GF - Grammatical Framework

Aarne Ranta

Workshop on GF and UNL, IIT Bombay, May 2008

Plan for the lectures

1. GF: Compiling Natural Language
2. Hindi Morphology in GF
3. Translation in GF
4. The GF Resource Grammar Library

GF: Compiling Natural Language

The GF grammar formalism

Multilingual grammars: English and Hindi

Demos of applications

The GF grammar formalism

Functionalities

Grammar: the definition of a language

A declarative program for

- **parsing**: from strings to syntax trees
- **linearization** (pretty printing): from syntax trees to strings
- ...

```
2 + 3 * x   <---->  EAdd (Eint 2) (EMul (EInt 3) (EVar x))
```

The BNF grammar formalism

Grammar = set of **labelled BNF rules** (Backus Naur Form)

Example (ignoring precedences)

```
EAdd. Exp ::= Exp "+" Exp
EMul. Exp ::= Exp "*" Exp
EVar. Exp ::= Ident
EInt. Exp ::= Integer
```

Writing essentially this in YACC gives a parser. Labels are **tree constructors**:

```
Exp '+' 'Exp' {new EAdd($1, $3)}
```

GF

GF was born in 1998 at Xerox Research Centre Europe, Grenoble

- multilingual document authoring: ”write a document in a language you don’t know while seeing it evolve in a language you know”

Demo: letter editor

Open-source software licensed under GNU

GF homepage: `digitalgrammars.com/gf`

BNFC

The **BNF Converter** (BNFC) is a compiler that generates both a parser and a pretty printer from the above rules.

BNFC is a spin-off of GF, born in 2002 at Chalmers

- specialized to describe programming languages
- less general but more efficient than GF
- generates codes for standard YACC-like tools

BNFC homepage: `digitalgrammars.com/bnfc`

We have used BNFC to make GF support new source and target formats of grammars.

Abstract and concrete syntax

Divide a Labelled BNF rule to two rules:

- **abstract syntax**: declares a tree constructing function
- **concrete syntax**: defines a translation from trees to strings

```
EAdd. Exp ::= Exp "+" Exp

fun EAdd : Exp -> Exp -> Exp
lin EAdd x y = x ++ "+" ++ y
```

This is the interpretation of Labelled BNF in GF.

Multilingual grammars

One abstract syntax + many concrete syntaxes

```
fun EAdd : Exp -> Exp -> Exp

lin EAdd x y = x ++ "+" ++ y     -- C, Java

lin EAdd x y = x ++ y ++ "iadd"  -- JVM

lin EAdd x y = "the" ++ "sum" ++ "of" ++ x ++ "and" ++ y  -- English
```

Quiz: which of these is ambiguous?

Translation

Translation from A to B = parsing in A and linearization to B

```
2 + 3 * x

EAdd (Eint 2) (EMul (EInt 3) (EVar x))

iconst_2  iconst_3  iload_0  imul  iadd
```

A multilingual grammar is a declarative compiler implementation.

Type checking in abstract syntax

GF can use **dependent types** to force type-correctness

```
fun EAdd : (t : Typ) -> Exp t -> Exp t -> Exp t
fun TVar : (t : Typ) -> Var t -> Exp t
fun TInt : Int -> Exp TInt
```

The type information can be used for instruction selection in linearization

```
lin EAdd t x y = x ++ y ++ addInstr t
```

Parsing and type checking

In usual notation, the type is suppressed in linearization

```
lin EAdd _ x y = x ++ "+" ++ y
```

Parsing involves **metavariable resolution** using dependent type checking

```
2 + x  --->
EAdd ?1 (EInt x) (EVar ?2 y)  ---> (?1 = ?2 = TInt)
EAdd TInt (EInt x) (EVar TInt y)
```

Type checking in natural language

The monkey ate the banana because ***it** *was hungry.*

*Le singe a mangé la banane parce qu'***il** *avait faim.*

The monkey ate the banana because ***it** *was ripe.*

*Le singe a mangé la banane parce qu'***elle** *était mûre.*

Pronouns and types

A pronoun is a noun phrase for a certain type of object:

```
fun Pron : Typ -> NP Typ
```

Its linearization only indicates the gender of the type:

```
lin Pron t = case t.g of {   -- English
  Male   => "he" ;
  Female => "she ;
  _ => "it"
  }
lin Pron t = case t.g of {   -- French
  Masc => "il" ;
  Fem  => "elle"
  }
```

Anaphora resolution is based on restoring the type.

The GF system

The compiler of the GF programming language

- generates parsing, linearization, and type checking code
- targets: C++, Haskell, Java, JavaScript

The interactive GF interpreter

- a tool for the grammarian
- shell for commands such as parsing, linearization, and random generation

Multilingual grammars: English and Hindi

A natural language grammar in BNF

File `food.cf` (`cf` = context-free)

```
Is.         Phrase  ::= Item "is" Quality ;
That.       Item    ::= "that" Kind ;
This.       Item    ::= "this" Kind ;
QKind.      Kind    ::= Quality Kind ;
Cheese.     Kind    ::= "cheese" ;
Fish.       Kind    ::= "fish" ;
Wine.       Kind    ::= "wine" ;
Italian.    Quality ::= "Italian" ;
Boring.     Quality ::= "boring" ;
Delicious.  Quality ::= "delicious" ;
Expensive.  Quality ::= "expensive" ;
Fresh.      Quality ::= "fresh" ;
Very.       Quality ::= "very" Quality ;
Warm.       Quality ::= "warm" ;
```

Using the grammar in GF, 1

Importing

```
> import food.cf
```

Parsing (in category `Phrase`)

```
> parse -cat=Phrase "this wine is very fresh"

Is (This Wine) (Very Fresh)
```

Linearization

```
> linearize Is (That Cheese) Italian

that cheese is Italian
```

Using the grammar in GF, 2

Pipe

```
> p -cat=Phrase "this wine is very fresh" | l

this wine is very fresh
```

Pipe with a trace

```
> p -cat=Phrase -tr "this wine is very fresh" | l

Is (This Wine) (Very Fresh)

this wine is very fresh
```

Random generation

```
> generate_random -cat=Phrase -tr | linearize

Is (This Cheese) (Very Warm)

this cheese is very warm
```

Refactoring the grammar

Two modules, two files: abstract and concrete syntax

Abstract: file `Food.gf`

```
abstract Food = {
  flags startcat = Phrase ;
  cat
    Phrase ; Item ; Kind ; Quality ;
  fun
    Is : Item -> Quality -> Phrase ;
    This, That : Kind -> Item ;
    QKind : Quality -> Kind -> Kind ;
    Wine, Cheese, Fish : Kind ;
    Very : Quality -> Quality ;
    Fresh, Warm, Italian, Expensive, Delicious, Boring : Quality ;
}
```

Refactoring the grammar: English concrete

Concrete: file `FoodEng.gf`

```
concrete FoodEng of Food = {
  lincat
    Phrase, Item, Kind, Quality = {s : Str} ;
  lin
    Is item quality = {s = item.s ++ "is" ++ quality.s} ;
    This kind = {s = "this" ++ kind.s} ;
    That kind = {s = "that" ++ kind.s} ;
    QKind quality kind = {s = quality.s ++ kind.s} ;
    Wine = {s = "wine"} ;
    Cheese = {s = "cheese"} ;
    Fish = {s = "fish"} ;
    Very quality = {s = "very" ++ quality.s} ;
    Fresh = {s = "fresh"} ;
    Warm = {s = "warm"} ;
    Italian = {s = "Italian"} ;
    Expensive = {s = "expensive"} ;
    Delicious = {s = "delicious"} ;
    Boring = {s = "boring"} ;
}
```

Notice: linearizations are records, not strings.

