Computational Syntax

Aarne Ranta

Language Technology Masters Course, Gothenburg, Spring 2011

Contents

Syntax in the resource grammar

"Linguistic ontology": syntactic structures common to languages

80 categories, 200 functions, which have worked for all resource languages so far

Sufficient for most purposes of expressing meaning: mathematics, technical documents, dialogue systems

Must be extended by language-specific rules to permit parsing of arbitrary text (ca. 10% more in English?)

A lot of work, easy to get wrong!

The key categories and functions

The key categories

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

The key functions

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

Feature design

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

agr = **agreement features**: gender, number, person

Predication: building clauses

Interplay between features

```
param Tense, Case, Agr

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

fun PredVP : NP -> VP -> Cl

lin PredVP np vp = {s = \\t => np.s ! subj ++ vp.s ! t ! np.a}

oper subj : Case
```

Feature passing

In general, combination rules just pass features: no case analysis (`table` expressions) is performed.

A special notation is hence useful:

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

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

Predication: examples

English

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

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

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

Predication: variations

Word order:

- *will I sleep* (English), *è stanca lei* (Italian)

Pro-drop:

- *io sono stanco* vs. *sono stanco* (Italian)

Ergativity:

- ergative case of transitive verb subject; agreement to object (Hindi)

Variable subject case:

- *minä olen lapsi* vs. *minulla on lapsi* (Finnish, ”I am a child” (nominative) vs. ”I have a child” (adessive))

Complementation: building verb phrases

Interplay between features

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

fun ComplV2 : V2 -> NP -> VP

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

Complementation: examples

English

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

Finnish

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

Complementation: variations

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

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

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

Clitics: the place of the subject can vary, as in Italian:

- *Maria ama Giovanni* vs. *Maria mi ama* ("Mary loves John" vs. "Mary loves me")

Determination: building noun phrases

Interplay between features

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

fun DetCN : Det -> CN -> NP

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

oper agr : Gender -> Number -> Person -> Agr
```

Determination: examples

English

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

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

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

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

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

Determination: variations

Systamatic number variation:

- *this-these*, *the-the*, *il-i* (Italian ”the-the”)

”Zero” determiners:

- *talo* (”a house”) vs. *talo* (”the house”) (Finnish)
- *a house* vs. *houses* (English), *une maison* vs. *des maisons* (French)

Specificity parameter of nouns:

- *varje hus* vs. *det huset* (Swedish, ”every house” vs. ”that house”)

Modification: adding adjectives to nouns

Interplay between features

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

fun AdjCN : AP -> CN -> CN

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

Modification: examples

English

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

Italian (”red wine”, ”red house”)

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

Finnish (”red house”)

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

Modification: variations

The place of the adjectival phrase

- Italian: *casa rossa*, *vecchia casa* (”red house”, ”old house”)
- English: *old house*, *house similar to this*

Specificity parameter of the adjective

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

Adjectival predication

Interplay between features

```
lincat AP = {s : Gender => Number => Case => Str}
lincat VP = {s : Tense => Agr => Str            }

fun CompAP : AP -> VP

lin CompAP ap =
  {s = \\t,a => copula t a ++ ap.s ! gender a ! number a ! nom}

oper
  copula : Tense -> Agr -> Str
  gender : Agr -> Gender
  number : Agr -> Number
  nom    : Case
```

Adjectival predication: examples

English

tense	**agr**	**indicative VP**
Pres	Sg 3	*is old*
Pres	Pl	*are old*
Perf	Sg 3	*has been old*

Italian

tense	**agr**	**indicative VP**
Pres	Fem Sg 1	*sono stanco*
Perf	Masc Sg 2	*sei stato stanco*
Perf	Fem Sg 2	*sei stata stanca*

Adjectival predication: variations

Form of adjective

- invariable predicative form in German: (*er ist, sie ist, wir sind*) *alt*
- case depends on number in Finnish: *olen vanha* ("I am old", singular nominative) vs. *olemme vanhoja* ("we are old", plural partitive)

No copula (in present tense)

- Russian: *Ivan staryi* ("John (is) old")
- Arabic:

- *ar-ragulu qadi:mun* (”the man (is) old”)

- but past tense: *ka:na r-ragulu qadi:man* (”the man was old”, accusative!)

Selecting tense and polarity

Category `S` (sentence) fixes the variable tense and polarity of `Cl`, by using **abstract parameters** (i.e. functions in abstract syntax).

```
cat
  S ; Tense ; Pol ;
fun
  UseCl : Tense -> Pol -> Cl -> S ;
  Pos, Neg : Pol ;
  Pres, Perf : Tense ;
```

Selecting tense and polarity: concrete syntax

The abstract parameters are mapped to concrete syntax realizations consisting of strings (which can be empty) and parameters passed to the clause.

```
lincat
  S     = {s : Str} ;
  Tense = {s : Str ; t : TTense} ;
  Conj  = {s : Str ; n : Number} ;
lin
  UseCl t p cl = {s = t.s ++ p.s ++ cl.s ! t.t ! p.b} ;
  Pos  = {s = ... ; b = True} ; Neg  = {s = ... ; b = False} ;
  Pres = {s = ... ; t = TPres} ; Perf = {s = ... ; t = TPerf} ;
```

(Notice: `TTense = TPres` | `TPerf` is a parameter type in the resource.)

Lexical insertion

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

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

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

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

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

However, for `UseV` in particular, this will usually be more complex.

The head of a phrase

The inserted word is the **head** of the phrases built from it:

- *house* is the head of *house*, *big house*, *big old house* etc

As a rule with many exceptions and modifications,

- variable features are passed from the phrase to the head
- inherent features of the head are inherited by the noun

This works for **endocentric** phrases: the head has the same type as the full phrase.

What is the head of a noun phrase?

In an `NP` of form `Det CN`, is `Det` or `CN` the head?

Neither, really, because features are passed in both directions:

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

Moreover, this `NP` is **exocentric**: no part is of the same type as the whole.

Structural words

Structural words = **function words**, words with special grammatical functions

- determiners: *the*, *this*, *every*
- pronouns: *I*, *she*
- conjunctions: *and*, *or*, *but*

Often members of **closed classes**, which means that new words are never (or seldom) introduces to them.

Linearization types are often specific and inflection are irregular.

A miniature resource grammar for Italian

We divide it to five modules - much fewer than the full resource!

