

# Embedded Grammars

Björn Bringert

February 10, 2005

# Abstract

This work explores the use of grammars as integral parts of computer programs and presents a number of tools and methods which facilitate such integration. An embeddable interpreter for the Grammatical Framework (GF) grammar formalism, a compiler from GF grammars to speech recognition grammars, and methods for writing multimodal grammars in GF are described. It is then shown how these tools and methods can be used to build multilingual multimodal dialog systems and precise domain-specific machine translation systems for both spoken and written language.

# Acknowledgments

FIXME: write acknowledgments

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	Motivation . . . . .	8
1.2	Grammatical Framework . . . . .	9
<b>2</b>	<b>The Embedded GF Interpreter</b>	<b>11</b>
2.1	Introduction . . . . .	11
2.2	Abstract Syntax Terms . . . . .	12
2.3	Parsing . . . . .	12
2.3.1	Lexical Analysis . . . . .	12
2.3.2	CFG Language . . . . .	12
2.3.3	Chart Parsing . . . . .	13
2.3.4	Tree Building . . . . .	14
2.4	Type Annotation . . . . .	16
2.5	Linearization . . . . .	17
2.5.1	Canonical GF Term Language . . . . .	17
2.5.2	Linearization Values . . . . .	17
2.5.3	Operational Semantics . . . . .	17
2.5.4	Linearization Operator . . . . .	19
2.5.5	Unlexing . . . . .	20
2.5.6	Implemented Optimizations . . . . .	20
2.5.7	Future Work . . . . .	20
2.6	Translation . . . . .	21
2.7	Java API . . . . .	21
2.8	Typed Abstract Syntax Trees . . . . .	21
2.9	OAA Agent . . . . .	22
2.10	Future Work . . . . .	22
2.10.1	Flexible Lexing and Unlexing . . . . .	22
2.10.2	Syntax Editing . . . . .	22
2.10.3	Type Checking . . . . .	22
2.10.4	Computation . . . . .	23
2.10.5	Higher-order Abstract Syntax . . . . .	23
2.10.6	Reducing GFCM and CFGM File Size . . . . .	23
<b>3</b>	<b>Multimodal Grammars</b>	<b>24</b>
3.1	Introduction . . . . .	24
3.2	Parallel Multimodality . . . . .	24
3.2.1	Introduction . . . . .	24
3.2.2	Implementation in GF . . . . .	24

3.3	Integrated Multimodality . . . . .	25
3.3.1	Introduction . . . . .	25
3.3.2	Implementation in GF . . . . .	25
3.4	Future Work . . . . .	26
<b>4</b>	<b>Generating Speech Recognition Grammars</b>	<b>27</b>
4.1	Introduction . . . . .	27
4.2	Speech Recognition Grammar Formats . . . . .	27
4.3	Implementation . . . . .	27
4.4	Removing Non-speech Modalities . . . . .	28
4.5	Future Work . . . . .	28
4.5.1	Compilation to Finite State Networks . . . . .	28
4.5.2	Automatic Conversion to Spoken Language Forms . . . . .	29
4.5.3	Extended Start Category Concept . . . . .	29
<b>5</b>	<b>Building Dialog Systems with Embedded Grammars</b>	<b>30</b>
<b>6</b>	<b>Example Application: Translation Applet</b>	<b>32</b>
<b>7</b>	<b>Example Application: Automatic Speech Translation</b>	<b>34</b>
<b>8</b>	<b>Example Application: Hello World Dialog System</b>	<b>36</b>
8.1	Introduction . . . . .	36
8.2	Grammars . . . . .	36
8.2.1	System . . . . .	36
8.2.2	User . . . . .	37
<b>9</b>	<b>Example Application: Tram Information</b>	<b>38</b>
9.1	Introduction . . . . .	38
9.2	Grammar Overview . . . . .	38
9.3	Transport Network Grammar . . . . .	40
9.3.1	Generic Abstract Syntax for Transport Networks . . . . .	40
9.3.2	Generic Concrete Syntax for Transport Networks . . . . .	40
9.3.3	Göteborg Abstract Syntax . . . . .	40
9.3.4	Göteborg Concrete Syntaxes . . . . .	40
9.4	Multimodal Input Grammars . . . . .	41
9.4.1	Common Declarations . . . . .	41
9.4.2	Click Modality . . . . .	42
9.4.3	Speech Modality . . . . .	42
9.4.4	Indexicality . . . . .	43
9.5	Ambiguity . . . . .	44
9.6	Multimodal Output . . . . .	44
9.6.1	Abstract Syntax for Multimodal Output . . . . .	44
9.6.2	Map Drawing Concrete Syntax . . . . .	45
9.6.3	English Concrete Syntax . . . . .	45
9.7	Example Interaction . . . . .	46
9.8	Multilinguality . . . . .	47
9.9	Component Overview . . . . .	47
9.10	Application Limitations . . . . .	48
9.11	Future Work . . . . .	48

9.12	Conclusions . . . . .	48
<b>10</b>	<b>Grammar Development Tools</b>	<b>49</b>
10.1	Introduction . . . . .	49
10.2	Resource Grammars . . . . .	49
10.3	Grammar Visualization . . . . .	49
10.4	Future Work . . . . .	50
10.4.1	Grammar development IDE . . . . .	50
<b>11</b>	<b>Related Work</b>	<b>51</b>
11.1	Embedded Context-Free Grammars . . . . .	51
11.2	Attribute Grammars . . . . .	51
11.3	Combinatory Categorical Grammars and OpenCCG . . . . .	51
11.4	Multimodal Grammars . . . . .	51
11.5	Generation of Speech Recognition Grammars . . . . .	52
11.6	External Grammars Tools . . . . .	52
11.7	GF Gramlets . . . . .	52
11.8	Embedding the Full GF System . . . . .	52
11.9	Speech Translation . . . . .	52
<b>12</b>	<b>Future Work</b>	<b>53</b>
12.1	Non-human Interaction Grammars . . . . .	53
<b>13</b>	<b>Conclusions</b>	<b>54</b>
13.1	Contributions . . . . .	54
13.2	Usefulness of Embedded Grammars . . . . .	54
13.3	Software Availability and Licensing . . . . .	55
<b>A</b>	<b>Embedded GF Interpreter OAA Agent</b>	<b>59</b>
A.1	Introduction . . . . .	59
A.2	Running . . . . .	59
A.2.1	Notes . . . . .	59
A.3	Solvables . . . . .	59
A.3.1	parse . . . . .	60
A.3.2	linearize . . . . .	61
A.3.3	translate . . . . .	61
A.3.4	list_grammars . . . . .	62
A.3.5	list_languages . . . . .	62
<b>B</b>	<b>Hello World Dialog System Source Code</b>	<b>64</b>

# Abbreviations

ABNF	Augmented BNF
ATK	Application Toolkit for HTK
BNF	Backus-Naur Form
CCG	Combinatory Categorical Grammar
CFG	Context-free grammar
CFGM	Multilingual Context-Free Grammar
EBNF	Extended Backus-Naur Form
GF	Grammatical Framework
GFC	Canonical GF
GFCM	Multilingual Canonical GF
GPL	GNU General Public License
GSL	Nuance Grammar Specification Language
ICL	Interagent Communication Language
IDE	Integrated Development Environment
JSAPI	Java Speech API
JSGF	JSpeech Grammar Format
LBNF	Labeled Backus-Naur Form
MCFG	Multiple CFG
OAA	Open Agent Architecture
SRG	Speech Recognition Grammar
SRGS	Speech Recognition Grammar Specification
XML	Extensible Markup Language

FIXME: make sure abbreviations list is complete

# Chapter 1

## Introduction

By *embedded grammars*, we refer to grammars which are used as parts of larger applications. A grammar is a specification of the values belonging to a language. Grammars can be used to describe both natural languages, such as those spoken and written by humans, and formal languages such as computer programming languages, document markup languages, mathematical language and many others. One may even use grammars to describe what many would not even consider languages, for example drawings and gestures, as long as it is possible to construct a string encoding of the values in the language. Thus, any computer program which gets input or produces output in some language could make use of embedded grammars.

We present a system for using embedded grammars written in the *Grammatical Framework* (GF) formalism, some methods and tools for using such grammars, and some example applications which use embedded grammars.

This introduction chapter gives some motivation for why embedded grammars are useful and a short introduction to the GF formalism. The following chapter details the main product of this thesis: the Embedded GF Interpreter. We then describe a method for writing multimodal grammars in GF. In chapter 4, the transformation of GF grammars to speech recognition grammars is described. We then give some overview of how to construct dialog systems using the tools presented in this thesis and describe a few example applications which we have developed. We then mention some tools which can be used to aid the development of GF grammars for use in systems such as those described earlier. Finally, some related work is discussed.

This work has been funded by the EU TALK project, IST-507802.

### 1.1 Motivation

Many existing dialog systems and other systems with natural language interfaces employ *statistical language models* (SLMs) for speech recognition and *phrase spotting* for extracting meaning from utterances. This gives robust systems which may be able to accept and extract some meaning even from utterances which fall outside of what the system has been explicitly constructed for. On the other hand, such systems are often not as accurate as one may wish for when presented with utterances within its intended coverage.



Grammars, on the other hand, may be used to get precise semantics for everything within their coverage, but tend to fail completely for unexpected inputs. One conclusion that one might draw is that grammars are suitable for domain-specific or otherwise limited systems. Another approach is to construct hybrid systems which fall back to SLMs and phrase spotting when the grammar fails.

Many existing speech recognition systems support the use of simple context-free grammars for guiding the recognition. On the other hand, parsers which are intended to capture the semantics of a recognized utterance often use a grammar more suited to that task, as opposed to the simpler recognition problem. This means that a complete system may involve one grammar for the speech recognizer, and a separate grammar for parsing. These grammars should ideally be equivalent in their coverage, and must at least be kept in sync during development. It would seem to be more efficient if this could be done automatically, for example by generating one of the grammars from the other.

In addition to parsing, grammars can also be used for *linearization*, the generation of (natural) language strings from some semantic description. Support for *linearization* is needed in order to do natural language output.

## 1.2 Grammatical Framework

Grammatical Framework (GF) is a grammar formalism based on Martin-Löf type theory [1]. This section is a very short introduction to GF; Aarne Ranta's *JFP* article [2] and the introduction chapter of Peter Ljunglöf's PhD thesis [3] give much more information.

GF makes an essential distinction between *abstract syntax* and *concrete syntax*. The abstract syntax represents the structure or meaning of values in the language, whereas the concrete syntax describes their appearance. The idea is that the abstract syntax is easy to analyze and synthesize in a program, and thus does not contain any irrelevant details or redundancy. The abstract syntax might also be shared between grammars for different languages. The concrete syntax, on the other hand, might be designed for readability or redundancy, or in the case of natural languages, evolved rather than designed. The distinction between abstract and concrete syntax is much used in the field of computer languages, but less so within computational linguistics.