Modules and judgements

An `abstract` module contain judgements of forms

- `cat` C, "C is a category"
- `fun` $f : T$, "f is a function of type T"

A `concrete of` A module contain judgements of forms

- `lincat` $C = L$, "C has linearization type L"
- `lin` $f = t$, "f has linearization function t"

A concrete is complete w.r.t. its abstract, if it has

- a *lincat* for each *cat*
- a *lin* for each *fun*

Linearization

A **compositional mapping**, i.e. a **homomorphism**:

for each abstract function

$$f : A_1 \rightarrow \ldots \rightarrow A_n \rightarrow A$$

the linearization is a function between the corresponding linearization types,

$$f^* : A_1^* \rightarrow \ldots \rightarrow A_n^* \rightarrow A^*$$

The linearization of a tree is obtained from the liearizations of the subtrees,

$$(f a_1 \ldots a_n)^* = f^* a_1^* \ldots a_n^*$$

A simple-minded Hindi grammar

```
concrete FoodHin0 of Food = {
  lincat
    Phrase, Item, Kind, Quality = {s : Str} ;
  lin
    Is item quality = {s = item.s ++ quality.s ++ "he"} ;
    This kind = {s = "yah" ++ kind.s} ;
    That kind = {s = "vah" ++ kind.s} ;
    QKind quality kind = {s = quality.s ++ kind.s} ;
    Wine = {s = "madira:"} ;
    Cheese = {s = "pani:r"} ;
    Fish = {s = "maCli:"} ;
    Very quality = {s = "bahut" ++ quality.s} ;
    Fresh = {s = "ta:za:"} ;
    Warm = {s = "garam"} ;
    Italian = {s = "it.ali:"} ;
    Expensive = {s = "maha*ga:"} ;
    Delicious = {s = "rucikar"} ;
    Boring = {s = "pEriya:"} ;
}
```

Dealing with Devanagari

One can use UTF8 in GF.

But I have used a lossless ASCII translitaration, which

- works in all text editors and terminals
- creates no need to switch keyboards

Conversion to Devanagari in UTF8 in two possible places

- convert the string literals in the grammar file
- convert input and output of GF running

Translation and its problems

Translate from A to B = parse in A and linearize to B

```
> p -lang=FoodEng "this cheese is expensive" | l -lang=FoodHin0

yah pani:r maha*ga: he
```

This works! But our Hindi grammar is not very good:

```
> p -lang=FoodEng "this fish is expensive" | l -lang=FoodHin0

yah maCli: maha*ga: he
```

See the whole misery by creating 100 random sentences and displaying them in Devanagari:

```
gr -number=100 | l -tr -lang=FoodHin0 | ? ./deva
```

Some linguistics for Hindi

Nouns (like *vin*) have a **gender**:

- *pani:r* is masculine
- *maCli:* is feminine

Adjectives (like *maha*ga:* are **inflected** for gender:

- *maha*ga:* is *maha*gi:* in feminine

The choice of gender for adjectives: **agreement** with a noun.

Thus

> *yah pani:r maha*ga: he*
> *yah maCli: maha*gi: he*

Parameters and tables

In a concrete syntax (thus, language-specifically), one can define **parameter types**, by listing their **constructors**:

```
param Gender = Masc | Fem ;
```

Like algebraic data types in ML/Haskell, but **finite**.

Table = **finite function** = function over a parameter type.

```
table {
  Masc => "maha*ga:" ;
  Fem  => "maha*gi:"
  }
```

Selection from table *t* with a parameter value *v*:

```
table {
  Masc => "maha*ga:" ;
  Fem  => "maha*gi:"
  } ! Fem

= "mahan*gi:"
```

Linearization types

Nouns have a gender (and a `Kind` is a noun):

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

Adjectives are inflected for gender (and a `Quality` is an adjective):

```
lincat Quality = {s : Gender => Str}
```

Adjectives agree to the gender of the noun:

```
lin QKind quality kind = {
  s = quality.s ! kind.g ++ kind.s ;
  g = kind.g
  }
```

A golden rule

We don't want to write

```
lin
  Expensive = {
    s = table {
      Masc => "maha*ga:" ;
      Fem  => "maha*gi:"
      }
    } ;

  Fresh = {
    s = table {
      Masc => "ta:za:" ;
      Fem  => "ta:zi:"
      }
    } ;
```

The golden rule of functional programming:

Whenever you find yourself programming by copy and paste, write a function instead''

Operations

One more judgement form for concrete syntax:

- `oper` $h : T = t$, "h is an auxiliary operation of type T, defined as t

Now we can write

```
oper mkAdj : Str -> Str -> {s : Gender => Str} = \m,f -> {
    s = table {
      Masc => m ;
      Fem  => f
      }
    } ;

lin
  Fresh = mkAdj "ta:za:" "ta:zi:" ;
  Expensive = mkAdj "maha*ga:" "maha*gi:" ;
  Warm = regAdj "garam" "garam" ;
```

More operations

To capture the invariant adjective inflextion:

```
oper invAdj : Str -> {s : Gender => Str} = \a ->
  mkAdj a a ;
```

Thus we can write

```
lin Warm = invAdj "garam" ;
```

Soon we will be able to write

```
lin Fresh = regAdj "maha*ga:" ;
```

and even better (Morphology lecture).

Revised Hindi grammar

```
concrete FoodHin1 of Food = {

  param
    Gender = Masc | Fem ;
  lincat
    Phrase = {s : Str} ;
    Item, Kind = {s : Str ; g : Gender} ;
    Quality = {s : Gender => Str} ;
  lin
    Is item quality = {s = item.s ++ quality.s ! item.g ++ "he"} ;
    This kind = {s = "yah" ++ kind.s ; g = kind.g} ;
    That kind = {s = "vah" ++ kind.s ; g = kind.g} ;
    QKind quality kind = {s = quality.s ! kind.g ++ kind.s ; g = kind.g} ;
    Wine = {s = "madira:" ; g = Masc} ;
    Cheese = {s = "pani:r" ; g = Masc} ;
    Fish = {s = "maCli:" ; g = Fem} ;
    Very quality = {s = table {g => "bahut" ++ quality.s ! g}} ;
    Fresh = mkAdj "ta:za:" "ta:zi:" ;
    Warm = invAdj "garam" ;
    Italian = invAdj "it.ali:" ;
    Expensive = mkAdj "maha*ga:" "maha*gi:" ;
    Delicious = invAdj "rucikar" ;
    Boring = mkAdj "pEriya:" "pEriyi:" ;

  oper
    mkAdj : Str -> Str -> {s : Gender => Str} = \m,f -> {
      s = table {
        Masc => m ;
        Fem  => f
        }
      } ;

    invAdj : Str -> {s : Gender => Str} = \a -> mkAdj a a ;
}
```

Testing the new grammar

We linearize to all languages (`-multi`) - new Hindi, old Hindi, English:

```
> p -lang=FoodEng "this cheese is fresh" | l -multi

yah pani:r ta:za: he
yah pani:r ta:za: he
this cheese is fresh

> p -lang=FoodEng "this fish is fresh" | l -multi

yah maCli: ta:za: he
yah maCli: ta:zi: he
this fish is fresh
```

OK so far.