```
abstract Grammar                              -- syntactic cats and funs

abstract Test = Grammar **...                 -- test lexicon added to Grammar

resource ResIta                               -- resource for Italian

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

concrete TestIta of Test = GrammarIta ** open ResIta in... -- It. lexicon
```

Extension vs. opening

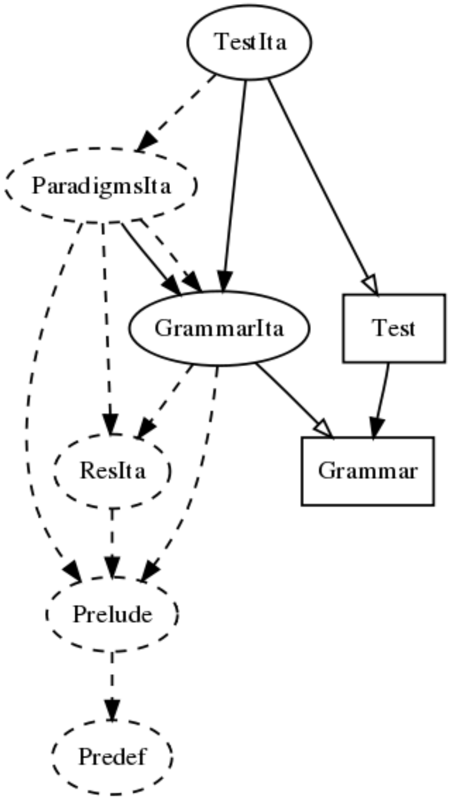
Module extension: `N = M1, M2, M3 ** {...}`

- module `N` **inherits** all judgements from `M1,M2,M3`

Module opening: `N = open R1, R2, R3 in {...}`

- module `N` can use all judgements from `R1,R2,R3` (but doesn't inherit them)

Module dependencies



rectangle = abstract, solid ellipse = concrete, dashed ellipse = resource

Producing the dependency graph

Using the command `dg` = `dependency_graph` and graphviz

```
> i -retain LangIta.gf
> dependency_graph
wrote graph in file _gfdepgraph.dot
> ! dot -Tpng _gfdepgraph.dot >testdep.png
```

The module Grammar

```
abstract Grammar = {
  cat
    S ; Cl ; NP ; VP ; AP ; CN ; Det ; N ; A ; V ; V2 ; Tense ; Pol ;
  fun
    UseCl   : Tense -> Pol -> Cl -> S ;
    PredVP  : NP  -> VP -> Cl ;
    ComplV2 : V2  -> NP -> VP ;
    DetCN   : Det -> CN -> NP ;
    ModCN   : CN  -> AP -> CN ;

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

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

    Pos, Neg : Pol ;
    Pres, Perf : Tense ;
}
```

English implementation

Parameters

Parameters are defined in `ResEng.gf`. Just 3 of the 5 verb forms.

```
Number = Sg | Pl ;
Case   = Nom | Acc ;
Agr    = Ag Number Person ;
TTense = TPres | TPerf ;
Person = Per1  | Per2 | Per3 ;

VForm = VInf | VPres | VPast | VPart ;
```

Tense and agreement of a verb phrase, in syntax

`UseV arrive_V`	**Pres, True**	**Pres, False**	**Perf**
Ag Sg Per3	*arrives*	*does not arrive*	*has (not) arrived*
Ag _ _	*arrive*	*do not arrive*	*have (not) arrived*

The forms of a verb, in morphology

`arrive_V`	**form**
VInf	*arrive*
VPres	*arrives*
VPart	*arrived*

The verb phrase type

Lexical insertion maps `V` to `VP`.

```
lincat VP = {
  verb  : AgrVerb ;
  compl : Str
  } ;

oper AgrVerb : Type = {
  s : TTense => Bool => Agr => Str
  } ;
```

An auxiliary

```
oper agrV : Verb -> AgrVerb = \v -> {
  s = \\t,p,a => case <t,p,a> of {
     <TPres,True, Ag Sg Per3> => v.s ! VPres ;
     <TPres,False,Ag Sg Per3> => "does not" ++ v.s ! VInf ;
     <TPres,True, _>          => v.s ! VInf ;
     <TPres,False,_>          => "do not" ++ v.s ! VInf ;
     <TPerf,_,Ag Sg Per3>     => "has" ++ neg p ++ v.s ! VPart ;
     <TPerf,_,_         >     => "have" ++ neg p ++ v.s ! VPart
     }
  } ;
```

Verb phrase formation, V and V2

```
lin
  ComplV2 v2 np = {
    verb = agrV v2 ;
    compl = v2.c ++ np.s ! Acc
    } ;

  UseV v = {
    verb = agrV v ;
    compl = []
    } ;
```

Verb phrase formation, copula

