

CHALMERS



Machine translation of trekking and climbing route descriptions using Grammatical Framework

Master of Science Thesis in Automation and Mechatronics

EMIL DJUPFELDT

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2012

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Machine translation of trekking and climbing route descriptions using Grammatical Framework

EMIL A. F. DJUPFELDT

© EMIL A. F. DJUPFELDT, August 2012.

Examiner: AARNE RANTA

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2012

Abstract

This thesis introduces a domain specific grammar for Grammatical Framework, as well as an iPhone application utilising the grammar and a C++ library to make parsing of the grammar possible on systems that does not easily include support for Java or Haskell.

Acknowledgements

I would like to thank my supervisor Professor Aarne Ranta for his insightful comments during the course of this work.

Emil Djupfeldt, Gothenburg 2012-08-01

Contents

1	Introduction	1
1.1	Goals	1
1.2	Structure	2
2	Background	3
3	GF and PGF	4
3.1	Grammatical Framework	4
3.1.1	Abstract syntax	4
3.1.2	Concrete syntax	4
3.1.3	Inheritance	4
3.2	Portable Grammar Format	5
4	PGF in C++	7
4.1	libpgf	7
4.2	JPGF	7
4.3	libpgf+	7
4.3.1	Memory handling	8
4.3.2	Exceptions	8
4.3.3	PGF	9
4.3.4	Reader	11
4.3.5	Parse tree	11
4.3.6	Lineariser	12
4.3.7	Parser	12
5	Mountaineering phrasebook in GF	14
5.1	Phrasebook	14
5.2	Mountaineering	14
5.2.1	Words	15
5.2.2	Phrases	15

5.2.3	Professions	16
6	The iPhone application	19
6.1	PhraseDroid	19
6.1.1	Sub-header	19
6.2	iPhrase	19
6.2.1	Grammarians	19
6.2.2	User interface	21
6.2.3	Reusability	24
7	Results	25
7.1	Results	25
7.1.1	Mountaineering phrasebook	25
7.1.2	libpgf+	25
7.1.3	iPhone application	25
7.2	Evaluation	26
7.2.1	Mountaineering phrasebook	26
7.2.2	libpgf+	26
7.2.3	iPhone application	26
7.3	Future work	27
7.3.1	Mountaineering phrasebook	27
7.3.2	libpgf+	27
7.3.3	The iPhone application	27
8	Conclusion	28

1

Introduction

THE GOAL is to create a system that can automatically translate descriptions and travel guides for trekking and climbing routes. Such texts are often found in the language of the country the route is situated in, which may not be the prospective climber's native tongue. It would be useful if the route descriptions could in some way be translated without the climber having to resort to dictionaries and their own sometimes limited knowledge of the foreign language. There are some pitfalls however. In both climbing and trekking, terms can be found that are not as common in everyday language. In order to handle this the system will need to be aware of these terms. It would be inconvenient and sometimes even dangerous not to translate them properly. There are multiple approaches to translating text. This work will concentrate on grammatically aware translation using Grammatical Framework (GF), a programming language for multilingual grammar applications. This is different from some other systems, e.g. Google Translate, that instead use statistical models to create translations.

1.1 Goals

During the course of this project there are a few tasks that need to be addressed:

- A sufficient amount of text in the target languages needs to be gathered. The texts should include enough domain specific jargon that the GF resource grammar library doesn't already cover it.
- A domain lexicon will be created to cover the jargon. The lexicon should have instances for at least Swedish, English and German. Support for Italian and/or French will be added later if there is time.
- An application that uses the domain lexicon will be developed. The application should be an iPhone app based on the existing GF Android application. Due to

differences in the two platforms the existing code most likely cannot be reused, and instead only used as a guideline. The new application should be easy to repurpose to use other lexicons so that it can be reused or extended for other projects.

- Translations generated by the system need to be validated to ensure that the system works as intended.

1.2 Structure

2

Background

CURRENT translation applications for the iPhone are usually based in some way on Google translate. This gives them a very large vocabulary, but it can also lead to invalid translations in cases where the models used by Google translate are lacking in data. This becomes more obvious when translating phrases to and from languages that are less prevalent on the internet which is Google main source of data, or when the phrase includes jargon from a specific domain.

