SDK. Please refer to the official android developer website for the latest instructions.

Let's now create our android project. For that we the command line tools bundled with the android SDK. You can of course use the eclipse plug-in to create the android project. Please refer to the page linked above for instructions on how to do that.

```
$ android create project \
>   --package com.example.translateapp \
>   --activity Translate \
>   --target 2 \
>   --path TranslateApp
```

This should create a new directory called `TranslateApp` containing the basic structure of an android application. Right now the application does not do much, but it is still possible to test your application: enter the newly created directory then build and install the application (for this to work you need to have either a running emulator or a connected android device):

```
$ cd TranslateApp
$ ant install
```

[4]`http://developer.android.com/resources/tutorials/hello-world.html`

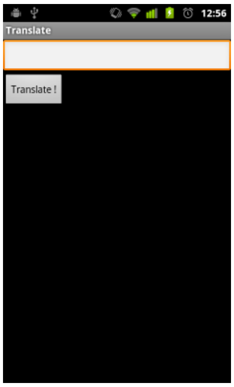


Figure 2.4: *Main interface of the application*

2.3.3 Application interface

We can now create the application's interface. Figure 2.4 shows what we are aiming for. To do that we modify the main layout file under `res/layout/main.xml`. We include a text field with a button and then a `ListView` to display the list of translation. The translation into the android user-interface language is given next:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <!-- The textbox where the user will enter a sentence -->
    <EditText
       android:id="@+id/edittext"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"/>
    <!-- The "Translate!" button -->
    <Button
       android:id="@+id/button"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:onClick="translate"
       android:text="Translate!" />
    <!-- the list to display the translations -->
    <ListView
       android:id="@+id/list"
       android:layout_width="fill_parent"
       android:layout_height="fill_parent"
       android:layout_weight="1"/>
```

```
</LinearLayout>
```

Test your application again. The interface should now look like the screen-shot in Figure 2.4.

2.3.4 Application code

Skeleton Let's now take a look at the Java code for the application. In the current state, the interface should display properly and let the user enter text but it does not do anything else. It might even crash if you press the button.

This is because we didn't implement the `translation()` function that is specified as `onClick` parameter for the button. Let's add a dummy function for now, to understand how it works.

Open the file `src/com/example/translateapp/Translate.java`, it should look like this:

```
package com.example.translateapp;

import android.app.Activity;
import android.os.Bundle;

public class Translate extends Activity
{
  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
}
```

Let's add our `translate` function. According to the android developer documentation, it should have the signature `public void translate(View v)`. Since we do not have to return anything, we can just add an empty function:

```
public void translate(View v) {

}
```

For this to work, we need to include the `View` class from the android library:

```
import android.view.View;
```

Your code should now look like this:

```
package com.example.translateapp;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;

public class Translate extends Activity
{
  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void translate(View v) {

  }
}
```

Reading the input text and populating the list The application does not crash anymore but it still does not do anything. Let's fix this.

We update our `translate` function so that it grabs the input text and copies it 10 times in the list. Reading the input text is done by first getting a handle to the corresponding view and then getting the text:

```
TextView tv = (TextView)findViewById(R.id.edittext);
String input = tv.getText().toString();
```

And import the necessary class:

```
import android.widget.TextView;
```

Now, we can copy it in the list. First, we should setup a data structure for the list, we use a `ArrayAdapter` in this example. In the `onCreate` function, just add:

```
mArrayAdapter = new ArrayAdapter(this, R.layout.listitem);
ListView list = (ListView)findViewById(R.id.list);
list.setAdapter(mArrayAdapter);
```

Again we need to import some classes from the code to work:

```
import android.widget.ListView;
import android.widget.ArrayAdapter;
```

Then we need add a new class member:

```java
private ArrayAdapter mArrayAdapter;
```

The resource id `R.layout.listitem` references a new layout that controls how each item is displayed in the list. Let's use a simple `TextView`. Create the file `res/listitem.xml` with this content:

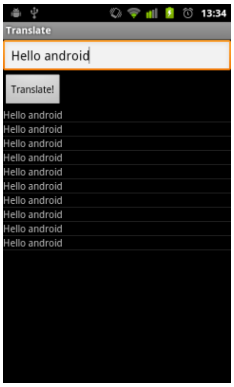
```xml
<?xml version="1.0" encoding="utf-8"?>
<TextView
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"/>
```

Finally, we can populate the list in the translate function:

```java
//... Getting the input string
mArrayAdapter.clear();
for (int i = 0; i < 10 ; i++)
  mArrayAdapter.add(input);
```

Now your `Translate.java` file should look like this:

```java
package com.example.translateapp;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
import android.widget.ListView;
import android.widget.ArrayAdapter;

public class Translate extends Activity
{

  private ArrayAdapter mArrayAdapter;

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mArrayAdapter = new ArrayAdapter(this, R.layout.listitem);
    ListView list = (ListView)findViewById(R.id.list);
    list.setAdapter(mArrayAdapter);
  }
```



```
Hello android
Hello android
Hello android
Hello android
Hello android
Hello android
Hello android
Hello android
Hello android
Hello android
Hello android
```

Figure 2.5: *Screen capture of the application where the translation functionality is not yet implemented and replaced by a test function that populates the list with the "Hello World" message ten times.*

```
  public void translate(View v) {
    TextView tv = (TextView)findViewById(R.id.edittext);
    String input = tv.getText().toString();
    mArrayAdapter.clear();
    for (int i = 0; i < 10 ; i++)
      mArrayAdapter.add(input);
  }
}
```

Run and test your application, you should be able to get something like the screen capture in Figure 2.5.

2.3.5 Add he JPGF library and the PGF file

Now that we have a working application, let's do something useful. First, we have to add the JPGF library to our project. Download the latest version of the library from GitHub[5]; and add the jar file to the `libs` folder in your project.

Next, we add the PGF file itself. In this example, we use the food grammar, but feel free to use your own file if you want.

We need to use extra option during the PGF compilation in order to make a PGF that is optimized and indexed. This allows android to cope more easily with big PGF files. Here is the command for the food grammar:

```
$ gf -make -s -optimize-pgf -mk-index Foods???.gf
```

Now copy the file `Foods.pgf` into res/raw and rename it to `foods.pgf` (resource file names should contain only lower case letters). Your project directory should now look like this:

[5]`https://github.com/GrammaticalFramework/JPGF`

```
TranslateApp
+ libs:
  + JPGF-1.0rc1.jar
+ res:
  + layout/
    + listitem.xml
    + main.xml
  + raw/
    + foods.pgf
  + values/
    + strings.xml
+ src:
  + com/
    + example/
      + translateapp/
        + Translate.java
+ AndroidManifest.xml
+ build.xml
+ local.properties
+ build.properties
+ default.properties
+ proguard.cfg
+ bin/
  ...
+ gen/
  ...
```

Compile and test the project again to make sure everything is in order.

2.3.6 Implement the PGF functions

Last but not least, we can now implement the translating functionality. We need to do two things:

- loading the PGF in memory in `onCreate`
- translate the input when `translate()` is called.

A word on performances. PGF operations are costly. On a cell phone, reading a PGF can take several seconds, depending on the size of the grammar and it is not unusual for parsing to take 1 or two seconds as well. If we do that in the main thread for our application, called the "UI thread", we block the user interface. This is not very good practice and it could even lead the OS to believe that our application is stalled and to display an error message.

To avoid this problem we need to put the PGF computations in other threads. The android framework offers different ways to do that. In this tutorial, we use the `AsynTask`[6] class.

Explaining how to use this class is not in the scope of this tutorial so I invite interested readers to look at the class documentation. There is only three important methods for our example in this class:

`onPreExecute` used to setup the interface when the task start (e.g. displaying a progress window.)

`doInBackground` used to do the expensive computation. This cannot modify the UI.

`onPostExecute` used to update the UI when the task is completed (e.g. removing the progress window.)

Loading the PGF. Loading a PGF file is done with the class `PGFBuilder`. It offers two static methods that both return a PGF object: `fromFile` and `fromInputStream`. The first expects a file name. Since, in an android project, static resource files are better accessed by resource id, we use the second one and open an `InputStream`[7]. This is done like this:

```
InputStream is = getResources().openRawResource(R.raw.foods);
```

Then we give this stream to `PGFBuilder`. In addition we give the list of desired concrete grammar so only those be kept in memory, this allow us to be more efficient in memory usage.

```
PGF pgf = PGFBuilder.fromInputStream(
    is, new String[] {"FoodsEng", "FoodsCat"});
```

Now, as explained above, we do not do this directly in `onCreate` to avoid blocking the UI. Instead we need to subclass `AsyncTask` and read the PGF in the `doInBackground` method. In addition, we add the code for the progress window:

```
/**
 * This class is used to load the PGF file asynchronously.
 * It display a blocking progress dialog while doing so.
 */
private class LoadPGFTask extends AsyncTask<Void, Void, PGF> {

  private ProgressDialog progress;

  protected void onPreExecute() {
    // Display loading pop-up
    this.progress =
```

[6] http://developer.android.com/refeorence/android/os/AsyncTask.html
[7] http://developer.android.com/reference/java/io/InputStream.html

```
        ProgressDialog.show(
            Translate.this, "Translate",
            "Loading grammar, please wait", true);
  }

  protected PGF doInBackground(Void... a) {
    int pgf_res = R.raw.foods;
    InputStream is = getResources().openRawResource(pgf_res);
    try {
      PGF pgf = PGFBuilder.fromInputStream(
          is, new String[] {"FoodsEng", "FoodsCat"});
      return pgf;
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }

  protected void onPostExecute(PGF result) {
    mPGF = result;
    if (this.progress != null)
      this.progress.dismiss(); // Remove loading pop-up
  }
}
```

Finally, we need a new class member for the PGF

```
private PGF mPGF;
```

And we need to launch the task from `onCreate`:

```
new LoadPGFTask().execute();
```

The translation task. Translation is just the combination of parsing and linearization. In JPGF, those tasks are respectively done with a `Parser` and a `Lnearizer` object. Those are easily created given the PGF and the concrete grammar:

```
Parser mParser = new Parser(mPGF, "FoodsEng");
Linearizer mLinearizer = new Linearizer(mPGF, "FoodsCat");
```

The parser object expect an array of tokens, which means that we need to tokenize the sentence first:

```
String[] tokens = sentence.split(" ");
```

And returns a `ParseState` object from which we can retrieve parse trees:

```
ParseState mParseState = parser.parse(token);
Tree[] trees = (Tree[])mParseState.getTrees();
```

The Linearizer takes a tree and return a string:

```
String s = mLinearizer.linearizeString(trees[0]);
```

Finally, once enclosed in an AsyncTask sub-class with the code for the progress window and some boilerplate code we get:

```
/**
 * This class is used to parse a sentence asynchronously.
 * It display a blocking progress dialog while doing so.
 */
private class TranslateTask
    extends AsyncTask<String, Void, String[]> {

  private ProgressDialog progress;

  protected void onPreExecute() {
    // Display loading pop-up
    this.progress =
        ProgressDialog.show(
            Translate.this, "Translate",
            "Parsing, please wait", true);
  }

  protected String[] doInBackground(String... s) {
    try {
      // Creating a Parser object for the FoodEng
      // concrete grammar
      Parser mParser = new Parser(mPGF, "FoodsEng");
      // Splitting the input (basic tokenization)
      String[] tokens = s[0].split(" ");
      // parsing the tokens
      ParseState mParseState = mParser.parse(tokens);
      Tree[] trees = (Tree[])mParseState.getTrees();

      String[] translations = new String[trees.length];
      /* Creating a Linearizer object for the FoodCat
       * concrete grammar */
      Linearizer mLinearizer = new Linearizer(
          mPGF, "FoodsCat");
      // Linearizing all the trees
      for (int i = 0 ; i < trees.length ; i++) {
        try {
          String t = mLinearizer.linearizeString(trees[i]);
          translations[i] = t;
        } catch (java.lang.Exception e) {
          translations[i] = "/!\\ Linearization error";
        }
      }
      return translations;
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }

  protected void onPostExecute(String[] result) {
    mArrayAdapter.clear();
    for (String sentence : result)
      mArrayAdapter.add(sentence);
    if (this.progress != null)
      this.progress.dismiss(); // Remove loading pop-up
  }
}
```

And again, we launch the task when appropriate: in the `translate` method

```
public void translate(View v) {
  TextView tv = (TextView)findViewById(R.id.edittext);
  String input = tv.getText().toString();
  new TranslateTask().execute(input);
}
```

Add the right classes to the imports:

```
import android.app.ProgressDialog;
import android.os.AsyncTask;
import java.io.InputStream;
import org.grammaticalframework.Linearizer;
import org.grammaticalframework.PGF;
import org.grammaticalframework.PGFBuilder;
import org.grammaticalframework.Parser;
import org.grammaticalframework.parser.ParseState;
import org.grammaticalframework.Trees.Absyn.Tree;
```

Full Translate.java. Here is the final `Translate.java` for this tutorial.

