The TALK Grammar Library: an Integration of GF with TrindiKit

Peter Ljunglöf (editor) Björn Bringert Robin Cooper
Ann-Charlotte Forslund David Hjelm Rebecca Jonsson
Staffan Larsson Aarne Ranta

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable 1.1

01/07/05

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	Jun 05
Actual date of delivery	01/07/05
Deliverable number	1.1
Deliverable title	The TALK Grammar Library: an Integration of GF with TrindiKit
Type	Report
Status & version	Public Final
Number of pages	42 (excluding front matter)
Contributing WP	1
WP/Task responsible	UGOT
Other contributors	
Author(s)	Peter Ljunglöf (editor), Björn Bringert, Robin Cooper, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonsson, Staffan Larsson and Aarne Ranta
EC Project Officer	Evangelia Markidou
Keywords	grammar, multilingual, dialogue systems, Grammatical Framework, TrindiKit, GoDiS

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

Contents

1 Introduction **1**

1.1 Why integrate GF and TrindiKit? 1

1.2 GoDiS 2

1.2.1 Mismatch between recognition and interpretation 2

1.2.2 Mismatch between interpretation and generation 3

1.3 Solutions to the mismatch problems 4

1.4 Outline of this deliverable 4

2 Integration of GF and TrindiKit **6**

2.1 Grammatical Framework in the dialogue domain 6

2.1.1 Abstract and concrete syntax 6

2.1.2 Multilinguality and resource grammars 6

2.1.3 Embedded grammars 7

2.1.4 Connecting GF to TrindiKit/GoDiS 7

2.2 The Embedded GF Interpreter 7

2.2.1 Parsing 8

2.2.2 Linearization 9

2.2.3 Translation 9

2.2.4 Java API 10

2.2.5 Typed Abstract Syntax Trees 10

2.2.6 OAA Agent 11

2.3 Representing TrindiKit/GoDiS dialogue moves in GF 11

2.3.1 Dialogue Moves in GoDiS 11

2.3.2 Categories in GF 14

2.3.3 Using dependent types 14

2.4 Extracting speech recognition grammars from application grammars 15

2.4.1 Speech Recognition Grammar Formats 16

2.4.2 Implementation 16

2.4.3 Evaluation 16

2.4.4 Related work 17

3 Resource grammars and grammar engineering 18
3.1 Library-based grammar engineering 18
3.2 GF support for modules and libraries 18
3.3 Grammar engineering 19
3.3.1 Resource grammar libraries 20

4 The unimodal GoDiS grammar library 21
4.1 The library file structure 21
4.2 The in-home abstract resource (API) 21
4.3 The core grammar for the GoDiS Dialogue Manager 22
4.3.1 The abstract syntax 22
4.3.2 The concrete English GoDiS core grammar 23
4.3.3 The concrete Swedish GoDiS core grammar 24
4.3.4 The concrete GoDiS dialogue move representation core grammar 25
4.4 The specific domain grammars 25
4.4.1 DJ GoDiS 25
4.4.2 AgendaTalk 28
4.4.3 GoDiS deLux 29
4.4.4 Video GoDiS 30

5 Conclusions 32
5.1 Contents of the associated software 32
5.1.1 Grammar statistics 33
5.2 Concluding remarks 33

A Downloading and installation instructions 34
A.1 Downloading instructions 34
A.1.1 Contents 34
A.2 Installation instructions 35
A.2.1 System requirements 35
A.2.2 Installation and usage 35
A.3 Testing the Unimodal GF Grammar Library 36
A.3.1 Testing the grammars within GF 36
A.3.2 Using the Embedded GF Interpreter 38
A.3.3 Producing speech recognition grammars 39

Chapter 1

Introduction

1.1 Why integrate GF and TrindiKit?

The dialogue toolkit TrindiKit and the generic dialogue system GoDiS built within TrindiKit do not provide any specific support for grammars. In previous systems built with these tools we have used simple phrase spotting to relate user and system utterances to dialogue moves. The correlations are expressed in GoDiS lexicons. For speech recognition we have used Nuance's grammar formalism. In what follows we explain why this situation needs to be improved and how the Grammatical Framework (GF) provides us with an engineering approach to grammar which is very well suited to the needs of maintaining a number of small related grammars as are needed by our approach to dialogue system development.

Recognition, interpretation and generation

When designing a spoken dialogue system, there are several different options for how to handle speech recognition, syntactic/semantic interpretation, and generation.

Speech recognition There are two main alternatives, either to use a statistical language model (SLM), or to use a grammar-based language model. The main problem with using a SLM is that there must be a corpus to extract the statistical data from, and it is often not feasible to create a corpus just for a single dialogue system. The main problem with using a grammar-based model, however, is lack of robustness.

Syntactic interpretation The simplest alternative is to use word or phrase spotting. This makes the system robust, but gives rise to problems when trying to handle complex utterances.

Natural language generation Template-based generation is often simple, and shares the same difficulties of handling complex utterances with word/phrase spotting. A more complex alternative is to use a grammar for generation purposes.

These options for recognition, interpretation and generation can be combined in various ways. In most commercial working dialogue systems the recognition is grammar-based since there is no corpus fitting the intended domain, but there are no grammars involved for interpretation and generation. In these systems, the dialogue designer has to manually ensure that the recognition, interpretation and generation modules are all up-to-date and in sync with each other.

Separating specific knowledge from general knowledge

Many dialogue systems are built more or less from scratch and do not fully separate application-specific knowledge from general and reusable knowledge in the way that we have attempted to do – for example, by separating dialogue strategies such as question accommodation from the particular implementation of a system for a given domain.

1.2 GoDiS

GoDiS is a dialogue system built using TrindiKit and based on the information state approach. It implements a theory of Issue-based Dialogue Management. Some of the goals of GoDiS are

- providing a domain-independent theory covering several dialogue genres
- modularity and reusability of system components
- enabling rapid prototyping of new applications
- focusing on dialogue management

These goals have been achieved on the level of dialogue management, but not for natural language processing, i.e. speech recognition, syntactic/semantic interpretation and natural language generation.

Recognition, interpretation and generation in GoDiS

GoDiS started out as text-based system, using phrase spotting for syntactic interpretation, and templates for natural language generation. Subsequently a grammar-based speech recognition component was added. The speech recognition grammar has to be written and maintained by hand.

1.2.1 Mismatch between recognition and interpretation

If the speech recognition grammar is out of sync with the phrase spotting interpretation module, correctly recognised speech can be misinterpreted. This in turn leads to strange and unexpected behaviour, i.e. unexpected interactions with the dialogue manager.

Example from the Video GoDiS system

```
S> Välkommen till videon! För att få hjälp, säg hjälp.
   Welcome to the video! For help, say help.
S> Låt oss se. Vad kan jag hjälpa dig med?
   Let’s see. What can I do for you?
U> jag vill spela in ett program (0.6)
   i want to record a/one program
S> Vill du lägga till en inspelning?
   Do you want to add a recording?
U> ja (0.98)
```

```
   yes
S> Låt oss se. Menar du från noll ett eller till noll ett?
   Let’s see. Do you mean from zero one or to zero one?
```

The utterance “jag vill spela in ett program” is covered by the speech recognition grammar, but not by the phrase spotting lexicon. However “spela in” (record) and “ett” (a/one) are recognized as phrases by the interpretation module.
The correct interpretation should be:

“jag vill spela in ett program” ⇒ `request(add_ rec_ job)`

but the actual interpretation is:

“spela in” ⇒ `request(add_ rec_ job)`
“ett” ⇒ `answer(time(1:00))`

The system assumes that “ett” specifies either when the recording should stop or end — thus the follow-up question, which seems to the user to come from nowhere.

1.2.2 Mismatch between interpretation and generation

If the interpretation and generation modules do not use the same language, then the system utterances can lead the users to believe they can say things that the system does not understand.

Example from the Video GoDiS system

```
S> Du kan lägga till en inspelning,
   ta bort en planerad inspelning, och
   lista planerade inspelningar.
   För att börja om, säg börja om.
   För att sluta, säg sluta.
   You can add a recording,
   remove a planned recording, and
   list planned recordings.
   To start over, say start over.
   To quit, say quit.
S> Huvudmenyn. Vad kan jag hjälpa dig med?
   Main menu. What can I do for you?
U> lista inspelningar (0.75)
   list recordings
S> Ockey. Gå till inspelningar.
   Okay. Go to recordings.
S> Låt oss se. Vill du lägga till en inspelning,
   radera dina inspelningar, eller
   få information om dina inspelningar?
   Let’s see. Do you want to add a recording,
   erase your recordings, or
   get information about your recordings?
```

The initial help message suggests that the user can say "lista inspelningar" (list recordings), but the help message is not up-to-date, and neither is the speech recognition grammar. Instead the interpretation module spots the phrase "inspelningar" (recordings), and the user is asked to choose between roughly the same alternatives that was presented in the help message.

1.3 Solutions to the mismatch problems

Mismatch between the speech recognition, interpretation and generation modules can be described as bugs in the system. Bugs can of course be fixed, but the problem is that it takes lots of time to keep these modules in sync with each other. There is no guarantee that new bugs won't be introduced, and no guarantee that all bugs have shown up.

These solutions are not reusable, instead we want a general and principled solution — we want to enable reuse and rapid prototyping of applications, including grammars.

Solution, part 1: a single grammar for speech recognition, interpretation and generation

If we use one single grammar for all natural language modules, there will be no mismatches between the grammars, and only one grammar needs to be written and maintained for each application. The different modules may require different kinds of grammars, in different grammar formats. So we need to be able to generate all three grammars from a single grammar.

However, we may not want the exact same coverage for the speech recognition, interpretation and generation grammars — e.g. maybe we do not want to understand everything that can be generated.