```
lin CompAP ap = {
    verb = copula ;
    compl = ap.s
    } ;

oper copula : AgrVerb = {
  s = \\t,p,a => case <t,a> of {
     <TPres,Ag Sg Per1> => "am" ++ neg p ;
     <TPres,Ag Sg Per3> => "is" ++ neg p ;
     <TPres,_         > => "are" ++ neg p ;
     <TPerf,Ag Sg Per3> => "has" ++ neg p ++ "been" ;
     <TPerf,_         > => "have" ++ neg p ++ "been"
     }
  } ;
```

English noun phrases

Worst case: as pronouns

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

lin
  i_NP =   pronNP "I" "me" Sg Per1 ;
  she_NP = pronNP "she" "her" Sg Per3 ;
  we_NP =  pronNP "we" "us" Pl Per1 ;

oper pronNP : (s,a : Str) -> Number -> Person -> NP =
\s,a,n,p -> {
  s = table {
    Nom => s ;
    Acc => a
    } ;
  a = Ag n p
  } ;
```

Determination

```
lincat Det = {s : Str ; n : Number} ;

lin
  DetCN det cn = {
    s = \\_ => det.s ++ cn.s ! det.n ;
    a = Ag det.n Per3
    } ;

  every_Det = mkDet "every" Sg ;
  the_Det = mkDet "the" Sg ;

oper mkDet : Str -> Number -> {s : Str ; n : Number} = \s,n -> {
  s = s ;
  n = n
  } ;
```

The indefinite article

Prefix-dependent token:

```
lin
  a_Det = mkDet (pre {#vowel => "an" ; _ => "a"}) Sg ;

oper
  vowel : pattern Str = #("a" | "e" | "i" | "o") ;
```

This is an approximation; spelling doesn't determine vocalicity.

Adjectival phrases

Trivial in English

```
lincat AP, A = {s : Str} ;

lin
  ModCN ap cn = {
    s = \\n => ap.s ++ cn.s ! n
    } ;

  UseA adj = adj ;
```

Predication, at last

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

```
lin
  PredVP np vp = {
      s = \\t,b => np.s ! Nom ++ vp.verb.s ! t ! b ! np.a ++ vp.compl
    } ;
```

Selection of tense and polarity

The abstract parameters are empty strings.

```
lincat
  S  = {s : Str} ;
  Cl = {s : TTense => Bool => Str} ;
lin
  UseCl t p cl = {s = t.s ++ p.s ++ cl.s ! t.t ! p.b} ;
  Pos  = {s = [] ; b = True} ;
  Neg  = {s = [] ; b = False} ;
  Pres = {s = [] ; t = TPres} ;
  Perf = {s = [] ; t = TPerf} ;
```

The extended syntax

The new things

Utterances: questions, declaratives

Extraction: *who does John walk with*

Subordinate clauses: *John runs because Mary walks*

More **verb categories**: *know that John walks*, *wonder if John walks*, *want to walk*

Adverbs, **prepositions**

Coordination: conjunctions of sentences, noun phrases, adjectival phrases,...

More **tenses**: past (*slept*), future (*will sleep*)

Abstract syntax: new categories

```
Utt ;     -- utterance (sentence or question) e.g. "does she walk"
QS ;      -- question (fixed tense)           e.g. "who doesn’t walk"
QCl ;     -- question clause (variable tense) e.g. "who walks"
ClSlash ; -- clause missing noun phrase       e.g. "she walks with"
Adv ;     -- adverb                           e.g. "here"
Prep ;    -- preposition (and/or case)        e.g. "with"
VS ;      -- sentence-complement verb         e.g. "know"
VQ ;      -- question-complement verb         e.g. "wonder"
VV ;      -- verb-phrase-complement verb      e.g. "want"
IP ;      -- interrogative pronoun            e.g. "who"
PN ;      -- proper name                      e.g. "John"
Subj ;    -- subjunction                      e.g. "because"
IAdv ;    -- interrogative adverb             e.g. "why"
```

Abstract syntax: new sentence-level combination rules

```
UttS  : S -> Utt ;
UttQS : QS -> Utt ;

UseQCl : Tense -> Pol -> QCl -> QS ;

QuestCl    : Cl -> QCl ;               -- does she walk
QuestVP    : IP -> VP -> QCl ;         -- who walks
QuestSlash : IP -> ClSlash -> QCl ;    -- who does she walk with
QuestIAdv  : IAdv -> Cl -> QCl ;       -- why does she walk

SubjCl : Cl -> Subj -> S -> Cl ;       -- she walks because we run

CompAdv : Adv -> VP ;                  -- (be) here
PrepNP  : Prep -> NP -> Adv ;          -- in the house
```

Question forms

1. **Sentential questions**, formed from a clause

```
QuestCl    : Cl -> QCl ;               -- does she walk ; is she old
```

2. **Wh questions**, formed with an interrogative pronoun

```
QuestVP    : IP -> VP -> QCl ;         -- who walks ; who walks with her
QuestSlash : IP -> ClSlash -> QCl ;    -- who does she walk with
```

3. **Interrogative adverbial question**, formed with an interrogative adverbial

```
QuestIAdv  : IAdv -> Cl -> QCl ;       -- why/where/when does she walk
```

Direct vs. indirect questions

Direct = as main clause; indirect = in subordinate clause

direct	**indirect (I wonder...)**
is she old	*if she is old*
does she walk	*if she walks*
who is there	*who there is*
who does she love	*who she loves*
why does she walk	*why she walks*

English makes them more different than many other languages; e.g. in Finnish, there is no difference.

Extraction

Also known as **wh movement**:

> *she loves* **him** -> **she loves* **who** -> **who** *she loves*, **who** *does she love*

We form this by introducing a **slash category**, `ClSlash`, ”clause missing an NP”

```
  SlashV2 : NP -> V2 -> ClSlash ;   -- she loves
```

Often denoted Cl/NP, whence the ”slash” (but this is not correct notation in GF).

