A GF Tutorial

MOLTO

Aarne Ranta

MOLTO Kick-Off Meeting, Barcelona, 11 March 2010

Plan

GF and multilingual grammars

Morphology and syntax

Building a translation system

Probabilities in GF

Compiling GF

Hands-on: Attempto in Catalan

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**, and **translation**

Multilingual Grammars

Multilingual grammars in compilers

Source and target language related by **abstract syntax**

```
                                                  iconst_2
                                                  iload_0
2 * x + 1  <----->  plus (times 2 x) 1  <------>  imul
                                                  iconst_1
                                                  iadd
```

A GF grammar for expressions

```
abstract Expr = {
  cat Exp ;
  fun plus : Exp -> Exp -> Exp ;
  fun times : Exp -> Exp -> Exp ;
  fun one, two : Exp ;
  }
```

```
concrete ExprJava of Expr = {
  lincat Exp = Str ;
  lin plus x y = x ++ "+" ++ y ;
  lin times x y = x ++ "*" ++ y ;
  lin one = "1" ;
  lin two = "2" ;
  }
```

```
concrete ExprJVM of Expr= {
  lincat Expr = Str ;
  lin plus x y = x ++ y ++ "iadd" ;
  lin times x y = x ++ y ++ "imul" ;
  lin one = "iconst_1" ;
  lin two = "iconst_2" ;
  }
```

Multilingual grammars in natural language

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

Natural language structures

Predication: *John* + *loves Mary*

Complementation: *love* + *Mary*

Noun phrases: *John*

Verb phrases: *love Mary*

2-place verbs: *love*

Abstract syntax of sentence formation

```
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" ;
}
```

Multilingual grammar

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 in GF

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 random generation: linearize into all languages

```
> gr | l
Pred Mary (Compl Love Mary)
Mary loves Mary
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, Dutch

```
concrete ZeroDut 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 = "Jan" ;
    Mary = "Marie" ;
    Love = {v = "heeft" ; p = "lief"} ;
}
```

The verb *heeft lief* is a **discontinuous constituent**.

Record types and records

The linearization type of `V2` is a **record type**

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

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