Solution, part 2: use the Grammatical Framework

GF is a powerful tool for mutilingual grammar development, which can be used to generate grammars in various other formats. Also, subgrammars can be extracted from larger grammars, meaning that speech recognition, interpretation and generation subgrammars may have different coverage.

1.4 Outline of this deliverable

In this deliverable we describe how we have integrated Grammatical Framework with TrindiKit/GoDiS.

In chapter 2 we describe how to get GF grammars working together with TrindiKit. The main idea is that GF works as a combined interpretation and generation module, translating user utterances to dialogue moves, and the system's dialogue moves to utterances. Furthermore, the grammar for speech recognition is automatically generated from the GF grammar, meaning that all three modules are in perfect sync with each other.

Chapter 3 is a general description of how the module system of GF can be used for building libraries that can be used and reused in several applications, much as common libraries for programming languages.

In chapter 4 we describe a cluster of grammars which have been used to build four example dialogue systems in the in-home domain.

The final chapter is a short discussion, and in the appendix we give instructions for downloading, installing and using the unimodal GoDiS grammar library.

Chapter 2

Integration of GF and TrindiKit

2.1 Grammatical Framework in the dialogue domain

Here we discuss the features of Grammatical Framework we make use of in building grammars for dialogue systems.

2.1.1 Abstract and concrete syntax

As explained in section 1.4, the dialogue moves for the dialogue manager are seen as yet another language, which means that in our approach it is crucial that the grammar formalism has support for multilinguality. The feature that makes GF so well suited for multilingual grammars is the clean separation between abstract and concrete syntax. A multilingual grammar then consists of one abstract syntax and several concrete syntaxes, one for each language (or modality representation).

GF is not the only grammar formalism with a clean separation of abstract and concrete syntax — other formalisms include *generalized context-free grammar* (Pollard, 1984), *multiple context-free grammar* (Seki et al., 1991), *linear context-free rewriting systems* (Vijay-Shanker et al., 1987), *higher-order grammar* (Pollard, 2004), *lambda-grammars* (Muskens, 2003) and *abstract categorial grammar* (de Groote, 2001).

However, most of these formalisms are purely theoretical with no working implementation, or just a simple toy implementation. GF is a formalism with a very complete implementation, including tools for grammar checking, parsing, generation and compilation to other grammar formats, including speech recognition grammars.

2.1.2 Multilinguality and resource grammars

There have been some research conducted on multilingual grammars, which is shown by the existence of the ESSLLI 2003 workshop of Multilingual Grammar Development. However, most of the research has been on building large-coverage grammars for several languages in parallel, sharing some common features such as syntactical structure or semantics. These parallel grammars are written in frameworks not ideally suited for multilinguality, such as the ParGram project[1] (Butt et al., 2003) which is written in

[1] `http://ling.uni-konstanz.de/pages/home/butt/pargram/`

LFG, and the LinGO Grammar Matrix[2] (Bender et al., 2002) which is written in HPSG.

GF has a rich module system that enables modularity and library-based grammar engineering. The key features needed are information hiding, high-level module interfaces, and separate compilation. This makes it possible to write resource grammars, which are broad-coverage grammars from which it is possible to extract only the features that are needed for the intended domain.

As of our knowledge there are no research on domain-specific multilingual grammar development. Together with its module system and the possibility to write resource grammars, GF is well suited for this task as well as for building large-coverage grammars.

2.1.3 Embedded grammars

There are several ways one can embed GF grammars in applications.

Embedding the Full GF System

Some applications have been written which use the full GF system as a resource. This can be done in two ways, either by communicating with the interactive GF program by using pipes (Khegai et al., 2003; Hähnle et al., 2002), or by using the GF Haskell API (Hallgren and Ranta, 2000; Burke and Johannisson, 2005).

GF Gramlets

The *GF Gramlets* system (Forsberg et al., 2005) produces syntax editors in the form of Java applets for a given GF grammar. Gramlets implement syntax editing and linearization using XML representations of GF grammars.

The Embedded GF Interpreter

This is an interpreter for compiled GF grammars, supporting parsing and linearization. It is written in Java and its aim is to be small, fast and portable. It can also be run as an OAA agent enabling other OAA agents to use it for parsing linearization and translation. This is the reason why we use the Embedded GF Interpreter to integrate GF with TrindiKit.

2.1.4 Connecting GF to TrindiKit/GoDiS

The main idea is that the dialogue moves of TrindiKit/GoDiS is seen as yet another concrete language in the multilingual GF grammar, just as English and Swedish are. All these languages share the same abstract syntax. Thus interpretation can be done by translating (via the Embedded GF Interpreter) from English or Swedish to dialogue moves, which are then sent to the dialogue manager. Generation is done conversely, by translating the dialogue moves produced by the dialogue manager into the preferred language.

2.2 The Embedded GF Interpreter

The GF system is primarily a command line application for working with GF grammars. It has a significant amount of functionality, such as parsing, linearization, computation, syntax editing, morphological

[2] `http://www.delph-in.net/matrix/`

analysis, compilation of source grammars to canonical GF grammars, conversion of grammars to various formats, translation and morphology quizzes, etc. GF has been in development for a number of years and has grown quite large. It is well-equipped for testing and working interactively with grammars. However, GF is more complex than necessary to be used with embedded grammars. We have therefore developed an interpreter for compiled GF grammars in the Java programming language, version 1.5 (Gosling et al., 2005). The goal of the Embedded GF Interpreter is to make a small and fast implementation of the features necessary for building applications to make use of embedded grammars. Thus, any functionality which is only used during the application development has been delegated to the full GF system.

GF itself is still essential for developing embedded grammars, but it need not be included in the finished system. GF is used to compile the source grammars to the various formats used for parsing, linearization and speech recognition by the finished system. This situation can be compared to that for programming languages such as Java, which can be compiled into byte-code. A compiler is used to convert the human-readable and human-writable source code to a simpler form. Users of the program then only need a runtime environment or virtual machine to run the compiled code.

The full GF system is a rather large executable program, requires a Haskell implementation for the given platform and has a large memory footprint. The aim of the Embedded GF Interpreter is to be small, fast and portable. The size of the compiled interpreter is around 300 kilobytes and it should run on any platform which has a Java 1.5.0 Runtime Environment.

2.2.1 Parsing

The parser computes a set of abstract syntax trees for a given string input.

Compiling GF grammars to parsable format

The full GF system converts the GF grammars to a format which the Embedded GF Interpreter can use for parsing. If the grammar contains finite type dependencies, it is transformed to an equivalent grammar without dependent types. This grammar is then converted to an equivalent Multiple Context-Free Grammar (MCFG; Seki et al., 1991) as described by Ljunglöf (2004a, chapter 3). The MCFG is finally converted to a context-free grammar (CFG), which is used by the Embedded GF Interpreter.

Lexical analysis

The first step in parsing input is to divide it into tokens. The Embedded GF Interpreter has a default lexer which divides the input into simple words (non-empty sequences of letters and digits), quoted strings and punctuation. The user can also write custom lexers which are loaded by the interpreter.

Chart parsing

The parser is a Kilbury bottom-up chart parser, similar to the chart parser described by Ljunglöf (2004b). The algorithm has been modified to support empty rules and to be better suited to implementation in an imperative language.

Tree building

After a successful parse, we need to build abstract syntax trees for the input. The tree building algorithm uses the chart produced by the chart parser.

In order to avoid non-termination with cyclic grammars, a set of used edges for a given input sub-sequence is kept. No edge can be used more than once for a given sub-sequence, which means that not all possible parse trees are generated. This may seem to be too harsh a restriction, but this decision was made based on the belief that for most applications, cyclic uses of the same rules are not essential for the semantics. Another possible solution to this problem would be to build graphs instead of trees. This might seem to be an elegant solution, but traversing such graphs could lead to non-termination if special care is not taken.

Filtering of parse trees

When the MCFG is converted to a CFG, information about e.g. discontinuous constituents will be lost. Thus, the grammar which is used for parsing is over-generating, and some resulting parse trees might be incorrect. Therefore the trees have to be filtered through a GF type-checker, before they are returned by the Embeded GF Interpreter.

2.2.2 Linearization

In GF, *linearization* refers to the inverse of parsing, i.e. the process of producing a string in the concrete syntax from an abstract syntax term.

For linearization, the Embedded GF Interpreter uses a Canonical GF (GFC) grammar, which is produced from a source grammar by the GF system. Canonical GF can be seen as a simple total functional language.

Unlexing

After linearization has produced a list of tokens, the *unlexer* joins the list to create a single output string. A naive unlexer would simply concatenate the tokens, adding a space character between the tokens. However, this does not produce acceptable strings in most languages. For example, in English there should not be a space before most punctuation characters.

The Embedded GF Interpreter currently uses a fairly simple heuristic for unlexing. We define two subsets of the set of all characters: those which should be preceded by a space (essentially all punctuation, closing brackets and closing parentheses), and those which should not be followed by a space (opening brackets and parentheses). These sets are used to determine whether to add a space between two tokens. The full GF system offers some more freedom in the choice of lexing and unlexing algorithms.

2.2.3 Translation

Translation is done by parsing with the source language and linearizing to the destination language. Since parsing may be ambiguous or fail, translation may produce zero or more results.

2.2.4 Java API

The Java API allows the programmer to call the interpreter directly from a Java program. The Java API supports all functionality, such as grammar loading, parsing, linearization and translation.

The method *createTranslator* in the *TranslatorFactory* class is used to create a *Translator* given CFGM and GFCM grammars, and some meta-data. The *Translator* class has methods for parsing, linearization and translation, which are described below. More documentation is available in the Embedded GF Interpreter API reference (Bringert, 2005).