3

GF and PGF

IN ORDER TO FULLY UNDERSTAND this work some basic understanding of the Grammatical Framework and the Portable Grammar format is needed.

3.1 Grammatical Framework

The Grammatical Framework (GF) is a type-theoretic grammar formalism based on Martin-Löf theory [?]. It can be used to write grammars for both natural and formal languages. A grammar for GF is made up of an abstract syntax common for all languages, and a set of concrete syntaxes, one for each supported language. Once a grammar is written it can be used for both parsing and linearisation of text in any of the grammar's supported languages.

3.1.1 Abstract syntax

The abstract syntax describes the semantical structure of the grammar. It declares the categories and functions of the grammar.[?]

3.1.2 Concrete syntax

A concrete syntax is a language specific implementation of the abstract syntax. It defines linearisations for the categories and functions declared in the abstract syntax. [?]

3.1.3 Inheritance

A very useful feature of GF is the ability for syntaxes to inherit from other syntaxes [?]. This makes it possible to make specialisations of already written grammars. For instance, the Mountaineering grammar discussed later extends the Phrasebook grammar that is part of the standard GF distribution. The inheritance in GF is a bit more advanced than

in i.e. Java or C++. Unlike Java (but like C++) it allows multiple inheritance [?] [?]. In addition to this it also allows negative inheritance [?]. That is, the ability to choose not to inherit some parts of the base grammar(s). This is similar to overriding in object oriented programming languages [?], the difference being the ability to completely remove a feature instead of just replacing it.

Allowing multiple inheritance can lead to situations where a grammar inherits from another grammar along to different paths, known as the diamond problem [?]. This is solved in GF by... [?]

Negative inheritance would lead to some issues in object oriented programming languages. If the static type [?] of a variable defines a method but the dynamic type [?] does not, this would most likely lead to a runtime error. However, when inheriting from another grammar in GF the sub-grammar does not inherit the type of the base grammar [?]. The closest thing to a base type for a concrete grammar is instead its abstract grammar. And the abstract grammar in turn has no base type. Thus the problem with the static type declaring a feature not present in the dynamic type does not arise.

3.2 Portable Grammar Format

A full description of the Portable Grammar Format (PGF) can be found in [?].

Types

PGF

Abstract

AbsFun

AbsCat

CatFun

Type

Hypo

Equation

Pattern

Expression

Literal

BindType

Concrete

PrintName

Sequence

Symbol

Alternative

CncFun

LinDef

ProductionSet

Production

PArg

CncCat

4

PGF in C++

THIS CHAPTER DESCRIBES the C++ library for reading pgf files and using the contained grammar to parse and linearise text. It is based largely on the existing Java library JPGF that is used in the Android application.

4.1 libpgf

There already exists a C library to work with PGF files [?]. However, it does not support predicting the next possible tokens given a sequence of previous tokens. As this was a rather prominent feature of the Android application, not providing it in the iPhone application was not an alternative which meant that this existing library unfortunately could not be used.

4.2 JPGF

The Android application uses the JPGF library [?]. It uses ... to parse the input and is thus able to predict possible continuations of the current token sequence.

The JPGF library is divided into four major parts: The PGF file reader, the lineariser, the parser, and finally the parse tree representation. These will be discussed further in the next section.

4.3 libpgf+

The C++ library retains most of the structure and api of the Java library. Some additions were necessary to account for the fact that C++ does not provide garbage collection, automatic reference counting or any other form of automatic memory management

	<i>Pros</i>	<i>Cons</i>
<i>Reference counting</i>	No external dependencies. Easy to implement.	Requires the programmer to always release acquired references by hand.
<i>Internal GC</i>	No external dependencies. Easy to use, the programmer does not need to do any manual release of references when they are no longer needed.	Very large project to implement. Outside the scope of this thesis.
<i>External GC</i>	Easy to use, the programmer does not need to do any manual release of references when they are no longer needed.	Adds external dependencies to the library.

Table 4.1: Comparison of memory management alternatives.

except on the stack [?]. Also, some changes were made in cases where there were duplicate methods with different names or where methods did not follow the general naming convention used in JPGF.

