Creating Linguistic Resources with the Grammatical Framework

Aarne Ranta

LREC Tutorial, Malta, 17 May 2010
## Contents

1 The GF System and Simple Multilingual Grammars ........................................... 4
   1.1 GF = Grammatical Framework ......................................................... 4
   1.2 Installing and using the GF system .................................................. 7
   1.3 Working with context-free grammars in GF ......................................... 8
   1.4 Abstract and concrete syntax .......................................................... 11
   1.5 Making a grammar multilingual ....................................................... 13
   1.6 Visualizing trees and word alignment ............................................... 17
   1.7 From abstract trees to parse trees .................................................... 18
   1.8 Scaling up the grammar .................................................................. 19
   1.9 Exercises ....................................................................................... 20

2 Morphological Paradigms and Lexicon Building .............................................. 21
   2.1 Morphology .................................................................................... 21
   2.2 Lexical categories ........................................................................... 22
   2.3 Example: resource module for English verb inflection ...................... 23
   2.4 Smart paradigms ............................................................................. 26
   2.5 Phases of morphology implementation ............................................. 29
   2.6 Building a lexicon .......................................................................... 30
   2.7 Nonconcatenative morphology: Arabic ............................................. 31
   2.8 Exercises ....................................................................................... 36

3 Basics of a Linguistic Syntax Implementation ................................................ 36
   3.1 Syntax in the resource grammar ....................................................... 36
   3.2 The key categories and functions ...................................................... 37
   3.3 Predication: building clauses ............................................................ 38
   3.4 Complementation: building verb phrases ......................................... 39
   3.5 Determination: building noun phrases .............................................. 40
   3.6 Modification: adding adjectives to nouns ........................................ 41
   3.7 Lexical insertion ............................................................................. 43
   3.8 A miniature resource grammar for Italian ......................................... 44
3.9 Italian verb phrases .................................................... 46
3.10 Italian noun phrases .................................................. 48
3.11 Predication, at last ...................................................... 50
3.12 To do ................................................................. 51
3.13 Ergativity in Hindi/Urdu ............................................. 51
3.14 Exercises ........................................................... 52

4 Using the Resource Grammar Library in Applications 53
  4.1 Software libraries ................................................... 53
  4.2 Using the library: natural language output .................... 54
  4.3 Two kinds of grammarians ......................................... 55
  4.4 Meaning-preserving translation .................................. 56
  4.5 Example: abstract syntax for a ”Face” community .......... 57
  4.6 Functors and interfaces ............................................. 60
  4.7 Modules of a domain grammar: ”Face” community ........ 62
  4.8 Free variation ...................................................... 63
  4.9 Overview of the resource grammar API ....................... 63
  4.10 Exercises ......................................................... 68

5 Developing a GF Resource Grammar 68
  5.1 The principal module structure ................................. 69
  5.2 Effort statistics, completed languages ......................... 74
  5.3 How to start building a language, e.g. Marathi .............. 74
  5.4 Assignment: a good start ....................................... 77
Notice. This tutorial is an updated version of the one used at the GF Summer School 2009 (grammaticalframework.org/summerschool.html). It was first presented on an on-line course in April 2009. The summer school in August 2009 had 30 participants from 20 countries. 15 new languages were started. Since the summer school, the library has grown from 12 to 16 languages.

The goal of this tutorial is to introduce a fast way to resource grammar writing, by explaining the practical use of GF and the linguistic concepts in the resource grammar library.

For more details, we recommend

- the tutorial on the GF homepage grammaticalframework.org
- GF Book by A. Ranta, forthcoming at CSLI Publications

The code examples in this tutorial are available in code.haskell.org/gf/examples/lrec-tutorial/.

We cannot stress enough the importance of your own work on the code examples and exercises using the GF system!

1 The GF System and Simple Multilingual Grammars

Contents

What GF is
Installing the GF system
A grammar for John loves Mary in English, French, Latin, Dutch, Hebrew
Testing grammars and building applications
The scope of the Resource Grammar Library
Exercises

1.1 GF = Grammatical Framework

GF is a grammar formalism: a notation for writing grammars
GF is a **functional programming language** with types and modules

GF programs are called **grammars**

A grammar is a declarative program that defines

- parsing
- generation
- translation

**Multilingual grammars**

Many languages related by a common **abstract syntax**

```
Mary loves John                           Maria Ioannem amat
     /                                   /
--Fred Mary (Compl Love John)          --
     /                                   /
Marie aime Jean                         Moran ama bat um roh
```

**The GF program**

**Interpreter** for testing grammars (the **GF shell**)

**Compiler** for converting grammars to useful formats

- PGF, Portable Grammar Format
- speech recognition grammars (Nuance, HTK, ...)
- JavaScript

**The GF Resource Grammar Library**

Morphology and basic syntax

Common API for different languages

Currently (April 2010) 17 languages: Bulgarian, Catalan, Danish, Dutch, English, Finnish, French, German, Interlingua, Italian, Norwegian, Polish, Romanian, Russian, Spanish, Swedish, Urdu.
Under construction for 16 languages: Arabic, Esperanto, Farsi, Greek (Ancient), Hebrew, Icelandic, Japanese, Latin, Latvian, Maltese, Mongol, Portuguese, Swahili, Thai, Tswana, Turkish.

Where GF is used

Natural language interfaces: WebALT, see webalt.math.helsinki.fi/PublicFiles/CD/Screencast/TextMathEditor%20Demo.swf
Dialogue systems: TALK, see www.youtube.com/watch?v=1bfaYHWS6zU
Translation: MOLTO, see www.molto-project.eu

GF run-time system

PGF grammars can be embedded in Haskell, Java, and Prolog programs
They can be used in web servers

- fridge magnet demo: grammaticalframework.org:41296/fridge
- translator demo: grammaticalframework.org:41296/translate
1.2 Installing and using the GF system

Go to the GF home page, and follow shortcuts to either

- **Download**: download and install binaries
- **Developers**: download sources, compile, and install

The *Developers* method is recommended for resource grammar developers:

- latest updates and bug fixes
- version control system

**Starting the GF shell**

The command `gf` starts the GF shell:

```
$ gf
```
This is GF version 3.1.6.
License: see help -license.
Bug reports: http://code.google.com/p/grammatical-framework/issues/list

Languages:

Using the GF shell: help

Command h = help

> help

gives a list of commands with short descriptions.

> help parse

gives detailed help on the command parse.

Commands have both short (1 or 2 letters) and long names.

1.3 Working with context-free grammars in GF

These are the simplest grammars usable in GF. Example:

Pred.  S ::= NP VP ;
Compl. VP ::= V2 NP ;
John.  NP ::= "John" ;
Mary.  NP ::= "Mary" ;
Love.  V2 ::= "loves" ;
The first item in each rule is a syntactic function, used for building trees: Pred = predication, Compl = complementation.

The second item is a category: S = Sentence, NP = Noun Phrase, VP = Verb Phrase, V2 = 2-place Verb.

**Importing and parsing**

Copy or write the above grammar in file zero.cf.

To use a grammar in GF: import = i

```
> i zero.cf
```

To parse a string to a tree: parse = p

```
> p "John loves Mary"
  Pred John (Compl Love Mary)
```

Parsing is, by default, in category S. This can be overridden.

**Random generation, linearization, and pipes**

Generate a random tree: generate_random = gr

```
> gr
  Pred Mary (Compl Love Mary)
```

To linearize a tree to a string: linearize = l

```
> l Pred Mary (Compl Love Mary)
  Mary loves Mary
```

To pipe a command to another one: |

```
> gr | l
  Mary loves Mary
```
Graphical view of abstract trees

In Mac:

> p "John loves Mary" | visualize_tree -view=open

In Ubuntu Linux:

> p "John loves Mary" | visualize_tree -view=oeg

You need the Graphviz program to see the view.
1.4 Abstract and concrete syntax

A context-free rule

\[
\text{Pred. } S ::= \text{NP } \text{VP}
\]

defines two things:

- **abstract syntax**: build a tree of form \text{Pred np vp}
- **concrete syntax**: this tree linearizes to a string of form \text{np vp}

The main idea of GF: separate these two things.

Separating abstract and concrete syntax

A context-free rule is converted to two judgements in GF:
• fun, declaring a syntactic function
• lin, giving its linearization rule

Pred. \( S ::= NP \ VP \implies fun\) Pred : NP -> VP -> S  
lin Pred np vp = np ++ vp

Functions and concatenation

Function type: \( A \to B \to C \), read "function from A and B to C"
Function application: \( f \ a \ b \), read "\( f \) applied to arguments \( a \) and \( b \)"
Concatenation: \( x \++ y \), read "string \( x \) followed by string \( y \)"
Cf. functional programming in Haskell.
Notice: in GF, ++ is between token lists and therefore "creates a space".

From context-free to GF grammars

The grammar is divided to two modules

• an abstract module, judgement forms cat and fun
• a concrete module, judgement forms lincat and lin

<table>
<thead>
<tr>
<th>Judgement</th>
<th>reading</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat ( C )</td>
<td>( C ) is a category</td>
</tr>
<tr>
<td>fun ( f : T )</td>
<td>( f ) is a function of type ( T )</td>
</tr>
<tr>
<td>lincat ( C = L )</td>
<td>( C ) has linearization type ( L )</td>
</tr>
<tr>
<td>lin ( f \ xs = t )</td>
<td>( f \ xs ) has linearization ( t )</td>
</tr>
</tbody>
</table>

Abstract syntax, example

abstract Zero = {
  cat
  S ; NP ; VP ; V2 ;
  fun
  Pred : NP -> VP -> S ;
  Compl : V2 -> NP -> VP ;
  John, Mary : NP ;
  Love : V2 ;
}
Concrete syntax, English

concrete ZeroEng of Zero = {
  lincat
    S, NP, VP, V2 = Str ;
  lin
    Pred np vp = np ++ vp ;
    Compl v2 np = v2 ++ np ;
    John = "John" ;
    Mary = "Mary" ;
    Love = "loves" ;
  }

Notice: Str (token list, "string") as the only linearization type.

1.5 Making a grammar multilingual

One abstract + many concretes

The same system of trees can be given

  • different words
  • different word orders
  • different linearization types

Concrete syntax, French

concrete ZeroFre of Zero = {
  lincat
    S, NP, VP, V2 = Str ;
  lin
    Pred np vp = np ++ vp ;
    Compl v2 np = v2 ++ np ;
    John = "Jean" ;
    Mary = "Marie" ;
    Love = "aime" ;
  }

Just use different words
Translation and multilingual generation

Import many grammars with the same abstract syntax

```plaintext
> i ZeroEng.gf ZeroFre.gf
Languages: ZeroEng ZeroFre
```

Translation: pipe linearization to parsing

```plaintext
> p -lang=ZeroEng "John loves Mary" | l -lang=ZeroFre
Jean aime Marie
```

Multilingual generation: linearize into all languages

```plaintext
> gr | l
Pred Mary (Compl Love Mary)
Mary loves Mary
Marie aime Marie
```

Multilingual treebanks

**Treebank:** show both trees and their linearizations

```plaintext
> gr | l -treebank
Zero: Pred Mary (Compl Love Mary)
ZeroEng: Mary loves Mary
ZeroFre: Marie aime Marie
```

Concrete syntax, Latin

```plaintext
concrete ZeroLat of Zero = {
  lincat
    S, VP, V2 = Str ;
    NP = Case => Str ;
  lin
    Pred np vp = np ! Nom ++ vp ;
    Compl v2 np = np ! Acc ++ v2 ;
    John = table {Nom => "Ioannes" ; Acc => "Ioannem"} ;
    Mary = table {Nom => "Maria" ; Acc => "Mariam"} ;
    Love = "amat" ;
  param
    Case = Nom | Acc ;
}
```
Different word order (SOV), different linearization type, parameters.

Parameters in linearization

Latin has cases: nominative for subject, accusative for object.