Methods in the Translator Class

The *parse* method takes a language name (the name of a concrete syntax) and an input string. It returns the result of parsing the input string in the given language:

```
Set<Tree> parse(String lang, String text)
```

The *linearize* method takes a language name and an abstract syntax tree and returns the result of linearizing the tree in the given language:

```
String linearize(String lang, Tree tree)
```

The *translate* method takes input and output language names, and a string in the input language, which it translates to a set of strings in the output language:

```
Set<String> translate(String fromLang, String toLang,
                      String text)
```

There are also versions of the parsing and linearization methods which try to use all available concrete syntaxes and return collections of pairs of language name and results:

```
Set<Pair<String,Tree> > parseWithAll(String text)

Set<Pair<String,String> > linearizeWithAll(Tree tree)
```

2.2.5 Typed Abstract Syntax Trees

The Java API uses generic untyped syntax terms, where there is a single class for functions which uses a string for the function name and an array of child terms. Constructing and analyzing such terms can be quite tedious in Java. An untyped abstract syntax term is constructed thus:

```
new Fun("GoTo", new Tree[]{ new Fun("Chalmers"),
                            new Fun("Valand")});
```

A tool, `Grammar2API`, has been written which creates Java classes for representing a given abstract syntax using typed trees. An abstract class is created for each category, and a concrete class inheriting from that class is created for each function in that category.

There is a Visitor (Gamma et al., 1995) interface for each category, which has methods for each function in the category. Code for converting between typed and untyped trees, as well as typed wrappers around the untyped parsing and linearization methods are also generated. The abstract syntax tree above can be built by simply using the constructors of the generated classes:

```
new GoTo(new Chalmers(), new Valand());
```

2.2.6 OAA Agent

To allow the GF interpreter to be used in multi-agent systems and from programs written in languages other than Java, an Open Agent Architecture (OAA; Martin et al., 1999) wrapper has been written.

OAA is a framework for multi-agent systems. Communication between the agents is done by sending terms in the Interagent Communication Language (ICL), a subset of Prolog. There are OAA implementations for several programming languages, including Java.

The GF OAA agent has *solvables* (methods) for parsing, linearization, translation, listing languages and grammars. Since OAA uses a unification-based approach to method calls, the GF agent solvables can be used quite flexibly. For example, if the language argument to the parsing solvable is uninstantiated (i.e. it is a variable), all available languages in the given grammar will be tried. The fact that an OAA agent can return multiple solutions to a request is used to return ambiguous parse results. When the language argument is uninstantiated, the parser tries to parse with all available concrete syntaxes.

The solvables are documented in detail in the Embedded GF Interpreter documentation.

2.3 Representing TrindiKit/GoDiS dialogue moves in GF

In order to use GF to write grammars to be used with TrindiKit/GoDiS one must first look at the semantic representations used by the system. One of the central ideas in GoDiS is that the user and the system share a part of the information state which they manipulate to achieve results. This is done by means of dialogue moves (Larsson, 2002).

All utterances are interpreted as a list of dialogue moves (DMs). Before the integration with GF this interpretation was done by finding key words or key phrases which corresponded to certain dialogue moves.

The approach to DMs in GoDiS is that the movea realized by an utterance is determined by the relation between the utterance content and the activity in which the utterance occurs. In effect this means that an utterance could realize different dialogue moves in different domains.

2.3.1 Dialogue Moves in GoDiS

greet and quit

To begin and end any dialogue session the `greet` and `quit` commands are used respectively. These are atomic DMs which correspond to greetings and closure phrases in natural language. The examples of greetings below are from one GoDiS application handling a VCR and one dealing with an Agenda.

```
greet
```

```
'Welcome to the VCR manager!'
'Hi! This is AgendaTalk, your personal talking agenda.'
```

ask

One of the central premises of Issue-based Dialogue Management as implemented in GoDiS is that one of the main purposes of dialogue is to raise and respond to questions. This is done by performing `ask` and `answer` moves. This section describes the former, and the latter is described in the following section.

`ask(`*q*`)`, where *q* : Question

The content *q* of the `ask` move is a question. Three types of questions are handled by GoDiS: y/n-questions, wh-questions, and alternative questions.

Y/n questions are formed from propositions. A proposition in GoDiS is a basic formula of predicate logic consisting of an n-ary predicate together with constants representing its arguments. As the domain is greatly limited in most dialogue systems it is often not necessary to keep a very elaborated semantic representation of utterances. The utterance “I want to go to Chalmers” in a dialogue system handling the Gothenburg Tram System would not, as one might expect, be represented as `want(user, go-to(user, chalmers))` but instead GoDiS would use the knowledge of the domain and reduce the utterance to a simple `dest-tram-stop(chalmers)`.

`ask( dest-tram-stop(chalmers) )`
“Do you want to go to Chalmers?”

Wh-questions are lambda abstractions of propositions. One can see these as partially uninstantiated propositions, waiting for additional information from some dialogue participant.

`ask( X^dest-tram-stop(X) )`
“Where do you want to go?”

Alternative questions are sets of y/n-questions.

`ask( {dest-tram-stop(chalmers), dest-tram-stop(lindholmen)} )`
“Do you want to go to Chalmers or do you want to go to Lindholmen?”

answer

The `answer` move is used to answer questions.

`answer(`*a*`)`, where *a* : ShortAnswer or *a* : Proposition

A ShortAnswer is a semantically underspecified proposition (Larsson, 2002, p23).

request

An action-oriented dialogue is one that involves requesting (and possibly performing) non-linguistic actions. Requesting an action is done using the `request` move, whose argument is an action.

`request`(α), where α : Action

confirm and report

After a dialogue participant (DP) has recieved a request and performed the requested action, the DP gives feedback on the successful execution of the action via the `confirm` move.

`confirm`(α), where α : Action

For more complex cases, e.g. where an action goes wrong or if an action takes a long time, there is also the possibility of using a `report` move to report on the current state of an action that is being carried out.

`report`(α, *Status*), where α : Action and *Status* : Status

The Status can be a number of things, ranging from `done` (which would correspond to the confirm move above) to `failed` or even a detailed description of what has gone wrong.

Interactive Communication Management

Larsson (2002) uses Interactive Communication Management (ICM) as a general term for coordination of the common ground. ICM dialogue moves are explicit signals enabling coordination of updates to the common ground, such as keeping track of topics currently under discussion, subactivities, sequencing and turn taking.
There are two types of ICM dialogue move patterns in GoDiS:

1. `icm:`*`level*polarity`*`{:`*`arguments`*`}`

   This pattern is used for ICM dealing with feedback and grounding. There are five *action levels* – contact, perception, semantic understanding, pragmatic understanding, and acceptance/reaction. These are abbreviated `con`, `per`, `sem`, `und` and `acc` respectively. There are three *polarities* – positive, negative and interrogative (checking), abbreviated `pos`, `neg` and `int`, respectively. Some feedback moves also require *arguments*.

2. `icm:`*`type`*`{:`*`arguments`*`}`

   This pattern is used for ICM other than feedback. The arguments are also here optional, and the *type* can e.g. be `reraise` for reraising issues or `accomodate` for accomodation.

Below are a few examples of ICM moves.

`icm:con*int`	“Are you there?”
`icm:per*pos`	“Ok”
`icm:per*pos:`*`'foo'`*	“I thought you said *’foo’*”
`icm:acc*neg:X^tram_stop(X)`	“I cannot answer questions about stations.”
`icm:loadplan`	“Let’s see. . . ”

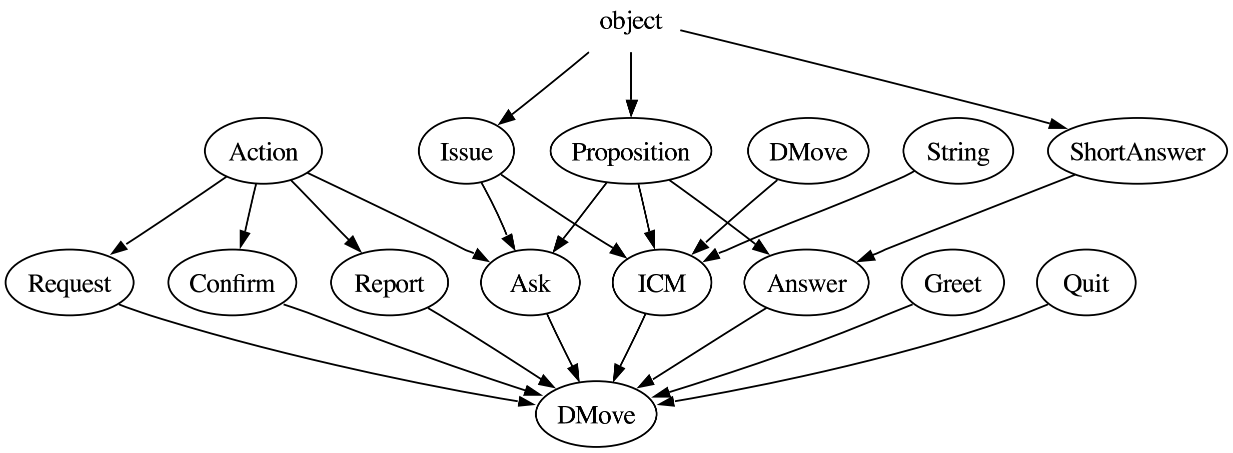

Figure 2.1: The hierarchy of categories in GF GoDiS.

2.3.2 Categories in GF

The dialogue moves themselves naturally become categories when making an abstract GF grammar used to translate between natural language and the semantic representations in GoDiS. There is also a need for the arguments from the section above to be represented in this way.

Creating an abstract grammar in GF means making use of categories and functions. As there are no different types of categories inherently in GF, the graph in figure 2.1 serves as a hierarchical tree of sorts. An arrow from node A to node B in the graph means that there is a function creating an object of B from an object of A.

