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

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
                \                               /
                  Pred Mary (Compl Love John)
                /                               \
Marie aime Jean                                   מרי אוהבת את ג׳ון
```

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 `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.

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)
```

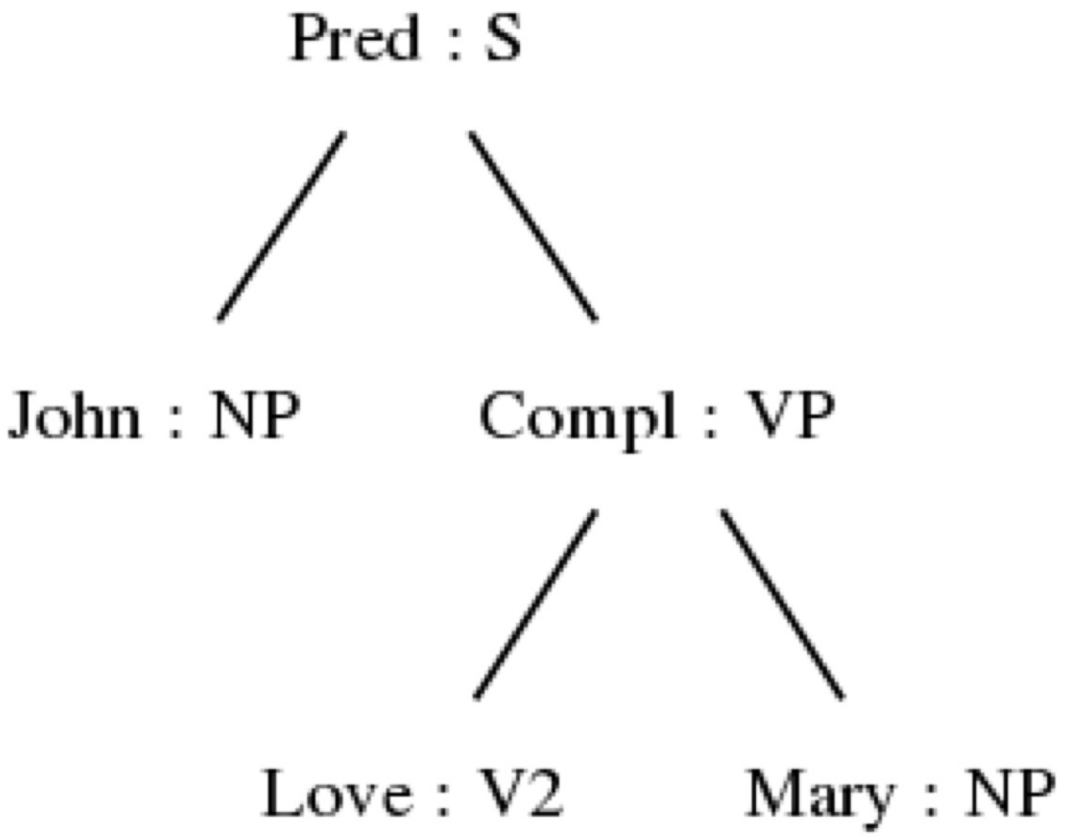
To **linearize** a tree to a string: `linearize = l`

```
> 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



```
> 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 **judgements** in GF:

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

```
Pred. S ::= NP VP  ===>    fun Pred : NP -> VP -> S
                           lin Pred np vp = np ++ vp
```

Functions and concatenation

Function type: `A -> B -> 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`

Judgement	**reading**
`cat` C	C is a category
`fun` $f : T$	f is a function of type T
`lincat` C = L	C has linearization type L
`lin` $f\,xs$ = t	$f\,xs$ has linearization t

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

```
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

```
> 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

```
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`,

```
lincat NP = Case => Str
```

The linearization of `John` is an **inflection table**,

```
lin John = table {Nom => "Ioannes" ; Acc => "Ioannem"}
```

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

```
Pred  np vp = np ! Nom ++ vp
Compl v2 np = np ! Acc ++ v2
```

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 `V2` is a **record type** with two **fields**

```
lincat V2 = {v : Str ; p : Str}
```

The linearization of `Love` is a **record**

```
lin Love = {v = "hat" ; p = "lieb"}
```

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

```
lin Compl v2 np = v2.v ++ 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 ++ "את" ++ np.s} ;
   John = {s = "ג’ון" ; g = Masc} ;
   Mary = {s = "מרי" ; g = Fem} ;
   Love = table {Masc => "אוהב" ; Fem => "אוהבת"} ;
 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

```
  lincat NP = {s : Str ; g : Gender}

  lin  Mary = {s = "mry" ; g = Fem}
```

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

```
  lincat VP = Gender => Str
```

In predication, the VP receives the gender of the NP

```
  lin Pred np vp = np.s ++ vp ! 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.

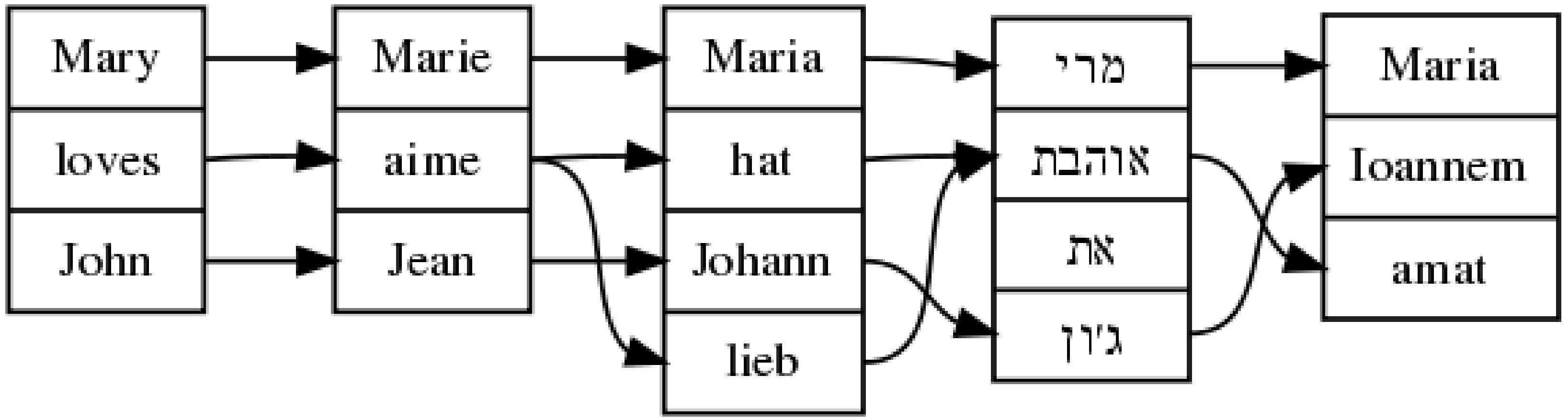
Example: French nouns

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

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