```
lin Love = {v = "heeft" ; p = "lief"}
```

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

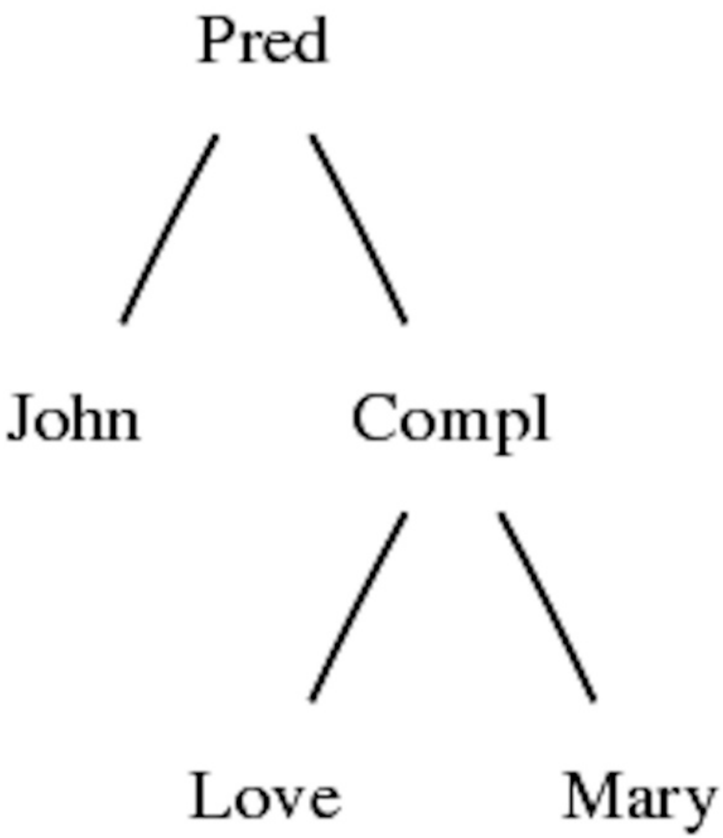
```
lin Compl v2 np = v2.v ++ np ++ v2.p
```

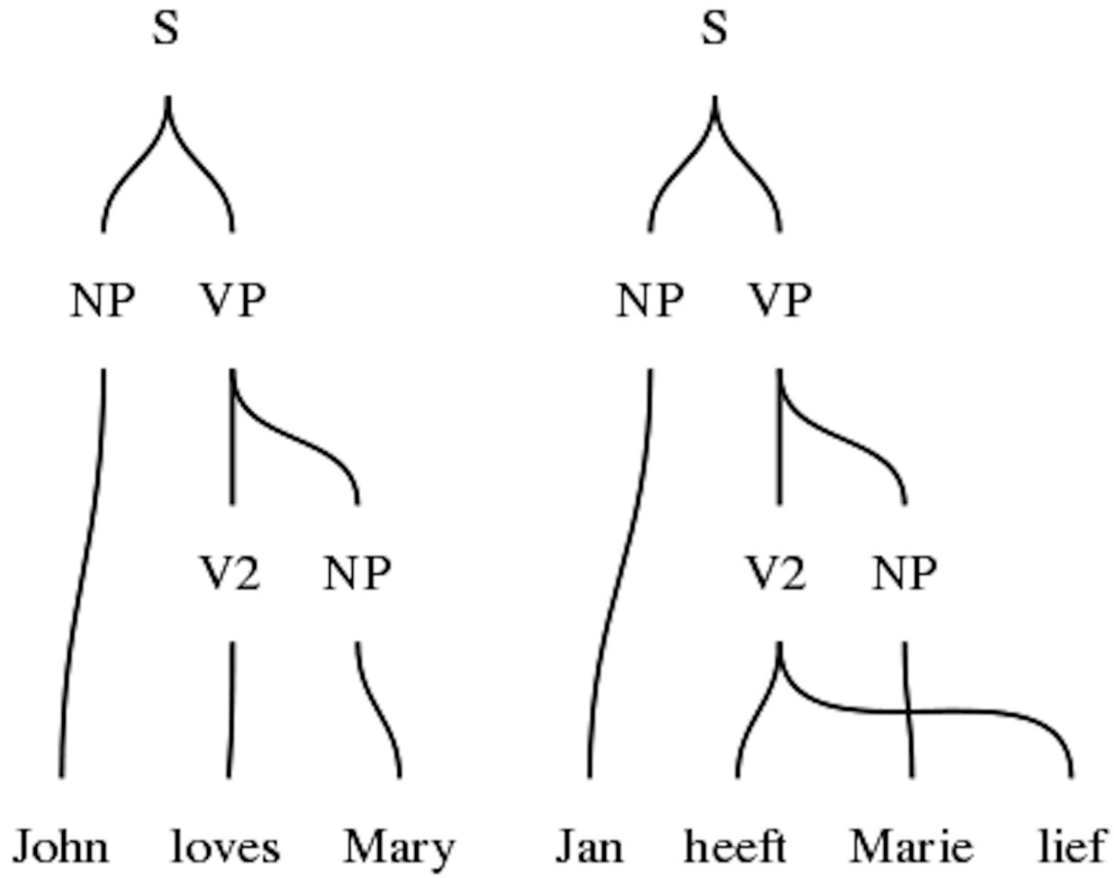
Concrete syntax, Hebrew

```
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.