Concrete syntaxes are written as linearization rules for abstract syntax terms. In other words, the grammar writer defines how each function in the abstract syntax is converted to a value in the concrete syntax. Using a concrete syntax, the GF system can both parse input to abstract syntax terms and linearize abstract syntax terms to values in the concrete syntax.

In GF there may be several concrete syntaxes for a single abstract syntax. This may be used to produce output or accept input in any of a number of languages, or to translate between languages. Formal and natural languages might share a common abstract syntax. This can for example be used to translate specifications in some formal language to informal natural language specifications [4].

Another application of multilingual grammars is multilingual syntax editing, where the user edits a document in multiple languages simultaneously by manipulating abstract syntax terms and observing the resulting concrete syn-

tax output for many languages in parallel. This may involve several natural languages [5], or both natural and formal languages [6].

GF is a suitable grammar formalism for general embedded grammars since it is quite powerful while being parsable in polynomial time [3], and since it can be used for both parsing and linearization. As is shown in chapter 4, other necessary grammars such as context-free grammars for external systems can be generated from GF grammars.

## Chapter 2

# The Embedded GF Interpreter

### 2.1 Introduction

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 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 an excellent environment 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 [7]. 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.

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 250 kilobytes and it should run on any platform which has a Java 1.5.0 Runtime Environment.

Following C.A.R. Hoare's maxim that "premature optimization is the root of all evil", the main principle guiding the implementation of the Embedded GF Interpreter has been to take the simplest and most understandable route available. After testing has revealed bottlenecks which lead to unacceptable performance, selected parts have been optimized.

## 2.2 Abstract Syntax Terms

Abstract syntax terms are:

- function applications of the form  $f A_1 \dots A_n$ , where  $f$  is a function and  $A_1 \dots A_n, n \geq 0$  are abstract syntax terms
- string constants
- integer constants
- meta-variables, written as ?

Full GF also supports lambda abstractions and bound variables in abstract syntax. This is not yet implemented in the Embedded GF interpreter.

## 2.3 Parsing

The parser computes a set of abstract syntax trees for a given string input. In the full GF system, parsing is done by using a context-free grammar in which the productions have labels and profiles [3]. GF first computes a Multiple Context-Free Grammar (MCFG) from a GF grammar. The MCFG is then converted to a context-free grammar (CFG). The Embedded GF Interpreter makes use of a CFG produced by the full GF system.

### 2.3.1 Lexical Analysis

The first step in parsing input is to divide it into tokens. A simple hard-coded lexer divides the input into simple words (non-empty sequences of letters and digits), quoted strings and punctuation. The full GF system allows flags which specify which of a number of hard-coded lexers and unlexers to use. The Embedded GF Interpreter does not support these flags, see section 2.10.1.

### 2.3.2 CFGM Language

The GF system is used to create a context-free grammar with labeled productions and *profiles* from a GF grammar. The general form of a production is:

$$f : C[p] \rightarrow \bar{s}$$

The function  $f$  is the name of the function that will be used to construct an abstract syntax term for the production. The right-hand side  $\bar{s}$  is a sequence of terminals and non-terminals. The profile  $p$  is a sequence of sets of integers which determine in which position of the function application each sub-term will be put. The  $n$ :th set in the profile contains the indexes of the non-terminals on the RHS which correspond to the  $n$ :th argument to the function  $f$ . In case of *suppression*, the set for the corresponding function argument will be empty. A set with more than one member means that that function argument is *reduplicated*.

### 2.3.3 Chart Parsing

The parser is a Kilbury bottom-up chart parser, similar to the chart parser described by Peter Ljunglöf [8]. The algorithm has been modified to support empty rules and to be better suited to implementation in an imperative language.

Below,  $S$  is the start symbol of the grammar,  $A$  and  $B$  are arbitrary non-terminals,  $\alpha$ ,  $\beta$  and  $\gamma$  are sequences of zero or more terminals or non-terminals,  $t$  is a terminal,  $f$  is an abstract syntax function, and  $p$  is a profile.

First the CFGM grammar is transformed so that all rules with terminals on the RHS have only a single terminal as the RHS. This is done by replacing all terminals in other rules by a category which has a single production with the terminal as the RHS.

The grammar rules are divided into three sets:

- Rules of the form  $f : A[p] \rightarrow B_1 \dots B_n$ , where  $n \geq 1$  and  $B_1 \dots B_n$  are all non-terminals.
- Rules of the form  $f : A[p] \rightarrow \epsilon$ .
- Rules of the form  $f : A[p] \rightarrow t$ , where  $t$  is a terminal. There are three kinds of terminal rules:
  - Constant terminal rules, which match a given string. This can be compared to keywords in programming language grammars.
  - String literal rules which match any token which is not an integer literal or matched by any constant terminal rule.
  - Integer literal rules, which match tokens consisting entirely of decimal digits.

Given a sequence of  $n$  input tokens  $t_k, 1 \leq k \leq n$ , the algorithm constructs the sets  $\Delta_{i,j}$  where  $0 \leq i \leq n$  and  $i \leq j \leq n$ . This collection of sets is called a *parse chart*. The sets  $\Delta_{i,j}$  contain *dotted rules*, which are grammar rules with a dot indicating how much of the RHS has been matched. We refer to a dotted rule  $A \rightarrow \alpha \bullet \beta$  in some set  $\Delta_{i,j}$  as an *edge* between nodes  $i$  and  $j$ . Such an edge means that the input sub-sequence consisting of tokens  $t_{i+1}$  through  $t_j$  has been matched by  $\alpha$ . An *active edge* is a dotted rule which does not have the dot at the end of the RHS. A *passive edge* has the dot at the end, and its rule has thus been fully matched.

#### Parsing Algorithm

1. Let  $k := 0$ .
2. (*Scan*) If  $k \geq 1$ , then for all rules  $f : A[p] \rightarrow t_k$ , add  $f : A[p] \rightarrow t_k \bullet$  to  $\Delta_{k-1,k}$ .
3. For all empty rules  $f : A[p] \rightarrow \epsilon$ , add  $f : A[p] \rightarrow \bullet$  to  $\Delta_{k,k}$ .
4. For all passive edges  $f' : C[p'] \rightarrow \alpha \bullet$  in  $\Delta_{j,k}$  where  $0 \leq j \leq k$ :
  - (a) (*Bottom-up predict*) For all rules  $f : A[p] \rightarrow C\beta$ , add edge  $f : A[p] \rightarrow C \bullet \beta$  to  $\Delta_{j,k}$ .

- (b) (*Combine*) For all active edges  $f : A[p] \rightarrow \beta \bullet C\gamma$  in  $\Delta_{i,j}$ , where  $0 \leq i \leq j$ , add edge  $f : A[p] \rightarrow \beta C \bullet \gamma$  to  $\Delta_{i,k}$ .
- 5. Repeat step 4 until no new edges are added.
- 6. If  $k < n$ , then let  $k := k + 1$  and go to step 2.
- 7. Parsing was successful if  $f : S[p] \rightarrow \alpha \bullet \in \Delta_{0,n}$ .

The algorithm uses a bottom-up prediction step in order for it to build all sub-parses even if there is no complete parse. This feature is currently not used, but in the future it could be exploited to produce at least some result even if the entire input cannot be understood. This could for example be useful when combined with syntax editing. The user could type in a phrase which, if not parsed completely, would produce some abstract syntax terms for parts of the input. Using syntax editing, these terms could then be used to construct the full term which the user intended. Another use might be in a dialog system which can ask for clarification of the parts which it did not understand.

### Implemented Optimizations

The *Predict* step is optimized by keeping a multimap which maps categories to all the rules which start with that category.

### Future Work

The parsing algorithm and its implementation could potentially be optimized in a number of ways:

- The *Scan* step is currently implemented using linear search through the terminal rules. For a grammar with a large number of terminal productions, using a trie to look up terminal productions would improve the algorithmic complexity of this step.
- Each  $\Delta_{i,j}$  set could be split into two sets: one for passive and one for active edges. This could make iteration through the sets faster, as no step uses both active and passive edges.
- In step 3 above, edges for all empty rules are added to each of the  $n + 1$  sets  $\Delta_{k,k}$ . The set of such edges is the same for each node and it is independent of the input tokens. Thus this set could be precomputed and shared between the nodes and between invocations of the parsing algorithm.

### 2.3.4 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 parsing algorithm described above. Tree building is done by the mutually recursive functions *buildTree* and *buildTrees* and gives as a result a set of parse trees. To build the set of abstract syntax trees for the entire input we use *buildTree*( $S, 0, n, \emptyset$ ), where  $S$  is the start category and  $n$  is the number of input tokens.

In order to avoid non-termination with circular 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.

### The buildTree Function

The function *buildTree* builds the set of abstract syntax trees in category  $C$  for the input between nodes  $i$  and  $j$ , i.e. for the input consisting of tokens  $t_{i+1}$  through  $t_j$ .

*buildTree*( $C, i, j, used$ ):

1. Let  $T := \emptyset$
2. For each passive edge  $E = f.C[p] \rightarrow \alpha \bullet \in \Delta_{i,j}$ ,
  - (a) If  $\alpha = \epsilon$ , then let  $T := T \cup \{f ?_1 \dots ?_n\}$ , where  $?_1 \dots ?_n$  are fresh meta-variables and  $n = length(p)$ , i.e. the arity of  $f$ .
  - (b) If  $\alpha = t$ , where  $t$  is a string or integer literal terminal, then let  $T := T \cup \{t_j\}$ . FIXME: why is  $f$  not used?
  - (c) If  $\alpha = t$ , where  $t$  is a constant terminal, then let  $T := T \cup \{f\}$ .
  - (d) If  $\alpha = A_1 \dots A_k$  and  $E \notin used$ , then:
    - i. Let  $used' := used \cup \{E\}$ .
    - ii. For each  $s \in buildTrees(\alpha, i, j, used')$ :
      - A. Let  $s' := s$  with terminals removed.
      - B. If there are  $c_1 \dots c_n$  such that  $n = length(p)$  and  $c_k = unifyAll(\{s'[x] | x \in p[k]\})$ , then let  $T := T \cup \{f c_1 \dots c_n\}$ .
3. Return  $T$ .

### The buildTrees Function

The function *buildTrees* builds a set of sequences of abstract syntax trees for a non-empty sequence of categories  $\langle C_1 \dots C_n \rangle$ , where  $n \geq 1$ , using tokens  $t_{i+1}$  through  $t_j$ .

*buildTrees*( $\langle C_1 \dots C_n \rangle, i, j, used$ ):

1. If  $n = 1$ , then return  $\{\langle h \rangle \mid h \in buildTree(C_1, i, j, used)\}$
2. If  $n > 1$ , then:
  - (a) Let  $R := \emptyset$ .
  - (b) For each  $k$  such that  $i \leq k \leq j$ :
    - i. Let  $H = buildTree(C_1, i, k, used')$ , where  $used' = used$  if  $k = j$  and  $used' = \emptyset$  otherwise.

