GF Tutorial for Resource Grammar Writers

Aarne Ranta

April 2009
Goals

Fast way to resource grammar writing

Practical use of GF

Linguistic concepts in the resource grammar library

For full details, read the Tutorial at GF homepage, digitalgrammars.com/gf
Contents of the course’s five lectures

1. The GF system, simple multilingual grammars

2. Morphological paradigms and lexica

3. Building up a linguistic syntax

4. Using the Resource Grammar Library in applications

5. Inside the Resource Grammar Library
Lecture 1

The GF system

Simple Multilingual Grammars
Contents

What GF is

Installing the GF system

A grammar for *John loves Mary* in English, French, Latin, German, Hebrew

Testing grammars and building applications

The scope of the Resource Grammar Library

Exercises
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

\[
\text{Pred Mary (Compl Love John)}
\]

Marie aime Jean

Maria Ioannem amat

מריא איוナンן אמאן ג’ון

מריה אימז ג’אן
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 2009) 13 languages: Bulgarian, Catalan, Danish, English, Finnish, French, German, Interlingua, Italian, Norwegian, Russian, Spanish, Swedish.

Under construction for 7 languages: Arabic, Hindi, Latin, Polish, Romanian, Thai, Turkish.
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: tournesol.cs.chalmers.se:41296/fridge

- translator demo: tournesol.cs.chalmers.se:41296/translate
Download and install 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 3.0-beta3.
License: see help -license.
Differences from GF 2.9: see help -changes.
Bug reports: http://trac.haskell.org/gf/

Languages:
>
Using the GF shell: help

Command \texttt{h} = \texttt{help}

\texttt{\textgreater{} help}

gives a list of commands with short descriptions.

\texttt{\textgreater{} help parse}

gives detailed help on the command \texttt{parse}.

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

Pred: S

/     \
John: NP  Compl: VP

/     \
Love: V2  Mary: NP

> p "John loves Mary" | visualize_tree -view=open
Abstract and concrete syntax

A context-free rule

Pred. S ::= NP VP

defines two things:

- **abstract syntax**: build a tree of form Pred np vp

- **concrete syntax**: this tree linearizes to a string of form np vp

The main idea of GF: separate these two things.
Separating abstract and concrete syntax

A context-free rule is converted to two judgments in GF:

- fun, declaring a syntactic function
- lin, giving its linearization rule

Pred. $S ::= NP \text{ } VP \impliesfun$ Pred : $NP \rightarrow VP \rightarrow S$
lin Pred np vp = np ++ vp
Functions and concatenation

Function type: $A \rightarrow B \rightarrow C$, 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><code>cat C</code></td>
<td><code>C</code> is a category</td>
</tr>
<tr>
<td><code>fun f : T</code></td>
<td><code>f</code> is a function of type <code>T</code></td>
</tr>
<tr>
<td><code>lincat C = L</code></td>
<td><code>C</code> has linearization type <code>L</code></td>
</tr>
<tr>
<td><code>lin f xs = t</code></td>
<td><code>f xs</code> has linearization <code>t</code></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.
Multilingual grammar

One abstract + many concretes

The same system of trees can be given

- different words

- different word orders

- different linearization types
Concrete syntax, French

congечет 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

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

Translation: pipe linearization to parsing

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

Multilingual generation: linearize into all languages

> gr | l
Pred Mary (Compl Love Mary)
Mary loves Mary
Marie aime Marie
Multilingual treebank

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

```
> 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):

```
param Case = Nom | 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,

\[
\begin{align*}
\text{Pred np vp} &= \text{np ! Nom ++ vp} \\
\text{Compl v2 np} &= \text{np ! Acc ++ v2}
\end{align*}
\]
Concrete syntax, German

concrete ZeroGer of Zero = {
    lincat
        S, NP, VP = Str ;
        V2 = {v : Str ; p : Str} ;
    lin
        Pred np vp = np ++ vp ;
        Compl v2 np = v2.v ++ np ++ v2.p ;
        John = "Johann" ;
        Mary = "Maria" ;
        Love = {v = "hat" ; p = "lieb"} ;
}

The verb hat lieb is a discontinuous constituent: Johann hat Maria lieb (contrived, since we also have liebt).
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** ($.$)

$$\text{lin Compl } v_2 np = v_2.v ++ np ++ v_2.p$$
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 ++ "нятие" ++ np.s} ;
    John = {s = "וול" ; g = Masc} ;
    Mary = {s = "מר" ; g = Fem} ;
    Love = table {Masc => "אהבה" ; Fem => "אהבה"} ;
    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

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

\[
\text{lin} \quad \text{Mary} = \{s = "mry" ; g = \text{Fem}\}
\]

VP has gender as its **variable feature** - an argument of a table

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

In predication, the VP receives the gender of the NP

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

- *cheval* pl. *chevaux* masc. noun

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

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

> p -lang=ZeroEng "Mary loves John" | align_words -view=open
Building applications

Compile the grammar to PGF:

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

The resulting file Zero.pgf can be e.g. included in fridge magnets:
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 (German): *wenn Johann Maria lieb hat, hat Maria Johann lieb*

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
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_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.
Lecture 2

Morphological Paradigms

and Lexica
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
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
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

resource MorphoEng = {oper regV : Str -> V ; ...}

Lexicon: abstract and concrete syntax

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

congcrete 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).
Example: resource module for English verb inflection

Use the library module Prelude.

