



Term Paper

THE WARRIOR STABS THE GOAT

Creating a Dothraki Resource Grammar with Grammatical Framework

Maria Henkel

2008709 • maria.henkel@hhu.de

Benno Kuckuck

2340409 • benno.kuckuck@hhu.de

17 October 2016

Lecturers: Yulia Zinova, Rainer Osswald

Course: Introduction to “Grammatical Framework”, SS 2016

Contents

1	Introduction	1
2	The Project	2
2.1	About Resource Grammars and the Standard Library	2
2.2	Project Goals and Structure	3
2.3	Implementation of Grammatical Categories in Dothraki	6
2.3.1	Nouns and Noun Phrases	6
2.3.2	Verbs and Verb Phrases	8
2.3.3	(Relative) Clauses and Sentences	9
2.3.4	Adjectives	10
2.3.5	Other Categories	12
2.4	Implementation of Syntactical Functions in Dothraki	13
2.4.1	Morphology	13
2.4.2	Syntax	16
2.5	Vocabulary Extraction	18
2.5.1	Dictionaries and Lexicons in the RGL	18
2.5.2	The Dothraki dictionary	19
2.6	Example	21
3	Discussion	22
3.1	Limitations	22
3.2	Outlook	22
3.3	Conclusion	22



“Khal Drogo shillo chories she krazaaj. Chiori fin
vijazer adraes zoqwe Khales Drogo. Khal Drogo
vinde ma chories ma adraes she krazaaj. Khal
Drogo vachrara.”

“Khal Drogo met a woman on a mountain. The
woman, who protected a turtle, kissed Khal Drogo.
Khal Drogo stabbed the woman and the turtle on
the mountain. Khal Drogo stinks.”

GF Run-Time System, 2016

1 Introduction

Grammatical Framework (GF) is a functional programming language, created in 1998 [Ranta, 2011a], doubling as a categorial grammar formalism that can be used to implement (natural) language grammars. As GF is “working from a language-independent representation of meaning” [Wikipedia, 2016] it can be used to generate or parse text in different languages, e.g. for use in machine translation. Since GF uses a symbolic approach to processing and translating language, in most cases its translations are more precise than those of the more commonly used statistical tools [Brown & Frederking, 1995]. For example, on somewhat complicated input, the “Google Translate” service [Google, 2016] will in many cases produce ungrammatical sentences, while the GF translator [Grammatical Framework, 2016a] has no problems producing a correct translation:

Input	Output Google Translate	Output GF Translator
I heal these women.	Ich diese Frauen zu heilen.	Ich verheile diese Frauen.
The warrior will kiss that cheese behind the tree.	Der Krieger wird Kuss, der Käse hinter dem Baum.	Der Krieger wird jenen Käse hinter dem Baum küssen.
Diese heiße Frau heilt diese Königin im Meer.	This hot woman heals the queen of the sea.	This hot woman heals this queen in the sea.

Symbolic formalisms like GF – unlike Google Translate – depend on a large knowledge base of predefined grammar rules to achieve high-quality language processing: Its own standard library, the GF Resource Grammar Library, covers more than 38 different languages [Ranta, 2015] – although many of them are not yet complete – and is still growing. In 2010, it already consisted of more than 500,000 lines of code [Ranta, 2011a]. As Grammatical Framework is an open-source project [Ranta, 2009] and still under development, everyone is invited to use it or even help further develop it, for example by adding languages to the library, which is the aim of the project described in this paper. Our goal is to write a GF resource grammar for the Dothraki language and, eventually, be able to add it to the GF Resource Grammar Library. The work described in this paper is meant to lay a foundation for this goal (see Section 2.2). Dothraki is a constructed language, developed by the language creator and writer David J. Peterson [Peterson, 2015]. The Dothraki are a fictional race of nomadic horse warriors in the “A Song of Ice and Fire” book series by George R. R. Martin [Martin, 2016]. As the series was adapted for television by the television network HBO, the language bits found in Martin’s books had to be extended to a fully functional language [Peterson, 2016]: The Dothraki language. Today, it has a vocabulary of over 1,400 words [Tongues of Ice and Fire Wiki, 2016]. Examples of Dothraki language grammar will appear in Section 2.3, but much more information can be found in the book “Living Language Dothraki” [Peterson, 2014],

on Peterson’s blog [Peterson, 2016] and on the Tongues of Ice and Fire Wiki (2016) among many other sources.

This paper aims to serve as a short documentation and explanation of our project. It will cover the project aim and scope, as well as its structure and an exposition of our implementation of Dothraki grammar rules in GF (Section 2.3). We will also discuss problems and limitations and briefly describe future plans before we summarize and evaluate the project in its current state and our experience while working on it (Section 3).

2 The Project

As mentioned before, the eventual aim of this project is to create a resource grammar for the Dothraki language and add it to the GF Resource Grammar Library, so that other Dothraki fans and/or GF enthusiasts may use it. While there are currently a few Dothraki language learning apps and online translators available (see [Cognitus Apps, 2016], [Fun Translations, 2016]), these are very limited, primarily relying on word-for-word translation unaware of any grammar rules. An open-source Dothraki resource grammar for GF, which already provides a wide range of tools and options for integrating with other projects, would undoubtedly be a significant addition to the available range of Dothraki language resources.

2.1 About Resource Grammars and the Standard Library

GF, like other symbolic approaches to machine translation and language processing, is well-suited for providing high-quality output in more narrowly focused applications (as opposed to statistical approaches, whose strengths lie more in providing acceptable output in a very wide range of scenarios). Most applications however share a large part of the rules necessary to parse or construct correct sentences, namely those that are concerned with the morphology and syntax of the involved language(s). While these could be implemented ad-hoc for every application, it is much more economic to provide language-specific but application-independent rules as a reusable library. This is the goal of GF’s Resource Grammar Library.

The idea is to free application programmers from having to deal with “specialized linguistic expertise” (Ranta, 2009). By providing a common interface across languages, it becomes considerably easier (in some cases even trivial) for the application programmer to add support for additional languages. Considerable work has been and is being done by the GF developers and resource grammar programmers to make this possible:

The idea of using a grammar as a software library is, to our knowledge, new in GF. It has required a considerable effort in the design and implementation of the GF programming language: a type system, a module system, compilation techniques, and optimizations. [...] The effort made in this work is supplemented by a substantial effort in the library

itself. The code included in the library is more than twice in size compared to the implementation of GF. [Ranta, 2009]

This division of labour benefits not just the application programmers, however, but also the linguistics experts implementing the application-independent libraries: Firstly, because a resource grammar, once written, can “serve an unlimited number of applications” [Ranta, 2011a, p. 200], but also because “the existing library specification will help to identify the linguistic issues and avoid pitfalls” by providing a well thought-out, language-independent interface.