- Ioannes Mariam amat "John-Nom loves Mary-Acc"
- Maria Ioannem amat "Mary-Nom loves John-Acc"

Parameter type for case (just 2 of Latin’s 6 cases):

\[
\text{param Case} = \text{Nom} \mid \text{Acc}
\]

Table types and tables

The linearization type of NP is a table type: from Case to Str,

\[
\text{lincat NP} = \text{Case} \Rightarrow \text{Str}
\]

The linearization of John is an inflection table,

\[
\text{lin John} = \text{table} \{\text{Nom} \Rightarrow "Ioannes" ; \text{Acc} \Rightarrow "Ioannem"\}
\]

When using an NP, select (!) the appropriate case from the table,

\[
\begin{align*}
\text{Pred np vp} &= \text{np} ! \text{Nom} \Rightarrow \text{vp} \\
\text{Compl v2 np} &= \text{np} ! \text{Acc} \Rightarrow \text{v2}
\end{align*}
\]

Concrete syntax, Dutch

concrete ZeroDut of Zero = {

\[
\begin{align*}
\text{lincat} \\
S, \text{NP, VP} &= \text{Str} \\
V2 &= \{\text{v} : \text{Str} ; \text{p} : \text{Str}\} \\
\text{lin} \\
\text{Pred np vp} &= \text{np} \Rightarrow \text{vp} \\
\text{Compl v2 np} &= \text{v2} \cdot \text{vp} \\
\text{John} &= \text{"Jan"} \\
\text{Mary} &= \text{"Marie"} \\
\text{Love} &= \{\text{v} = \text{"heeft"} ; \text{p} = \text{"lief"}\}
\end{align*}
\]

The verb heeft lief is a discontinuous constituent.
Record types and records

The linearization type of $V_2$ is a record type with two fields

$$\text{lincat } V_2 = \{v : \text{Str} ; p : \text{Str}\}$$

The linearization of Love is a record

$$\text{lin Love} = \{v = "\text{hat}" ; p = "\text{lieb}"\}$$

The values of fields are picked by projection ($.$)

$$\text{lin Compl v2 np} = v2.v ++ \text{np} ++ v2.p$$

Concrete syntax, Hebrew

```
concrete ZeroHeb of Zero = {
    flags coding=utf8,
    lincat
    S = Str;
    NP = \{s : Str ; g : Gender\};
    VP, V2 = Gender => Str;
    lin
    Pred np vp = np.s ++ vp ! np.g;
    Compl v2 np = table \{g => v2 ! g ++ "\text{v}N\text{n}" ++ np.s\};
    John = \{s = "\text{m}N\text{n}" ; g = Masc\};
    Mary = \{s = "\text{m}N\text{n}" ; g = Fem\};
    Love = table \{Masc => "\text{L}O\text{V}E" ; Fem => "\text{L}O\text{V}E"\};
    param
    Gender = Masc | Fem;
}
```

The verb agrees to the gender of the subject.

Variable and inherent features, agreement

NP has gender as its inherent feature - a field in the record

$$\text{lincat NP} = \{s : \text{Str} ; g : \text{Gender}\}$$

$$\text{lin Mary} = \{s = "\text{mry}" ; g = Fem\}$$
VP has gender as its \textbf{variable feature} - an argument of a table

\[ \text{lincat } \text{VP} = \text{Gender} \Rightarrow \text{Str} \]

In predication, the VP receives the gender of the NP

\[ \text{lin \ Pred np vp = np.s ++ vp ! np.g} \]

\textbf{Feature design}

Deciding on variable and inherent features is central in GF programming. Good hint: dictionaries give forms of variable features and values of inherent ones. Example: French nouns

- \textit{cheval} pl. \textit{chevaux} masc. noun

From this we infer that French nouns have variable number and inherent gender

\[ \text{lincat N = \{s : Number => Str ; g : Gender\}} \]

\section*{1.6 Visualizing trees and word alignment}

![Diagram of tree structures](image)
1.7 From abstract trees to parse trees

Link every word with its smallest spanning subtree
Replace every constructor function with its value category

Generating word alignment

In L1 and L2: link every word with its smallest spanning subtree
Delete the intervening tree, combining links directly from L1 to L2
Notice: in general, this gives phrase alignment
Notice: links can be crossing, phrases can be discontinuous

Word alignment via trees

> parse "John loves Mary" | aw -view=open
A more involved word alignment

Building applications

Compile the grammar to PGF:

$ gf -make ZeroEng.gf ZeroFre.gf ZeroLat.gf ZeroGer.gf ZeroHeb.gf

The resulting file Zero.pgf can be e.g. included in fridge magnets:

1.8 Scaling up the grammar

Zero.gf is a tiny fragment of the Resource Grammar
The current Resource Grammar has 80 categories, 200 syntactic functions, and a minimal lexicon of 500 words.

Even S, NP, VP, V2 will need richer linearization types.

**More to do on sentences**

The category S has to take care of

- tenses: *John has loved Mary*
- negation: *John doesn’t love Mary*
- word order (Dutch): *als Jan Marie lief heeft, heeft Marie Jan lief*

Moreover: questions, imperatives, relative clauses

**More to do on noun phrases**

NP also involves

- pronouns: *I, you, she, we*
- determiners: *the man, every place*

Moreover: common nouns, adjectives

1.9 **Exercises**

1. Install gf on your computer.
2. Learn and try out the commands `align_words`, `empty`, `generate_random`, `generate_trees`, `help`, `import`, `linearize`, `parse`, `put_string`, `quit`, `read_file`, `translation_quiz`, `unicode_table`, `visualize_parse`, `visualize_tree`, `write_file`.
3. Write a concrete syntax of Zero for yet another language (e.g. your summer school project language).
4. Extend the Zero grammar with ten new noun phrases and verbs.
5. Add to the Zero grammar a category A of adjectives and a function `ComplA : A -> VP`, which forms verb phrases like *is old*. 
2 Morphological Paradigms and Lexicon Building

Contents

Morphology, inflection, paradigm - example: English verbs
Regular patterns and smart paradigms
Overloaded operations
Inherent features in the lexicon
Building and bootstrapping a lexicon
Nonconcatenative morphology: Arabic

2.1 Morphology

Inflectional morphology: define the different forms of words

- English verb sing has the forms sing, sings, sang, sung, singing

Derivational morphology: tell how new words are formed from old words

- English verb sing produces the noun singer

We could do both in GF, but concentrate now on inflectional morphology.

Good start for a resource grammar

Complete inflection system: 1-6 weeks
Comprehensive lexicon: days or weeks
Morphological analysis: up to 200,000 words per second
Export to SQL, XFST, ...

What is a word?

In abstract syntax: an object of a basic type, such as Love : V2
In concrete syntax,

- primarily: an inflection table, the collection of all forms
• secundarily: a string, i.e. a single form

Thus love, loves, loved are

• distinct words as strings

• forms of the same word as an inflection table or an abstract syntax object

2.2 Lexical categories

Part of speech = word class = lexical category

In GF, a part of speech is defined as a cat and its associated lincat.

In GF, there is no formal difference between lexical and other cats.

But in the resource grammar, we maintain a discipline of separate lexical categories.

The main lexical categories in the resource grammar

<table>
<thead>
<tr>
<th>cat</th>
<th>name</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>noun</td>
<td>house</td>
</tr>
<tr>
<td>A</td>
<td>adjective</td>
<td>small</td>
</tr>
<tr>
<td>V</td>
<td>verb</td>
<td>sleep</td>
</tr>
<tr>
<td>V2</td>
<td>two-place verb</td>
<td>love</td>
</tr>
<tr>
<td>Adv</td>
<td>adverb</td>
<td>today</td>
</tr>
</tbody>
</table>

Typical feature design

<table>
<thead>
<tr>
<th>cat</th>
<th>variable</th>
<th>inherent</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>number, case</td>
<td>gender</td>
</tr>
<tr>
<td>A</td>
<td>number, case, gender, degree</td>
<td>position</td>
</tr>
<tr>
<td>V</td>
<td>tense, number, person, ...</td>
<td>auxiliary</td>
</tr>
<tr>
<td>V2</td>
<td>as V</td>
<td>complement case</td>
</tr>
<tr>
<td>Adv</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Module structure

Resource module with inflection functions as operations

\[
\text{resource MorphoEng} = \{\text{oper regV : Str -> V ; ...}\}
\]
Lexicon: abstract and concrete syntax

abstract Lex = {fun Walk : V ; ...}

concrete LexEng of Lex =
  open MorphoEng in {lin Walk = regV "walk" ; ...}

The same resource can be used (opened) in many lexica.
Abstract and concrete are **top-level** - they define trees, parsing, linearization.
Resource modules and **opers** are not top-level - they are "thrown away" after compilation (i.e. not preserved in PGF).

2.3 Example: resource module for English verb inflection

Use the library module Prelude.
Start by defining parameter types and parts of speech.

resource Morpho = open Prelude in {

  param
    VForm = VInf | VPres | VPast | VPastPart | VPresPart ;

  oper
    Verb : Type = {s : VForm => Str} ;

Judgement form **oper**: auxiliary operation.

Start: worst-case function

To save writing and to abstract over the Verb type

mkVerb : (_,_,_,_,_ : Str) -> Verb = \go,goes,went,gone,going -> {
  s = table {
    VInf => go ;
    VPres => goes ;
    VPast => went ;
    VPastPart => gone ;
    VPresPart => going
  }
} ;
Testing computation in resource modules

Import with retain option

> i -retain Morpho.gf

Use command cc = compute_concrete

> cc mkVerb "use" "uses" "used" "used" "using"
{s : Morpho.VForm => Str
  = table Morpho.VForm {
    Morpho.VInf => "use";
    Morpho.VPres => "uses";
    Morpho.VPast => "used";
    Morpho.VPastPart => "used";
    Morpho.VPresPart => "using"
  }
}

Defining paradigms

A paradigm is an operation of type

Str -> Verb

which takes a string and returns an inflection table.
Let’s first define the paradigm for regular verbs:

regVerb : Str -> Verb = \walk ->
 mkVerb walk (walk + "s") (walk + "ed") (walk + "ed") (walk + "ing") ;

This will work for walk, interest, play.
It will not work for sing, kiss, use, cry, fly, stop.

More paradigms

For verbs ending with s, x, z, ch

s_regVerb : Str -> Verb = \kiss ->
 mkVerb kiss (kiss + "es") (kiss + "ed") (kiss + "ed") (kiss + "ing") ;
For verbs ending with *e*

\[
\text{e\_regVerb : Str} \rightarrow \text{Verb} = \text{\textbackslash use} \rightarrow \\
\text{let us = init use} \\
\text{in mkVerb use (use + "s") (us + "ed") (us + "ed") (us + "ing") ;}
\]

Notice:

- the local definition \text{let } c = d \text{ in ...}
- the operation \text{init} from \text{Prelude}, dropping the last character

**More paradigms still**

For verbs ending with *y*

\[
\text{y\_regVerb : Str} \rightarrow \text{Verb} = \text{\textbackslash cry} \rightarrow \\
\text{let cr = init cry} \\
\text{in mkVerb cry (cr + "ies") (cr + "ied") (cr + "ied") (cry + "ing") ;}
\]

For verbs ending with *ie*

\[
\text{ie\_regVerb : Str} \rightarrow \text{Verb} = \text{\textbackslash die} \rightarrow \\
\text{let dy = Predef.tk 2 die + "y"} \\
\text{in mkVerb die (die + "s") (die + "d") (die + "d") (dy + "ing") ;}
\]

**What paradigm to choose**

If the infinitive ends with *s, x, z, ch*, choose \text{s\_regVerb: munch, munches}

If the infinitive ends with *y*, choose \text{y\_regVerb: cry, cries, cried}

- except if a vowel comes before: \text{play, plays, played}

If the infinitive ends with *e*, choose \text{e\_regVerb: use, used, using}

- except if an *i* precedes: \text{die, dying}
- or if an *e* precedes: \text{free, freeing}
2.4 Smart paradigms

Let GF choose the paradigm by **pattern matching on strings**

```haskell
smartVerb : Str -> Verb = \v -> case v of {
  _ + ("s"|"z"|"x"|"ch") => s_regVerb v ;
  _ + "ie" => ie_regVerb v ;
  _ + "ee" => ee_regVerb v ;
  _ + "e" => e_regVerb v ;
  _ + ("a"|"e"|"o"|"u") + "y" => regVerb v ;
  _ + "y" => y_regVerb v ;
  _ => regVerb v
}
```

**Pattern matching on strings**

Format: `case string of { pattern => value }`

Patterns:

- `_` matches any string
- A string in quotes matches itself: "ie"
- `+` splits into substrings: `_ + "y"
- `|` matches alternatives: "a"|"e"|"o"

Common practice: last pattern a catch-all `_`

**Testing the smart paradigm**

```plaintext
> cc -all smartVerb "munch"
munch munches munched munching

> cc -all smartVerb "die"
die dies died died dying

> cc -all smartVerb "agree"
agree agrees agreed agreed agreeing

> cc -all smartVerb "deploy"
deploy deploys deployed deployed deploying
```
> cc -all smartVerb "classify"
classify classifies classified classified classifying

The smart paradigm is not yet perfect

Irregular verbs are obviously not covered

> cc -all smartVerb "sing"
sing sings singed singed singing

Neither are regular verbs with consonant duplication

> cc -all smartVerb "stop"
stop stops stoped stoped stoping

The final consonant duplication paradigm

Use the Prelude function `last`

```haskell
  dupRegVerb : Str -> Verb = \stop ->
    let stopp = stop + last stop
    in
      mkVerb stop (stop + "s") (stopp + "ed") (stopp + "ed") (stopp + "ing") ;
```

String pattern: relevant consonant preceded by a vowel

```
_ + ("a"|"e"|"i"|"o"|"u") + ("b"|"d"|"g"|"m"|"n"|"p"|"r"|"s"|"t") => dupRegVerb v ;
```

Testing consonant duplication

Now it works

> cc -all smartVerb "stop"
stop stops stopped stopped stopping

But what about
> cc -all smartVerb "coat"
cocat coats coatted coatted coatting

Solution: a prior case for diphthongs before the last char (? matches one char)

_ + ("ea"|"ee"|"ie"|"oa"|"oo"|"ou") + ? => regVerb v ;

There is no waterproof solution

Duplication depends on stress, which is not marked in English:

- *omit* [o’mit]: omitted, omitting
- *vomit* [’vomit]: vomited, vomiting

This means that we occasionally have to give more forms than one.

We knew this already for irregular verbs. And we cannot write patterns for each of them either, because e.g. *lie* can be both *lie, lied, lied* or *lie, lay, lain*.

