Development of multimodal and multilingual grammars: viability and motivation

Björn Bringert Robin Cooper Peter Ljunglöf
Aarne Ranta

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable 1.2a

19/01/05

Project funded by the European Community
under the Sixth Framework Programme for
Research and Technological Development

The deliverable identification sheet is to be found on the reverse of this page.

Project ref. no.	IST-507802
Project acronym	TALK
Project full title	Talk and Look: Tools for Ambient Linguistic Knowledge
Instrument	STREP
Thematic Priority	Information Society Technologies
Start date / duration	01 January 2004 / 36 Months

Security	Public
Contractual date of delivery	Dec 04
Actual date of delivery	19/01/05
Deliverable number	1.2a
Deliverable title	Development of multimodal and multilingual grammars: viability and motivation
Type	Report
Status & version	Public Final
Number of pages	31 (excluding front matter)
Contributing WP	1
WP/Task responsible	UGOT
Other contributors	
Author(s)	Björn Bringert, Robin Cooper, Peter Ljunglöf and Aarne Ranta
EC Project Officer	Kimmo Rossi
Keywords	grammar, multilingual, multimodal, Grammatical Framework, dialogue systems

The partners in TALK are:

Saarland University	USAAR
University of Edinburgh HCRC	UEDIN
University of Gothenburg	UGOT
University of Cambridge	UCAM
University of Seville	USE
Deutches Forschungszentrum fur Künstliche Intelligenz	DFKI
Linguamatics	LING
BMW Forschung und Technik GmbH	BMW
Robert Bosch GmbH	BOSCH

For copies of reports, updates on project activities and other TALK-related information, contact:

The TALK Project Co-ordinator
Prof. Manfred Pinkal
Computerlinguistik
Fachrichtung 4.7 Allgemeine Linguistik
Postfach 15 11 50
66041 Saarbrücken, Germany
pinkal@coli.uni-sb.de
Phone +49 (681) 302-4343 - Fax +49 (681) 302-4351

Copies of reports and other material can also be accessed via the project's administration homepage, http://www.talk-project.org

Contents

Chapter 1

Introduction

Many large-scale working dialogue systems in research and development do not use a grammar but instead use statistical language models (SLMs) for speech recognition and word or phrase spotting instead of deep parsing in order to create input to the dialogue manager. On the other hand, less ambitious but commercially deployed systems often make use of elementary grammars, e.g. using VoiceXML. Grammar based systems are more accurate when the user speaks within the coverage of the grammar whereas robust systems using SLMs and phrase spotting give better results when the grammar fails. This leads naturally to the suggestion (e.g., Knight et al. (2001)) that systems should be hybrid and make use of both grammars and statistical models.

We are interested in building multilingual multimodal dialogue systems which are clearly recognisable to the user as the same system even if they use the system in different languages or in different domains using different mixes of modalities (e.g. in-house vs in-car, and within the in-house domain with vs without a screen for visual interaction and touch/click input). We wish to be able to guarantee that the functionality of the system is the same under the different conditions. We in addition would ultimately like the user to be able to change language or mode in the middle of a dialogue and be able to continue without restarting the system. Scenarios for changing language might be a user beginning a dialogue with a system in English and then realising that they prefer to continue in their native language. Scenarios for changing mode mid-dialogue might involve a user moving an application (e.g. a PDA) from the house to the car in the middle of a dialogue or simply walking away from the screen within the house.

Our previous experience with building such multilingual dialogue systems is that there is a software engineering problem keeping the linguistic coverage in sync for different languages. If all necessary grammars are constructed purely by hand it is very difficult to guarantee that everything that needs to be said is covered in a collection of different languages. This problem is compounded by the fact that for each language it is normally the case that a dialogue system requires more than one grammar, e.g. one grammar for speech recognition and another for interaction with the dialogue manager. Thus multilingual systems become very difficult to develop and maintain.

In this deliverable we will explain the nature of the Grammatical Framework and how it may provide us with a solution to this problem. The system is oriented towards the writing of multilingual and multimodal grammars and forces the grammar writer to keep a collection of grammars in sync. It does this by using computer science notions of abstract and concrete syntax. Essentially abstract syntax corresponds to the domain knowledge representation of the system and several concrete syntaxes characterising both natural language representations of the domain and representations in other modalities are related to a

single abstract syntax. The system forces the concrete syntaxes to give complete coverage of the abstract syntax and thus will immediately tell the grammar writer if the grammars are not in sync. In addition the framework provides possibilities for converting from one grammar format to another and for combining grammars and extracting subgrammars from larger grammars.

Chapter 2

Introduction to GF and multilingual grammar writing[1]

2.1 Separating abstract and concrete syntax

The main idea of Grammatical Framework (GF) is the separation of abstract and concrete syntax. The abstract part of a grammar defines a set of abstract syntactic structures, called abstract terms or trees; and the concrete part defines a relation between abstract structures and concrete structures.

The distinction between abstract and concrete syntax has been made by several authors since the late 1950's; McCarthy (1963) and Landin (1966) made the distinction in describing the syntax for programming languages; Chomsky (1957, 1965) made the distinction between (abstract) *deep structure* and (concrete) *surface structure*, together with transformations between the structures; Curry (1963) introduced the distinction under the headings of *tectogrammatic* and *phenogrammatic* structure; and Montague (1974) viewed a grammar as a set of rules linearizing logically interpreted (abstract) analysis trees into (concrete) strings of a natural language.

GF has a *linearization* perspective to grammar writing, where the relation between abstract and concrete is viewed as a mapping from abstract to concrete structures, called *linearization terms*. In some cases the mapping can be partial or even many-valued.

2.2 Linguistic advantages

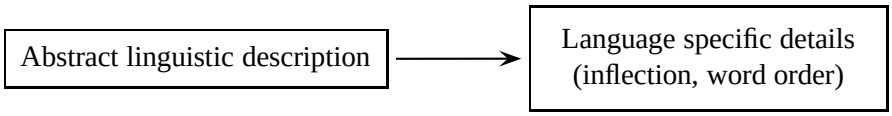
Although not exploited in many well-known grammar formalisms, a clear separation between abstract and concrete syntax gives some advantages.

2.2.1 Higher-level language descriptions

The grammar writer has a greater freedom in describing the syntax for a language. When describing the abstract syntax he/she can choose not to take certain language specific details into account, such as

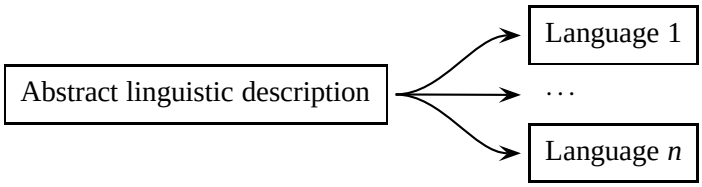
[1] This section is an excerpt from the introduction chapter of Ljunglöf (2004)

inflection and word order. Abstracting away smaller details can make the grammars simpler, both to read and understand, and to create and maintain.



2.2.2 Multilingual grammar writing

It is possible to define several different concrete syntax mappings for one particular abstract syntax. The abstract syntax could e.g. give a high-level description of a family of similar languages, and each concrete mapping gives a specific language instance.



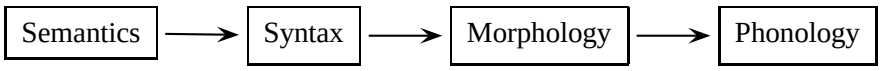
This kind of multilingual grammar can be used as a model for interlingua translation between languages. But we do not have to restrict ourselves to only multilingual grammars; different concrete syntaxes can be given for different modalities. As an example, consider a grammar for displaying time table information. We can have one concrete syntax for writing the information as plain text, but we could also present the information in the form of a table output as a LaTeX file or in Excel format, and a third possibility is to output the information in a format suitable for speech synthesis.