Start by defining parameter types and parts of speech.

```plaintext
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

```haskell
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

```plaintext
> i -retain Morpho.gf

Use command `cc = compute_concrete`

```plaintext
> 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

\[ \text{Str} \rightarrow \text{Verb} \]

which takes a string and returns an inflection table.

Let’s first define the paradigm for regular verbs:

\[ \text{regVerb} : \text{Str} \rightarrow \text{Verb} = \backslash \text{walk} \rightarrow \]
\[ \text{mkVerb } \text{walk} \ (\text{walk} + \ "s") \ (\text{walk} + \ "ed") \ (\text{walk} + \ "ed") \ (\text{walk} + \ "ing") ; \]

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

\[
\text{\texttt{s\_regVerb : Str -> Verb = \( \text{kiss} \rightarrow \text{mkVerb kiss (kiss + "es") (kiss + "ed") (kiss + "ed") (kiss + "ing") ;} \)}
\]

For verbs ending with e

\[
\text{\texttt{e\_regVerb : Str -> Verb = \( \text{\texttt{use}} \rightarrow \)}
\]
\[
\begin{align*}
\text{let us = init use} \\
\text{in \ mkVerb use (use + "s") (us + "ed") (us + "ed") (us + "ing") ;}
\end{align*}
\]

Notice:

- the \textbf{local definition} \texttt{let c = d in ...}
- the operation \texttt{init} from Prelude, dropping the last character
More paradigms still

For verbs ending with *y*

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

For verbs ending with *ie*

\[ ie\_regVerb : \text{Str} \to \text{Verb} = \lambda die \to \]
\[ \quad \text{let } dy = \text{Predef.tk } 2 \ die + \ "y" \]
\[ \quad \text{in} \]
\[ \quad \text{mkVerb } die \ (die + \ "s") \ (die + \ "d") \ (die + \ "d") \ (dy + \ "ing") \] ;
What paradigm to choose

If the infinitive ends with $s$, $x$, $z$, $ch$, choose $s\_\text{regVerb}$: munch, munches

If the infinitive ends with $y$, choose $y\_\text{regVerb}$: cry, cries, cried

- except if a vowel comes before: play, plays, played

If the infinitive ends with $e$, choose $e\_\text{regVerb}$: use, used, using

- except if an $i$ precedes: die, dying

- or if an $e$ precedes: free, freeing
A smart paradigm

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

> cc -all smartVerb "munch"
munch munches munched 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 classified classifying
The smart paradigm is not 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

\[\text{dupRegVerb : Str} \rightarrow \text{Verb} = \lambda \text{stop} \rightarrow\]
\[
\begin{align*}
\text{let stopp} &= \text{stop} + \text{last stop} \\
\text{in} \\
\text{mkVerb stop (stop + "s") (stopp + "ed") (stopp + "ed") (stopp + "ing")}
\end{align*}
\]

String pattern: relevant consonant preceded by a vowel

\[\_ + ("a"|"e"|"i"|"o"|"u") + ("b"|"d"|"g"|"m"|"n"|"p"|"r"|"s"|"t") \rightarrow \text{dupRegVerb v ;}\]
Testing consonant duplication

Now it works

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

But what about

> cc -all smartVerb "coat"
  coat 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.

```haskell
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.