A paradigm for irregular verbs

Arguments: three forms instead of one.

Pattern matching done in regular verbs can be reused.

irregVerb : (_,_,_ : Str) -> Verb = \sing,sang,sung ->
let v = smartVerb sing
in
mkVerb sing (v.s ! VPres) sang sung (v.s ! VPresPart) ;

Putting it all together

We have three functions:

smartVerb : Str -> Verb
irregVerb : Str -> Str -> Str -> Verb
mkVerb : Str -> Str -> Str -> Str -> Str -> Verb

As all types are different, we can use overloading and give them all the same name.
An overloaded paradigm

For documentation: variable names showing examples of arguments.

\[
\begin{align*}
\text{mkV} &= \text{overload} \{ \\
& \quad \text{mkV} : (\text{cry} : \text{Str}) \to \text{Verb} = \text{smartVerb} ; \\
& \quad \text{mkV} : (\text{sing}, \text{sang}, \text{sung} : \text{Str}) \to \text{Verb} = \text{irregVerb} ; \\
& \quad \text{mkV} : (\text{go}, \text{goes}, \text{went}, \text{gone}, \text{going} : \text{Str}) \to \text{Verb} = \text{mkVerb} ; \\
& \} ; \\
\end{align*}
\]

Testing the overloaded paradigm

\[
\begin{align*}
> \text{cc} -\text{all} \text{ mkV} \ "\text{lie}" \\
\text{lie} \text{ lies} \text{ lied} \text{ lied} \text{ lying} \\
> \text{cc} -\text{all} \text{ mkV} \ "\text{lie}" \ "\text{lay}" \ "\text{lain}" \\
\text{lie} \text{ lies} \text{ lay} \text{ lain} \text{ lying} \\
> \text{cc} -\text{all} \text{ mkV} \ "\text{omit}" \\
\text{omit} \text{ omits} \text{ omitted} \text{ omitted} \text{ omitting} \\
> \text{cc} -\text{all} \text{ mkV} \ "\text{vomit}" \\
\text{vomit} \text{ vomits} \text{ vomitted} \text{ vomitted} \text{ vomitting} \\
> \text{cc} -\text{all} \text{ mkV} \ "\text{vomit}" \ "\text{vomited}" \ "\text{vomited}" \\
\text{vomit} \text{ vomits} \text{ vomited} \text{ vomited} \text{ vomiting} \\
> \text{cc} -\text{all} \text{ mkV} \ "\text{vomit}" \ "\text{vomits}" \ "\text{vomited}" \ "\text{vomited}" \ "\text{vomiting}" \\
\text{vomit} \text{ vomits} \text{ vomited} \text{ vomited} \text{ vomiting} \\
\end{align*}
\]

Surely we could do better for \textit{vomit}...

2.5 Phases of morphology implementation

1. Linearization type, with parametric and inherent features.
2. Worst-case function.
3. The set of paradigms, traditionally taking one argument each.
4. Smart paradigms, with relevant numbers of arguments.
5. Overloaded user function, collecting the smart paradigms.

Other parts of speech

Usually recommended order:
1. Nouns, the simplest class.
2. Adjectives, often using noun inflection, adding gender and degree.
3. Verbs, usually the most complex class, using adjectives in participles.

**Morphophonemic functions**

Many operations are common to different parts of speech.

Example: adding an s to an English noun or verb.

\[
\text{add_s : Str} \rightarrow \text{Str} = \lambda v \rightarrow \text{case } v \text{ of } \{ \\
\quad _ + ("s"|"z"|"x"|"ch") \Rightarrow v + "es" ; \\
\quad _ + ("a"|"e"|"o"|"u") + "y" \Rightarrow v + "s" ; \\
\quad cr + "y" \Rightarrow cr + "ies" ; \\
\quad _ \Rightarrow v + "s" \\
\};
\]

This should be defined separately, not directly in verb conjunctions.

Notice: pattern variable \( cr \) matches like \( _ \) but gets bound.

**2.6 Building a lexicon**

Boringly, we need abstract and concrete modules even for one language.

\[
\text{abstract } \text{Lex} = \{ \\
\quad \text{cat } V ; \\
\quad \text{fun } \text{lin} \\
\quad \text{play}_V : V ; \\
\quad \text{sleep}_V : V ; \\
\}
\]

\[
\text{concrete } \text{LexEng} = \text{open } \text{Morpho in } \{ \\
\quad \text{lin} \text{cat } V = \text{Verb} ; \\
\quad \text{lin} \text{play}_V = \text{mkV } "\text{play}" ; \\
\quad \text{lin} \text{sleep}_V = \text{mkV } "\text{sleep}" "\text{slept}" "\text{slept}" ; \\
\}
\]

Fortunately, these modules can be mechanically generated from a POS-tagged word list

\[
V \text{ play} \\
V \text{ sleep slept slept}
\]

**Bootstrapping a lexicon**

Alt 1. From a morphological POS-tagged word list: trivial
Alt 2. From a plain word list, POS-tagged: start assuming regularity, generate, correct, and add forms by iteration

\[
\begin{array}{c}
V \text{ play} \quad ====> \quad V \text{ play} \quad \text{played} \quad \text{played} \\
V \text{ sleep} \quad ====> \quad V \text{ sleep} \quad \text{sleeped} \quad \text{sleeped} \quad \text{V sleep} \quad \text{slept} \quad \text{slept}
\end{array}
\]

Example: Finnish nouns need 1.42 forms in average (to generate 26 forms).

### 2.7 Nonconcatenative morphology: Arabic

Semitic languages, e.g. Arabic: \textit{kataba} has forms \textit{kaAtib, yaktubu, ...}

Traditional analysis:

- word = \textbf{root} + \textbf{pattern}
- root = three consonants (\textbf{radicals})
- pattern = function from root to string (notation: string with variables \textit{F,C,L} for the radicals)

Example: \textit{yaktubu} = \textit{ktb} + \textit{yaFCuLu}

Words are datastructures rather than strings!

**Datastructures for Arabic**

Roots are records of strings.

\[
\text{Root} \quad : \quad \text{Type} = \{F,C,L : \text{Str}\} \\
\]

Patterns are functions from roots to strings.

\[
\text{Pattern} \quad : \quad \text{Type} = \text{Root} \rightarrow \text{Str} \\
\]

A special case is filling: a record of strings filling the four slots in a root.

\[
\text{Filling} \quad : \quad \text{Type} = \{F,FC,CL,L : \text{Str}\} \\
\]

This is enough for everything except middle consonant duplication (e.g. \textit{FaCCaLa}).
Applying a pattern

A pattern obtained from a filling intertwines the records:

\[ \text{fill} : \text{Filling} \rightarrow \text{Pattern} = \lambda p, r \rightarrow p.F + r.F + p.FC + r.C + p.CL + r.L + p.L ; \]

Middle consonant duplication also uses a filling but duplicates the \( C \) consonant of the root:

\[ \text{dfill} : \text{Filling} \rightarrow \text{Pattern} = \lambda p, r \rightarrow p.F + r.F + p.FC + r.C + r.C + p.CL + r.L + p.L ; \]

Encoding roots by strings

This is just for the ease of programming and writing lexica.
F = first letter, C = second letter, L = the rest.

\[ \text{getRoot} : \text{Str} \rightarrow \text{Root} = \lambda s \rightarrow \text{case } s \text{ of} \{ \]
\[ \text{F@?} + \text{C@?} + L \Rightarrow \{ \text{F} = \text{F} ; \text{C} = \text{C} ; \text{L} = \text{L} \} ; \]
\[ _ \Rightarrow \text{Predef.error ("cannot get root from" ++ s)} \]
\[ \} ; \]

The \textbf{as-pattern} \( x@p \) matches \( p \) and binds \( x \).
The \textbf{error function} \texttt{Predef.error} stops computation and displays the string. It is a typical catch-all value.

Encoding patterns by strings

Patterns are coded by using the letters F, C, L.

\[ \text{getPattern} : \text{Str} \rightarrow \text{Pattern} = \lambda s \rightarrow \text{case } s \text{ of} \{ \]
\[ \text{F} + \text{"F"} + \text{FC} + \text{"CC"} + \text{CL} + \text{"L"} + \text{L} \Rightarrow \]
\[ \text{dfill} \{ \text{F} = \text{F} ; \text{FC} = \text{FC} ; \text{CL} = \text{CL} ; \text{L} = \text{L} \} ; \]
\[ \text{F} + \text{"F"} + \text{FC} + \text{"C"} + \text{CL} + \text{"L"} + \text{L} \Rightarrow \]
\[ \text{fill} \{ \text{F} = \text{F} ; \text{FC} = \text{FC} ; \text{CL} = \text{CL} ; \text{L} = \text{L} \} ; \]
\[ _ \Rightarrow \text{Predef.error ("cannot get pattern from" ++ s)} \]
\[ \} ; \]
A high-level lexicon building function

Dictionary entry: root + pattern.

```haskell
getWord : Str -> Str -> Str = \r,p ->
  getPattern p (getRoot r);
```

Now we can try:

```haskell
> cc getWord "ktb" "yaFCuLu"
"yaktubu"
> cc getWord "ktb" "muFaCCiLu"
"mukattibu"
```

Parameters for the Arabic verb type

Inflection in tense, number, person, gender.

```haskell
param
  Number = Sg | Dl | Pl ;
  Gender = Masc | Fem ;
  Tense = Perf | Impf ;
  Person = Per1 | Per2 | Per3 ;
```

But not in all combinations. For instance: no first person dual.

(We have omitted most tenses and moods.)
Example of Arabic verb inflection

<table>
<thead>
<tr>
<th>Persona</th>
<th>Numerus</th>
<th>Perfectum</th>
<th>Imperfectum</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. masc. sing.</td>
<td>كُتِبْنَ</td>
<td>كُتِبَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>3. fem. sing.</td>
<td>كُتِبْنَ</td>
<td>كُتِبَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>2. masc. sing.</td>
<td>كُتِبْنَ</td>
<td>كُتِبَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>2. fem. sing.</td>
<td>كُتِبْنَ</td>
<td>كُتِبَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>1. sing.</td>
<td>كُتِبْنَ</td>
<td>كُتِبَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>3. masc. dual.</td>
<td>كُتِبْنَا</td>
<td>كُتِبْنَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>3. fem. dual.</td>
<td>كُتِبْنَا</td>
<td>كُتِبْنَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>2. dual.</td>
<td>كُتِبْنَا</td>
<td>كُتِبْنَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>3. masc. plur.</td>
<td>كُتِبْنَا</td>
<td>كُتِبْنَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>3. fem. plur.</td>
<td>كُتِبْنَا</td>
<td>كُتِبْنَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>2. masc. plur.</td>
<td>كُتِبْنَا</td>
<td>كُتِبْنَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>2. fem. plur.</td>
<td>كُتِبْنَا</td>
<td>كُتِبْنَ</td>
<td>كُتِبْ</td>
</tr>
<tr>
<td>1. plur.</td>
<td>كُتِبْنَا</td>
<td>كُتِبْنَ</td>
<td>كُتِبْ</td>
</tr>
</tbody>
</table>

Arabic verb type: implementation

We use an algebraic datatype to include only the meaningful combinations.

```plaintext
param VPer =
  Vp3    Number Gender
| Vp2Sg  Gender
| Vp2D1
| Vp2P1  Gender
| Vp1Sg
| Vp1P1  ;

oper Verb : Type = {s : Tense => VPer => Str} ;
```

Thus $2^3(3^2 + 2 + 1 + 2 + 1 + 1) = 2^3 = 36$ forms, not $2^3*3^2*3 = 36$.

An Arabic verb paradigm

```plaintext
pattV_u : Tense -> VPer -> Pattern = \t,v -> getPattern (case t of {
```
Perf => case v of {
  Vp3 Sg Masc => "FaCaLa" ;
  Vp3 Sg Fem => "FaCaLato" ; -- o is the no-vowel sign ("sukun")
  Vp3 Dl Masc => "FaCaLaA" ;
  -- ...
} ;
Impf => case v of {
  -- ...
  Vp1Sg => "A?aFoCuLu" ;
  Vp1Pl => "naFoCuLu"
}
);

u_Verb : Str -> Verb = \s -> {
  s = \t,p => appPattern (getRoot s) (pattV_u t p)
} ;

Applying an Arabic paradigm

Testing in the resource module:

> cc -all u_Verb "ktb"
kataba katabato katabaA katabataA katabuwA katabona katabota kataboti
katabotumaA katabotum katabotunv2a katabotu katabonaA yakotubu takotubu
yakotubaAni takotubaAni yakotubuwna yakotubna takotubu takotubiyna
yakotubaAni takotubuwna takotubona A?akotubu nakotubu

Building a lexicon:

fun ktb_V : V ;
lin ktb_V = u_Verb "ktb" ;

How we did the printing (recreational GF hacking)

We defined a HTML printing operation

  oper verbTable : Verb -> Str

and used it in a special category Table built by

  fun Tab : V -> Table ;
  lin Tab v = verbTable v ;
We then used

```
> l Tab ktb_V | ps -env=quotes -to_arabic | ps -to_html | wf -file=ara.html
> ! tr "\"" " " <ara.html >ar.html
```

### 2.8 Exercises

1. Learn to use the commands `compute_concrete`, `morpho_analyse`, `morpho_quiz`.
2. Try out some smart paradigms in the resource library files `Paradigms` for some languages you know (or don’t know yet). Use the command `cc` for this.
3. Write a morphology implementation for some word class and some paradigms in your target language. Start with feature design and finish with a smart paradigm.
4. Bootstrap a GF lexicon (abstract + concrete) of 100 words in your target language.
5. (Recreational GF hacking.) Write an operation similar to `verbTable` for printing nice inflection tables in HTML.