```
package com.example.translateapp;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
```

```java
import android.widget.ListView;
import android.widget.ArrayAdapter;

import android.app.ProgressDialog;
import android.os.AsyncTask;
import java.io.InputStream;
import org.grammaticalframework.Linearizer;
import org.grammaticalframework.PGF;
import org.grammaticalframework.PGFBuilder;
import org.grammaticalframework.Parser;
import org.grammaticalframework.parser.ParseState;
import org.grammaticalframework.Trees.Absyn.Tree;

public class Translate extends Activity
{
  private ArrayAdapter mArrayAdapter;
  private PGF mPGF;

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    new LoadPGFTask().execute();

    mArrayAdapter = new ArrayAdapter(this, R.layout.listitem);
    ListView list = (ListView)findViewById(R.id.list);
    list.setAdapter(mArrayAdapter);
  }

  public void translate(View v) {
    TextView tv = (TextView)findViewById(R.id.edittext);
    String input = tv.getText().toString();
    new TranslateTask().execute(input);
  }


  /**
   * This class is used to load the PGF file asynchronously.
   * It display a blocking progress dialog while doing so.
   */
  private class LoadPGFTask extends AsyncTask<Void, Void, PGF> {
```

```java
  private ProgressDialog progress;

  protected void onPreExecute() {
    // Display loading pop-up
    this.progress = ProgressDialog.show(
        Translate.this, "Translate",
        "Loading grammar, please wait", true);
  }

  protected PGF doInBackground(Void... a) {
    int pgf_res = R.raw.foods;
    InputStream is = getResources().openRawResource(pgf_res);
    try {
      PGF pgf = PGFBuilder.fromInputStream(
          is, new String[] {"FoodsEng", "FoodsCat"});
      return pgf;
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }

  protected void onPostExecute(PGF result) {
    mPGF = result;
    if (this.progress != null)
      this.progress.dismiss(); // Remove loading pop-up
  }
}

/**
 * This class is used to parse a sentence asynchronously.
 * It display a blocking progress dialog while doing so.
 */
private class TranslateTask
    extends AsyncTask<String, Void, String[]> {

  private ProgressDialog progress;

  protected void onPreExecute() {
    // Display loading pop-up
    this.progress =
        ProgressDialog.show(
            Translate.this, "Translate",
            "Parsing, please wait", true);
  }
```

```
    protected String[] doInBackground(String... s) {
      try {
        // Creating a Parser object for the FoodEng
        // concrete grammar
        Parser mParser = new Parser(mPGF, "FoodsEng");
        // Spliting the input (basic tokenization)
        String[] tokens = s[0].split(" ");
        // parsing the tokens
        ParseState mParseState = mParser.parse(tokens);
        Tree[] trees = (Tree[])mParseState.getTrees();

        String[] translations = new String[trees.length];
        // Creating a Linearizer object for the FoodCat
        // concrete grammar
        Linearizer mLinearizer = new Linearizer(mPGF, "FoodsCat");
        // Linearizing all the trees
        for (int i = 0 ; i < trees.length ; i++) {
          try {
            String t = mLinearizer.linearizeString(trees[i]);
            translations[i] = t;
          } catch (java.lang.Exception e) {
            translations[i] = "/!\\ Linearization error";
          }
        }
        return translations;
      } catch (Exception e) {
        throw new RuntimeException(e);
      }
    }