Extending the grammar

Let us add plural determiners:

```
abstract Foods = Food ** {
  fun
    These, Those : Kind -> Item ;
}
```

The operation `**` between modules expresses **extension**, a.k.a. **inheritance**.

English concrete syntax

```
concrete FoodsEng of Foods = FoodEng1 ** {
  lin
    These kind = {s = "these" ++ kind.s ! Pl ; n = Pl} ;
    Those kind = {s = "those" ++ kind.s ! Pl ; n = Pl} ;
  }
```

==Revised English grammar: add number==

```
concrete FoodEng1 of Food = {

  param
    Number = Sg | Pl ;

  lincat
    Phrase, Quality = {s : Str} ;
    Item = {s : Str ; n : Number} ;
    Kind = {s : Number => Str} ;
```

```
oper
  mkN : Str -> Str -> {s : Number => Str} = \man,men -> {
    s = table {
      Sg => man ;
      Pl => men
      }
    } ;

 regN : Str -> {s : Number => Str} = \car -> mkN car (car + "s") ;

 copula : Number -> Str = \n -> case n of {
   Sg => "is" ;
   Pl => "are"
   } ;

lin
  Is item quality = {s = item.s ++ copula item.n ++ quality.s} ;
  This kind = {s = "this" ++ kind.s ! Sg ; n = Sg} ;
  That kind = {s = "that" ++ kind.s ! Sg ; n = Sg} ;
```

```
    QKind quality kind = {s = \\n => quality.s ++ kind.s ! n} ;
    Wine = regN "wine" ;
    Cheese = regN "cheese" ;
    Fish = mkN "fish" "fish" ;
    Very quality = {s = "very" ++ quality.s} ;
    Fresh = {s = "fresh"} ;
    Warm = {s = "warm"} ;
    Italian = {s = "Italian"} ;
    Expensive = {s = "expensive"} ;
    Delicious = {s = "delicious"} ;
    Boring = {s = "boring"} ;
}
```

Revised Hindi

```
concrete FoodHin2 of Food = {

  param
    Gender = Masc | Fem ;
    Number = Sg | Pl ;
  lincat
    Phrase  = {s : Str} ;
    Item    = {s : Str ; g : Gender ; n : Number} ;
    Kind    = {s : Number => Str ; g : Gender} ;
    Quality = {s : Gender => Number => Str} ;
  lin
    Is item quality = {
      s = item.s ++ quality.s ! item.g ! item.n ++ copula item.n
      } ;
    This kind = {s = "yah" ++ kind.s ! Sg ; g = kind.g ; n = Sg} ;
    That kind = {s = "is" ++ kind.s ! Sg ; g = kind.g ; n = Sg} ;
    QKind quality kind = {
      s = \\n => quality.s ! kind.g ! n ++ kind.s ! n ;
      g = kind.g
      } ;
    Wine = regN "madira:" ;
    Cheese = regN "pani:r" ;
    Fish = regN "maCli:" ;
    Very quality = {s = \\g,n => "bahut" ++ quality.s ! g ! n} ;
    Fresh = regAdj "ta:za:" ;
    Warm = regAdj "garam" ;
    Italian = regAdj "it.ali:" ;
    Expensive = regAdj "maha*ga:" ;
    Delicious = regAdj "rucikar" ;
    Boring = regAdj "pEriya:" ;

  oper
    mkN : Str -> Str -> Gender -> {s : Number => Str ; g : Gender} =
      \s,p,g -> {
        s = table {
          Sg => s ;
          Pl => p
          } ;
        g = g
      } ;

    regN : Str -> {s : Number => Str ; g : Gender} = \s -> case s of {
      lark + "a:" => mkN s (lark + "e") Masc ;
      lark + "i:" => mkN s (lark + "iya~") Fem ;
      _           => mkN s s Masc
      } ;

    mkAdj : Str -> Str -> Str -> {s : Gender => Number => Str} = \ms,mp,f -> {
      s = table {
        Masc => table {
          Sg => ms ;
          Pl => mp
          } ;
        Fem  => \\_ => f
        }
      } ;
```

```
    regAdj : Str -> {s : Gender => Number => Str} = \a -> case a of {
      acch + "a:" => mkAdj a (acch + "e") (acch + "i:") ;
      _           => mkAdj a a a
      } ;

    copula : Number -> Str = \n -> case n of {
      Sg => "hE" ;
      Pl => "hE*"
      } ;
}
```

Hindi extended

```
concrete FoodsHin of Foods = FoodHin2 ** {
  lin
    These kind = {s = "ye" ++ kind.s ! Pl ; g = kind.g ; n = Pl} ;
    Those kind = {s = "ve" ++ kind.s ! Pl ; g = kind.g ; n = Pl} ;
  }
```

The GF Resource Grammar Library

Hiding linguistic details

The `Food` example: very simple language leads to linguistic complexities.

They can be tackled nicely in GF - but they require a lot of knowledge.

Solution: library-based software engineering.

The GF Resource Grammar Library implements the morphology and basic syntax of 11 languages (and it is growing): Bulgarian, Danish, English, Finnish, French, German, Italian, Norwegian, Russian, Spanish, Swedish.

These details are hidden behind a high-level API.

API of categories needed for the Food grammar

Category	**Explanation**	**Example**
`Cl`	clause (sentence), with all tenses	*she looks at this*
`AP`	adjectival phrase	*very warm*
`CN`	common noun (without determiner)	*red house*
`NP`	noun phrase (subject or object)	*the red house*
`AdA`	adjective-modifying adverb,	*very*
`QuantSg`	singular quantifier	*this*
`A`	one-place adjective	*warm*
`N`	common noun	*house*

API of functions needed for the Food grammar

Function	**Type**	**Example**
`mkCl`	`NP -> AP -> Cl`	*John is very old*
`mkNP`	`QuantSg -> CN -> NP`	*this old man*
`mkCN`	`N -> CN`	*house*
`mkCN`	`AP -> CN -> CN`	*very big blue house*
`mkAP`	`A -> AP`	*old*
`mkAP`	`AdA -> AP -> AP`	*very very old*

Whenever possible, the functions are **overloaded**: their name is `mk`*C*, if the value type is *C*.

API of structural words needed for the Food grammar

Function	**Type**	**In English**
`this_QuantSg`	`QuantSg`	*this*
`that_QuantSg`	`QuantSg`	*that*
`very_AdA`	`AdA`	*very*

Overloading does not work, because there are many functions of the same type.

Notice: everything so far is implemented for all 11 languages.

API of inflection paradigms needed for the Swedish Food grammar

Function	Type	Examples
`mkN`	`(apa : Str) -> N`	*apa*, *bil*
`mkN`	`(man,mannen,män,männen : Str) -> N`	*man*, *mus*
`mkA`	`(fin : Str) -> A`	*fin*,*fager*

Overloading does not work, because there are many functions of the same type.

Notice: everything so far is implemented for all 11 languages.

