CHALMERS

Grammatical Framework on the iPhone using a C++ PGF parser

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 2013

Grammatical Framework on the iPhone using a C++ PGF parser

EMIL A. F. DJUPFELDT

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 September 2013

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.

The grammar covers phrases and words related to mountainering. It is based on and extends the Phrasebook grammar from the Grammatical Framework.

The C++ library is a port of the existing Java parser and retains a similar API and structure, with allowances for differences in the two languages.

The iPhone application provides a graphical user interface for the C++ library and utilises the mountaineering grammar, allowing the user to easily input phrases and browse translations.

Acknowledgements

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

Emil Djupfeldt, Gothenburg 2013-09-11

Contents

1

Introduction

The goal is to create an iPhone application that can automatically translate phrases related to trekking and climbing.

Climbers often find themselves in foreign countries since the mountains are located where they are, and it is not always the case that the climber knows the native tongue. Using English to communicate and ask for help is not always an option even if both parties of the conversation can speak it to some degree. This is due to the fact that climbing and trekking, like many other special areas of interest, involves quite a bit of jargon that both parties would need to know not only in their native tongue but now also in English and preferably the other party's native tongue.

To solve this, a phrasebook application would be useful. With a grammar containing jargon for this specific domain as well as some more common words and phrases it could be used to bridge some of the language gap between speakers, as well as help a climber understand climbing and trekking route descriptions in foreign languages.

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 domain specific jargon in the target languages needs to be gathered. The jargon should consist of words and phrases that the GF resource grammar library does not cover already.
- A domain lexicon will be created to cover the jargon. The lexicon should have instances for at least Swedish, English and German.

- 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

This work is divided into several sections. First an introduction of the Grammatical Framework is given. Thereafter follows three chapters describing the c++ library, the grammar and the iPhone application. Finally the results are presented and conclusions are drawn.

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.

There is currently no other translation applications based on the Grammatical Framework for the iPhone, though one exists for Android [**?**].

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 sementical 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 beeing 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 not a problem in GF though, as everything inherited are constants and the compiler can recognise if two contants come from the same source[**?**].

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

When parsing using a GF grammar, it is most suitably represented by a Parallel Multiple Context-Free Grammar (PMCFG) [**?**] [**?**]. This grammar can be written to file using the Portable Grammar Format (PGF) [**?**].

There are some differences between PMCFG and GF abstract and concrete syntaxes. PCMFG does not allow parameters or nested records and tables, which are both used in the concrete syntaxes of GF. The lack of parameters is addressed by instantiating all parameter variables with all values they can possibly receive when compiling a PGF file. The nested records and tables are by simply flattening them.

Another step that is performed when compiling a PGF file is to replace all linearisation rules with concrete functions. One linearisation rule will be replaced with one or more concrete functions. A simple example is a linearisation for a category generating one singular and one plural concrete function. This expansion then propagates, so if another linearisation depends on the one from the previous example it will also generate both a singular and a plural concrete function.

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, at the start of this work it did not support predicting the next possible tokens given a sequence of previous tokens. As this was a rather prominent feature of the Android aplication, 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 represenatation. 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 necesarry to account for the fact that C++ does not provide garbage collection, automatic reference counting or any other form of automatic memory management

	Pros	*Cons*
Reference counting	No external dependencies. Easy to implement.	Requires the programmer to always release acquired references by hand.
Internal GC	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.
External GC	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 simultaenously 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 exceptions, which can fairly easy be translated to C++ exceptions.

```
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.

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 in the PGF file format [**?**]. 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.

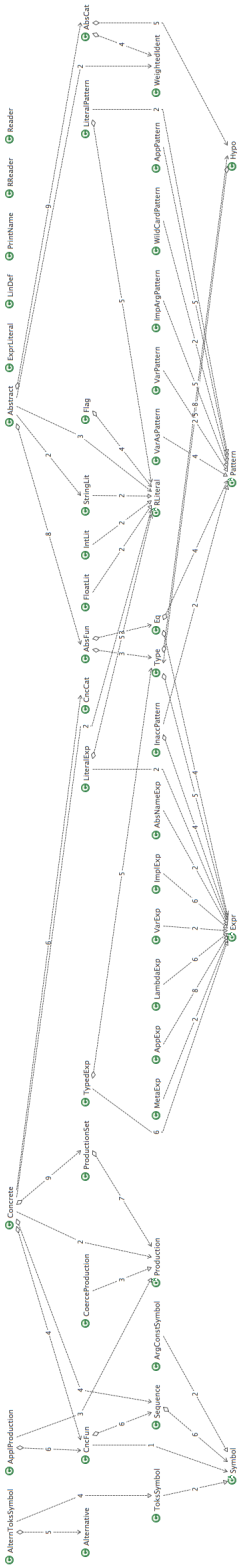

Figure 4.1: Class diagram of the grammar model.

5

Mountaineering phrasebook

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 where identified, namely actions, exclamations, gear, holds, knots, places and people. Some of these could be placed in existing categories from the Phrasebook grammar, while other where given their own categories.

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

The words where categorised as follows:

VerbPhrase Actions like 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.”

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.

```
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.

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;
  ProBelayerMale, ProBelayerFemale = mkN "belayer";
  ProClimberMale, ProClimberFemale = mkN "climber";
  ProLeaderMale, ProLeaderFemale = mkN "leader";
  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

PhraseDroid is an android application utilising the JPGF library to parse and translate the user's input. The application welcomes the user with a language selection page with a flag for each language. After selecting a language the user can begin input by touching one of the available tokens presented on the screen, similar to the fridge magnets GF web service. There is also an option to change the target language.