```
mkV = overload {
    mkV : (cry : Str) -> Verb = smartVerb ;
    mkV : (sing,sang,sung : Str) -> Verb = irregVerb ;
    mkV : (go,goes,went,gone,going : Str) -> Verb = mkVerb ;
} ;
```
Testing the overloaded paradigm

> cc -all mkV "lie"
lie lies lied lied lying
> cc -all mkV "lie" "lay" "lain"
lie lies lay lain lying
> cc -all mkV "omit"
omit omits omitted omitted omitting
> cc -all mkV "vomit"
vomit vomits vomitted vomitted vomitting
> cc -all mkV "vomit" "vomited" "vomited"
vomit vomits vomited vomited vomitting
> cc -all mkV "vomit" "vomits" "vomited" "vomited" "vomiting"
vomit vomits vomited vomited vomitting

Surely we could do better for vomit...
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.

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

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

Notice: pattern variable \text{cr} matches like _ but gets bound.
Building a lexicon

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

abstract Lex = {
    cat V;
    fun
        play_V : V;
        sleep_V : V;
}

concrete LexEng = open Morpho in {
    lincat V = Verb;
    lin
        play_V = mkV "play";
        sleep_V = mkV "sleep" "slept" "slept";
}

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

V play
V sleep slept slept
Bootstrapping a lexicon

Alt 1. From a morphological POS-tagged word list: trivial

V play played played
V sleep slept slept

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

V play ===> V play played played ===>
V sleep V sleep sleeped sleeped V sleep slept slept

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

Semitic languages, e.g. Arabic: *kataba* has forms *kaAtib*, *yaktubu*, ...

Traditional analysis:

- **word** = **root** + **pattern**

- **root** = three consonants (radicals)

- **pattern** = function from root to string (notation: string with variables $F, C, L$ for the radicals)

Example: *yaktubu* = *ktb* + *yaFCuLu*

Words are datastructures rather than strings!
Datastructures for Arabic

Roots are records of strings.

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

Patterns are functions from roots to strings.

\[ \text{Pattern} : \text{Type} = \text{Root} \to \text{Str} ; \]

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

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

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:

\[
\text{fill} : \text{Filling} \rightarrow \text{Pattern} = \lambda p, r \rightarrow \\
\]

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

\[
\text{dfill} : \text{Filling} \rightarrow \text{Pattern} = \lambda p, r \rightarrow \\
\]
Encoding roots by strings

This is just for the ease of programming and writing lexica.

F = first letter, C = second letter, L = the rest.

得到根: Str -> Root = \s -> case s of {
    F@? + C@? + L => {F = F ; C = C ; L = L} ;
    _ => Predef.error ("cannot get root from" ++ s)
} ;

The as-pattern x@p matches p and binds x.

The error function 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.

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

Dictionary entry: root + pattern.

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

Now we can try:

```
> cc getWord "ktb" "yaFCuLu"
"yaktubu"
> cc getWord "ktb" "muFaCCiLu"
"mukattibu"
```
Parameters for the Arabic verb type

Inflection in tense, number, person, gender.

```plaintext
param
    Number = Sg | D1 | 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.</td>
<td>sing.</td>
<td>َُْـَ</td>
<td>َُْـَ</td>
</tr>
<tr>
<td>3. fem.</td>
<td>sing.</td>
<td>َُْـَ</td>
<td>َُْـَ</td>
</tr>
<tr>
<td>2. masc.</td>
<td>sing.</td>
<td>َُْـَ</td>
<td>َُْـَ</td>
</tr>
<tr>
<td>2. fem.</td>
<td>sing.</td>
<td>َُْـَ</td>
<td>َُْـَ</td>
</tr>
<tr>
<td>1.</td>
<td>sing.</td>
<td>َُْـَ</td>
<td>َُْـَ</td>
</tr>
<tr>
<td>3. masc.</td>
<td>dual.</td>
<td>ِْـََّ</td>
<td>ِْـََّ</td>
</tr>
<tr>
<td>3. fem.</td>
<td>dual.</td>
<td>ِْـََّ</td>
<td>ِْـََّ</td>
</tr>
<tr>
<td>2.</td>
<td>dual.</td>
<td>ِْـََّ</td>
<td>ِْـََّ</td>
</tr>
<tr>
<td>3. masc.</td>
<td>plur.</td>
<td>َِْـََّ</td>
<td>َِْـََّ</td>
</tr>
<tr>
<td>3. fem.</td>
<td>plur.</td>
<td>َِْـََّ</td>
<td>َِْـََّ</td>
</tr>
<tr>
<td>2. masc.</td>
<td>plur.</td>
<td>َِْـََّ</td>
<td>َِْـََّ</td>
</tr>
<tr>
<td>2. fem.</td>
<td>plur.</td>
<td>َِْـََّ</td>
<td>َِْـََّ</td>
</tr>
<tr>
<td>1.</td>
<td>plur.</td>
<td>َِْـََّ</td>
<td>َِْـََّ</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
| Vp2Dl
| Vp2P1 Gender
| Vp1Sg
| Vp1P1 ;