### 3 Basics of a Linguistic Syntax Implementation

#### Contents

- The key categories and rules
- Morphology-syntax interface
- Examples and variations in English, Italian, French, Finnish, Swedish, German, Hindi
- A miniature resource grammar: Italian
- Module extension and dependency graphs
- Ergativity in Hindi/Urdu

*Don’t worry if the details of this lecture feel difficult! Syntax is difficult and this is why resource grammars are so useful!*

#### 3.1 Syntax in the resource grammar

"Linguistic ontology": syntactic structures common to languages

80 categories, 200 functions, which have worked for all resource languages so far
Sufficient for most purposes of expressing meaning: mathematics, technical documents, dialogue systems

Must be extended by language-specific rules to permit parsing of arbitrary text (ca. 10% more in English?)

A lot of work, easy to get wrong!

3.2 The key categories and functions

The key categories

<table>
<thead>
<tr>
<th>cat</th>
<th>name</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cl</td>
<td>clause</td>
<td>every young man loves Mary</td>
</tr>
<tr>
<td>VP</td>
<td>verb phrase</td>
<td>loves Mary</td>
</tr>
<tr>
<td>V2</td>
<td>two-place verb</td>
<td>loves</td>
</tr>
<tr>
<td>NP</td>
<td>noun phrase</td>
<td>every young man</td>
</tr>
<tr>
<td>CN</td>
<td>common noun</td>
<td>young man</td>
</tr>
<tr>
<td>Det</td>
<td>determiner</td>
<td>every</td>
</tr>
<tr>
<td>AP</td>
<td>adjectival phrase</td>
<td>young</td>
</tr>
</tbody>
</table>

The key functions

<table>
<thead>
<tr>
<th>fun</th>
<th>name</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>PredVP : NP -&gt; VP -&gt; Cl</td>
<td>predication</td>
<td>every man loves Mary</td>
</tr>
<tr>
<td>ComplV2 : V2 -&gt; NP -&gt; VP</td>
<td>complementation</td>
<td>loves Mary</td>
</tr>
<tr>
<td>DetCN : Det -&gt; CN -&gt; NP</td>
<td>determination</td>
<td>every man</td>
</tr>
<tr>
<td>AdjCN : AP -&gt; CN -&gt; CN</td>
<td>modification</td>
<td>young man</td>
</tr>
</tbody>
</table>

Feature design

<table>
<thead>
<tr>
<th>cat</th>
<th>variable</th>
<th>inherent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cl</td>
<td>tense</td>
<td>-</td>
</tr>
<tr>
<td>VP</td>
<td>tense, agr</td>
<td>-</td>
</tr>
<tr>
<td>V2</td>
<td>tense, agr</td>
<td>case</td>
</tr>
<tr>
<td>NP</td>
<td>case</td>
<td>agr</td>
</tr>
<tr>
<td>CN</td>
<td>number, case</td>
<td>gender</td>
</tr>
<tr>
<td>Det</td>
<td>gender, case</td>
<td>number</td>
</tr>
<tr>
<td>AP</td>
<td>gender, number, case</td>
<td>-</td>
</tr>
</tbody>
</table>

agr = agreement features: gender, number, person
### 3.3 Predication: building clauses

**Interplay between features**

```
param Tense, Case, Agr

lincat Cl = {s : Tense => Str }
lincat NP = {s : Case => Str ; a : Agr}
lincat VP = {s : Tense => Agr => Str }

fun PredVP : NP -> VP -> Cl
lin PredVP np vp = {s = \t => np.s ! subj ++ vp.s ! t ! np.a}

oper subj : Case
```

**Feature passing**

In general, combination rules just pass features: no case analysis (table expressions) is performed.

A special notation is hence useful:

\[ \langle p, q \rangle \mapsto t \quad \text{===} \quad \text{table } \{ p \mapsto \text{table } \{ q \mapsto t \} \} \]

It is similar to lambda abstraction (\( \langle x, y \rangle \mapsto t \) in a function type).

**Predication: examples**

**English**

<table>
<thead>
<tr>
<th>np.agr</th>
<th>present</th>
<th>past</th>
<th>future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sg Per1</td>
<td>I sleep</td>
<td>I slept</td>
<td>I will sleep</td>
</tr>
<tr>
<td>Sg Per3</td>
<td>she sleeps</td>
<td>she slept</td>
<td>she will sleep</td>
</tr>
<tr>
<td>Pl Per1</td>
<td>we sleep</td>
<td>we slept</td>
<td>we will sleep</td>
</tr>
</tbody>
</table>

**Italian** ("I am tired", "she is tired", "we are tired")

<table>
<thead>
<tr>
<th>np.agr</th>
<th>present</th>
<th>past</th>
<th>future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Masc Sg Per1</td>
<td>io sono stanco</td>
<td>io ero stanco</td>
<td>io sarò stanco</td>
</tr>
<tr>
<td>Fem Sg Per3</td>
<td>lei è stanca</td>
<td>lei era stanca</td>
<td>lei sarà stanca</td>
</tr>
<tr>
<td>Fem Pl Per1</td>
<td>noi siamo stanche</td>
<td>noi eravamo stanche</td>
<td>noi saremo stanche</td>
</tr>
</tbody>
</table>
Predication: variations

Word order:

- *will I sleep* (English), *è stanca lei* (Italian)

Pro-drop:

- *io sono stanco* vs. *sono stanco* (Italian)

Ergativity:

- ergative case of transitive verb subject; agreement to object (Hindi)

Variable subject case:

- *minä olen lapsi* vs. *minulla on lapsi* (Finnish, "I am a child" (nominative) vs. "I have a child" (adessive))

3.4 Complementation: building verb phrases

Interplay between features

\[
\text{lincat NP} = \{ s : \text{Case} \Rightarrow \text{Str} ; a : \text{Agr} \}\n\]

\[
\text{lincat VP} = \{ s : \text{Tense} \Rightarrow \text{Agr} \Rightarrow \text{Str} \}\n\]

\[
\text{lincat V2} = \{ s : \text{Tense} \Rightarrow \text{Agr} \Rightarrow \text{Str} ; c : \text{Case} \}\n\]

fun ComplV2 : V2 -> NP -> VP

\[
\text{lin ComplV2 v2 vp} = \{ s = \text{\textbackslash t, a} \Rightarrow v2.s \! t \! a \Rightarrow \text{np.s} ! v2.c \}\n\]

Complementation: examples

English

<table>
<thead>
<tr>
<th>v2.case</th>
<th>infinitive VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acc</td>
<td>love me</td>
</tr>
<tr>
<td>at + Acc</td>
<td>look at me</td>
</tr>
</tbody>
</table>

Finnish
### Complementation: variations

**Prepositions**: a two-place verb usually involves a preposition in addition case

\[
\text{lincat } \text{V2} = \{s : \text{Tense} => \text{Agr} => \text{Str} ; c : \text{Case} ; \text{prep} : \text{Str}\}
\]

\[
\text{lin ComplV2 } \text{v2} \text{ vp} = \{s = \backslash t, a => \text{v2.s} \uparrow \text{t} \uparrow a \uparrow \text{v2.prep} \uparrow \text{np.s} \uparrow \text{v2.c}\}
\]

**Clitics**: the place of the subject can vary, as in Italian:

- Maria ama Giovanni vs. Maria mi ama ("Mary loves John" vs. "Mary loves me")

### 3.5 Determination: building noun phrases

**Interplay between features**

\[
\text{lincat } \text{NP} = \{s : \text{Case} => \text{Str} ; a : \text{Agr} \}
\]

\[
\text{lincat } \text{CN} = \{s : \text{Number} => \text{Case} => \text{Str} ; g : \text{Gender}\}
\]

\[
\text{lincat } \text{Det} = \{s : \text{Gender} => \text{Case} => \text{Str} ; n : \text{Number}\}
\]

\[
\text{fun DetCN} : \text{Det} \rightarrow \text{CN} \rightarrow \text{NP}
\]

\[
\text{lin DetCN } \text{det} \text{ cn} = \{
    s = \backslash c => \text{det.s} \uparrow \text{cn.g} \uparrow \text{c} \uparrow \text{cn.s} \uparrow \text{det.n} \uparrow \text{c} ;
    a = \text{aggr} \text{cn.g} \text{det.n Per3}
\}
\]

\[
\text{oper agr} : \text{Gender} \rightarrow \text{Number} \rightarrow \text{Person} \rightarrow \text{Agr}
\]

**Determination: examples**

English
Italian ("this wine", "this pizza", "those pizzas")

<table>
<thead>
<tr>
<th>Det.num</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sg</td>
<td>every house</td>
</tr>
<tr>
<td>Pl</td>
<td>these houses</td>
</tr>
</tbody>
</table>

Finnish ("every house", "these houses")

<table>
<thead>
<tr>
<th>Det.num</th>
<th>CN.gen</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sg</td>
<td>Masc</td>
<td>questo vino</td>
</tr>
<tr>
<td>Sg</td>
<td>Fem</td>
<td>questa pizza</td>
</tr>
<tr>
<td>Pl</td>
<td>Fem</td>
<td>quelle pizze</td>
</tr>
</tbody>
</table>

### Determination: variations

Systematic number variation:

- *this*-these, *the*-the, *il*-i (Italian "the-the")

"Zero" determiners:

- *talo* ("a house") vs. *talo* ("the house") (Finnish)

- *a house* vs. *houses* (English), *une maison* vs. *des maisons* (French)

Specificity parameter of nouns:

- *varje hus* vs. *det huset* (Swedish, "every house" vs. "that house")

### 3.6 Modification: adding adjectives to nouns

**Interplay between features**

\[
\text{lincat AP} = \{s : \text{Gender} \Rightarrow \text{Number} \Rightarrow \text{Case} \Rightarrow \text{Str} \}
\]

\[
\text{lincat CN} = \{s : \text{Number} \Rightarrow \text{Case} \Rightarrow \text{Str} ; g : \text{Gender}\}
\]
fun AdjCN : AP -> CN -> CN

\[ \begin{align*}
\text{lin AdjCN ap cn} &= \{ \\
\hspace{1em} s &= \backslash n,c \mapsto \text{ap.s} ! \text{cn.g} ! n ! c \mathbin{\ddagger} \text{cn.s} ! n ! c ; \\
\hspace{1em} g &= \text{cn.g} \\
\} \\
\end{align*} \]

**Modification: examples**

**English**

<table>
<thead>
<tr>
<th>CN, singular</th>
<th>CN, plural</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{new house}</td>
<td>\textit{new houses}</td>
</tr>
</tbody>
</table>

**Italian ("red wine", "red house")**

<table>
<thead>
<tr>
<th>CN.gen</th>
<th>CN, singular</th>
<th>CN, plural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Masc</td>
<td>\textit{vino rosso}</td>
<td>\textit{vini rossi}</td>
</tr>
<tr>
<td>Fem</td>
<td>\textit{casa rossa}</td>
<td>\textit{case rosse}</td>
</tr>
</tbody>
</table>

**Finnish ("red house")**

<table>
<thead>
<tr>
<th>CN, sg, nominative</th>
<th>CN, sg, ablative</th>
<th>CN, pl, essive</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{punainen talo}</td>
<td>\textit{punaiselta talolta}</td>
<td>\textit{punaisina taloina}</td>
</tr>
</tbody>
</table>

**Modification: variations**

The place of the adjectival phrase

- Italian: \textit{casa rossa}, \textit{vecchia casa} ("red house", "old house")
- English: \textit{old house}, \textit{house similar to this}

Specificity parameter of the adjective

- German: \textit{ein rotes Haus} vs. \textit{das rote Haus} ("a red house" vs. "the red house")
3.7 Lexical insertion

To “get started” with each category, use words from lexicon.

There are lexical insertion functions for each lexical category:

\[
\begin{align*}
\text{UseN} & : N \rightarrow CN \\
\text{UseA} & : A \rightarrow AP \\
\text{UseV} & : V \rightarrow VP
\end{align*}
\]

The linearization rules are often trivial, because the lin\_cats match

\[
\begin{align*}
\text{lin UseN } n & = n \\
\text{lin UseA } a & = a \\
\text{lin UseV } v & = v
\end{align*}
\]

However, for UseV in particular, this will usually be more complex.

The head of a phrase

The inserted word is the head of the phrases built from it:

- *house* is the head of *house, big house, big old house* etc

As a rule with many exceptions and modifications,

- variable features are passed from the phrase to the head
- inherent features of the head are inherited by the noun

This works for endocentric phrases: the head has the same type as the full phrase.

What is the head of a noun phrase?

In an NP of form Det CN, is Det or CN the head?

