Multilingual Packages of Controlled Languages

An Introduction to GF

Aarne Ranta

CNL-2010, Marettimo, 14 September 2010

Multilingual On-Line Translation, FP7-ICT-247914
Contents

Controlled languages and multilinguality

GF: a multilingual grammar formalism

Example: a ”John and Mary” grammar in five languages

The GF Resource Grammar Library

Example: a Facebook message grammar

Example: Attempto in six languages

Hands-on: port Attempto to a new language; add new rules
Controlled languages and multilinguality
Our definition of a controlled language

(Not the only one!)

Controlled language = language defined by a formal grammar

Programming languages are controlled languages

Fragments of natural languages can be made into controlled languages

N.B. The language may be ambiguous!
Translation with controlled languages

Due to formal grammar, the analysis part of translation is easier.

Our approach: grammars map to a common abstract syntax.

The abstract syntax is an interlingua of translation.

Source and target languages via different concrete syntaxes.

Cf. compilation as translation between computer languages.
Multilingual grammars in compilers

Source and target language related by abstract syntax

\[ 2 \times x + 1 <----- \text{plus} (\text{times} 2 \times x) 1 <----- \]

\[ \text{iconst}_2 \]
\[ \text{iload}_0 \]
\[ \text{imul} \]
\[ \text{iconst}_1 \]
\[ \text{iadd} \]
Multilingual grammars for natural languages
The rationale for multilinguality

(A Almost) any controlled language has a **multilingual generalization**

It gives translation

It also gives **collaborative authoring**: input and output in all involved languages

Desired features:

- **reversible** mapping between abstract and concrete syntax (**parsing** and **linearization**)

- reuse of natural language grammars as **libraries**
Grammatical Framework (GF): a multilingual grammar formalism
History

Background: type theory, logical frameworks (LF)

GF = LF + concrete syntax

Started at XRCE in 1998 for multilingual document authoring, in particular for controlled languages

Demo: multilingual phrasebook in molto-project.eu

Demo: query language in molto.ontotext.com
Factoring out functionalities

GF grammars are declarative programs that define

- parsing
- generation
- translation
- editing

Some of this can also be found in BNF/Yacc, HPSG/LKB, LFG/XLE...
Factoring out linguistics

The GF Resource Grammar Library

Morphology, syntax, and lexicon for 16 languages

Controlled languages can be defined as subsets of these languages

Some of this can be found in CLE and Regulus
Obtaining GF

Homepage

http://www.grammaticalframework.org

Minimal installation: go to "Download", and obtain

• binary for your platform (Linux, Mac, Windows)

• the resource grammar library (optional, platform independent)
To know more

"Tutorial" on GF homepage


2nd GF Summer School, Barcelona, 15-26 August 2011.
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 Exp = 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
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 parsing to linearization

> 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,

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

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

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

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

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

\[ \text{lincat } V2 = \{v : \text{Str} ; p : \text{Str}\} \]

The linearization of Love is a record

\[ \text{lin Love} = \{ v = "heeft" ; p = "lief"\} \]

The values of fields are picked by projection (.)

\[ \text{lin Compl v2 np} = \text{v2.v} ++ \text{np} ++ \text{v2.p} \]
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 = "יון" ; g = Masc} ;
    Mary = {s = "מריא" ; g = Fem} ;
    Love = table {Masc => "אהבה" ; Fem => "dbe'a"} ;
    param
        Gender = Masc | Fem ;
}

The verb **agrees** to the gender of the subject.
Abstract trees and parse trees

```
Pred
  /
John
/
Love
/
Compl
/
Mary

S
  /
NP  VP
   /
V2  NP
   /
John  loves  Mary

S
  /
NP  VP
   /
V2  NP
   /
Jan  heeft  Marie  lief
```
From abstract trees to parse trees

Link every word with its smallest spanning subtree

Replace every constructor function with its value category
Word alignment via trees

John  
loves  
Mary  

Jan  
heeft  
Marie  lief
A more involved word alignment
Exercises

1. Implement the "John and Mary" grammar for another language.

2. Add the pronouns *I* and *you* to NP's - both plural and singular *you*.

3. Add adjectival predication, e.g. *John is old*
The GF Resource Grammar Library
Scope

Morphology and basic syntax

Common API for different languages

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

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

Or: how to avoid giving three forms of all new verbs.