4.3.1 Memory handling

As C++ does not provide automatic memory management and JPGF relies on the garbage collector in Java taking care of all allocated objects that it no longer needs, something was needed to take care of this in the new implementation. There were some alternatives. One was to implement reference counting [?] in the api. Another alternative would have been to implement garbage collection (GC) [?] or rely on an external library to provide it. A comparison of the three alternatives can be seen in table 4.1.

From these three alternatives, reference counting was chosen. The implementation uses a base class which provides methods for counting references that is then inherited either directly or indirectly by all other classes in the library. The interface for the reference counting class can be seen in listing 4.1.

The reference counting implementation also provides a convenience function to simultaneously release a reference and clear the pointer to prevent lingering references. This function can be seen in listing 4.2.

4.3.2 Exceptions

There are a number of different things that can go wrong when handling a grammar. First of all, it might not be possible to read it from the PGF file for some reason. Other failures may arise in the parser or lineariser. JPGF indicates these failures with Java


```

class RefBase {
private:
    int referenceCounter;

public:
    RefBase();
    virtual ~RefBase();

    virtual void addReference();
    virtual void release();

    virtual std::string toString() const;
};

```

Listing 4.1: Base class for all reference counted classes.

```

template<class T> static inline void release(T*& ptr) {
    if (ptr != NULL) {
        ptr->release();
        ptr = NULL;
    }
}

```

Listing 4.2: Convenience method to release and clear references.

exceptions, which can fairly easy be translated to C++ exceptions.

To further simplify this, all exceptions thrown by the library were given a common base class shown in listing 4.3. Unlike most other classes in the library, this does not inherit from the reference counting base class. The reason for this is how C++ exception handling works. When an exception is thrown it is first constructed on the stack, and then copied into a buffer provided by the system. This buffer is then the responsibility of the system and will be automatically deallocated once the exception has been caught.

4.3.3 PGF

This is the main class representing the grammar loaded from a PGF file. The interfaces is almost identical to the corresponding class in JPGF and can be seen in listing 4.4. It provides methods to retrieve the abstract syntax of the grammar, to retrieve any of the concrete syntaxes available in the grammar and also to enumerate all of them. In addition it provides methods for getting the version information of the PGF file that it represents.

```

class Exception : public std::exception {
private:
    std::string message;

public:
    Exception();
    Exception(const std::string& message);
    virtual ~Exception() throw();

    virtual const std::string& getMessage() const;

    virtual std::string toString() const;

    virtual const char* what() const throw();
};

```

Listing 4.3: Base class for all exceptions thrown in the library.

```

class PGF : public RefBase {
public:
    virtual std::set<std::string> getConcreteNames() const;
    virtual gf::reader::Concrete* getConcrete(const std::
        string& name) const;
    virtual uint32_t getMajorVersion() const;
    virtual uint32_t getMinorVersion() const;
    virtual gf::reader::Abstract* getAbstract() const;
    virtual bool hasConcrete(const std::string& name) const
        ;
    virtual std::string toString() const;
};

```

Listing 4.4: Main class representing the grammar.

```

class PGFReader {
public:
    PGFReader(FILE* inputStream);
    PGFReader(FILE* inputStream, const std::set<std::string>
        & languages);
    virtual PGF* readPGF() throw(gf::IOException, gf::
        UnknownLanguageException);
};

```

Listing 4.5: Base class for all exceptions thrown in the library.

```

Lambda. Tree ::= "\\\" Ident "->" Tree ;
Variable. Tree ::= "$" Integer ;
Application. Tree ::= "(" Tree Tree ")" ;
Literal. Tree ::= Lit ;
MetaVariable. Tree ::= "META" Integer ;
Function. Tree ::= Ident ;

IntLiteral. Lit ::= Integer ;
FloatLiteral. Lit ::= Double ;
StringLiteral. Lit ::= String ;

```

Listing 4.6: BNF grammar used to generate the classes modelling the parse tree.

4.3.4 Reader

The reader consists of two parts. The first part is the PGFReader class that does the actual parsing of the PGF file. Its public api can be seen in listing 4.5. It consists only of two constructors and a method to read the PGF file and create a PGF object. Both constructors take a c stream as argument that will be used when reading the file. The second constructor also accepts a set of language names that will be used to filter which concrete syntaxes are loaded from the file.