Neither, really, because features are passed in both directions:

\[
\text{lin DetCN } \text{det} \text{ cn} = \{
    \begin{align*}
    s & = \backslash c \Rightarrow \text{det}.s ! \text{cn}.g ! c +\text{cn}.s ! \text{det}.n ! c \\
    a & = \text{agr} \text{cn}.g \text{det}.n \text{Per3}
    \end{align*}
\}
\]

Moreover, this NP is exocentric: no part is of the same type as the whole.
Structural words

Structural words = function words, words with special grammatical functions

- determiners: the, this, every
- pronouns: I, she
- conjunctions: and, or, but

Often members of closed classes, which means that new words are never (or sel-
dom) introduces to them.

Linearization types are often specific and inflection are irregular.

3.8 A miniature resource grammar for Italian

We divide it to five modules - much fewer than the full resource!

abstract Grammar -- syntactic cats and funs

abstract Lang = Grammar **... -- test lexicon added to Grammar

resource ResIta -- resource for Italian

concrete GrammarIta of Grammar = open ResIta in... -- Italian syntax

concrete LangIta of Lang = GrammarIta ** open ResIta in... -- It. lexicon

Extension vs. opening

Module extension: N = M1, M2, M3 ** {...}

- module N inherits all judgements from M1,M2,M3

Module opening: N = open R1, R2, R3 in {...}

- module N can use all judgements from R1,R2,R3 (but doesn’t inherit them)
Module dependencies

rectangle = abstract, solid ellipse = concrete, dashed ellipse = resource

Producing the dependency graph

Using the command `dg = dependency_graph` and graphviz

```
> i -retain LangIta.gf
> dependency_graph
wrote graph in file _gfdepgraph.dot
> ! dot -Tjpg _gfdepgraph.dot >testdep.jpg
```

Before calling dot, removed the module `Predef` to save space.

The module Grammar

```plaintext
abstract Grammar = {
  cat
  Cl ; NP ; VP ; AP ; CN ; Det ; N ; A ; V ; V2 ;
  fun
  PredVP : NP -> VP -> Cl ;
  ComplV2 : V2 -> NP -> VP ;
  DetCN : Det -> CN -> NP ;
  ModCN : CN -> AP -> CN ;
```
Parameters

Parameters are defined in ResIta.gf. Just 11 of the 56 verb forms.

\[
\begin{align*}
\text{Number} &= \text{Sg} | \text{Pl} ; \\
\text{Gender} &= \text{Masc} | \text{Fem} ; \\
\text{Case} &= \text{Nom} | \text{Acc} | \text{Dat} ; \\
\text{Aux} &= \text{Avere} | \text{Essere} ; \quad \text{--- the auxiliary verb of a verb} \\
\text{Tense} &= \text{Pres} | \text{Perf} ; \\
\text{Person} &= \text{Per1} | \text{Per2} | \text{Per3} ; \\
\text{Agr} &= \text{Ag Gender Number Person} ; \\
\text{VForm} &= \text{VInf} | \text{VPres Number Person} | \text{VPart Gender Number} ;
\end{align*}
\]

3.9 Italian verb phrases

Tense and agreement of a verb phrase, in syntax

<table>
<thead>
<tr>
<th>UseV arrive_V</th>
<th>Pres</th>
<th>Perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ag Masc Sg Per1</td>
<td>arrivo</td>
<td>sono arrivato</td>
</tr>
<tr>
<td>Ag Fem Sg Per1</td>
<td>arrivo</td>
<td>sono arrivata</td>
</tr>
<tr>
<td>Ag Masc Sg Per2</td>
<td>arrivi</td>
<td>sei arrivato</td>
</tr>
<tr>
<td>Ag Fem Sg Per2</td>
<td>arrivi</td>
<td>sei arrivata</td>
</tr>
<tr>
<td>Ag Masc Sg Per3</td>
<td>arriva</td>
<td>è arrivato</td>
</tr>
<tr>
<td>Ag Fem Sg Per3</td>
<td>arriva</td>
<td>è arrivata</td>
</tr>
<tr>
<td>Ag Masc Pl Per1</td>
<td>arriviamo</td>
<td>siamo arrivati</td>
</tr>
<tr>
<td>Ag Fem Pl Per1</td>
<td>arriviamo</td>
<td>siamo arrive</td>
</tr>
<tr>
<td>Ag Masc Pl Per2</td>
<td>arrivate</td>
<td>siete arrivati</td>
</tr>
<tr>
<td>Ag Fem Pl Per2</td>
<td>arrivate</td>
<td>siete arrivati</td>
</tr>
<tr>
<td>Ag Masc Pl Per3</td>
<td>arrivano</td>
<td>sono arrivati</td>
</tr>
<tr>
<td>Ag Fem Pl Per3</td>
<td>arrivano</td>
<td>sono arrive</td>
</tr>
</tbody>
</table>

46
The forms of a verb, in morphology

<table>
<thead>
<tr>
<th>arrive_V</th>
<th>form</th>
</tr>
</thead>
<tbody>
<tr>
<td>VInf</td>
<td>arrivate</td>
</tr>
<tr>
<td>VPres Sg Per1</td>
<td>arrivo</td>
</tr>
<tr>
<td>VPres Sg Per2</td>
<td>arrivi</td>
</tr>
<tr>
<td>VPres Sg Per3</td>
<td>arriva</td>
</tr>
<tr>
<td>VPres Pl Per1</td>
<td>arriviamo</td>
</tr>
<tr>
<td>VPres Pl Per2</td>
<td>arrivate</td>
</tr>
<tr>
<td>VPres Pl Per3</td>
<td>arrivano</td>
</tr>
<tr>
<td>VPart Masc Sg</td>
<td>arrivato</td>
</tr>
<tr>
<td>VPart Fem Sg</td>
<td>arrivata</td>
</tr>
<tr>
<td>VPart Masc Pl</td>
<td>arrivati</td>
</tr>
<tr>
<td>VPart Fem Pl</td>
<td>arrive</td>
</tr>
</tbody>
</table>

Inherent feature: aux is essere.

The verb phrase type

Lexical insertion maps V to VP.
Two possibilities for VP: either close to Cl,

\[
\text{lincat VP} = \{s : \text{Tense} \Rightarrow \text{Agr} \Rightarrow \text{Str}\}
\]

or close to V, just adding a clitic and an object to verb,

\[
\text{lincat VP} = \{v : \text{Verb} ; \text{clit} : \text{Str} ; \text{obj} : \text{Str}\} ;
\]

We choose the latter. It is more efficient in parsing.

Verb phrase formation

Lexical insertion is trivial.

\[
\text{lin UseV v} = \{v = v ; \text{clit, obj} = []\}
\]

Complementation assumes NP has a clitic and an ordinary object part.
lin ComplV2 =
  let
   nps = np.s ! v2.c
 in {
   v = {s = v2.s ; aux = v2.aux};
   clit = nps.clit;
   obj = nps.obj
}

3.10 Italian noun phrases

Being clitic depends on case

lin cat NP = {s : Case => {clit, obj : Str} ; a : Agr} ;

Examples:

lin she_NP = {
  s = table {
    Nom => {clit = [] ; obj = "lei"};
    Acc => {clit = "la" ; obj = []};
    Dat => {clit = "le" ; obj = []}
  } ;
  a = Ag Fem Sg Per3
}
lin John_NP = {
  s = table {
    Nom | Acc => {clit = [] ; obj = "Giovanni"};
    Dat => {clit = [] ; obj = "a Giovanni"}
  } ;
  a = Ag Fem Sg Per3
}

Noun phrases: alternatively

Use a feature instead of separate fields,

lin cat NP = {s : Case => {s : Str ; isClit : Bool} ; a : Agr} ;

The use of separate fields is more efficient and scales up better to multiple clitic positions.
Determination

No surprises

\[
\text{Det} = \{ s : \text{Gender} \rightarrow \text{Case} \rightarrow \text{Str} ; n : \text{Number} \} ;
\]

\[
\text{DetCN} \det \ cn = \{
    s = \\{ \text{obj} = \det.s ! \text{cn.g} ! \text{c} \rightarrow \text{cn.s} ! \det.n ; \text{clit} = [] \};
    a = \text{Ag} \ \text{cn.g} \ \det.n \ \text{Per3}
\}
\]

Building determiners

Often from adjectives:

\[
\text{this_Det} = \text{adjDet} (\text{mkA "questo"}) \ Sg ;
\]

\[
\text{these_Det} = \text{adjDet} (\text{mkA "questo"}) \ Pl ;
\]

\[
\text{prepCase} : \text{Case} \rightarrow \text{Str} = \\{ \text{Dat} \rightarrow "a" ;\}
\]

\[
\text{adjDet} : \text{Adj} \rightarrow \text{Number} \rightarrow \text{Determiner} = \\{ \text{s} = \\{ \text{g} , \text{c} \rightarrow \text{prepCase} \ \text{c} ++ \text{adj.s} ! \text{g} ! \text{n} ; \text{n} = \text{n} \}
\}
\]

Articles: see GrammarIta.gf

Adjectival modification

Recall the inherent feature for position

\[
\text{AP} = \{ s : \text{Gender} \rightarrow \text{Number} \rightarrow \text{Str} ; \text{isPre} : \text{Bool} \} ;
\]

\[
\text{ModCN} \ cn \ ap = \{
    s = \\{ \text{n} \rightarrow \text{preOrPost} \ \text{ap.isPre} (\text{ap.s} ! \text{cn.g} ! \text{n}) (\text{cn.s} ! \text{n}) ; \text{g} = \text{cn.g} \}
\}
\]

Obviously, separate pre- and post- parts could be used instead.
Italian morphology

Complex but mostly great fun:

```haskell
regNoun : Str -> Noun = \
vino -> case vino of {
  fuo + c@("c"|"g") + "o" => mkNoun vino (fuo + c + "hi") Masc ;
  ol + "io" => mkNoun vino (ol + "i") Masc ;
  vin + "o" => mkNoun vino (vin + "i") Masc ;
  cas + "a" => mkNoun vino (cas + "e") Fem ;
  pan + "e" => mkNoun vino (pan + "i") Masc ;
  _ => mkNoun vino vino Masc
} ;
```

See ResIta for more details.

3.11 Predication, at last

Place the object and the clitic, and select the verb form.

```haskell
lin PredVP np vp =
  let
    subj = (np.s ! Nom).obj ;
    obj = vp.obj ;
    clit = vp.clit ;
    verb = table {
      Pres => agrV vp.v np.a ;
      Perf => agrV (auxVerb vp.v.aux) np.a ++ agrPart vp.v np.a
    }
  in {
    s = \t => subj ++ clit ++ verb ! t ++ obj
  } ;
```

Selection of verb form

We need it for the present tense

```haskell
oper agrV : Verb -> Agr -> Str = \v,a -> case a of {
  Ag _ n p => v.s ! VPres n p
} ;
```

The participle agrees to the subject, if the auxiliary is essere
3.12 To do

Full details of the core resource grammar are in ResIta (150 loc) and GrammarIta (80 loc).

One thing is not yet done correctly: agreement of participle to accusative clitic object: now it gives *io la ho amato*, and not *io la ho amata*.

This is left as an exercise!

3.13 Ergativity in Hindi/Urdu

Normally, the subject is nominative and the verb agrees to the subject. However, in the perfective tense:

- the subject of a transitive verb is in an ergative ”case” (particle *ne*)
- the verb agrees to the object

Example: ”the boy/girl eats the apple/bread”

<table>
<thead>
<tr>
<th>subj</th>
<th>obj</th>
<th>gen. present</th>
<th>perfective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Masc</td>
<td>Masc</td>
<td>ladka: seb Ka:ta: hai</td>
<td>ladke ne seb Ka:ya:</td>
</tr>
<tr>
<td>Masc</td>
<td>Fem</td>
<td>ladka: roTi: Ka:ta: hai</td>
<td>ladke ne roTi: Ka:yi:</td>
</tr>
<tr>
<td>Fem</td>
<td>Masc</td>
<td>ladki: seb Ka:ti: hai</td>
<td>ladki: ne seb Ka:ya:</td>
</tr>
<tr>
<td>Fem</td>
<td>Fem</td>
<td>ladki: roTi: Ka:ti: hai</td>
<td>ladki: ne roTi: Ka:yi:</td>
</tr>
</tbody>
</table>
A Hindi clause in different tenses

<table>
<thead>
<tr>
<th>Clause Type</th>
<th>Hindi Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPGenPres True</td>
<td>लड़की सेव खाती है</td>
</tr>
<tr>
<td>VPGenPres False</td>
<td>लड़की सेव नहीं खाती है</td>
</tr>
<tr>
<td>VPImpPast True</td>
<td>लड़की सेव खाती थी</td>
</tr>
<tr>
<td>VPImpPast False</td>
<td>लड़की सेव नहीं खाती थी</td>
</tr>
<tr>
<td>VPContPres True</td>
<td>लड़की सेव खा रही है</td>
</tr>
<tr>
<td>VPContPres False</td>
<td>लड़की सेव नहीं खा रही है</td>
</tr>
<tr>
<td>VPContPast True</td>
<td>लड़की सेव खा रही थी</td>
</tr>
<tr>
<td>VPContPast False</td>
<td>लड़की सेव नहीं खा रही थी</td>
</tr>
<tr>
<td>VPPref True</td>
<td>लड़की ने सेव खाया</td>
</tr>
<tr>
<td>VPPref False</td>
<td>लड़की ने सेव नहीं खाया</td>
</tr>
<tr>
<td>VPPrefPres True</td>
<td>लड़की सेव खायी है</td>
</tr>
<tr>
<td>VPPrefPres False</td>
<td>लड़की सेव नहीं खायी है</td>
</tr>
<tr>
<td>VPPrefPast True</td>
<td>लड़की सेव खायी थी</td>
</tr>
<tr>
<td>VPPrefPast False</td>
<td>लड़की सेव नहीं खायी थी</td>
</tr>
<tr>
<td>VPSubj True</td>
<td>लड़की सेव खाये</td>
</tr>
<tr>
<td>VPSubj False</td>
<td>लड़की सेव न खाये</td>
</tr>
<tr>
<td>VPFut True</td>
<td>लड़की सेव खायेगी</td>
</tr>
<tr>
<td>VPFut False</td>
<td>लड़की सेव न खायेगी</td>
</tr>
</tbody>
</table>

3.14 Exercises

1. Learn the commands `dependency_graph`, `print_grammar`, `system escape !`, and `system pipe ?`.

2. Write tables of examples of the key syntactic functions for your target languages, trying to include all possible forms.