```
  lincat N = {s : Number => Str ; g : Gender}
```

Word alignment



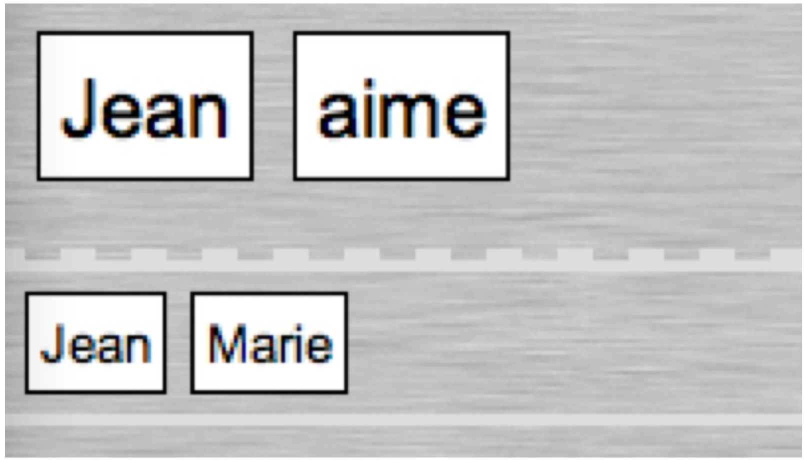
```
> p -lang=ZeroEng "Mary loves John" | align_words -view=open
```

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 $\mathtt{S}$ 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

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 `cat`s.

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

The main lexical categories in the resource grammar

`cat`	**name**	**example**
`N`	noun	*house*
`A`	adjective	*small*
`V`	verb	*sleep*
`V2`	two-place verb	*love*
`Adv`	adverb	*today*

Typical feature design

`cat`	**variable**	**inherent**
`N`	number, case	gender
`A`	number, case, gender, degree	position
`V`	tense, number, person, ...	auxiliary
`V2`	as `V`	complement case
`Adv`	-	-

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 ; ...}

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

The same resource can be used (`open`ed) in many lexica.

Abstract and concrete are **top-level** - they define trees, parsing, linearization.

Resource modules and `oper`s 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.

```
  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*

```
e_regVerb : Str -> Verb = \use ->
  let us = init use
  in  mkVerb use (use + "s") (us + "ed") (us + "ed") (us + "ing") ;
```

Notice:

- the **local definition** `let` *c* `=` *d* `in` ...
- the operation `init` from `Prelude`, dropping the last character

More paradigms still

For verbs ending with *y*

```
y_regVerb : Str -> Verb = \cry ->
  let cr = init cry
  in
  mkVerb cry (cr + "ies") (cr + "ied") (cr + "ied") (cry + "ing") ;
```

For verbs ending with *ie*

```
ie_regVerb : Str -> Verb = \die ->
  let dy = Predef.tk 2 die + "y"
  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 `s_regRerb`: *munch*, *munches*

If the infinitive ends with *y*, choose `y_regRerb`: *cry*, *cries*, *cried*

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

If the infinitive ends with *e*, choose `e_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**

```
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 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`

```
  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"
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.

```
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 vomiting
```

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.

```
add_s : Str -> Str = \v -> case v of {
  _  + ("s"|"z"|"x"|"ch")       => v  + "es" ;
  _  + ("a"|"e"|"o"|"u") + "y" => v  + "s" ;
  cr + "y"                      => cr + "ies" ;
  _                             => v  + "s"
  } ;
```

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

Notice: pattern variable `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 mechnically 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.

```
  Root    : Type = {F,C,L : Str} ;
```

Patterns are functions from roots to strings.

```
  Pattern : Type = Root -> Str ;
```

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

```
  Filling : Type = {F,FC,CL,L : Str} ;
```

This is enough for everything except middle consonant duplication (e.g. *FaCCaLa*).

Applying a pattern

A pattern obtained from a filling intertwines the records:

```
  fill : Filling -> Pattern = \p,r ->
    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:

```
  dfill : Filling -> Pattern = \p,r ->
    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.

```
  getRoot : 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`.

```
getPattern : Str -> Pattern = \s -> case s of {
  F + "F" + FC + "CC" + CL + "L" + L =>
    dfill {F = F ; FC = FC ; CL = CL ; L = L} ;
  F + "F" + FC + "C" + CL + "L" + L =>
    fill {F = F ; FC = FC ; CL = CL ; L = L} ;
  _ => 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

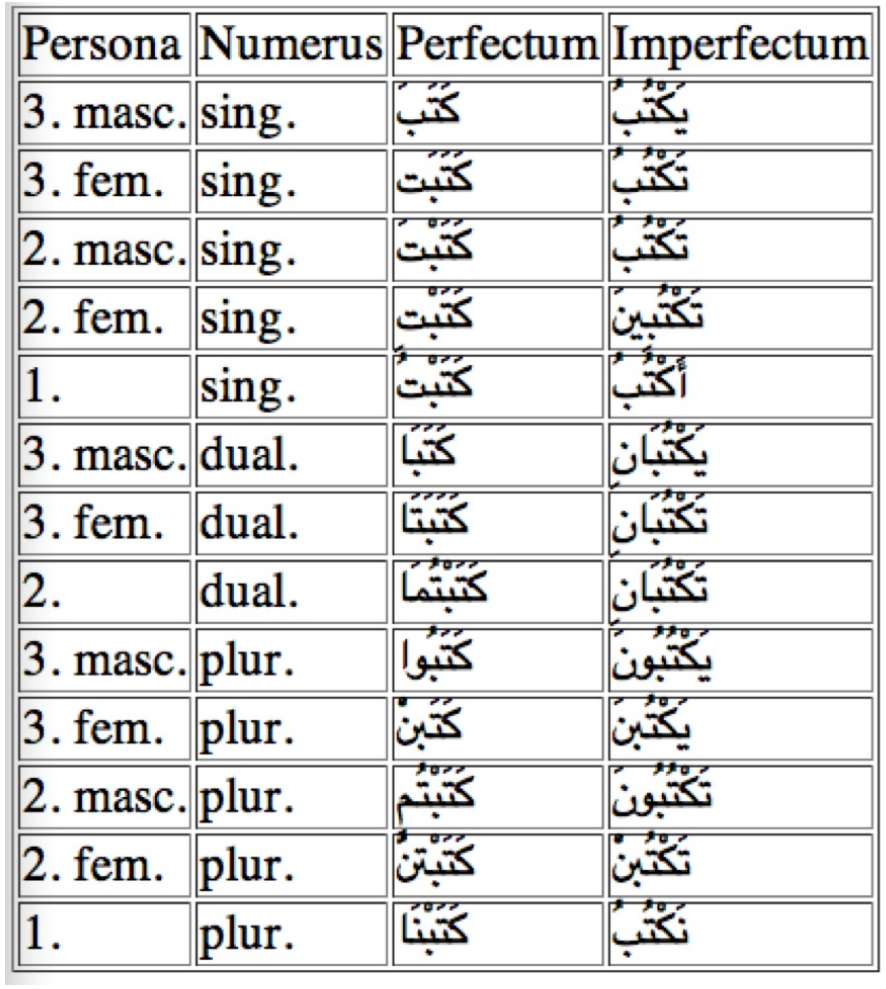
Inflection in tense, number, person, gender.