Long distance dependencies

The ”missing NP” can be arbitrarily deep in the verb phrase.

We consider just one way of doing this:

```
  SlashPrep : Cl -> Prep -> ClSlash ; -- she walks with
```

This gives us

who does she walk with

who does she walk in the street in the morning with

More on extraction

Other forms of ”slash propagation”, e.g. *who do you want me to meet*

Usage in **relative clauses**, e.g. *the man that she walks with*

Interrogative adverbs could be seen as instances of Cl/Adv:

you live **here** -> **where** *you live*

English and Swedish: the preposition may follow or stay; the latter is more common in languages.

- *who do we walk with* (**preposition stranding**)
- *with who do we walk* (**pied piping**)

Subordinate clauses

```
  SubjCl : Cl -> Subj -> S -> Cl ;

  although_Subj, because_Subj, when_Subj : Subj ;
```

John runs because Mary walks

Verb categories

Also known as **subcategorization patterns** of verbs:

cat	**complement**	**example**
VS	sentence	*know that John walks*
VQ	question	*wonder if John walks*
VV	verb phrase	*want to walk*

VS and VQ also introduce subordinate clauses

More still possible: V3 (*she talks to him about it*), VA (*she becomes hungry*), V2A (*she paints it red*), V2V (*she asks him to leave*),...

Adverbs and prepositions

Lexical adverbs: *here*, *now*

Prepositional phrases: *with Mary*, *in every village*

```
PrepNP  : Prep -> NP -> Adv ; -- in the house
```

More generally:

- **ad-adjectives** (`AdA`): *very*
- **sentential adverbs**: *always* (different position: *she always runs here*)

Category `Adv` also usable as predicate: *she is here*

```
CompAdv : Adv -> VP ;        -- be here
```

Coordination

Sentences: *John walks and Mary runs*

```
ConjS   : Conj -> S  -> S  -> S ;
```

Adjectival phrases: *big or small*

```
ConjAP  : Conj -> AP -> AP -> AP ;
```

Noun phrases: *John and Mary*

```
ConjNP  : Conj -> NP -> NP -> NP ;
```

More generally:

- for many more categories X: `ConjX : Conj -> X -> X -> X'`
- list conjunction: `ConjX : Conj -> ListX -> X'`

The tense system

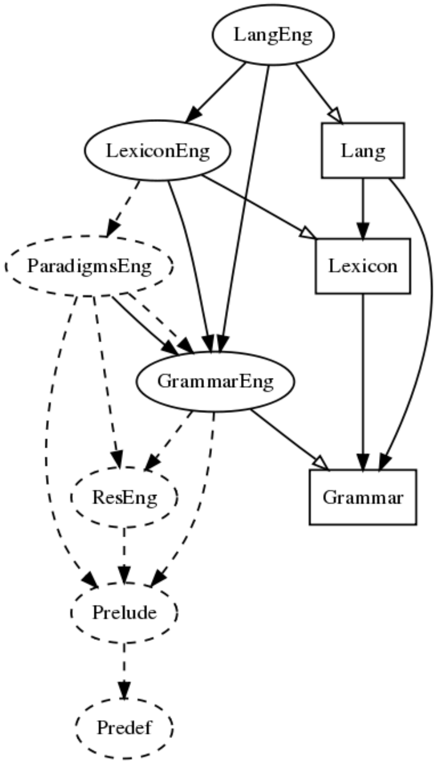
More **tenses**: past (*slept*), future (*will sleep*)

More genarally:

- tense (present, past, future, conditional) * order (simultaneous, anterior)
- mood: indicative vs. conjunctive/subjunctive ; imperative
- nominal forms: infinitives, participles

English implementation

Module dependencies



Notice: names like in RGL; separate Lexicon and Lang.

Need to generalize the VP type

Main challenge: **inversion**

she **is old**	**is** *she* **old**
she **has walked**	**has** *she* **walked**
she **does not walk**	**does** *she* **not walk**
she **walks**	**does** *she walk* (not: *walks she* !)

We need to add **discontinuity** to VP. Moreover, we have the irregular behaviour in simple tenses with positive polarity.

The new VP type

Separate fields for **finite** and **infinite** parts.

```
  VP = {
    verb  : AgrVerb ;
    compl : Str
    } ;
  AgrVerb = {
    s : ClForm => TTense => Bool => Agr => {fin,inf : Str} ;
    inf : Str
    } ;
  param ClForm = ClDir | ClInv ;
```

The finite part is the one that goes before the subject in inversion: **has** *she walked*. (The `inf` field is used in constructions like *want to VP*; see later)

How the parts look depends on the clause form, tense, and polarity.

Using the VP type: predication

```
lin PredVP np vp = {
      s = \\d,t,b =>
        let
          vps = vp.verb.s ! d ! t ! b ! np.a
        in case d of {
          ClDir => np.s ! Nom ++ vps.fin ++ vps.inf ++ vp.compl ;
          ClInv => vps.fin ++ np.s ! Nom ++ vps.inf ++ vp.compl
          }
      } ;
```

Producing a VP verb from a verb

```
oper agrV : Verb -> AgrVerb = \v ->
    let
      vinf  = v.s ! VInf ;
      vpart = v.s ! VPart
    in {
    s = \\d,t,p,a => case <d,t,p,a> of {
       <ClDir,TPres,True, Ag Sg Per3> => {fin = v.s ! VPres ; inf = []} ;
       <_,    TPres,_,    Ag Sg Per3> => {fin = "does"      ; inf = neg p ++ vinf} ;
       <ClDir,TPres,True, _         > => {fin = vinf        ; inf = []} ;
       <_,    TPres,_,    _         > => {fin = "do"        ; inf = neg p ++ vinf} ;
       <_,    TPerf,_,    Ag Sg Per3> => {fin = "has"       ; inf = neg p ++ vpart} ;
       <_,    TPerf,_,    _         > => {fin = "have"      ; inf = neg p ++ vpart} ;
       <ClDir,TPast,True, _         > => {fin = v.s ! VPast ; inf = []} ;
       <_,    TPast,_,    _         > => {fin = "did"       ; inf = neg p ++ vinf} ;
       <_,    TFut, _,    _         > => {fin = "will"      ; inf = neg p ++ vinf}
       } ;
    inf = vinf
    } ;
```

A similar generalization for `copula`.