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

Thus $2 \times (3 \times 2 + 2 + 1 + 2 + 1 + 1) = 26$ forms, not $2 \times 3 \times 2 \times 3 = 36$. 
An Arabic verb paradigm

```haskell
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 katabuwaA katabona katabota kataboti katabotum katabotuny2a katabotu katabonaA yakotubu takotubu yakotubaAni takotubaAni yakotubuwna yakotubna takotubu takotubiyna takotubaAni takotubu 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
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.
Lecture 3
Building up a linguistic syntax
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!
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!
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
Predication: 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:

$$\forall p,q \Rightarrow t \quad == \quad \text{table } \{p \Rightarrow \text{table } \{q \Rightarrow t\}\}$$

It is similar to lambda abstraction ($\lambda x,y \Rightarrow t$ 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:

- agreement to object rather than subject (Hindi)
Variable subject case:

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

lincat NP = {s : Case => Str ; a : Agr }
lincat VP = {s : Tense => Agr => Str }
lincat V2 = {s : Tense => Agr => Str ; c : Case}

fun ComplV2 : V2 -> NP -> VP

lin ComplV2 v2 vp = {s = \t,a => v2.s ! t ! a ++ np.s ! v2.c}
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

<table>
<thead>
<tr>
<th>v2.case</th>
<th>VP, infinitive</th>
<th>translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accative</td>
<td>tavata minut</td>
<td>”meet me”</td>
</tr>
<tr>
<td>Partitive</td>
<td>rakastaa minua</td>
<td>”love me”</td>
</tr>
<tr>
<td>Elative</td>
<td>pitää minusta</td>
<td>”like me”</td>
</tr>
<tr>
<td>Genitive + perään</td>
<td>katsoa minun perääni</td>
<td>”look after me”</td>
</tr>
</tbody>
</table>
Complementation: variations

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

\[
\text{lincat } V_2 = \{s : \text{Tense} \rightarrow \text{Agr} \rightarrow \text{Str} ; c : \text{Case} ; \text{prep} : \text{Str}\}
\]

\[
\text{lin ComplV2 } v_2 \text{ vp } = \{s = \emptyset, t, a \rightarrow v_2.s \oplus t \oplus a \oplus v_2.prep \oplus np.s \oplus v_2.c\}
\]

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")
Interplay between features: determination

lincat NP = {s : Case => Str ; a : Agr }
lincat CN = {s : Number => Case => Str ; g : Gender}
lincat Det = {s : Gender => Case => Str ; n : Number}

fun DetCN : Det -> CN -> NP

lin DetCN det cn = {
  s = \c => det.s ! cn.g ! c ++ cn.s ! det.n ! c ;
  a = agr cn.g det.n Per3
}

oper agr : Gender -> Number -> Person -> Agr
Determination: examples

English

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

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

<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><em>questo vino</em></td>
</tr>
<tr>
<td>Sg</td>
<td>Fem</td>
<td><em>questa pizza</em></td>
</tr>
<tr>
<td>Pl</td>
<td>Fem</td>
<td><em>quelle pizze</em></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Det.num</th>
<th>NP, nominative</th>
<th>NP, inessive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sg</td>
<td><em>jokainen talo</em></td>
<td><em>jokaisessa talossa</em></td>
</tr>
<tr>
<td>Pl</td>
<td><em>nämä talot</em></td>
<td><em>näissä taloissa</em></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")
Interplay between features: modification

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

\[ \text{fun AdjCN} : \text{AP} \rightarrow \text{CN} \rightarrow \text{CN} \]

\[ \text{lin AdjCN ap cn} = \{ \]
\[ s = \text{ap.s} n,c \Rightarrow \text{cn.g} ! n \! c \text{++ cn.s} ! n \! c ; \]
\[ g = \text{cn.g} \]
\[ \} \]
Modification: examples

English

<table>
<thead>
<tr>
<th>CN, singular</th>
<th>CN, plural</th>
</tr>
</thead>
<tbody>
<tr>
<td>new house</td>
<td>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>vino rosso</td>
<td>vini rossi</td>
</tr>
<tr>
<td>Fem</td>
<td>casa rossa</td>
<td>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>punainen talo</td>
<td>punaiselta talolta</td>
<td>punaisina taloina</td>
</tr>
</tbody>
</table>
Modification: variations

The place of the adjectival phrase

- Italian: *casa rossa, vecchia casa* ("red house", "old house")

- English: *old house, house similar to this*

Specificity parameter of the adjective

- German: *ein rotes Haus vs. das rote Haus* ("a red house" vs. "the red house")
Lexical insertion

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

There are **lexical insertion functions** for each lexical category:

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

The linearization rules are often trivial, because the lincats match

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

However, for \text{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:

```plaintext
lin DetCN det cn = {
    s = \ \ c => det.s ! cn.g ! c ++ cn.s ! det.n ! c ;
    a = agr cn.g det.n Per3
}
```

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 seldom) introduces to them.

Linearization types are often specific and inflection are irregular.
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 Test = Grammar **... -- test lexicon built on Grammar

resource ResIta -- resource for Italian

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

concrete TestIta of Test = GrammarIta ** open ResIta in... -- It. lexicon
Extension vs. opening

Module extension: \( N = M_1, M_2, M_3 \) ** \{ \ldots \}**

- module \( N \) inherits all judgements from \( M_1, M_2, M_3 \)

Module opening: \( N = \text{open } R_1, R_2, R_3 \) in \{ \ldots \}**

- module \( N \) can use all judgements from \( R_1, R_2, R_3 \) (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`