For instance, an object of the category DMove (Dialogue Move) can be created by a function that takes something of the category Request, which in turn has been made with a function that takes an Action. The *object* in the graph is not a specific category but a name for those objects from the lexicon which are not actions. In other words, these may be locations and furniture in an in-home domain, or artists and songs in a music related domain.

Note that the hierarchy of GF categories is not fully consistent with the description of dialogue moves in GoDiS in section 2.3.1. However, this is not a problem since the GoDiS dialogue moves are specified by just another concrete language in the GF grammar. As long as there is a consistent translation from GF abstract syntax to GoDiS dialogue moves, any hierarchy of GF categories is sufficient.

2.3.3 Using dependent types

GF allows for the use of dependent types, a useful tool when it comes to creating meaningful lists of dialogue moves from utterances. With a set of activities, or tasks, as dependent types it is easy to dictate a setting in which a certain answer move is accepted together with a request or an answer. When creating a speech recognition grammar from the GF grammar these restrictions are essential.

The example grammar in figure 2.2 is a somewhat simplified excerpt from the DJ GoDiS grammars. There are two Actions, `play` and `shift` corresponding to two of the system's dialogue plans. There are also two Propositions, `madonna` and `left`. The dependent type Tasks are used to make sure that actions and objects within the same utterance belong to the same task. In this grammar the Tasks are `playTask` which deals

```
playTask           : Task;
speakerTask        : Task;
play               : Action playTask;
shift              : Action speakerTask;
madonna            : Proposition playTask;
left               : Proposition speakerTask;
makeAnswer         : (t : Task) -> Proposition t -> Answer t;
requestCompounded  : (t : Task) -> Action t -> Answer t -> CompoundedRequest;
```

Figure 2.2: Dependent types in the GF grammar for DJ GoDiS (somewhat simplified).

```
play    = {s = "play"};
shift   = {s = "shift the balance"};
madonna = {s = "Madonna"};
left    = {s = "to the left"};
makeAnswer        _t prop    = {s = prop.s};
requestCompounded _t act ans = {s = act.s ++ ans.s};
```

Figure 2.3: Examples of concrete linearizations for the grammar in figure 2.2.

with the scenario of playing music and `speakerTask`, which captures the speaker control dialogue moves. In order to make a compounded Request by pairing an Action and an Answer, their corresponding tasks need to match. For example, assuming the concrete linearizations shown in figure 2.3, there are four possible combinations of an Action and an Answer. However, two of them ("shift the balance Madonna" and "play to the left") are ungrammatical. The remaining two correct compounded Requests are

- "play Madonna", with the abstract syntax term
  `requestCompounded playTask play (makeAnswer playTask madonna)`
- "shift the balance to the left", with the abstract syntax term
  `requestCompounded speakerTask shift (makeAnswer speakerTask left)`

2.4 Extracting speech recognition grammars from application grammars

In order to improve recognition accuracy, speech recognition engines often use grammars to determine which inputs are to be expected. Speech recognition grammars (SRGs) are often simple context-free grammars. In this application, the grammar is simply used to determine whether a given string belongs to the language or not, the so-called *recognition problem*.

Writing a separate grammar for the speech recognizer, and keeping it in sync with the grammar used by the parser requires some effort. To eliminate this problem, a compiler from the internal CFG (Ljunglöf, 2004a) format used by GF to some speech generation grammar formats has been implemented.

2.4.1 Speech Recognition Grammar Formats

There are a number of existing formats for speech recognition grammars:

JSpeech Grammar Format (JSGF)
JSGF (Hunt, 2000) is a plain-text language for context-free grammars used in the Java Speech API (JSAPI; JavaSpeech, 1998a,b).

Nuance Grammar Specification Language (GSL)
GSL (2003) is a plain-text language for context-free grammars used by the Nuance (2003) speech recognizer.

Speech Recognition Grammar Specification (SRGS)
SRGS (2004) is a W3C standard for speech recognition grammars. It has two equivalent syntactic forms, Augmented Backus-Naur Form (ABNF) and XML.

2.4.2 Implementation

The internal context-free grammar for a given concrete syntax is first transformed to a generic simple context-free format for the speech modality:

- Removal of explicit and implicit left recursion by Paull's algorithm (Moore, 2000). The algorithm does not preserve the structure of the grammar, but as speech recognition grammars are not used to produce parse trees, this is not a problem.
- Removal of productions which use categories in which there are no productions. This is done by fix-point recursion as each step may create new empty categories.

The generic context-free speech grammar is then converted to either GSL or JSGF and printed. Punctuation is removed before printing, as it is not part of the spoken language. All upper case characters in tokens are converted to lower case, for the same reason. If the source grammar contains punctuation or upper case characters, the CFGM grammar, which is used for parsing (see section 2.2.1), will not be able to parse all output from the speech recognizer. This problem could be solved by having GF remove punctuation and capitalization before producing the CFGM grammar, instead of when creating the speech recognition grammar.
Speech recognition grammar compilation has been added to the GF system. The `pg` ("print_grammar") command has been given two additional values for the `-printer` flag: `gsl` and `jsgf`.

2.4.3 Evaluation

To evaluate the speech recognition grammar compiler we compiled the DJGoDiS GF grammar to a Nuance grammar and ran batch tests on the resulting recognition package. To collect a test set we let students figure out how they would address a speech-enabled MP3 player by writing Nuance grammars that would cover the domain and its functionality. Another group of students evaluated these grammars by recording utterances they thought they would say to an MP3 player.

Evaluation	WER	SER
Out-of-coverage evaluation	77.23	95.01
In-grammar evaluation	6.30	8.31

Table 2.1: *Word error rates (WER) and Sentence Error Rates (SER) for the Nuance Grammar*

The recording test set was made up partly of the students' recordings. Additional recordings were carried out by letting people at the department record randomly chosen utterances from the evaluation test set. The final test set included 522 recorded utterances from 18 persons (9 female and 9 male voices). This test set was used to compare recognition performance between the different models under consideration.

The test sets are just an approximation to the real task and conditions as the students only capture how they think they would act in an MP3 task. Their actual interaction in a real dialogue situation may differ considerably so ideally, we would want recordings from the dialogue system interactions. We plan to make subsequent tests using the actual system.

In addition to the recorded evaluation test set a second set of recordings was created covering only in-grammar utterances by randomly generating a test set of 300 utterances from the GF grammar. These were recorded by 8 persons. This test set was used to evaluate in-grammar recognition performance.

As the reader may have noticed, the word error rates are very high, which is partly due to a totally independent test set with a lot of out-of-vocabulary words indicating that domain language grammar writing is very subjective. The students have captured a quite different language for the same domain and functionality. This shows the risk of a hand-tailored domain grammar and the difficulty of predicting what users may say. It should be pointed out that the GF MP3 grammar used for this evaluation was in a preliminary stage and was for example missing imperatives which seem to be a common form in the test corpus. A closer look at the results gives a hint that in a lot of the cases the transcription reference and the recognition hypothesis hold the same semantic content in the domain (e.g. "höj ljudet" vs "höja ljudet").

However, the in-grammar evaluation seems promising and shows that the compiler works perfectly. The compiled grammar seems to be in perfect sync with the GF grammar as it recognizes the recorded utterances generated from the GF grammar. This would mean that we have managed to save the dialogue system developer a lot of work by giving him automatically a speech recognition grammar in sync with his GF grammar used for parsing.

2.4.4 Related work

Most work on compiling grammar formalisms to CFG has been made on unification-based formalisms, such as context-free approximations of HPSG (Kiefer and Krieger, 2000) and compact translations of restricted unification grammars (Moore, 1999). Dowding et al. (2001) discusses different algorithms for generation of context-free speech recognition grammars. The Regulus system (Rayner et al., 2003) is an open source implementation of these ideas.

GF is not unification-based, but instead based on linearization functions on abstract syntax terms. This makes the underlying algorithms for compilation to CFG different from the unification-based algorithms (Ljunglöf, 2004a, chapter 3).

Chapter 3

Resource grammars and grammar engineering

3.1 Library-based grammar engineering

In general-purpose software engineering, the idea of using *libraries* as the way of structuring and reusing code is well-established (Parnas, 1972): the idea is to divide the code into reusable units each of which formalizes a specific area of expertise. Prime examples of this are libraries for scientific computing, which implement methods of numerical analysis. More mundane examples are sorting algorithms, graphical user interface toolkits, and other general-purpose libraries that are used in almost any piece of software. Libraries make advanced techniques available to all programmers, without everyone having to learn the details of all techniques.

In the domain of Natural Language Processing, the idea of *grammar libraries* appears as useful and natural instance of library based software engineering. To take an example from the TALK domain: a dialogue system for communicating with MP3 software may want to say things about the *current song*, the *current artist*, the *current playlist*, etc. To translate this information into different languages, many kinds of linguistic data are needed: the translations of the words *current*, *song*, *artist*, etc; the inflection of these words in different forms; and their proper syntactic combinations. It is natural to expect such data to be found in a grammar library written by an expert of the language. Thus for instance the programmer that wants to add the voice command "save the current playlist" in Spanish should only need to call functions from such library, something like

```
generate(Spanish, Action(save, Modif(current,playlist)))
```

and be sure that the proper Spanish phrase will appear. In English, the corresponding code would be

```
generate(English, Action(save, Modif(current,playlist)))
```

3.2 GF support for modules and libraries

GF has a module system that enables modularity and library-based grammar engineering (Ranta, 2005). The key features needed are information hiding, high-level module interfaces, and separate compilation.

Optimizations that reduce the run-time penalty of using libraries and other abstractions are also essential.