2.2 Project Goals and Structure

While in the long-term, our work is aimed towards a full implementation of a Dothraki resource grammar, this was not a realistic goal for this term paper, for two main reasons: First, Ranta estimates that an interval of three to six months of full-time work is needed for an experienced GF programmer to implement a new resource grammar [Ranta, 2011a, p. 222], which far exceeds the time frame for this term paper. And secondly, the Dothraki language itself is still a work-in-progress, with significant parts of the grammar either as yet undocumented, or possibly non-existent, which at present makes a full implementation of all the grammatical functions covered by GF’s RGL arguably impossible (for more on this point, see the discussion in Section 3).

In Ranta’s book [Ranta, 2011a, pp. 237–245] (see Figure 1), a mini resource grammar for Italian is implemented as an example.

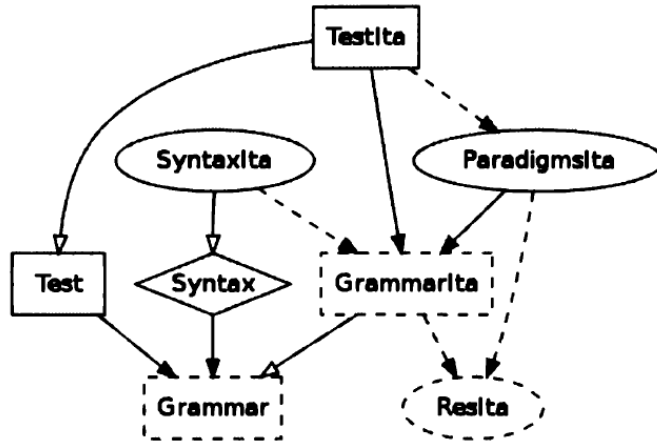


Figure 1: Mini Resource Grammar Modules and Dependencies [Ranta, 2011a, p. 209]

Our work lies somewhere between this example and a full resource grammar. Like in the Italian example, the abstract grammar specification our resource gram-

mar is built on, is only a subset of that of the full RGL. Unlike the Italian example, it adheres much more closely to the structure of the full RGL (see Figure 2), employing the same file structure and API, in hopes of easing a later expansion to a full resource grammar. It is also much more extensive than Ranta’s example, covering a much wider range of grammatical constructions.¹

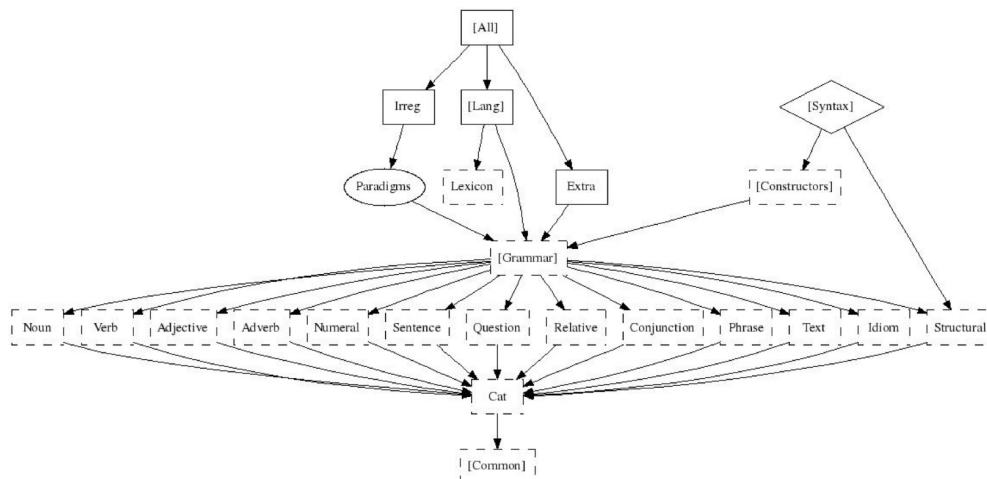


Figure 2: Module Structure of a Full Resource Grammar [Ranta, 2011b]

Just like the GF repositories [Détrez & Camilleri, 2016], our full project is hosted on GitHub [Henkel & Kuckuck, 2016] so that others may see, use and contribute to it.

The GF Resource Grammar Library consists essentially of three parts:

1. The *API* is the outward-facing part of the library. It contains those modules which will be used by application grammar programmers. It, in turn, is composed of two parts:
 - A (large) language-independent part (contained in the `api` folder of the RGL), primarily the module `Syntax`, which (roughly) consist of the module `Cat`, defining the abstract categories of GF’s language-independent representation of syntax and the module `Constructors`, which defines a wide array of functions for building up larger syntactical components (e.g. noun phrases, verb phrases, clauses) from constituent parts.
 - A (smaller) part specific to each language of the RGL, defined in a module called `ParadigmsXXX`, which contains functions for constructing instances of the basic lexical categories (e.g. nouns, verbs, adjectives).

¹As a quick comparison, the example from the book implements 12 grammatical categories, whereas our Dothraki implementation covers 27 of the 48 categories of the full RGL. In terms of API methods implemented, the comparison is even starker: Ranta’s example grammar implements 19 API methods, our grammar implements almost 200.

2. The API functions are defined in terms of more basic operations defined in the *abstract part* of the RGL,² which is contained in the folder `abstract`

These are the operations which must be implemented for each language.

3. The *concrete part* of the RGL, consists of concrete implementations of all abstract modules for each language that is part of the RGL. These are mostly contained in the folders carrying the name of the language.

Some of the abstract operations and categories are implemented similarly for almost all languages. Such implementations are contained in the common folder.

4. Additionally, but optionally, many languages also supply a large dictionary of words available in that language (allowing application grammar writers to use e.g. `rise_N` directly, instead of having to use `mk_V "rise" "rose" "risen"` and risking getting the irregular forms wrong).

As mentioned before, our project structure adheres closely to this template (see Figure 3):

- The `simpleapi` and `simpleabstract` folder contain the API part and abstract part respectively.

Their contents are selectively copied over from the corresponding files in the `api` and `abstract` folders of the full RGL.

This allows us to build a working resource grammar, without having to implement every single category and operation first. While our Dothraki resource grammar grew and continues to grow these are gradually expanded, until they converge towards the full RGL.

- The `simpleenglish` folder contains those parts of the English resource grammar, which are relevant for our trimmed-down API.
- The `common` folder is taken straight from the full RGL.
- Finally, the `simpladothraki` folder contains our original contribution, the (somewhat trimmed-down) resource grammar for Dothraki.

There are three more folders, which are not part of the resource grammar:

- The folder `example` contains an application grammar that uses our Dothraki resource grammar and the English resource grammar.
- The folder `documentation` contains this documentation.