3. Implement `Grammar` and `Lang` for your target language.

4. Even if you don’t know Italian, you may try this: add a parameter or something in `GrammarIta` to implement the rule that the participle in the perfect tense agrees in gender and number with an accusative clitic. Test this with the sentences `lei la ha amata` and `lei ci ha amati` (where the current grammar now gives `amato` in both...
cases).


## 4 Using the Resource Grammar Library in Applications

### Contents

Software libraries: programmer’s vs. users view
Semantic vs. syntactic grammars
Example of semantic grammar and its implementation
Interfaces and parametrized modules
Free variation
Overview of the Resource Grammar API

### 4.1 Software libraries

Collections of reusable functions/types/classes

API = **Application Programmer’s Interface**

- show enough to enable use
- hide details

Example: maps (lookup tables, hash maps) in Haskell, C++, Java, ...

```haskell
type Map
lookup : key -> Map -> val
update : key -> val -> Map -> Map
```

Hidden: the definition of the type `Map` and of the functions `lookup` and `update`.

### Advantages of software libraries

Programmers have
• less code to write (e.g. how to look up)
• less techniques to learn (e.g. efficient Map datastructures)

Improvements and bug fixes can be inherited

Grammars as software libraries

Smart paradigms as API for morphology

\[ \text{mkN} : (\text{talo} : \text{Str}) \rightarrow \text{N} \]

Abstract syntax as API for syntactic combinations

\[
\begin{align*}
\text{PredVP} & : \text{NP} \rightarrow \text{VP} \rightarrow \text{C1} \\
\text{ComplV2} & : \text{V2} \rightarrow \text{NP} \rightarrow \text{VP} \\
\text{NumCN} & : \text{Num} \rightarrow \text{CN} \rightarrow \text{NP}
\end{align*}
\]

4.2 Using the library: natural language output

Task: in an email program, generate phrases saying *you have n message(s)*

Problem: avoid *you have one messages*

Solution: use the library

\[
\begin{align*}
\text{PredVP youSg_NP} (\text{ComplV2 have_V2} (\text{NumCN two_Num} (\text{UseN (mkN "message"))})) \\
\Rightarrow & \text{ you have two messages} \\
\text{PredVP youSg_NP} (\text{ComplV2 have_V2} (\text{NumCN one_Num} (\text{UseN (mkN "message"))})) \\
\Rightarrow & \text{ you have one message}
\end{align*}
\]

Software localization

Adapt the email program to Italian, Swedish, Finnish...

\[
\begin{align*}
\text{PredVP youSg_NP} (\text{ComplV2 have_V2} (\text{NumCN two_Num} (\text{UseN (mkN "messaggio"))})) \\
\Rightarrow & \text{ hai due messaggi} \\
\text{PredVP youSg_NP} (\text{ComplV2 have_V2} (\text{NumCN two_Num} (\text{UseN (mkN "meddelande"))})) \\
\Rightarrow & \text{ du har två meddelanden}
\end{align*}
\]
The new languages are more complex than English - but only internally, not on the API level!

Correct number in Arabic

<table>
<thead>
<tr>
<th>Number</th>
<th>Arabic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>risālatun</td>
</tr>
<tr>
<td>2</td>
<td>risālatāni</td>
</tr>
<tr>
<td>(3-10)</td>
<td>rasāila</td>
</tr>
<tr>
<td>(11-99)</td>
<td>risālatan</td>
</tr>
<tr>
<td>x100</td>
<td>risālatin</td>
</tr>
</tbody>
</table>

(From "Implementation of the Arabic Numerals and their Syntax in GF" by Ali Dada, ACL workshop on Arabic, Prague 2007)

Use cases for grammar libraries

Grammars need very much very special knowledge, and a lot of work - thus an excellent topic for a software library!

Some applications where grammars have shown to be useful:

- software localization
- natural language generation (from formalized content)
- technical translation
- spoken dialogue systems

4.3 Two kinds of grammarians

Application grammarians vs. resource grammarians

<table>
<thead>
<tr>
<th>grammarian</th>
<th>applications</th>
<th>resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>expertise</td>
<td>application domain</td>
<td>linguistics</td>
</tr>
<tr>
<td>programming skills</td>
<td>programming in general</td>
<td>GF programming</td>
</tr>
<tr>
<td>language skills</td>
<td>practical use</td>
<td>theoretical knowledge</td>
</tr>
</tbody>
</table>
We want a division of labour.

---

Two kinds of grammars---

Application grammars vs. resource grammars

<table>
<thead>
<tr>
<th>grammar</th>
<th>application</th>
<th>resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract syntax</td>
<td>semantic</td>
<td>syntactic</td>
</tr>
<tr>
<td>concrete syntax</td>
<td>using resource API</td>
<td>parameters, tables, records</td>
</tr>
<tr>
<td>lexicon</td>
<td>idiomatic, technical</td>
<td>just for testing</td>
</tr>
<tr>
<td>size</td>
<td>small or bigger</td>
<td>big</td>
</tr>
</tbody>
</table>

A.k.a. semantic grammars vs. syntactic grammars.

4.4 Meaning-preserving translation

Translation must preserve meaning.
It need not preserve syntactic structure.
Sometimes it is even impossible:

- *John likes Mary* in Italian is *Maria piace a Giovanni*

The abstract syntax in the semantic grammar is a logical predicate:

fun Like : Person -> Person -> Fact
lin Like x y = x ++ "likes" ++ y -- English
lin Like x y = y ++ "piace" ++ "a" ++ x -- Italian

Translation and resource grammar

To get all grammatical details right, we use resource grammar and not strings

lincat Person = NP ; Fact = Cl ;
lin Like x y = PredVP x (ComplV2 like_V2 y) -- English
lin Like x y = PredVP y (ComplV2 piacere_V2 x) -- Italian

From syntactic point of view, we perform transfer, i.e. structure change.
GF has compile-time transfer, and uses interlingua (semantic abstract syntax) at run time.
Domain semantics

"Semantics of English”, or of any other natural language as a whole, has never been built.

It is more feasible to have semantics of fragments - of small, well-understood parts of natural language.

Such languages are called domain languages, and their semantics, domain semantics.

Domain semantics = ontology in the Semantic Web terminology.

Examples of domain semantics

Expressed in various formal languages

- mathematics, in predicate logic
- software functionality, in UML/OCL
- dialogue system actions, in SISR
- museum object descriptions, in OWL

GF abstract syntax can be used for any of these!

4.5 Example: abstract syntax for a ”Face” community

What messages can be expressed on the community page?

abstract Face = {

  flags startcat = Message ;

  cat
    Message ; Person ; Object ; Number ;
  fun
    Have : Person -> Number -> Object -> Message ; -- p has n o’s
    Like : Person -> Object -> Message ; -- p likes o
    You : Person ;
    Friend, Invitation : Object ;
    One, Two, Hundred : Number ;
  }

Notice the startcat flag, as the start category isn’t S.
Presenting the resource grammar

In practice, the abstract syntax of Resource Grammar is inconvenient

- too deep structures, too much code to write
- too many names to remember

We do the same as in morphology: overloaded operations, named \( mkC \) where \( C \) is the value category.

The resource defines e.g.

\[
\begin{align*}
\text{mkCl} : \text{NP} & \rightarrow \text{V2} \rightarrow \text{NP} \rightarrow \text{Cl} = \text{subj},\verb,\text{obj} \rightarrow \\
& \quad \text{PredVP subj (Comp1V2 verb obj)} \\
\text{mkCl} : \text{NP} & \rightarrow \text{V} \rightarrow \text{Cl} = \text{subj},\verb \rightarrow \\
& \quad \text{PredVP subj (UseV verb)}
\end{align*}
\]

Relevant part of Resource Grammar API for ”Face”

These functions (some of which are structural words) are used.

<table>
<thead>
<tr>
<th>Function</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{mkCl} : \text{NP} \rightarrow \text{V2} \rightarrow \text{NP} \rightarrow \text{Cl} )</td>
<td>John loves Mary</td>
</tr>
<tr>
<td>( \text{mkNP} : \text{Numeral} \rightarrow \text{CN} \rightarrow \text{NP} )</td>
<td>five cars</td>
</tr>
<tr>
<td>( \text{mkNP} : \text{Quant} \rightarrow \text{CN} \rightarrow \text{NP} )</td>
<td>that car</td>
</tr>
<tr>
<td>( \text{mkNP} : \text{Pron} \rightarrow \text{NP} )</td>
<td>we</td>
</tr>
<tr>
<td>( \text{mkCN} : \text{N} \rightarrow \text{CN} )</td>
<td>car</td>
</tr>
<tr>
<td>( \text{thisQuant} : \text{Quant} )</td>
<td>this, these</td>
</tr>
<tr>
<td>( \text{youSgPron} : \text{Pron} )</td>
<td>you (singular)</td>
</tr>
<tr>
<td>( \text{n1Numeral}, \text{n2Numeral} : \text{Numeral} )</td>
<td>one, two</td>
</tr>
<tr>
<td>( \text{n100Numeral} : \text{Numeral} )</td>
<td>one hundred</td>
</tr>
<tr>
<td>( \text{haveV2} : \text{V2} )</td>
<td>have</td>
</tr>
</tbody>
</table>

Concrete syntax for English

How are messages expressed by using the library?

```plaintext
concrete FaceEng of Face = open SyntaxEng, ParadigmsEng in {
    lincat
    Message = Cl ;
    Person = NP ;
```
Concrete syntax for Finnish

Exactly the same rules of combination, only different words:

```plaintext
congre PhaseFin of Face = open SyntaxFin, ParadigmsFin in {
  lincat
    Message = Cl ;
    Person = NP ;
    Object = CN ;
    Number = Numeral ;
  lin
    Have p n o = mkCl p have_V2 (mkNP n o) ;
    Like p o = mkCl p like_V2 (mkNP this_Quant o) ;
    You = mkNP youSg_Pron ;
    Friend = mkCN friend_N ;
    Invitation = mkCN invitation_N ;
    One = n1_Numeral ;
    Two = n2_Numeral ;
    Hundred = n100_Numeral ;
  oper
    like_V2 = mkV2 "like" ;
    invitation_N = mkN "invitation" ;
    friend_N = mkN "friend" ;
}
```
4.6 Functors and interfaces

English and Finnish: the same combination rules, only different words!
Can we avoid repetition of the \texttt{lincat} and \texttt{lin} code? Yes!
New module type: \texttt{functor}, a.k.a. \texttt{incomplete} or \texttt{parametrized} module

\begin{verbatim}
awkward concrete FaceI of Face = open Syntax, LexFace in ...
\end{verbatim}

A functor may open \texttt{interfaces}.
An interface has \texttt{oper} declarations with just a type, no definition.
Here, \texttt{Syntax} and \texttt{LexFace} are interfaces.

The domain lexicon interface

\texttt{Syntax} is the Resource Grammar interface, and gives

- combination rules
- structural words

Content words are not given in \texttt{Syntax}, but in a \texttt{domain lexicon}

\begin{verbatim}
interface LexFace = open Syntax in {

oper
  like\_V2 : V2 ;
  invitation\_N : N ;
  friend\_N : N ;
}
\end{verbatim}

Concrete syntax functor ”FaceI”

\begin{verbatim}
incomplete concrete FaceI of Face = open Syntax, LexFace in {

lincat
  Message = Cl ;
  Person = NP ;
  Object = CN ;
  Number = Numeral ;
lin
\end{verbatim}
Have $p \ n \ o = \text{mkCl}_p \ \text{have}_V^2 (\text{mkNP} \ n \ o)$;
Like $p \ o = \text{mkCl}_p \ \text{like}_V^2 (\text{mkNP} \ \text{this}_\text{Quant} \ o)$;
You = \text{mkNP} \ \text{youSg}_\text{Pron}$;
Friend = \text{mkCN} \ \text{friend}_N$;
Invitation = \text{mkCN} \ \text{invitation}_N$;
One = n1$_\text{Numeral}$;
Two = n2$_\text{Numeral}$;
Hundred = n100$_\text{Numeral}$;

An English instance of the domain lexicon

Define the domain words in English

instance LexFaceEng of LexFace = open SyntaxEng, ParadigmsEng in {
  oper
  like$_V^2 = \text{mkV2} "like"$;
  invitation$_N = \text{mkN} "invitation"$;
  friend$_N = \text{mkN} "friend"$;
}

Put everything together: functor instantiation

Instantiate the functor FaceI by giving instances to its interfaces

--# -path=./:present

congrete FaceEng of Face = FaceI with
  (Syntax = SyntaxEng),
  (LexFace = LexFaceEng)$;

Also notice the domain search path.