The second part is the set of classes modelling the grammar that was read from the file. There are classes corresponding to each type described in section 3.2. The model is the same as the one used in JPGF and can be seen in figure 4.1.

4.3.5 Parse tree

The classes modelling the parse tree are automatically generated from the BNF grammar shown in listing 4.6 using bnfc [?]. This grammar is identical to the one used in JPGF and theoretically a tree parsed with libpgf+ could be converted to a string using this grammar and then read from the string with JPGF and linearised there, or vice versa.

4.3.6 Lineariser

4.3.7 Parser

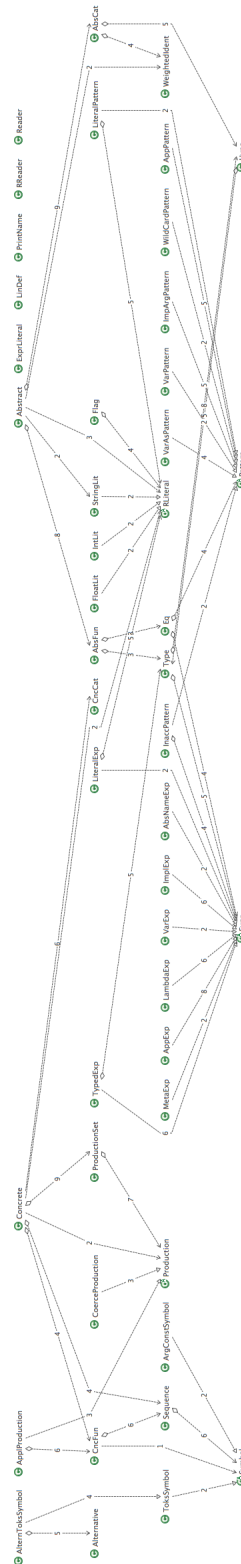


Figure 4.1: Class diagram of the grammar model.

5

Mountaineering phrasebook in GF

IN MOUNTAINEERING, like in all special areas of interest, there is a certain jargon. It includes some words and phrases which are uncommon or even has a different meaning when used without this context. However, a large amount of words and expressions are still common with plain language. (Otherwise it wouldn't be considered the same language any more.)

In GF there is already a tourist phrasebook grammar [?]. This can be used as a basis for the Mountaineering grammar so that only the jargon has to be added.

5.1 Phrasebook

The phrasebook grammar in the GF distribution provides several forms of useful questions and phrases. Some examples include:

- Where is the airport?
- Can you buy an apple?
- I can swim.
- This Italian pizza is good.

5.2 Mountaineering

The words and phrases present in mountaineering jargon varies from language to language. Some languages, like Norwegian, has a very rich vocabulary for this area. Others, like Swedish, mostly use loan words. English and German fall somewhere in between,

with German being on the richer side of the spectrum while English borrows some from German but not as much as Swedish does from English. [?]

5.2.1 Words

When deciding what words to include a few categories were identified, namely actions, exclamations, gear, holds, knots, places and people. Some of these could be placed in existing categories from the Phrasebook grammar, while others were given their own categories.

The main sources for words were climbing and mountaineering dictionaries on the internet [?] [?] [?].

The words were categorised as follows:

VerbPhrase Actions like abseil, climb, fall or stem.

Greeting Exclamations like "belay on", "falling!" or "rock!".

Kind Gear like carabiner, harness or rope.

HoldKind Holds like bucket, pocket or sloper.

KnotKind Knots like bowline, fisherman's knot or munter hitch.

PlaceKind Places like belay station, glacier or summit.

Profession People like belayer, climber or physician.

VerbPhrase, Kind and PlaceKind are inherited from the Phrasebook grammar, while the other four are new.

Profession is probably the most interesting category as it is used in the replacement of certain phrases from the Phrasebook grammar, discussed below.

5.2.2 Phrases

The Phrasebook grammar provides most of the phrases needed to use the added words. Some phrases were added though:

ADoVerbPhraseDirection Someone walks/climbs/etc to somewhere: "I walk to the hotel."