²For example the API contains 22 overloads for the function `mkCl`, used to construct clauses, but almost all of these are based on the single method `predVP` from module `Sentence`.

- The folder `vocabulary` contains a Python script used for scraping the wiki source code of the publically available vocabulary list [Tongues of Ice and Fire Wiki, 2016] and automatically generating a GF-compatible Dothraki dictionary (see Section 2.5).
- `simpleapi`
- `simpleabstract`
- `simpledothraki`
- `simpleenglish`
- `common`
- `example`
- `documentation`
- `vocabulary`

Figure 3: The project’s folder structure

2.3 Implementation of Grammatical Categories in Dothraki

In this section we will go into some of the design decisions involved in the implementation of the Dothraki resource grammar. In order to get a high-level overview, we will first take a look at the linearizations of some of the categories defined in the API part of the abstract resource grammar library, showing how Dothraki grammar can be fit into the GF scheme.

2.3.1 Nouns and Noun Phrases

The relevant categories here are N (nouns), CN (common nouns), NP (noun phrases), Pron (pronouns) and PN (proper names), which are realized as follows:³

```
N, CN = {s : Number => Case => Str ; a : Animacy } ;
NP, Pron = {s : Case => Str ; agr : Agr } ;
PN = {s : Case => Str } ;
```

These implementations follow in a fairly straightforward fashion from the basic features of nouns in the Dothraki language: In Dothraki, every noun falls into one of two classes, *animate* and *inanimate* (which play a very similar role to grammatical

³Here, and in the following examples, the code snippets do not necessarily appear exactly like this in the actual source code, since usually some parts are defined in the `ResDot` module and some in `CatSimpleDot`, for reasons of reusability and implementation hiding. However, in all cases, the actual implementations are equivalent to the listings supplied here.

gender in other languages). The language uses five cases: nominative, accusative, genitive, ablative and allative. Hence, we define:

```
Animacy = Anim | Inanim ;
Case = Nom | Gen | Acc | All | Abl ;
```

Noun phrases, as well as pronouns, have a fixed number and person, which is stored in the `agr` field of type `Agr`, simply defined as follows:

```
Agr = Ag Person Number ;
```

There does not currently seem to be a need to store animacy in the agreement record. This is unlike English, where the NPs gender can affect the verb phrase (“The boy loves himself.” but “The girl loves herself.”) and thus needs to be stored in the NP. Finally, proper names are (so far as we can tell) always animate and third person singular, so none of that information needs to be stored in PN.

In order to turn a common noun into a noun phrase, we usually need a determiner. The relevant categories here are `Num` (numerals), `Quant` (quantifiers), `Predet` (predeterminers) and `Det` (determiners, roughly composed of optional predeterminers, a quantifier and a numeral). Numerals in the proper sense, as well as predeterminers, are not yet implemented in our resource grammar (there is a lack of good information on determiner structure in Dothraki), but since the `Num` category is used in the GF Resource Grammar Library not just for representing numerals but also to determine the number of the determiner, we have a dummy implementation in our resource grammar:

```
Num = {s: Str ; n : Number} ;
Quant = {s : QuForm => Case => Str } ;
Det = {s : Animacy => Case => Str ; n : Number ; s2 : Str } ;
```

The `s` field of `Num` is always empty in our implementation, this is necessary for technical reasons related to the implementation of GFs parser [Google Groups, 2016].

Quantifiers in Dothraki (such as “jin” ~ “this” and “rek” ~ “that”) are inflected for animacy, number and case, but the singular and plural inanimate forms are always identical, so `QuForm` is defined as follows:

```
QuForm = QAnim Number | QInanim ;
```

Unlike a `Quant`, a `Det` has a fixed number. An interesting feature of determiners in Dothraki is that they can appear either before the noun (“jinak adra” = “this turtle”) or after (such as determiners built from possessive pronouns: “adra anni” = “my turtle”). Conceivably,⁴ a determiner could even be split into two parts (“my two turtles” [?] “akat adrasi anni”). In anticipation of this, `Det` has two `Str` components, `s` and `s2`. Since possessive pronouns are the only examples of postposed determiners we know so far, and these happen not to inflect with animacy or case, the `s2` part is simply a `Str`, whereas the `s` part is a table of inflected forms.

⁴we cannot currently say for certain, due to lack of examples and documentation on Dothraki determiner structure

2.3.2 Verbs and Verb Phrases

The relevant categories here are V (intransitive verbs), V2 (transitive verbs with an NP complement) and VP (verb phrases), whose lincats are as follows:

```

param VFormPN = Pers1 Number | Pers2 | Pers3 Number ;
param VForm =
  APast Polarity Number
  | APresent Polarity VFormPN
  | AFuture Polarity VFormPN
  | ImpFormal Polarity
  | ImpInformal Polarity ;

oper Verb : Type = {
  s : VForm => Str ;
  inf : Str ;
  part : Str
} ;

V = Verb ;
V2 = Verb ** {objCase : Case} ;
VPSlash = Verb ** {objCase : Case ; subjpost : Str} ;
VP = Verb ** {compl : Str; subjpost : Str} ;

```

The basic structure Verb contains the infinitive and participle as well as all conjugated forms of a verb. The somewhat convoluted setup of the VForm parameter type is owed to the peculiarities of conjugation in Dothraki: First off, somewhat unusually,⁵ in all tenses and moods, verb forms vary with the polarity of the sentence (“Me dothrae” = “He rides”, “Me vos dothrao” = “He does not ride”). In the present and future tenses, verbs are inflected for person and number, but the second person singular and plural forms are always identical (hence the definition of VFormPN). In the past tense, verbs are inflected for number only, not for person (but unlike in the present and future tenses, second person singular and plural forms do differ). There are also formal and informal imperative forms.

Transitive verbs are basically the same as intransitive verbs, but they also assign a case to their NP complement (mostly accusative, but other cases do appear), which is stored in the objCase field.

A verb phrase essentially consists of a verb with a complement. There is also a field subjpost which again encodes a peculiar feature of Dothraki grammar: A number of constructions, which, in other languages, are rendered by verbal auxiliaries (can, must) or verbs with VP complements (try to), are realized in Dothraki via non-inflecting particles (e.g. “laz” ~ “can” and “eth” ~ “must”), which do not otherwise affect the conjugation of the main verb (“Me dothrak” = “He rides”, “Me laz dothrak” = “He can ride”, “Me kis dothrak” = “He tries to ride”)⁶. These have to be

⁵from the Indo-European viewpoint, that is, not for natural languages in general

⁶This could be compared to the way negation is expressed in many Indo-European languages, namely by a non-inflecting particle (“not”, “nicht”, “ne ... pas”, etc.), which is adposed to the verb in some manner, but does not otherwise affect inflection.

regarded as part of the verb phrase but, syntactically, are not preposed to the verb phrase but postposed to the subject of the clause (this makes a difference, e.g., in relative clauses, which have VSO word order instead of the standard SVO word order). The same is true of passive constructions, which are indicated by “nem” postposed to the subject (“Lajak dothrae” = “The warrior rides”, “Hrazef nem dothrae” = “The horse is ridden”; note that the verb still agrees in number and person with the syntactical subject of the clause).