Porting the grammar to Finnish

1. Domain lexicon: use Finnish paradigms and words

instance LexFaceFin of LexFace = open SyntaxFin, ParadigmsFin in {
  oper
  like$_V^2 = \text{mkV2} (\text{mkV} "pitää") \text{elative}$;
invitation_N = mkN "kutsu" ;
friend_N = mkN "ystävää" ;
}

2. Functor instantiation: mechanically change Eng to Fin

--# -path=.:present

congrete FaceFin of Face = FaceI with
  (Syntax = SyntaxFin),
  (LexFace = LexFaceFin) ;

4.7 Modules of a domain grammar: "Face" community

1. Abstract syntax, Face
2. Parametrized concrete syntax: FaceI
3. Domain lexicon interface: LexFace
4. For each language $L$: domain lexicon instance LexFace$L$
5. For each language $L$: concrete syntax instantiation Face$L$

Module dependency graph

red = to do, orange = to do (trivial), blue = to do (once), green = library
Porting the grammar to Italian

1. Domain lexicon: use Italian paradigms and words

```plaintext
instance LexFaceIta of LexFace = open SyntaxIta, ParadigmsIta in {
    oper
    like_V2 = mkV2 (mkV (piacere_64 "piacere")) dative ;
    invitation_N = mkN "invito" ;
    friend_N = mkN "amico" ;
}
```

2. Functor instantiation: restricted inheritance, excluding Like

```plaintext
concrete FaceIta of Face = FaceI - [Like] with
    (Syntax = SyntaxIta),
    (LexFace = LexFaceIta) ** open SyntaxIta in {
    lin Like p o =
        mkCl (mkNP this_Quant o) like_V2 p ;
}
```

4.8 Free variation

There can be many ways of expressing a given semantic structure.
This can be expressed by the variant operator |.

```plaintext
fun BuyTicket : City -> City -> Request
lin BuyTicket x y =
    (("I want" ++ ("to buy" | []) ++ ("a ticket")) | "to go")
    | (("can you" | [] ) ++ "give me" ++ "a ticket")
    | [] ++
    "from" ++ x ++ "to" ++ y
```

The variants can of course be resource grammar expressions as well.

4.9 Overview of the resource grammar API

For the full story, see the resource grammar synopsis in
Main division:

- **Syntax**, common to all languages
- **Paradigms** $L$, specific to language $L$

Main categories and their dependencies
Categories of complex phrases

<table>
<thead>
<tr>
<th>Category</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>sequence of utterances</td>
<td><em>Does John walk? Yes.</em></td>
</tr>
<tr>
<td>Utt</td>
<td>utterance</td>
<td><em>does John walk</em></td>
</tr>
<tr>
<td>Imp</td>
<td>imperative</td>
<td><em>don’t walk</em></td>
</tr>
<tr>
<td>S</td>
<td>sentence (fixed tense)</td>
<td><em>John wouldn’t walk</em></td>
</tr>
<tr>
<td>QS</td>
<td>question sentence</td>
<td><em>who hasn’t walked</em></td>
</tr>
<tr>
<td>C1</td>
<td>clause (variable tense)</td>
<td><em>John walks</em></td>
</tr>
<tr>
<td>QC1</td>
<td>question clause</td>
<td><em>who doesn’t walk</em></td>
</tr>
<tr>
<td>VP</td>
<td>verb phrase</td>
<td><em>love her</em></td>
</tr>
<tr>
<td>AP</td>
<td>adjectival phrase</td>
<td><em>very young</em></td>
</tr>
<tr>
<td>CN</td>
<td>common noun phrase</td>
<td><em>young man</em></td>
</tr>
<tr>
<td>Adv</td>
<td>adverbial phrase</td>
<td><em>in the house</em></td>
</tr>
</tbody>
</table>

Lexical categories for building predicates

<table>
<thead>
<tr>
<th>Cat</th>
<th>Explanation</th>
<th>Compl</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>one-place adjective</td>
<td>-</td>
<td><em>smart</em></td>
</tr>
<tr>
<td>A2</td>
<td>two-place adjective</td>
<td>NP</td>
<td><em>married (to her)</em></td>
</tr>
<tr>
<td>Adv</td>
<td>adverb</td>
<td>-</td>
<td><em>here</em></td>
</tr>
<tr>
<td>N</td>
<td>common noun</td>
<td>-</td>
<td><em>man</em></td>
</tr>
<tr>
<td>N2</td>
<td>relational noun</td>
<td>NP</td>
<td><em>friend (of John)</em></td>
</tr>
<tr>
<td>NP</td>
<td>noun phrase</td>
<td>-</td>
<td><em>the boss</em></td>
</tr>
<tr>
<td>V</td>
<td>one-place verb</td>
<td>-</td>
<td><em>sleep</em></td>
</tr>
<tr>
<td>V2</td>
<td>two-place verb</td>
<td>NP</td>
<td><em>love (her)</em></td>
</tr>
<tr>
<td>V3</td>
<td>three-place verb</td>
<td>NP, NP</td>
<td><em>show (it to me)</em></td>
</tr>
<tr>
<td>VS</td>
<td>sentence-complement verb</td>
<td>S</td>
<td><em>say (that I run)</em></td>
</tr>
<tr>
<td>VV</td>
<td>verb-complement verb</td>
<td>VP</td>
<td><em>want (to run)</em></td>
</tr>
</tbody>
</table>
Functions for building predication clauses

<table>
<thead>
<tr>
<th>Fun</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkCl</td>
<td>NP -&gt; V -&gt; Cl</td>
<td>John walks</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; V2 -&gt; NP -&gt; Cl</td>
<td>John loves her</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; V3 -&gt; NP -&gt; NP -&gt; Cl</td>
<td>John sends it to her</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; VV -&gt; VP -&gt; Cl</td>
<td>John wants to walk</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; VS -&gt; S -&gt; Cl</td>
<td>John says that it is good</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; A -&gt; Cl</td>
<td>John is old</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; A -&gt; NP -&gt; Cl</td>
<td>John is older than Mary</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; A2 -&gt; NP -&gt; Cl</td>
<td>John is married to her</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; AP -&gt; Cl</td>
<td>John is very old</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; N -&gt; Cl</td>
<td>John is a man</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; CN -&gt; Cl</td>
<td>John is an old man</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; NP -&gt; Cl</td>
<td>John is the man</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP -&gt; Adv -&gt; Cl</td>
<td>John is here</td>
</tr>
</tbody>
</table>

Noun phrases and common nouns

<table>
<thead>
<tr>
<th>Fun</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkNP</td>
<td>Quant -&gt; CN -&gt; NP</td>
<td>this man</td>
</tr>
<tr>
<td>mkNP</td>
<td>Numeral -&gt; CN -&gt; NP</td>
<td>five men</td>
</tr>
<tr>
<td>mkNP</td>
<td>PN -&gt; NP</td>
<td>John</td>
</tr>
<tr>
<td>mkNP</td>
<td>Pron -&gt; NP</td>
<td>we</td>
</tr>
<tr>
<td>mkNP</td>
<td>Quant -&gt; Num -&gt; CN -&gt; NP</td>
<td>these (five) man</td>
</tr>
<tr>
<td>mkCN</td>
<td>N -&gt; CN</td>
<td>man</td>
</tr>
<tr>
<td>mkCN</td>
<td>A -&gt; N -&gt; CN</td>
<td>old man</td>
</tr>
<tr>
<td>mkCN</td>
<td>AP -&gt; CN -&gt; CN</td>
<td>very old Chinese man</td>
</tr>
<tr>
<td>mkNum</td>
<td>Numeral -&gt; Num</td>
<td>five</td>
</tr>
<tr>
<td>n100_Numeral</td>
<td>Numeral</td>
<td>one hundred</td>
</tr>
<tr>
<td>plNum</td>
<td>Num</td>
<td>(plural)</td>
</tr>
</tbody>
</table>
Questions and interrogatives

<table>
<thead>
<tr>
<th>Fun</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkQCl</td>
<td>Cl -&gt; QC1</td>
<td>does John walk</td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; V -&gt; QC1</td>
<td>who walks</td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; V2 -&gt; NP -&gt; QC1</td>
<td>who loves her</td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; NP -&gt; V2 -&gt; QC1</td>
<td>whom does she love</td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; AP -&gt; QC1</td>
<td>who is old</td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; NP -&gt; QC1</td>
<td>who is the boss</td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; Adv -&gt; QC1</td>
<td>who is here</td>
</tr>
<tr>
<td>mkQCl</td>
<td>IAdv -&gt; Cl -&gt; QC1</td>
<td>where does John walk</td>
</tr>
<tr>
<td>mkIP</td>
<td>CN -&gt; IP</td>
<td>which car</td>
</tr>
<tr>
<td>who_IP</td>
<td>IP</td>
<td>who</td>
</tr>
<tr>
<td>why_IAdv</td>
<td>IAdv</td>
<td>why</td>
</tr>
<tr>
<td>where_IAdv</td>
<td>IAdv</td>
<td>where</td>
</tr>
</tbody>
</table>

Sentence formation, tense, and polarity

<table>
<thead>
<tr>
<th>Fun</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkS</td>
<td>Cl -&gt; S</td>
<td>he walks</td>
</tr>
<tr>
<td>mkS</td>
<td>(Tense)-&gt;(Ant)-&gt;(Pol)-&gt;Cl -&gt; S</td>
<td>he wouldn’t have walked</td>
</tr>
<tr>
<td>mkQS</td>
<td>QC1 -&gt; QS</td>
<td>does he walk</td>
</tr>
<tr>
<td>mkQS</td>
<td>(Tense)-&gt;(Ant)-&gt;(Pol)-&gt;QC1 -&gt; QS</td>
<td>wouldn’t he have walked</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>conditionalTense</td>
<td>Tense</td>
<td>(he would walk)</td>
</tr>
<tr>
<td>futureTense</td>
<td>Tense</td>
<td>(he will walk)</td>
</tr>
<tr>
<td>pastTense</td>
<td>Tense</td>
<td>(he walked)</td>
</tr>
<tr>
<td>presentTense</td>
<td>Tense</td>
<td>(he walks)</td>
</tr>
<tr>
<td>anteriorAnt</td>
<td>Ant</td>
<td>(he has walked)</td>
</tr>
<tr>
<td>negativePol</td>
<td>Pol</td>
<td>(he doesn’t walk)</td>
</tr>
</tbody>
</table>
Utterances and imperatives

<table>
<thead>
<tr>
<th>Fun</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkUtt</td>
<td>C1 -&gt; Utt</td>
<td>he walks</td>
</tr>
<tr>
<td>mkUtt</td>
<td>S -&gt; Utt</td>
<td>he didn’t walk</td>
</tr>
<tr>
<td>mkUtt</td>
<td>QS -&gt; Utt</td>
<td>who didn’t walk</td>
</tr>
<tr>
<td>mkUtt</td>
<td>Imp -&gt; Utt</td>
<td>walk</td>
</tr>
<tr>
<td>mkImp</td>
<td>V -&gt; Imp</td>
<td>walk</td>
</tr>
<tr>
<td>mkImp</td>
<td>V2 -&gt; NP -&gt; Imp</td>
<td>find it</td>
</tr>
<tr>
<td>mkImp</td>
<td>AP -&gt; Imp</td>
<td>be brave</td>
</tr>
</tbody>
</table>

More

Relative clauses: man who owns a donkey
Adverbs: in the house
Subjunction: if a man owns a donkey
Coordination: John and Mary are English or American

4.10 Exercises

1. Compile and make available the resource grammar library, latest version. Compilation is by make in GF/lib/src. Make it available by setting GF_LIB_PATH to GF/lib.

2. Compile and test the grammars face/FaceL (available in course source files).

3. Write a concrete syntax of Face for some other resource language by adding a domain lexicon and a functor instantiation.