Using clauses in sentential questions

```
lincat  QCl = {s : QForm => TTense => Bool => Str} ;

lin QuestCl cl = {s = \\q,t,p =>
      case q of {
        QDir   => cl.s ! ClInv ! t ! p ;
        QIndir => "if" ++ cl.s ! ClDir ! t ! p
        }
      } ;

param QForm = QDir | QIndir ;
```

Using clauses in adverbial questions: almost the same

```
lin QuestIAdv iadv cl = {s = \\q,t,p =>
      iadv.s ++
      case q of {
        QDir   => cl.s ! ClInv ! t ! p ;
        QIndir => cl.s ! ClDir ! t ! p
        }
      } ;
```

The slash category

Clause + complement case

```
  lincat ClSlash = {s : ClForm => TTense => Bool => Str ; c : Str} ;
```

Using it: append the preposition to the clause (preposition stranding) *or* to the interrogative pronoun (pied piping):

```
  lin QuestSlash ip cls = {
      s = (\\q,t,p => ip.s ++ cls.s ! ClInv ! t ! p ++ cls.c)
        | (\\q,t,p => cls.c ++ ip.s ++ cls.s ! ClInv ! t ! p)
      } ;
```

Producing slashes

Like predication, V2 alone as verb phrase:

```
lin SlashV2 np v2 = {
    s = \\d,t,b =>
      let
        vps = (agrV v2).s ! d ! t ! b ! np.a
      in case d of {
        ClDir => np.s ! Nom ++ vps.fin ++ vps.inf ;
        ClInv => vps.fin ++ np.s ! Nom ++ vps.inf
        } ;
    c = v2.c
    } ;
```

Complementation with the new verb categories

```
ComplVS v s = {
  verb = agrV v ;
  compl = "that" ++ s.s
  } ;
ComplVQ v q = {
  verb = agrV v ;
  compl = q.s ! QIndir
  } ;
ComplVV v vp = {
  verb = v.s ;
  compl = case v.isAux of {
    True => infVP vp ;
    False => "to" ++ infVP vp
    }
  } ;
```

Auxiliaries

Another complication with English: VV's that are auxiliaries vs. VV's that are not (*I can walk* vs. *I want* **to** *walk*)

Auxiliaries

- don't use *to* (except in compounds: *I have been able to walk*)
- don't use auxiliary *do*: *can she walk* vs. *does she want to walk*

The most common VV's are classified as structural words and not in `Lexicon` (as they often have a special place in grammars, e.g. behave like auxiliaries).

Coordination

Straightforward for sentences and adjectival phrases:

```
ConjS  co x y = {s = x.s ++ co.s ++ y.s} ;
ConjAP co x y = {s = x.s ++ co.s ++ y.s} ;
```

For noun phrases, we need to return correct agreement features:

John or Mary **walks**: singular + singular = singular with *or*

John and Mary **walk**: singular + singular = plural with *and*

John or all other boys **walk**: singular + plural = plural

NP coordination, the idea

Idea: the agreement feature of the complex is a function of

- the features of the conjuncts
- the feature of the conjunction (*and* is plural, *or* is singular)

Thus:

- the number is singular iff every number is singular
- the person is 1st iff any person is 1st, 3rd iff all persons are 3rd

In other words: minimum function with `Pl` and `Per1` as minimal values.

NP coordination, the code

```
lincat
  NP = {s : Case => Str ; a : Agr} ;
  Conj = {s : Str ; n : Number} ;
lin ConjNP co nx ny = {
      s = \\c => nx.s ! c ++ co.s ++ ny.s ! c ;
      a = conjAgr co.n nx.a ny.a
      } ;
oper
  conjAgr : Number -> Agr -> Agr -> Agr = \n,xa,ya ->
    case <xa,ya> of {
      <Ag xn xp, Ag yn yp> =>
        Ag (conjNumber (conjNumber xn yn) n) (conjPerson xp yp)
      } ;
  conjNumber : Number -> Number -> Number = \m,n ->
    case m of {Pl => Pl ; _ => n} ;
  conjPerson : Person -> Person -> Person = \p,q ->
    case <p,q> of {
      <Per1,_> | <_,Per1> => Per1 ;
      <Per2,_> | <_,Per2> => Per2 ;
      _                   => Per3
      } ;
```

Italian implementation

First the basic part, as in the book Chapter 9.

Parameters

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

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

Agr = Ag Gender Number Person ;

VForm = VInf | VPres Number Person | VPart Gender Number ;
```

Algebraic datatypes

Parameter types that are not just enumerated, but have a **hierarchy**.

Instead of plain constants, **constructors** that take arguments.