- ii. Let  $T = \text{buildTrees}(\langle C_2 \dots C_n \rangle, k, j, \text{used}')$ , where  $\text{used}' = \text{used}$  if  $k = i$  and  $\text{used}' = \emptyset$  otherwise.
  - iii. Let  $R := R \cup \{ \langle h, t_1 \dots t_n \rangle \mid h \in H, \langle t_1 \dots t_n \rangle \in T \}$ .
- (c) Return  $R$ .

### Unification

The unification function used in *buildTree* is defined below. In the function definitions,  $?$  is a fresh meta-variable. Note that the unification function is partial.

$$\begin{aligned}
 \text{unifyAll}(\emptyset) &= ? \\
 \text{unifyAll}(\{x\} \cup T) &= \text{unify}(x, \text{unifyAll}(T)) \\
 \\ 
 \text{unify}(?, y) &= y \\
 \text{unify}(x, ?) &= x \\
 \text{unify}(f \ a_1 \dots a_n, f \ b_1 \dots b_n) &= f \ \text{unify}(a_1, b_1) \dots \text{unify}(a_n, b_n) \\
 \text{unify}(s, s) &= s \\
 \text{unify}(i, i) &= i
 \end{aligned}$$

### Implemented Optimizations

In order to speed up the iteration through the passive edges in step 2 of *buildTree*, the edge sets in the chart keep the edges indexed by their LHS category.

### Future Work

The tree building algorithm produces all parse trees at once. Since the number of parse trees may be very large, and we may only be interested in some of them, this can be inefficient. Peter Ljunglöf describes a more efficient way of handling parse forests [3] which could be implemented also in the Embedded GF Interpreter.

## 2.4 Type Annotation

All meta-variables are annotated with their type. This is a prerequisite for syntax editing. Note that the type annotation only uses non-dependent types, and thus all types are categories. We write a meta-variable  $?$  as  $? : C$  when annotated with the type  $C$ . Type annotation of the abstract syntax term  $T$  in category  $C$  is done by the function  $\text{ann}(T, C)$ , shown below.

$$\begin{aligned}
 \text{ann}(f \ A_1 \dots A_n, C) &= f \ \text{ann}(A_1, C_1) \dots \text{ann}(A_n, C_n) \\
 \text{ann}(?, C) &= ? : C \\
 \text{ann}(s, C) &= s \\
 \text{ann}(i, C) &= i
 \end{aligned}$$

Above,  $f$  has the type  $C_1 \dots C_n \rightarrow C$ ,  $s$  is a string constant, and  $i$  is an integer constant.



## 2.5 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. This section only considers the sub-language of Canonical GF used for concrete and abstract modules. Canonical GF can also be used to describe resource modules, but these are not used directly in linearization.

This section defines the sets of GFC terms and values, a relation  $\Downarrow$  which evaluates terms to values and finally how these are used in the linearization of abstract syntax terms.

### 2.5.1 Canonical GF Term Language

The GFC term language supported by the Embedded GF Interpreter is defined by the EBNF grammar in figure 2.1. This grammar describes the abstract rather than the concrete syntax of GFC; see the LBNF [9] grammar for GFC included in the Embedded GF Interpreter source distribution for the complete grammar.

$$\begin{array}{lcl} \langle \text{Term} \rangle & ::= & \{ (\langle \text{Ident} \rangle = \langle \text{Term} \rangle)^* \} \\ & | & \mathbf{table} \{ \langle \text{Case} \rangle^* \} \\ & | & \langle \text{Term} \rangle . \langle \text{Ident} \rangle \\ & | & \langle \text{Term} \rangle ! \langle \text{Term} \rangle \\ & | & < \langle \text{Ident} \rangle \langle \text{Term} \rangle^* > \\ & | & \langle \text{Term} \rangle ++ \langle \text{Term} \rangle \\ & | & \mathbf{variants} \{ \langle \text{Term} \rangle^* \} \\ & | & [ ] \\ & | & \langle \text{Token} \rangle \\ & | & \langle \text{Variable} \rangle \end{array}$$

Figure 2.1: The set of GFC terms.

In figures 2.1 and 2.2, we use  $C^*$  to denote a possibly empty sequence of Cs. The  $\langle \text{Token} \rangle$  category contains ordinary string tokens and *prefix-dependent choice* tokens. Prefix-dependent choice is handled by the unlexer, but is not further described in this thesis. The language described above is a subset of the full GFC language as it does not support higher-order abstract syntax.

### 2.5.2 Linearization Values

The set of values to which abstract syntax terms can be linearized is shown in figure 2.2.

### 2.5.3 Operational Semantics

We define an evaluation relation  $\Downarrow$  for evaluating a GFC term to a value. The definition of  $\Downarrow$  is shown in figure 2.3.

$$\begin{array}{lcl}
\langle \text{Value} \rangle & ::= & [ \langle \text{Token} \rangle^* ] \\
& | & \mathbf{table} \{ \langle \text{Case} \rangle^* \} \\
& | & \{ (\langle \text{Label} \rangle = \langle \text{Value} \rangle)^* \} \\
& | & < \langle \text{CIdent} \rangle \langle \text{Value} \rangle^* >
\end{array}$$

Figure 2.2: The set of linearization values.

$$\begin{array}{c}
\text{RECORD CONSTRUCTION} \\
\frac{t_1 \Downarrow v_1 \quad \dots \quad t_n \Downarrow v_n}{\{l_1 = t_1; \dots; l_n = t_n\} \Downarrow \{l_1 = v_1; \dots; l_n = v_n\}} \\
\\
\text{TABLE CONSTRUCTION} \\
\frac{t_1 \Downarrow v_1 \quad \dots \quad t_n \Downarrow v_n}{\text{table } T \{ \overline{p_1} \Rightarrow t_1; \dots; \overline{p_n} \Rightarrow t_n \} \Downarrow \text{table } \{ \overline{p_1} \Rightarrow v_1; \dots; \overline{p_n} \Rightarrow v_n \}} \\
\\
\text{RECORD PROJECTION} \\
\frac{t \Downarrow \{ \dots; l = v; \dots \}}{t.l \Downarrow v} \\
\\
\text{TABLE PROJECTION} \\
\frac{t \Downarrow \text{table } \{ \dots; \langle \dots | p | \dots \rangle \Rightarrow v'; \dots \} \quad t' \Downarrow v \quad v \text{ matches } p}{t!t' \Downarrow v'} \\
\\
\begin{array}{cc}
\text{PARAMETER CONSTRUCTION} & \text{CONCATENATION} \\
\frac{t_1 \Downarrow v_1 \quad \dots \quad t_n \Downarrow v_n}{\langle i \ t_1 \dots t_n \rangle \Downarrow \langle i \ v_1 \dots v_n \rangle} & \frac{t \Downarrow [v_1, \dots, v_n] \quad u \Downarrow [w_1, \dots, w_m]}{t + +u \Downarrow [v_1, \dots, v_n, w_1, \dots, w_m]}
\end{array} \\
\\
\begin{array}{ccc}
\text{NON-EMPTY VARIANTS} & \text{EMPTY TOKEN LIST} & \text{TOKEN} \\
\frac{n \geq 1 \quad t_1 \Downarrow v}{\text{variants } \{t_1 \dots t_n\} \Downarrow v} & \frac{}{\overline{\square} \Downarrow \overline{\square}} & \frac{t \text{ is a token}}{t \Downarrow [t]}
\end{array}
\end{array}$$

Figure 2.3: Definition of the GFC term evaluation relation  $\Downarrow$ .

<p>FUNCTION APPLICATION</p> $\frac{\text{lin } f = \lambda x_0 \dots x_n \rightarrow t \quad t[x_0 = a_0^\circ; \dots; x_n = a_n^\circ] \Downarrow v}{(f \ a_0 \dots a_n)^\circ = v}$	
<p>INTEGER CONSTANT</p> $\frac{i \text{ is a integer} \quad n \text{ is the decimal string representation of } i}{i^\circ = \{s = [n]\}}$	
<p>STRING CONSTANT</p> $\frac{a \text{ is a string}}{a^\circ = \{s = [a]\}}$	<p>META-VARIABLE</p> $\frac{v \text{ is a string representation of } ? : T}{(? : T)^\circ = \{s = [v]\}}$

Figure 2.4: Definition of the linearization operator  $^\circ$ .**Note on Empty Variants**

Note that the evaluation relation is partial even for well-typed terms, since an empty variants term cannot be evaluated. This is intentional, as there can be abstract syntax terms which cannot be represented in a given concrete syntax. For example, in some languages, certain word forms do not exist, even though one might expect them to. In Swedish, certain adjectives such as *rädd* (“afraid”) do not have a neuter form, which makes some very reasonable constructions impossible.

**Note on Free Variation**

Note that the semantics that we give to free variation mean that we always choose the first alternative. This was done for simplicity and predictability. One possible design decision could have been to make the choice non-deterministic, or to make linearization ambiguous, i.e. to have linearization produce a set of values. In the latter case there are two alternative ways to handle linearization of reduplicated terms. In each linearization, a reduplicated term may be linearized using the same variant for each occurrence. Alternatively, each linearization could contain all combinations of the possible variants. Ljunglöf [3] refers to these interpretations as *intensional disjunction* and *extensional disjunction* respectively and argues that the latter constitutes a strict extension of GF by making the formalism more powerful. The full GF system interprets free variation as *extensional disjunction*.

**2.5.4 Linearization Operator**

We use  $a^\circ$  to denote the linearization of the abstract syntax term  $a$  using a given concrete syntax. The linearization operator  $^\circ$  is defined in figure 2.4. In the figure,  $t[x = v; \dots]$  means simultaneous substitution.

**Note on Meta-variables**

Note that the rule for meta-variables in figure 2.4 is not actually sufficient for all cases. If the linearization type of  $T$  is not  $\{s : Str\}$ ,  $(? : T)^\circ$  will have the

wrong type; see section 2.5.7 for some more information on this topic.

### 2.5.5 Unlexing

After linearization has produced a list of tokens, the *unlexer* joins the list to create a single output string. A naïve unlexer would simply concatenate the tokens, adding a space character between each token. 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, see section 2.10.1 for more details.

### 2.5.6 Implemented Optimizations

Concatenated strings are kept as lists of strings throughout the linearization algorithm to avoid the  $\Theta(n^2)$  behavior which can occur with left-associated concatenations.