2.2.3 Syntax editing

It is possible to write documents by directly editing the abstract syntax, and let the program display the resulting concrete syntax. This was done for programming languages in e.g. the systems Mentor (Donzeau-Gouge et al., 1975) and Cornell Program Synthesizer (Teitelbaum and Reps, 1981); and has been generalized to natural language grammars and even *multilingual document authoring* (Dymetman et al., 2000; Khegai et al., 2003), where a document is written simultaneously in several languages. One example of multilingual authoring is when writing technical user manuals which should have exactly the same interpretation in any language.

2.2.4 Several descriptional levels

Having only two descriptional levels is not a restriction; this can be generalized to as many levels as is wanted, by equating the concrete syntax of one grammar level with the abstract syntax of another level. As an example we could have a spoken dialogue system with a semantical, a syntactical, a morphological and a phonological level. This system has to define three mappings; *i*) a mapping from semantical descriptions to syntax trees; *ii*) a mapping from syntax trees to sequences of lexical tokens; and *iii*) a mapping from lexical tokens to lists of phonemes.



This formulation makes grammars similar to transducers (Karttunen et al., 1996; Mohri, 1997) which are mostly used in morphological analysis, but has been generalized to dialogue systems by Lager and Kronlid (2004).

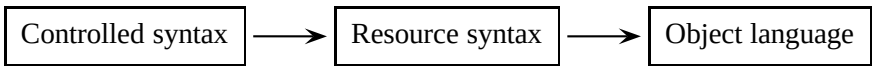
2.2.5 Grammar composition

A multi-level grammar as described above, can be viewed as a "black box", where the intermediate levels are unknown to the user. Then we are back in our first view as a grammar specifying an abstract and a concrete level together with a mapping. In this way we can talk about *grammar composition*, where the composition $G_2 \circ G_1$ of two grammars is possible if the abstract syntax of G_2 is equal to the concrete syntax of G_1. The result of the composition is the grammar inheriting the abstract syntax from G_1, the concrete syntax from G_2, and having the linearization mapping $f_2 \circ f_1$, where f_1, f_2 are the linearization mappings for G_1, G_2 respectively.

If the grammar formalism supports this, a composition of several grammars can be pre-compiled into a compact and efficient grammar which doesn't have to mention the intermediate domains and structures. This is the case for e.g. finite state transducers, but also for GF as has been shown by Ranta (2004b).

2.2.6 Resource grammars

The possibility of separate compilation of grammar compositions, opens up for writing *resource grammars* (Ranta, 2004b). A resource grammar is a fairly complete linguistic description of a specific language. Many applications do not need the full power of a language, but instead want to use a more well-behaved subset, which is often called a *controlled language*. Now, if we already have a resource grammar, we do not even have to write a concrete syntax for the desired controlled language, but instead we can specify the language by mapping structures in the controlled language into structures in the resource grammar.



2.3 Comparison with some grammar formalisms

Here we compare some existing grammar formalisms from the perspective of the ability to separate abstract and concrete syntax. We have no intention of giving a full description of the formalisms, and the reader can safely skip any part of this section.

2.3.1 Context-free grammar (CFG)

A context-free grammar has no separation of abstract and concrete syntax whatsoever. There is only one level of syntax rules, defining both the abstract syntax trees and the concrete language. The concrete syntax is not structured at all, making it impossible, or at least very complicated, to have several descriptional levels.

2.3.2 Head grammar (HG)

Head grammar (Pollard, 1984) is an extension of CFG, where the concrete syntax is *headed strings*, which can be concatenated or *wrapped* inside another headed string. There is not much structure in the concrete syntax, and the abstract syntax is tightly connected to the concrete word order.

2.3.3 Categorial grammar (CG) Combinatory categorial grammar (CCG)

Categorial grammar (Ajdukiewicz, 1935; Bar-Hillel, 1953; Lambek, 1958) is equivalent to CFG, but instead of grammar rules it has complex *functional categories*, together with rules for *function application*. Combinatory categorial grammar (Steedman, 1985, 1986) also adds rules for *function composition* to the framework, thus yielding an extension of CFG.

The notion corresponding to abstract syntax is the derivation trees, and they are tightly bound to the order of the given words. There are extensions (e.g. type logical grammar; Morrill, 1994) that add some word order freedom, but the concrete syntax is nevertheless simple strings. This means that CG and relatives are similar to CFG when it comes to separating abstract and concrete syntax.

2.3.4 Indexed grammar (IG) Linear indexed grammar (LIG)

Indexed grammar (Aho, 1968) and linear indexed grammar (Gazdar, 1987) are also extensions of CFG. In these formalisms the context-free categories are augmented with a *stack of indices*. On each application of a rule, an index can be pushed onto or popped from a stack. But the abstract syntax as represented by the syntax tree is still tightly connected to the concrete syntax of strings.

2.3.5 Tree adjoining grammar (TAG)

Tree adjoining grammar (Joshi et al., 1975; Joshi and Schabes, 1997) is a formalism based on trees and a tree rewriting operation called *adjunction*. It shares the basic problem with CFG, that there is only one descriptional level; syntax trees are directly correlated to the concrete word order.

2.3.6 Generalized context-free grammar (GCFG)

Generalized context-free grammar was introduced by Pollard (1984) as a mathematical framework for describing HG. GCFG can be seen as a very nice example of the separation of abstract and concrete syntax. Since GCFG is a very expressive grammar formalism involving general (Turing-complete) partial functions, its main usage is as a framework for specifying more restricted grammar formalisms.

2.3.7 Linear context-free rewriting systems (LCFRS) Parallel multiple context-free grammar (PMCFG)

Linear context-free rewriting systems (Vijay-Shanker et al., 1987) and parallel multiple context-free grammar (Seki et al., 1991) are defined as instances of GCFG where the linguistic objects are tuples of strings. The operations associated with syntax rules are only allowed to use tuple projection and string concatenation, and LCFRS has some extra restrictions on the linearization functions to ensure mild context-sensitivity. Since they are defined as GCFG, they share the same separation of abstract and concrete syntax. The only drawback is that the concrete syntax is restricted to string tuples.

2.3.8 Literal movement grammar (LMG) Range concatenation grammar (RCG)

These formalisms are very similar; a grammar is seen as a collection of Horn-like clauses over predicates, just as in the programming language Prolog. Groenink (1997a,b) introduced literal movement grammar, where predicates range over tuples of strings, making the formalism Turing-complete. There are also restricted variants called simple LMG and range concatenation grammar (Boullier, 2000a,b), which characterize the class of languages recognizable in polynomial time. LMG and RCG are similar to GCFG, and share the same representation of abstract syntax. The drawbacks are that the concrete syntax is restricted to strings, and that the abstract and concrete syntax are defined simultaneously, making it difficult to use the same abstract syntax with several concrete.

2.3.9 Lexical functional grammar (LFG)

Lexical functional grammar (Bresnan and Kaplan, 1982) has a clean division between *c-structures* and *f-structures*; the former represents concrete syntax as trees, and the latter represents the "functional" (or abstract) structure as feature structures. Since the structures are clearly specified, it is difficult to implement several levels of abstraction; apart from that, LFG inherits all advantages of a clear separation between abstract and concrete syntax.

2.3.10 Dependency grammar (DG)

Dependency grammar consists of a large and diverse family of grammar formalisms, all sharing the assumption that syntactic structure consists of *lexical nodes* linked by binary relations called *dependencies* (see e.g. Mel'cuk, 1988; Hudson, 1990); meaning that DG do not have the idea of phrases. Because of the diversity it is difficult to make general comments regarding the separation of abstract and concrete syntax. There are formalisms (Hays, 1964; Gaifman, 1965) having no separation at all; and there are more recent formalisms (Debusmann et al., 2004) where the concrete syntax is not even limited to strings.

2.3.11 Head-driven phrase structure grammar (HPSG)

The syntactical structures in head-driven phrase structure grammar (Pollard and Sag, 1994) are *typed feature structures*, similar to but more powerful than records.