Abstract trees and parse trees





From abstract trees to parse trees

Link every **word** with its **smallest spanning subtree**

Replace every **constructor function** with its **value category**

Generating word alignment

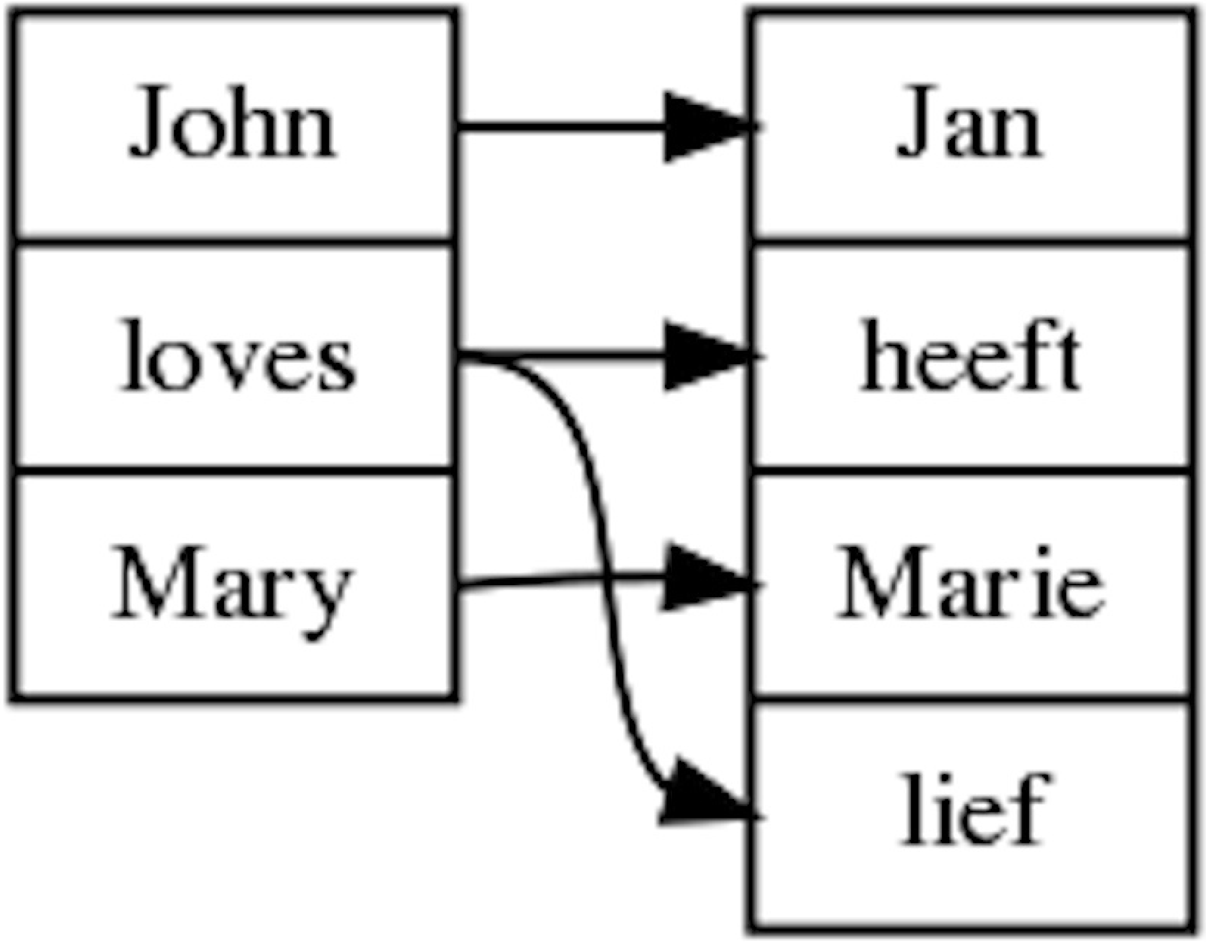
In L1 and L2: link every word with its smallest spanning subtree