### 2.5.7 Future Work

#### Unimplemented Linearization Features

These features of Canonical GF are not yet implemented in the Embedded GF Interpreter:

- No resource module functionality is implemented, since a resource module is not used for linearization without first producing a concrete module from it.
- The GFC language supports *wild-card patterns* in tables. However, since the current GF system always generates fully expanded tables, wild-card pattern support has not yet been implemented in the Embedded GF Interpreter.
- *Printnames* are used to give human-readable names to functions. This is useful mainly for syntax editing, which is not yet implemented.
- Tables without patterns, see section 2.10.6, are not implemented yet.
- Higher-order abstract syntax is currently not supported, see section [?].
- The linearization of meta-variables is incomplete, and there is no support for *lindef* judgements. Such judgements are used to declare a default linearization for a category, i.e. how meta-variables in that category are to be linearized.

### Further Optimizations

The linearization algorithm could be optimized by replacing parameters by integers. This can be done since a total ordering of all the parameter values in a parameter type can be easily constructed. This would allow tables to be represented as arrays instead of maps.

## 2.6 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.

## 2.7 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. The *parse* method takes a language name (the name of a concrete syntax) and a string, and returns a set of abstract syntax trees. The *linearize* method takes a language name and an abstract syntax tree and returns a string. 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. There are also versions of the parsing and linearization methods which try to use all available concrete syntaxes.

More documentation is available in the Embedded GF Interpreter API reference [10].