Modules can be extended, imported and instantiated by other modules. This can be done on several modules at the same time, thus forming a module hierarchy. Furthermore, operations in resource modules can act as translation functions from concrete syntax into abstract syntax. This means that grammars can be composed by letting the concrete syntax of one grammar serve as the abstract syntax of another. The compilation process then dismisses the intermediate syntaxes, similar to compilation of the composition of finite state transducers (Karttunen et al., 1996; Mohri, 1997).

Translators from GF code into more well-known formats (such as Java) make it possible to use GF grammars in other programs without even knowing the GF formalism. The Embedded GF Interpreter described in section 2.2 is an example of this.

3.3 Grammar engineering

Grammar engineering is the process of designing and implementing grammars on a computer. The field is dominated by projects in which small groups (down to the size of one individual) of linguists produce big grammars, with the ambition of covering all or most of a natural language and parsing a corpus of texts. The grammars often reflect advanced linguistic theories, and are inaccessible not only to non-linguists but also to linguists in another "school". Their goal is clearly not to serve as libraries that formalize known facts of language, but to push further the limits of grammatical research. **Our comment**: Once a grammar is finished, it should be possible to build up an interface through which the grammar can be accessed without thorough understanding of it.

Because existing grammar implementations are hard to reuse, new ones are usually built from scratch: even the basic, uncontroversial facts of languages are defined again and again. Part of the motivation has to do with research ambitions: a new grammar can be meant to "change our view" of the grammar of a language completely and cannot therefore take anything for granted. **Our comment**: With a modern programming language with features such as higher-order functions and parameterized modules, it is usually possible to reuse an old implementation, hide its details behind a new interface, and thereby change completely the view of what is implemented.

There are also practical reasons for starting grammars from scratch: NLP is notoriously a field with restrictive licenses, unavailable resources, and exotic (proprietary, experimental, platform-dependent, etc) code formats losing all processing tools in less than a decade.

In general, one can say that the *culture* of developing software as reusable, modular, publicly accessible libraries is not as wide-spread in grammar writing as e.g. scientific computing. In a representative position paper on grammar engineering (Copestake and Flickinger, 2000), the whole idea of modularity and information hiding is declared inadequate in the domain of grammars: "information hiding is the very antithesis of productivity in grammar writing. . . for instance, a feature used in morphology may surprisingly turn out to be useful in semantics". Blocking the view of other modules prevents the working grammarian from finding generalizations. As a more general argument, Copestake and Flickinger (2000) points out that grammar engineering is still in the state of research rather than engineering, and that normal software engineering ideas do not apply. **Our comment**: We admit that research remains to be done about the facts of natural languages, but grammar writing which is part of software production could get far with existing knowledge if it was properly transmitted from linguists to programmers.

An exception to the big research-oriented grammar idea is the CLE (Core Language Engine) project

(Alshawi, 1992). CLE grammars are meant to be used as libraries in other (Prolog) programs that need natural language parsing, generation, or semantics. The idea of specializing a big grammar to a small situation is stated explicitly in Rayner et al. (2000). The major problem with CLE is that it is no longer continued, and that its results are mostly unavailable because of licenses and bit rot. Also the level of modularity and abstraction, partly because of the limitations of the Prolog programming language, is not what more recent library-based software engineering expects. But in many ways, we see CLE as the most important point of comparison to our work.

3.3.1 Resource grammar libraries

Resource grammar libraries encapsulate linguistic knowledge that can be used in other programs. In other research projects we have started writing such grammars for ten languages: Danish, English, Finnish, French, German, Italian, Norwegian, Russian, Spanish, and Swedish. Each resource grammar provides the following:

Morphology. A program that is able to generate and analyze all inflectional word forms is more or less standard knowledge for many languages, and an obvious part of the resource grammar.

Syntax. Phrase categories and combination rules of phrases should give a full account of word order and agreement. A sizable fragment of syntax for both written and spoken language was identified in the CLE project. It is well understood how to define a corresponding fragment for several languages. To meet the needs of applied grammar engineering, we need not be able to cover all linguistic phenomena, as we would in a grammar whose purpose is to parse running text.

Lexicon. Inflectional and subcategorization data of the most frequent, say, 3,000 words of each language is useful. In addition, a lexicon extension tool is needed to make it easy to add domain-specific vocabulary, which could never be completely included in a prebuilt lexicon.

Common APIs. Corresponding structures in different languages should be accessible via similar library calls, even though the structures have different implementations and are not always translation equivalents. For instance, the rule of modifying a noun with an adjective (e.g. *current playlist*) exists in all languages that we are considering. Moreover, related languages (e.g. Danish, Norwegian, and Swedish) can share more interface elements than unrelated ones.

Transfer lexica. Given a word in e.g. Swedish, it is useful to know what its possible equivalents are in Russian. Such correspondences are collected in a Swedish-Russian transfer lexicon. The lexicon can be used as a starting point of software localization, although in general the choice of the right equivalent must be made manually by an expert of the application domain.

As of the moment, these resource grammars are not used in the unimodal grammar library presented in chapter 4. However, in the future we aim to incorporate them to be able to make use of their linguistic coverage.

Chapter 4

The unimodal GoDiS grammar library

4.1 The library file structure

The **resource** library of the unimodal GoDiS grammar library consists of a number of different modules presented in section 4.2. The **core** library (section 4.3) contains the resource grammar used as a base for specific application grammars. It is divided into three parts, one for the user specifics, one for the system and one shared. Then there is the **domain** library which contains the specific application grammars, see section 4.4.

4.2 The in-home abstract resource (API)

The *objects* discussed in section 2.3.2 are gathered in the in-home abstract resource by means of small, grouped grammars handling a few (one to three) different categories each. In the Media module, for instance, there are grammars dealing with song titles, artists and the different radio and TV stations available.

Module	**Content**
Locations	A collection of locations, i.e. countries, cities, buildings etc.
Numbers	Numbers, ordered and regular.
Media	Artists, Songs, Radio and TV stations etc.
Home	Furniture, rooms etc
Time	Dates, Days and Time.
Events	different events such as wedding, meeting etc.

These are small grammars that are mobile, making it easy to reuse code. The idea is that you create a data base grammar for each of your domain specific grammars and make use of the Modules you need. Figure 4.1 shows how grammars from the Media module are used to create a database of *objects* to be used with our DJ GoDiS grammars (described in section 4.4.1).

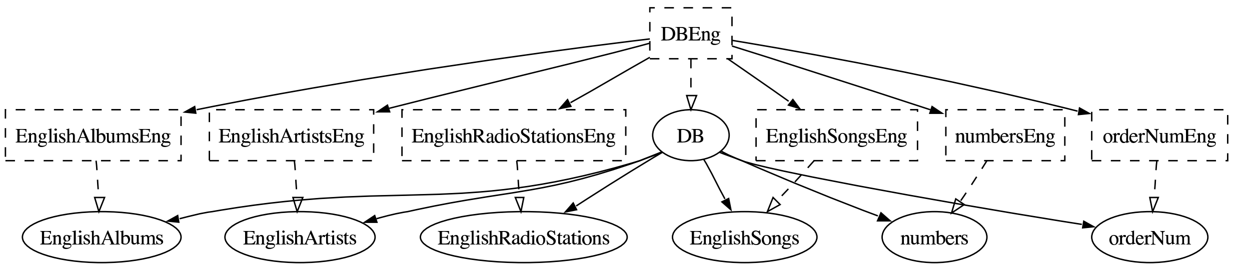

Figure 4.1: The grammars from the Media module: Dotted boxes are concrete linearization modules and solid ovals are abstract syntax modules

4.3 The core grammar for the GoDiS Dialogue Manager

4.3.1 The abstract syntax

The core grammars contain domain independent categories and functions to manipulate them. They are divided into three parts, one for the User specific parts and one for the System as well as a collection of shared grammars where the details they have in common are handled.

Shared contains the following categories:

Category	**Usage**
S	Top Category
DMove	The Dialogue Move
Action Task	The basic Action with its dependent type
SingleAction	An Action that does not have a type
Task	The dependent type
Greet	Greet move
Quit	Quit move
Answer Task	The Answer move with its dependent type
NegAnswer Task	A negative Answer with its dependent type
Ask Task	The Ask move with its dependent type
SingleAsk	An Ask that does not have a type
Request	The Request Move
ICM	A collection term for ICM
Per_ICM	An ICM
Acc_ICM	— ” —
Per_ICM_Followed	An ICM that can be followed by something
Acc_ICM_Followed	— ” —

User contains the following categories:

Category	**Usage**
CompoundedRequest	A Request which also has an Answer attached
CompoundedAsk	An Ask which also has an Answer attached
AnswerList Task	A collection of answers with its dependent type

System contains the following categories:

Category	Usage
DMoves	A collection of DMoves
Proposition	The Propositions
Other_ICM	An ICM
Sem_ICM	— ” —
Und_ICM	— ” —
Other_ICM_Followed	An ICM that can be followed by something
Sem_ICM_Followed	— ” —
Und_ICM_Followed	— ” —
Confirm	The Confirm move
Report	The Report move
SystemAsk	Asks specific for the system
Issue	A collection term for the Issues
PropIssue	An issue made from a Proposition
AskIssue	An issue made from an Ask
ListIssue	— ” —
IssueList	Related to the lists of Issues
ListItem	— ” —

The Categories are molded into DMoves by chains of functions. For example a Proposition is made into an Answer which in turn is transformed into a DMove by the use of the two functions below. Task is a category which is inherited from the Proposition by the Answer, but it is not needed in the DMove itself.

```
makeAnswer : (t : Task) -> Proposition t -> Answer t;
makeAnswerMove : (t : Task) -> Answer t -> DMove;
```

4.3.2 The concrete English GoDiS core grammar

The linearization to English is fairly straight forward. There are the concrete grammars for English and a small resource grammar containing operations one might want to reuse.

The following are the functions dealing with making a DMove out of a SingleAction.