```
param VForm = VInf | VPres Number Person | VPart Gender Number ;
```

The **values** are thus:

```
VInf
VPres Sg Per1, VPres Sg Per2, VPres Sg Per3,
  VPres Pl Per1, VPres Pl Per2, VPres Pl Per3
VPart Masc Sg, VPart Masc Pl, VPart Fem Sg, VPart Fem Pl
```

Italian verb phrases

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

The forms of a verb, in morphology

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

Inherent feature: `aux` is *essere*.

The verb phrase type

Lexical insertion maps `V` to `VP`.

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

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

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

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

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

Verb phrase formation

Lexical insertion is trivial.

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

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

```
lin ComplV2 =
  let
    nps = np.s ! v2.c
  in {
    v = {s = v2.s ; aux = v2.aux} ;
    clit = nps.clit ;
    obj  = nps.obj
    }
```

Italian noun phrases

Being clitic depends on case

```
 lincat NP = {s : Case => {clit,obj : Str} ; a : Agr} ;
```

Examples:

```
lin she_NP = {
   s = table {
     Nom => {clit = []   ; obj = "lei"} ;
     Acc => {clit = "la" ; obj = []} ;
     Dat => {clit = "le" ; obj = []}
     } ;
   a = Ag Fem Sg Per3
   }
lin John_NP = {
   s = table {
     Nom | Acc => {clit = [] ; obj = "Giovanni"} ;
     Dat       => {clit = [] ; obj = "a Giovanni"}
     } ;
   a = Ag Fem Sg Per3
   }
```

Noun phrases: alternatively

Use a feature instead of separate fields,

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

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

Determination

No surprises

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

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

Building determiners

Often from adjectives:

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

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

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

Articles: see `GrammarIta.gf`

Adjectival modification

Recall the inherent feature for position

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

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

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

Italian morphology

Complex but mostly great fun:

```
regNoun : Str -> Noun = \vino -> case vino of {
  fuo + c@("c"|"g") + "o" => mkNoun vino (fuo + c + "hi") Masc ;
  ol  + "io" => mkNoun vino (ol + "i") Masc ;
  vin + "o"  => mkNoun vino (vin + "i") Masc ;
  cas + "a"  => mkNoun vino (cas + "e") Fem ;
  pan + "e"  => mkNoun vino (pan + "i") Masc ;
  _ => mkNoun vino vino Masc
  } ;
```

See `ResIta` for more details.

Predication, at last

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

```
lin PredVP np vp =
    let
      subj = (np.s ! Nom).obj ;
      obj  = vp.obj ;
      clit = vp.clit ;
      verb = table {
        Pres => agrV vp.v np.a ;
        Perf => agrV (auxVerb vp.v.aux) np.a ++ agrPart vp.v np.a
        }
    in {
      s = \\t => subj ++ clit ++ verb ! t ++ obj
    } ;
```

Selection of verb form

We need it for the present tense

```
oper agrV : Verb -> Agr -> Str = \v,a -> case a of {
  Ag _ n p => v.s ! VPres n p
  } ;
```

The participle agrees to the subject, if the auxiliary is *essere*

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

The definite article

Notorious: there are prefix-dependent forms

default	*il vino*	"the wine"
before vowel	*l'albero*	"the tree"
before 'impure s'	*lo stato*	"the state"

Moreover, there are contractions between preposition and article

bare preposition	*a Giovanni*	"to Giovanni"
with default article	*al vino*	"to the wine"
article before vowel	*all'albero*	"to the tree"
article before 'impure s'	*allo stato*	"to the state"

Solution in GF: case parameter

Treat these prepositions as case (needed for pronouns anyway):

```
param Case = Nom | Acc | Dat | Gen | C_in | C_con | C_da ;
```

In "bare" usage, they produce strings

```
oper prepCase : Case -> Str = \c -> case c of {
  Dat => "a" ;
  Gen => "di" ;
  C_con => "con" ;
  C_in => "in" ;
  C_da => "da" ;
  _ => []
  } ;
```

Contractions with articles

```
lin the_Det = {
    s = table {
      Masc => table {
        Nom | Acc => elisForms "lo" "l’" "il" ;
        Dat => elisForms "allo" "all’" "al" ;
        Gen => elisForms "dello" "dell’" "del" ;
        C_in => elisForms "nello" "nell’" "nel" ;
      -- etc
oper
  elisForms : (_,_,_ : Str) -> Str = \lo,l’,il ->
    pre {#s_impuro => lo ; #vowel => l’ ; _ => il} ;
  vowel : pattern Str = #("a" | "e" | "i" | "o" | "u" | "h") ;
  s_impuro : pattern Str =
    #("z" | "s" + ("b"|"c"|"d"|"f"|"m"|"p"|"q"|"t")) ;
```

Similar solution as for English *a,an* but it works perfectly for Italian.

Italian, extended

More verb forms

Introducing mood and more tenses.

```
param
  Mood = Ind | Con ;

  VForm =
     VInf
   | VInfContr         -- contracted infinitive, "amar"
   | VPres Mood Number Person
   | VPast Mood Number Person
   | VFut Number Person
   | VPart Gender Number ;
```

How many values of `VForm` are there now?

Sentences and clauses with mood

Because of `VS` and `Subj`, we need to vary `S` in mood.