An HPSG grammar has several descriptional levels, for phonology, syntax, semantics etc., but the separation is not always that clear. The different levels all live together in one single feature structure, as different features. E.g. concrete strings reside under the feature PHON, whereas the syntactic structure is split into several parts. This makes it difficult to generalize HPSG to multilingual grammar, but also to perform compilation to remove intermediate levels.

Later work on linearization-based HPSG has separated the concrete word order from the feature structures (Reape, 1991; Daniels and Meurers, 2002), thus giving a better separation of concrete and abstract syntax.

2.4 Grammatical Framework

The abstract theory of Grammatical Framework (GF; Ranta, 2004a) is a version of dependent type theory, similar to LF (Harper et al., 1993), ALF (Magnusson and Nordström, 1994) and COQ (Coq, 1999). What GF adds to the logical framework is a possibility to define concrete syntax, that is, notations expressing formal concepts in user-readable ways. In this sense GF fits well into the idea of separating abstract and concrete syntax.

The development of GF started as a notation for type-theoretical grammar (Ranta, 1994), which uses Martin-Löf's type theory (1984) to express the semantics of natural language. The development of GF as an authoring system started as a plug-in to the proof editor ALF, to permit natural-language rendering of formal proofs (Hallgren and Ranta, 2000). The extension of the scope outside mathematics was made in the Multilingual Document Authoring project at Xerox (Dymetman et al., 2000). In continued work, GF has been used in areas like software specifications (Hähnle et al., 2002) and dialogue systems (Ranta and Cooper, 2004).

After the first publication (Mäenpää and Ranta, 1999), the expressiveness of the concrete syntax has developed into a functional programming language. As such it is similar to a restricted version of programming languages like Haskell (Peyton Jones, 2003) and ML (Milner et al., 1997). The language is restricted enough to be possible to compile into an efficient canonical format, but expressive enough to incorporate modern programming language constructs such as user-definable data types, higher-order functions, and a module system for defining grammatical resources.

2.4.1 Type theory

The abstract syntax of a GF grammar is defined by declaring a number of basic types (called *categories*), and a number of basic functions. A function is declared by giving its *typing*,

$$\texttt{fun}\ f\ \texttt{:}\ B_1\ \texttt{->}\ \cdots\ \texttt{->}\ B_\delta\ \texttt{->}\ A\texttt{;}$$

This declaration states that f is a function taking δ arguments of types $B_1, \ldots, B_\delta$, resulting in a term of type A. A function with no arguments ($\delta = 0$) is called a *constant*, and is simply declared as,

$$\texttt{fun}\ a\ \texttt{:}\ A\texttt{;}$$

In general we write $t : T$ if the term t is of type T. By applying the basic functions to each other, compound terms can be formed,

$$f\ b_1\ \cdots\ b_\delta\ \ \texttt{:}\ \ A$$

whenever each $b_i : B_i$ and f is declared as above.

2.4.2 Higher-order functions and dependent types

It is also possible to declare higher-order functions and dependent types in a GF grammar. A higher-order function is a function where some of the arguments are functions themselves; and a dependent type is declared to depend on (one or more) terms of other types.

These features are more thoroughly described by Ranta (2004a). Instead we concentrate on the very important subclass *context-free* GF, which does not contain higher-order functions or dependent types.

2.4.3 Concrete linearizations

The novel thing about GF with respect to a logical framework is that it adds a mapping from abstract terms to concrete *linearizations*. To define a concrete syntax of a grammar, we only need to do the following:

- For each basic category A defined in the abstract syntax, we define a corresponding *linearization type* $T = A^\circ$ by the declaration,

  $$\texttt{lincat}\ A\ =\ T\texttt{;}$$

- For each basic function f defined in the abstract syntax, we define a corresponding *linearization function* f°. If the original function f has a typing,

  $$f\ :\ B_1\ \texttt{->}\ \cdots\ \texttt{->}\ B_\delta\ \texttt{->}\ A$$

  then the linearization function f° has the typing,

  $$f^\circ\ :\ B_1^\circ\ \texttt{->}\ \cdots\ \texttt{->}\ B_\delta^\circ\ \texttt{->}\ A^\circ$$

- The linearization function f° is defined by a declaration,

  $$\texttt{lin}\ f\ x_1\ \ldots\ x_\delta\ =\ t\texttt{;}$$

  where $x_1 : B_1^\circ, \ldots, x_\delta : B_\delta^\circ$ are linearization variables, and $t : A^\circ$ is a linearization term in which the variables $x_1, \ldots, x_\delta$ can occur.

The linearization $[\![a]\!] : A^\circ$ of a term $a : A$ can now be defined as,

$$[\![a]\!]\ =\ f^\circ\ [\![b_1]\!]\ \cdots\ [\![b_\delta]\!]$$

whenever $a = f\ b_1\ \cdots\ b_\delta$ and $b_1 : B_1, \ldots, b_\delta : B_\delta$. The constraints on the linearization definitions assure that linearizations always have the correct type. Grammars are thus *compositional* in the sense that a linearization is a function of the argument linearizations, not of the arguments themselves.

2.4.4 The module system

GF has a module system, inspired by ideas from programming languages. There are three kinds of modules: abstract, concrete and resource modules.

- An abstract module defines an abstract theory, with categories and functions.
- A concrete module defines the concrete syntax of an abstract theory, by giving linearization types and linearization functions.
- A resource module defines parameter types, and operations that can be used as helper functions in concrete modules.

Modules can *extend* other modules by adding new definitions, thus opening the possibilities for modular grammar engineering. Another useful feature is that a concrete module (together with the corresponding abstract module) can be translated into a resource module. Since a resource module can be used by another concrete module, this makes it possible to perform grammar compositions as described in section 2.2.

2.5 Example of a multilingual GF grammar

In this chapter we give some examples of how to write grammars in GF, just to get a feeling for the possibilities.
We start with a simple context-free grammar for a fragment of English. It consists of the context-free categories S, VP, NP, D, N and V (standing for Sentence, Verb Phrase, Noun Phrase, Determiner, Noun and Verb respectively), and has the following rules;

S	→	NP VP		
VP	→	V NP		
NP	→	D N		
NP	→	N		
D	→	*"a"*		
D	→	*"many"*		
N	→	*"lion"*	|	*"lions"*
N	→	*"fish"*		
V	→	*"eats"*	|	*"eat"*

2.5.1 The abstract syntax

To get a corresponding GF grammar, we start by giving the abstract syntax. First we have to give a name to each of the CFG rules, and then we can introduce the type declarations,

```
fun s1  : NP -> VP -> S;
    vp1 : V  -> NP -> VP;
    np1 : D  -> N  -> NP;
    np2 : N  -> NP;
    d1  : D;
    d2  : D;
    n1  : N;
```

```
n2  : N;
v1  : V;
```

The predication function `s1` forms a sentence out of a noun phrase and a verb phrase. There are two ways of forming noun phrases; either by a determiner and a noun (*"a lion"*, *"many lions"*), or just a plural noun (*"lions"*). We assume that all verbs are transitive, so we only have the transitive verb phrase forming function `vp1`. The determiners `d1`, `d2` are singular and plural indefinites (*"a"* and *"many"*); `n1`, `n2` are the nouns *"lion"* and *"fish"*; and `v1` is the verb *"eat"*.

2.5.2 A simple concrete syntax for English

If we only want a GF grammar that is equivalent to the original CFG, we can assign each category the same linearization type `{s : Str}`, which is a record consisting of only one string.[2] This is done by the following declarations,

```
lincat S  = {s : Str};
       VP = {s : Str};
       NP = {s : Str};
       D  = {s : Str};
       N  = {s : Str};
       V  = {s : Str};
```

The concrete linearizations then look like follows,[3]

```
lin s1  x y = {s = x.s ++ y.s};
    vp1 x y = {s = x.s ++ y.s};
    np1 x y = {s = x.s ++ y.s};
    np2 x   = {s = x.s};
    d1      = {s = "a"};
    d2      = {s = "many"};
    n1      = {s = variants {"lion" ; "lions"}};
    n2      = {s = "fish"};
    v1      = {s = variants {"eats" ; "eat"}};
```

2.5.3 A concrete syntax that takes care of agreement

If we want to change the grammar so that it also takes care of agreement, we can do as follows. First we introduce the parameter type `Num` with the two values or constructors `Sg` and `Pl`,

```
param Num = Sg | Pl;
```

[2] The reason for using records and not just strings will become apparent later.