Food grammar written by using the API

```
concrete FoodSwe3 of Food = open SyntaxSwe, ParadigmsSwe in {
  lincat
    Phrase = Cl ;
    Item = NP ;
    Kind = CN ;
    Quality = AP ;
  lin
    Is item quality = mkCl item quality ;
    This kind = mkNP this_QuantSg kind ;
    That kind = mkNP that_QuantSg kind ;
    QKind quality kind = mkCN quality kind ;
    Wine = mkCN (mkN "vin" "vinet" "viner" "vinerna") ;
    Cheese = mkCN (mkN "ost") ;
    Fish = mkCN (mkN "fisk") ;
    Very quality = mkAP very_AdA quality ;
    Fresh = mkAP (mkA "färsk") ;
    Warm = mkAP (mkA "varm") ;
    Italian = mkAP (mkA "italiensk") ;
    Expensive = mkAP (mkA "dyr") ;
    Delicious = mkAP (mkA "läcker") ;
    Boring = mkAP (mkA "tråkig") ;
}
```

Advantages of using the library

Less code to write

Less knowledge needed

Guaranteed correctness

Simpler GF code (no tables and records by the user)

Portability to other supported languages

Port to a new language: just change content words

```
concrete FoodFin of Food = open SyntaxFin, ParadigmsFin in {
  lincat
    Phrase = Cl ;
    Item = NP ;
    Kind = CN ;
    Quality = AP ;
  lin
    Is item quality = mkCl item quality ;
    This kind = mkNP this_QuantSg kind ;
    That kind = mkNP that_QuantSg kind ;
    QKind quality kind = mkCN quality kind ;
    Wine = mkCN (mkN "viini") ;
    Cheese = mkCN (mkN "juusto") ;
    Fish = mkCN (mkN "kala") ;
    Very quality = mkAP very_AdA quality ;
    Fresh = mkAP (mkA "tuore") ;
    Warm = mkAP (mkA "lämmin") ;
    Italian = mkAP (mkA "italialainen") ;
    Expensive = mkAP (mkA "kallis") ;
    Delicious = mkAP (mkA "herkullinen") ;
    Boring = mkAP (mkA "tylsä") ;
}
```

Avoiding copy and paste: functors

The resource API, module `Syntax` is an **interface**

- just declarations of types and functions
- definitions are given in **instance** modules

A **functor**, a.k.a. **parametrized** or **incomplete** module, is one that uses an interface.

```
incomplete concrete FoodI of Food = open Syntax in {
  lincat
    Phrase = Cl ;
    Item = NP ;
  lin
    Is item quality = mkCl item quality ;
    This kind = mkNP this_QuantSg kind ;
  ...}
```

An **instantiation** tells what instances are given to interfaces:

```
  concrete FoodGer of Food = FoodI with
    (Syntax = SyntaxGer) ;
```

Hmm... where do the content words come from?

A design pattern for multilingual grammars

Domain semantics:

```
abstract Food = ...
```

Domain lexicon interface:

```
interface LexFood = ...
```

Concrete syntax functor:

```
incomplete concrete FoodI of Food = open Syntax, LexFood in ...
```

Domain lexicon instance:

```
instance LexFoodGer of LexFood = open ParadigmsGer in ...
```

Concrete syntax instantiation:

```
concrete FoodGer of Food = FoodI with
  (Syntax = SyntaxGer),
  (LexFood = LexFoodGer) ;
```

A domain lexicon example

```
interface LexFoods = open Syntax in {
  oper
    wine_N, cheese_N : N, fish_N : N ;
    fresh_A : A, warm_A : A, italian_A : A ;
    expensive_A, delicious_A, boring_A : A ;
}

instance LexFoodsGer of LexFoods = open SyntaxGer, ParadigmsGer in {
  oper
    wine_N = mkN "Wein" ;
    cheese_N = mkN "Käse" "Käsen" masculine ;
    fish_N = mkN "Fisch" ;
    fresh_A = mkA "frisch" ;
    warm_A = mkA "warm" "wärmer" "wärmste" ;
    italian_A = mkA "italienisch" ;
    expensive_A = mkA "teuer" ;
    delicious_A = mkA "köstlich" ;
    boring_A = mkA "langweilig" ;
}
```

Morphology in GF

Smart paradigms

Paradigms for Hindi

See GF tutorial (on-line) for smart paradigms.

See the last lecture (Resource Grammars) for details on Hindi.

Translation in GF

Plan

Domain-specific translation.

Example: mathematics.

An idea for GF+UNL.

Basic idea: abstract syntax as interlingua

Abstract syntax: domain semantics

Concrete syntax: domain vocabulary and idiom

Translation is semantics-preserving by definition

Advantage: much of the context is fixed

Limitation: unlimited text has no domain semantics

Early example from XRCE Grenoble

Alarm system instruction:

If the board is not empty, just leave the premises.

The Systran translation (on-line in `www.systran.co.uk`) of this into French is

Si le conseil n'est pas vide, laissez juste les lieux.

The word *boad* is translated wrongly, giving it the meaning "counsil". The German translation

Wenn das Brett nicht leer ist, lassen Sie einfach die Voraussetzungen.

renders *premises* with a word used for the premises of a logical inference.

The example in GF

Domain grammar translations:

> *Si le table n'est pas vide, juste quittez les lieux.*
>
> *Wenn die Tafel nicht leer ist, verlassen Sie die Räume nur.*

Semantic analysis that was accurate enough

- for correct translation
- for formal verification of the instructions: to prove that, if the instructions are followed, the alarm is on if and only if the board is empty, if and only if there is no-one in the building.

The price to pay: the system was very restricted:

```
please leave the premises

Unknown word "please", no tree found.
```

Examples of translation domains

GF-KeY (2001–2005)

- software specifications in OCL, English, German
- semantic control using dependent types

WebALT (2005–2006)

- mathematical exercises from MathML to seven languages
- continues in WebALT Inc., Helsinki and Barcelona

TALK (2004–2006)

- spoken dialogue systems in six languages
- route finding, MP3 players in cars, agenda

More examples of translation domains

Demonat 2 (2007–)

- mathematical proofs in the PhoX system
- thesis project of Muhammad Humayoun at Chambéry, France

MedSLT+ (2008?)

- doctor-patient spoken diagnosis questions
- builds on MedSLT built in Regulus (Manny Rayner)
- interest from IITB?

None of the above is really A-to-B translation...

Example: mathematics translation

Background: formal mathematics in LF (Logical Framework)

GF = LF + concrete syntax

LF can encode any logic and mathematical theory

- focus on machine-assisted proofs
- e.g. four-colour theorem at INRIA and Microsoft Research

Example math exercise

In English, French, and Finnish:

Let x be a natural number. Assume that x is prime. Prove that the factorial of x is even .

Soit x un nombre entier. Supposons que x est premier. Démontrer que le factoriel de x est pair.

Olkoon x luonnollinen luku. Oletetaan että x on alkuluku. Osoita että x:n kertoma on parillinen.