```
lincat
  S  = {s : Mood => Str} ;
  Cl = {s : Mood => ResIta.Tense => Bool => Str} ;
  VS = Verb ** {m : Mood} ;
  Subj = {s : Str ; m : Mood} ;

lin
  SubjCl cl subj s = {
    s = \\m,t,b => cl.s ! m ! t ! b ++ subj.s ++ s.s ! subj.m
    } ;
```

Sharing the complementation code

To avoid copy and paste: we define in `ResIta`

```
oper useV : Verb -> (Agr => Str) -> VP = \v,o -> {
  v = v ;
  clit = [] ;
  clitAgr = CAgrNo ;
  obj = o
 } ;
```

Then we can write in `GrammarIta`

```
lin
  UseV v       = useV v         (\\_ => []) ;
  CompAdv adv = useV essere_V (\\_ => adv.s) ;
  ComplVS v s = useV v         (\\_ => "che" ++ s.s ! v.m) ;
  ComplVQ v q = useV v         (\\_ => q.s ! QIndir) ;
```

Next steps

Look at the last lecture of LREC Tutorial, and Book Chapter 10

Look at the Resource Library Synopsis in

http://www.grammaticalframework.org/lib/doc/synopsis.html

Application of Computational Syntax

Contents

Software libraries: programmer's vs. users view

Semantic vs. syntactic grammars

Example of semantic grammar and its implementation

Interfaces and parametrized modules

Free variation

Overview of the Resource Grammar API

Software libraries

Collections of reusable functions/types/classes

API = **Application Programmer's Interface**

- show enough to enable use
- hide details

Example: maps (lookup tables, hash maps) in Haskell, C++, Java, ...

```
type Map
lookup : key -> Map -> val
update : key -> val -> Map -> Map
```

Hidden: the definition of the type `Map` and of the functions `lookup` and `update`.

Advantages of software libraries

Programmers have

- less code to write (e.g. *how* to look up)
- less techniques to learn (e.g. efficient Map datastructures)

Improvements and bug fixes can be inherited

Grammars as software libraries

Smart paradigms as API for morphology

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

Abstract syntax as API for syntactic combinations

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

Using the library: natural language output

Task: in an email program, generate phrases saying *you have n message(s)*

Problem: avoid *you have one messages*

Solution: use the library

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

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

Software localization

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

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

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

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

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

Correct number in Arabic

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

(From "Implementation of the Arabic Numerals and their Syntax in GF" by Ali Dada, ACL workshop on Arabic, Prague 2007)

Use cases for grammar libraries

Grammars need *very* much *very* special knowledge, and a *lot* of work - thus an excellent topic for a software library!

Some applications where grammars have shown to be useful:

- software localization
- natural language generation (from formalized content)
- technical translation
- spoken dialogue systems

Two kinds of grammarians

Application grammarians vs. **resource grammarians**

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

We want a **division of labour**.

Two kinds of grammars

Application grammars vs. **resource grammars**

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

A.k.a. **semantic grammars** vs. **syntactic grammars**.

Meaning-preserving translation

Translation must preserve meaning.

It need not preserve syntactic structure.

Sometimes it is even impossible:

- *John likes Mary* in Italian is *Maria piace a Giovanni*

The abstract syntax in the semantic grammar is a logical predicate:

```
fun Like : Person -> Person -> Fact
lin Like x y = x ++ "likes" ++ y          -- English
lin Like x y = y ++ "piace" ++ "a" ++ x  -- Italian
```

Translation and resource grammar

To get all grammatical details right, we use resource grammar and not strings

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

  lin Like x y = PredVP x (ComplV2 like_V2 y)      -- Engligh
  lin Like x y = PredVP y (ComplV2 piacere_V2 x)  -- Italian
```

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

GF has **compile-time transfer**, and uses interlingua (semantic abstrac syntax) at run time.

Domain semantics

”Semantics of English”, or of any other natural language as a whole, has never been built.

It is more feasible to have semantics of **fragments** - of small, well-understood parts of natural language.

Such languages are called **domain languages**, and their semantics, **domain semantics**.

Domain semantics = **ontology** in the Semantic Web terminology.

Examples of domain semantics

Expressed in various formal languages

- mathematics, in predicate logic
- software functionality, in UML/OCL
- dialogue system actions, in SISR
- museum object descriptions, in OWL

GF abstract syntax can be used for any of these!

Example: abstract syntax for a ”Face” community

What messages can be expressed on the community page?

```
abstract Face = {

flags startcat = Message ;

cat
  Message ; Person ; Object ; Number ;
fun
  Have : Person -> Number -> Object -> Message ;  -- p has n o’s
  Like : Person -> Object -> Message ;            -- p likes o
  You : Person ;
  Friend, Invitation : Object ;
  One, Two, Hundred : Number ;
}
```

Notice the `startcat` flag, as the start category isn’t `S`.

Presenting the resource grammar

In practice, the abstract syntax of Resource Grammar is inconvenient

- too deep structures, too much code to write
- too many names to remember

We do the same as in morphology: overloaded operations, named `mk`*C* where *C* is the value category.

The resource defines e.g.

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

Relevant part of Resource Grammar API for "Face"

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

Function	**example**
`mkUtt : Cl -> Utt`	*John loves Mary*
`mkCl : NP -> V2 -> NP -> Cl`	*John loves Mary*
`mkNP : Numeral -> CN -> NP`	*five cars*
`mkNP : Det -> CN -> NP`	*that car*
`mkCN : N -> CN`	*car*
`this_Det : Det`	*this, these*
`you_NP : NP`	*you* (singular)
`n1_Numeral, n2_Numeral : Numeral`	*one, two*
`n100_Numeral : Numeral`	*one hundred*
`have_V2 : V2`	*have*

Concrete syntax for English

How are messages expressed by using the library?

```
concrete FaceEng of Face = open SyntaxEng, ParadigmsEng in {
lincat
  Message = Utt ;
  Person = NP ;
  Object = CN ;
  Number = Numeral ;
lin
  Have p n o = mkUtt (mkCl p have_V2 (mkNP n o)) ;
  Like p o = mkUtt (mkCl p like_V2 (mkNP this_Det o)) ;
  You = you_NP ;
  Friend = mkCN friend_N ;
  Invitation = mkCN invitation_N ;
  One = n1_Numeral ;
  Two = n2_Numeral ;
  Hundred = n100_Numeral ;
oper
  like_V2 = mkV2 "like" ;
  invitation_N = mkN "invitation" ;
  friend_N = mkN "friend" ;
}
```