[3] The operation `variants`$\{a\,;\,b\}$ is the GF version of the non-deterministic choice $(a \mid b)$. The alert reader might notice that this makes the linearization functions non-deterministic.

Then we make a decision that nouns, verbs and verb phrases are *parameterized* over the number, whereas determiners and noun phrases have an *inherent* number.[4] A phrase parameterized over `Num` is stored as an inflection table `Num => Str`; and an inherited parameter is stored in a record together with the linearized string,

```
lincat S  = {s : Str};
       NP = {s : Str; n : Num};
       D  = {s : Str; n : Num};
       VP = {s : Num => Str};
       N  = {s : Num => Str};
       V  = {s : Num => Str};
```

To give the value of an inherent parameter, we simply form a record; and to access the value of an inherent parameter, we use record projection (just as we do to access the linearized string). An inflection table is formed by,

$$\texttt{table } \{p_1 \texttt{ => } t_1\texttt{; } \ldots\texttt{; } p_n \texttt{ => } t_n\}$$

where $p_1, \ldots, p_n$ are inflection *patterns*; and to apply an inflection table to a parameter, we use the *selection* operation (`!`). Returning to our example, we get the following concrete syntax for the English grammar with number agreement between the subject and the verb,[5]

```
lin s1  x y = {s = x.s ++ y.s!x.n};
    vp1 x y = {s = table {z => x.s!z ++ y.s}};
    np1 x y = {s = x.s ++ y.s!x.n; n = x.n};
    np2 x   = {s = x.s!Pl; n = Pl};
    d1      = {s = "a";    n = Sg};
    d2      = {s = "many"; n = Pl};
    n1      = {s = table {Sg => "lion"; Pl => "lions"}};
    n2      = {s = table {_  => "fish"}};
    v1      = {s = table {Sg => "eats"; Pl => "eat"}};
```

Note that the table in `vp1` has only one pattern matching any parameter, binding it to the variable `z` which can be used in the table body. Also note that the table in `n2` has an *anonymous* pattern, meaning that the value is *"fish"* regardless of the inflection parameter. Both uses are examples of that tables can sometimes be compacted.

Examples of phrases that are disallowed by this concrete syntax are *i*) noun phrases consisting of just a singular noun; *ii*) noun phrases where the determiner and noun do not agree; and *iii*) sentences where the subject noun phrase does not agree with the verb.

2.5.4 A concrete syntax for Swedish

Swedish has a more complex morphology than English; nouns do not only depend on number, they also have an inherent *gender* (neuter and uter) associated with them. Determiners, on the other hand, have

[4] GF has a functional perspective on linearizations, meaning that parameters have to be either parameterized over or inherited. The principal way of making parameters agree is to apply a parameterized inflection table to an inherited parameter.

[5] A notational convention in GF is that record projection (`.`) binds harder than table selection (`!`), which in turn binds harder that concatenation (`++`).

number as an inherent feature and depend on the gender of the noun. First we have to declare the corresponding parameter type `Gen`;

```
param Gen = Neu | Utr;
```

then the linearization types for nouns and determiners can be declared as,

```
lincat N = {s : Num => Str; g : Gen};
       D = {s : Gen => Str; n : Num};
```

Now we can define the linearizations for the determiners `d1`, `d2` and the nouns `n1`, `n2`;

```
lin d1 = {s = table {Utr => "en"; Neu => "ett"}; n = Sg};
    d2 = {s = table {_   => "många"};            n = Pl};
    n1 = {s = table {_  => "lejon"};               g = Neu};
    n2 = {s = table {Sg => "fisk"; Pl => "fiskar"; g = Utr};
```

Noun phrases, on the other hand, do not influence the inflection of verbs, which means that they can have simple linearization types $\texttt{NP}^\circ = \texttt{V}^\circ = \texttt{\{s : Str\}}$.[6] Now we are ready to give the linearization functions for noun phrase formation,

```
lin np1 x y = {s = x.s!y.g ++ y.s!x.n};
    np2 x   = {s = x.s!Pl};
```

Finally, the word order of sentences depend on the context of the sentence. There are three different word orders (direct, indirect and subordinate), introducing yet another parameter type `Order`,

```
param Order = Dir | Indir | Sub;
```

The indirect order (used e.g. in questions) puts the subject noun phrase inside the verb phrase. The way to solve this in GF is to use discontinuous verb phrases. The linearization of sentences and verb phrases will be,[7]

```
lincat S  = {s  : Order => Str};
       VP = {s1 : Str; s2 : Str};

lin s1  x y = {s = table {Indir => y.s1 ++ x.s  ++ y.s2;
                          _    => x.s  ++ y.s1 ++ y.s2}};
    vp1 x y = {s1 = x.s; s2 = y.s};
```

A fourth possible word order could be *topicalized*, which is used when the object is put in front of the sentence for focusing purposes; e.g. the sentence *"fiskar äter många lejon"* (fish eat many lions) has the preferred reading (it is fish that many lion eat). This can be solved by adding a new constructor `Top` to the type `Order`, and a new row to the table in the linearization definition of `s1` above,

```
lin s1 x y = {s = table {Top => y.s2 ++ x.s ++ y.s1; ...}};
```

[6] This is a simplification; when adding pronouns and/or adjectives, Swedish noun phrases can get quite complex.

[7] The difference between direct and subordinate word order only shows up in the presence of negation, which we don't have in this example.

Chapter 3

Extending multilinguality to multimodality

3.1 Parallel multimodality

Parallel multimodality is a straightforward instance of multilinguality. It means that the concrete syntaxes associated with an abstract syntax are not just different natural languages, but different representation modalities, encoded by language-like notations such as graphic representation formalisms. An example of parallel multimodality is given below when a route is described, in parallel, by speech and by a line drawn on a map (Section 4.6.2). Both descriptions convey the full information alone, without support from the other.

This raises the dialogue management issue of whether all information should be presented in all modalities. For example, in the implementation described below all stops are indicated on the graphical presentation of a route whereas in the natural language presentation only stops where the user must change are presented. Because GF permits the suppression of information in concrete syntax, this issue can be treated on the level of grammar instead of dialogue management.

3.2 Integrated multimodality

Integrated multimodality means that one concrete syntax representation is a combination of modalities. For instance, the spoken utterance “I want to go from here to here” can be combined with two pointing gestures corresponding to the two “here”s. It is the two modalities in combination that convey the full information: the utterance alone or the clicks alone are not enough.

How to define integrated multimodality with a grammar is less obvious than parallel multimodality. The GF solution we will present in Section 4.4 makes essential use of records, and not just strings, as outcomes of linearization. In brief, different modality “channels” are stored in different fields of a record, and it is the combination of the different fields that is sent to the dialogue system parser.

Chapter 4

Proof of concept implementation

4.1 Overview

We have implemented a multimodal route planning system for public transport networks. The example system uses the Göteborg tram/bus network, but it can easily be adapted to other networks.

The system uses multimodal grammars for user and system utterances. The user modalities are speech and map clicks, and the system modalities are speech and drawings on the map. Input and output in all the modalities are handled by multilingual, multimodal grammars. For brevity and clarity, the following sections show English concrete syntax exclusively. For every concrete English module shown below, the application also contains a corresponding module for Swedish concrete syntax.

4.2 Grammar overview

The user and system grammars are split up into a number of modules in order to make reuse and modification simpler.

The query and answer grammar modules are shown in figures 4.1 and 4.2, respectively. The following sections show the details of the grammar modules.