```bash
> i -retain TestIta.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

abstract Grammar = {
    cat
        C1 ; NP ; VP ; AP ; CN ; Det ; N ; A ; V ; V2 ;
    fun
        PredVP : NP -> VP -> C1 ;
        ComplV2 : V2 -> NP -> VP ;
        DetCN : Det -> CN -> NP ;
        ModCN : CN -> AP -> CN ;

        UseV : V -> VP ;
        UseN : N -> CN ;
        UseA : A -> AP ;

        a_Det, the_Det : Det ; this_Det, these_Det : Det ;
        i_NP, she_NP, we_NP : NP ;
}
Parameters

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

    Number = Sg | Pl ;
    Gender = Masc | Fem ;
    Case = Nom | Acc | Dat ;
    Aux = Avere | Essere ; -- the auxiliary verb of a verb
    Tense = Pres | Perf ;
    Person = Per1 | Per2 | Per3 ;

    Agr = Ag Gender Number Person ;

    VForm = VInf | VPres Number Person | VPart Gender Number ;
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 arrivati</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 arrivata</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 arrivata</td>
</tr>
</tbody>
</table>
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>arrivare</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>arrivate</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,

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

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

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

We choose the latter. It is more efficient in parsing.
Verb phrase forming

Lexical insertion is trivial.

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

Complementation assumes NP has a clitic and an ordinary object part.

\[
\text{lin ComplV2} = \\
\text{let} \\
\text{nps} = \text{np.s} ! \text{v2.c} \\
\text{in} \\
\text{v} = \{s = \text{v2.s} ; \text{aux} = \text{v2.aux}\} ; \\
\text{clit} = \text{nps.clit} ; \\
\text{obj} = \text{nps.obj} \\
\]
Noun phrases

Being clitic depends on case

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

Examples:

\[
\text{lin she\_NP} = \{
  s = \text{table} \{
    \text{Nom} \Rightarrow \{\text{clit} = [] ; \text{obj} = "lei"\} ;
    \text{Acc} \Rightarrow \{\text{clit} = "la" ; \text{obj} = []\} ;
    \text{Dat} \Rightarrow \{\text{clit} = "le" ; \text{obj} = []\}
  \} ;
  a = \text{Ag Fem Sg Per3}
\}
\]

\[
\text{lin John\_NP} = \{
  s = \text{table} \{
    \text{Nom} | \text{Acc} \Rightarrow \{\text{clit} = [] ; \text{obj} = "Giovanni"\} ;
    \text{Dat} \Rightarrow \{\text{clit} = [] ; \text{obj} = "a Giovanni"\}
  \} ;
  a = \text{Ag Fem Sg Per3}
\}
\]
Noun phrases: alternatively

Use a feature instead of separate fields,

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

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

No surprises

```plaintext
lincat Det = {s : Gender => Case => Str ; n : Number} ;

lin DetCN det cn = {
    s = \c => {obj = det.s ! cn.g ! c ++ cn.s ! det.n ; clit = []} ;
    a = Ag cn.g det.n Per3
}
```
Building determiners

Often from adjectives:

\[
\begin{align*}
\text{lin this}_{\text{Det}} & = \text{adjDet} (\text{mkA "questo"}) \ \text{Sg} ; \\
\text{lin these}_{\text{Det}} & = \text{adjDet} (\text{mkA "questo"}) \ \text{Pl} ; \\
\end{align*}
\]

\text{oper prepCase : Case} \rightarrow \text{Str} = \lambda c \rightarrow \text{case c of} \{ \\
\text{Dat} \rightarrow "a" ; \\
_ \rightarrow [] \\
\}_ ;

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

Articles: see GrammarIta.gf
Adjectival modification

Recall the inherent feature for position

lincatid AP = {s : Gender => Number => Str ; isPre : Bool} ;

lin ModCN cn ap = {
    s = \n => preOrPost ap.isPre (ap.s ! cn.g ! n) (cn.s ! n) ;
    g = cn.g
} ;

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

Complex but mostly great fun:

```
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.
Predication, at last

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

```plaintext
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

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

oper agrPart : Verb -> Agr -> Str = \v,a -> case v.aux of {
   Avere => v.s ! VPart Masc Sg ;
   Essere => case a of {
      Ag g n _ => v.s ! VPart g n
   }
};
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!
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

| VPGenPres True | लड़की सेव खाती है |
| VPGenPres False | लड़की सेव नहीं खाती है |
| VPImpPast True | लड़की सेव खाती थी |
| VPImpPast False | लड़की सेव नहीं खाती थी |
| VPContPres True | लड़की सेव खा रही है |
| VPContPres False | लड़की सेव नहीं खा रही है |
| VPContPast True | लड़की सेव खा रही थी |
| VPContPast False | लड़की सेव नहीं खा रही थी |
| VPPers True | लड़की ने सेव खाया |
| VPPers False | लड़की ने सेव नहीं खाया |
| VPPersPres True | लड़की सेव खायी है |
| VPPersPres False | लड़की सेव नहीं खायी है |
| VPPersPast True | लड़की सेव खायी थी |
| VPPersPast False | लड़की सेव नहीं खायी थी |
| VPSubj True | लड़की सेव खाये |
| VPSubj False | लड़की सेव न खाये |
| VPFut True | लड़की सेव खायेगी |
| VPFut False | लड़की सेव न खायेगी |
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 Test 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).