4. Add functions to Face and write their concrete syntax for at least some language.

5. Design your own domain grammar and implement it for some languages.

5 Developing a GF Resource Grammar

Contents

Module structure
Statistics
How to start building a new language
How to test a resource grammar
The Assignment

5.1 The principal module structure

Division of labour

Written by the resource grammarian:

- concrete of the row from Structural to Verb
- concrete of Cat and Lexicon
- Paradigms
- abstract and concrete of Extra, Irreg

Already given or derived mechanically:

- all abstract modules except Extra, Irreg
- concrete of Common, Grammar, Lang, All
- Constructors, Syntax, Try
Roles of modules: Library API

**Syntax:** syntactic combinations and structural words

**Paradigms:** morphological paradigms

**Try:** (almost) everything put together

**Constructors:** syntactic combinations only

**Irreg:** irregularly inflected words (mostly verbs)

Roles of modules: Top-level grammar

**Lang:** common syntax and lexicon

**All:** common grammar plus language-dependent extensions

**Grammar:** common syntax

**Structural:** lexicon of structural words

**Lexicon:** test lexicon of 300 content words

**Cat:** the common type system

**Common:** concrete syntax mostly common to languages

Roles of modules: phrase categories

<table>
<thead>
<tr>
<th>module</th>
<th>scope</th>
<th>value categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjective</td>
<td>adjectives</td>
<td>AP</td>
</tr>
<tr>
<td>Adverb</td>
<td>adverbial phrases</td>
<td>AdN, Adv</td>
</tr>
<tr>
<td>Conjunction</td>
<td>coordination</td>
<td>Adv, AP, NP, RS, S</td>
</tr>
<tr>
<td>Idiom</td>
<td>idiomatic expressions</td>
<td>Cl, QCl, VP, Utt</td>
</tr>
<tr>
<td>Noun</td>
<td>noun phrases and nouns</td>
<td>Card, CN, Det, NP, Num, Ord</td>
</tr>
<tr>
<td>Numeral</td>
<td>cardinals and ordinals</td>
<td>Digit, Numeral</td>
</tr>
<tr>
<td>Phrase</td>
<td>suprasentential phrases</td>
<td>PConj, Phr, Utt, Voc</td>
</tr>
<tr>
<td>Question</td>
<td>questions and interrogatives</td>
<td>IAdv, IComp, IDet, IP, QCl</td>
</tr>
<tr>
<td>Relative</td>
<td>relat. clauses and pronouns</td>
<td>RCl, RP</td>
</tr>
<tr>
<td>Sentence</td>
<td>clauses and sentences</td>
<td>Cl, Imp, QS, RS, S, SC, SSlash</td>
</tr>
<tr>
<td>Text</td>
<td>many-phrase texts</td>
<td>Text</td>
</tr>
<tr>
<td>Verb</td>
<td>verb phrases</td>
<td>Comp, VP, VPSlash</td>
</tr>
</tbody>
</table>
Type discipline and consistency

Producers: each phrase category module is the producer of value categories listed on previous slide.

Consumers: all modules may use any categories as argument types.

Contract: the module Cat defines the type system common for both consumers and producers.

Different grammarians may safely work on different producers.

This works even for mutual dependencies of categories:

Sentence.UseCl : Temp -> Pol -> Cl -> S -- S uses Cl
Sentence.PredVP : VP -> NP -> Cl -- uses VP
Verb.ComplVS : VS -> S -> VP -- uses S

Auxiliary modules

resource modules provided by the library:

- Prelude and Predef: string operations, booleans
- Coordination: generic formation of list conjunctions
- ParamX: commonly used parameter, such as Number = Sg | Pl

resource modules up to the grammarian to write:

- Res: language specific parameter types, morphology, VP formation
- Morpho, Phono,...: possible division of Res to more modules

Dependencies

Most phrase category modules:

concrete VerbGer of Verb = CatGer ** open ResGer, Prelude in ...

Conjunction:

concrete ConjunctionGer of Conjunction = CatGer **
open Coordination, ResGer, Prelude in ...
Lexicon:

concrete LexiconGer of Lexicon = CatGer **
    open ParadigmsGer, IrregGer in {

Functional programming style

The Golden Rule: Whenever you find yourself programming by copy and paste, write a function instead!

- Repetition inside one definition: use a let expression
- Repetition inside one module: define an oper in the same module
- Repetition in many modules: define an oper in the Res module
- Repetition of an entire module: write a functor

Functors in the Resource Grammar Library

Used in families of languages

- Romance: Catalan, French, Italian, Spanish
- Scandinavian: Danish, Norwegian, Swedish

Structure:

- Common, a common resource for the family
- Diff, a minimal interface extended by interface Res
- Cat and phrase structure modules are functors over Res
- Idiom, Structural, Lexicon, Paradigms are ordinary modules

Example: DiffRomance

Words and morphology are of course different, in ways we haven’t tried to formalize.
In syntax, there are just eight parameters that fundamentally make the difference:
Prepositions that fuse with the article (Fre, Spa de, a; Ita also con, da, in, su).

    param Prepos ;
Which types of verbs exist, in terms of auxiliaries. (Fre, Ita avoir, être, and refl; Spa only haber and refl).

    param VType ;

Derivatively, if/when the participle agrees to the subject. (Fre elle est partie, Ita lei è partita, Spa not)

    oper partAgr : VType -> VPagr ;

Whether participle agrees to foregoing clitic. (Fre je l’ai vue, Spa yo la he visto)

    oper vpAgrClit : Agr -> VPagr ;

Whether a preposition is repeated in conjunction (Fre la somme de 3 et de 4, Ita la somma di 3 e 4).

    oper conjunctCase : NPForm -> NPForm ;

How infinitives and clitics are placed relative to each other (Fre la voir, Ita vederla). The Bool is used for indicating if there are any clitics.

    oper clitInf : Bool -> Str -> Str -> Str ;

To render pronominal arguments as clitics and/or ordinary complements. Returns True if there are any clitics.

    oper pronArg : Number -> Person -> CAgr -> CAgr -> Str * Str * Bool ;

To render imperatives (with their clitics etc).

    oper mkImperative : Bool -> Person -> VPC -> {s : Polarity => AAgr => Str} ;

**Pros and cons of functors**

+ intellectual satisfaction: linguistic generalizations
+ code can be shared: of syntax code, 75% in Romance and 85% in Scandinavian
+ bug fixes and maintenance can often be shared as well
+ adding a new language of the same family can be very easy
- difficult to get started with proper abstractions
- new languages may require extensions of interfaces

Workflow: don’t start with a functor, but do one language normally, and refactor it to an interface, functor, and instance.
Suggestions about functors for new languages

Romance: Portuguese probably using functor, Romanian probably independent
Germanic: Dutch maybe by functor from German, Icelandic probably independent
Slavic: Bulgarian and Russian are not functors, maybe one for Western Slavic (Czech, Slovak, Polish) and Southern Slavic (Bulgarian)
Fenno-Ugric: Estonian maybe by functor from Finnish
Indo-Aryan: Hindi and Urdu most certainly via a functor
Semitic: Arabic, Hebrew, Maltese probably independent

5.2 Effort statistics, completed languages

<table>
<thead>
<tr>
<th>language</th>
<th>syntax</th>
<th>morpho</th>
<th>lex</th>
<th>total</th>
<th>months</th>
<th>started</th>
</tr>
</thead>
<tbody>
<tr>
<td>common</td>
<td>413</td>
<td>-</td>
<td>-</td>
<td>413</td>
<td>2</td>
<td>2001</td>
</tr>
<tr>
<td>abstract</td>
<td>729</td>
<td>-</td>
<td>468</td>
<td>1197</td>
<td>24</td>
<td>2001</td>
</tr>
<tr>
<td>Bulgarian</td>
<td>1200</td>
<td>2329</td>
<td>502</td>
<td>4031</td>
<td>3</td>
<td>2008</td>
</tr>
<tr>
<td>English</td>
<td>1025</td>
<td>772</td>
<td>506</td>
<td>2303</td>
<td>6</td>
<td>2001</td>
</tr>
<tr>
<td>Finnish</td>
<td>1471</td>
<td>1490</td>
<td>703</td>
<td>3664</td>
<td>6</td>
<td>2003</td>
</tr>
<tr>
<td>German</td>
<td>1337</td>
<td>604</td>
<td>492</td>
<td>2433</td>
<td>6</td>
<td>2002</td>
</tr>
<tr>
<td>Russian</td>
<td>1492</td>
<td>3668</td>
<td>534</td>
<td>5694</td>
<td>18</td>
<td>2002</td>
</tr>
<tr>
<td>Romance</td>
<td>1346</td>
<td>-</td>
<td>-</td>
<td>1346</td>
<td>10</td>
<td>2003</td>
</tr>
<tr>
<td>Catalan</td>
<td>521</td>
<td>*9000</td>
<td>518</td>
<td>*10039</td>
<td>4</td>
<td>2006</td>
</tr>
<tr>
<td>French</td>
<td>468</td>
<td>1789</td>
<td>514</td>
<td>2771</td>
<td>6</td>
<td>2002</td>
</tr>
<tr>
<td>Italian</td>
<td>423</td>
<td>*7423</td>
<td>500</td>
<td>*8346</td>
<td>3</td>
<td>2003</td>
</tr>
<tr>
<td>Spanish</td>
<td>417</td>
<td>*6549</td>
<td>516</td>
<td>*7482</td>
<td>3</td>
<td>2004</td>
</tr>
<tr>
<td>Scandinavian</td>
<td>1293</td>
<td>-</td>
<td>-</td>
<td>1293</td>
<td>4</td>
<td>2005</td>
</tr>
<tr>
<td>Danish</td>
<td>262</td>
<td>683</td>
<td>486</td>
<td>1431</td>
<td>2</td>
<td>2005</td>
</tr>
<tr>
<td>Norwegian</td>
<td>281</td>
<td>676</td>
<td>488</td>
<td>1445</td>
<td>2</td>
<td>2005</td>
</tr>
<tr>
<td>Swedish</td>
<td>280</td>
<td>717</td>
<td>491</td>
<td>1488</td>
<td>4</td>
<td>2001</td>
</tr>
<tr>
<td>total</td>
<td>12545</td>
<td>*36700</td>
<td>6718</td>
<td>*55963</td>
<td>103</td>
<td>2001</td>
</tr>
</tbody>
</table>

Lines of source code in April 2009, rough estimates of person months. * = generated code.

5.3 How to start building a language, e.g. Marathi

1. Create a directory GF/lib/src/marathi
2. Check out the ISO 639-3 language code: Mar
3. Copy over the files from the closest related language, e.g. hindi
4. Rename files marathi/*Hin.gf to marathi/*Mar.gf
5. Change imports of Hin modules to imports of Mar modules
6. Comment out every line between header { and the final }
7. Now you can import your (empty) grammar: i marathi/LangMar.gf

Suggested order for proceeding with a language

1. ResMar: parameter types needed for nouns
2. CatMar: lincat N
3. ParadigmsMar: some regular noun paradigms
4. LexiconMar: some words that the new paradigms cover
5. (1.-4.) for V, maybe with just present tense
6. ResMar: parameter types needed for Cl, CN, Det, NP, Quant, VP
7. CatMar: lincat Cl, CN, Det, NP, Quant, VP
9. VerbMar: lin UseV
10. SentenceMar: lin PredVP

Character encoding for non-ASCII languages

GF internally: 32-bit unicode
Generated files (.gfo, .pgf): UTF-8
Source files: whatever you want, but use a flag if not isolatin-1.
UTF-8 and cp1251 (Cyrillic) are possible in strings, but not in identifiers. The module must contain

    flags coding = utf8 ; -- OR coding = cp1251

Transliterations are available for many alphabets (see help unicode_table).

Using transliteration

This is what you have to add in GF/src/GF/Text/Transliterations.hs

    transHebrew :: Transliteration
    transHebrew = mkTransliteration allTrans allCodes where
```
allTrans = words $ 
  "A b g d h w z H T y K k l M m N" ++
  "n S O P p Z. Z q r s t -- - - - - - - - - - - - - - - - - - - - - - - - " ++
  "w2 w3 y2 g1 g2"
allCodes = [0x05d0..0x05f4]
```

Also edit a couple of places in GF/src/GF/Command/Commands.hs.
You can later convert the file to UTF-8 (see help put_string).

**Diagnosis methods along the way**

Make sure you have a compilable LangMar at all times!
Use the GF command `pg -missing` to check which functions are missing.
Use the GF command `gr -cat=C | l -table` to test category C

**Regression testing with a treebank**

Build and maintain a treebank: a set of trees with their linearizations:

1. Create a file `test.trees` with just trees, one by line.
2. Linearize each tree to all forms, possibly with English for comparison.

```
> i english/LangEng.gf
> i marathi/LangMar.gf
> rf -lines -tree -file=test.trees | l -all -treebank | wf -file=test.treebank
```

3. Create a gold standard `gold.treebank` from `test.treebank` by manually correcting the Marathi linearizations.
4. Compare with the Unix command `diff test.treebank gold.treebank`
5. Rerun (2.) and (4.) after every change in concrete syntax; extend the tree set and the gold standard after every new implemented function.

**Sources**

A good grammar book

- lots of inflection paradigms
- reasonable chapter on syntax
• traditional terminology for grammatical concepts

A good dictionary

• inflection information about words
• verb subcategorization (i.e. case and preposition of complements)

Wikipedia article on the language
Google as "gold standard": is it rucola or ruccola?
Google translation for suggestions (can’t be trusted, though!)

Compiling the library

The current development library sources are in GF/lib/src.
Use make in this directory to compile the libraries.
Use runghc Make lang api langs=Mar to compile just the language Mar.

5.4 Assignment: a good start

1. Build a directory and a set of files for your target language.
2. Implement some categories, morphological paradigms, and syntax rules.
3. Give the lin rules of at least 100 entries in Lexicon.
4. Send us: your source files and a treebank of 100 trees with linearizations in English and your target language. These linearizations should be correct, and directly generated from your grammar implementation.