Once a valid sentence is formed the user can touch a button to translate. The application will then present the user with the available translations along with disambiguations. If the target language is supported by the Android OS text-to-speech service, a button is available for each translation to read the translation out loud.

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 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.

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

```
@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.

just left of the input text field.

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 advaned as if the corresponding token button had been touched.

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.

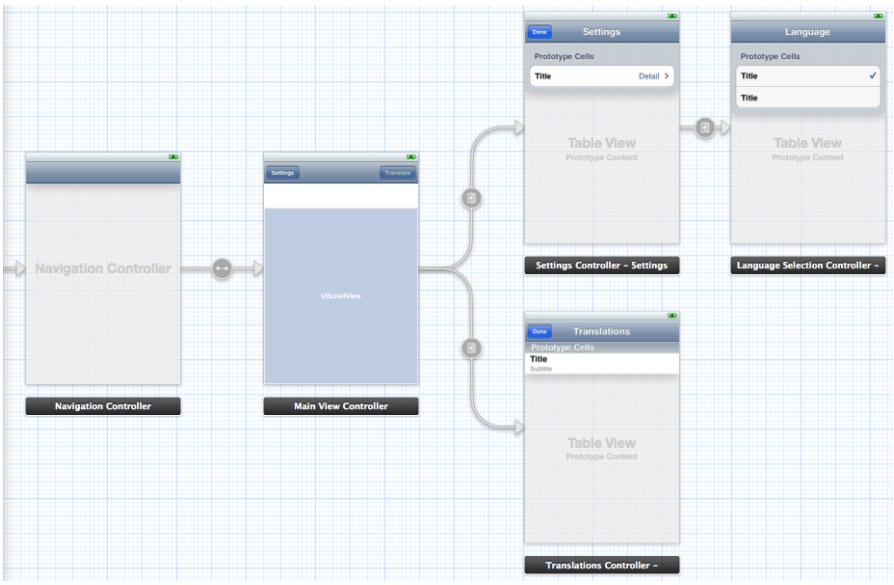

Figure 6.1: UI flow

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

Figure 6.2: Input view (portrait)

(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.

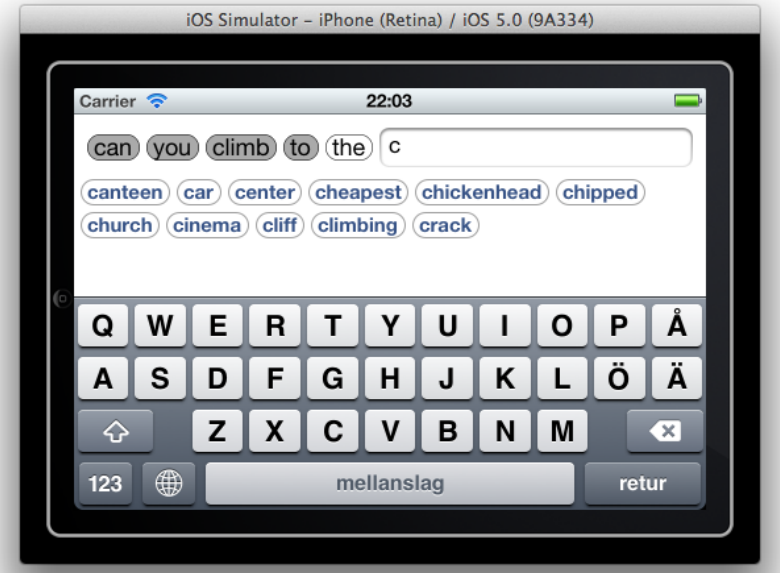

Figure 6.3: Input view (landscape)

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+ 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+ 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.

	PGF load time	*Average parse time*
GF	171 ms	6.1 ms
JPGF	1129 ms	8.5 ms
libpgf+	56 ms	17 ms

Table 7.1: Comparison of parsers.

	Phrasedroid (Android)	*iPhrase (iPhone)*
Load from PGF	Yes	Yes
Change input language	Yes	Yes
Change output language	Yes	Yes
Token touch input	Yes	Yes
Keyboard input	No	Yes
List all possible translations	Yes	Yes
Text-to-speach of translation	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. This means that it provides all the features necesarry 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-speach, the iPhone application should be considered a successful reimplementation. 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+

The api could be better documented. This would also benefit JPGF, since they share a common api with the exception for memory handling. There might also be bugs in the code that has not been found yet.

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

Surprisingly, the Java version of the parser was actually faster than the C++ version. This is most likely due to recursive nature of the algorithm and the fact that the Java version is partly written in Scala which handles recursion better than Java or C++. Another factor which might affect the result is the Java just-in-time compiler which would further optimize the code at run time, compared to the static optimizations done for the C++ code at compile time.

The grammar includes a little under 100 new words and a few new phrases relating to climbing in three different languages. This is a great start for a climbing and trekking grammar, but it can of course be extended.

Finally, the iPhone application works as intended and presents a user interface similar to the Android application. The grammar in the application can easily be replaced by another and thus the reusability requirements are met. Translations and prediction can be a bit slow for very long sentences which is a result of the somewhat limited cpu in the iPhone and the fact that the libpgf+ library is slower than the JPGF library.

8.1 Source code

The source code for the grammar, library and application can be found at the following urls:
`http://emil.djupfeldt.se/mscthesis/src/`
`http://emil.djupfeldt.se/mscthesis/git/`