VPSlash represents a verb phrase missing a complement and structurally kind of sits between V2 and VP. It is generated from a V2 (via `SlashV2 : V2 -> SlashVP`) and can then either be turned into a VP by adding an object (via `CompSlash : VPSlash -> NP -> VP`) or into a C1Slash (clause missing an object) by adding a subject (via `SlashVP : VPSlash -> NP -> C1Slash`), to be used, e.g., in a relative sentence (“the goat which the warrior stabbed”). As such it does not have a complement, like VP, and retains the `objCase` from V2, but it does have the `subjpost` field from VP (so we can form, e.g., “the goat which the warrior can stab” = “dorvi fin vindee lajak laz”).

In order to implement verbal auxiliaries such as “can” and “must”, which are represented as VVs in GE, we also have a dummy lincat for VV:

```
VV = {s : Str} ;
```

Since “can”, “must” etc. are simply uninflected particles in Dothraki, this is enough to represent those. This should of course not be considered a final design, since it is unlikely that all verbs with verb phrase complement can be realized like this in Dothraki. However, there is currently very little documentation available on how other VV verbs are realized, so this dummy implementation will have to do for now.

2.3.3 (Relative) Clauses and Sentences

There are no big surprises in the lincats of C1 and QC1 (clauses and question clauses, respectively) or S and QS (sentences and question sentences):

```
S, QS = {s : Str} ;
C1, QC1 = {s : Tense => Anteriority => Polarity => Str} ;
```

The lincats of these categories are fairly canonical in many languages and in Dothraki they are basically the same as in English or German. The main differences between these languages come from whether the linearization also depends on some kind of `Order` parameter, for example distinguishing indirect from direct questions or subordinate from main clauses. It might be necessary to add something like that to our lincat later, if more information about, e.g., indirect questions in Dothraki becomes available, but for now there is no indication that we need such an extra parameter.

Relative clauses are more interesting. The relevant categories here are RP (relative pronoun), C1Slash (a clause missing an object), RC1 (relative clauses) and RS (relative sentence):

```

RP = {s : QuForm => Case => Str } ;
RC1 = {s : Tense => Anteriority => Polarity => Animacy => Number
      => Str } ;
C1Slash = {s : Tense => Anteriority => Polarity => Str;
           subj: Str ; objCase : Case} ;
RS = {s : Animacy => Number => Str } ;

```

Relative pronouns in Dothraki inflect according to the animacy and number of the noun phrase they attach to, and the case that the relativized noun phrase would have been assigned in the embedded relative sentence (similar to German). Hence, the linearization of RS depends on the animacy and number of the noun phrase it is eventually attached to. RC1 is to RS what C1 is to S, i.e., it is a relative sentence, whose tense, anteriority and polarity have not yet been determined. C1Slash has to remember the case of the missing object, so it can assign it to the relative pronoun, if used in a relative sentence (via RelSlash : RP → C1Slash → RC1).

There is one more important difference between C1 and C1Slash. When a clause is constructed from a verb phrase and subject via PredVP : NP → VP → C1, the noun phrase is just prepended to the verb phrase. We cannot do this when constructing a C1Slash via SlashVP : NP → VPSlash → C1Slash, because of a quirk in Dothraki word order: Whereas regular main clauses and questions follow SVO word order, relative clauses have VSO word order. So depending on whether a C1Slash is eventually turned into a relative sentence (via RelSlash : RP → C1Slash → RC1) or a question (via QuestSlash : IP → C1Slash → QC1), we have to choose the word order accordingly (“dorvi fin vindee lajak” = “the goat which the warrior stabs” vs “Fin lajak vindee?” = “What does the warrior stab?”). Therefore, C1Slash contains the (linearized) subject in a field, subj, instead of embedding it in the s.

2.3.4 Adjectives

The relevant categories here are A (adjectives), AP (adjectival phrases) and Comp (“complement of a copula”, though see below):

```

A = {s : Degree => Number => ACase => Str ;
     pred : VForm => Str } ;
AP = {s : Number => ACase => Str ; pred : VForm => Str } ;
Comp = {s : VForm => Str} ;

```

There are two basic uses for adjectival phrases: attributive (as in “the strong warrior”) and predicative (as in “the warrior is strong”). Attributive use works similarly in Dothraki to, e.g., English and German: The adjectival phrase inflects for number and case in agreement with the noun it modifies. In Dothraki the forms for all the cases other than nominative are always identical, however, so we define

```
ACase = ANom | AOther ;
```

instead of using Case and having to duplicate these forms in the table.

In English or German, predicative use of an adjective employs a copula (“to be” or “sein”), which is inflected for tense, whereas the adjective is invariant (the same

happens in sentences like “the woman is a warrior” or “the man is on the mountain”, where the copula “to be” connects two noun phrases or a noun phrase and an adverbial phrase). Dothraki, however, is an entirely copulaless language. When an adjective is used predicatively, the adjective itself is essentially turned into a verb and inflected for tense and polarity:

“the strong warrior” = “lajak haj”
 “The warrior is strong” = “lajak haja”
 “the warrior will be strong” = “lajak vahaja”
 “the warrior was not strong” = “lajak vos ahajo”

Similarly, in a sentence like “the woman is a warrior”, the NP itself inflects for tense, with the ablative and allative form indicating past and future tense, respectively:

“The woman is a warrior” = “Chiori lajak”
 “The woman was a warrior” = “Chiori lajakoon”
 “The woman will be a warrior” = “Chiori lajakaan”

As a result, the category Comp (i.e., “strong” in “the warrior is strong” and “a warrior” in “the woman is a warrior”) essentially behaves like an intransitive verb in Dothraki. A and AP have to contain all conjugated forms of the verb as it might be used in a predicative context.