    protected void onPostExecute(String[] result) {
      mArrayAdapter.clear();
      for (String sentence : result)
          mArrayAdapter.add(sentence);
      if (this.progress != null)
          this.progress.dismiss(); // Remove loading pop-up
    }
  }
}
```

If you compile and test your application now, you should be able to translate sentences from the `Foods` grammar from English to Catalan. Feel free to play with other languages and other grammars.

Figure 2.6: *PhraseDroid icon picturing the "o" from the MOLTO project logo (`http://www.molto-project.eu/`).*

2.4 PhraseDroid

We developed a simple phrasebook application to demonstrate a possible use of the library. The application allows the user to enter simple sentences in a controlled language and translate them in different languages. This application is based on the MOLTO phrasebook project[8]. This is a relevant use case as it has a clear potential for usage because of the high quality of the translations and the variety of languages for which the grammar was devised. It is also worth mentioning that the reasonable coverage of the grammar makes the phrasebook applicable in many day-to-day situations for tourists traveling abroad.

To allow easy and fast input while restraining the user to the controlled language, we used an interface similar to the fridge magnets application[9]. This demonstrate the utility of predictive parsing on the cell phone. This feature is a great aid for users of a controlled language, since they can always be aware of the coverage, and the possibilities that the grammar offers. (See screen captures of the application in figure 2.7.)

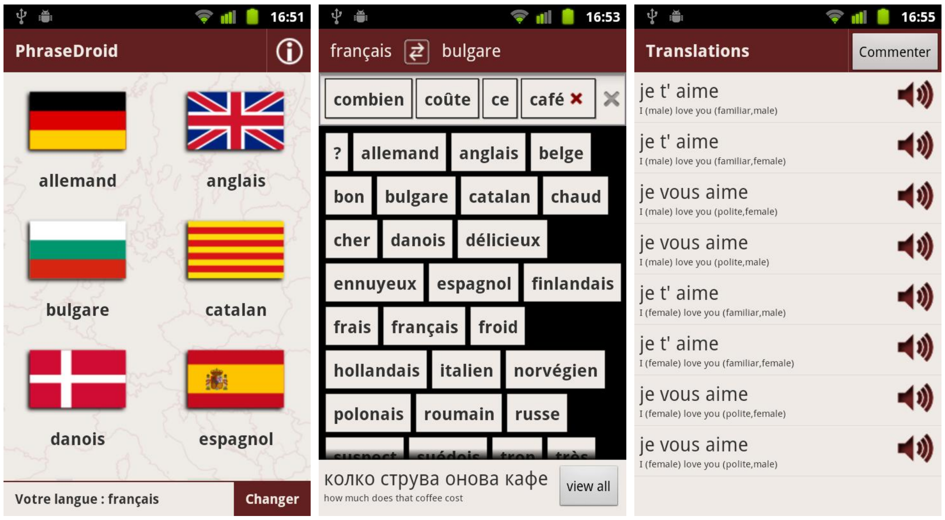
In addition, the Android platform provides services for high-quality voice synthesis for a number of languages, which can be plugged to the grammar applications. This gives our approach a great advantage over the traditional phrase books.

License: The application has been released under the terms of the GNU lesser general public license, same as the JPGF library itself, and the source code is available at `https://github.com/GrammaticalFramework/PhraseDroid`.

Distribution: PhraseDroid has been submitted to the Android Market (now Google Play) and made available at no cost (Figure 2.8). During the first six months, the application was installed by more than 300 users. Figure 2.9 shows the number of devices on which the application is installed at time t (Number of installation minus number of removal until t.)

[8]`http://www.molto-project.eu/demo/phrasebook`

[9]The original fridge magnets application at `http://tournesol.cs.chalmers.se:41296/fridge` is no longer available but a more recent version, the "GF minibar" can be found at `http://cloud.grammaticalframework.org/minibar/minibar.html`



колко струва онова кафе
how much does that coffee cost

je t' aime
I (male) love you (familiar,male)

je t' aime
I (male) love you (familiar,female)

je vous aime
I (male) love you (polite,female)

je vous aime
I (male) love you (polite,male)

je t' aime
I (female) love you (familiar,male)

je t' aime
I (female) love you (familiar,female)

je vous aime
I (female) love you (polite,female)

je vous aime
I (female) love you (polite,male)

Figure 2.7: *Screen captures from the PhraseDroid app. From left to right: the home screen where the user can choose the target language (by default the source language is set to the language of the phone interface but it can also be changed); the "fridge magnet" interface demonstrating the application of predictive input; and the result page showing all possible translations of the input sentence, with disambiguation information.*

2.5 Related work

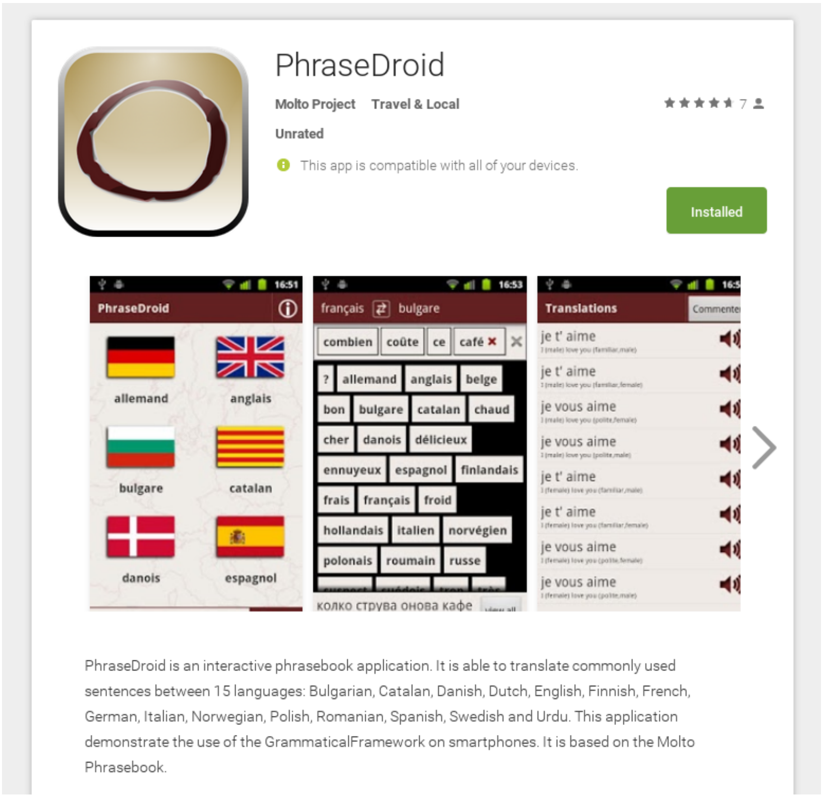
Java implementation: Preexisting work includes an embedded GF interpreter written in Java. Among other differences, this implementation uses an earlier algorithm than the one describe in Angelov 2009 based on approximating of the GF grammar by a (potentially over-generating) context-free grammar. It was also written for an earlier version of the PGF format (the binary format used by GF to represent compiled grammars). For more information, see Bringert 2005.

C binding: Since this work was conducted, a new runtime has been implemented in the C programming language named `libpgf` and the recommended way to use a PGF file in Java is to use the Java binding to the C library, see *GF Developers Guide* for more information. This provides much better performances and reduces the maintenance effort (as only one runtime needs to be maintained) at the expanse of portability, as one need to build the C library for every particular architecture.

2.6 Conclusion and acknowledgments

This chapter gave an overview of JPGF, a PGF runtime for the JVM and provided a tutorial on how to use it to build a PGF based android application.

We would like to thank Krasimir Angelov for his explanation of the GF algorithms.



PhraseDroid is an interactive phrasebook application. It is able to translate commonly used sentences between 15 languages: Bulgarian, Catalan, Danish, Dutch, English, Finnish, French, German, Italian, Norwegian, Polish, Romanian, Spanish, Swedish and Urdu. This application demonstrate the use of the GrammaticalFramework on smartphones. It is based on the Molto Phrasebook.

Figure 2.8: *Screen capture of the Android app-store from Google (Google Play) where the PhraseDroid application was released and made available at no cost.*

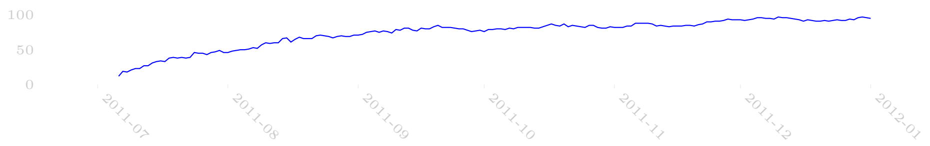


Figure 2.9: *Evolution of the number of PhraseDroid installations according to the data collected by Google Play.*

Chapter 3

A GF Mailing list

Communication is essential in community. It is the metaphorical highway that connects the many towns and people in your world. Effective communication brings together your community members in a manner that is free-flowing, productive, and accessible.

Bacon, *The Art of Community*

This chapter describe the usage, implementation and evaluation of the *GF user and developer mailing list*[1]. Before the GF mailing list was created, there were two ways to contact members of the community: contacting them directly using their personal email address or using the *GF Resource Grammar Summer School 2009* mailing list[2]. While the later solution is very similar to the one described here, as the name indicates it has a different scope: it was used to conduct an on-line course in resource grammar development prior to the summer school event of 2009 and for practical questions concerning the event itself (travel, accommodation, sharing pictures…). In addition, its name didn't make it clear that this was a place where one could receive help about GF in general. The earlier solution, contacting community members directly, suffers from different problems, in particular scalability and the lack of a public archive (see § 3.1).

In § 3.1 we first describe what a mailing is and what advantages it brings to an free-software community; in § 3.2 we describe the technical choices made to implement the GF mailing list; in § 3.3 we rapidly go through how one can subscribe to the mailing list and contribute to discussions; in § 3.4 we report some usage statistics and finally, in § 3.5 we conclude with some recommendations on creating such a list.

3.1 What's a mailing list

In Fogel 2014, a mailing list is described as "the bread and butter of project communications". The principle of a mailing list is very simple and probably familiar to most Internet users today: a new email address is created and is associated with a list of

[1] `https://groups.google.com/forum/#!forum/gf-dev`

[2] `https://groups.google.com/forum/#!forum/gf-resource-school-2009`

subscribers (also by their email addresses). When a message is sent to the list's address, it is forwarded to all the subscribers addresses. Most list management softwares offer extra functionalities like automatic email and/or web based subscription, digest-mode (the subscriber receive only one mail per day, with all the day's list activity), moderation features (more or less automatized, those are there to reduce Spam and abuse), header manipulation (the answer to a message should go to the whole list and not just the original poster) and archiving.

The list archives are, in our view, one of the main benefits that a proper mailing list brings to a free-software community. But first and foremost, the list provides a public forum to share information and ask and answer questions. As a project grows and the number of questions, support and feature requests and information queries grows, it is important to avoid the unsustainable situation where only one or two people, usually the same who are also maintaining the software, are in the position to answer.

The list archives play an central role in this by making available a public and searchable record of those discussions. This allows a community member to search in previous conversations if their particular question has already been answered. The mailing list archive also serves as a repository of decisions taken by the community and the discussions that led to these decisions (e.g. conflict resolution, direction of the development, etc.) acting as a sort of "commons law" for the community (Fogel 2014).

3.2 Implementation

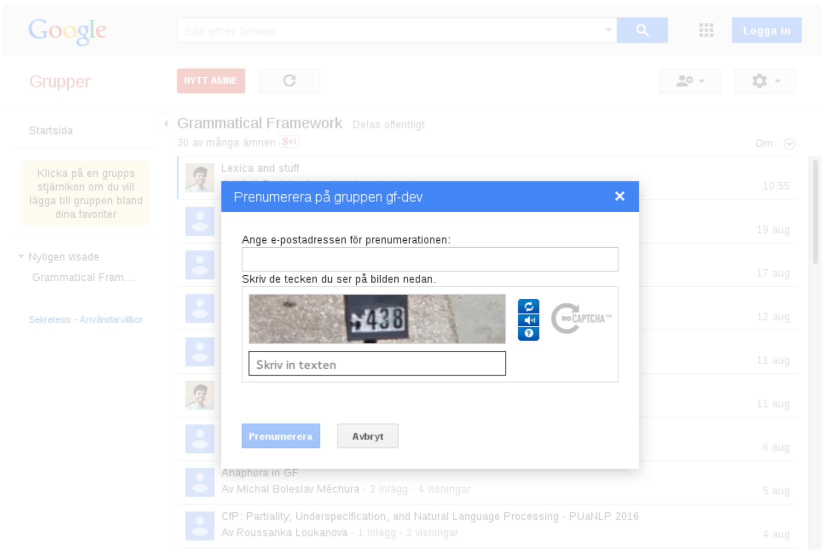
As described above, most list management softwares provide the same basic set of functionalities. For the GF mailing list implementation, Google Groups, a hosted solution provided by Google, was chosen instead of hosting a list management software on a local server[3] Not only the user interface makes it easy to create a list, but more importantly it was already used by the GF community for the *GF Resource Grammar Summer School 2009* mailing list.

We have nonetheless stumbled upon two drawbacks to using Google Groups instead of hosting the mailing list ourselves:

- The lack of control over the future of the mailing list: both in term of features, which are sometimes added or removed by Google without any choice offered to the list users, but also in term of existence, as Google has become known for terminating services they do not consider profitable enough.
- the impossibility of downloading the lists archives for off-line use (e.g. metrics computation, see § 3.4). Most free-software mailing list management systems offer the possibility to download the list archives, for instance in the common `mbox` format but to our knowledge this feature is absent from Google group.

The list was configured to be public with limited moderation. In particular:

[3]It has come to our knowledge that both Chalmers University of Technology and University of Gothenburg, our home universities, offer mailing-list hosting (resp. `https://lists.chalmers.se/` and `https://listserv.gu.se/`). There weren't considered at this time as we were unaware of their existence.


Google
Sök efter ämnen
Logga in
Grupper
NYTT ÄMNE
Startsida
Klicka på en grupps stjärnikon om du vill lägga till gruppen bland dina favoriter
Nyligen visade
Grammatical Fram...
Sekretess - Användarvillkor
Grammatical Framework Delas offentligt
30 av många ämnen
Om

Lexica and stuff — 10:55
19 aug
17 aug
12 aug
11 aug
11 aug
6 aug
Anaphora in GF
Av Michal Boleslav Měchura · 3 inlägg · 4 visningar — 5 aug
CfP: Partiality, Underspecification, and Natural Language Processing - PUaNLP 2016
Av Roussanka Loukanova · 1 inlägg · 2 visningar — 4 aug

Prenumerera på gruppen gf-dev

Ange e-postadressen för prenumerationen:

Skriv de tecken du ser på bilden nedan.

Skriv in texten

Prenumerera — Avbryt

Figure 3.1: *Screen capture of the mailing-list join form. In addition to providing the email address to subscribe to the list, one must also fill-in a test designed to prevent spamming software from subscribing automatically.*

- everyone can see the list's messages and archives
- everyone can subscribe to the list
- messages from non-members and recent members are moderated (to prevent spam and other abuse of the list).

3.3 Usage

3.3.1 Subscribing

On the web: Navigate to `https://groups.google.com/forum/#!forum/gf-dev/join` and enter your email address. You might also need to pass a test designed to prevent spammers from automatically subscribing to lists (see figure 3.1).

Via email: Send a message to `gf-dev+subscribe@googlegroups.com`. Neither the subject nor the content of the message have an importance for this to work. The system assumes that the provenance address (`from`) is the one that should be subscribed to the mailing list. Because emails from most addresses are easy to forge, a mail is sent to the address to be subscribed asking to confirm the suscription request. Confirmation can be given either by clicking on a link included in the confirmation email or by responding.

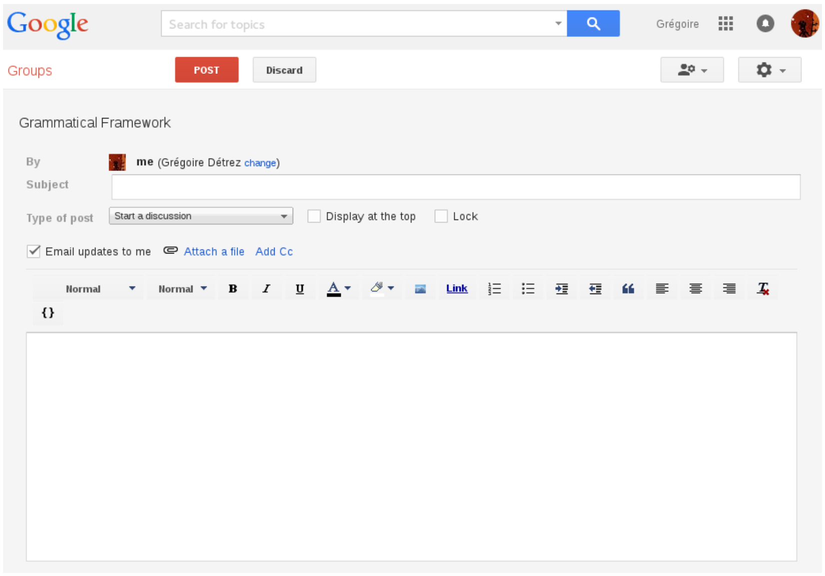


Figure 3.2: *Screen capture of the mailing-list posting interface. This allows subscribers to the mailing list who do not want to get the list messages in their email in-box to use the mailing list as a web forum.*

3.3.2 Posting

On the web: Once you are subscribed to the list and authenticated on Google Groups, go to `https://groups.google.com/forum/#!newtopic/gf-dev` to start a new discussion thread (see figure 3.2). Answering existing threads is done in a similar interface after clicking the "post reply" button.

Via email: Send a new message to `gf-dev@googlegroups.com`. Answering to an existing post is similarly done by answering the corresponding email, assuming that you have set your list preference to receive each post as an individual email. Note that, answering to a list message can be ambiguous: the answer can go to the whole list, to the original poster only or to both (for instance if the original poster is not a member of the list). The default action implemented by the "reply" button depends on the software used to read and write emails.

3.4 Statistics

We collected list statistics for the first six years of existence (October 2010 to August 2015). To gather the statistics, the mailing list archives have been downloaded using a

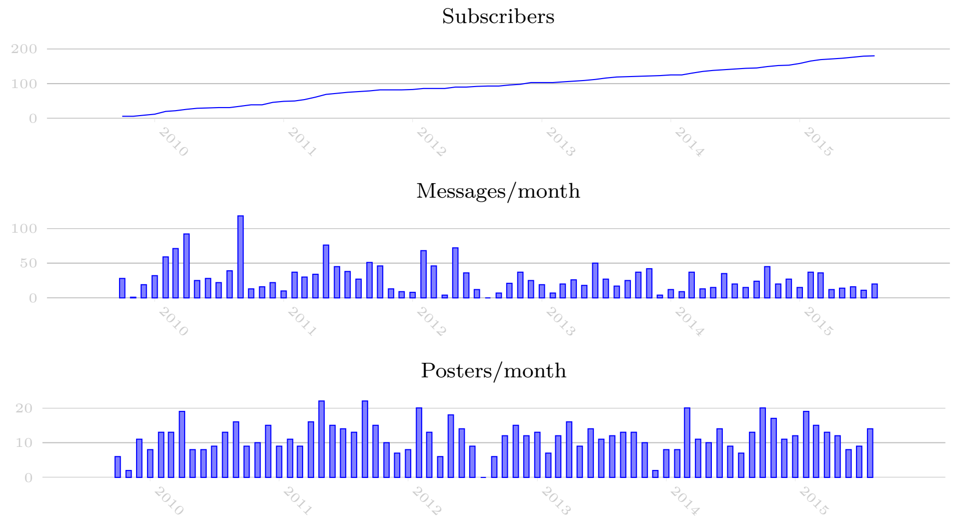


Figure 3.3: *GF mailing list metrics. Top: the evolution of the number of subscribers. Middle: the number of messages per month. Bottom: the number of unique posters per month (i.e., the number of people that have posted at least one message to the mailing list during a given month).*

custom build web crawler[4] and the `mlstats`[5] tool was used to index the messages.[6]

Figure 3.3 shows the evolution of three different metrics: on top, the evolution of the total number of subscriber is plotted and one can see that the number have been steadily growing to reach 182 members at the time of this writing. In the middle, the number of messages to the list per months with an average of 29 messages/month and a peak at 118 messages/month in October 2010. Finally, on the bottom the number of posters (people posting to the list) per month with an average of 12 posters/month.

3.5 Conclusion

We have seen how and why the GF mailing list was created and how much it was used. As shown in Section 3.4, the list has been growing continuously with constant activity since its creation. We conclude that it has been a very useful tool to grow the GF community

In 2013 members of the community started an IRC channel for instant discussion[7], thus completing the "minimum, standard set of tools for managing information [in an

[4]as mentioned in Section 3.2, Google Groups does not provide archive downloads to the contrary to many free/libre and open source alternative

[5]`https://github.com/MetricsGrimoire/MailingListStats`

[6] The analysis of the data was conducted using the python programming language and is available as a Jupyter Notebook
`https://nbviewer.jupyter.org/gist/gdetrez/463ee4cd129448fa9e7d405a9873a044`

[7]the #gf channel on Freenode

open source project]” (Fogel 2014): website, mailing list, version control, bug tracking and real-time chat.

We would recommend someone else thinking about setting up a mailing list for a community to consider hosting it under their project’s domain to keep a better control over the list future and the list management software features.

Chapter 4

A Build Server

4.1 Introduction

When writing software together, a problem quickly appears: once a developer have added a feature to the software, they need to integrate those changes to the main code repository, a process known as *system integration*. If they are the only active developer, or if no one else has made changes to the code since they started working on their feature, integration is easy. If, on the other hand, other developers have also added features or modified the code base in parallel, it may become really complicated as the changes from one developer may break the code written by an other. In the developer community, this problem is known as *integration hell*.

This problem is not specific to open-source software but it grows with the number of developers involved and it can become a bottleneck to growing a contributor community.

Several solutions have been proposed to counter this problem. One of them is *module ownership*, where the software is split into "modules" which are arbitrary divisions in the source code (e.g., a network module, server module, etc.) and one person is assigned to be the *owner* of each module. The owner of a module is the only one who can make significant changes (or, in some cases, any changes) to the corresponding source code. This has been reported to work in software-development companies but is ill adapted in free software communities. Nowadays it is increasingly becoming an obsolete practice in industry as well as more and more companies move to more agile methods to developing software.

An other method, which is the one advocated by agile proponents is *continuous integration*. The idea between continuous integration is really simple: every developer should integrate their code frequently, usually at least once a day, so that integration problems are detected early. To get an intuition on how this might solve the problem we can look at real-time collaborative editors, such as *Google Docs*: by decreasing the time between integrations to (almost) 0, conflicts also appear in real time and can be located and resolved immediately, or avoided altogether.

This method introduces a new problem though. When everyone was working on their own branch of the code any change that would break an important feature or prevents the code from being compiled at all would only affect this single developer on their own machine. In a continuous integration setting on the other hand, if a developer breaks the code base this affects all the developers who have integrated their changes since the faulty commit. Not only it might prevent them from continuing their work but they also might think that they are the one responsible from the problem.

A common recommendation to solve this problem is to have an *automated build server* checking the code automatically: as soon as a developer pushes a change to the shared repository, the automated build system checks out the new version and runs a series of predefined tests. If the tests detect an issue with the code, the automated build system notifies the developers. Then the one who made the changes can investigate to fix the problem while the other developers known that they should wait for the issue to be resolved before integrating their own changes. In this chapter, we describe the implementation of such a server for the Grammatical Framework community.

§ 4.2 describes the implementation of the continuous build server using Jenkins; § 4.3 explains how the GF GitHub mirror is updated; § 4.4 shows how the same automation architecture has been used to collect and update metrics and status reports about GF; § 4.5 presents future work; and finally § 4.6 concludes this chapter.

4.2 Implementation

An automated build server is not a complicated piece of software: some event (the *trigger*), which may be a new commit push, an email or a timer, causes the server to download the latest version of the source code and to execute a predefined set of commands (the *build script*); finally, a report is produced from the commands' output.

Although one could create a simple automated build server in a few hours using standard Unix tools (such as `cron`, `bash`, `sendmail`, etc.), it is preferable to use a specialized tool as they include useful additional features like:

- Master/slave architecture: one process, the master, is responsible for starting build tasks on slave processes that can either run on the same machine or be distributed to other computers. This can be used to parallelize long build tasks but also to run jobs in different contexts (different architecture, operating systems, compiler versions, etc.) The master collects the results from the slaves and generates the report.
- Archiving of build reports and artifacts
- Advanced built-in reporting tools that can publish reports via email, IRC or web pages and provides the ability to collect and visualize metrics from the commands' output (e.g. test results trends, warnings, build times, etc. see figure 4.1 for an example).

Jenkins Several tools were considered to implement the GF build server. The first criterion was that we needed a solution that integrates with the existing tools used by

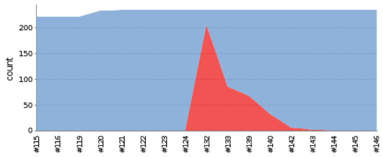


Figure 4.1: *Example of graph produced by the reporting tools included in Jenkins. This represents the number of automated tests executed for each build, where the red areas are failing tests and the blue areas are successful tests. (Note: this particular graph is shown as an example but it is not related to GF.)*

GF developers (for instance, not all continuous build server are compatible with Darcs, the version control system used to develop GF). We also wanted to use free software as much as possible, which led to two final candidates: Jenkins[1] and buildbot[2], among which Jenkins was chosen as it was easier to install and provided greater flexibility thought the wealth of available plugins.

Jenkins was originally created at Sun Microsystems by Kohsuke Kawaguchi as the Hudson project. In 2010, after Sun had been acquired by Oracle, the community split following a dispute over licensing issues and the creator, together with many contributors renamed the project Jenkins. (Oracle also continued the development independently under the trademarked Hudson name.)

At the time of this writing, the latest version of Jenkins is 1.627 and 1076 plugins are available.

Trigger: For the GF automated build we are using the repository as the source of the trigger: each time someone pushes new changes to the main repository, Jenkins is notified. Listing 4 shows the command which is executed as a hook[3]in the main Darcs repository.