4.3 Transport network grammar

The transport network is represented by a set of modules which are used in both the query and answer grammars. Since the transport network is described in a separate set of modules, the Göteborg transport network may be replaced easily.

4.3.1 Generic transport network abstract syntax

The interface for transport network grammars is very simple. Such a grammar simply exports a number of constants in the Stop category:

```
abstract Transport = {
```

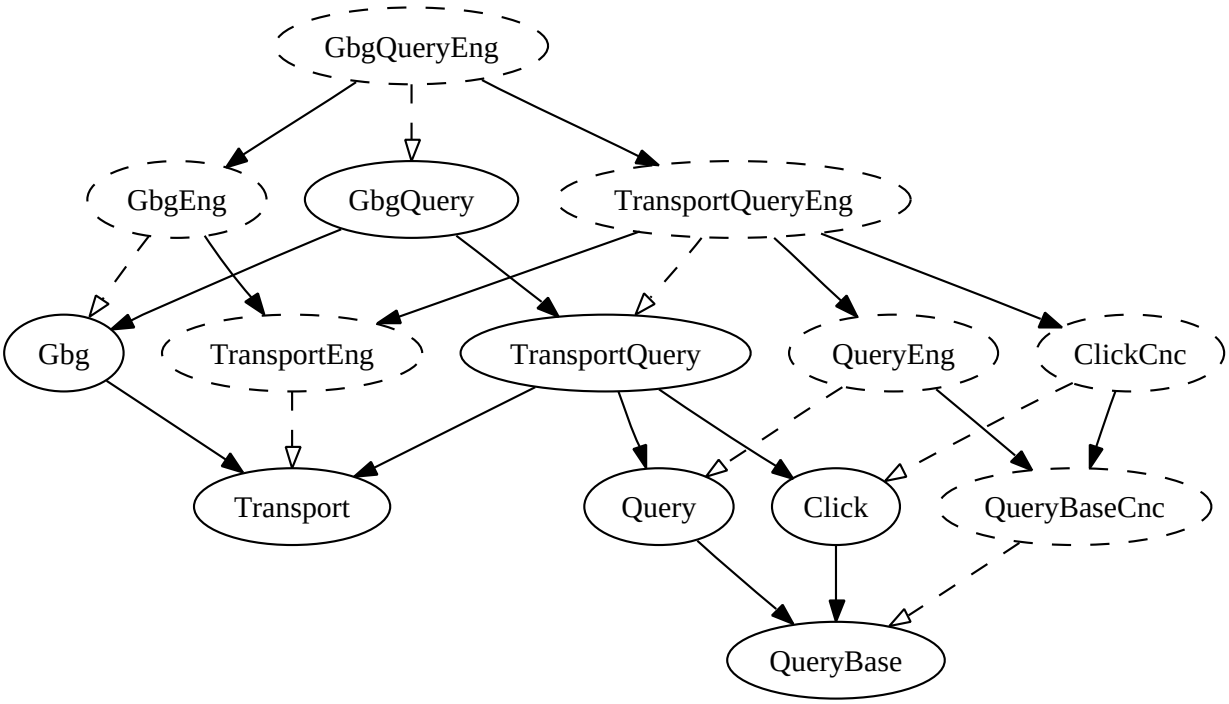


Figure 4.1: Query grammar modules

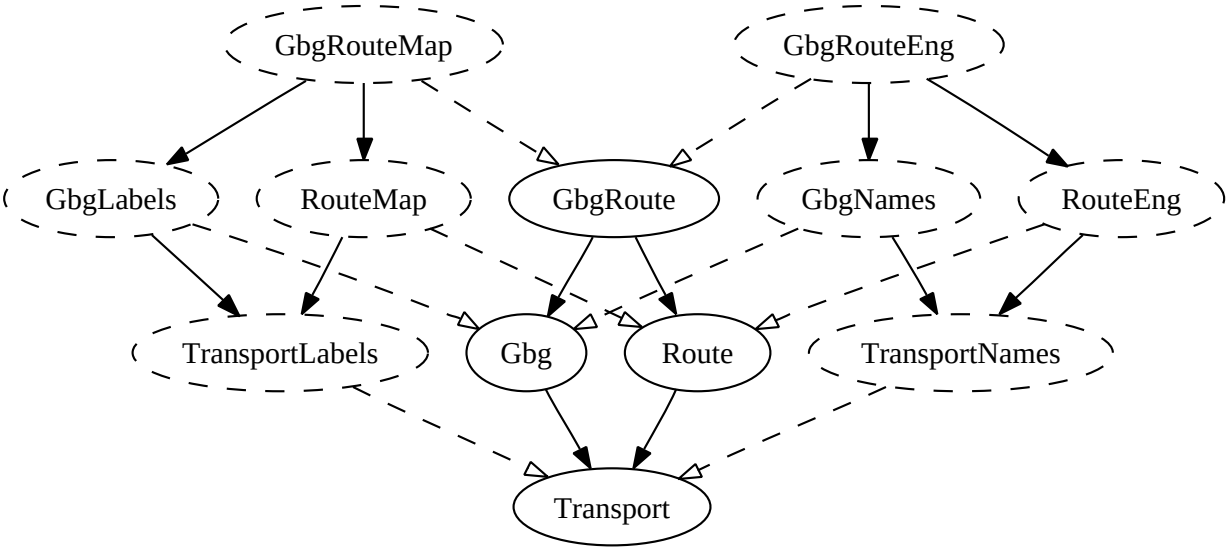


Figure 4.2: Answer grammar modules

```
    cat
      Stop ;
}
```

4.3.2 Generic transport network concrete syntax

The English concrete syntax is equally simple. Languages which inflect proper nouns might need a more complex linearization type for stops.

```
concrete TransportEng of Transport = {
    lincat
      Stop = { s : Str } ;
}
```

4.3.3 Göteborg abstract syntax

The abstract syntax for a given transport network lists the stops.

```
abstract Gbg = Transport ** {
    fun Angered : Stop ;
    fun AxelDahlstromsTorg : Stop ;
    fun Bergsjon : Stop ;
    fun Biskopsgarden : Stop ;
    ...
}
```

4.3.4 Göteborg concrete syntaxes

Since names are not normally translated between languages, they introduce a problem for speech recognition. We would like the map to show the names of tram/bus stops in their native orthography. This is done with one concrete syntax:

```
concrete GbgNames of Gbg = TransportNames ** {
    lin Angered = { s = ["Angered"] } ;
    lin AxelDahlstromsTorg = { s = ["Axel Dahlströms torg"] } ;
    lin Bergsjon = { s = ["Bergsjön"] } ;
    lin Biskopsgarden = { s = ["Biskopsgården"] } ;
    ...
}
```

However, speech recognizers often do not support characters not used in the language which they recognize. Furthermore, some recognizers, such as Nuance, do not allow capitals in the recognized text. Therefore, we introduce a different concrete syntax for stop names for each language. In the English syntax, accented characters have the diacritics removed and all letters are converted to lower case.

```
concrete GbgEng of Gbg = TransportEng ** {
    lin Angered = { s = ["angered"] } ;
    lin AxelDahlstromsTorg = { s = ["axel dahlstroms torg"] } ;
    lin Bergsjon = { s = ["bergsjon"] } ;
    lin Biskopsgarden = { s = ["biskopsgarden"] } ;
    ...
}
```

Since stop names are also used in the click and drawing modalities, we also need an easily machine readable and writable syntax. This is achieved by removing spaces and diacritics from the stop names:

```
concrete GbgLabels of Gbg = TransportLabels ** {
    lin Angered = { s = ["Angered"] } ;
    lin AxelDahlstromsTorg = { s = ["AxelDahlstromsTorg"] } ;
    lin Bergsjon = { s = ["Bergsjon"] } ;
    lin Biskopsgarden = { s = ["Biskopsgarden"] } ;
    ...
}
```

4.4 Multimodal input grammars

User input is done with integrated speech and click modalities. The user may use speech only, or speech combined with clicks on the map. Clicks are expected when the user makes a query containing “here” (though “here” might also be used without a click, see Section 4.4.4).