Abstract grammar

```
play : SingleAction;
makeRequest : SingleAction -> Request;
makeRequestMove : Request -> DMove;
```

Concrete grammar

```
top_command = {s = variants { ["play"] ; ["start the music"]
                            ; ["start playing"] }};
makeRequest req = {s = req.s};
makeRequestMove req = {s = variants { (choosePre ! Req) ; [] }
```

```
                                 ++ req.s ++
                                 variants { (choosePre ! Req) ; [] }};
```

Resource module

```
oper choosePre : Form => Str = table {
     Ques => ["can i"];
     Req  => variants{ ["i want to"] ; ["i would like to"] }};
oper choosePost : Form => Str = table {
     Ques => [];
     Req  => ["please"]};
```

This particular grammar fragment is equivalent to the regular expression

(*i want to* | *i would like to*)?
(*play* | *start the music* | *start playing*) (*please*)?

4.3.3 The concrete Swedish GoDiS core grammar

The Swedish linearization follow the same rules as the English one. There are concrete files for linearization to Swedish for each of the abstract ones and a resource file for frequently used operations.

Abstract grammar

```
play : SingleAction;
makeRequest : SingleAction -> Request;
makeRequestMove : Request -> DMove;
```

Concrete grammar

```
top_command = {s = variants { ["spela"] ; ["starta musiken"]
                            ; ["börja spela"] }};
makeRequest req = {s = req.s};
makeRequestMove req = {s = variants { (chooesPre ! Req) ; [] }
                           ++ req.s ++
                           variants { (choosePost ! Req) ; [] }};
```

Resource module

```
oper choosePre : Form => Str = table {
     Ques => ["kan jag"];
     Req  => variants{ ["jag vill"] ; ["jag skulle vilja"] }};
oper choosePost : Form => Str = table {
     Ques => [];
     Req  => ["tack"] };
```

This fragment is equivalent to the regular expression

(*jag vill* | *jag skulle vilja*)?
(*spela* | *starta musiken* | *börja spela*) (*tack*)?

4.3.4 The concrete GoDiS dialogue move representation core grammar

For the linearization to Dialogue Moves fewer variants are needed.

Abstract grammar

```
play : SingleAction;
makeRequest : SingleAction -> Request;
makeRequestMove : Request -> DMove;
```

Concrete grammar

```
play = {s = ["start"]};
makeRequest req = {s = ["request"] ++ ["("] ++ req.s ++ [")"] };
makeRequestMove req = {s = req.s};
```

No resource module

Subsequently the concrete grammar for the semantics used in GoDiS, for the same abstract grammar as for English and Swedish above, produces just one linearization:

request(start)

4.4 The specific domain grammars

The abstract domain files contain no categories, as the core grammars work as an API. These grammars are also divided into Shared, User and System grammars where the User and System make use of their Shared elements. Figure 4.2 shows the module dependencies of one of the DJ GoDiS grammars, namely the system specific concrete grammar for English.

As visible in the figure the Domain grammars inherit the Core grammars and also the DB (database) grammar which is a collection of the in-home resource modules needed for the particular application (as described in section 4.2).

4.4.1 DJ GoDiS

The DJ GoDiS grammar needs a lexicon consisting of music and playback related terminology. Most importantly the basic functions of an audio player as well as a large collection of possible artists, songs, radio stations etc.

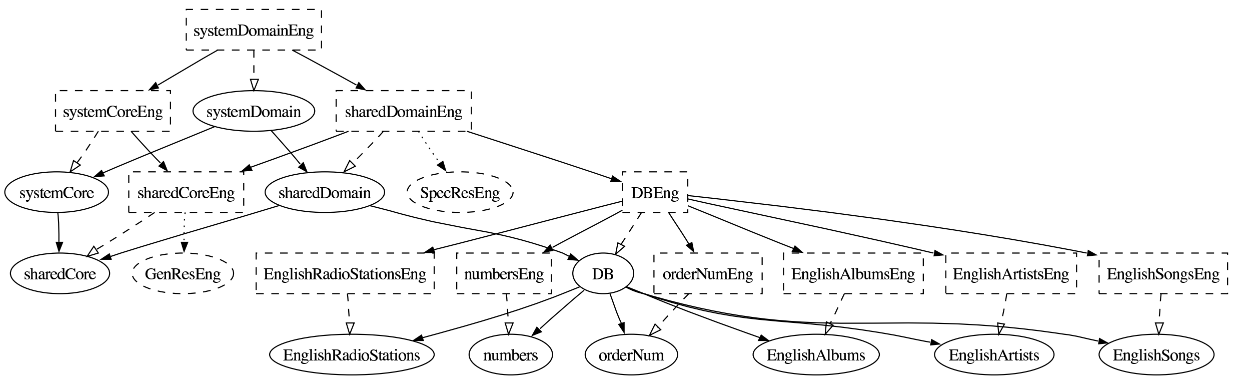

Figure 4.2: The module dependencies of one the DJ GoDiS grammars: Dotted boxes are concrete linearization modules, solid ovals are abstract syntax modules and dotted ovals are resource modules.

Use of resources

From the GoDiS resource library this application makes use of the Media modules, giving it access to a fair number of artists, songs, radio stations and albums. In order to handle the use of a playlist it also uses the Numbers module.

Dialogue plans

The DJ GoDiS dialogue system is used for managing our Player agent. Therefore there are plans dealing with most of the solvables provided by the agent. The most important functionality is listed below.

- Add an item to the playlist
- Deleting items from the playlist
- Playing
- Pausing
- Stopping
- Resuming
- Moving between the items in the playlist (Next, Previous)
- Shuffling the list
- Moving within the items in the playlist (Fast Forward, Rewind)
- Changing the volume
- Changing the balance between the speakers
- Asking about items, artists and songs

Special feedback

In order to make the conversations more natural, feedback from the system does not conform to a preset mold. For each plan there is a specific confirmation linearization as the following examples show.

Plan	Confirmation
Adding to a playlist	"the playlist is increased"
Raising the volume	"turning up the volume"
Resuming the playback	"resuming the music"
Shuffling the playlist	"the playlist has been shuffled"
Rewinding	"rewinding"

Tasks

In order to make correct Dialogue Move combinations there are several tasks.

Task	Usage
playTask	Playing a specific item in the playlist
addTask	Adding an item to the playlist
removeTask	Removing an item from the playlist
speakerTask	Changing the balance between the speakers
artistQuestion	Asking about an artist
songQuestion	Asking about a song

Application specific solutions

For this system the user needs to be able to give an AnswerList for the addTask and the playTask containing a song and an artist. In the User specific grammars:

```
answerSongArtistPlay : Song -> Artist -> AnswerList playTask;
answerSongArtistAdd  : Song -> Artist -> AnswerList addTask;
```

Example dialogue

```
U> I would like to add the song crazy with madonna please
   [request(add), answer(item('crazy')),
    answer(group('madonna'))]
S> [icm:per*pos:'crazy', icm:und*neg, ask(X^item(X))]
   I thought you said 'crazy'.
   I am sorry I do not understand what you mean.
   What song do you want to add to the playlist?
U> like a prayer
   [answer(item('like a prayer'))]
S> [icm:sem*pos, icm:und*int:groupToAdd('madonna')]
   Ok. Madonna, is that correct?
U> yes
   [answer(yes)]
S> [confirm(add)]
```

```
   The playlist has been increased.
U> play
   [request(start)]
S> [confirm(start)]
   Starting the player.
```

4.4.2 AgendaTalk

The AgendaTalk grammars have been created to be used with the AgendaTalk application. The GF grammars written for the system demand that the system is rewritten slightly, so they have not yet been fully incorporated.

The Agenda application is a speech enabled scheduling management system. The GoDiS application, AgendaTalk, works as a voice interface to a web-based calendar in the in-home environment or on a handheld computer that could be used in the in-car environment.

The information sent to the calendar database will be type of event, location, start time and end time. In case end time is not given the calendar will add this automatically.

The system is supposed to work in three languages: Swedish, English and Spanish. At the moment only English and Swedish are covered by the GF grammars.

Use of resources

The AgendaTalk grammars use the Date, Time, Location and Event modules.

Dialogue plans

The GoDiS application, AgendaTalk, is the spoken interface to the calendar and will support the following capabilities corresponding to the calendar device:

- Add/Edit/Delete events
- Repeating events
- Ask for repeating events
- Add restricted types of events such as meetings, appointment, presentations etc.
- Change and delete events talked about during the dialogue but yet not added to the calendar
- Ask for the location, date or time of a certain event
- Ask if booked a certain time or date
- Ask for all bookings during a whole day
- Ask for all bookings of a certain type e.g. all meetings during during a time period e.g. next week

The minimum of parameters of an event is: type of event, start time, start date, end time and end date. However the user may also add other information about the event such as e.g. location of the event.

Special feedback

Apart from the specific confirms which work the same way as for the DJGoDiS application, AgendaTalk gives complex feedback on notes given to the Agenda. These consist of non-atomic propositions.

```
note(event(meeting),
     start_time(16:15),
     start_date(2005.05.10),
     end_time(17:15),
     end_date(2005.05.10))

"meeting at four fifteen on the tenth of may,
 ending at five fifteen on sunday"
```

Tasks

There are tasks for adding, removing and changing entries to the Agenda as well as two different ones for entry lookup.

Application specific solutions

This domain specific grammar contains complex propositions.

Example dialogue

```
S> [ask(set([action(add_details), action(change_details),
           action(clear_agenda)]))]
   Do you want to  add some more information,
   change some of the information or delete it all?
U> add some information
   [request(add_details)]
S> [icm:und*pos, icm:loadplan, ask(X^location(X))]
   Okay.  Lets see.  Where would the meeting take place?
U> at plaza
   [answer(location(plaza))]
```

4.4.3 GoDiS deLux

The GoDiS deLux application grammar is written to use with a dialogue system that controls lamps in an intelligent home.