AModVerbPhraseDirection Someone can/can't do the above: "I can walk to the hotel."

IsAProfession Someone is a climber/physician/etc: "I am a climber."

```

cat
  Profession ;

fun
  ThisProfession , ThatProfession : Profession -> Person ;
    — this teacher , that teacher
  TheseProfessions , ThoseProfessions : Profession -> Person ;
    — these teachers , those teachers
  TheProfession , TheProfessions : Profession -> Person ;
    — the teacher , the teachers
  IsAProfession : Person -> Profession -> Action ;

```

Listing 5.1: New abstract syntax for Professions.

5.2.3 Professions

In the Phrasebook grammar there is a group of Actions on the form "I am a student." However, these are not very flexible and does not allow the profession to be reused in other phrases were a person is needed. I.e. "The student walks to the hotel." This is somewhat limiting as being able to say "The leader climbs to the belay station." can be rather useful.

To address this, the old professions were excluded when inheriting from the Phrasebook grammar, and instead a new system to express both the old phrases and the new were designed.

First, a new category Profession as described in section 5.2.1 were added. Then functions to use professions as persons were created. Finally the function IsAProfession was added to replicate the "I am a student." type of phrases. The abstract and concrete syntax for this is shown in listings 5.1, 5.2 and 5.3. This can be compared with the old functions from Phrasebook in listings 5.4 and 5.5. The new way makes the grammar a bit more complex, but in return adding more professions is easier and the ways they can be used are more flexible.


```

lincat
  Profession = N;

lin
  ThisProfession pro = {name = mkNP this_Quant pro ; isPron =
    False ; poss = this_Quant};
  ThatProfession pro = {name = mkNP that_Quant pro ; isPron =
    False ; poss = that_Quant};
  TheseProfessions pro = {name = mkNP this_Quant plNum pro ;
    isPron = False ; poss = this_Quant};
  ThoseProfessions pro = {name = mkNP that_Quant plNum pro ;
    isPron = False ; poss = that_Quant};
  TheProfession pro = {name = mkNP the_Quant pro ; isPron =
    False ; poss = the_Quant};
  TheProfessions pro = {name = mkNP the_Quant plNum pro ;
    isPron = False ; poss = the_Quant};
  ProTeacherMale, ProTeacherFemale = teacher_N;
  ProPhysicianMale, ProPhysicianFemale = doctor_N;

```

Listing 5.2: New shared concrete syntax for Professions.

```

lin
  IsAProfession p pro = mkProfession pro p;
  ProBelayMale, ProBelayFemale = mkN "belay";
  ProClimbMale, ProClimbFemale = mkN "climb";
  ProLeadMale, ProLeadFemale = mkN "lead";
  ProSecondMale, ProSecondFemale = mkN "second";

```

Listing 5.3: New english concrete syntax for Professions.

```

fun
  ADoctor : Person -> Action ;
  AProfessor : Person -> Action ;
  ALawyer : Person -> Action ;
  AEngineer : Person -> Action ;
  ATeacher : Person -> Action ;
  ACook : Person -> Action ;
  AStudent : Person -> Action ;
  ABusinessman : Person -> Action ;

```

Listing 5.4: Old abstract syntax for Professions.

```
lin
  ADoctor = mkProfession (mkN "doctor") ;
  AProfessor = mkProfession (mkN "professor") ;
  ALawyer = mkProfession (mkN "lawyer") ;
  AEngineer = mkProfession (mkN "engineer") ;
  ATeacher = mkProfession (mkN "teacher") ;
  ACook = mkProfession (mkN "cook") ;
  AStudent = mkProfession (mkN "student") ;
  ABusinessman = mkProfession (mkN "businessman" "businessmen"
    ")") ;

oper
  mkProfession : N -> NPPerson -> Cl = \n,p -> mkCl p.name n
    ;
```

Listing 5.5: Old english concrete syntax for Professions.

6

The iPhone application

HERE AN INTRODUCTION of the Android application is given, after which follows a more in-depth description of the structure of the iPhone application.