Indeed our lincats for A, AP and Comp, while very much unlike the lincats in the English and German resource grammars, are quite similar to the lincats in the resource grammar for Japanese, another language in which the adjective itself can be inflected for tense and polarity in predicative use via a complex set of suffixes. As in our resource grammar, this leads to AP and Comp being structurally very similar to VP:

```
VP = {
  verb : Speaker => Animateness => Style => TTense => Polarity
    => Str ;
  a_stem, i_stem : Speaker => Animateness => Style => Str ;
  te, ba : Speaker => Animateness => Style => Polarity => Str ;
  prep : Str ; obj : Style => Str ;
  prepositive : Style => Str ; needSubject : Bool} ;
Comp = {
  verb : Animateness => Style => TTense => Polarity => Str ;
  a_stem, i_stem : Animateness => Style => Str ;
  te, ba : Animateness => Style => Polarity => Str ;
  obj : Style => Str ; prepositive : Style => Str ;
  needSubject : Bool} ;
AP = {
  pred : Style => TTense => Polarity => Str ;
  attr, adv, dropNaEnging, prepositive : Style => Str ;
  te, ba : Style => Polarity => Str ;
  needSubject : Bool} ;
```

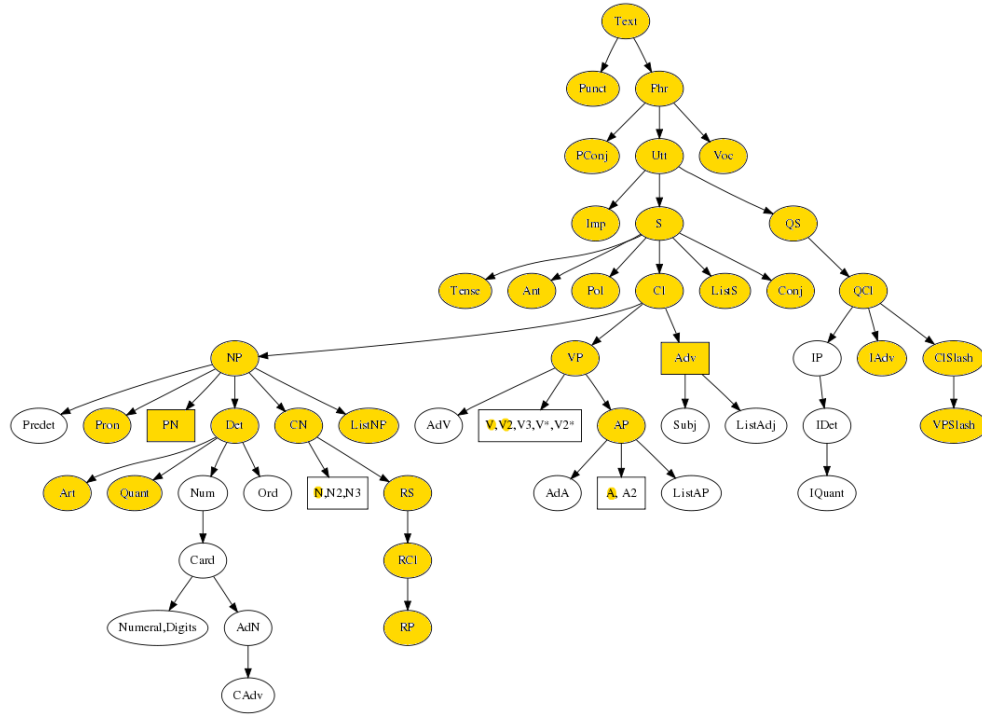


Figure 4: GF Resource Grammar Library Categories [Ranta, 2011b]

2.3.5 Other Categories

Adverbial phrases, like in most other languages, are simply uninflected token strings, as implemented in CommonX:

```
Adv = {s : Str} ;
Prep = {s : Str ; c : Case} ;
```

Adverbial phrases can be formed from noun phrases using prepositions, which assign case in Dothraki, or sometimes are expressed *only* by case. For example, `to_Prep` from module `Structural` is implemented in Dothraki simply as `{s = [] ; c = All}`: What is expressed in other languages via the preposition “to”, is understood in Dothraki from the modified noun phrase appearing in allative case.

Conjunctions in Dothraki have different forms depending on whether they are used to conjoin noun phrases/sentences, or as phrasal conjunctions to start a sentence:

```
Conj = {s : Str; p : Str; n : Number} ;
```

E.g., the conjunction “and”, when used to connect noun phrases, is “ma” (as in “ma lajak ma khaleesi” = “the warrior and the queen”), but translates as “majin”, when used as a phrasal conjunction (“Majin lajak zoqwe khaleesies.” = “And the warrior kissed the queen.”).

Figure 4 shows the main categories of the GF Resource Grammar Library and the current status of our implementation. The yellow categories have been fully or at least partially implemented in our resource grammar.

2.4 Implementation of Syntactical Functions in Dothraki

2.4.1 Morphology

Dothraki, generally speaking, has a much richer morphology than English, making the constructors for lexical categories supplied in `ParadigmsSimpleDot` fairly complex. As an example, let us describe the construction of verbs (nouns and adjectives present similar challenges). Here is the constructor `mk2V : Str -> Str -> V` which builds a `V` from the infinitive form and the past singular (the two forms which would often be given in dictionaries):

```
mk2V : Str -> Str -> Verb = \zalat,zal -> let {stem = stemV zalat
  zal} in {
  inf = zalat ;
  s = case stem of {
    fati@(fat + ("a"|"e"|"i"|"o")) => table {
      APast Pos Sg => zal ;
      APast Pos Pl => fati + "sh" ;
      APast Neg Sg => fat + "o" ;
      APast Neg Pl => fat + "osh" ;

      APresent pol pn => presForm fati pol pn ;

      AFuture Pos pn => presForm (pre {"a"|"e"|"i"|"o" => "v" ; _
=> "a"} + fati) Pos pn ;
      AFuture Neg pn => presForm (pre {"a"|"e"|"i"|"o" => "v" ; _
=> "o"} + fati) Neg pn ;

      ImpFormal Pos => stem ;
      ImpFormal Neg => fat + "o" ;

      ImpInformal Pos => fati + "s" ;
      ImpInformal Neg => fat + "os"
    } ;
  em => table {
    APast Pos Sg => zal ;
    APast Pos Pl => em + "ish" ;
    APast Neg Sg => em + "o" ;
    APast Neg Pl => em + "osh" ;

    APresent pol pn => presForm em pol pn ;

    AFuture Pos pn => presForm (pre {"a"|"e"|"i"|"o" => "v" ; _
=> "a"} + em) Pos pn ;
```

```
    AFuture Neg pn => presForm (pre {"a|"e|"i|"o" => "v" ; _
=> "o"} + em) Neg pn ;

    ImpFormal Pos => em + "i" ;
    ImpFormal Neg => em + "o" ;

    ImpInformal Pos => em + "as" ;
    ImpInformal Neg => em + "os"
  }
} ;
part = case stem of {
  fat + ("a|"e|"i|"o") => stem + "y" ;
  _ => stem + "ay"
}
} ;
```