```
apply posthook curl --max-time 4 --insecure --silent \
  https://ci.zjyto.net/job/GF/job/trigger/build?token=<secret-token>
```

Listing 4: The trigger for the GF automated build system. We use the tool `curl` to notify the build server via the HTTPS protocol. The URL is determined by our Jenkins installation where `ci.zjyto.net` is the host name of the Jenkins server and `<secret-token>` is a placeholder for a randomly generated token.

Build script: The build script used in Jenkins for GF is given in Listing 5 The first line creates a new cabal sandbox, which is an isolated environment in which dependencies can be installed. The second lines installs the necessary dependencies. The third line

[1]`http://jenkins-ci.org/`

[2]`http://buildbot.net/`

[3]Hooks are a way to trigger the execution of user-written scripts when certain events occur and are available in many version control systems

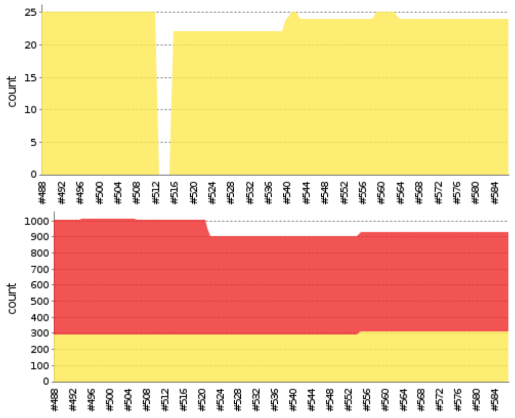


Figure 4.2: *Example of trend graphs generated by Jenkins after a GF build. Top: the number of warnings printed during compilation. Bottom: the number of task codetags (`TODO` and `FIXME`) found by scanning the source code.*

builds GF itself using the standard `cabal build` command. The last line creates a source distribution, which is a compressed archive of the sources that can be distributed to users.