Clicks are represented as a list of places that the click might refer to. Normally this is a singleton list containing a single bus/tram stop, but some stops might be close enough that a click could refer to more than one stop. The set might also be empty if the click was not close to any stop.

In the concrete syntax, the click data is appended to the speech input to give the parser a single string to parse. These are some examples using the English concrete syntax:

- “i want to go from brunnsparken to vasaplatsen;”
- “i want to go from vasaplatsen to here; [Chalmers]”
- “i want to go from here to here; [Chalmers] [Saltholmen]”

4.4.1 Common declarations

The QueryBase module contains declarations common to all input modalities:

```
abstract QueryBase = {
    cat
      Query ;  -- sequentialized input representation
      Input ;  -- user input: parallel text and clicks
```

```
      Click ;  -- map clicks
    fun
      QInput : Input -> Query ; -- sequentialize user input
}
```

QueryBase has a single concrete syntax since it is language neutral:

```
concrete QueryBaseCnc of QueryBase = {
    lincat
      Query = { s : Str } ;
      Input = { s1 : Str ; s2 : Str } ;
      Click = { s : Str } ;
    lin
      QInput i = { s = i.s1 ++ ";" ++ i.s2 } ;
}
```

4.4.2 Click modality

Clicks are represented by a list of stops that the click might refer to:

```
abstract Click = QueryBase ** {
    cat
      StopList ; -- a list of stop names
    fun
      CStops : StopList -> Click ;
      NoStop : StopList ;
      OneStop : String -> StopList ;
      ManyStops : String -> StopList -> StopList ;
}
```

The same concrete syntax is used for clicks in all languages:

```
concrete ClickCnc of Click = QueryBaseCnc ** {
    lincat
      StopList = { s : Str } ;
    lin
      CStops xs = { s = "[" ++ xs.s ++ "]" } ;
      NoStop = { s = "" } ;
      OneStop x = { s = x.s } ;
      ManyStops x xs = { s = x.s ++ "," ++ xs.s } ;
}
```

4.4.3 Speech modality

The Query module adds basic user queries and a way to use a click to indicate a place:

```
abstract Query = QueryBase ** {
    cat
      Place ; -- any way to identify a place
    fun
      GoFromTo : Place -> Place -> Input ;
      GoToFrom : Place -> Place -> Input ;
      PClick   : Click -> Place ;  -- "here" together with a click
}
```

The corresponding English concrete syntax is:

```
concrete QueryEng of Query = QueryBaseCnc ** {
    lincat
      -- speech and click representations of a place
      Place = {s1 : Str; s2 : Str} ;
    lin
      GoFromTo x y = {
        s1 = ["i want to go from"] ++ x.s1 ++ "to" ++ y.s1 ;
        s2 = x.s2 ++ y.s2
      } ;
      GoToFrom x y = {
        s1 = ["i want to go to"] ++ x.s1 ++ "from" ++ y.s1 ;
        s2 = x.s2 ++ y.s2
      } ;
      PClick c = { s1 = "here" ; s2 = c.s } ;
}
```

4.4.4 Indexicality

To refer to her current location, the user can use “here” without a click, or omit either origin or destination. The system is assumed to know where the user is located. Examples in English concrete syntax:

- “i want to go from here to centralstationen;”
- “i want to go to valand;”
- “i want to come from brunnsparken;”

These are the abstract syntax declarations for this feature (in the Query module):

```
    fun
      -- indexical "here", without a click
      PHere    : Place ;
      -- user input "want to come from a" (to where I am now)
      ComeFrom : Place -> Input ;
      -- user input "want to go to a" (from where I am now)
      GoTo     : Place -> Input ;
```

The English concrete syntax for this is (in the QueryEng module):

```
lin
  PHere = { s1 = "here" ; s2 = [] } ;
  ComeFrom x = {
    s1 = ["i want to come from"] ++ x.s1 ;
    s2 = x.s2
  } ;
  GoTo x = {
    s1 = ["i want to go to"] ++ x.s1 ;
    s2 = x.s2
  } ;
```

4.5 Ambiguity

Some strings may be parsed in more than one way. Since “here” may be used with or without a click, input with two occurrences of “here” and only one click are ambiguous:

- “I want to go from here to here; [Valand]”

A query might also be ambiguous even if it can be parsed unambiguously, since one click can correspond to multiple stops:

- “I want go go from Chalmers to here; [Klareberg,Tagene]”

The current application fails to produce any output for ambiguous queries. A real system should handle this through dialogue management.

4.6 Multimodal output

The system’s answers to the user’s queries are presented with speech and drawings on the map. This is an example of parallel multimodality as the speech and the map drawings are independent.

The information presented in the two modalities is however not identical as the spoken output only contains information about when to change trams/buses. The map output shows the entire path, including intermediate stops.

Parallel multimodality is from the system’s point of view just a form of multilinguality. The abstract syntax representation of the system’s answers has one concrete syntax for the drawing modality, and one for each natural language. The only difference between the natural language syntaxes and the drawing one is that the latter is a formal language rather than a natural one.

4.6.1 Abstract syntax

The abstract syntax for answers (routes) contains the information needed by all the concrete syntaxes. All concrete syntaxes might not use all of the information. A route is a non-empty list of legs, and a leg consists of a line and a list of at least two stops.

```
abstract Route = Transport ** {
    cat
      Route; -- route description
      Leg;   -- route segment on a single line
      Line;  -- bus/tram line
      Stops; -- list of at least two stops
    fun
      Then : Leg -> Route -> Route ;      -- leg followed by a route
      OneLeg : Leg -> Route ;             -- single leg
      LineLeg : Line -> Stops -> Leg ;    -- leg on a line
      NamedLine : String -> Line ;        -- line labelled by a string
      ConsStop : Stop -> Stops -> Stops ; -- stop followed by some stops
      TwoStops : Stop -> Stop -> Stops ;  -- last two stops
}
```

4.6.2 Map drawing concrete syntax

The map drawing language contains sequences of labelled edges to be drawn on the map. The following string:

- “drawEdge (6, [Chalmers, Vasaplatsen]); drawEdge (2, [Vasaplatsen, Gronsakstorget, Brunnsparken]);”

is an example of a string in the map drawing language described by this concrete syntax:

```
concrete RouteMap of Route = TransportLabels ** {
    lincat
      Route = { s : Str } ;
      Leg = { s : Str } ;
      Line = { s : Str } ;
      Stops = { s : Str } ;
    lin
      Then l r = { s = l.s ++ ";" ++ r.s } ;
      OneLeg l = { s = l.s ++ ";" } ;
      LineLeg l ss =
        { s = "drawEdge" ++ "(" ++ l.s ++ ","
               ++ "[" ++ ss.s ++ "]" ++ ")" } ;
      NamedLine n = { s = n.s } ;
      ConsStop s ss = { s = s.s ++ "," ++ ss.s } ;
      TwoStops s1 s2 = { s = s1.s ++ "," ++ s2.s } ;
}
```

4.6.3 English concrete syntax

In the English concrete syntax we wish to list only the first and last stop of each leg of the route.

```
concrete RouteEng of Route = TransportNames ** {
    lincat
      Route = { s : Str } ;
      Leg = { s : Str } ;
      Line = { s : Str } ;
      -- a list of stops is linearized to its first and last stop
      Stops = { start : Str; end : Str } ;
    lin
      Then l r = { s = l.s ++ "." ++ r.s } ;
      OneLeg l = { s = l.s ++ "." } ;
      LineLeg l ss =
        { s = "Take" ++ l.s ++ "from" ++ ss.start ++ "to" ++ ss.end } ;
      NamedLine n = { s = n.s } ;
      ConsStop s ss = { start = s.s; end = ss.end } ;
      TwoStops s1 s2 = { start = s1.s; end = s2.s } ;
}
```

4.7 Example interaction

The user says “i want to go from chalmers to here” and clicks on Frihamnen. This is represented by the input string:

- “i want to go from chalmers to here; [Frihamnen]”

The parser produces this abstract syntax representation:

```
QInput (GoFromTo (PStop Chalmers)
                 (PClick (CStops (OneStop "Frihamnen"))))
```

The system responds with this answer:

```
Then (LineLeg (NamedLine "6") (TwoStops Chalmers Vasaplatsen))
 (Then (LineLeg (NamedLine "2")
       (ConsStop Vasaplatsen
                 (TwoStops Gronsakstorget Brunnsparken)))
  (OneLeg (LineLeg (NamedLine "5")
                   (ConsStop Brunnsparken
                             (TwoStops LillaBommen Frihamnen)))))
```

This is linearized to this speech output:

- “Take 6 from Chalmers to Vasaplatsen. Take 2 from Vasaplatsen to Brunnsparken. Take 5 from Brunnsparken to Frihamnen.”