Lecture 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
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, ...

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

\[ \text{PredVP} : \text{NP} \rightarrow \text{VP} \rightarrow \text{Cl} \]
\[ \text{Comp1V2} : \text{V2} \rightarrow \text{NP} \rightarrow \text{VP} \]
\[ \text{NumCN} : \text{Num} \rightarrow \text{CN} \rightarrow \text{NP} \]
Using the library: natural language output

Task: in an email program, generate phrases saying \textit{you have n message(s)}

Problem: avoid \textit{you have one messages}

Solution: use the library

\[
\text{PredVP youSg_NP (ComplV2 have_V2 (NumCN two_Num (UseN (mkN "message"))))} \\
\implies \text{you have two messages}
\]

\[
\text{PredVP youSg_NP (ComplV2 have_V2 (NumCN one_Num (UseN (mkN "message"))))} \\
\implies \text{you have one message}
\]
Software localization

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

```
PredVP youSg_NP (ComplV2 have_V2 (NumCN two_Num (UseN (mkN "messaggio")))))
===> hai due messaggi
```

```
PredVP youSg_NP (ComplV2 have_V2 (NumCN two_Num (UseN (mkN "meddelande")))))
===> du har två meddelanden
```

```
PredVP youSg_NP (ComplV2 have_V2 (NumCN two_Num (UseN (mkN "viesti")))))
===> sinulla on kaksi viestiä
```

The new languages are more complex than English - but only internally, not on the API level!
Correct number in Arabic

1 message  رسالة
2 messages رسالةتان
(3–10) messages رسائل
(11–99) messages رسائلتان
x100 messages رسالات

(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
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.
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

\[
\text{lincat Person} = \text{NP} \; ; \; \text{Fact} = \text{Cl} \; ; \\
\text{lin Like} \; x \; y = \text{PredVP} \; x \; (\text{ComplV2} \; \text{like}_V2 \; y) \; \quad \text{-- English} \\
\text{lin Like} \; x \; y = \text{PredVP} \; y \; (\text{ComplV2} \; \text{piacere}_V2 \; x) \; \quad \text{-- Italian}
\]

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!
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 \texttt{mkC} where \texttt{C} is the value category.

The resource defines e.g.

\texttt{mkC1 : NP -> V2 -> NP -> C1 = \textbackslash subj,verb,obj ->}
\texttt{PredVP subj (ComplV2 verb obj)}

\texttt{mkC1 : NP -> V -> C1 = \textbackslash subj,verb ->}
\texttt{PredVP subj (UseV verb)}
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>mkCl : NP -&gt; V2 -&gt; NP -&gt; Cl</td>
<td>John loves Mary</td>
</tr>
<tr>
<td>mkNP : Numeral -&gt; CN -&gt; NP</td>
<td>five cars</td>
</tr>
<tr>
<td>mkNP : Quant -&gt; CN -&gt; NP</td>
<td>that car</td>
</tr>
<tr>
<td>mkNP : Pron -&gt; NP</td>
<td>we</td>
</tr>
<tr>
<td>mkCN : N -&gt; CN</td>
<td>car</td>
</tr>
<tr>
<td>this_Quant : Quant</td>
<td>this, these</td>
</tr>
<tr>
<td>youSg_Pron : Pron</td>
<td>you (singular)</td>
</tr>
<tr>
<td>n1_Numeral, n2_Numeral : Numeral</td>
<td>one, two</td>
</tr>
<tr>
<td>n100_Numeral : Numeral</td>
<td>one hundred</td>
</tr>
<tr>
<td>have_V2 : V2</td>
<td>have</td>
</tr>
</tbody>
</table>
Concrete syntax for English

How are messages expressed by using the library?

concrete FaceEng of Face = open SyntaxEng, ParadigmsEng 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" ;
}
Concrete syntax for Finnish

Exactly the same rules of combination, only different words:

concrete FaceFin 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 "pitää" elative ;
    invitation_N = mkN "kutsu" ;
    friend_N = mkN "ystävä" ;
}
Parametrized modules

Can we avoid repetition of the lincat and lin code? Yes!