```
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



Persona	Numerus	Perfectum	Imperfectum
3. masc.	sing.	كَتَبَ	يَكْتُبُ
3. fem.	sing.	كَتَبَتْ	تَكْتُبُ
2. masc.	sing.	كَتَبْتَ	تَكْتُبُ
2. fem.	sing.	كَتَبْتِ	تَكْتُبِينَ
1.	sing.	كَتَبْتُ	أَكْتُبُ
3. masc.	dual.	كَتَبَا	يَكْتُبَانِ
3. fem.	dual.	كَتَبَتَا	تَكْتُبَانِ
2.	dual.	كَتَبْتُمَا	تَكْتُبَانِ
3. masc.	plur.	كَتَبُوا	يَكْتُبُونَ
3. fem.	plur.	كَتَبْنَ	يَكْتُبْنَ
2. masc.	plur.	كَتَبْتُمْ	تَكْتُبُونَ
2. fem.	plur.	كَتَبْتُنَّ	تَكْتُبْنَ
1.	plur.	كَتَبْنَا	نَكْتُبُ

Arabic verb type: implementation

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

```
param VPer =
    Vp3   Number Gender
  | Vp2Sg Gender
  | Vp2Dl
  | Vp2Pl Gender
  | Vp1Sg
  | Vp1Pl ;

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

Thus $2*(3*2 + 2 + 1 + 2 + 1 + 1) = 26$ forms, not $2*3*2*3 = 36$.

An Arabic verb paradigm

```
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 kata
katabotum katabotunv2a katabotu katabonaA yakotubu takotubu yakotubaAni
takotubaAni yakotubuwna yakotubna takotubu takotubiyna takotubaAni takotu
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.htm
  > ! 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

`cat`	**name**	**example**
`Cl`	clause	*every young man loves Mary*
`VP`	verb phrase	*loves Mary*
`V2`	two-place verb	*loves*
`NP`	noun phrase	*every young man*
`CN`	common noun	*young man*
`Det`	determiner	*every*
`AP`	adjectival phrase	*young*

The key functions

`fun`	**name**	**example**
`PredVP : NP -> VP -> Cl`	predication	*every man loves Mary*
`ComplV2 :  V2 -> NP -> VP`	complementation	*loves Mary*
`DetCN : Det -> CN -> NP`	determination	*every man*
`AdjCN : AP -> CN -> CN`	modification	*young man*

Feature design

`cat`	**variable**	**inherent**
`Cl`	tense	-
`VP`	tense, agr	-
`V2`	tense, agr	case
`NP`	case	agr
`CN`	number, case	gender
`Det`	gender, case	number
`AP`	gender, number, case	-

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:

```
  \\p,q => t    ===   table {p => table {q => t}}
```

It is similar to lambda abstraction (`\x,y -> t` in a function type).

Predication: examples

English

np.agr	**present**	**past**	**future**
Sg Per1	*I sleep*	*I slept*	*I will sleep*
Sg Per3	*she sleeps*	*she slept*	*she will sleep*
Pl Per1	*we sleep*	*we slept*	*we will sleep*

Italian (”I am tired”, ”she is tired”, ”we are tired”)

np.agr	**present**	**past**	**future**
Masc Sg Per1	*io sono stanco*	*io ero stanco*	*io sarò stanco*
Fem Sg Per3	*lei è stanca*	*lei era stanca*	*lei sarà stanca*
Fem Pl Per1	*noi siamo stanche*	*noi eravamo stanche*	*noi saremo stanche*

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

v2.case	**infinitive VP**
Acc	*love me*
at + Acc	*look at me*

Finnish

v2.case	**VP, infinitive**	**translation**
Accusative	*tavata minut*	"meet me"
Partitive	*rakastaa minua*	"love me"
Elative	*pitää minusta*	"like me"
Genitive + *perään*	*katsoa minun perääni*	"look after me"

Complementation: variations

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

```
lincat V2 = {s : Tense => Agr => Str ; c : Case ; prep : Str}

lin ComplV2 v2 vp = {s = \\t,a => v2.s ! t ! a ++ v2.prep ++ np.s ! 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")

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

Det.num	**NP**
Sg	*every house*
Pl	*these houses*

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

Det.num	**CN.gen**	**NP**
Sg	Masc	*questo vino*
Sg	Fem	*questa pizza*
Pl	Fem	*quelle pizze*

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

Det.num	**NP, nominative**	**NP, inessive**
Sg	*jokainen talo*	*jokaisessa talossa*
Pl	*nämä talot*	*näissä taloissa*

Determination: variations

Systamatic 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

```
lincat AP  = {s : Gender => Number => Case => Str                }
lincat CN  = {s :           Number => Case => Str ; g : Gender}

fun AdjCN : AP -> CN -> CN

lin AdjCN ap cn = {
  s = \\n,c => ap.s ! cn.g ! n ! c ++ cn.s ! n ! c ;
  g = cn.g
  }
```

Modification: examples

English

CN, singular	**CN, plural**
new house	*new houses*

Italian (”red wine”, ”red house”)

CN.gen	**CN, singular**	**CN, plural**
Masc	*vino rosso*	*vini rossi*
Fem	*casa rossa*	*case rosse*

Finnish (”red house”)

CN, sg, nominative	**CN, sg, ablative**	**CN, pl, essive**
punainen talo	*punaiselta talolta*	*punaisina taloina*

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:

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

The linearization rules are often trivial, because the `lincats` match

```
  lin UseN n = n
  lin UseA a = a
  lin UseV v = v
```

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:

```
  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

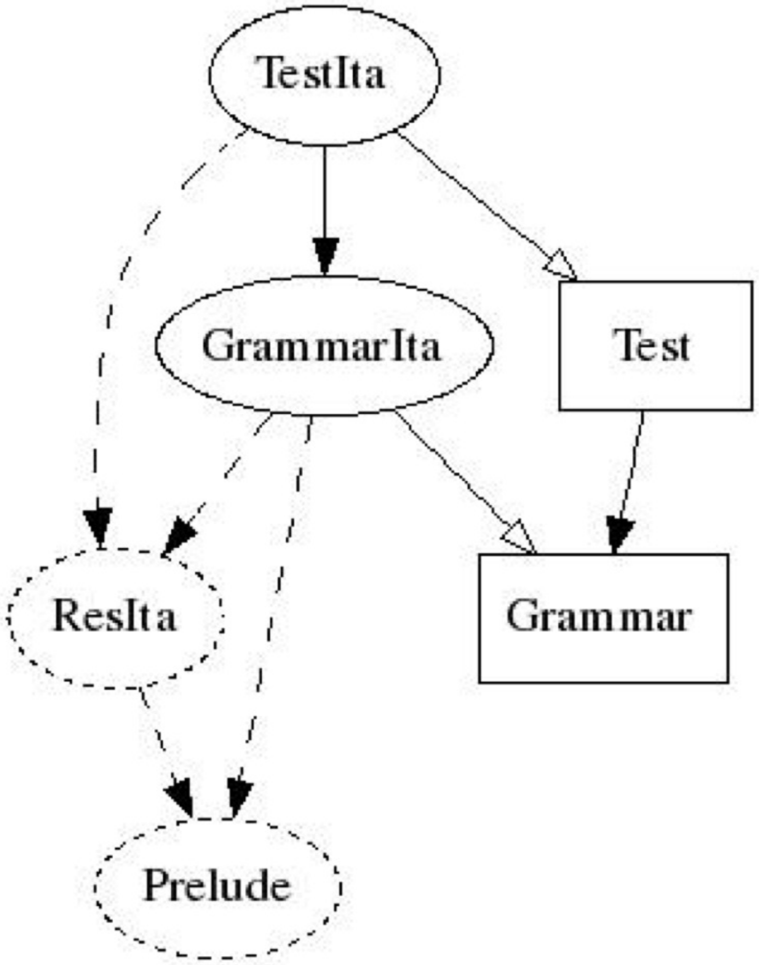
Module extension: `N = M1, M2, M3 ** {...}`

- 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 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
    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 ;

    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

`UseV arrive_V`	**Pres**	**Perf**
Ag Masc Sg Per1	*arrivo*	*sono arrivato*
Ag Fem Sg Per1	*arrivo*	*sono arrivata*
Ag Masc Sg Per2	*arrivi*	*sei arrivato*
Ag Fem Sg Per2	*arrivi*	*sei arrivata*
Ag Masc Sg Per3	*arriva*	*è arrivato*
Ag Fem Sg Per3	*arriva*	*è arrivata*
Ag Masc Pl Per1	*arriviamo*	*siamo arrivati*
Ag Fem Pl Per1	*arriviamo*	*siamo arrivate*
Ag Masc Pl Per2	*arrivate*	*siete arrivati*
Ag Fem Pl Per2	*arrivate*	*siete arrivate*
Ag Masc Pl Per3	*arrivano*	*sono arrivati*
Ag Fem Pl Per3	*arrivano*	*sono arrivate*

The forms of a verb, in morphology

`arrive_V`	**form**
VInf	*arrivare*
VPres Sg Per1	*arrivo*
VPres Sg Per2	*arrivi*
VPres Sg Per3	*arriva*
VPres Pl Per1	*arriviamo*
VPres Pl Per2	*arrivate*
VPres Pl Per3	*arrivano*
VPart Masc Sg	*arrivato*
VPart Fem Sg	*arrivata*
VPart Masc Pl	*arrivati*
VPart Fem Pl	*arrivate*

Inherent feature: `aux` is *essere*.

The verb phrase type

Lexical insertion maps `V` to `VP`.

Two possibilities for `VP`: either close to `Cl`,

```
  lincat VP = {s : Tense => Agr => Str}
```

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

```
  lincat VP = {v : Verb ; clit : Str ; obj : Str} ;
```

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

Verb phrase forming

Lexical insertion is trivial.

```
  lin UseV v = {v = v ; 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
      }
```

Noun phrases

Being clitic depends on case

```
  lincat 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,

```
  lincat NP = {s : Case => {s : Str ; isClit : Bool} ; a : Agr} ;
```

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

Determination

No surprises

```
  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:

```
lin this_Det  = adjDet (mkA "questo") Sg ;
lin these_Det = adjDet (mkA "questo") Pl ;

oper prepCase : Case -> Str = \c -> case c of {
  Dat => "a" ;
  _ => []
  } ;

oper adjDet : Adj -> Number -> Determiner = \adj,n -> {
  s = \\g,c => prepCase c ++ adj.s ! g ! n ;
  n = n
  } ;
```

Articles: see `GrammarIta.gf`

Adjectival modification

Recall the inherent feature for position

```
  lincat 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.

```
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.

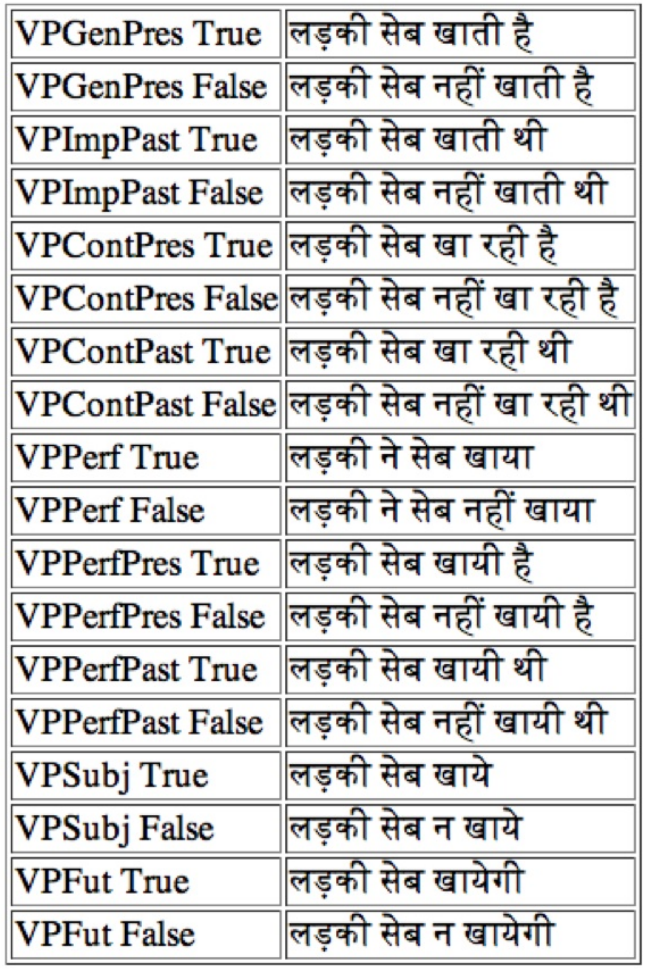
However, in the perfective tense:

- 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"

subj	**obj**	**gen. present**	**perfective**
Masc	Masc	*ladka: seb Ka:ta: hai*	*ladke ne seb Ka:ya:*
Masc	Fem	*ladka: roTi: Ka:ta: hai*	*ladke ne roTi: Ka:yi:*
Fem	Masc	*ladki: seb Ka:ti: hai*	*ladki: ne seb Ka:ya:*
Fem	Fem	*ladki: roTi: Ka:ti: hai*	*ladki: ne roTi: Ka:yi:*

A Hindi clause in different tenses



VPGenPres True	लड़की सेब खाती है
VPGenPres False	लड़की सेब नहीं खाती है
VPImpPast True	लड़की सेब खाती थी
VPImpPast False	लड़की सेब नहीं खाती थी
VPContPres True	लड़की सेब खा रही है
VPContPres False	लड़की सेब नहीं खा रही है
VPContPast True	लड़की सेब खा रही थी
VPContPast False	लड़की सेब नहीं खा रही थी
VPPerf True	लड़की ने सेब खाया
VPPerf False	लड़की ने सेब नहीं खाया
VPPerfPres True	लड़की सेब खायी है
VPPerfPres False	लड़की सेब नहीं खायी है
VPPerfPast True	लड़की सेब खायी थी
VPPerfPast False	लड़की सेब नहीं खायी थी
VPSubj True	लड़की सेब खाये
VPSubj False	लड़की सेब न खाये
VPFut True	लड़की सेब खायेगी
VPFut False	लड़की सेब 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).

5. Learn some linguistics! My favourite book is *Introduction to Theoretical Linguistics* by John Lyons (Cambridge 1968, at least 14 editions).

Lecture 4

Using the Resource Grammar

Library in applications

Contents

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

```
  mkN : (talo : Str) -> N
```

Abstract syntax as API for syntactic combinations

```
  PredVP  : NP -> VP -> Cl
  ComplV2 : V2 -> NP -> VP
  NumCN   : Num -> CN -> NP
```

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

```
  PredVP youSg_NP (ComplV2 have_V2 (NumCN two_Num (UseN (mkN "message"))))
  ===> you have two messages

  PredVP youSg_NP (ComplV2 have_V2 (NumCN one_Num (UseN (mkN "message"))))
  ===> you have one message
```

Software localization

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

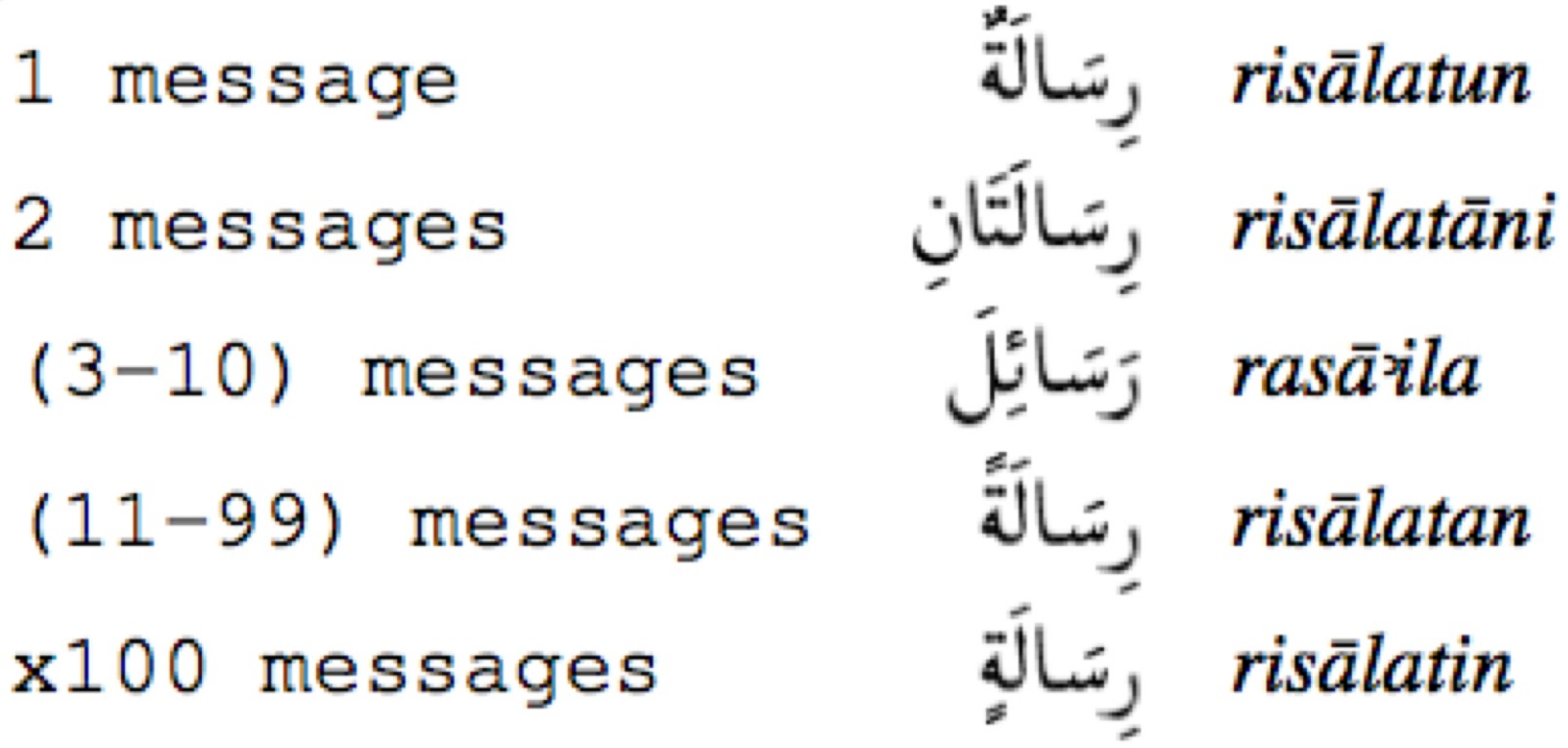
```
  PredVP youSg_NP (ComplV2 have_V2 (NumCN two_Num (UseN (mkN "messaggio"))))
  ===> hai due messaggi

  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	رِسَالَةٌ	*risālatun*
2 messages	رِسَالَتَانِ	*risālatāni*
(3–10) messages	رَسَائِلَ	*rasāʾila*
(11–99) messages	رِسَالَةً	*risālatan*
x100 messages	رِسَالَةٍ	*risālatin*

(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**

grammarian	**applications**	**resources**
expertise	application domain	linguistics
programming skills	programming in general	GF programming
language skills	practical use	theoretical knowledge

We want a **division of labour**.

Two kinds of grammars

Application grammars vs. **resource grammars**

grammar	**application**	**resource**
abstract syntax	semantic	syntactic
concrete syntax	using resource API	parameters, tables, records
lexicon	idiomatic, technical	just for testing
size	small or bigger	big

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