Delete the intervening tree, combining links directly from L1 to L2

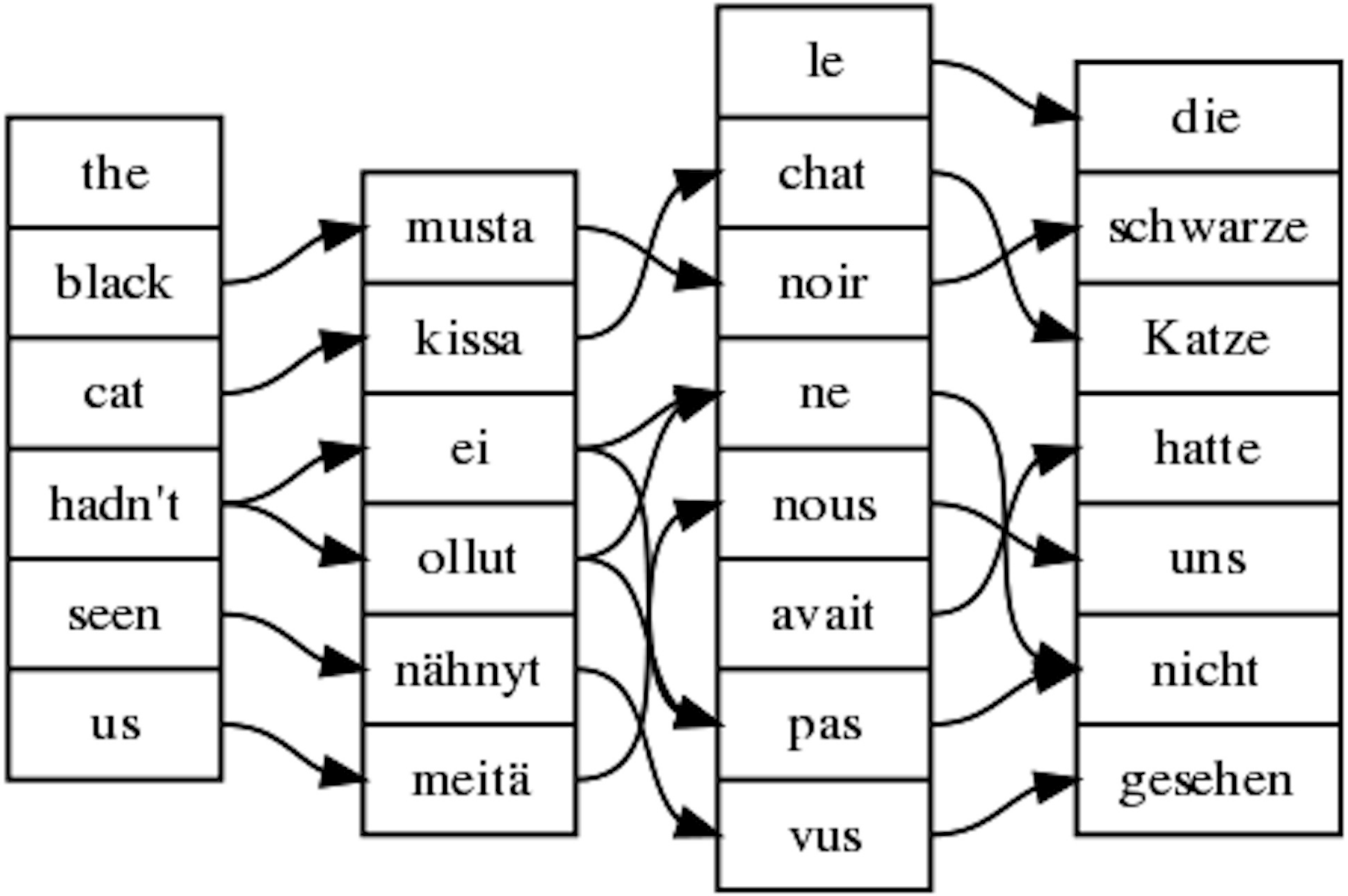
Notice: in general, this gives **phrase alignment**

Notice: links can be crossing, phrases can be discontinuous

Word alignment via trees



A more involved word alignment



The GF Resource Grammar Library

Morphology and basic syntax

Common API for different languages

Currently (March 2010) 16 languages: Bulgarian, Catalan, Danish, Dutch, English, Finnish, French, German, Interlingua, Italian, Norwegian, Polish, Romanian, Russian, Spanish, Swedish.

Under construction for more languages: Arabic, Farsi, Hebrew, Hindi/Urdu, Icelandic, Japanese, Latin, Latvian, Maltese, Mongol, Portuguese, Swahili, Thai, Tswana, Turkish. (Summer School 2009)

Programming in GF: Morphology and smart paradigms

Inflectional morphology

Goal: a complete system of inflection paradigms

Paradigm: a function from "basic form" to full inflection table

GF morphology is inspired by

- Zen (Huet 2005): typeful functional programming
- XFST (Beesley and Karttunen 2003): regular expressions

Example: English verb inflection

Start by defining parameter types and parts of speech.

```
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

```
oper
  mkVerb : (_,_,_,_,_ : Str) -> Verb = \go,goes,went,gone,going -> {
    s = table {
      VInf => go ;
      VPres => goes ;
      VPast => went ;
      VPastPart => gone ;
      VPresPart => going
      }
    } ;
```

Defining paradigms

A paradigm is an operation of type

```
Str -> Verb
```

which takes a string and returns an inflection table.

E.g. 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 in GF

```
> 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 ;
} ;
```

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 and patterns are records of strings.

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

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

Applying a pattern is intertwining the records.