New module type: **functor**, a.k.a. **incomplete** or **parametrized** module

```
incomplete concrete FaceI of Face = open Syntax, LexFace in ...```

A functor may open **interfaces**.

An interface has **oper** declarations with just a type, no definition.

Here, Syntax and LexFace are interfaces.
The domain lexicon interface

Syntax is the Resource Grammar interface, and gives

- combination rules

- structural words

Content words are not given in Syntax, but in a **domain lexicon**

interface LexFace = open Syntax in {

oper
    like_V2 : V2 ;
    invitation_N : N ;
    friend_N : N ;
}
Concrete syntax functor "FaceI"

incomplete concrete FaceI of Face = open Syntax, LexFace 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 ;
}
An English instance of the domain lexicon

Define the domain words in English

instance LexFaceEng of LexFace = open SyntaxEng, ParadigmsEng in {

oper
    like_V2 = mkV2 "like" ;
    invitation_N = mkN "invitation" ;
    friend_N = mkN "friend" ;
}

Put everything together: functor instantiation

Instantiate the functor FaceI by giving instances to its interfaces

```bash
--# -path=.:present

concrete 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

```haskell
instance LexFaceFin of LexFace = open SyntaxFin, ParadigmsFin in {
  oper
    like_V2 = mkV2 (mkV "pitää") elative ;
    invitation_N = mkN "kutsu" ;
    friend_N = mkN "ystävä" ;
}
```

2. Functor instantiation: mechanically change Eng to Fin

```haskell
--# -path=.:present

concrete FaceFin of Face = FaceI with
  (Syntax = SyntaxFin),
  (LexFace = LexFaceFin) ;
```
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 $\text{LexFace}_L$

5. For each language $L$: concrete syntax instantiation $\text{Face}_L$
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

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

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 ;
}
Free variation

There can be many ways of expressing a given semantic structure.