And to these drawing instructions:

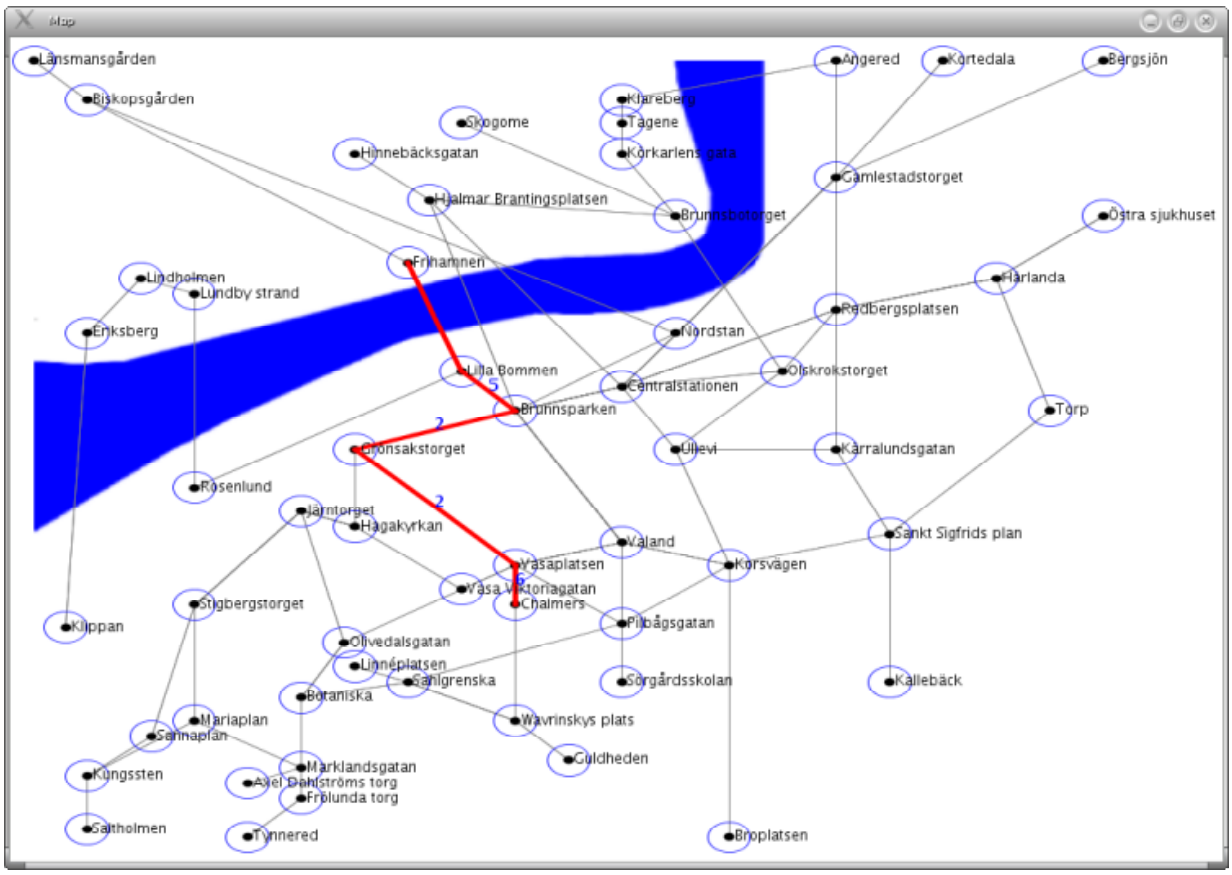


Figure 4.3: The map showing the path from Chalmers to Frihamnen.

- “drawEdge (6, [Chalmers, Vasaplatsen]); drawEdge (2, [Vasaplatsen, Gronsakstorget, Brunnsparken]); drawEdge (5, [Brunnsparken, LillaBommen, Frihamnen]);”

The map with this output is shown in figure 4.3.

4.8 Multilinguality

Currently, speech input and output in English and Swedish are implemented. The dialogue system itself accepts input in either language, but speech recognizers can often only handle a single language at a time. System output is linearized using the same language as the speech input was in.
Adding support for a new language requires writing concrete syntaxes for the user and system grammars.

4.9 Component overview

The application consists of the following agents:

- Speech recognizer - Nuance through OAA using NuanceWrapper

- Clickable map + Path drawing - An OAA agent written in Java
- Parser + Linearizer (multilingual and multimodal) - Java GF interpreter
- Shortest path finder - OAA agent written in Java
- Speech synthesis - FreeTTS over OAA using FreeTTSAgent

The demo application (Tramdemo), NuanceWrapper, the Java GF interpreter and FreeTTSAgent are all available from the Göteborg TALK software library at `http://www.ling.gu.se/projekt/talk/software/`.

4.10 Limitations

There is no dialogue management in this version. Queries that do not have exactly one interpretation are not answered. The purpose of this application is to demonstrate use of multimodal and multilingual grammars. Adding dialogue management should be orthogonal to this.

There is no handling of departure times, only time between stops. Adding support for this would be relatively straightforward, but would require some effort to support time expressions. The shortest-path algorithm would also need to be changed to take waiting times into account.

The current system is not usable for practical route planning since the Göteborg public transit network description is incomplete and out of date.

Chapter 5

Conclusion

GF provides a solution to the problems named in the introduction to this deliverable. Abstract syntax can be used to characterise the linguistic functionality of a system in an abstract language and modality independent way. The system forces the programmer to define concrete syntaxes which completely cover the abstract syntax. In this way, the system forces the programmer to keep all the concrete syntaxes in sync. In addition, since GF is oriented towards creating grammars from other grammars, our philosophy is that it should not be necessary for a grammar writer to have to create by hand any equivalent grammars in different formats. For example, if the grammar for the speech recogniser is to be the same as that used for interaction with dialogue management but the grammars are needed in different formats, then there should be a compiler which takes the grammar from one format to the other. Thus, for example, we have a compiler which converts a GF grammar to Nuance's format for speech recognition grammars.

Another reason for using GF grammars has to do with the use of resource grammars and cascades of levels of representation as described in section 2.2. This allows for the hiding of grammatical detail from language and the precise implementation of modal interaction for other modalities. This enables the dialogue system developer to reuse previous grammar or modal interaction implementations without herself having to reprogram the details for each new dialogue system. Thus the dialogue engineer need not be a grammar engineer or an expert in multimodal interfaces.

5.1 Future work

The proof of concept dialogue system presented in section 4 is the first complete GF-based application built within the TALK project. Another project is in progress within the smart house domain, where a set of extensive GF grammars has been written for programming a video recorder in English and Swedish. Of particular interest is the use of the module system to manage the plurality of different but related devices. For instance, MP3 players have many shared functionalities with video recorders, but even more with CD players. The challenge is to avoid the duplication of grammar rules, thereby also giving the user a feeling of uniformity which makes it easier to learn to control new devices.