```
  lincat Person = NP ; Fact = Cl ;

  lin Like x y = PredVP x (ComplV2 like_V2 y)      -- Engligh
  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 abstrac 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 `mk`*C* where *C* is the value category.

The resource defines e.g.

```
  mkCl : NP -> V2 -> NP -> Cl = \subj,verb,obj ->
    PredVP subj (ComplV2 verb obj)
  mkCl : NP -> V -> Cl = \subj,verb ->
    PredVP subj (UseV verb)
```

Relevant part of Resource Grammar API for "Face"

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

Function	**example**
`mkCl : NP -> V2 -> NP -> Cl`	*John loves Mary*
`mkNP : Numeral -> CN -> NP`	*five cars*
`mkNP : Quant -> CN -> NP`	*that car*
`mkNP : Pron -> NP`	*we*
`mkCN : N -> CN`	*car*
`this_Quant : Quant`	*this, these*
`youSg_Pron : Pron`	*you* (singular)
`n1_Numeral, n2_Numeral : Numeral`	*one, two*
`n100_Numeral : Numeral`	*one hundred*
`have_V2 : V2`	*have*

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

```
--# -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

```
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`

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

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

Modules of a domain grammar: ”Face” community

1. Abstract syntax, `Face`

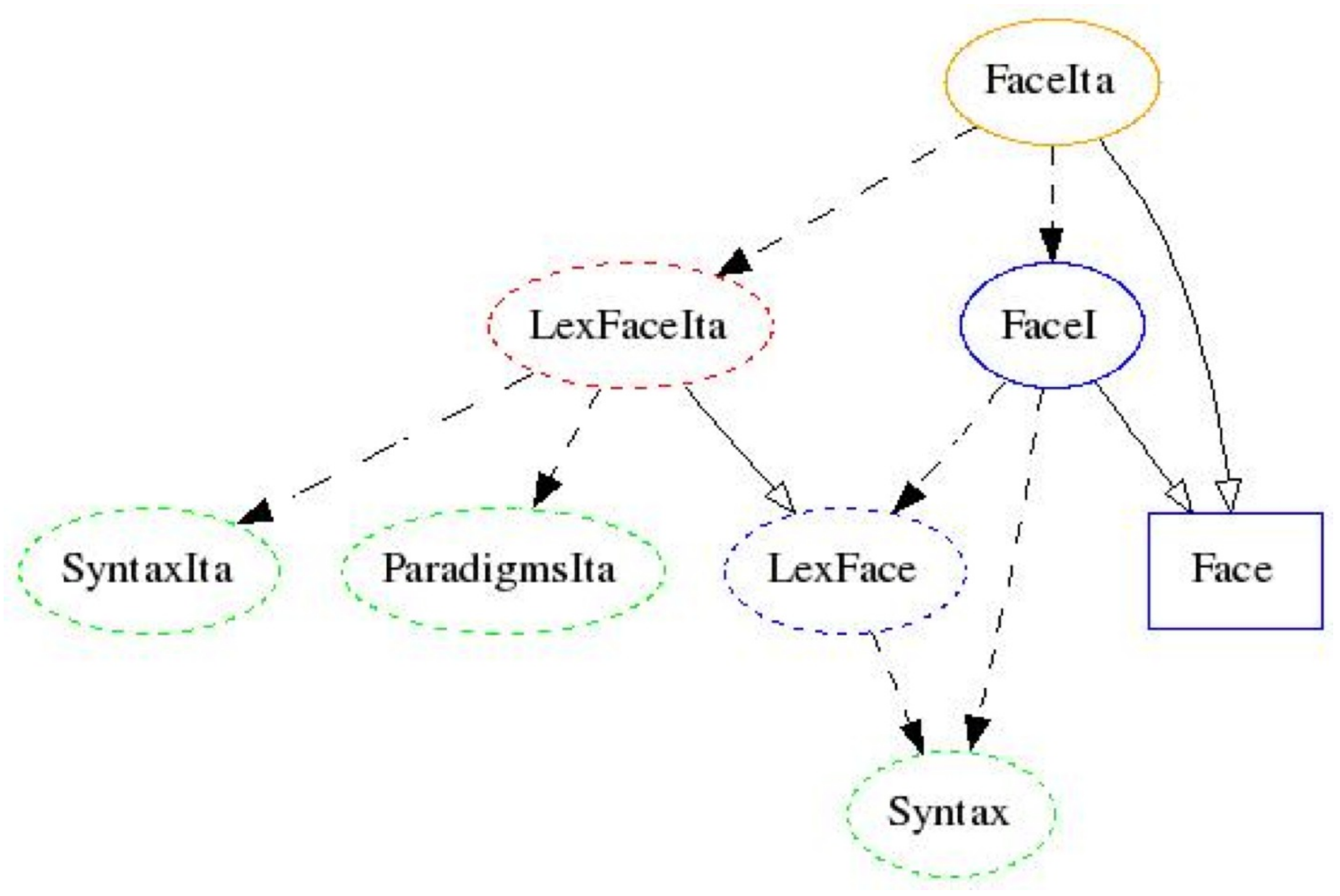
2. Parametrized concrete syntax: `FaceI`

3. Domain lexicon interface: `LexFace`

4. For each language *L*: domain lexicon instance `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

```
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 |.