This can be expressed by the **variant** operator `|`.

```haskell
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.
Overview of the resource grammar API

For the full story, see the resource grammar synopsis in digitalgrammars.com/gf/lib/resource/doc/synopsis.html

Main division:

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

The Synopsis for next-lib has some more functions.
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>Cl</td>
<td>clause (variable tense)</td>
<td><em>John walks</em></td>
</tr>
<tr>
<td>QCl</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>smart</td>
</tr>
<tr>
<td>A2</td>
<td>two-place adjective</td>
<td>NP</td>
<td>married (to her)</td>
</tr>
<tr>
<td>Adv</td>
<td>adverb</td>
<td>-</td>
<td>here</td>
</tr>
<tr>
<td>N</td>
<td>common noun</td>
<td>-</td>
<td>man</td>
</tr>
<tr>
<td>N2</td>
<td>relational noun</td>
<td>NP</td>
<td>friend (of John)</td>
</tr>
<tr>
<td>NP</td>
<td>noun phrase</td>
<td>-</td>
<td>the boss</td>
</tr>
<tr>
<td>V</td>
<td>one-place verb</td>
<td>-</td>
<td>sleep</td>
</tr>
<tr>
<td>V2</td>
<td>two-place verb</td>
<td>NP</td>
<td>love (her)</td>
</tr>
<tr>
<td>V3</td>
<td>three-place verb</td>
<td>NP, NP</td>
<td>show (it to me)</td>
</tr>
<tr>
<td>VS</td>
<td>sentence-complement verb</td>
<td>S</td>
<td>say (that I run)</td>
</tr>
<tr>
<td>VV</td>
<td>verb-complement verb</td>
<td>VP</td>
<td>want (to run)</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 $\rightarrow$ V $\rightarrow$ Cl</td>
<td>John walks</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ V2 $\rightarrow$ NP $\rightarrow$ Cl</td>
<td>John loves her</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ V3 $\rightarrow$ NP $\rightarrow$ NP $\rightarrow$ Cl</td>
<td>John sends it to her</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ VV $\rightarrow$ VP $\rightarrow$ Cl</td>
<td>John wants to walk</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ VS $\rightarrow$ S $\rightarrow$ Cl</td>
<td>John says that it is good</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ A $\rightarrow$ Cl</td>
<td>John is old</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ A $\rightarrow$ NP $\rightarrow$ Cl</td>
<td>John is older than Mary</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ A2 $\rightarrow$ NP $\rightarrow$ Cl</td>
<td>John is married to her</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ AP $\rightarrow$ Cl</td>
<td>John is very old</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ N $\rightarrow$ Cl</td>
<td>John is a man</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ CN $\rightarrow$ Cl</td>
<td>John is an old man</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ NP $\rightarrow$ Cl</td>
<td>John is the man</td>
</tr>
<tr>
<td>mkCl</td>
<td>NP $\rightarrow$ Adv $\rightarrow$ 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><em>this man</em></td>
</tr>
<tr>
<td>mkNP</td>
<td>Numeral -&gt; CN -&gt; NP</td>
<td><em>five men</em></td>
</tr>
<tr>
<td>mkNP</td>
<td>PN -&gt; NP</td>
<td><em>John</em></td>
</tr>
<tr>
<td>mkNP</td>
<td>Pron -&gt; NP</td>
<td><em>we</em></td>
</tr>
<tr>
<td>mkNP</td>
<td>Quant -&gt; Num -&gt; CN -&gt; NP</td>
<td><em>these (five) man</em></td>
</tr>
<tr>
<td>mkCN</td>
<td>N -&gt; CN</td>
<td><em>man</em></td>
</tr>
<tr>
<td>mkCN</td>
<td>A -&gt; N -&gt; CN</td>
<td><em>old man</em></td>
</tr>
<tr>
<td>mkCN</td>
<td>AP -&gt; CN -&gt; CN</td>
<td><em>very old Chinese man</em></td>
</tr>
<tr>
<td>mkNum</td>
<td>Numeral -&gt; Num</td>
<td><em>five</em></td>
</tr>
<tr>
<td>n100_Numeral</td>
<td>Numeral</td>
<td><em>one hundred</em></td>
</tr>
<tr>
<td>plNum</td>
<td>Num</td>
<td><em>(plural)</em></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>C1 -&gt; QC1</td>
<td><em>does John walk</em></td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; V -&gt; QC1</td>
<td><em>who walks</em></td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; V2 -&gt; NP -&gt; QC1</td>
<td><em>who loves her</em></td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; NP -&gt; V2 -&gt; QC1</td>
<td><em>whom does she love</em></td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; AP -&gt; QC1</td>
<td><em>who is old</em></td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; NP -&gt; QC1</td>
<td><em>who is the boss</em></td>
</tr>
<tr>
<td>mkQCl</td>
<td>IP -&gt; Adv -&gt; QC1</td>
<td><em>who is here</em></td>
</tr>
<tr>
<td>mkQCl</td>
<td>IAdv -&gt; C1 -&gt; QC1</td>
<td><em>where does John walk</em></td>
</tr>
<tr>
<td>mkIP</td>
<td>CN -&gt; IP</td>
<td><em>which car</em></td>
</tr>
<tr>
<td>who_IP</td>
<td>IP</td>
<td><em>who</em></td>
</tr>
<tr>
<td>why_IAdv</td>
<td>IAdv</td>
<td><em>why</em></td>
</tr>
<tr>
<td>where_IAdv</td>
<td>IAdv</td>
<td><em>where</em></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) [default]</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>Cl -&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
Exercises

1. Compile and make available the resource grammar library, latest version. Compilation is by `make` in GF/next-lib/src. Make it available by setting `GF_LIB_PATH` to GF/next-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.
Lecture 5

Inside the Resource Grammar Library
Contents

Module structure

Statistics

How to start building a new language

How to test a resource grammar

The Assignment
The principal module structure

solid = API, dashed = internal, ellipse = abstract+concrete, rectangle = resource-instance, diamond = interface, green = given, blue = mechanical, red = to do
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, SSSlash</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 \texttt{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:

\begin{align*}
\text{Sentence.UseCl} & : \text{Temp} \to \text{Pol} \to \text{Cl} \to S \quad \text{-- } S \text{ uses } \text{Cl} \\
\text{Sentence.PredVP} & : \text{VP} \to \text{NP} \to \text{Cl} \quad \text{-- } \text{uses } \text{VP} \\
\text{Verb.ComplVS} & : \text{VS} \to S \to \text{VP} \quad \text{-- } \text{uses } S
\end{align*}
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 `op` in the same module
- Repetition in many modules: define an `op` 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 -> VP Agr ;

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 \textit{la somme de 3 et de 4}, Ita \textit{la somma di 3 e 4}).

oper conjunctCase : NPForm -> NPForm ;

How infinitives and clitics are placed relative to each other (Fre \textit{la voir}, Ita \textit{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 => AAggr => 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
## 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 v. 1.4, rough estimates of person months. * = generated code.
Languages under construction

<table>
<thead>
<tr>
<th>language</th>
<th>started</th>
<th>% morpho</th>
<th>% lexicon</th>
<th>% syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arabic</td>
<td>2005</td>
<td>90</td>
<td>95</td>
<td>20</td>
</tr>
<tr>
<td>Hindi</td>
<td>2008</td>
<td>70</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Latin</td>
<td>2008</td>
<td>80</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Polish</td>
<td>2007</td>
<td>95</td>
<td>60</td>
<td>2</td>
</tr>
<tr>
<td>Romanian</td>
<td>2009</td>
<td>95</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>Thai</td>
<td>2007</td>
<td>70</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Turkish</td>
<td>2007</td>
<td>30</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

Most of these figures are rough estimates!
How to start building a language, e.g. Marathi

1. Create a directory GF/next-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: 16-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

```haskell
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 with the `put_string` command.
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=trs | l -all -treebank | wf 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/next-lib/src.

Use make in this directory to compile the libraries.

Use runghc Make lang api langs=Mar to compile just the language Mar.
Assignment

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.