```
appPattern : Root -> Pattern -> Str = \r,p ->
  p.F + r.F + p.FC + r.C + p.CL + r.L + p.L ;
```

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

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
```

Grammars as software libraries

Complexity of grammar writing

Typical GF tasks:

- natural language interfaces
- localization of programs

We need

- domain expertise: technical and idiomatic expression
- linguistic expertise: how to inflect words and build phrases

Example: an email program

Task: generate phrases saying *you have n message*(*s*)

Domain expertise: choose correct words (in Swedish, not *budskap* but *meddelande*)

Linguistic expertise: avoid *you have one messages*

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 El Dada, ACL workshop on Arabic, Prague 2007)

Division of labour

Application grammars

- abstract syntax: semantic model of domain

- authors: domain experts

Resource grammars

- abstract syntax: grammatical categories and rules

- authors: linguists

Resource grammar API

Smart paradigms for morphology

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

Abstract syntax functions for syntax

```
mkCl : NP -> V2 -> NP -> Cl   -- John loves Mary
mkNP : Numeral -> CN -> NP    -- five houses
```

Using the library in English

```
mkCl youSg_NP have_V2 (mkNP n2_Numeral (mkN "message"))
===> you have two messages

mkCl youSg_NP have_V2 (mkNP n1_Numeral (mkN "message"))
===> you have one message
```

Localization

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

```
mkCl youSg_NP have_V2 (mkNP n2_Numeral (mkN "messaggio"))
===> hai due messaggi

mkCl youSg_NP have_V2 (mkNP n2_Numeral (mkN "meddelande"))
===> du har två meddelanden

mkCl youSg_NP have_V2 (mkNP n2_Numeral (mkN "viesti"))
===> sinulla on kaksi viestiä
```

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

Meaning-preserving translation

Translation must preserve meaning.

It need not preserve syntactic structure.

Sometimes this 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 = mkCl x like_V2 y      -- Engligh
lin Like x y = mkCl y 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 any other natural language, 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, **type theory**, can be used for any of these!

Example: abstract syntax for a "Facebook" community

What messages can be expressed on the community page?

```
abstract Face = {

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 ;
}
```

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 : Det -> CN -> NP`	*that car*
`mkNP : Pron -> NP`	*we*
`mkCN : N -> CN`	*car*
`this_Det : Det`	*this*
`youSg_Pron : Pron`	*you* (singular)
`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_Det o) ;
  You = mkNP youSg_Pron ;
  Friend = mkCN friend_N ;
  Invitation = mkCN invitation_N ;
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_Det o) ;
  You = mkNP youSg_Pron ;
  Friend = mkCN friend_N ;
  Invitation = mkCN invitation_N ;
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_Det o) ;
  You = mkNP youSg_Pron ;
  Friend = mkCN friend_N ;
  Invitation = mkCN invitation_N ;
}
```

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

```
concrete FaceEng of Face = FaceI with
  (Syntax = SyntaxEng),
  (LexFace = LexFaceEng) ;
```

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`

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

Porting the grammar to Italian

1. Domain lexicon: use Italian paradigms and words, e.g.

```
  like_V2 = mkV2 (mkV (piacere_64 "piacere")) dative ;
```

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_Det o) like_V2 p ;
}
```

Grammars with probabilities

Configurations files: map funs to probs

```
ActVP  0.9
PassVP 0.1
```

These define tree probabilities. Applications:

- Ranking of parse trees

- Biasing random generation, which can also be guided by a tree pattern:

  ```
  > gr -probs=file (PredVP ? (ComplAP ?))
  ```

  "Generate adjectival predications biased by probs in file".

Compiling GF grammars

PGF = Portable Grammar Format

The "machine language" of GF

Equivalent to PMCFG (Parallel Multiple Context-Free Grammars), (Seki & al. 1991)

Polynomial parsing

Binary format, interpreters in Haskell, JavaScript, Java (under construction).

The two languages

Source GF	**Target PGF**
algebraic datatypes	integers
tables, records	tuples
pattern matching	tuples
variables	indices
operations	-
module hierarchy	just abstract + concretes
functors	-

Example of PGF

```
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" ;
```

```
Pred = $0.0 ++ $1
Compl = $1.1 ++ $0
John =
  <"Ioannes","Ioannem"
Mary =
  <"Maria","Mariam">
Love = "amat"
```

Grammar composition

Using the abstract syntax of a resource to give the concrete syntax of an application.

The complex trees of the resource disappear in grammar compilation:

```
  lin HaveMess n
   ==> PredVP
         (UsePron youSg_Pron)
         (ComplV2 have_V2
                  (DetCN (DetQuant Indef (Num n)) (UseN message_N)))
   ==> "you" ++ "have" ++ n.0 ++ <"message","messages">.(n.1)
```

Method: partial evaluation, deforestation.