```
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

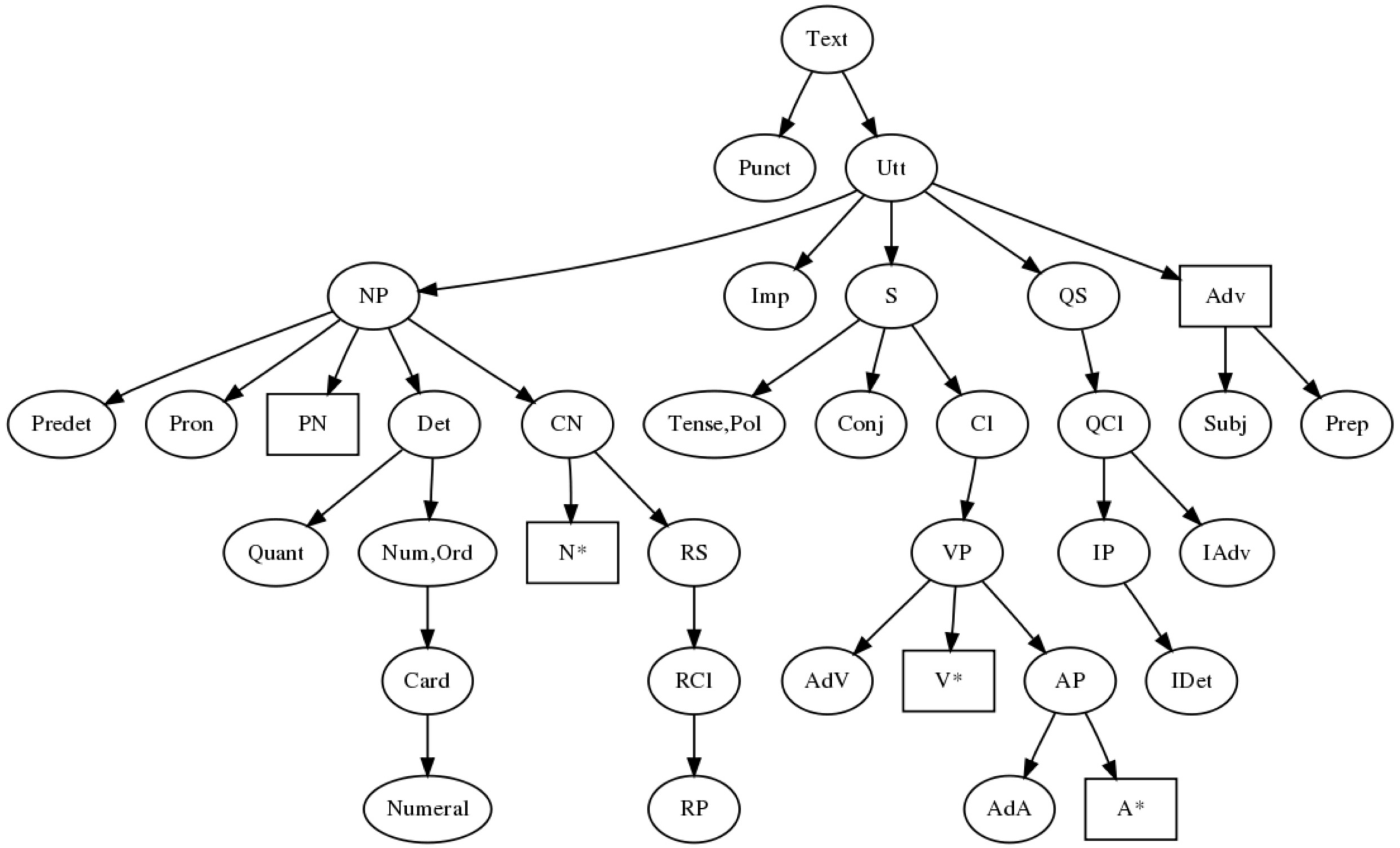
`digitalgrammars.com/gf/lib/resource/doc/synopsis.html`

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

Category	**Explanation**	**Example**
`Text`	sequence of utterances	*Does John walk? Yes.*
`Utt`	utterance	*does John walk*
`Imp`	imperative	*don't walk*
`S`	sencence (fixed tense)	*John wouldn't walk*
`QS`	question sentence	*who hasn't walked*
`Cl`	clause (variable tense)	*John walks*
`QCl`	question clause	*who doesn't walk*
`VP`	verb phrase	*love her*
`AP`	adjectival phrase	*very young*
`CN`	common noun phrase	*young man*
`Adv`	adverbial phrase	*in the house*

Lexical categories for building predicates

Cat	Explanation	Compl	Example
`A`	one-place adjective	`-`	*smart*
`A2`	two-place adjective	`NP`	*married* (*to her*)
`Adv`	adverb	`-`	*here*
`N`	common noun	`-`	*man*
`N2`	relational noun	`NP`	*friend* (*of John*)
`NP`	noun phrase	`-`	*the boss*
`V`	one-place verb	`-`	*sleep*
`V2`	two-place verb	`NP`	*love* (*her*)
`V3`	three-place verb	`NP, NP`	*show* (*it to me*)
`VS`	sentence-complement verb	`S`	*say* (*that I run*)
`VV`	verb-complement verb	`VP`	*want* (*to run*)

Functions for building predication clauses

Fun	Type	Example
`mkCl`	`NP -> V -> Cl`	*John walks*
`mkCl`	`NP -> V2 -> NP -> Cl`	*John loves her*
`mkCl`	`NP -> V3 -> NP -> NP -> Cl`	*John sends it to her*
`mkCl`	`NP -> VV -> VP -> Cl`	*John wants to walk*
`mkCl`	`NP -> VS -> S -> Cl`	*John says that it is good*
`mkCl`	`NP -> A -> Cl`	*John is old*
`mkCl`	`NP -> A -> NP -> Cl`	*John is older than Mary*
`mkCl`	`NP -> A2 -> NP -> Cl`	*John is married to her*
`mkCl`	`NP -> AP -> Cl`	*John is very old*
`mkCl`	`NP -> N -> Cl`	*John is a man*
`mkCl`	`NP -> CN -> Cl`	*John is an old man*
`mkCl`	`NP -> NP -> Cl`	*John is the man*
`mkCl`	`NP -> Adv -> Cl`	*John is here*

Noun phrases and common nouns

Fun	**Type**	**Example**
`mkNP`	`Quant -> CN -> NP`	*this man*
`mkNP`	`Numeral -> CN -> NP`	*five men*
`mkNP`	`PN -> NP`	*John*
`mkNP`	`Pron -> NP`	*we*
`mkNP`	`Quant -> Num -> CN -> NP`	*these (five) man*
`mkCN`	`N -> CN`	*man*
`mkCN`	`A -> N -> CN`	*old man*
`mkCN`	`AP -> CN -> CN`	*very old Chinese man*
`mkNum`	`Numeral -> Num`	*five*
`n100_Numeral`	`Numeral`	*one hundred*
`plNum`	`Num`	(plural)