## 2.8 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 [11] 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());
```

## 2.9 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) [12] 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 interfaces 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 appendix A.

## 2.10 Future Work

### 2.10.1 Flexible Lexing and Unlexing

The hard-coded lexer and unlexer might be insufficient for some formal languages, or natural languages with certain lexical features. The full GF system has a similar problem, since while flags in the grammar can be used to specify which of the available hard-coded lexers and unlexers to use, there is no flexible way to give a specification of the lexical features in the grammar. Extending GF to allow specifying lexical rules could solve this problem. Such rules would have to be reversible to allow them to be used for both lexing and unlexing.

### 2.10.2 Syntax Editing

GF grammars can be used for syntax editing [13, 14]. In order to support this, some functionality for keeping track of the current position and for editing and navigation commands would have to be implemented.

### 2.10.3 Type Checking

GF abstract syntax terms can be dependently typed. Dependent types are not used when GF grammars are translated to context-free grammars for parsing. Instead, an over-generating grammar is created, and ill-typed terms may be discarded by type checking after parsing. We have made a preliminary

implementation of a slight variation on Thierry Coquand's algorithm for type-checking dependent types [15], but due to lack of time, this has not yet been implemented in the Embedded GF Interpreter.

#### 2.10.4 Computation

GF grammars may contain computation rules. This has not been implemented in the Embedded GF Interpreter yet. Having support for computation would allow more of the application to be put in the grammar and is necessary for a complete implementation of dependent type checking.

#### 2.10.5 Higher-order Abstract Syntax

In full GF, abstract syntax terms may have function types. Such terms are constructed using lambda expressions and can be used for binding constructs such as universal quantification. Such *higher-order abstract syntax* requires support in the GFC term and value languages and in the GFC interpreter. This has not yet been implemented in the Embedded GF Interpreter.

#### 2.10.6 Reducing GFCM and CFGM File Size

The GFCM and CFGM files produced by the GF system can become rather large for certain grammars. GFCM files are mainly made large because of the fact that all operations are fully evaluated and all tables are fully expanded, including fully qualified parameter patterns for each table entry. The category names in CFGM files are constructed from category, parameter and field names from the original GF grammars. This means that category names can become quite long. Certain GF grammar feature may also cause the size of the generated context-free grammar to explode [3]. For the reasons above, a large number of sub-strings of CFGM and GFCM files are repeated many times, making such files highly compressible. One approach to reducing the size of the files would thus be to compress them using some general purpose text compression algorithm. Another method could be to change the GFCM and CFGM formats so that they contain less redundant information. The full GF system implements such an optimization where tables can be produced completely without patterns. This can be done since the tables are always fully expanded and an ordering of the table parameters can be constructed. This optimization is optional in GF, and the grammars produced cannot yet be used by the Embedded GF Interpreter.

## Chapter 3

# Multimodal Grammars

### 3.1 Introduction

The most natural method of communication between humans is not always spoken natural language. For example, people may draw maps or point when asked to describe how to get to a certain place. Different modes of communication, such as speech, drawing, pointing are referred to as modalities. It will be shown in this chapter how GF grammars can be used to describe other modalities than speech, and how multimodal grammars can be written.

Since GF grammars can in themselves only describe strings, we use some string encoding for all non-string domains.

We distinguish between parallel and integrated multimodality. This distinction is due to Aarne Ranta [16].

### 3.2 Parallel Multimodality

#### 3.2.1 Introduction

We use the term *parallel multimodality* to describe a situation where all necessary information is available in each of several modalities. For example, driving directions given both as a written description of actions to take and as a route drawn on a map is an instance of parallel multimodality. Figure 3.1 shows an example of parallel multimodality where an abstract syntax term representing a route through a tram network is shown both as a natural language description and a drawing on a map.

#### 3.2.2 Implementation in GF

Parallel multimodality in GF is implemented in the same way as multilinguality. For a given abstract syntax, there is a separate concrete syntax for each modality. In the driving directions example, the abstract syntax would contain some abstract description of the route. There would be one concrete syntax in which the abstract route is linearized to written instructions. Another concrete syntax would create map drawings by using some drawing language. Section 9.6 describes an example of a GF grammar with parallel multimodality.



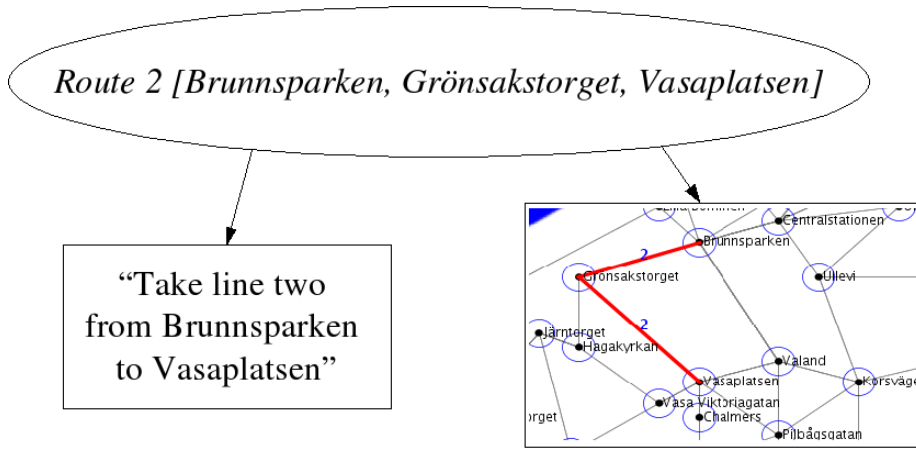


Figure 3.1: An example of parallel multimodality

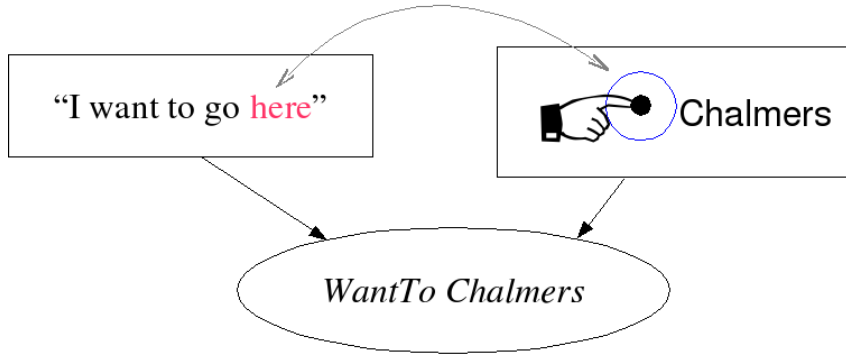


Figure 3.2: An example of integrated multimodality

### 3.3 Integrated Multimodality

#### 3.3.1 Introduction

In *integrated multimodality*, the information in each modality contributes to the total information. For example, the utterance “turn left at this intersection” combined with pointing to an intersection on a map gives the complete information. The information from each individual modality is insufficient. Figure 3.2 shows an example of integrated multimodality where a natural language phrase and a pointing gesture together represent an abstract syntax term.

#### 3.3.2 Implementation in GF

Integrated multimodality is implemented by exploiting the fact that GF linearization types are records. An abstract syntax category whose content we would like to represent with integrated multimodality is given a linearization type which is a record with one field for each modality. See section 9.4 for an example of a GF grammar using integrated multimodality.

### 3.4 Future Work

Currently only speech and click input, and speech and drawing output modalities have been tested. Possibilities for future experiments with other modalities include:

- More advanced drawing modality support (Petri Mäenpää and Greek mathematics)
- Drawing input, e.g. “I want to go to this area”.
- User location, “Switch off the light on my right”
- Gestures, “Open that door”.

## Chapter 4

# Generating Speech Recognition Grammars

### 4.1 Introduction

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 [3] format used by GF to some speech generation grammar formats has been implemented.

### 4.2 Speech Recognition Grammar Formats

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

**JSpeech Grammar Format (JSGF)** [17] Used in the Java Speech API (JS-API) [18, 19]. A plain-text language for context-free grammars.

**Nuance Grammar Specification Language (GSL)** [20] Used by the Nuance [21] speech recognizer. A plain-text language for context-free grammars.

**Speech Recognition Grammar Specification (SRGS)** [22] There are two equivalent syntactic forms, Augmented Backus-Naur Form (ABNF) and XML.

### 4.3 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 [23]. 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 (categories with no productions are often the result of removing non-speech modalities, see section 4.4). 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, but see section 4.5.2. 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 2.3), 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`.

## 4.4 Removing Non-speech Modalities

Since the speech recognition engine is only concerned with the speech modality, all other modalities should be removed from the grammar before producing a speech recognition grammar.

This is not an issue for parallel multimodality, since a speech recognition grammar will only be produced for the concrete syntaxes in the speech modality.

In the case of integrated multimodality, some more work is required. Since different modalities are encoded in record fields, we need a way to create a speech recognition grammar just for the speech field. This is achieved by adding new abstract and concrete module which extend the modules for which we wish to create a speech recognition grammar. With the new modules, we add a new (start) category, whose linearization type contains a single string field. The category contains a single function which takes an element of the start category of the original concrete syntax. The linearization of the function keeps just the speech field from the linearization type of the original start category.

## 4.5 Future Work

### 4.5.1 Compilation to Finite State Networks

Some speech recognition systems, for example ATK [24], use finite state networks instead of context-free grammars to guide speech recognition. It would be useful to be able to generate finite state approximations of GF grammars for use with such systems. Since finite-state automata are weaker than GF grammars and context-free grammars, the finite state approximation would have to be over-generating.

### 4.5.2 Automatic Conversion to Spoken Language Forms

The orthography of written language is not always optimal for representing spoken language. For example, punctuation in written language might correspond to various pauses or intonation changes. It might be possible to automatically transform for example punctuation in the given grammar to the corresponding features of the spoken language.

### 4.5.3 Extended Start Category Concept

If GF was extended to allow the start category to be not just a category, but optionally also a field of the linearization type of a category, stripping non-speech modalities could be done without adding any new modules.

## Chapter 5

# Building Dialog Systems with Embedded Grammars

Figure 5.1 illustrates the architecture of a multimodal dialog system built using the Embedded GF Interpreter. The Embedded GF Interpreter and the speech recognition grammar compiler are new components developed as part of this thesis. For speech recognition and speech synthesis, existing off-the-shelf components can be used. The following components are specific to the dialog system being created:

- Multimodal GF grammars for input and output.
- A *dialog manager*. The dialog manager can be a simple program written in a general-purpose programming language as in the examples shown later in this thesis, or it can be built using a framework such as Trindikit [25].
- Any domain resources for the specific domain (such as databases, sensors, actuators etc.).
- Any non-speech input devices and their wrappers.

Communication between the components of the system (the solid arrows in figure 5.1) may be done using normal method calls, or within some agent framework. As described in sections 2.7 and 2.9, there is a Java API and an OAA wrapper for the Embedded GF Interpreter.

Note that figure 5.1 does not show a multilingual system. In a multilingual system, there would be one speech recognition grammar, speech recognizer and speech synthesizer per language.

FIXME: make diagram larger on page? rotate?

FIXME: make diagram b/w in ps

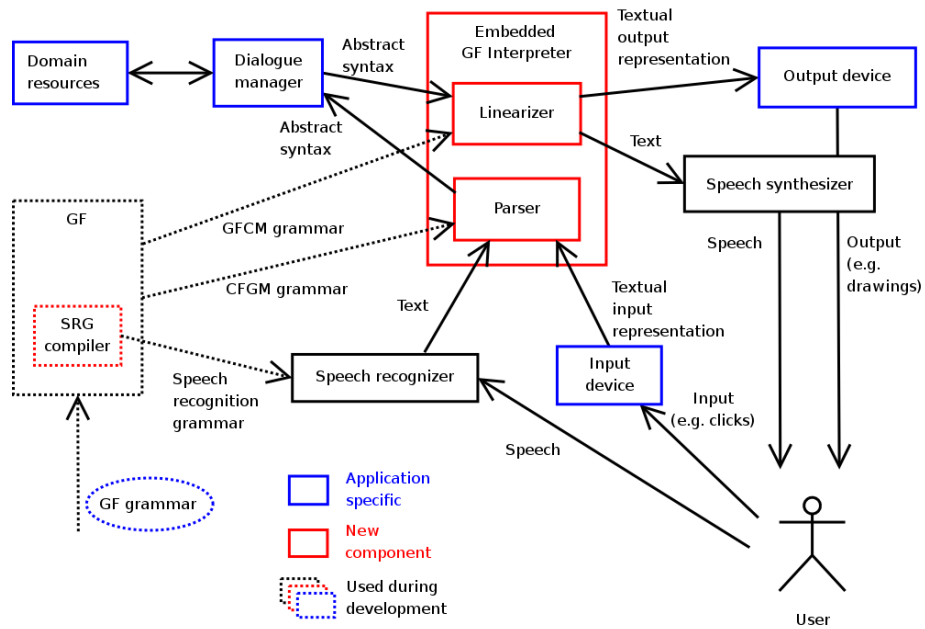


Figure 5.1: Architecture of a multimodal dialog system using the Embedded GF Interpreter

## Chapter 6

# Example Application: Translation Applet

The translation applet uses GF grammars to perform translation between a number of concrete syntaxes which use the same abstract syntax. The user can specify the input and output languages, or use all available languages.

The applet essentially takes user input and calls the translation function in the Embedded GF Interpreter to translate it to the desired output language(s). If the user opts to use all input languages, all the available parsers are tried. If all output languages are used, as list of linearizations is shown.

Figure 6.1 shows the translator applet being used with the numerals grammars [26].



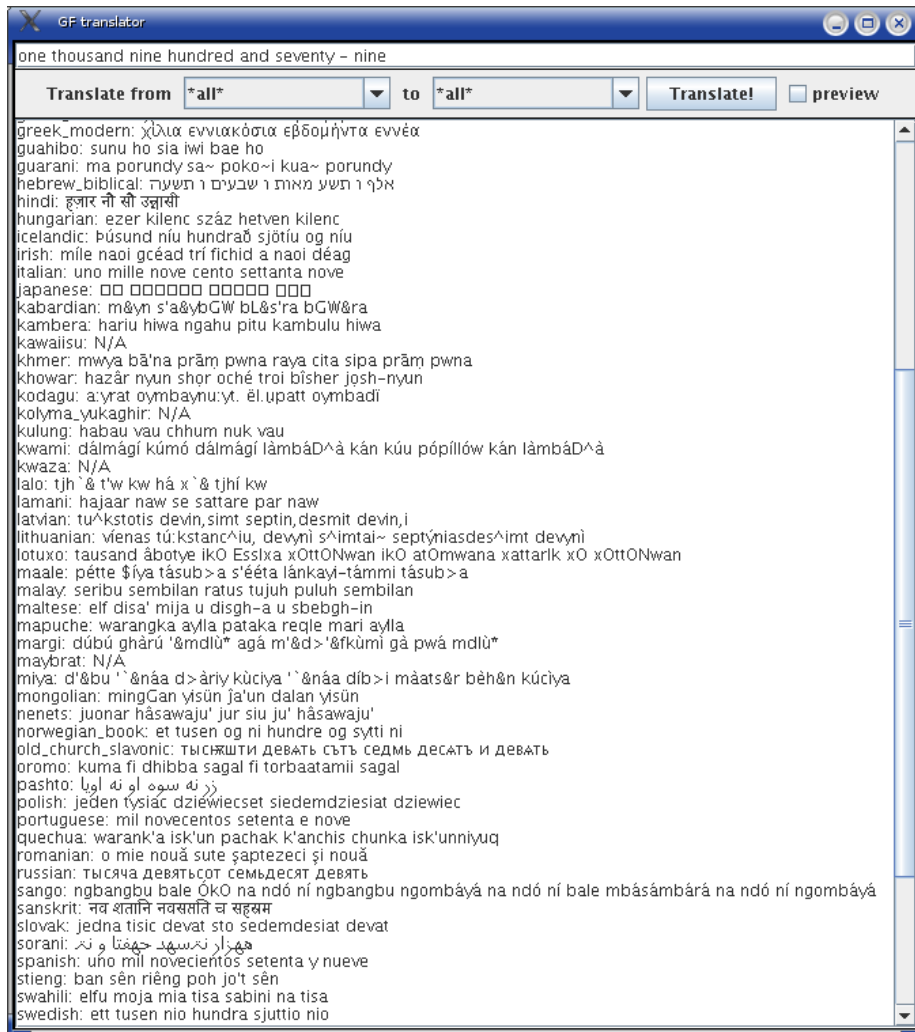


Figure 6.1: The GF translation applet using the numerals grammars.

## Chapter 7

# Example Application: Automatic Speech Translation

We have implemented a system for producing domain-specific automatic unidirectional speech translators. The application uses the translation feature of the Embedded GF Interpreter, along with off-the-shelf speech recognition and speech synthesis systems.

An overview of the architecture of the speech translator is shown in figure 7.1. As might be understood from comparing figures 7.1 and 5.1, a speech translator can be seen as a special case of a unimodal dialog system, where the dialog manager is the identity function and the output language is different from the input language.

To use the translation system for a new domain or language pair, all that is needed is a bilingual GF grammar, a speech recognizer for the input language and a speech synthesizer for the output language. The GF system and the speech recognition grammar compiler (see chapter 4) are used to create grammars for the speech recognizer, parser and linearizer. The output of the speech recognizer is translated using the translation function in the Embedded GF Interpreter and the output is fed to the speech synthesizer.

Implementing the speech translator required less than 100 lines of Java code, demonstrating the relative ease with which natural language applications can be developed when using embedded grammars. Of course the real complexity of any useful translator constructed using this system is in the grammars, but writing the grammars is essentially the same as specifying the translation function of the system. The developer need not worry about creating all the supporting infrastructure.

As fully general speech translation would seem to require artificial intelligence comparable to that of humans, this system cannot be expected evolve into a *Babel fish* [27]. However, for restricted domains where grammars with acceptable coverage can be written, the concept seems promising. So far the system has only been tested with very small grammars, and it might be dangerous to extrapolate from such experiments. For example, if the grammar is ambiguous it can be difficult for the system to present all possible translations in some

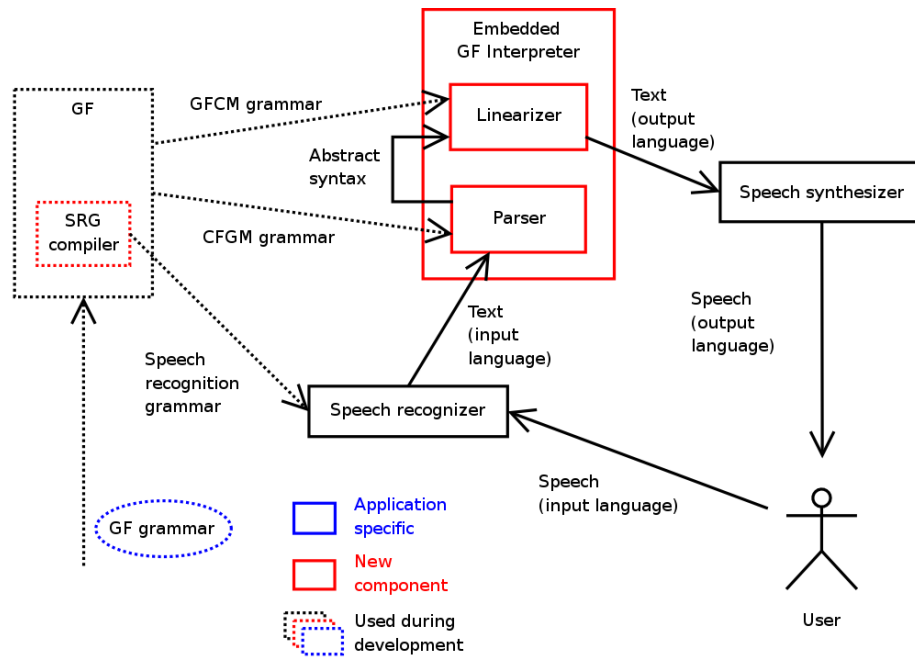


Figure 7.1: Architecture of the GF Speech Translator.

way understandable to the user, not to mention choosing the right alternative.

FIXME: make diagram larger on page? rotate?

FIXME: make diagram b/w in ps

## Chapter 8

# Example Application: Hello World Dialog System

### 8.1 Introduction

The Hello World dialog system is a very simple multilingual single-modality dialog system. The system accepts the speech inputs “Here you go” and “Thanks”, and responds with “Thanks” and “You are welcome”. The user may speak in English or Swedish, and the system responds in the same language as the user used.

The purpose of this system is to make a simple dialog system which is small enough to easily understood. The grammars are extremely basic, but the framework should be sufficient to develop a more sophisticated dialog system. For an example of a larger system, see chapter 9. The source code of the Hello World dialog system is shown in appendix B.

### 8.2 Grammars

#### 8.2.1 System

##### Abstract Syntax