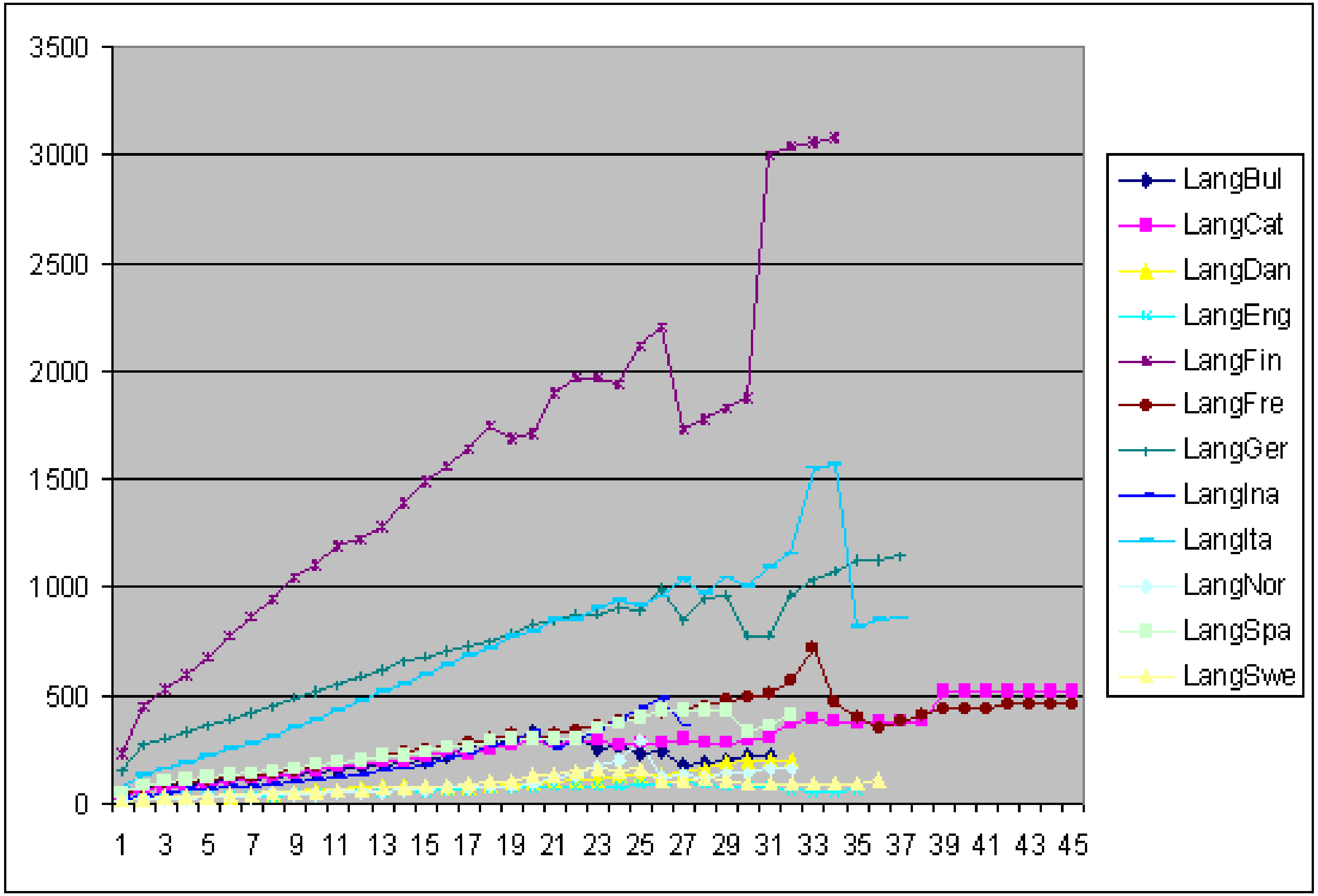
Incremental parsing

(Krasimir Angelov, EACL 2009)

Produce predictions while parsing

Useful as authoring aids

Parsing complexity in the resource grammar



Web applications

PGF grammars embedded in Haskell can be used in **web servers**

- fridge magnet demo: `tournesol.cs.chalmers.se:41296/fridge`
- translator demo: `tournesol.cs.chalmers.se:41296/translate`

More information in Krasimir's lecture.

Further reading

The GF Summer School Tutorial, a longer version of this one.

Video lectures of the Summer School Tutorial.

A longer tutorial.

AR, *Grammatical Framework: A Programming Language for Multilingual Grammars and Their Applications*, CSLI Publications, 2010, to appear.

`www.grammaticalframework.org`