6.1 PhraseDroid

6.1.1 Sub-header

6.2 iPhrase

The developed iPhone application can roughly be divided into two parts. There is the user interface to provide interactivity and present results, and the Grammarian that interfaces with libpgf+.

6.2.1 Grammarian

The Grammarian class is the glue between the C++ api of libpgf+ and the Objective C code in the rest of the application. It provides methods to enumerate available languages, to translate between three-letter languages codes and full language names, and most importantly to parse input, generate translations and predict continuations of the current input. The public interface of the class is shown in listing 6.1.

The enumeration of available languages is done by querying libpgf+ for the list of concrete syntaxes for the current grammar. These names are not very user friendly though. Therefor a method to generate a human readable name is provided. This method extracts the three letter code at the end of the concrete syntax name and looks it up in a table with all the ISO 639 [?] language codes and their corresponding language names.

Parsing is done by accepting one token at a time and passing it on to libpgf+, keeping a reference to the current parser state in the grammarian. This state is then queried for

```

@interface Grammarian : NSObject
- (id) init ;
- (id) initWithLanguage:( NSString*) language ;

+ (NSArray*) languages ;
+ (BOOL) hasLanguage:( NSString*) language ;
+ ( NSString*) codeForLanguage:( NSString*) language ;
+ ( NSString*) languageForCode:( NSString*) code ;
+ ( NSString*) humanReadableNameOfLanguageFromCode:( NSString*)
    code ;
+ ( NSString*) humanReadableNameOfLanguage:( NSString*) language ;

- ( NSString*) sourceLanguage ;

- (NSArray*) predict:( NSString*) prefix ;
- (NSArray*) predict:( NSString*) prefix  withEditDistance:( int )
    distance ;
- (NSArray*) match:( NSString*) token  withEditDistance:( int )
    distance ;
- (NSArray*) matchIgnoringCase:( NSString*) token ;

- (BOOL) accept:( NSString*) token ;
- (void) reset ;
- ( int ) acceptedTokenCount ;

- (NSArray*) parseTrees ;
- (NSArray*) translationsForLanguage:( NSString*) language ;
@end

```

Listing 6.1: Public interface of the Grammarian class.

predictions which are cached until needed.

Translations are generated by asking the current parser state for all available parse trees and then handing them over to the concrete syntax of the grammar corresponding to the requested target language. The resulting linearisations are then returned to the caller.

There are two methods to predict continuations. The first method uses simple prefix matching on the list of cached predictions. The second method calculates the Damerau–Levenshtein distance [?] between the supplied string and each token, and only returns those tokens that either has the supplied string as a prefix or has an edit distance less than or equal to the supplied number.

6.2.2 User interface

The user interface of the application consists of five different views that can be accessed through the flow shown in 6.1. The starting view is the input view. The main part of this view is occupied by the token input and the keyboard. At the top of the view are two buttons to transition to either the settings view or the translations view. The translation button is only available if the current input can be parsed to a top level production by the grammar.

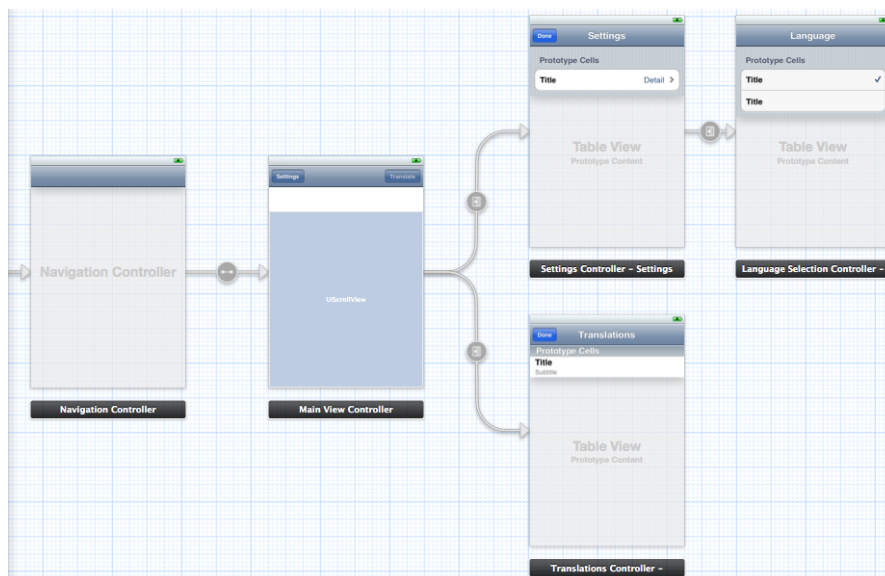


Figure 6.1: UI flow

Token input

Token input can be done in two ways. One way is to touch one of the token buttons shown in the token input view. The other way is to enter text manually into the provided text field.