```
abstract System = {  
  cat Output ;  
  fun Thanks : Output ;  
  fun Welcome : Output ;  
}
```

##### English Concrete Syntax

```
concrete SystemEng of System = {  
  flags startcat = Output ;  
  lincat Output = { s : Str } ;  
  lin Thanks = { s = "thanks" } ;  
  lin Welcome = { s = "you" ++ "are" ++ "welcome" } ;  
}
```

**Swedish Concrete Syntax**

```
concrete SystemSwe of System = {  
  flags startcat = Output ;  
  lincat Output = { s : Str } ;  
  lin Thanks = { s = "tack" } ;  
  lin Welcome = { s = "varsågod" } ;  
}
```

**8.2.2 User****Abstract Syntax**

```
abstract User = {  
  cat Input ;  
  fun HereYouGo : Input ;  
  fun Thanks : Input ;  
}
```

**English Concrete Syntax**

```
concrete UserEng of User = {  
  flags startcat = Input ;  
  lincat Input = { s : Str };  
  lin HereYouGo = { s = "here" ++ "you" ++ "go" } ;  
  lin Thanks = { s = "thanks" };  
}
```

**Swedish Concrete Syntax**

```
concrete UserSwe of User = {  
  flags startcat = Input ;  
  lincat Input = { s : Str };  
  lin HereYouGo = { s = "varsågod" } ;  
  lin Thanks = { s = "tack" };  
}
```

## Chapter 9

# Example Application: Göteborg Tram Information System (GOTTIS)

### 9.1 Introduction

The Göteborg Tram Information System (GOTTIS) is a demonstration of a multilingual multimodal dialog system. It finds the shortest path through (a subset of) the Göteborg public transportation network. User input consist of spoken queries along with optional clicks on a map of the transport network. The system responds with spoken instructions and drawings on the network map.

The system is easily adaptable to other transport networks or other systems which can be represented as weighted directed graphs.

The system uses multimodal GF 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.

### 9.2 Grammar Overview

The user and system grammars are split up into a number of modules in order to make reuse and modification simpler. Overviews of the query and answer grammar modules are shown in figures 9.1 and 9.2, respectively. These module graphs have been produced with a module dependency visualization tool developed as part of this thesis. See section 10.3 for a description of this tool and the meaning of the different kinds of nodes and edges in the graphs. The following sections show the details of the grammar modules.

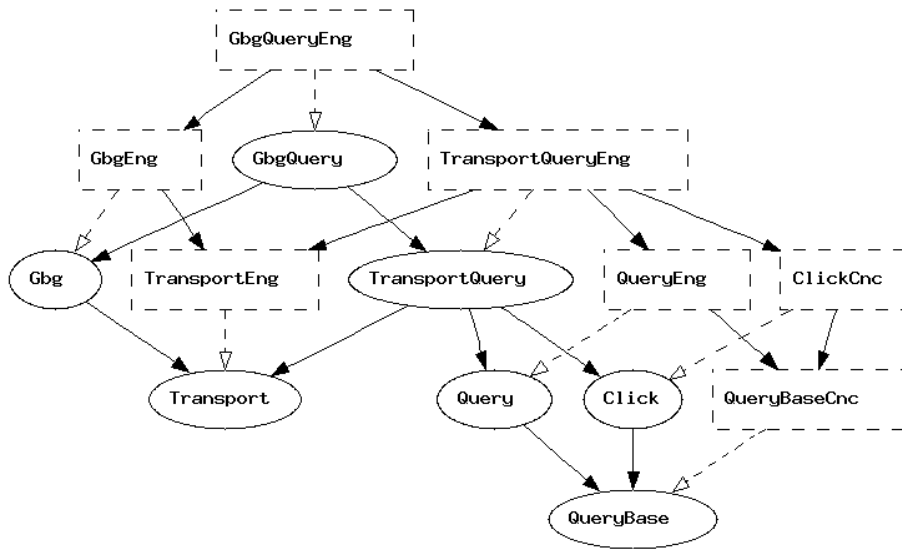


Figure 9.1: Query grammar modules

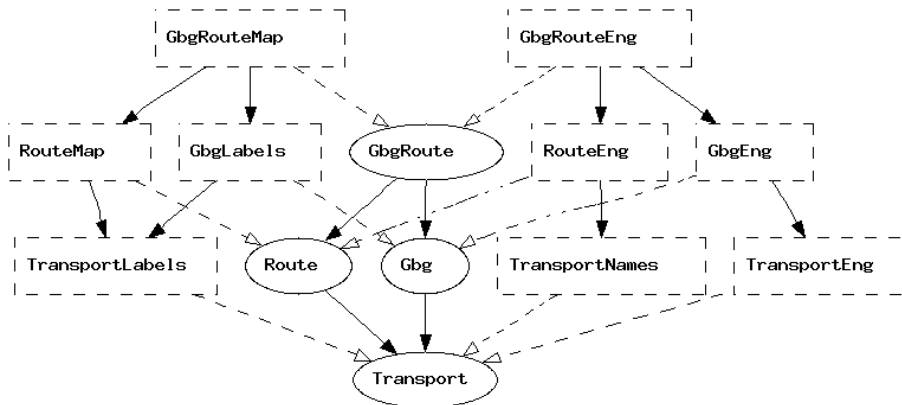


Figure 9.2: Answer grammar modules

### 9.3 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.

#### 9.3.1 Generic Abstract Syntax for Transport Networks

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

```
abstract Transport = {
  cat
  Stop ;
}
```

#### 9.3.2 Generic Concrete Syntax for Transport Networks

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 } ;
}
```

#### 9.3.3 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 ;
  ...
}
```

#### 9.3.4 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 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 [21], do not allow capitals in the recognized text [20]. 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 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 Bergsjon = { s = ["Bergsjon"] } ;
  lin Biskopsgarden = { s = ["Biskopsgarden"] } ;
  ...
}
```

## 9.4 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 9.4.4).

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

### 9.4.1 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 } ;
}

```

### 9.4.2 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 } ;
}

```

### 9.4.3 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" and 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 } ;
}

```

#### 9.4.4 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 brunnsparcken;”

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

```

  fun
    -- indexical "here", without a click
    PHere   : Place ;
    -- "want to come from a" (to where I am now)
    ComeFrom : Place -> 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
} ;

```

## 9.5 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 dialog management.

## 9.6 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.

### 9.6.1 Abstract Syntax for Multimodal Output

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
    -- leg followed by a route
    Then : Leg -> Route -> Route ;
    -- single leg
    OneLeg : Leg -> Route ;
    -- leg on a line
    LineLeg : Line -> Stops -> Leg ;
    -- line labelled by a string
    NamedLine : String -> Line ;
    -- stop followed by some stops
    ConsStop : Stop -> Stops -> Stops ;
    -- last two stops
    TwoStops : Stop -> Stop -> Stops ;
  }

```

### 9.6.2 Map Drawing Concrete Syntax

The map drawing language contains sequences of labeled 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 } ;
  }

```

### 9.6.3 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 } ;
  -- stop list is linearized to its first and last stops
  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 } ;
}