Mapping semantics to syntax

Assumption:

- English: 2nd person imperative: *Assume that...*
- French: 1st person plural imperative: *Supposons que...*
- Finnish: passive: *Oletetaan että...*

The primeness predicate

- English and French: adjective: *prime*, *premier*
- Finnish: noun: *alkuluku*

Abstract syntax

```
abstract Math = {
  cat
    Exercise ; Given ; Sought ;
    Prop ; Typ ; Object ; Symbol ;
  fun
    ToDo    : Given -> Sought -> Exercise ;
    NoGiven : Given ;
    Declar  : Symbol -> Typ -> Given -> Given ;
    Assumpt : Prop -> Given -> Given ;
    Prove   : Prop -> Sought ;
    Compute : Object -> Sought ;
    Symbol  : Symbol -> Object ;
}
```

Subdomain abstract syntax

Added by a maths teacher to the base grammar:

```
abstract Arithm = Math ** {
  fun
    Nat   : Typ ;
    Fac   : Object -> Object ;
    Sum   : Object -> Object -> Object ;
    Prime : Object -> Prop ;
    Even  : Object -> Prop ;
}
```

Wanted: an easy way to define the concrete syntax.

Abstract syntax of the example

Let x be a natural number. Assume that x is prime. Prove that the factorial of x is even .

```
  ToDo
    (Declar x_Symb Nat (Assumpt (Prime (Symb x_Symb)) NoGiven))
    (Prove (Even (Fac (Symb x_Symb))))
```

Olkoon x luonnollinen luku. Oletetaan että x on alkuluku. Osoita että x:n kertoma on parillinen.

Two ways to define concrete syntax

"String-based", from scratch

- full freedom and control
- a lot of work
- difficult to get right

Library-based, using the resource API

- grammaticality guaranteed
- sharing of effort (functors)
- restricted to library operations

A string-based concrete syntax

```
concrete MathEng of Math = {
  lincat
    Exercise, Given, Sought,
    Prop, Typ, Object, Symbol = {s : Str} ;
  lin
    ToDo given sought = {s = given.s ++ sought.s} ;
    NoGiven = {s = []} ;
    Declar sym typ cont = {
      s = "Let" ++ sym.s ++ "be" ++ "a" ++ typ.s ++ "." ++ cont.s
      } ;
    Assumpt prop cont = {
      s = "Assume" ++ "that" ++ prop.s ++ "." ++ cont.s
      } ;
    Prove prop = {s = "Prove" ++ "that" ++ prop.s ++ "."} ;
    Compute obj = {s = "Compute" ++ obj.s ++ "."} ;
}
```

A library-based concrete syntax functor

```
incomplete concrete MathI of Math = open Syntax, LexMath in {
  lincat
    Exercise = Text ;
    Given = Text ;
    Sought = Text ;
    Prop = S ;
    Typ = CN ;
    Object = NP ;
    Symbol = PN ;
  lin
    ToDo given sought = mkText given sought ;
    NoGiven = emptyText ;
    Declar sym typ cont = mkText (declar symb (mkVP indefNP typ)) cont ;
    Assumpt prop cont = mkText (assumpt prop) cont ;
    Prove prop = mkText (prove prop) ;
    Compute obj = mkText (compute obj) ;
    Symb sym = mkNP sym ;
```

The mathematics lexicon interface

```
interface LexMath = open Syntax in {
  oper
   declar  : PN -> VP -> Phr ;
   assumpt : S -> Phr ;
   prove   : S -> Phr ;
   compute : NP -> Phr ;
}
```

Examples of instances

```
  assumpt s = mkPhr (mkImp assume_VS Phr)        -- English
  assumpt s = mkPhr (plurP1Imp supposer_VS Phr) -- French
  assumpt s = mkPhr (passCl olettaa_VS Phr)     -- Finnish
```

Transfer: map interlingua to different syntactic structures.

This kind of transfer in GF happens at compile time.

Domain lexicon implementations

```
concrete ArithmEng of Arithm = MathEng **
  open ParadigmsEng, SyntaxEng, PredicationEng in {
  lin
    Nat     = mkCN (mkA "natural") (mkN "number) ;
    Fac x   = app  (mkN2 "factorial") x ;
    Sum x y = app  (mkN "sum") x y ;
    Prime x = pred (mkA "prime") x ;
    Even  x = pred (mkA "even") x ;
}
```

The predication API

```
pred : V  -> NP -> Cl         -- 1-place verb:   "x converges"
pred : V2 -> NP -> NP -> Cl   -- 2-place verb:   "x intersects y"
pred : A  -> NP -> Cl         -- 1-place adje:   "x is even"
pred : A2 -> NP -> NP -> Cl   -- 2-place adj:    "x is divisible by y"
pred : A  -> NP -> NP -> Cl   -- collective adj: "x and y are parallel"
pred : N  -> NP -> Cl         -- 1-place noun:   "x is a point"

app  : N2 -> NP -> NP         -- 1-place funct:  "the successor of x"
app  : N3 -> NP -> NP -> NP   -- 2-place funct:  "the distance from x to y"
app  : N2 -> NP -> NP -> NP   -- collect. funct: "the sum of x and y"
```

Partial application

One can write `f = t` instead of `f x = t x`. Thus

```
  lin
    Fac   = app  (mkN2 "factorial") ;
    Sum   = app  (mkN "sum") ;
    Prime = pred (mkA "prime") ;
    Even  = pred (mkA "even") ;
```

In Finnish,

```
    Prime = pred (mkN "alkuluku") ;
```

Conclusions on domain translation

Interlingua: abstract syntax encoding domain semantics.

Linguistic details from resource grammar library.

Functor + domain lexicon + compile-time transfer.

Base grammar + user-extensible lexicon.

Demo: WebALT problem editor

Running on top of computer algebra.

Employs Embedded GF in Java.

Uses incremental parsing of GF grammars.

Demo on the web: `http://webalt.math.helsinki.fi/PublicFiles/CD/Screencast/`

The MedSLT project

Based on Regulus and an interlingua.

Much of the focus is on disambiguation, and helping the doctor to formulate questions.

The interlingua is "one of the languages" in Regulus.

Mature system: GF will be used to build new components (e.g. new target language generation).

Communication between Regulus and GF: the Regulus interlingua is a concrete syntax in GF.

An idea for GF+UNL

Write a resource grammar implementation for UNL!

Then one can translate between UNL and any grammars in the resource library.

Example:

```
fun PredVP : NP -> VP -> Cl ;

lin PredVP np vp = {
  s = \\t =>
      "[S]" ++
      "agt" ++ "(" ++ vp.s ++ ".@entry" ++ tense t ++ np.s ++ ")" ++
      "[/S]"
  }
```

The GF Resource Grammar Library

Plan

Aims, scope, size, coverage

The implementation effort

A summary of structures

Implementation: Hindi

Scaling up to full coverage

Aims, scope, size, coverage

Aims

To provide a standard library for GF applications

To cover the linguistic details of languages

- and present them in a high-level API

Criteria:

- grammatical correctness
- semantic/expressive completeness

Non-criteria:

- completeness as for language corpora (which is a secondary aim)
- semantics of abstract syntax

Linguistic ontology

Ontology: grammatical categories and structures, as known from the venerable tradition of linguistics

- nouns, verbs, noun phrases, sentences, ...
- predication, complementization, modification, ...