Start by defining parameter types and parts of speech.

```latex
\begin{verbatim}
param
    VForm = VInf | VPres | VPast | VPastPart | VPresPart ;

oper
    Verb : Type = \{ s : VForm => Str \} ;
\end{verbatim}
```

Judgement form oper: auxiliary operation.
Start: worst-case function

To save writing and to abstract over the Verb type

\[
\text{mkVerb} : (_,_,_,_,_,_ : \text{Str}) \to \text{Verb} = \{ \text{go,goes,went,gone,going} \to \{
\begin{align*}
\text{VInf} & \Rightarrow \text{go} ; \\
\text{VPres} & \Rightarrow \text{goes} ; \\
\text{VPast} & \Rightarrow \text{went} ; \\
\text{VPastPart} & \Rightarrow \text{gone} ; \\
\text{VPresPart} & \Rightarrow \text{going}
\end{align*}
\}
\} ;
\]
Defining paradigms

A paradigm is an operation of type

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

which takes a string and returns an inflection table.

E.g. regular verbs:

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

This will work for \textit{walk, interest, play}.

It will not work for \textit{sing, kiss, use, cry, fly, stop}.
More paradigms

For verbs ending with s, x, z, ch

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

For verbs ending with e

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

Notice:

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

For verbs ending with \textit{y}

\begin{verbatim}
  y_regVerb : Str -> Verb = \cry ->
    let cr = init cry
    in
    mkVerb cry (cr + "ies") (cr + "ied") (cr + "ied") (cry + "ing") ;
\end{verbatim}

For verbs ending with \textit{ie}

\begin{verbatim}
  ie_regVerb : Str -> Verb = \die ->
    let dy = Predef.tk 2 die + "y"
    in
    mkVerb die (die + "s") (die + "d") (die + "d") (dy + "ing") ;
\end{verbatim}
What paradigm to choose

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

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

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

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

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

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

Let GF choose the paradigm by pattern matching on strings

\[
\text{smartVerb} : \text{Str} \to \text{Verb} = \backslash v \to \text{case } v \text{ of }
\]

\[
\begin{align*}
_ + ("s" | "z" | "x" | "ch") & \Rightarrow \text{s_regVerb } v \\
_ + "ie" & \Rightarrow \text{ie_regVerb } v \\
_ + "ee" & \Rightarrow \text{ee_regVerb } v \\
_ + "e" & \Rightarrow \text{e_regVerb } v \\
_ + ("a" | "e" | "o" | "u") + "y" & \Rightarrow \text{regVerb } v \\
_ + "y" & \Rightarrow \text{y_regVerb } v \\
_ & \Rightarrow \text{regVerb } v
\end{align*}
\]
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 classified classifying
The smart paradigm is not perfect

Irregular verbs are obviously not covered

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

Neither are regular verbs with consonant duplication

> cc -all smartVerb "stop"
stop stops stoped stoped stoping
The final consonant duplication paradigm

Use the Prelude function last

\[
\text{dupRegVerb} : \text{Str} \to \text{Verb} = \lambda \text{stop} \to \\
\begin{array}{l}
\text{let stopp} = \text{stop} + \text{last} \ \text{stop} \\
\text{in} \\
\text{mkVerb stop (stop + "s") (stopp + "ed") (stopp + "ed") (stopp + "ing")}
\end{array}
\]

String pattern: relevant consonant preceded by a vowel

_ + ("a"|"e"|"i"|"o"|"u") + ("b"|"d"|"g"|"m"|"n"|"p"|"r"|"s"|"t")

=> \text{dupRegVerb} \ v ;
Testing consonant duplication

Now it works

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

But what about

> cc -all smartVerb "coat"
  coat coats coatted coatted coatting

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

_ + ("ea"|"ee"|"ie"|"oa"|"oo"|"ou") + ? => regVerb v ;
There is no waterproof solution

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

- *omit* [o’mit]: omitted, omitting

- *vomit* ['vomit]: vomited, vomiting

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

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

Arguments: three forms instead of one.