```

## 9.7 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)))))

```

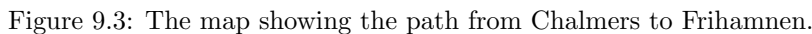
This is linearized to this speech output:

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

And to these drawing instructions:

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

The map with this output is shown in figure 9.3.



Currently, speech input and output in English and Swedish are implemented. The dialog system itself accepts input in either language, but speech recognizers can often only handle a single language at a time.

Adding support for a new language requires writing concrete syntaxes for the user and system grammars.

The application consists of a number of OAA [12] agents:

- Speech recognizer - The Nuance [21] speech recognizer using Nuance-Wrapper [28]. The speech recognition grammar is generated from the GF user grammar.
- Clickable map + Path drawing - An OAA agent written in Java
- Parser + Linearizer (multilingual and multimodal) - Embedded GF Interpreter through its OAA interface.
- Shortest path finder - OAA agent written in Java
- Speech synthesis - FreeTTS [29] over OAA using FreeTTSAgent [30]

## 9.10 Application Limitations

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

## 9.11 Future Work

Whereas we have only investigated the use of drawings for system output (and quite simple ones at that), one could also imagine using them for user input. In the example system, the user could for example circle a number of stops on the map to indicate a set of possible origin or destination stops.

## 9.12 Conclusions

The system presented in this paper makes the combination of information from different modalities part of the grammar, whereas many other systems do this in a fairly ad-hoc way.

Since GF warns the user (or gives an error message) when the abstract and concrete syntaxes are inconsistent, it is easy to keep the grammars in sync during development.

Automatic generation of speech recognition grammars eliminates the work of writing a separate grammar for the speech recognizer and keeping it consistent with the grammar used by the dialog system.



## Chapter 10

# Grammar Development Tools

### 10.1 Introduction

Writing a GF grammar for a new domain or language can be a demanding task, which may require in-depth knowledge both of the domain and of the target language. Large grammars can also be fairly complex and difficult to grasp. If embedded GF grammars are to be useful, it must be feasible to develop new grammars. This chapter discusses some tools aimed at supporting grammar writers.

### 10.2 Resource Grammars

Resource grammars [13] are general grammars for some language, whose writer is knowledgeable in the grammar of the target language. They contain the syntactic constructs and morphology of the language, along with general purpose vocabulary. Resource grammars can be used by writers of domain-specific grammars to avoid reimplementing the basic machinery of the target language. Resource grammars for a number of languages have already been developed.

### 10.3 Grammar Visualization

As shown in chapter 9, GF grammars can consist of many modules which may use each other. In order to give an intuition about the relationships between the different modules, a tool for visualizing module dependencies has been developed as part of this thesis. The GF system has been extended to produce a graph description in the DOT [31] language. The `dot` program can then be used to produce images from such descriptions. The module graphs in chapter 9 (figures 9.1 and 9.2) have been produced with this tool.

In the module dependency graphs, nodes with a solid border represent abstract modules and square nodes with a dashed border are concrete modules. Between a concrete module and the abstract module for which it is the concrete syntax, a dashed edge is drawn. Solid edges represent module inheritance for

both abstract and concrete modules. Dashed round nodes are resource modules and a dotted edge from one module to another means that the former opens the latter.

## 10.4 Future Work

### 10.4.1 Grammar development IDE

For many programming languages, there are development environments which assist the developer in writing programs. Some features inspired by such IDEs may be useful when writing GF grammar. This might include features such as module overviews, code highlighting, support for testing parsing and linearization, documentation browsers, the ability to chose functions from a list of available ones and generation of linearization terms by paring and example string. Janna Khegai has recently demonstated a system with some of these capabilities.

# Chapter 11

## Related Work

### 11.1 Embedded Context-Free Grammars

Parser generators such as Yacc [32] and Happy [33] can be seen as tools for embedding context-free grammars into applications. Such tools normally support only context-free grammars, and are aimed at parsing formal languages. They do generally not support linearization. BNFC [9] is a compiler front-end generator which supports parsing and linearization of context-free languages. Compared to such tools, the main advantages of the work described in this thesis is the power of the GF grammar formalism, and the natural support for linearization and multilinguality.

### 11.2 Attribute Grammars

Attribute grammars [34, 35] are context-free grammars where each symbol has a set of associated attributes. There are semantic rules associated with the productions which define how the attribute values are computed. Whereas embedded grammars as described in this thesis are grammars embedded in programs, attribute grammars can be seen as programs embedded in grammars.

### 11.3 Combinatory Categorical Grammars and OpenCCG

OpenCCG [36] is a Java library which provides parsing and linearization for Combinatory Categorical Grammars (CCG) [37]. Compared to GF, the notion of abstract syntax in CCG is fairly tightly coupled to the concrete syntax [3]. This seems to make it difficult to write multiple concrete syntaxes with the same abstract syntax, something which is essential for the way in which multilinguality is achieved when using GF grammars.

### 11.4 Multimodal Grammars

Michael Johnston describes unification-based multimodal parsing [38]. His approach is to extend chart parsing to multiple dimensions and to use unification to integrate information from different modalities. The approach described in

chapter 3 achieves a similar result by using records along with the existing unification mechanism for resolving reduplication.

OpenCCG implements multimodal extensions to CCG by Jason Baldridge [39].  
 FIXME: is this multimodality in the same sense as we mean? I don't understand the MMCCG paper.

## 11.5 Generation of Speech Recognition Grammars

The idea of generating context-free speech recognition grammars from grammars in a more expressive formalism has been described by Dowding et al [40].

## 11.6 External Grammars Tools

There are a number of stand-alone grammar tools which implement parsing in many different grammar formalisms. While these share some features with the systems described in this thesis, they are often designed for experimenting with writing grammars in the formalism which they support and seldom meant to be used in an embedded fashion.

FIXME: examples of external grammar tools

## 11.7 GF Gramlets

The *GF Gramlets* [14] system 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. While Gramlets and the Embedded GF Interpreter do not share any code, Gramlets provided inspiration for and a hint of the usefulness of writing a general GF interpreter in Java. In the future, we hope to adapt the Gramlets system to use the Embedded GF Interpreter.

## 11.8 Embedding the Full GF System

Before this work was started, some applications [5, 6] 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, or by using the GF Haskell API.

## 11.9 Speech Translation

FIXME: find some related work on speech translation

## Chapter 12

# Future Work

Many of the previous chapters list future work to be done in their respective areas. This chapter will outline some possibilities for future work on embedded grammars in general.

### 12.1 Non-human Interaction Grammars

Grammars can be used also for non-human computer interaction tasks, such as communication between system components or with external systems.

Many communication standards and data formats are described by grammars, and it might be feasible to use embedded grammars to implement support for such systems.

## Chapter 13

# Conclusions

### 13.1 Contributions

This thesis has described the following contributions to the area of embedded grammars:

**The Embedded GF Interpreter** An embeddable lightweight interpreter for GF grammars which supports parsing, linearization and translation.

**GF to Speech Recognition grammar compiler** Compiles GF grammars to common speech recognition grammar formats to ease integration with speech recognition and remove the problem of keeping multiple equivalent grammars in sync.

**Grammar module visualization tool** Visualizes the dependencies between grammar modules as a directed graph.

**Speech translator** A general unidirectional speech translator which translates between any two concrete GF syntaxes which share a common abstract syntax.

**Demonstration of a multimodal multilingual dialog system** A multimodal and multilingual dialog systems based on the use of multimodal and multilingual grammars.

### 13.2 Usefulness of Embedded Grammars

Applications such as the GF Speech Translator described in chapter 7 illustrate the power of embedded grammars. The application source code is essentially just glue between existing components. Changing the application to a new domain or language pair only requires new grammars.

As the dialog system example in chapter 9 shows, the use of embedded grammars reduces the task of creating a multimodal and multilingual natural language user interface to writing grammars and programs which work on abstract syntax terms.

### 13.3 Software Availability and Licensing

All stand-alone software written as part of this thesis is available from <http://www.cs.chalmers.se/~bringert/>. The parts integrated into the GF system are available as part of GF from <http://www.cs.chalmers.se/~aarne/GF/>.

All code written as part of this thesis is distributed under the GNU General Public License (GPL) [41].

# Bibliography

- [1] P. Martin-Löf, *Intuitionistic Type Theory*. Naples: Bibliopolis, 1984.
- [2] A. Ranta, “Grammatical Framework, a type-theoretical grammar formalism,” *The Journal of Functional Programming*, vol. 14, no. 2, pp. 145–189, 2004.
- [3] P. Ljunglöf, *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Göteborg University, Gothenburg, Sweden, November 2004.
- [4] D. A. Burke, “Improving the natural language translation of formal software specifications,” Master’s thesis, Chalmers University of Technology, December 2004.
- [5] J. Khagai, B. Nordström, and A. Ranta, “Multilingual syntax editing in GF,” in *CICLing*, pp. 453–464, 2003.
- [6] R. Hähnle, K. Johansson, and A. Ranta, “An authoring tool for informal and formal requirements specifications,” in *Fundamental Approaches to Software Engineering* (R.-D. Kutsche and H. Weber, eds.), no. 2306 in LNCS, 2002.
- [7] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*. Sun Microsystems, Inc., third ed., 2005. Proposed third edition: [http://java.sun.com/docs/books/jls/java\\_language-3\\_0-mr-spec.zip](http://java.sun.com/docs/books/jls/java_language-3_0-mr-spec.zip).
- [8] P. Ljunglöf, “Functional chart parsing of context-free grammars,” *The Journal of Functional Programming*, vol. 14, no. 6, pp. 669–680, 2004.
- [9] M. Forsberg and A. Ranta, “The BNF Converter: A high-level tool for implementing well-behaved programming languages,” in *NWPT’02 proceedings, Proceedings of the Estonian Academy of Sciences*, December 2003.
- [10] B. Bringert, “Embedded GF Interpreter Java API.” <http://www.cs.chalmers.se/~bringert/gf/gf-java.html>.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [12] D. L. Martin, A. J. Cheyer, and D. B. Moran, “The Open Agent Architecture: A framework for building distributed software systems,” *Applied Artificial Intelligence*, vol. 13, pp. 91–128, January-March 1999.



- [13] J. Khagai, “Language engineering in Grammatical Framework.” Licentiate Thesis, Chalmers University of Technology, Gothenburg, Sweden, 2003.
- [14] M. Forsberg, K. Johannisson, J. Khagai, and A. Ranta, “GF Gramlets.” <http://www.cs.chalmers.se/~krijo/gramlets.html>.
- [15] T. Coquand, “An algorithm for type-checking dependent types,” *Science of Computer Programming*, vol. 26, no. 1-3, pp. 167–177, 1996.
- [16] B. Bringert, R. Cooper, P. Ljunglöf, and A. Ranta, “Development of multi-modal and multilingual grammars: viability and motivation.” Deliverable D1.2a, TALK Project, IST-507802, December 2004.
- [17] A. Hunt, “JSpeech Grammar Format.” W3C Note, June 2000.
- [18] Sun Microsystems, Inc., *Java Speech API Programmer’s Guide*, October 1998.
- [19] Sun Microsystems, Inc., *Java Speech API Specification*, 1998.
- [20] Nuance Communications, Inc., 1005 Hamilton Avenue, Menlo Park, CA 94025, USA, *Nuance Speech Recognition System 8.5: Grammar Developer’s Guide*, December 2003.
- [21] Nuance Communications, Inc., 1005 Hamilton Avenue, Menlo Park, CA 94025, USA, *Nuance Speech Recognition System 8.5: Introduction to the Nuance System*, December 2003.
- [22] “Speech recognition grammar specification version 1.0.” W3C Recommendation, March 2004.
- [23] R. C. Moore, “Removing left recursion from context-free grammars,” in *Proceedings of the first meeting of the North American chapter of the Association for Computational Linguistics*, pp. 249–255, Morgan Kaufmann Publishers Inc., 2000.
- [24] S. Young, *ATK - An Application Toolkit for HTK*. Machine Intelligence Laboratory, Cambridge University Engineering Dept, Trumpington Street, Cambridge, CB2 1PZ, United Kingdom, 1.4.1 ed., July 2004.
- [25] S. Larsson and D. Traum, “Information state and dialogue management in the TRINDI dialogue move engine toolkit,” *Natural Language Engineering Special Issue on Best Practice in Spoken Language Dialogue Systems Engineering*, pp. 323–340, 2000.
- [26] H. Hammarström and A. Ranta, “Cardinal Numerals Revisited in GF.” Abstract accepted to Workshop on Numerals in the World’s Languages, Dept. of Linguistics Max Planck Institute for Evolutionary Anthropology, Leipzig, Germany, March 2004.
- [27] D. Adams, *The Hitchhiker’s Guide to the Galaxy*. Pan Macmillan, October 1979.
- [28] D. Hjelm, *Nuance Wrapper manual*. Göteborg University, Gothenburg, Sweden, June 2004.

- [29] Sun Microsystems, Inc., *FreeTTS Programmer's Guide*, 2003. <http://freetts.sourceforge.net/docs/ProgrammerGuide.html>.
- [30] H. Burden, *FreeTTS Agent for OAA*. Göteborg University, Gothenburg, Sweden. <http://www.ling.gu.se/projekt/talk/software/reports/freeTTSAgentReport.%pdf>.
- [31] E. Koutsofios and S. C. North, *Drawing graphs with dot*. AT&T Bell Laboratories, Murray Hill, NJ.
- [32] S. C. Johnson, "Yacc: Yet another compiler compiler," in *UNIX Programmer's Manual*, vol. 2, pp. 353–387, New York, NY, USA: Holt, Rinehart, and Winston, 1979.
- [33] A. Gill and S. Marlow, "Happy: The parser generator for Haskell." <http://haskell.org/happy/>.
- [34] D. E. Knuth, "Semantics of context-free languages.," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127–145, 1968.
- [35] D. E. Knuth, "Correction: Semantics of context-free languages.," *Mathematical Systems Theory*, vol. 5, no. 1, pp. 95–96, 1971.
- [36] "The OpenCCG Homepage." <http://openccg.sourceforge.net/>.
- [37] M. Steedman, "A very short introduction to CCG." <ftp://ftp.cis.upenn.edu/pub/steedman/ccg/ccgintro.ps.gz>, November 1996.
- [38] M. Johnston, "Unification-based multimodal parsing," in *Proceedings of the 36th conference on Association for Computational Linguistics*, pp. 624–630, Association for Computational Linguistics, 1998.
- [39] J. Baldridge and G.-J. M. Kruijff, "Multi-modal combinatory categorial grammar," in *Proceedings of EACL'03: 10th Conference of the European Chapter of the Association for Computational Linguistics*, 2003.
- [40] J. Dowding, B. A. Hockey, J. M. Gawron, and C. Culy, "Practical issues in compiling typed unification grammars for speech recognition," in *Meeting of the Association for Computational Linguistics*, pp. 164–171, 2001.
- [41] Free Software Foundation, Inc., *GNU General Public License*, June 1991. <http://www.fsf.org/licenses/gpl.txt>.

## Appendix A

# Embedded GF Interpreter OAA Agent

### A.1 Introduction

The Open Agent Architecture (OAA) is “a framework for integrating a community of heterogeneous software agents in a distributed environment”.

This document describes the GF OAA Agent included with the Embedded GF Interpreter.

### A.2 Running

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

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

#### A.2.1 Notes

- The path to the properties file is loaded as a java resource, so the leading slash means to look at the root of the package hierarchy to find the file in the current directory.
- The following must be on the classpath:
  - OAA 2.3.0 and all its requirements
  - The Trindikit Java library (for the OAAAgent class)

### A.3 Solvables

The OAA agent declares these solvables:

- parse (cf. Section A.3.1)
- linearize (cf. Section A.3.2)

- `translate` (cf. Section A.3.3)
- `list_grammars` (cf. Section A.3.4)
- `list_languages` (cf. Section A.3.5)

### A.3.1 `parse`

`parse(Grammar, Lang, Text, Tree)`

#### Parameters

**Grammar** The name of the grammar. This is the value of the `name` parameter in the properties file.

**Lang** The name of the concrete syntax that should be used for parsing. If `Lang` is not instantiated, the parser will try all available languages in the given grammar, and return results for each language that the text can be parsed in.

**Text** The text to parse. Must be instantiated.

**Tree** The parse tree. Normally not instantiated. The parse tree from the parser is unified with this value. Parse trees are represented as ICL structs.

#### Examples

- Language given, unambiguous parse:

```
parse(numerals, german, 'drei hundert', Tree)
=>
parse(numerals, german, 'drei hundert',
      num(pot2as3(pot2(pot0(n3)))))
```

- Language uninstantiated, unambiguous parse:

```
parse(numerals, Lang, 'three hundred', Tree)
=>
parse(numerals, english, 'three hundred',
      num(pot2as3(pot2(pot0(n3)))))
```

- Language uninstantiated, parses with several languages to same parse tree:

```
parse(numerals, Lang, 'tre', Tree)
=>
parse(numerals, albanian, tre,
      num(pot2as3(pot1as2(pot0as1(pot0(n3)))))
parse(numerals, danish, tre,
      num(pot2as3(pot1as2(pot0as1(pot0(n3)))))
parse(numerals, italian, tre,
      num(pot2as3(pot1as2(pot0as1(pot0(n3)))))
parse(numerals, norwegian_book, tre,
```

```

        num(pot2as3(pot1as2(pot0as1(pot0(n3))))))
parse(numerals, swedish, tre,
      num(pot2as3(pot1as2(pot0as1(pot0(n3))))))

```

- Language uninstantiated, parses with several languages to different parse trees:

```

parse(numerals, Lang, 'tres', Tree)
=>
parse(numerals, catalan, tres,
      num(pot2as3(pot1as2(pot0as1(pot0(n3))))))
parse(numerals, danish, tres,
      num(pot2as3(pot1as2(pot1(n6))))))
parse(numerals, spanish, tres,
      num(pot2as3(pot1as2(pot0as1(pot0(n3))))))

```

### A.3.2 linearize

```
linearize(Grammar, Lang, Tree, Text)
```

#### Parameters

**Grammar** The name of the grammar. This is the value of the name parameter in the properties file.

**Lang** The name of the concrete syntax that should be used for linearization. If Lang is not instantiated, linearizations for all available languages in the given grammar will be returned.

**Tree** The abstract syntax tree to linearize. Must be instantiated.

**Text** The linearization of the given tree.

#### Examples

- Language given:

```

linearize(numerals, english,
          num(pot2as3(pot1as2(pot0as1(pot0(n3))))), Str)
=>
linearize(numerals, english,
          num(pot2as3(pot1as2(pot0as1(pot0(n3))))), three)

```

### A.3.3 translate

```
translate(Grammar, FromLang, Input, ToLang, Output)
```

**Parameters**

**Grammar** The name of the grammar. This is the value of the name parameter in the properties file.

**FromLang** The name of the concrete syntax that should be used for parsing the input text. If Lang is not instantiated, all available languages in the given grammar will be tried.

**Input** The input text. Must be instantiated.

**ToLang** The name of the concrete syntax that should be used for linearizing the output.

**Output** The output text. Normally not instantiated.

**Examples**

- Input language not given:

```
translate(numerals, Lang, 'tres', english, Str)
=>
translate(numerals,catalan,tres,english,three)
translate(numerals,danish,tres,english,sixty)
translate(numerals,spanish,tres,english,three)
```

**A.3.4 list\_grammars**

```
list_grammars(Grammars)
```

**Parameters**

**Grammars** The list of available grammars.

**Examples**

- Get all grammars:

```
list_grammars(Grammars)
=>
list_grammars([query,answer])
```

**A.3.5 list\_languages**

```
list_languages(Grammar, InputLangs, OutputLangs)
```

**Parameters**

**Grammar** The grammar to get the languages for. If uninstantiated, there will be one answer for each available grammar.

**InputLangs** A list of the available input languages for the grammar.

**OutputLangs** A list of the available output languages for the grammar.

**Examples**

- Get languages for all grammars (Grammar uninstantiated):

```
list_languages(Grammar, InputLangs, OutputLangs)
=>
list_languages(query,
                ['*all*', 'GbgQueryEng', 'GbgQuerySwe'],
                ['*all*', 'GbgQueryEng', 'GbgQuerySwe'])
list_languages(answer,
                ['*all*', 'GbgRouteEng', 'GbgRouteMap',
                 'GbgRouteSwe'],
                ['*all*', 'GbgRouteEng', 'GbgRouteMap',
                 'GbgRouteSwe'])
```

- Get languages for the query grammar (Grammar instantiated):

```
list_languages('query', InputLangs, OutputLangs)
=>
list_languages(query,
                ['*all*', 'GbgQueryEng', 'GbgQuerySwe'],
                ['*all*', 'GbgQueryEng', 'GbgQuerySwe'])
```

## Appendix B

# Hello World Dialog System Source Code

```
import user.*;
import system.*;
import se.chalmers.cs.gf.util.Pair;
import se.chalmers.cs.gf.dialogutil.*;
import se.chalmers.cs.gf.dialogutil.sr.*;
import se.chalmers.cs.gf.dialogutil.tts.*;

import java.util.List;

/**
 * A very simple demo of a single-modality multilingual dialog
 * system using GF in Java.
 */
public class SimpleDemo {

    private TextListenerList listeners = new TextListenerList();

    private UserMain user;

    private SystemMain system;

    public SimpleDemo(UserMain user, SystemMain system) {
        this.user = user;
        this.system = system;
    }

    /**
     * Parses the user input, creates system output by calling answer(),
     * linearizes the system output in the language that the user used
     * and outputs the answer.
     */
    private void handleInput(String text) {
```



```

        System.err.println("Input: " + text);

        // parse the input with all available languages
        List<Pair<String,Input>> inputs =
            user.parseInputWithAll(text);

        // disambiguate the input, do nothing if that fails
        Pair<String,Input> p = disambiguate(inputs);
        if (p == null)
            return;

        // get the unambiguous input
        String inputLang = p.fst;
        Input input = p.snd;
        System.err.println("Parsed: " + input + " (" + inputLang + ")");

        // figure out answer to user input
        String outputLang = getOutputLang(inputLang);
        Output output = answer(input);
        System.err.println("Output: " + output);

        // linearize and output answer
        String outputText = system.linearizeOutput(outputLang, output);
        System.err.println("Saying: " + outputText);
        listeners.fireTextEvent(outputText);
    }

    /**
     * Disambiguate parse results.
     * @return The single unambiguous parse result, if any. Otherwise null.
     */
    private Pair<String,Input> disambiguate(List<Pair<String,Input>> is) {
        if (is.size() == 0) {
            System.err.println("No parse");
            return null;
        } else if (is.size() > 1) {
            System.err.println("Ambiguous parse:");
            for (Pair<String,Input> i : is)
                System.err.println(i.fst + ": " + i.snd);
            return null;
        }
        return is.get(0);
    }

    /**
     * Calculates a system output from a user input.
     */
    private Output answer(Input input) {
        return input.accept(new Oracle(), null);
    }

```

```

    }

    private class Oracle implements Input.Visitor<Output,Object> {
        public Output visit(user.Thanks p, Object arg) {
            return new Welcome();
        }
        public Output visit(HereYouGo p, Object arg) {
            return new system.Thanks();
        }
    }

    /**
     * Gets the output language to use for a given input language.
     */
    private String getOutputLang(String inputLang) {
        String lang = inputLang.substring(inputLang.length()-3);
        return "System" + lang;
    }

    /**
     * Add a source of user input, e.g. a dialog box or a speech
     * recognition engine.
     */
    public void addInputSource(TextInput inputSource) {
        inputSource.addTextListener(new TextListener() {
            public void textEvent(TextEvent e) {
                handleInput(e.getText());
            }
        });
    }

    /**
     * Adds a sink to which system output will be sent.
     */
    public void addOutputSink(TextListener l) { listeners.add(l); }

    public static void main(String [] args) throws java.io.IOException {
        UserMain user = new UserMain("/user.properties");
        SystemMain system = new SystemMain("/system.properties");

        SimpleDemo demo = new SimpleDemo(user, system);

        // Get input from a simple dialog
        //demo.addInputSource(new DialogInput());

        // For Nuance OAA text input
        Recognizer recog = new Recognizer("simpldemo", args);
        demo.addInputSource(new RecognizerInput(recog));

        // Print output to System.err

```

```
demo.addOutputSink(new ConsoleOutput());

// Speak output with a JSAPI speech synthesizer
demo.addOutputSink(new JavaSpeechOutput());
}

}
```