Not a domain ontology

No guarantee of meaning preservation across languages

The nucleus of resource syntax

```
cat
  Cl ; NP ; VP ; CN ; AP ; Det ; N ; A ; V ; V2 ;
fun
  PredVP  : NP  -> VP -> Cl ;
  ComplV2 : V2  -> NP -> VP ;
  ComplAP : AP  -> VP ;
  DetCN   : Det -> CN -> NP ;
  ModCN   : AP  -> CN -> CN ;
  UseN    : N   -> CN ;
  UseA    : A   -> AP ;
  UseV    : V   -> VP ;
```

The complete size

Syntax

- benchmark: CLE (Core Language Engine, Rayner 2000)
- ca. 40 categories, 190 functions
- expands to 50,000 or more context-free rules (approximation)
- language-specific extra categories and functions

Morphology

- complete feature sets of nouns, adjectives, verbs
- complete sets of inflection paradigms

Lexicon

- ca. 100 structural words (incl. numerals)
- test lexicon of ca. 350 content words (incl. Swadesh)
- some languages have irregular verbs, plus larger lexica

Presentation

Ground grammar: fun/lin rules, minimal, non-redundant

```
PredVP     : NP -> VP -> Cl
SlashV2    : V2 -> VPSlash
ComplSlash : VPSlash -> NP -> VP
```

User API: overloaded operations, shortcuts, redundancies

```
mkCl       : NP -> VP -> Cl
mkVPSlash  : V2 -> VPSlash
mkVP       : VPSlash -> NP -> VP
mkVP       : V2 -> NP -> VP       -- ComplSlash (SlashV2 v2) np
mkCl       : NP -> V2 -> NP -> Cl -- PredVP np (ComplSlash (SlashV2 v2) np')
```

Advantages of the two levels

Ground grammar

- minimal number of rules for the linguist to implement
- usable for parsing without (so much) ambiguity

User API

- easy names to remember
- bypasses theoretical categories (e.g. Slash, VP)
- data abstaction: linguists can change the ground grammar without destroying backward compatibility

The morphology API

Language-dependent: `Paradigms`*L*

Smart paradigms (cf. Tuesday's lecture)

One-argument `mkN`, `mkA`,`mkV` for each language

Worst-case functions (preferably) made unnecessary via an irregularity lexicon

Languages covered

Complete API: Bulgarian, Danish, English, Finnish, French, German, Italian, Norwegian, Russian, Spanish, Swedish; Interlingua.

Close to complete: Catalan, Arabic

Started: Chinese, Hindi/Urdu, Swahili, Thai

Projected: the remaining 14 (= 23–9) official EU languages

Desired: the remaining 5,975 (= 6,000–25) world's languages

The implementation effort

Work needed so far

Start 2002, Version 1.0 in 2006

Ca. 3 person years

4,000 LoC/language

3–9 months/language

Less for new members of language families using functors (Romance share 80% of syntax, Scandinavian 90%)

Level and prerequisites

Advanced MSc or part of PhD; fun for a postdoc or a professor

Knowledge

- GF (exercises from the Tutorial)
- general programming skills
- linguistic concepts (at least advanced traditional grammar)
- the target language (need not be native)
  Manual: Resource Grammar HOWTO

Reward

Often publishable in specialist worshops:

- Arabic in a book, *Perspectives on Arabic Linguistics XX* (El Dada and Ranta)
- Urdu in a workshop on languages using Arabic script (Humayoun and Hammarström)
- Russian in ACL/Coling 2006 (Khegai)
- Bulgarian submitted (Angelov)

Make old applications portable to the new language

Gain insights on the language, and on relations of languages

An overview of structures

Module system

See Resource Grammar HOWTO

Categories and functions

See Resource Grammar Synopsis

Implementation: Hindi

How to get started

Acquired the latest GF sources from the Darcs repository

Created a directory for the ground resource files

```
  GF/lib/resource-1.4/hindi
```

Copied English files and renamed `Eng` to `Hin` (ISO-639 language code)

Commented out contents, except module headers

The grammar now compiles, but is almost empty:

```
  gf LangHin.gf
```

How to proceed

Bottom-up:

1. Lexical categories `N`, `A`, `V`: parameters, linearization types, main inflection
2. Some entries in `LexiconHin`, to be able to test
3. Categories `NP`, `VP`, and `Cl`
4. Predication, determination: test *the boy sleeps*
5. `AP` and adjectival predication: test *the boy is good*
6. `V2` and complementation: test *the boy eats an apple*
7. `V2` and complementation: test *the boy eats an apple*
8. Adjectival modification: test *the good boy eats a big apple*

Nouns: the linearization type

In `ResHin.gf`

```
  param
    Case = Dir | Obl | Voc ;
    Gender = Masc | Fem ;

  oper
    Noun = {s : Number => Case => Str ; g : Gender} ;
```

In `CatHin.gf`

```
  lincat N = Noun ;
```

Nouns: inflection

In `ResHin.gf`: worst-case and some more regular

```
mkNoun : (x1,_,_,_,_,x6 : Str) -> Gender -> Noun =
  \sd,so,sv,pd,po,pv,g -> {
  s = table Number [table Case [sd;so;sv] ; table Case [pd;po;pv]] ;
  g = g
  } ;

regNoun : Str -> Noun = \s -> case s of {
  x + "iya:" =>
    mkNoun s s       s   (x + "iya:~") (x + "iyo*") (x + "iyo") Fem ;
  x + "a:"   =>
    mkNoun s (x + "e") (x + "e") (x + "e") (x + "o*") (x + "o") Masc ;
  x + "i:"   =>
    mkNoun s s       s   (x + "iya:~") (x + "iyo*") (x + "iyo") Fem ;
  _   =>
    mkNoun s s       s   s             (s + "o*")   (s + "o")   Masc
  } ;
```

```
reggNoun : Str -> Gender -> Noun = \s,g -> case <s,g> of {
  <-(_ + ("a:" | "i:")), Fem> =>
       mkNoun s s s (s + "e~") (s + "o~") (s + "o") Fem ;
  _ => regNoun s ** {g = g}
  } ;
```

Nouns: paradigm API

In `ParadigmsHin.gf`: start from regular, hide definitions

```
  mkN : overload {
    mkN : Str -> N ;
    ---  = \s -> regNoun s ** {lock_N = <>} ;
    mkN : Str -> Gender -> N
    mkN : (x1,_,_,_,_,x6 : Str) -> Gender -> N ;
    } ;
```

Entries in Lexicon

In `LexiconHin`:

```
apple_N = mkN "seb" ;
boy_N   = mkN "laRka:" ;
bread_N = mkN "roTi:" ;
girl_N = mkN "laRki:" ;
```

Test inflection

```
> i LangHin.gf
> l -all -tr boy_N | ? 'runghc hdeva.hs'
> l -all -tr girl_N | ? 'runghc hdeva.hs'
```

Adjectives: the linearization type

In `ResHin.gf`

```
  oper Adjective = {s : Gender => Number => Case => Str} ;
```

In `CatHin.gf`

```
  lincat A = Adjective ;
```

Adjectives: inflection

In `ResHin.gf`: worst-case and some more regular

```
mkAdjective : (x1,x2,x3 : Str) -> Adjective = \smd,sm,f -> {
  s = \\g,n,c => case <g,n,c> of {
    <Masc,Sg,Dir> => smd ;
    <Masc>        => sm ;
    _             => f
    }
  } ;

regAdjective : Str -> Adjective = \s -> case s of {
  acch + "a:" => mkAdjective s (acch + "e") (acch + "i:") ;
  _ => mkAdjective s s s
  } ;
```

Adjectives: paradigm API

In `ParadigmsHin.gf`: start from regular

```
  mkA : overload {
    mkA : Str -> A ;
    mkA : (x1,_,x3 : Str) -> A ;
    } ;
```