The token input view shows the possible continuations of the current input. The list of possible tokens is provided by the Grammarian as described in 6.2.1. A button is created for each token. The buttons are then laid out to fit in the current width of the token main input view as seen in figures 6.2 and 6.3. The height of the token input view is then adjusted to fit all the buttons to enable scrolling in the parent view.

Touching a token will tell the advance the grammarian using the corresponding token, after which the token input view will be updated to reflect the newly available predictions.

When the grammarian is advanced the new token will also be added to a list of processed tokens along with a corresponding button visible at the top of the token input just left of the input text field.

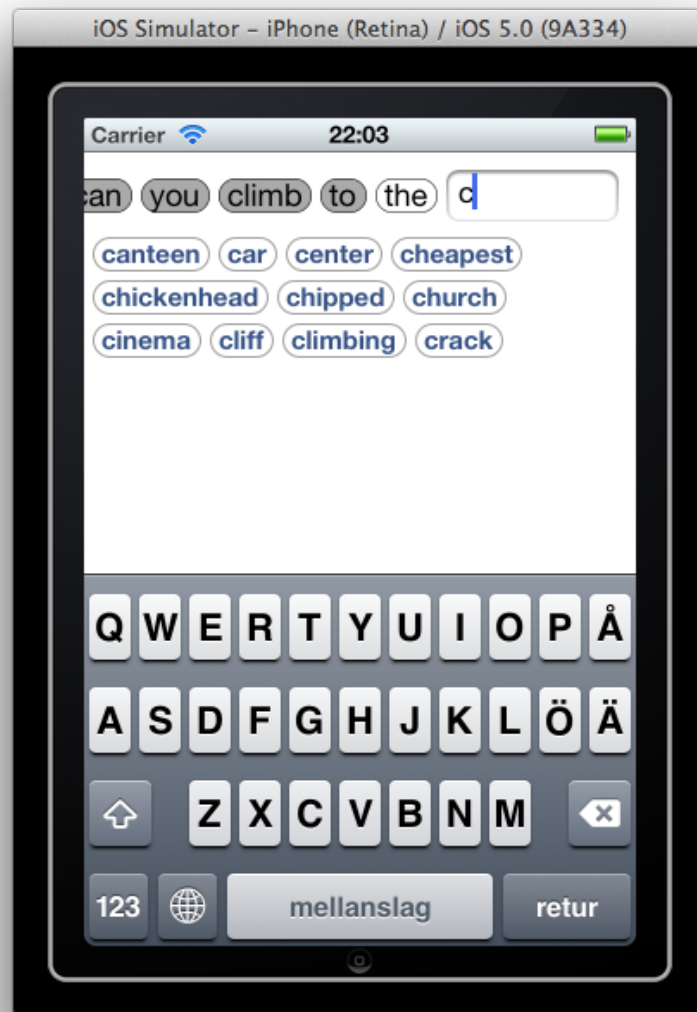


Figure 6.2: Input view (portrait)

If text is entered into the text field the entered text will be used as a prefix to limit the list of tokens returned by the grammarian. If no tokens are returned, a second query for tokens is performed but this time with an allowed maximum edit distance of one as explained in 6.2.1. This is to make allowances for the user misspelling a token.

If a space is entered and the current text is a valid token or has a maximum edit distance of one from one and only one valid token the grammarian is advanced as if the corresponding token button had been touched.



Figure 6.3: Input view (landscape)

If the text field is empty and a back space is entered, the previous input token will be removed from the list of processed tokens and instead be placed in the text field. This same effect can also be achieved by touching the button corresponding to the last processed token.