Pattern matching done in regular verbs can be reused.

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

Rarely used: the library IrregEng.gf gives sing_V etc.
Putting it all together

We have three functions:

\[
\begin{align*}
\text{smartVerb} : \text{Str} \to \text{Verb} \\
\text{irregVerb} : \text{Str} \to \text{Str} \to \text{Str} \to \text{Verb} \\
\text{mkVerb} : \text{Str} \to \text{Str} \to \text{Str} \to \text{Str} \to \text{Str} \to \text{Verb}
\end{align*}
\]

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.

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

Only the first commonly used, thanks to the library IrregEng.gf.

Library convention: functions for constructing \( C \) are named \( \text{mk}C \).
Grammars as software libraries
Complexity of grammar writing

To implement a controlled language, 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
2 messages
(3–10) messages
(11–99) messages
x100 messages

(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

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

Abstract syntax functions for syntax

\[ \text{mkCl} : \text{NP} \rightarrow \text{V2} \rightarrow \text{NP} \rightarrow \text{Cl} \quad -- \quad \text{John loves Mary} \]
\[ \text{mkNP} : \text{Numeral} \rightarrow \text{CN} \rightarrow \text{NP} \quad -- \quad \text{five houses} \]
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

\[ \text{lin} \text{cat} \text{ Person} = \text{NP} ; \text{Fact} = \text{Cl} ; \]

\[ \text{lin} \text{ Like} \ x \ y = \text{mkCl} \ x \ \text{like}_V^2 \ y \quad -- \text{English} \]
\[ \text{lin} \text{ Like} \ x \ y = \text{mkCl} \ y \ \text{piacere}_V^2 \ x \quad -- \text{Italian} \]

From syntactic point of view, we perform transfer, i.e. structure change.

GF has compile-time transfer, and uses interlingua (semantic abstract syntax) at run time.
Domain semantics

"Semantics of English", or 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkCl : NP -&gt; V2 -&gt; NP -&gt; Cl</td>
<td>John loves Mary</td>
</tr>
<tr>
<td>mkNP : Numeral -&gt; CN -&gt; NP</td>
<td>five cars</td>
</tr>
<tr>
<td>mkNP : Det -&gt; CN -&gt; NP</td>
<td>that car</td>
</tr>
<tr>
<td>mkNP : Pron -&gt; NP</td>
<td>we</td>
</tr>
<tr>
<td>mkCN : N -&gt; CN</td>
<td>car</td>
</tr>
<tr>
<td>this_Det : Det</td>
<td>this</td>
</tr>
<tr>
<td>youSg_Pron : Pron</td>
<td>you (singular)</td>
</tr>
<tr>
<td>have_V2 : V2</td>
<td>have</td>
</tr>
</tbody>
</table>
Concrete syntax for English

Use 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

Use the library.

custom 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;
   }


Exercise

Port the Face grammar to another language.

Add words and message forms.
Attempto in GF
ACE, Attempto Controlled English

University of Zurich

http://attempto.ifi.uzh.ch/
What has been done

"Full Attempto" in six languages

Syntax: 200 rules

Lexicon: 100 words
Mini Attempto

Syntax: 60 rules

Lexicon: 50 words
Module structure

- TestAttemptoEng
- AttemptoEng
- Attemptol
- LexAttemptoEng
- LexAttempto
- TestAttempto
- Attempto
Roles of modules

Attempto: core syntax

TestAttempto: test lexicon
Hands-on 1

1. Clone modules to \( L \)

2. Write \texttt{LexAttemptoL} \( L \)

3. Write \texttt{TestAttemproL}
Hands-on 2

Add a couple of words (*man, animal*)

Add a syntax rule (*NP is a CN*)