Adjective entries in Lexicon

In `LexiconHin`:

```
big_A = mkA "baRa:" ;
good_A = mkA "acCa:" ;
red_A = mkA "la:l" ;
```

Test inflection

```
> i LangHin.gf
> l -all -tr good_A | ? 'runghc hdeva.hs'
> l -all -tr red_A | ? 'runghc hdeva.hs'
```

Verbs: the linearization type

In `ResHin.gf`

```
param
  VForm =
     VInf
   | VStem
   | VImpf Gender Number
   | VPerf Gender Number
   | VSubj Number Person
   | VFut  Number Person Gender
   | VAbs
   | VReq
   | VImp
   | VReqFut
   ;

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

In `CatHin.gf`

```
lincat V = Verb ;
```

Verbs: inflection

In `ResHin.gf`: worst-case and regular

```
mkVerb : (x1,_,_,_,_,_,_,_,_,_,_,_,_,_,x15 : Str) -> Verb =
  \inf,stem,ims,imp,ifs,ifp,pms,pmp,pfs,pfp,ss1,ss2,sp2,sp3,r -> {
    s =
    let ga : Number -> Gender -> Str =
        \n,g -> (regAdjective "ga:").s ! g ! n ! Dir
    in table {
      VInf => inf ;
      VStem => stem ;
      VImpf Masc Sg => ims ;
      VImpf Masc Pl => imp ;
      VImpf Fem  Sg => ifs ;
      VImpf Fem  Pl => ifp ;
      VPerf Masc Sg => pms ;
      VPerf Masc Pl => pmp ;
      VPerf Fem  Sg => pfs ;
      VPerf Fem  Pl => pfp ;
      VSubj Sg   P1 => ss1 ;
      VSubj Sg   _  => ss2 ;
      VSubj Pl   P2 => sp2 ;
      VSubj Pl   _  => sp3 ;
      VFut  Sg   P1 g => ss1 + ga Sg g ;
      VFut  Sg   _  g => ss2 + ga Sg g ;
      VFut  Pl   P2 g => sp2 + ga Pl g ;
      VFut  Pl   _  g => sp3 + ga Pl g ;
      VAbs  => stem + "kar" ; --- ke
      VReq  => r ;
      VImp  => sp2 ;
      VReqFut => stem + "i-ega:"
      }
    } ;

regVerb : Str -> Verb = \cal ->
  let caly : Str = case cal of {
    _ + ("a:" | "e") => cal + "y" ;
    c + "u:" => c + "uy" ;
    c + "i:" => c + "iy" ;
  _ => cal
  }
in
mkVerb
  (cal + "na:") cal
  (cal + "ta:") (cal + "te") (cal + "ti:") (cal + "ti:")
  (caly + "a:")  (caly + "e")  (caly + "i:")  (caly + "i:*")
  (caly + "u:~") (caly + "e")  (caly + "o")   (caly + "e*")
  (caly + "i-e") ;
```

Verbs: paradigm API

In `ParadigmsHin.gf`: start from regular

```
  mkV = overload {
    mkV : Str -> V ;
    mkV : (x1,_,_,_,_,_,_,_,_,_,_,_,_,_,x15 : Str) -> V ;
    } ;
```

Verb entries in Lexicon

In `LexiconHin`:

```
go_V = mkV "cal" ;
eat_V2 = mkV2 "Ka:" ;
hit_V2 = mkV2 (mkV "ma:r") "ko" ;
```

Test inflection

```
> i LangHin.gf
> l -all -tr good_A | ? 'runghc hdeva.hs'
> l -all -tr red_A | ? 'runghc hdeva.hs'
```

Noun phrases

In `ResHin.gf`:

```
  param
    Agr = Ag Gender Number Person ;
    NPCase = NPC Case | NPErg ;

  oper
    NP : Type = {s : NPCase => Str ; a : Agr} ;

    agrP3 : Gender -> Number -> Agr = \g,n -> Ag g n P3 ;
    defaultAgr : Agr = agrP3 Masc Sg ;

    toNP : (Case => Str) -> NPCase -> Str = \pn, npc -> case npc of {
      NPC c => pn ! c ;
      NPErg => pn ! Obl ++ "ne"
      } ;
```

In `CatHin.gf`:

```
lincat NP = ResHin.NP ;
lincat CN = Noun ;
```

Determination with articles

In `NounHin.gf`

```
DetArtSg art cn = {
  s = \\c => art.s ++ toNP (cn.s ! Sg) c ;
  a = agrP3 cn.g Sg
  } ;

DetArtPl art cn = {
  s = \\c => art.s ++ toNP (cn.s ! Pl) c ;
  a = agrP3 cn.g Pl
  } ;

DefArt = {s = []} ;
IndefArt = {s = []} ;

UseN n = n ;
```

Test determination

```
> i LangHin.gf
> l -all -tr (DetArtSg DefArt boy_N) | ? 'runghc hdeva.hs'
> l -all -tr (DetArtPl IndefArt girl_N) | ? 'runghc hdeva.hs'
```

Clauses

In `ResHin.gf`:

```
param
  VPHTense =
     VPGenPres  -- impf hum       nahim    "I go"
   | VPImpPast  -- impf Ta        nahim    "I went"
   | VPContPres -- stem raha hum  nahim    "I am going"
   | VPContPast -- stem raha Ta   nahim    "I was going"
   | VPPerf     -- perf           na/nahim "I went"
   | VPPerfPres -- perf hum       na/nahim "I have gone"
   | VPPerfPast -- perf Ta        na/nahim "I had gone"
   | VPSubj     -- subj           na       "I may go"
   | VPFut      -- fut            na/nahim "I shall go"
   ;

oper Clause : Type = {s : VPHTense => Bool => Str} ;
```

In `CatHin.gf`:

```
lincat Cl = ResHin.Clause ;
```

Verb phrases

In `ResHin.gf`:

```
param
  VPHForm =
     VPTense VPHTense Agr -- 9 * 12
    | VPReq
    | VPImp
    | VPReqFut
    | VPInf
    | VPStem
    ;

  VType = VIntrans | VTrans | VTransPost ;
```

The Hindi VP type, cont'd

```
  oper
    VPH : Type = {
      s    : Bool => VPHForm => {fin, inf, neg : Str} ;
      obj  : {s : Str ; a : Agr} ;
      subj : VType ;
      comp : Agr => Str
      } ;
```

In `CatHin.gf`:

```
  lincat VP = ResHin.VPH ;
```

The copula

In `ResHin.gf`:

```
param
  CTense = CPresent | CPast | CFuture ;
oper
  copula : CTense -> Number -> Person -> Gender -> Str = \t,n,p,g ->
    case <t,n,p,g> of {
      <CPresent,Sg,P1,_   > => "hu:~" ;
      <CPresent,Sg,P2,_   > => "hE" ;
      <CPresent,Sg,P3,_   > => "hE" ;
      <CPresent,Pl,P1,_   > => "hE*" ;
      <CPresent,Pl,P2,_   > => "ho" ;
      <CPresent,Pl,P3,_   > => "hE*" ;
      <CPast,   Sg,_ ,Masc> => "Ta:" ;
      <CPast,   Sg,_ ,Fem > => "Ti:" ;
      <CPast,   Pl,_ ,Masc> => "Te" ;
      <CPast,   Pl,_ ,Fem > => "Ti:*" ;
      <CFuture, Sg,P1,Masc> => "hu:*ga:" ;
      <CFuture, Sg,P1,Fem > => "hu:*gi:" ;
      <CFuture, Sg,_ ,Masc> => "hoga:" ;
      <CFuture, Sg,_ ,Fem > => "hogi:" ;
      <CFuture, Pl,P2,Masc> => "hoge" ;
      <CFuture, Pl,_ ,Masc> => "ho*ge" ;
      <CFuture, Pl,P2,Fem > => "hogi:" ;
      <CFuture, Pl,_ ,Fem > => "ho*gi:"
      } ;
```

The simplest verb phrase

In `ResHin.gf`:

```
oper
  predV : Verb -> VPH = \verb -> {
    s = \\b,vh =>
     let
       na = if_then_Str b [] "na" ;
       nahim = if_then_Str b [] "nahi:*" ;
     in
     case vh of {
       VPTense VPGenPres (Ag g n p) =>
         {fin = copula CPresent n p g ; inf = verb.s ! VImpf g n ;
          neg = nahim} ;
       VPTense VPImpPast (Ag g n p) =>
         {fin = copula CPast n p g ; inf = verb.s ! VImpf g n ;
          neg = nahim} ;
       VPTense VPContPres (Ag g n p) =>
         {fin = copula CPresent n p g ;
```

```
      inf = verb.s ! VStem ++ raha g n ; neg = nahim} ;
    VPTense VPContPast (Ag g n p) =>
     {fin = copula CPast n p g ;
      inf = verb.s ! VStem ++ raha g n ; neg = nahim} ;
    VPTense VPPerf (Ag g n _) =>
     {fin = verb.s ! VPerf g n ; inf = [] ; neg = nahim} ;
    VPTense VPPerfPres (Ag g n p) =>
     {fin = copula CPresent n p g ; inf = verb.s ! VPerf g n ;
      neg = nahim} ;
    VPTense VPPerfPast (Ag g n p) =>
     {fin = copula CPast n p g ; inf = verb.s ! VPerf g n ;
      neg = nahim} ;
    VPTense VPSubj (Ag _ n p) =>
     {fin = verb.s ! VSubj n p ; inf = [] ; neg = na} ;
    VPTense VPFut (Ag g n p) =>
     {fin = verb.s ! VFut n p g ; inf = [] ; neg = na} ;
    VPInf => {fin = verb.s ! VStem ; inf = [] ; neg = na} ;
    _ => {fin = verb.s ! VStem ; inf = [] ; neg = na} ----
    } ;
 obj = {s = [] ; a = defaultAgr} ;
```

```
      subj = VIntrans ;
      comp = \\_ => []
      } ;

    raha : Gender -> Number -> Str = \g,n ->
      (regAdjective "raha:").s ! g ! n ! Dir ;
```

In `VerbHin.gf`:

```
  lin UseV = predV ;
```

Predication

In `ResHin.gf`:

```
    mkClause : NP -> VPH -> Clause = \np,vp -> {
      s = \\vt,b =>
        let
          subjagr : NPCase * Agr = case <vp.subj,vt> of {
            <VTrans,VPPerf> => <NPErg, vp.obj.a> ;
            <VTransPost,VPPerf> => <NPErg, defaultAgr> ;
            _ => <NPC Dir, np.a>
            } ;
          subj = subjagr.p1 ;
          agr  = subjagr.p2 ;
          vps  = vp.s ! b ! VPTense vt agr ;
        in
        np.s ! subj ++ vp.obj.s ++ vp.comp ! np.a ++ vps.neg ++
        vps.inf ++ vps.fin
      } ;
```

In `SentenceHin.gf`

```
lin PredVP = mkClause ;
```

Test predication

```
> l -all -tr (PredVP (DetArtSg DefArt boy_N) (UseV go_V)) | ? 'runghc hdeva.hs'
> l -all -tr (PredVP (DetArtSg DefArt girl_N) (UseV go_V)) | ? 'runghc hdeva.hs'
> l -all -tr (PredVP (DetArtPl DefArt boy_N) (UseV go_V)) | ? 'runghc hdeva.hs'
> l -all -tr (PredVP (DetArtPl DefArt girl_N) (UseV go_V)) | ? 'runghc hdeva.hs'
```

Two-place verbs: the linearization type

In `ResHin.gf`

```
  param VType = VIntrans | VTrans | VTransPost ;
  oper Compl : Type = {s : Str ; c : VType} ;
```

In `CatHin.gf`

```
  lincat V2 = Verb ** {c2 : Compl} ;
```

Two-place verbs: paradigm API

In `ParadigmsHin.gf`: start from regular transitive

```
mkV2 = overload {
  mkV2 : Str -> V2
    = \s -> regVerb s ** {c2 = {s = [] ; c = VTrans} ; lock_V2 = <>} ;
  mkV2 : V -> V2
    = \v -> v ** {c2 = {s = [] ; c = VTrans} ; lock_V2 = <>} ;
  mkV2 : V -> Str -> V2
    = \v,p -> v ** {c2 = {s = p ; c = VTransPost} ; lock_V2 = <>} ;
  } ;
```

Two-place verb entries in Lexicon

In `LexiconHin`:

```
eat_V2 = mkV2 "Ka:" ;
hit_V2 = mkV2 (mkV "ma:r") "ko" ;
```

Test inflection

```
> i LangHin.gf
> l -all -tr eat_V2 | ? 'runghc hdeva.hs'
> l -all -tr hit_V2 | ? 'runghc hdeva.hs'
```

Complementation

In `ResHin.gf`

```
VPHSlash = VPH ** {c2 : Compl} ;

insertObject : NP -> VPHSlash -> VPH = \np,vps -> {
    s = vps.s ;
    obj = {s = vps.obj.s ++ np.s ! NPC Obl ++ vps.c2.s ; a = np.a} ;
    subj = vps.c2.c ;
    comp = vps.comp
    } ;
```

In `CatHin.gf`

```
lincat VPSlash = VPHSlash ;
```

In `VerbHin.gf`

```
lin SlashV2a v = predV v ** {c2 = v.c2} ;
lin ComplSlash vp np = insertObject np vp ;
```

Test complementation

Linearize all tenses of

```
(PredVP (DetArtSg DefArt (UseN girl_N))
  (ComplSlash (SlashV2a eat_V2) (DetArtSg DefArt (UseN apple_N))))
```

and variations

```
DetArtPl, boy_N, hit_V2, bread_N / girl_N / boy_N
```

Adjectival modification

In `CatHin.gf`

```
  lincat AP = Adjective ;
```

In `AdjectiveHin.gf`

```
  lin PositA a = a ;
```

In `NounHin.gf`

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

Testing modification

```
  l -all -tr (AdjCN (PositA good_A) (UseN boy_N)) | ? 'runghc hdeva.hs'
```

with variants with other nouns and adjectives.

Scaling up coverage

Components

Lexicon

Extra syntax

Shallowing and ambiguity reduction

- via application grammar
- to improve parsing performance

Ongoing project: English

First targeting the FraCaS test suite

Completed with semantics and theorem proving

Björn Bringert (2008)

Proposed project: Swedish

SALDO lexicon: 70,000 lemmas

Lars Borin and Markus Forsberg (2008)