Translations

The translations view shows a list of the generated translations for the given input and also a using a disambiguation concrete syntax if the current grammar supports it. Touching any of the translations will take the user to the translation details view. The translations view, like the settings view, features a button to return to the input view.

Translation details

The translation details view shows the full text of the translation and below it the full text of the disambiguation if available. There is also a back button to return to the translations view.

Settings

The settings view shows a list of the available options in the application. Currently this is the from and to languages for the translation. Touching one of these will take the user

to the language selection view. The settings view also features a button to return to the input view.

Language selection

The language selection view dynamically creates a list of all the languages available in the current grammar. The active language has a mark to indicate it is in use. Touching any of the languages in the list will make that language active for the current setting (to/from) and return to the settings view. There is also a back button to return to the settings view without changing the active language.

6.2.3 Reusability

The grammar used in the application is loaded from a PGF file. This allows the grammar to be replaced without having to change the whole application.

7

Results

THIS CHAPTER PRESENTS the results of this work and discusses possible further work.

7.1 Results

7.1.1 Mountaineering phrasebook

The mountaineering phrasebook extends the Phrasebook grammar with 78 new words specific to this domain. It adds two new kinds of phrases, and replaces one old with a new implementation. It also extends the Person category to include professions.

7.1.2 libpgf+

A comparison of the GF, JPGF, libpgf and libpgf+ parsers can be seen in table 7.1. The comparison was done using the Mountaineering grammar and a list of 1000 randomly generated phrases supported by the grammar. Time measurements for GF were taken with the unix *time* command while running GF in batch mode, and is the sum of both user and system time (that is, total cpu time). The time measurements for JPGF were taken using the Netbeans profiler while the measurements for libpgf and libpgf+ were taken using the OS X profiler (Instruments).

7.1.3 iPhone application

A comparison of the available features in the Android and the iPhone applications can be seen in 7.2.

	<i>Predictions</i>	<i>PGF load time</i>	<i>Average parse time</i>
<i>GF</i>	Yes	X ms	Y ms
<i>JPGF</i>	Yes	X ms	Y ms
<i>libpgf</i>	No	X ms	Y ms
<i>libpgf+</i>	Yes	X ms	Y ms

Table 7.1: Comparison of parsers.

	<i>Phrasedroid (Android)</i>	<i>iPhrase (iPhone)</i>
<i>Load from PGF</i>	Yes	Yes
<i>Change input language</i>	Yes	Yes
<i>Change output language</i>	Yes	Yes
<i>Token touch input</i>	Yes	Yes
<i>Keyboard input</i>	No	Yes
<i>List all possible translations</i>	Yes	Yes
<i>Text-to-speech of translation</i>	Yes	No

Table 7.2: Comparison of memory management alternatives.

7.2 Evaluation

7.2.1 Mountaineering phrasebook

The new words in the grammar relates mostly to climbing. While it is one important aspect of mountaineering, there are other areas that would benefit from being covered by the grammar as well.

7.2.2 libpgf+

The libpgf+ library has the same functionality as the JPGF library. While this does not include full support for everything that can be expressed using PGF (most notably XXXX), they provide all the features necessary to implement a working phrase translation application.

7.2.3 iPhone application

With the same feature set as the Android application except for text-to-speech, the iPhone application should be considered a successful reimplementaion. Also, the addition of the keyboard input is very useful when the number of possible continuations is

very large.

7.3 Future work

7.3.1 Mountaineering phrasebook

The grammar can of course be extended with a larger vocabulary and more phrases. Some examples include alpine flora and fauna, and phrases for asking for/giving directions to get from one place to another.

7.3.2 libpgf+

Adding support for XXXX would make the library feature complete with regards to the Portable Grammar Format. This would also benefit JPGF, since the code base is similar enough that porting features from one to the other would not be much of a problem.

7.3.3 The iPhone application

There is always room for improvement in the user interface of an application. The input interface works fairly well, but the presentation of results could need some improvement. An additional setting to allow the user to choose between several different installed grammars would also be useful.

8

Conclusion

THIS WAS A TRIUMF! I'm making a note here: "HUGE SUCCESS!!"