```
cabal sandbox init
cabal install --only-dependencies \
  --build-log=install-dependencies.log
cabal build
make sdist
```

Listing 5: GF continuous build script

Report and artifacts Once the build script has terminated, Jenkins collects informations and artifacts to archives. By default, only output printed during the build script is archived; for the GF build, we have extended this behavior in the following ways:

- archive and fingerprint the source distribution as an artifact,
- collect warnings printed during compilation,
- scan the source files for *codetags*: `TODO` and `FIXME`, indicating respectively missing or broken features.

From the collected data Jenkins automatically produces "trend graphs", see Figure 4.2 for an example.

When the build terminates without problems, the report is only produced as an HTML page, otherwise (if any steps of build script fails) an email is sent to the developer who pushed the last commits, which caused the build to fail.

In the next two sections, we present other tools and services that were possible to implement by taking advantage of the automated build server.

4.3 Github code mirror

As mentioned in the previous section, the GF developers use Darcs as the version control system to maintain the source code. This choice has sometimes been criticized by member of the community[4]. While the criticism sometimes targets Darcs itself—mostly for its (lack of) speed—most are caused by external factors, related to the relative obscurity of Darcs compared to more popular tools such as Git[5]. One problem of using a less popular tool is the lack of familiarity of most potential community members. And although distributed version control systems, of which both Darcs and Git are examples, share a basic model and set of commands, there is enough differences in the underlying assumptions and the effects of the commands to make it less than trivial to move from one tool to an other without difficulties. Another common criticism is the lack of tooling. While there is a wealth of tools that have been built around git both for collaboration (code-sharing services such as Gitlab offering an online interface to browse the source code and many associated tools such as issue trackers, wikis, etc.) and for personal use, such as the many different graphical and command line tools available to work on git repositories as well as integration in IDEs. The choice of tools available for Darcs repositories is much more limited. For instance, Darcs support was a limiting factor in the possible choices for a build server software for GF and although Jenkins does support Darcs, the integration is much less extensive than with Git.

To provide a partial solution to those complains, we decided to create a mirror of the GF Darcs repository using Git. This mirror should not only include the latest version of the source code but also the commit history with all the associated meta-data.

The implementation is done as a new Jenkins job: the tool `darcs-fast-export` is used to export the Darcs repository in the fast-import stream format which is then imported by git using the `fast-import` command.[6][7]

This approach has some limitations thought: as the conversion mechanism can only handle a one-way synchronization automatically, the git repository can only be used as a read-only mirror of the Darcs repository. So while it makes it easier for users familiar with git to get the latest version of the code it does not lower the barriers for them to contribute. In addition it makes it impossible to accept contribution through the Github

[4] See `https://groups.google.com/d/topic/gf-dev/zSU0lqboRy8/discussion` for a recent discussion about this on the mailing list.

[5] According to data collected by the Debian distribution, the number of people who regularly use Darcs is more than two orders of magnitude higher than regular Darcs users. Source: Debian Popularity Contest.

[6] `darcs-fast-export` is available at `https://github.com/warner/darcs-fast-export`. Note that more recent version of Darcs include a new command `darcs convert export` that directly export the repository in the fast-import protocol. This should allow us to update the mirror without the need for an external tool but at the time of this writing, this new version is not yet available on the system that runs the build server.

[7] See `https://git-scm.com/docs/git-fast-import` for both a description of the `git fast-import` command and of the data format.

platform (so called "Pull requests").

4.4 Continuous Evaluation

It is common for build servers to be tasked with keeping track of measurements. By measurement, we mean "the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way so as to describe them according to clearly defined rules." (Fenton and Bieman 2014).

This is a rather broad definition of measurement. Indeed, according to this definition, the simple status of a build job (whether it succeed or not) is a measurement with a nominal scale that can take one of the two values PASS or FAIL[8].

As argued above, even the build status can be seen as a measurement. Other measures commonly tracked by build servers are: build time, number of tests executed, number of tests that succeeded/failed, disk space used, artifact fingerprints; etc.. We gave two example of measurements that are used in GF in the previous Section: the number of warnings produced by the compiler and the numeber of open tasks indicated in the source code (see Figure 4.2).

While some of those measurements are tracked by Jenkins itself, most are available as plugins that either track one or more specific measurements (e.g., the Disk Usage or the Static Code Analysis Plug-ins) or provide the ability to track custom measurements (e.g., the Plot Plugin that tracks and plot numerical measures). It is also possible to integrate the build server with specialized tools that provide more advanced functionalities to collect, aggregate, analyze and visualize measurements (one such tool is SonarQube).

While those tools mainly implement software engineering metrics, which measure either the code and artifact themselves (static analysis) or some runtime parameters (memory used, benchmark, etc.), researchers have proposed many specific metrics for the evaluation of natural-language processing systems. However those measurements are only done very occasionally when needed to present new research findings that are deemed worthy of publication.

Instead we would like to suggest that those evaluations should be used in the day-to-day work of maintaining software systems and that, if properly automated, they can yield greats benefits. By analogy to continuous integration and continuous delivery, we propose to call with practice *continuous evaluation.*

For developers, the repeated evaluation can be seen as a form of regression testing: if the system is found to perform worse than it did in a previously published evaluation, it probably indicates that a unwanted change has been made or that a change has had unwanted side-effects. As such, the same arguments that are commonly put forward to support tho integration of automated regression tests in a continuous integration pipeline are valid here as well. By running the evaluation often (ideally on each commit separately) the exact change or changes in the code that caused a change in the measurements, thus facilitating the 'debugging" process. On the other hand, it provides a safety net

[8] This is a bit of a simplification, at least in the case of Jenkins in which builds can have a few more possible results such as UNSTABLE (somewhere between PASS and FAIL, can be used for example to indicate that the compilation succeeded but some automated tests failed), ABORTED for manually aborted builds or NOT_BUILT if the build was skipped for some reason.

to maintainers and developers who, knowing that changes in the performances of the evaluated functionality would be caught by the automated build server, might be more confident in making large changes to the code.

But continuous evaluation is also, we believe, beneficial to the research community in general. Of course, it is nice to be able to check that ones own results published some time ago are still valid in the latest version of the evaluated system by checking the build server's report page, but this becomes a more capital question when it comes to building upon ones or someone else's results. Without the use of continuous evaluation, the researcher who wishes to build upon previously published results is left with three choices: 1. using the exact same version as the original study. While easier for the authors of the original study, this presents multiple difficulties. First, papers not always give enough details to be able to get the exact same version with confidence. Then, the version might not be available anymore. This is less a problem with free software where there is a public version control repository that contains the whole history of the code, but it is common for nonfree software where the publisher only offer the latest version. The original author might have made some modifications to the code or the configuration that weren't published. Then, even when available this version might not work anymore on recent operating systems or newer hardware (or the published results somehow depends on those, such as benchmarking results) which leads to the never-ending quest of reproducing the exact environment in which the original results were obtained. Finally, using an old version prevents the researcher from beneficing from bug fixes and using the latest features of the system and if the new research depends on those added features, it make this option impossible.

2. Reproducing the original evaluation with the newer version. This can be very easy or very hard depending on the level of details given in the publication and whether the original researcher shared their analysis and data in a reproducible format. This might thus lead to substantial albeit avoidable work duplication.

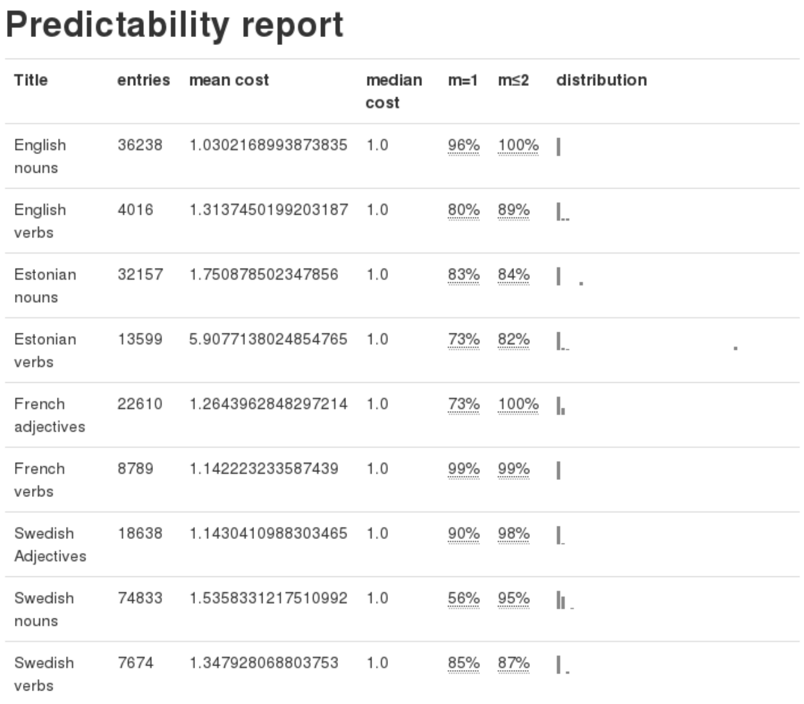
3. if none of the above are possible, the last option is to use the latest version with the hope that previously published results still hold.

On the contrary, the use of continuous evaluation makes it possible and easy to check that published studies are still reproducible on the new version and see if ary of the results have evolved.

Another benefit is that it encourages a culture of publishing more reproducible results through automation and code sharing.

As an first example, we have automated the evaluation of GF smart paradigms from the study published at EACL 2012 (Détrez and Ranta 2012). Since this is more computationally intensive it is only ran once a week on the latest version of GF from the code repository but the results (smart paradigm predictability) are archived and can be tracked over time. Figure 4.3 shows an example of the generated predictability report.

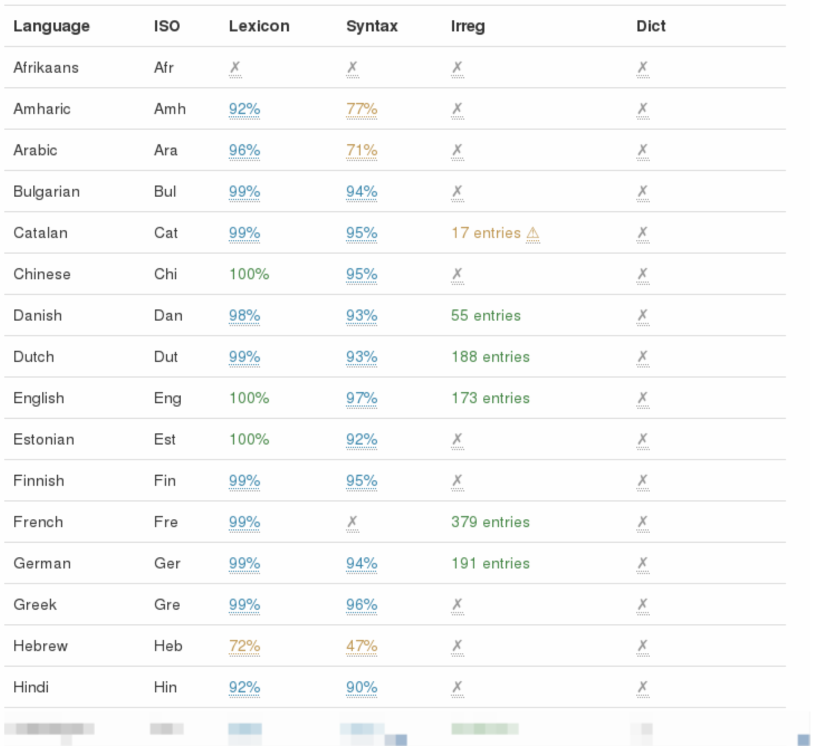
As a second example, we have created an automatically updated version of the GF status page. One of the big contribution of GF is the development of the Resource Grammar Library (Ranta 2009). While at the time of this writing, there are officially 37 languages available in the resource grammar library, they greatly vary in their level of completeness. For instance, some modules may be partially implemented in some language, or not at all in others, and existing monolingual dictionaries vary a lot in size



Predictability report

Title	entries	mean cost	median cost	m=1	m≤2	distribution
English nouns	36238	1.0302168993873835	1.0	96%	100%	
English verbs	4016	1.3137450199203187	1.0	80%	89%	
Estonian nouns	32157	1.750878502347856	1.0	83%	84%	
Estonian verbs	13599	5.9077138024854765	1.0	73%	82%	
French adjectives	22610	1.2643962848297214	1.0	73%	100%	
French verbs	8789	1.142223233587439	1.0	99%	99%	
Swedish Adjectives	18638	1.1430410988303465	1.0	90%	98%	
Swedish nouns	74833	1.5358331217510992	1.0	56%	95%	
Swedish verbs	7674	1.347928068803753	1.0	85%	87%	

Figure 4.3: *Example of the predictability report generated by the Jenkins job.*



Language	ISO	Lexicon	Syntax	Irreg	Dict
Afrikaans	Afr	✗	✗	✗	✗
Amharic	Amh	92%	77%	✗	✗
Arabic	Ara	96%	71%	✗	✗
Bulgarian	Bul	99%	94%	✗	✗
Catalan	Cat	99%	95%	17 entries ⚠	✗
Chinese	Chi	100%	95%	✗	✗
Danish	Dan	98%	93%	55 entries	✗
Dutch	Dut	99%	93%	188 entries	✗
English	Eng	100%	97%	173 entries	✗
Estonian	Est	100%	92%	✗	✗
Finnish	Fin	99%	95%	✗	✗
French	Fre	99%	✗	379 entries	✗
German	Ger	99%	94%	191 entries	✗
Greek	Gre	99%	96%	✗	✗
Hebrew	Heb	72%	47%	✗	✗
Hindi	Hin	92%	90%	✗	✗

Figure 4.4: *Example of the status page automatically generated by the Jenkins job.*

and coverage.

To give a better overview of the status of different languages, a web-page exists that gives qualitative information about the languages of the RGL, such as the existence of different modules, whether the grammar has been tested in an application and whether it is associated with a existing publication.

This page is helpful but we have identified the following two potentials problems with it: it has to be kept up-to-date manually and it lacks in precision.

We created a tool that generates a similar page based on information available in the GF source repository. While some of the information is missing (such as the publication in which the grammar was presented) it has the advantage that the present information is automatically updated and with an increased precision (percentage of completeness for standard modules and number of entries for dictionaries). Figure 4.4 shows an example of the generated status page.

4.5 Future Work

Although the amount of automation is modest, the build server has already been useful in maintaining the quality of the code by catching mistakes such as missing files or syntax errors. The checks should progressively be extended to cover more and more quality aspects of the code.

The first possible direction is to improve the guarantee offered by the automated build server by adding a robust automated test suite. This would guarantee not only that the code can be compiled but that the basic functionalities work as expected. As automated tests also become code that also has to be maintained, there is a trade-of between the stronger guarantees that a more extensive test suite offers and the work required to create and maintain it. On way to find a good balance is to first conduct a risk analysis to determine which part of the code would benefit most of automated testing and what level of testing is appropriate (unit, integration or system testing, see ISO/IEC TR 19759:2015 for a standard definition of test levels).

Continuous delivery offers a second direction for future work. As popularized by Humble 2010, continuous delivery is a software engineering practice where the delivery pipeline, the different steps leading from the source code to software that can be delivered to end users, is entirely automated and is run on every source change. This often begins with the same tasks as continuous integration (building the software and running automated tests) but also requires extended system/acceptance testing that are run in different contexts (different operating systems, architectures, etc.) and a fully automated packaging process.

Finally, we would like to extend the work started here with continuous evaluation and automate more research evaluations. For example we could be measuring BLEU score (Papineni et al. 2002) or other relevant translation metrics between pairs of language, or compute grammar coverage on existing corpus, etc.. We believe that this can brings two benefits: first it helps maintaining the hight quality of GF as a tool upon which to build further research. Second, the automation of research evaluations makes the research more reproducible and guarantees that the published results are still valid in ulterior versions of GF.

4.6 Conclusion

In this chapter, we reviewed the benefits of a build server in a collaborative software-engineering project and presented the implementation of one for GF. We have also outlined other uses of the build server beyond just building and testing software.

We conclude that a build server is a beneficial tool in a collaborative software-engineering effort and that the existing tools are mature enough to make its deployment well worth the effort. In addition, in the particular case of software produced as part of a research project, the build server can be used to implement *continuous evaluation* and to encourage reproducible results.

Chapter 5

A GF document translator

While GF can translate plain text, translating an existing document often requires, in addition to translating the textual content itself, to preserve the structure of the document (titles, paragraphs, lists, etc.) and transfer in-line annotations such as emphases, links or colors.

In this section, we present `gfdt`, a tool that automates the translation of fully formatted documents using GF to translate the textual content and `pandoc` to parse the document structure and inline formatting. In §5.1 we present the idea and related work, in §5.2 we give a short introduction to the tool usage and in §5.3 we describe how `gfdt` works.

5.1 Idea and related work

The Apertium machine translation toolkit (Forcada et al. 2011), supports the process of translating formated documents, which in the documentation is referred to as format handling. For each supported format, Apertium provides a pair of utilities `apertium-desFORMAT` and `apertium-resFORMAT`, for instance respectively `apertium-deshtml` and `apertium-reshtml` for HTML formated documents. The first utility encapsulates formatting information in so called "superblanks" which are then seen as spaces by the translation pipeline, and thus ignored. The second utiliy restores the formatting from the superblanks.

One limitation with this approach is that in-line formatting (e.g. a single word in italics inside a sentence) is not correctly restored in case of reordering. It happens for instance that some adjectives have to be moved after the noun when translating from English to French, in which case the rendered formatting on the translated string may be incorrect. For instance the English "a **pointy** hat" when translated to French with Apertium using format handling produces "un **chapeau** pointu" instead of the expected "un chapeau **pointu**".

In the implementation of `gfdt`, we solved this problem by using the alignment provided by GF to transfer the formatting from the source document to the correct place in the target document (see §5.3 for more details).

5.2 Usage

Installation: To build this project, a fairly recent version of the haskell platform is required.

At the time of this writing, `gfdt` depends on a patched version of GF, so a simple cabal install in the root of the repository does not work. You can use the included scripts/bootstrap script to install the dependencies:

```
$ ./scripts/bootstrap
$ cabal build
```

Usage: `gfdt` does not know how to parse formatted document by itself. Instead it is meant to be used in conjunction with the excellent tool `pandoc`[1].

The following pipeline demonstrate how to parse a document formatted using Markdown[2] (as indicated by the `.md` file extension), translate it using the GF grammars in `Foods.pgf` using `FoodsEng` as the grammar for the source language and `FoodsFre` as the grammar for the target language (that is, translating from English to French). Finally, the document is saved, still in the Markdown format, in a new file `mondocument.md`.

```
$ cat mydocument.md
this **warm** cheese is *very* boring
$ pandoc mydocument.md -t json \
  | gfdt --pgf=Foods.pgf --from FoodsEng --to FoodsFre \
  | pandoc -f json -o mondocument.md
$ cat mondocument.md
ce fromage **chaud** est *très* ennuyeux
```

5.3 Implementation

The result of `pandoc` parsing the formatted document is a syntax tree representing the structure of the document (not to be confused with GF abstract syntax trees that model the syntactic structure of sentences). Nodes in the tree represent either structural elements (titles, paragraphs, list items, etc.) or formatting elements (bold, italics, links, etc.).

This distinction between structural elements and formatting elements is important as `gfdt` treats them differently: The content of each structural element is translated as a separate unit of text and the translated text is inserted at the same place in the output document tree whereas formatting elements are extracted from the text in which they appear, which is then translated as a whole and then the formatting is re-applied to the translated text using the translation alignment to adjust their position. Figure 5.1 summarize this process.

This last part (extracting and re-applying formatting elements) is the most complex operation and probably deserves a bit more explanation. Considering the example above:

[1] `http://pandoc.org/`
[2] `http://commonmark.org/`

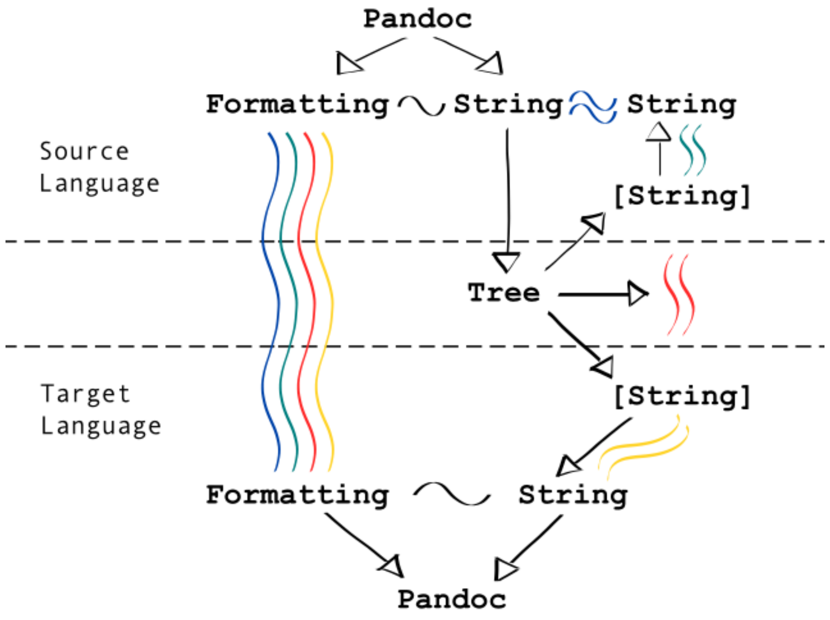


Figure 5.1: *Diagram depicting the way* `gfdt` *transfers formatting instruction. Alignments are computed between different steps of the translation process (the colored wavy lines on the right) and then, combined, they are used to transfer the text formatting (the multi-color lines on the left).*

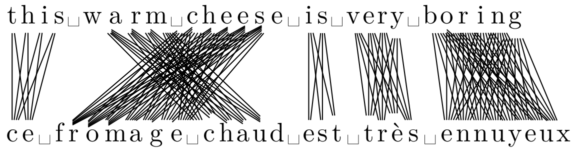

this␣warm␣cheese␣is␣very␣boring
ce␣fromage␣chaud␣est␣très␣ennuyeux

Figure 5.2: *Final character-level alignment between the source and target strings.*

(4) this **warm** cheese is *very* boring

The extraction process produces on one hand a version of the text without any annotation: `this␣warm␣cheese␣is␣very␣boring␣` and on the other hand a data structure with the formatting annotations and their original position in the input:

```
<5,9,bold>, <20,14,italics>
```

Then, the GF API is used to parse the sentence and obtain an abstract-syntax tree. From this, we can generate a token-level alignment between the tree's linearization in both the source and the target language:

```
[this,warm,theese,is,very,boring]
[ce,fromage,chaud,est,très,ennuyeux]
[<0,0>,<1,2>,<2,1>,<3,3>,<4,4>,<5,5>]
```

Because the linearization of the parse tree in the source language might differ from the original string, we use the Smith–Waterman algorithm to align them together on the character level.

By combining this alignment with the token-level alignment returned by gf, we obtain a character-level alignment between the source string and the translated string. Note that because the alignment returned by GF is at the token level, the final alignment align each character of one input token too every character of the corresponding target token, see example in Figure 5.2.

We can now use this character level alignment to translate the ranges to which the formatting extracted elements apply:

(5) ce fromage **chaud** est *ennuyeux*

Finally, the formatting elements are applied to the string and the now formatted translated text is inserted in the formatted document at the same position as the source text.

An additional difficulty appears when tokens formated together in the source string appear separated in the translation. Say our example was formatted like this;

(6) **this warm** cheese is boring

The formatting extraction produces `<0,9,bold>` but as shown in Figure 5.2, the formatted sub-string `this␣warm` is now aligned with a discontinuous sub-string in the translation: `ce…chaud`. In this case the formatting transfer needs to create two bold formatting elements: one for each part of the now discontinuous sub-string: `<0,2,bold>`, `<11,16,bold>`.

Chapter 6

A GF notebook kernel

6.1 Introduction

In this chapter, we present the implementation of a new feature in Grammatical Framework (hereafter GF) that allows one to use the GF shell in a Jupyter notebook. The primary motivations for integrating GF with Jupyter are on one hand to make GF more accessible by providing an alternative, more user-friendly interface to the GF shell, and on the other hand to improve the reproducibility of research using GF by providing an easy way to publish tutorials and work sessions in an easily reproducible way using the open notebook format offered by Jupyter.

In addition, it allows the GF community to benefit from the ecosystem of tools that have been built around Jupyter and its notebook format allowing for example to publish them on the web or to typeset a notebook as a $\LaTeX$ document.

6.2 A short overview of Jupyter

The Jupyter Notebook (previously known as iPython Notebook) is an web interface to many programming languages with some features that makes it well suited for reproducible data analysis tasks.

6.2.1 Presentation

Jupyter is an example of a very common class of applications known as read-eval-print loop (REPL), also informally known as shell. The basic principle is that the application *reads* the user input, *evaluates* it according to the rules of the languages and *prints* the result of the evaluation. And it does that in a loop, meaning that after printing the result, the application is ready to accept the next input from the user. What distinguishes those applications from *interpreters* is that an interpreter does not usually print the result of an expression unless explicitly told to do so, read all the input at once (typically from a file called a *script*) and exit after evaluating the user input.

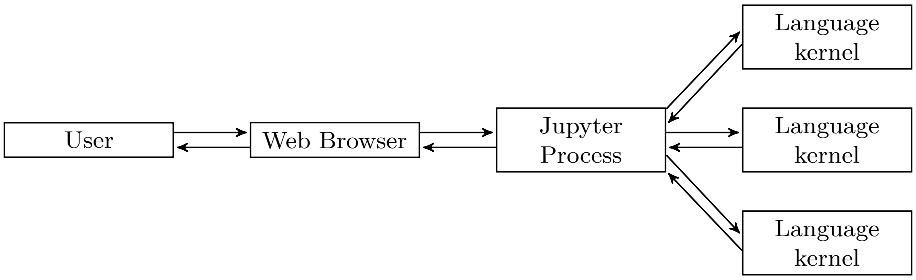


Figure 6.1: *This figure shows a simplified representation of the Jupyter architecture. The user interacts with the web browser which in turn communicates with Jupyter (using websockets). Finally, when code needs to be evaluated, Jupyter communicates with a language-specific* kernel*. By using multiple language kernels, Jupyter is able to maintain multiple sessions simultaneously and to have notebooks in different languages.*

Traditional REPLs are implemented as command-line applications, where the only mean of interaction with the user is through text. This is to say, they are meant to be invoked from a terminal emulator and text is the only medium available for both the user input and the program output. Jupyter differs from this by implementing the REPL as a web application. This allow the use of all media supported by modern web browser in the interaction with the user. For instance, a Jupyter notebook can include images, animations and even interactive visualizations.

An second particularity of Jupyter is the possibility to interleave commands with richly formated text. This is akin to comments in many programming language but just as the command output can include richly formated text and media, the user can include their own formated text, images etc. in the text.

Finally, not only Jupyter notebook sessions are saved and can easily be re-run and modified, thus preserving one's work to be continued later and facilitation reproducing results, but they are saved in an open document format, allowing for easy sharing of analysis details or even publishing the Jupyter notebook directly thanks for tools like nbviewer that render notebook on the web without the need to install Jupyter itself. Because the format used is open and documented, other application can make use of notebooks. One example is GitHub that directly renders them as HTML pages.

6.2.2 Architecture

The Jupyter architecture is separated in to three components as depicted in Figure 6.1

The interaction with the user and rendering of the notebook is done directly in a web browser. This makes it possible to use Jupyter from any device that has a web browser (which nowadays means almost any device which has a screen).

The web browser communicates with Jupyter using websockets. This is handled by the Jupyter process which also takes care of managing notebooks (saving, creating and deleting them) and allow access to local files. Jupyter is an open source application written in the python programming language.

Finally, the evaluation of the code itself is delegated to a *language kernel*. This is a very important architectural choice as it allow Jupyter to support many different programming languages. Adding a new language requires only the implementation of a new language kernel, thus skipping many tedious details such a network communication, generating the user interface and managing notebook files. At the time of this writing, over forty languages can be used in Jupyter notebooks.

The next section describe the implementation of a new kernel for Jupyter that allow the use of the GF REPL form Jupyter Notebooks.

6.3 iGF implementation

The GF kernel has been written as an extension to the GF compiler. In order to do that, it was most convenient to use the Haskell programming language as it is the language in which the current compiler is written.

We took advantage of the existing library IHaskell, created by Andrew Gibiansky and which offers facility for creating Jupyter kernels in Haskell. It was originally written to implement a Haskell kernel based on GHC. The name is a reference to iPython, which is the old name of Jupyter but now only references the python language kernel.

We reused as much code as possible from the existing GF shell, as this ensures that the functionalities differ as little as possible between the two interfaces. While all commands in the GF shell are returning string values, as the only available modality is text, we wanted to have the option to display different types of results in different way.

For instance, GF can generate different kind of diagrams (parse tree, word alignments, etc.) which are generated in the dot language. The dot language is a plain text graph description language that provides a simple way to describe graphs. There exist many tools that can produce or consume dot scripts, the most well known being probably the Graphviz suite of tools. In Jupyter, we didn't want to output the DOT code but instead, we wanted to automatically generate the graphical representation of the diagram and embed it in the notebook. So whenever a command or a chain of commands (GF shell commands can be chained together using pipes, just like in Bash) is returning dot code, we pass it to the `dot` executable (which is part of the Graphviz suite) to generate a PNG image with is then sent as part of the command output. See Figure 6.4 for an example.

To facilitate those distinctions, we replaced the `String` return type of the GF shell commands by the type shown bellow. This allows distinguishing between five different types of outputs:

NoOutput For command that have no real output and might only have side effects such as printing information of the screen. Example: the help command.

TextOutput For commands that return natural language text. Example: linearize

ForestOutput For commands that return trees. Example: parse

GraphOutput For commands that returns diagrams. Example: align_words or aw

TableOutput For commands that return a table.

```
data CommandOutput  = NoOutput
    | TextOutput [String]
    | ForestOutput [Expr]
    | GraphOutput [String]
    | TableOutput [(String, String)]
  deriving (Show, Eq)
```

6.4 Usage

6.4.1 Installation

This requires Jupyter to be installed. For more information about Jupyter installation, see the official documentation.

Install GF : If you have not yet installed GF, or if the installed version does not include the Jupyter Kernel, you need to install it from source. Download a copy of GF with the Jupyter kernel support and unpack it in a working directory (subsequently `$HOME/src/gf`). Then, you can install GF using the standard command `cabal install`. Although, if your cabal version supports it, we recommend that you use a sandbox for experimenting:

```
$ cd $HOME/src/gf
$ cabal sandbox init
$ cabal install
```

Make Jupyter aware of the GF kernel. The next step is to make sure that Jupyter knows how to use the GF kernel. To make this step easier, we have included a kernel specification in `src/tools/kernelspec/`. You can install it using the `jupyter kernelspec install` command in one of the following ways:

for all users

```
$ jupyter kernelspec install src/tools/kerneslpec/gf
```

for the current user

```
$ jupyter kernelspec install --user src/tools/kerneslpec/gf
```

in a virtualenv

```
$ jupyter kernelspec install --prefix=$VIRTUAL_ENV \
    src/tools/kerneslpec/gf
```

Starting Jupyter. You may now start the Jupyter notebook as usual by entering the command `jupyter notebook`. If, as recommended above, you installed GF in a Cabal sandbox and Jupyter cannot find the GF executable, you can use the following command to start Jupyter from the GF source directory (`$HOME/src/gf` in our example):

```
$ cabal exec -- jupyter notebook
```

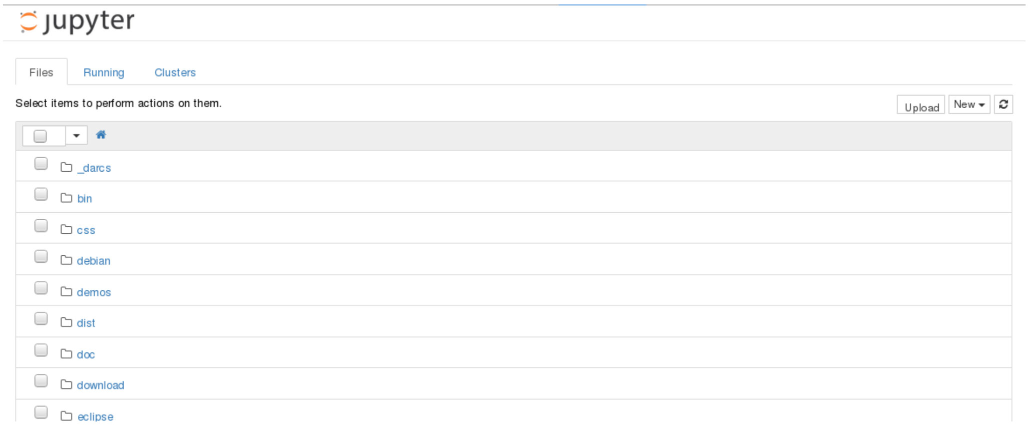


Figure 6.2: *The Jupyter notebook homepage, which should open automatically when starting Jupyter with the command* `jupyter notebook`*.*

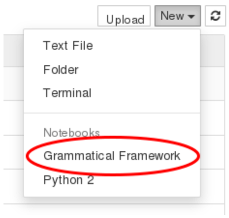


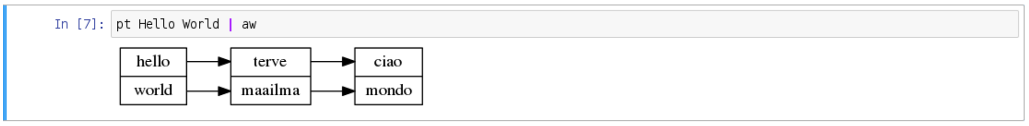
Figure 6.3: *To create a new GF notebook, click "New" and then select "Grammatical Framework".*

6.4.2 Quick start

If Jupyter and GF have been installed correctly, you should now have a browser window open that looks similar to the one in Figure 6.2. You can now create a new GF notebook by clicking on "New" in the top-right corner of the page and selecting "Grammatical Framework" (Figure 6.3).

You should now have a new browser tab with an empty notebook. Any command available in the regular GF shell can be used here (type `help` for the list of available commands). The main difference is that graphs printed as DOT code in the GF Shell are be automatically rendered as images, see Figure 6.4 for an example.

At any point you can save your work in Jupyter directly (in addition, Jupyter au-