Use of resources

The grammar uses the Home module, which contains lamps and locations within a home such as rooms and floors.

Dialogue plans

- Turning on/off a light
- Adding/removing a light
- Asking about the status of lights
- Asking about available lights

Tasks

There are Tasks for each of the different plans.

Application specific solutions

The *objects* used in this system are not named entities but rather simple nouns.

Example dialogue

```
U> I would like to turn on the table lamp please
   [request(TurnOn), answer(lamp(tablelight))]]
S> [report(turnOn,noSuchLamp)]
   There is no such lamp.
U> what lamps do I have
   [ask(X^lamp(X))]
S> [answer(lamp(ceilinglamp)), answer(lamp(floorlamp))]
   Ceiling lamp and floor lamp.
```

4.4.4 Video GoDiS

The Video GoDiS grammar was written to be used with the Video GoDiS dialogue system. The system was created to handle the programming of a VCR.

Use of resources

This grammar use the Media and Time modules.

Dialogue plans

- Adding a recording
- Clearing all recordings
- Asking about current recordings

Tasks

There are Tasks for adding recordings and asking about them.

Application specific solutions

This grammar uses complex Propositions. The *recording* is a proposition made out of the propositions date, start_time, stop_time and channel.

Example dialogue

```
U> I would like to add a recording today at ten
   [request(add_recording), answer(date(today)),
    answer(start_time(10:00))]
S> [icm:sem*pos, icm:loadplan, ask(X^channel(X))]
   OK. Lets see. What channel do you want to record from?
U> BBC Prime until eleven
   [answer(channel(bbc_prime)), answer(stop_time(11:00))]
S> [icm:sem*pos, confirm(add_recording)]
   Ok. I have added a recording.
U> what recordings do I have now
   [ask(X^recording(X))]
S> [answer(recording([date(today), start_time(10:00),
                      stop_time(11:00), channel(bbc_prime)]))]
   There is a recording programmed for today
   between 10 and 11 from BBC Prime.
```

Chapter 5

Conclusions

5.1 Contents of the associated software

The software package for this deliverable consists of a library of grammar and dialogue system development tools and a collection of grammars. These together constitute what is loosely called *The TALK Grammar Library* in the Technical Annex. It consists of:

- Grammatical Framework, version 2.2, consisting of among others:
  - Speech Recognition Grammar Compiler (see section 2.4)
- Embedded GF Interpreter (see section 2.2)
- TrindiKit, version 4 (alpha release)
- GoDiS core dialogue system, which is used by the following dialogue applications:
  - DJ GoDiS, a dialogue system for controlling an MP3 player
  - GoDiS deLux, a dialogue system for controlling lights
- Unimodal GoDiS Grammar Library, consisting of:
  - resource modules for Locations, Numbers, Media, Home, Time and Events (see section 4.2)
  - the Core grammar for the GoDiS dialogue manager (see section 4.3)
  - the application grammars:[1]
    - DJ GoDiS (see section 4.4.1)
    - AgendaTalk (see section 4.4.2)
    - GoDiS deLux (see section 4.4.3)
    - Video GoDiS (see section 4.4.4)

[1]Currently we only have working dialogue systems for the DJ GoDiS and GoDiS deLux applications.

5.1.1 Grammar statistics

The Unimodal GoDiS Grammar Library consists of 146 grammar files, summing up to around 3500 lines of GF code. Since one of our aims was to be as modular as possible, we ended up in a large number of quite small grammar files. The applications only consist of 16 grammar files each, the rest of the grammars are either shared among all applications, or are considered resources that can be used in future applications.

5.2 Concluding remarks

Working with a set of GF grammars to cover the natural language to semantics translation has made it easier to keep the interpretation modules up to date in all languages at the same time. If the coverage in one language changes the other language has to follow or it will not correspond to the abstract grammar that link them.

GF also makes it easy to create a corpus over what can be generated and recognized by the system. This makes it easy to spot any flaws or shortcomings right away and fix them.

The fact that we are now using a grammar, interpreting an entire utterance, makes it virtually impossible for the system to make interpretations mistakes on its own accord. Previously it has been very easy for the sentence “jag vill spela in ett program” (I want to record a/one movie) to be interpreted as a request for a recording (“spela in”) and a time to start (“ett”), while it is supposed to be simply a request for recording, as discussed in section 1.2.1.

Currently the development of the GoDiS dialogue plans and domain knowledge in TrindiKit and the development of the application grammars is still conducted separately and thus we have not achieved the same kind of integration as has been achieved for the various grammars involved in dialogue systems. Future research can involve using GF to write the actual GoDiS dialogue plans and the domain knowledge. This way one can make sure that all moves that can be performed by the system can be realized as natural language and that there are natural language expressions for all move sequences that the system should be able to understand.

With the framework we have created it should be straightforward to make new application grammars. All the new grammarian has to do is figure out what dialogue plans the system has, what objects the conversation participants can discuss and what relationships exist between them (i.e. what answers go with which questions and requests).

Appendix A

Downloading and installation instructions

A.1 Downloading instructions

The TALK Unimodal Grammar Library can be downloaded from `http://www.ling.gu.se/projekt/talk/software`.

A.1.1 Contents

The distribution contains a bundle consisting of the following directories:

GF_GoDiS The GF grammars. See chapter 4 for a more thorough description. The directory also contains scripts for compiling the application grammars into the GFCM and CFGM formats for use with the Embedded GF interpreter and GSL grammars for use with Nuance ASR.

trindikit4 Alpha version of TrindiKit4. TrindiKit4 is a new version of TrindiKit where system components can be distributed across several OAA agents. Together with a new concurrent control mechanism this architecture replaces the previous implementation of asynchronicity in TrindiKit. No documentation is given in this release. Used by GoDiS, DJ GoDIS and GoDiS deLux.

godis The GoDiS core dialogue system, consisting of the dialogue move engine (update and select modules), a control algorithm and GoDiS-specific resource interface definitions and datatype definitions. Used by the DJ GoDIS and GoDiS deLux applications.

djgodis The DJ GoDIS application, a dialogue system for controlling a mp3 player. See section 4.4.1 for a short description of the system's functionality.

delux The GoDiS delux application, which is a dialogue system for controlling lights. See section 4.4.3 for a short description of the system's functionality.

jars This directory contains compiled Java code needed to run the applications. The file `tkit_oaa.jar` consists of base classes for creating OAA agents. The file `tkit_inout_text.jar` contains InOutTextScore, a simple OAA agent and Trindikit4 module for text input/output. The file `tkit_nuance.jar` contains NuanceWrapper, which is a OAA wrapper agent for Nuance ASR and TTS. The file `gfc2java.jar` contains the GFAgent, which is used by the applications for translating between natural language and GoDiS dialogue moves.

The AgendaTalk and Video GoDiS applications for which there are grammars in the GF_GoDiS directory are not included in the distribution.
In addition the distribution contains:

Grammatical Framework v2.2 which is needed to compile the grammars to Nuance GSL format and the formats used by the Embedded GF Interpreter. It can also be downloaded from `http://www.cs.chalmers.se/~aarne/GF`.

A.2 Installation instructions

A.2.1 System requirements

The system has been tested on Linux and Windows. Any platform should work that is supported by SICStus, Java and OAA. To run the system with speech, Windows and a SoundBlaster compatible sound card is required.

SICStus Prolog is needed to run TrindiKit systems. SICStus prolog can be downloaded for evaluation at `http://www.sics.se/sicstus`.

Java 1.5 or later is needed to run the OAA agents written in Java. It can be downloaded from `http://java.sun.com/`.

OAA 2.3.0 or later is needed. OAA can be downloaded from `http://www.ai.sri.com/oaa`.

Nuance ASR and Nuance Vocalizer is needed to run the system in speech mode.

A.2.2 Installation and usage

The following binary executables must be in the user's PATH variable: `sicstus`, `java` and `gf`. On Windows these should be `sicstus.exe`, `java.exe` and `gf.exe`.
Set the environment variable OAA_HOME to the full path of the directory containing the OAA distribution (e.g. `/home/david/oaa2.3.0`).
If using Nuance, create a file called `nuance-license.txt` containing the Nuance license code and put it in each application directory (`djgodis` and `delux`).
Enter either of the directories `delux` or `djgodis`. The script `run.bat` (`run.sh` for Linux/Unix) launches the OAA Startit agent which is used for running the system. Select a configuration from the Projects menu of the Startit agent to run the system in text-mode or speech mode. Click on the blue start button. Start speaking, or type into the text field that appears.

A.3 Testing the Unimodal GF Grammar Library

The grammars can be tested separately by loading them into GF. The relevant concrete syntaxes are:

Application	Grammar	Concrete syntax
DJ GoDiS	user	usr_domain_player_english.gf
		usr_domain_player_svenska.gf
		usr_domain_player_sem.gf
	system	sys_domain_player_english.gf
		sys_domain_player_svenska.gf
		sys_domain_player_sem.gf
GoDiS deLux	user	usr_domain_lamps_english.gf
		usr_domain_lamps_svenska.gf
		usr_domain_lamps_sem.gf
	system	sys_domain_lamps_english.gf
		sys_domain_lamps_svenska.gf
		sys_domain_lamps_sem.gf
Video GoDiS	user	usr_domain_video_english.gf
		usr_domain_video_svenska.gf
		usr_domain_video_sem.gf
	system	sys_domain_video_english.gf
		sys_domain_video_svenska.gf
		sys_domain_video_sem.gf
AgendaTalk	user	usr_domain_agenda_english.gf
		usr_domain_agenda_svenska.gf
		usr_domain_agenda_sem.gf
	system	sys_domain_agenda_english.gf
		sys_domain_agenda_svenska.gf
		sys_domain_agenda_sem.gf

A.3.1 Testing the grammars within GF

The following is an example of the capabilities of the GF program. For more information about how to use GF, see the documentation on `http://www.cs.chalmers.se/~aarne/GF`. This example assumes we are testing the DJ GoDiS user grammar, which of course can be replaced by any of the other grammars in the library.