Here `presForm : Str -> Polarity -> VFormPN -> Str` is another function, that generates the present tense forms of a verb. This is reused for the future tense forms, which have the same endings, but are distinguished by a prefix.

```
presForm : Str -> Polarity -> VFormPN -> Str = \stem,pol,pn ->
  case stem of {
    fati@(fat + ("a|"e|"i|"o")) => case <pol,pn> of {
      <Pos, Pers1 Sg> => fati + "k" ;
      <Pos, Pers1 Pl> => fati + "ki" ;
      <Pos, Pers2> => fati + "e" ;
      <Pos, Pers3 Sg> => fati + "e" ;
      <Pos, Pers3 Pl> => fati + "e" ;

      <Neg, Pers1 Sg> => fat + "ok" ;
      <Neg, Pers1 Pl> => fat + "oki" ;
      <Neg, Pers2> => fati + "o" ;
      <Neg, Pers3 Sg> => fati + "o" ;
      <Neg, Pers3 Pl> => fati + "o"
    } ;
    em => case <pol,pn> of {
      <Pos, Pers1 Sg> => em + "ak" ;
      <Pos, Pers1 Pl> => em + "aki" ;
      <Pos, Pers2> => em + "i" ;
      <Pos, Pers3 Sg> => em + "a" ;
      <Pos, Pers3 Pl> => em + "i" ;

      <Neg, Pers1 Sg> => em + "ok" ;
      <Neg, Pers1 Pl> => em + "oki" ;
      <Neg, Pers2> => em + "i" ;
      <Neg, Pers3 Sg> => em + "o" ;
      <Neg, Pers3 Pl> => em + "i"
    }
  }
} ;
```

As can be seen here, suffixes often differ according to whether the stem ends in a vowel or a consonant, which is handled by GF's pattern matching mechanisms. Future tense forms are indicated by a prefix ("v" if the stem starts with a vowel, "a" if the stem starts with a consonant), which we implement employing GF's special "prefix-dependent choice" type.

In case the past singular form of the verb is not explicitly supplied, it needs to be guessed, which can be complicated:

```
mk1V : Str -> Verb = \w -> case w of {
  ezo + "lat" => mk2V w ezo ;
  riss + "at" => mk2V w (addepenthesis riss)
} ;
```

In general, the past singular form is just the stem of the verb, which mostly is derived by removing the "-lat" or "-at" infinitive ending.⁷ However, this is complicated by the rules of epenthesis: The stem of "rissat" is "riss-", but the past singular is "risse", because a Dothraki word cannot end in "-ss". The rules for which consonant clusters may appear at the end of a word are complicated and for the most part captured in the function `addepenthesis : Str -> Str`, which adds an epenthesis "-e" to a string, if it ends in a disallowed consonant cluster:

```
addepenthesis : Str -> Str = \w -> case w of {
  _ + ("a"|"e"|"i"|"o") => w ;
  _ + ("w"|"g"|"q") => w + "e" ;
  x + ("y"|"r"|"l") => case x of {
    _ + ("a"|"e"|"i"|"o") => w ;
    _ => w + "e"
  } ;
  x + ("m"|"n") => case x of {
    _ + ("a"|"e"|"i"|"o") => w ;
    y + ("w"|"y"|"r"|"l") => case y of {
      _ + ("a"|"e"|"i"|"o") => w ;
      _ => w + "e"
    } ;
    _ => w + "e"
  } ;
  x + ("th"|"s"|"sh"|"z"|"zh"|"kh"|"f"|"v") => case x of {
    _ + ("a"|"e"|"i"|"o") => w ;
    y + ("w"|"y"|"r"|"l"|"m"|"n") => case y of {
      _ + ("a"|"e"|"i"|"o") => w ;
      _ => w + "e"
    } ;
    _ => w + "e"
  } ;
  x + ("j"|"ch"|"t"|"d"|"k") => case x of {
    _ + ("a"|"e"|"i"|"o") => w ;
```

⁷Some verbs, such as "zalat" end in "-lat" but the "l" is part of the stem, not the infinitive ending. This cannot be guessed and has to be indicated in a dictionary.

```

y + "w"|"y"|"r"|"l"|"m"|"n"|"th"|"s"|"sh"|"z"|"zh"|"kh"|"f"|"
v") => case y of {
  _ + ("a"|"e"|"i"|"o") => w ;
  _ => w + "e"
} ;
_ => w + "e"
}
} ;

```

This heuristic will give the right result in most (but not all) cases. Like in other resource grammars, there is always the option of supplying an irregular form by hand, if the heuristic fails.

2.4.2 Syntax

Many of the considerations about Dothraki syntax, that have gone into the design of our resource grammar, have already been mentioned in 2.3. As one example of how the design of our lincats is used to implement syntactical functions, let us go through the various ways of constructing clauses:

```

PredVP np vp = {s = \t,a,p =>
  np.s!Nom
  ++ vp.subjpost
  ++ verbStr vp t a p np.agr
  ++ vp.compl ;
} ;

```

The function `PredVP : NP -> VP -> C1` from module `Sentence` is the basic way of constructing a clause from a subject and a verb phrase (23 out of the 31 overloads of `mkC1` from the resource grammar API are implemented in terms of `PredVP`). In our implementation this function concatenates four elements:

1. The nominative form of the subject noun phrase,
2. any particles indicating the mood of the verb phrase (such as “nem” indicating passive voice, or “laz”/“eth” corresponding roughly to “can”/“must” in English)
3. the verb inflected for tense, anteriority, polarity and person/number of the noun phrase,
4. and the complement of the verb phrase.

Conjugation of the verb is implemented in the function `verbStr` defined in the module `ResDot`:

```

verbStr : Verb -> Tense -> Anteriority -> Polarity -> Agr -> Str
  = \v,t,a,p,agr -> let vf = (tapaToVForm t a p agr) in
  case <t,a> of {
    <Present,Anter> => "ray" ;    -- the perfect tense marker

```

```

_ => []
} ++
case p of {
  Pos => [] ;
  Neg => "vos"
} ++
v.s!vf ;

```

The argument `vp` in `PredVP` above is of type `VP` which is defined as `VP = Verb ** {compl : Str; subjpost : Str}`. GF has a mostly structural (as opposed to nominative) type system and thus `VP` is automatically subtype of `Verb`, and variables of `VP` type can be used where a `Verb` is expected (as we do above).

The function `verbStr` is reused in other functions concerned with the construction of clauses.

```

SlashVP np vpsl = {
  s = \\t,a,p => verbStr vpsl t a p np.agr ;
  subj = np.s!Nom ++ vpsl.subjpost;
  objCase = vpsl.objCase
} ;

```

The function `SlashVP : NP -> VPSlash -> C1Slash` is used in several overrides of the API functions `mkC1Slash` as well as in some overrides of `mkQC1` and `mkRC1`. Since `VPSlash` is also an extension of `Verb`, we can reuse `verbStr` to conjugate the verb. The subject phrase is not prepended to the verb (for reasons explained in Section 2.3.2), but instead kept in the field `subj`. Unlike `VP`, a `VPSlash` has no complement, but the case of the missing object has to be remembered.

```

RelVP rp vp = {
  s = \\t,a,p,anim,n =>
    rp.s!(anToQuForm anim n)!Nom
    ++ vp.subjpost
    ++ verbStr vp t a p (Ag P3 n)
    ++ vp.compl ;
} ;

```

The function `RelVP : RP -> VP -> RC1` is the basic way of creating relative clauses in which the subject of the sentence is being relativized (22 of the 25 overrides of `mkRC1` are based on `RelVP`). This implementation is quite similar to `PredVP`, but with the relative pronoun taking the place of the subject, and being inflected for animacy and number.