Questions and interrogatives

Fun	Type	Example
`mkQCl`	`Cl -> QCl`	*does John walk*
`mkQCl`	`IP -> V -> QCl`	*who walks*
`mkQCl`	`IP -> V2 -> NP -> QCl`	*who loves her*
`mkQCl`	`IP -> NP -> V2 -> QCl`	*whom does she love*
`mkQCl`	`IP -> AP -> QCl`	*who is old*
`mkQCl`	`IP -> NP -> QCl`	*who is the boss*
`mkQCl`	`IP -> Adv -> QCl`	*who is here*
`mkQCl`	`IAdv -> Cl -> QCl`	*where does John walk*
`mkIP`	`CN -> IP`	*which car*
`who_IP`	`IP`	*who*
`why_IAdv`	`IAdv`	*why*
`where_IAdv`	`IAdv`	*where*

Sentence formation, tense, and polarity

Fun	Type	Example
`mkS`	`Cl -> S`	*he walks*
`mkS`	`(Tense)->(Ant)->(Pol)->Cl -> S`	*he wouldn't have walked*
`mkQS`	`QCl -> QS`	*does he walk*
`mkQS`	`(Tense)->(Ant)->(Pol)->QCl -> QS`	*wouldn't he have walked*

Function	Type	Example
`conditionalTense`	`Tense`	(*he would walk*)
`futureTense`	`Tense`	(*he will walk*)
`pastTense`	`Tense`	(*he walked*)
`presentTense`	`Tense`	(*he walks*) [default]
`anteriorAnt`	`Ant`	(*he has walked*)
`negativePol`	`Pol`	(*he doesn't walk*)

Utterances and imperatives

Fun	**Type**	**Example**
`mkUtt`	`Cl -> Utt`	*he walks*
`mkUtt`	`S -> Utt`	*he didn't walk*
`mkUtt`	`QS -> Utt`	*who didn't walk*
`mkUtt`	`Imp -> Utt`	*walk*
`mkImp`	`V -> Imp`	*walk*
`mkImp`	`V2 -> NP -> Imp`	*find it*
`mkImp`	`AP -> Imp`	*be brave*

More

Texts: *Who walks? John. Where? Here!*

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/Face`*L* (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.

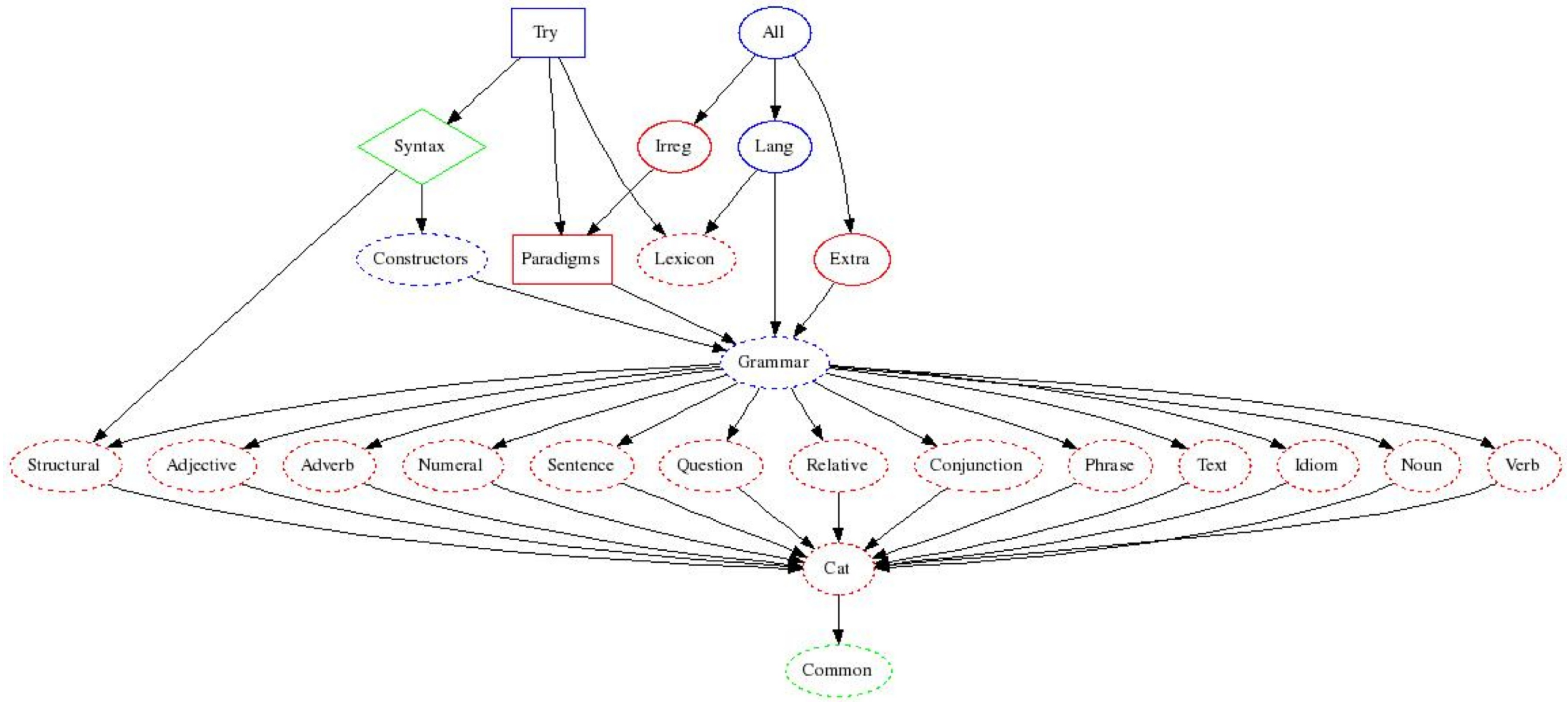
Lecture 5

Inside the

Resource Grammar Library

Contents

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

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

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

Effort statistics, completed languages

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

Lines of source code in v. 1.4, rough estimates of person months. * = generated code.

Languages under construction

language	started	% morpho	% lexicon	% syntax
Arabic	2005	90	95	20
Hindi	2008	70	3	5
Latin	2008	80	2	5
Polish	2007	95	60	2
Romanian	2009	95	60	1
Thai	2007	70	2	2
Turkish	2007	30	10	5

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`
8. `NounMar`: `lin DetCN, DetQuant`
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`

```
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.