1. Start GF in the directory where the grammars are located:

```
$ cd GF_GoDiS/Domain/DJGoDiS/User/
$ gf
```

2. Load the source module(s) into GF:

```
> i -conversion=finite usr_domain_player_english.gf
> i -conversion=finite usr_domain_player_svenska.gf
> i -conversion=finite usr_domain_player_sem.gf
```

The option `-conversion=finite` compiles away finite dependent types, which are used as described in section 2.3.3. Without this option the parser returns too many parse trees, which have to be filtered by the GF command `pt -transform=solve`.

3. Select the English concrete grammar:

```
> sf -lang=usr_domain_player_english
```

4. Parse an English utterance:

```
> p -cfg "shift the balance to the left"
requestCompounded speakerTask shift (makeAnswer speakerTask left)
```

The option `-cfg` selects an improved context-free parsing algorithm. The default parsing algorithm is overgenerating on grammars with dependent types, such as this one, and the resulting parse trees have to be filtered by `pt -transform=solve`.

5. Translate (i.e. parsing followed by linearization) from English to Swedish:

```
> p -cfg "shift the balance to the left" | l -all -lang=usr_domain_player_svenska
jag vill ändra balansen vänster tack / jag vill ändra balansen till vänster tack
/ jag vill skifta vänster tack / jag vill skifta till vänster tack / jag skulle
vilja ändra balansen vänster tack / jag skulle vilja ändra balansen till vänster
tack / jag skulle vilja skifta vänster tack / jag skulle vilja skifta till vänster
tack / jag vill ändra balansen vänster / jag vill ändra balansen till vänster /
jag vill skifta vänster / jag vill skifta till vänster / jag skulle vilja ändra
balansen vänster / jag skulle vilja ändra balansen till vänster / jag skulle vilja
skifta vänster / jag skulle vilja skifta till vänster / ändra balansen vänster tack
/ ändra balansen till vänster tack / skifta vänster tack / skifta till vänster tack
/ ändra balansen vänster / ändra balansen till vänster / skifta vänster / skifta
till vänster
```

The option `-all` shows all possible variants of linearizing a syntax term.

6. Translate from English to GoDiS dialogue moves:

```
> p -cfg "shift the balance to the left" | l -lang=usr_domain_player_sem
[request(set_balance),answer(-1.0)]
```

7. Generate 5 random Swedish utterances:

```
> gr -number=5 | l -lang=usr_domain_player_svenska
in the city med eagle eye cherry
rant radio
va
jag vill ändra balansen mitten tack
jag vill spela nummer tre tack
```

8. Quitting GF:

```
> q
```

A.3.2 Using the Embedded GF Interpreter

Producing the grammars

The GF interpreter needs two representations of the grammar to do linearization and parsing. These two representations can be generated from a GF source grammar by using the GF system. This example assumes that we use the GoDiS deLux system grammars.

1. Start GF:

   ```
   $ cd GF_GoDiS/Domain/deLux/System/
   $ gf
   ```

2. Load all the source modules into GF:

   ```
   > i sys_domain_lamps_english.gf
   > i sys_domain_lamps_svenska.gf
   > i sys_domain_lamps_sem.gf
   ```

3. Create a GFCM file (for linearization):

   ```
   > pm -utf8 -utf8id -printer=header | wf sys_lamps.gfcm
   ```

   The command `pm` prints multiple grammars in the format specified by the `-printer` flag, and `wf` writes to the specified file.

4. Create a CFGM file (for parsing):

   ```
   > pm -utf8 -utf8id -printer=cfgm | wf sys_lamps.cfgm
   ```

5. Create a properties file (here `sys_lamps.properties`) so that the interpreter can find these files. The file should have these contents:

   ```
   name: sys_lamps
   gfcm: sys_lamps.gfcm
   cfgm: sys_lamps.cfgm
   ```

Running the GF agent

If the grammar properties file is `sys_lamps.properties` and the facilitator is running on $FAC_HOST, port $FAC_PORT, the GF agent is started with:

```
$ java -cp $CLASSPATH:gfc2java.jar:. se.chalmers.cs.gf.oaa.GFAgent \
    sys_lamps.properties -oaa_connect "tcp('${FAC_HOST}',${FAC_PORT})"
```

A.3.3 Producing speech recognition grammars

The GF system is used to produce speech recognition grammars from GF grammars. This example assumes that we want to produce a Nuance GSL grammar from the English Video GoDiS user grammar.

1. Start GF:

   ```
   $ cd GF_GoDiS/Domain/VideoGoDiS/User/
   $ gf
   ```

2. Load the source module into GF:

   ```
   > i usr_domain_video_english.gf
   ```

3. Create a GSL grammar (here in the file `usr_video_english.grammar`):

   ```
   > pg -printer=gsl | wf usr_video_english.grammar
   ```

   The command `pg` prints a single grammar in the format specified by the flag `-printer`. To create a JSGF grammar, use `pg -printer=jsgf` instead.

Bibliography

Alshawi, H. (1992). *The Core Language Engine*. MIT Press, Cambridge, Ma.

Bender, E., Flickinger, D., and Oepen, S. (2002). The grammar matrix: an open-source starter-kit for the rapid development of cross-linguistically consistent broad-coverage precision grammars. In *Workshop on Grammar Engineering and Evaluation at 19th International Conference on Computational Linguistics*, pages 8–14, Taipei, Taiwan.

Bringert, B. (2005). Embedded GF Interpreter Java API. `http://www.cs.chalmers.se/~bringert/gf/gf-java.html`.

Burke, D. A. and Johannisson, K. (2005). Translating formal software specifications to natural language — a grammar-based approach. To be published in proceedings of LACL'05.

Butt, M., Frost, M., King, T. H., and Kuhn, J. (2003). The feature space in parallel grammar writing. In Bender, E., Flickinger, D., Fouvry, F., and Siegel, M., editors, *Workshop on Ideas and Strategies for Multilingual Grammar Development*, pages 9–16, Vienna, Austria.

Copestake, A. and Flickinger, D. (2000). An open-source grammar development environment and broad-coverage english grammar using hpsg. *Proceedings of the Second conference on Language Resources and Evaluation (LREC-2000)*.

de Groote, P. (2001). Towards abstract categorial grammars. In *39th Meeting of the Association for Computational Linguistics*, Toulouse, France.

Dowding, J., Hockey, B. A., Gawron, J. M., and Culy, C. (2001). Practical issues in compiling typed unification grammars for speech recognition. In *Meeting of the Association for Computational Linguistics*, pages 164–171.

Forsberg, M., Johannisson, K., Khegai, J., and Ranta, A. (2005). GF Gramlets. `http://www.cs.chalmers.se/~krijo/gramlets.html`.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification*. Sun Microsystems, Inc., third edition. Proposed third edition: `http://java.sun.com/docs/books/jls/java_language-3_0-mr-spec.zip`.

GSL (2003). *Nuance Speech Recognition System 8.5: Grammar Developer's Guide*. Nuance Communications, Inc., Menlo Park, CA, USA.

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*, number 2306 in LNCS.

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.

Hunt, A. (2000). JSpeech Grammar Format. W3C Note.

JavaSpeech (1998a). *Java Speech API Programmer's Guide*. Sun Microsystems, Inc.

JavaSpeech (1998b). *Java Speech API Specification*. Sun Microsystems, Inc.

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 *CICLing*, pages 453–464.

Kiefer, B. and Krieger, H.-U. (2000). A context-free approximation of Head-Driven Phrase Structure Grammar. In *6th International Workshop on Parsing Technologies, IWPT2000*, pages 135–146.

Larsson, S. (2002). *Issue-based Dialogue Management*. PhD thesis, Göteborg University, Göteborg, Sweden.

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

Ljunglöf, P. (2004b). Functional chart parsing of context-free grammars. *The Journal of Functional Programming*, 14(6):669–680.

Martin, D. L., Cheyer, A. J., and Moran, D. B. (1999). The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1–2):91–128.

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

Moore, R. C. (1999). Using natural-language knowledge sources in speech recognition. In Ponting, K., editor, *Computational Modeling of Speech Pattern Processing*, pages 304–327. Springer Verlag.

Moore, R. C. (2000). Removing left recursion from context-free grammars. In *Proceedings of the first meeting of the North American chapter of the Association for Computational Linguistics*, pages 249–255. Morgan Kaufmann Publishers Inc.

Muskens, R. (2003). Language, lambdas, and logic. In Kruijff, G.-J. and Oehrle, R., editors, *Reosurce Sensitivity in Binding and Anaphora*, pages 23–54. Kluwer.

Nuance (2003). *Nuance Speech Recognition System 8.5: Introduction to the Nuance System*. Nuance Communications, Inc., Menlo Park, CA, USA.

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058.

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

Pollard, C. (2004). Type-logical HPSG. In *Formal Grammar Workshop*, Nancy, France.

Ranta, A. (2005). Modular Grammar Engineering in GF. *Research in Language and Computation*. To appear.

Rayner, M., Carter, D., Bouillon, P., Digalakis, V., and Wirén, M. (2000). *The Spoken Language Translator*. Cambridge University Press, Cambridge.

Rayner, M., Hockey, B. A., and Dowding, J. (2003). An open-source environment for compiling typed unification grammars into speech recognisers. In *EACL*, pages 223–226.

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

SRGS (2004). Speech recognition grammar specification version 1.0. W3C Recommendation.

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*.