The verb agrees in number with the relative pronoun and is always in the third person.

Finally, clauses can be formed using `ExistNP : NP -> C1`.

```

ExistNP np = {s = \\t,a,p =>
  np.s!Gen
  ++ (verbStr vekhat_V t a p np.agr)
} ;

```

This is for creating sentences such as “Lajaki vekha” = “There is a warrior”. Unlike any other verb in Dothraki, the verb “vekhat” assigns genitive case to the subject of the sentence. Apart from that, we again use `verbStr` to conjugate the verb “vekhat” (`vekhat_V` is defined in module `ExtraDot`).

2.5 Vocabulary Extraction

2.5.1 Dictionaries and Lexicons in the RGL

As mentioned earlier, many of the resource grammars in GF’s library also come with a dictionary often called `DictXXX`.

For example, in `DictGer` from the German resource grammar, we find entries such as:

```
entziehen_V = irregV "entziehen" "entzieht" "entzog" "entzöge"
               "entzogen" ;
kreis_N = mkN "Kreis" "Kreise" masculine ;
```

An application grammar writer can then use these definitions and does not have to worry about the gender of nouns, plural forms or irregular conjugations.

In fact, the dictionaries in the RGL come in the form of an abstract grammar with a concrete implementation. The abstract grammar, often called `DictXXXAbs`, contains entries like

```
entziehen_V : V ;
kreis_N : N ;
```

These dictionaries, of course, cannot be used directly for translation. However, there are two ways in which the RGL supports direct translation between two languages. First, there is the abstract grammar Lexicon in the abstract folder, which covers about 350 more or less commonly used words:

```
...
drink_V2 : V2 ;
dry_A : A ;
dull_A : A ;
dust_N : N ;
...
```

This abstract grammar is then implemented in many of the languages of the RGL. From `LexiconGer`:⁸

```
drink_V2 = dirV2 Irreg.trinken_V ;
dry_A = regA "trocken" ;
dull_A = regA "stumpf" ;
dust_N = reg2N "Staub" "Stäube" masculine ;
```

The Lexicon module is part of the abstract grammar `Lang` and its concrete implementations in various languages. Of course, the vocabulary is far too small to

⁸The choice of names here is unfortunate: In the common linguistic sense of the words, `DictGer` is a lexicon, while `LexiconGer` is a dictionary...

be useful for applications, but it is useful for experimentation and for testing the resource grammars.

The RGL also has a folder translator which mirrors this structure on a much larger scale: The abstract grammar Dictionary contains 65,000 English words,⁹ and the concrete implementations DictionaryXXX contain their translations into various languages. This is used, e.g., in the GF online translator [Grammatical Framework, 2016a].

2.5.2 The Dothraki dictionary

The publicly available vocabulary list for Dothraki [Tongues of Ice and Fire Wiki, 2016] currently contains about 1,500 words.¹⁰ In order to make these accessible for GF application grammars and for the purpose of translation from and to English, we supply, as part of our project, a Python script that translates the wiki source of this list into a GF-compatible format.

The script creates five files in total, adhering closely to the structure established in the full RGL, as explained above:

- the abstract grammar DictDotAbs.gf and its concrete implementation DictDot.gf in the folder simpledothraki,
- the abstract grammar BigLexicon.gf in simpleabstract and its implementations BigLexiconEng.gf and BigLexiconDot.gf in simpleenglish and simpledothraki, respectively.

Figure 5 shows an example of the GF code the script generates. A word is added to BigLexicon whenever a corresponding word is found in DictEng, the massive English dictionary that is part of the RGL.

The wiki vocabulary list has 1,380 entries at the time of this writing. From these, our script creates 1,410 definitions in DictDot.¹¹

There are a total of 1,916 English translations in the vocabulary list. The script is able to extract 1,059 entries for BigLexicon from this. Many translations cannot be directly added to BigLexicon for various reasons:

- Some English words appear as translations for more than one Dothraki word. The script picks the first one it finds.¹²
- Many Dothraki words do not correspond to any one English word. For example the list contains entries for “defensive sword strike”, “sound of fire going out”, “roast quail” or “time period of approximately two minutes”. While for

⁹as well as 450 Finnish proper names...

¹⁰This is still not enough to implement the abstract Lexicon module from GF’s resource grammar library: Many common words do not yet have a Dothraki equivalent.

¹¹the somewhat larger number comes from the fact that many words can be used in two different ways, for example both as an intransitive and as a transitive verb, giving more than one entry in DictDot

¹²Obviously, that might not result in the best or most appropriate translation being picked, but that issue can be addressed by later manual editing of the generated list.

```
*'''aranikh''' [a?a'nix]
:''ni.'' subject, topic, theme

*'''gerat''' [ge'?at]
:''vtr.&rarr;abl.'' to lack
:{{Vsup|ger}}
```

```
in simpledothraki/DictDotAbs.gf:
aranikh_N : N ;
gerat_V2 : V2 ;

in simpledothraki/DictDot.gf:
aranikh_N = mkN "aranikh" inanimate ;
gerat_V2 = mkV2 "gerat" "ger"
          ablative ;

in simpleabstract/BigLexicon.gf:
subject_N : N ;
topic_N : N ;
theme_N : N ;
lack_V2 : V2 ;

in simpledothraki/BigLexiconDot.gf:
subject_N = DictSimpleEng.subject_N;
topic_N = DictSimpleEng.topic_N;
theme_N = DictSimpleEng.theme_N;
lack_V2 = DictDot.gerat_V2;

in simpleenglish/BigLexiconEng.gf:
subject_N = DictDot.aranikh_N;
topic_N = DictDot.aranikh_N;
theme_N = DictDot.aranikh_N;
lack_V2 = DictSimpleEng.lack_V2;
```

Figure 5: Two vocabulary wiki entries and generated GF definitions

some of these, it might be desirable to have dictionary entries,¹³ this process is almost impossible to automate and must be done by hand.

While DictDot and DictDotAbs are meant to be used in application grammars, the BigLexiconXXX files can be used for testing and translation. An example of this is shown in the next section.