```
In [7]: pt Hello World | aw
```

Figure 6.4: *Diagrams are automatically rendered as images in the GF notebook.*

tomatically saves regular “Checkpoints” in case something bad happens) or export it in various format, including HTML and PDF for publication. Annex B reproduces a notebook exported in LaTeX with examples of different commands.

6.5 Related work

As mention previously, many other language kernels have been implemented for Jupyter. We do not intend to give a list but we should mention the one written for the Haskell language. Not only there are existing connections between GF and Haskell—the GF language has many similarity with Haskell, and the GF compiler itself is written in Haskell. But more importantly, the implementation of the GF kernel benefited directly from the Haskell kernel implementation by re-using the IHaskell library.

Hallgren, Enache, and Ranta 2015 describe “A Cloud-Based Editor for Multilingual Grammars”, a grammar engineering tool based on GF and offered as software as a service. Although much larger in scope, it shares many feature with the work presented here, in particular the use of HTML to build a richer and more user-friendly interface. It includes more features than just an interface to the GF shell such as an editor for writing grammar in the browser, interactive tools for testing grammars (the *Minibar* and *Quiz* widgets) and the possibility of sharing grammars with other users. As a Jupyter kernel, our work on the other hand focuses more on writing documents (in natural language) with reproducible example. And, as Jupyter is meant to be installed and ran locally, it alleviates concerns of security (for the host) and privacy (for the user) associated with “cloud” services. (Both concerns are nicely summarized by this campaign slogan from the FSFE: *There is no cloud, just other people’s computers.*[1])

Finally Camilleri 2012 describes a plug-in to use the Eclipse IDE to write GF grammars. Here the focus is mainly on providing grammar writing support and although the plug-in gives a convenient way to start a GF REPL directly from the Eclipse interface, it provides the same interaction as is available in a terminal.

6.6 Conclusion and future work

This new kernel functionality allows one to use the GF shell inside of a Jupyter notebook kernel, providing a potentially more accessible user interface and making it easy to publish reproducible documents.

We plan to extend it by allowing the user to include grammars in the notebook directly, instead of relying on external files as it is the case right now. We also would like to explore the possibility of adding new commands to the GF notebook kernel that take advantage of the possible interaction offered by the web browser. Such command could for example display a “fridge-magnet” widget directly in the notebook, or show foldable parse-tree diagram by taking advantage of the ability of GF to generate JavaScript code from grammars.

[1]Free Software Foundation Europe is a charity that empowers users to control technology. For more information visit `https://fsfe.org`

Appendix A

Code for `Tokenizer.hs`

This is the complete code of the tokenizer presented in Chapter 1

```haskell
{-
GF Tokenizer.

In this module are implemented function that build a fst-based
tokenizer from a Concrete grammar.
-}

module PGF.Tokenizer
       ( mkTokenizer
       ) where

--import Data.List (intercalate)
--import Test.QuickCheck
import FST.TransducerInterface
import PGF.Morphology (fullFormLexicon, buildMorpho)
import PGF.Data (PGF, Language)



data LexSymbol = Tok String
    deriving (Show, Read)

type Lexicon = [LexSymbol]

-- | This is the construction function. Given a PGF and
-- a Language, it extract the lexicon for this language and
-- build a tokenization fst from it.
mkTokenizer :: PGF -> Language -> (String -> Maybe [String])
mkTokenizer pgf lang = mkTrans lexicon
```

```
  where lexicon = map (Tok . fst) lexicon'
        lexicon' = fullFormLexicon $ buildMorpho pgf lang

mkTrans :: Lexicon -> (String -> Maybe [String])
mkTrans = applyDown . lexiconTrans

lexiconTrans :: Lexicon -> Transducer Char
lexiconTrans lexicon =
    compile (words |> star ((spaces <|> glue) |> words))
      "abcdefghijklmnopqrstuvwxyz "
  where words = foldr (<|>) (empty) $ map tokToRR lexicon
        glue = eps <*> stringReg " &+ "

stringReg :: String -> Reg Char
stringReg str = foldr (\x y -> s x |> y) eps str

tokToRR:: LexSymbol -> RReg Char
tokToRR (Tok str) = foldr ((|>) . idR . s) (idR eps) str

spaces :: RReg Char
spaces = idR $ s ' '



-- TESTING

-- verry small test lexicon
-- testLexicon :: Lexicon
-- testLexicon
--    = [ Tok "car"
--      , Tok "elf"
--      ]

-- myTrans :: String -> Maybe [String]
-- myTrans = mkTrans testLexicon

-- data TestCase = TestCase String String
--      deriving (Show, Read)

-- instance Arbitrary TestCase where
--      arbitrary      = arbitraryTestCase
--      --coarbitrary c = variant (ord c `rem` 4)

-- arbitraryTestCase:: Gen TestCase
-- arbitraryTestCase = do
--   words <- listOf1 $ elements [t | Tok t <- testLexicon]
--   tokens <- intercalateSometime "+&+" words
--   return $ TestCase (linearize tokens) (intercalate " " tokens)
--   where intercalateSometime :: a -> [a] -> Gen [a]
--         intercalateSometime x (x1:x2:xs) = do
--           b <- arbitrary
--           let pre = case b of
--                 True -> x1:x:[]
--                 False -> x1:[]
--           suf <- intercalateSometime x (x2:xs)
--           return (pre++suf)
--         intercalateSometime _ xs = return xs

-- linearize :: [String] -> String
-- linearize = linearize' False
--   where linearize' :: Bool [String]  --^ boolean indicates if
--                                       -- the last token was a
--                                       -- real word and not +&+
--   where linearize' _ [] = ""
--         linearize' _  ("+&+":ss) = linearize' False ss
--         linearize' True (s:ss) = ' ':s ++ linearize' True ss
--         linearize' False (s:ss) = s ++ linearize' True ss

-- testTrans :: (String -> Maybe [String]) -> TestCase -> Bool
-- testTrans t (TestCase s1 s2) =
--   case t s1 of
--     Nothing -> False
--     Just l -> elem s2 l

-- main :: IO ()
-- main = do
--   putStrLn "\n=== Transducer ==="
--   print $ lexiconTrans lexicon
--   putStrLn "\n=== example output ==="
--   putStrLn $ "Input:  " ++ show "car elfcar elf"
--   putStrLn $ "Output: " ++ show (mkTrans lexicon "car elfcar elf")
--   putStrLn "\n=== QuickCheck tests ==="
--   quickCheck (testTrans myTrans)
--   putStrLn "\n=== Examples of test cases ==="
--   sample (arbitrary :: Gen TestCase)
```

Appendix B

iGF notebook demo

This is a short demonstration of the GF kernel for iPython. It works just like the regular GF shell but a bit more fancy. It allows you to use the gf shell in your browser, annotate you session with titles and *rich* **text** formatting, and share it on nbviewer.
You can even include images:

B.1 Examples

First, let's import some grammars from the GF tutorial:

```
In [1]: i HelloEng.gf HelloIta.gf HelloFin.gf

        Languages: HelloEng HelloFin HelloIta
```

We can do anything here we could do in the GF shell, like generating trees:

```
In [2]: gt

        Hello Friends
        Hello Mum
        Hello World
```

Linearizing them

```
In [3]: gt | l

        hello friends
        terve ystävät
        ciao amici
        hello mum
        terve äiti
        ciao mamma
        hello world
```

```
terve maailma
ciao mondo
```

Or parsing a string

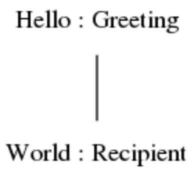
```
In [4]: p "hello world"
```

```
The parser failed at token "hello"

Hello World
```

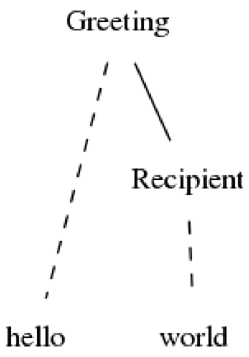
B.1.1 Graphs

But what's really cool is that we can also vizualize graph directly in the session:

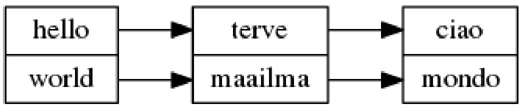
```
In [5]: pt Hello World | vt
```



```
In [6]: pt Hello World | vp
```



```
In [7]: pt Hello World | aw
```



Appendix C

References

References

Angelov, K. (2009). “Incremental parsing with parallel multiple context-free grammars”. In: *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics. DOI: `10.3115/1609067.1609074`.

— (2015). “Orthography Engineering in Grammatical Framework”. In: *ACL-IJCNLP 2015*, p. 33. DOI: `10.18653/v1/w15-3305`.

Angelov, K., B. Bringert, and A. Ranta (2010). “PGF: A Portable Run-time Format for Type-theoretical Grammars”. In: *J. of Logic, Lang. and Inf.* 19.2, pp. 201–228. ISSN: 0925-8531. DOI: `10.1007/s10849-009-9112-y`.

Angelov, K. and A. Ranta (2010). *Loosely Coupled Synchronous Parallel Multiple Context-Free Grammars for Machine Translation.*

Bacon, J. (2012). *The Art of Community.* 2nd ed. O’Reilly Media. URL: `http://www.artofcommunityonline.org/`.

ISO/IEC TR 19759:2015 (2015). *Software Engineering – Guide to the software engineering body of knowledge (SWEBOK).* ISO/IEC 19759.

Bringert, B. (2005). “Embedded Grammars”. MA thesis. Göteborg, Sweden: Chalmers University of Technology.

Bringert, B., K. Angelov, and T. Hallgren. *GF Developers Guide.* URL: `http://www.grammaticalframework.org/doc/gf-developers.html` (visited on 03/29/2016).

Camilleri, J. J. (2012). “An IDE for the Grammatical Framework”. In: *Free/Open-Source Rule-Based Machine Translation* 14, p. 1.

Détrez, G. and A. Ranta (2012). “Smart Paradigms and the Predictability and Complexity of Inflectional Morphology”. In: *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics.* Association for Computational Linguistics, pp. 645–653.

Fenton, N. and J. Bieman (2014). *Software metrics: a rigorous and practical approach.* CRC Press.

Fogel, K. (2014). *Producing Open Source Software: How to Run a Successful Free Software Project*. 2nd ed. O'Reilly Media. URL: `http://www.producingoss.com/`.

Forcada, M. L. et al. (2011). "Apertium: a free/open-source platform for rule-based machine translation". In: *Machine Translation* 25.2. Special Issue: Free/Open-Source Machine Translation, pp. 127–144. DOI: `10.1007/s10590-011-9090-0`.

Hallgren, T., R. Enache, and A. Ranta (2015). "A Cloud-Based Editor for Multilingual Grammars". In: *ACL-IJCNLP 2015*, p. 41. DOI: `10.18653/v1/w15-3306`.

He, Y. and M. Kayaalp (2006). "A Comparison of 13 Tokenizers on MEDLINE". In: *Bethesda, MD: The Lister Hill National Center for Biomedical Communications.*

Humble, J. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional. ISBN: 0321601912.

Papineni, K. et al. (2002). "BLEU: a Method for Automatic Evaluation of Machine Translation". In: *ACL '02: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 311–318. DOI: `10.3115/1073083.1073135`.

Ranta, A. (2009). "The GF Resource Grammar Library". In: *Linguistic Issues in Language Technology* 2.2.