To make GF grammar writing more accessible to authors of dialogue systems, we will continue the work on resource grammars and their documentation. In particular, we are developing a resource grammar API giving easy access to constructs that are needed in dialogue systems.

For those projects that are not using GF grammars directly we are developing tools that generate corpora.

The idea is to generate a corpus by “bootstrapping”: to use a small corpus as a filter that extracts a domain grammar from a resource grammar, an then generate a larger corpus from the domain grammar. This corpus can then be used e.g. as data for a statistic language model. By the use of dependent types (Section 2.4.2), it is possible to prevent the generation of expressions that although linguistically correct are semantically nonsense and would hence never occur in a real corpus.

Bibliography

Aho, A. (1968). Indexed grammars—an extension to context-free grammars. *Journal of the ACM*, 15:647–671.

Ajdukiewicz, K. (1935). Die syntaktische Konnexität. *Studia Philosophica*, 1:1–27.

Bar-Hillel, Y. (1953). A quasi-arithmetical notation for syntactic description. *Language*, 29:47–58.

Boullier, P. (2000a). A cubic-time extension of context-free grammars. *Grammars*, 3:111–131.

Boullier, P. (2000b). Range concatenation grammars. In *6th International Workshop on Parsing Technologies*, pages 53–64, Trento, Italy.

Bresnan, J. and Kaplan, R. (1982). Lexical-functional grammar: A formal system for grammatical representation. In Bresnan, J., editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, Cambridge, MA.

Chomsky, N. (1957). *Syntactic Structures*. Mouton, The Hague.

Chomsky, N. (1965). *Aspects of the Theory of Syntax*. MIT Press, Cambridge, MA.

Coq (1999). *The Coq Proof Assistant Reference Manual*. The Coq Development Team. Available at `http://pauillac.inria.fr/coq/`

Curry, H. B. (1963). Some logical aspects of grammatical structure. In Jacobson, R., editor, *Structure of Language and its Mathematical Aspects: Proceedings of the 12th Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society.

Daniels, M. and Meurers, D. (2002). Improving the efficiency of parsing with discontinuous constituents. In *NLULP-02: 7th International Workshop on Natural Language Understanding and Logic Programming*, Copenhagen, Denmark.

Debusmann, R., Duchier, D., and Kruijff, G.-J. M. (2004). Extensible dependency grammar: A new methodology. In *COLING 2004 Workshop on Recent Advances in Dependency Grammar*.

Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., and Levy, J. J. (1975). A structure-oriented program editor: a first step towards computer assisted programming. In *International Computing Symposium (ICS'75)*.

Dymetman, M., Lux, V., and Ranta, A. (2000). XML and multilingual document authoring: Convergent trends. In *COLING*, pages 243–249, Saarbrücken, Germany.

Gaifman, H. (1965). Dependency systems and phrase-structure systems. *Information and Control*, 8:304–337.

Gazdar, G. (1987). Applicability of indexed grammars to natural languages. In Reyle, U. and Rohrer, C., editors, *Natural Language Parsing and Linguistic Theories*, pages 69–94. D. Reidel Publishing Company.

Groenink, A. (1997a). Mild context-sensitivity and tuple-based generalizations of context-free grammar. *Linguistics and Philosophy*, 20:607–636.

Groenink, A. (1997b). *Surface without Structure — Word order and tractability issues in natural language analysis*. PhD thesis, Utrecht University.

Hallgren, T. and Ranta, A. (2000). An extensible proof text editor. In Parigot, M. and Voronkov, A., editors, *LPAR-2000*, volume 1955 of *LNCS/LNAI*, pages 70–84. Springer.

Harper, R., Honsell, F., and Plotkin, G. (1993). A framework for defining logics. *Journal of the ACM*, 40(1):143–184.

Hays, D. (1964). Dependency theory: A formalism and some observations. *Language*, 40:511–525.

Hudson, R. (1990). *English Word Grammar*. Blackwell.

Hähnle, R., Johannisson, K., and Ranta, A. (2002). An authoring tool for informal and formal requirements specifications. In Kutsche, R.-D. and Weber, H., editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 233–248. Springer.

Joshi, A. and Schabes, Y. (1997). Tree-adjoining grammars. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages. Vol 3: Beyond Words*, chapter 2, pages 69–123. Springer-Verlag, Berlin/Heidelberg/New York.

Joshi, A. K., Levy, L. S., and Takahashi, M. (1975). Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1):136–163.

Karttunen, L., Chanod, J.-P., Grefenstette, G., and Schiller, A. (1996). Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Khegai, J., Nordström, B., and Ranta, A. (2003). Multilingual syntax editing in GF. In Gelbukh, A., editor, *CICLing-2003: Intelligent Text Processing and Computational Linguistics*, LNCS 2588, pages 453–464. Springer.

Knight, S., Gorrell, G., Rayner, M., Koeling, R., and Lewin, I. (2001). Comparing grammar-based and robust approaches to speech understanding: a case study. In *Eurospeech 2001: Proceedings of the 7th European Conference on speech communication and technology*, pages 1779–1782.

Lager, T. and Kronlid, F. (2004). The Current platform: Building conversational agents in Oz. In *2nd International Mozart/Oz Conference*.

Lambek, J. (1958). The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170.

Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, 9(3):157–166.

Ljunglöf, P. (2004). *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Göteborg University, Gothenburg, Sweden.

Magnusson, L. and Nordström, B. (1994). The ALF proof editor and its proof engine. In *Types for Proofs and Program*, volume 806 of *LNCS*, pages 213–237. Springer.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Bibliopolis, Napoli.

McCarthy, J. (1963). Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, Amsterdam. North-Holland.

Mel'cuk, I. (1988). *Dependency Syntax: Theory and Practice*. State University of New York Press.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The Definition of Standard ML – Revised*. MIT Press, Cambridge, MA.

Mohri, M. (1997). Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–312.

Montague, R. (1974). *Formal Philosophy*. Yale University Press, New Haven. Collected papers edited by R. Thomason.

Morrill, G. (1994). *Type Logical Grammar: Categorial Logic of Signs*. Dordrecht.

Mäenpää, P. and Ranta, A. (1999). The type theory and type checker of GF. In *PLI-1999 workshop on Logical Frameworks and Meta-languages*, Paris, France.

Peyton Jones, S. (2003). *Haskell 98 Language and Libraries*. Cambridge University Press, New York.

Pollard, C. (1984). *Generalised Phrase Structure Grammars, Head Grammars and Natural Language*. PhD thesis, Stanford University.

Pollard, C. and Sag, I. (1994). *Head-Driven Phrase Structure Grammar*. University of Chicago Press.

Ranta, A. (1994). *Type-Theoretical Grammar*. Oxford University Press.

Ranta, A. (2004a). Grammatical Framework, a type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189.

Ranta, A. (2004b). Modular grammar engineering in GF. Submitted.

Ranta, A. and Cooper, R. (2004). Dialogue systems as proof editors. *Journal of Logic, Language and Information*, 13(2):225–240.

Reape, M. (1991). Parsing bounded discontinuous constituents: Generalisations of some common algorithms. In Reape, M., editor, *Word Order in Germanic and Parsing*, pages 41–70. Centre for Cognitive Science, Edinburgh.

Seki, H., Matsumara, T., Fujii, M., and Kasami, T. (1991). On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229.

Steedman, M. (1985). Dependency and coordination in the grammar of Dutch and English. *Language*, 61:523–568.

Steedman, M. (1986). Combinators and grammars. In Oehrle, R., Bach, E., and Wheeler, D., editors, *Categorial Grammars and Natural Language Structures*, pages 417–442. Foris, Dordrecht.

Teitelbaum, T. and Reps, T. (1981). The Cornell Program Synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573.

Vijay-Shanker, K., Weir, D., and Joshi, A. (1987). Characterizing structural descriptions produced by various grammatical formalisms. In *25th Meeting of the Association for Computational Linguistics*.