2.6 Example

With the current implementation status, it is already possible to parse and translate a wide range of phrases. For experimentation and testing we supply the LangSimpleXXX abstract and concrete grammars which simply import the SyntaxSimpleXXX and BigLexiconXXX modules (completely analogous to the LangXXX module in the full RGL) and can be used directly from the GF command line:

```
> i simpleenglish/LangSimpleEng.gf
linking ... OK

Languages: LangSimpleEng
2745 msec
LangSimple> i simpledothraki/LangSimpleDot.gf
linking ... OK

Languages: LangSimpleDot LangSimpleEng
187 msec
LangSimple> p "the warrior which protected the turtle has stabbed
    my goat with a dagger ." | l
lajak fin viljazer adraes ray vinde dorv anni ma mihesofoon .
the warrior which protected the turtle has stabbed my goat with a
dagger .
lajak fin viljazer adraes ray vinde dorv ma mihesofoon anni .
the warrior which protected the turtle has stabbed my goat with a
dagger .

31 msec
```

The multiple outputs for the last command are quite typical and derive from the fact that the parse of the English sentence is ambiguous: Was the duck stabbed with a dagger or was a duck with a dagger stabbed? Note that, in this case, the Dothraki translations differ, because the possessive pronoun “anni” follows the noun phrase (either “dorv” = “goat” or “dorv ma mihesofoon” = “goat with a dagger”).

¹³For example, there could be an entry `roast_quail_CN : CN in BigLexicon with linearizations roast_quail_CN = mkCN DictEng.roast_A (mkCN DictEng.quail_N) in BigLexiconEng and roast_quail_CN = mkCN DictDot.mechikh_N in BigLexiconDot.`

3 Discussion

To conclude our project documentation, we will now discuss problems and limitations as well as briefly describe future plans before we summarize the project in its current state and our experience while working on it.

3.1 Limitations

To begin programming a resource grammar for the first time was not easy, since it is not possible to start with, for example, just one category and then test the implementation. As there are so many dependencies in the resource grammar library, it is recommended to take the resource grammar of a similar language and change it step by step. But this way, as long as our implementation of the Dothraki language was incomplete, the resource grammar would fill the gaps with the wrong language grammar. Another way is to start out with an independent mini resource grammar, but in that case you cannot make use of the resources of the library. Finding our own solution for this problem took time, but in the end we are satisfied with how it turned out.

For the implementation of a resource grammar, a good grammar source is needed (Grammatical Framework, 2016b). Although the full resource grammar for the Dothraki language is not yet complete, it is becoming harder and harder to find reliable resources regarding further aspects of grammar we would like to implement (e.g. usage of structural words). As it is a constructed language, and relatively new, we hope that further and more precise information will be documented and published with time. Yet, even if we had all the information needed for a full implementation, it would not have been possible for us to write a full resource grammar in such a short amount of time, especially as we are still new to the GF programming language and not experienced when it comes to implementing resource grammars. Hence, we focus on providing a good basis of implemented rules, so that we and others can expand the project in the future.

3.2 Outlook

In the future, we would like to expand our resource grammar to a full resource grammar and add it to the GF resource grammar library. We are already working on and hosting the project online, for everyone to see, so that the GF and/or Dothraki community may help us to achieve this goal. Some more parts we can implement right now, for some we have to wait until more information regarding the Dothraki language is being published. In the end, we also want to add the full vocabulary to our dictionary.

3.3 Conclusion

Now that the first milestone of our project lies behind us, we can evaluate the working process and the result. As we had some starting problems at the beginning of the

project, we are proud to have been able to create such a big portion of the resource grammar in the given time. By relying on the basics learned during the seminar and by aiming for such an ambitious goal, we surely learned a lot during the process. As we are convinced that our project can be helpful and interesting for other people in the future, we were motivated to keep on working on the resource grammar and to try to find the best solution for every problem. This term paper does not mark the end of this project, but we are curious to hear what you, our lecturers, and other people think of the work we have done so far.

Appendix

Source Code The complete source code can be viewed and downloaded via GitHub: https://github.com/mahen2/dothraki_gf

References

Brown, R. & Frederking, R. (1995). *Applying statistical English language modeling to symbolic machine translation*. In Proceedings of the sixth international conference on theoretical and methodological issues in machine translation (TMI-95), pp. 221–239.

Cognitus Apps (2016). Flamingo Dothraki. Retrieved from <https://play.google.com/store/apps/details?id=com.cognitusapps.flamingo.dothraki&hl=de>

Fun Translations (2016). Dothraki translator. Retrieved from <http://funtranslations.com/dothraki>

Détrez, G. & Camilleri, J.J. (2016). *Grammatical Framework*. GitHub.com. Retrieved from <https://github.com/GrammaticalFramework>

Google (2016). *Google Translate*. Retrieved from <https://translate.google.com/>

Google Groups (2016). *Problem with strange metavariables in parsing*. Retrieved from https://groups.google.com/d/msg/gf-dev/aRjt_2JfvA0/iTcZbTu5AAAJ

Grammatical Framework (2016a). *Translation with GF: Powered by multilingual grammars*. Retrieved from <http://www.grammaticalframework.org/demos/translation.html>

Grammatical Framework (2016b). *GF Resource Grammar Library: Synopsis*. Retrieved from <http://www.grammaticalframework.org/lib/doc/synopsis.html>

Martin, G.R.R. (2016). *Bibliography*. Retrieved from <http://www.georgerrmartin.com/bibliography/>

Henkel, M. & Kuckuck, B. (2016). *Building a resource grammar for Dothraki*. GitHub.com. Retrieved from https://github.com/mahen2/dothraki_gf

Peterson, D.J. (2014). *Living Language Dothraki: A Conversational Language Course Based on the Hit Original HBO Series Game of Thrones*. New York: Living Language.

Peterson, D.J. (2015). *The Art of Language Invention*. Penguin Books.

REFERENCES

- Peterson, D.J. (2016). *About Dothraki*. Retrieved from <http://www.dothraki.com/about-dothraki/>
- Ranta, A. (2009). *The GF Resource Grammar Library*. *Linguistic Issues in Language Technology*, 2(2), pp. 1–63.
- Ranta, A. (2011a). *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications.
- Ranta, A. (2011b). *Grammatical Framework: Programming with Multilingual Grammars. Slides for the GF book*. Retrieved from <http://www.grammaticalframework.org/gf-book/gf-book-slides.pdf>
- Ranta, A. (2015). *The Status of the GF Resource Grammar Library*. *Grammatical Framework*. Retrieved from <http://www.grammaticalframework.org/lib/doc/status.html>
- Tongues of Ice and Fire Wiki (2016). *Vocabulary*. Retrieved from <http://wiki.dothraki.org/Vocabulary>
- Wikipedia (2016). *Grammatical Framework*. Retrieved from https://en.wikipedia.org/wiki/Grammatical_Framework