Concrete syntax for Finnish

How are messages expressed by using the library?

```
concrete FaceFin of Face = open SyntaxFin, ParadigmsFin in {
lincat
  Message = Utt ;
  Person = NP ;
  Object = CN ;
  Number = Numeral ;
lin
  Have p n o = mkUtt (mkCl p have_V2 (mkNP n o)) ;
  Like p o = mkUtt (mkCl p like_V2 (mkNP this_Det o)) ;
  You = you_NP ;
  Friend = mkCN friend_N ;
  Invitation = mkCN invitation_N ;
  One = n1_Numeral ;
  Two = n2_Numeral ;
  Hundred = n100_Numeral ;
oper
  like_V2 = mkV2 "pitää" elative ;
  invitation_N = mkN "kutsu" ;
  friend_N = mkN "ystävä" ;
}
```

Functors and interfaces

English and Finnish: the same combination rules, only different words!

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 = Utt ;
  Person = NP ;
  Object = CN ;
  Number = Numeral ;
lin
  Have p n o = mkUtt (mkCl p have_V2 (mkNP n o)) ;
  Like p o = mkUtt (mkCl p like_V2 (mkNP this_Det o)) ;
  You = you_NP ;
  Friend = mkCN friend_N ;
  Invitation = mkCN invitation_N ;
  One = n1_Numeral ;
  Two = n2_Numeral ;
  Hundred = n100_Numeral ;
}
```

An English instance of the domain lexicon

Define the domain words in English

```
instance LexFaceEng of LexFace = open SyntaxEng, ParadigmsEng in {

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

Put everything together: functor instantiation

Instantiate the functor `FaceI` by giving instances to its interfaces

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

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

Also notice the domain search path.

Porting the grammar to Finnish

1. Domain lexicon: use Finnish paradigms and words

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

2. Functor instantiation: mechanically change `Eng` to `Fin`

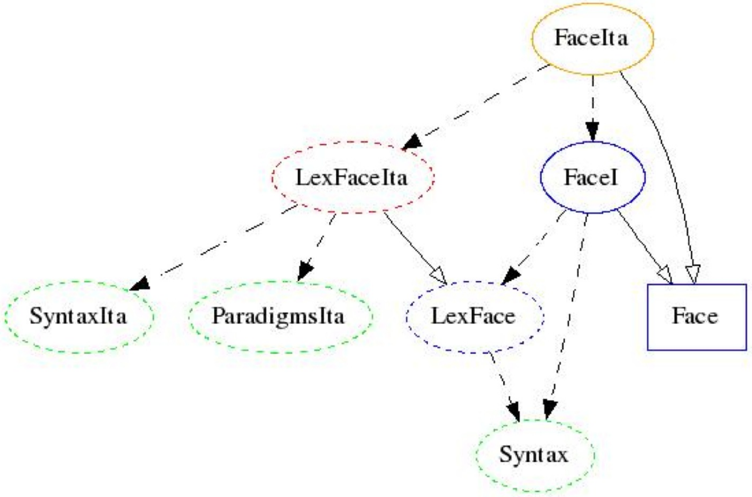
```
--# -path=.:present

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

Modules of a domain grammar: ”Face” community

1. Abstract syntax, `Face`
2. Parametrized concrete syntax: `FaceI`
3. Domain lexicon interface: `LexFace`
4. For each language *L*: domain lexicon instance `LexFace`*L*
5. For each language *L*: concrete syntax instantiation `Face`*L*

Module dependency graph



red = to do, orange = to do (trivial), blue = to do (once), green = library

Porting the grammar to Italian

1. Domain lexicon: use Italian paradigms and words

```
instance LexFaceIta of LexFace = open SyntaxIta, ParadigmsIta in {
oper
  like_V2 = mkV2 (mkV (piacere_64 "piacere")) dative ;
  invitation_N = mkN "invito" ;
  friend_N = mkN "amico" ;
}
```

2. Functor instantiation: **restricted inheritance**, excluding `Like`

```
concrete FaceIta of Face = FaceI - [Like] with
  (Syntax = SyntaxIta),
  (LexFace = LexFaceIta) ** open SyntaxIta in {
lin Like p o =
  mkUtt (mkCl (this_Det o) like_V2 p) ;
}
```

Building a web application

1. Compile the grammar to **PGF** (**Portable Grammar Format**)

```
$ gf -make -optimize-pgf FaceEng.gf FaceFin.gf FaceIta.gf
```

2. Start the PGF server (see also http://www.grammaticalframework.org/doc/quickstart.html)

```
$ pgf-http
```

3. Copy the PGF file to your server grammar repository, which is documentRoot + `/grammars/`

```
$ cp -p Face.pgf /home/aarne/.cabal/share/gf-server-1.0/www/grammars/
```

4. Open http://localhost:41296/minibar/minibar.html in your web browser, and select grammar Face.

Other applications

Translation Quiz: http://www.grammaticalframework.org/demos/TransQuiz/

Theorem proving: http://www.grammaticalframework.org:41297/syllogism/sy

Dialogue system: http://www.youtube.com/watch?v=1bfaYHWS6zU

More: http://www.grammaticalframework.org/demos/index.html

Free variation

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

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

```
fun BuyTicket : City -> City -> Request

lin BuyTicket x y =
  (("I want" ++ ((("to buy" | []) ++ ("a ticket")) | "to go"))
   |
  (("can you" | [] ) ++ "give me" ++ "a ticket")
   |
  []) ++
  "from" ++ x ++ "to" ++y
```

The variants can of course be resource grammar expressions as well.

Overview of the resource grammar API

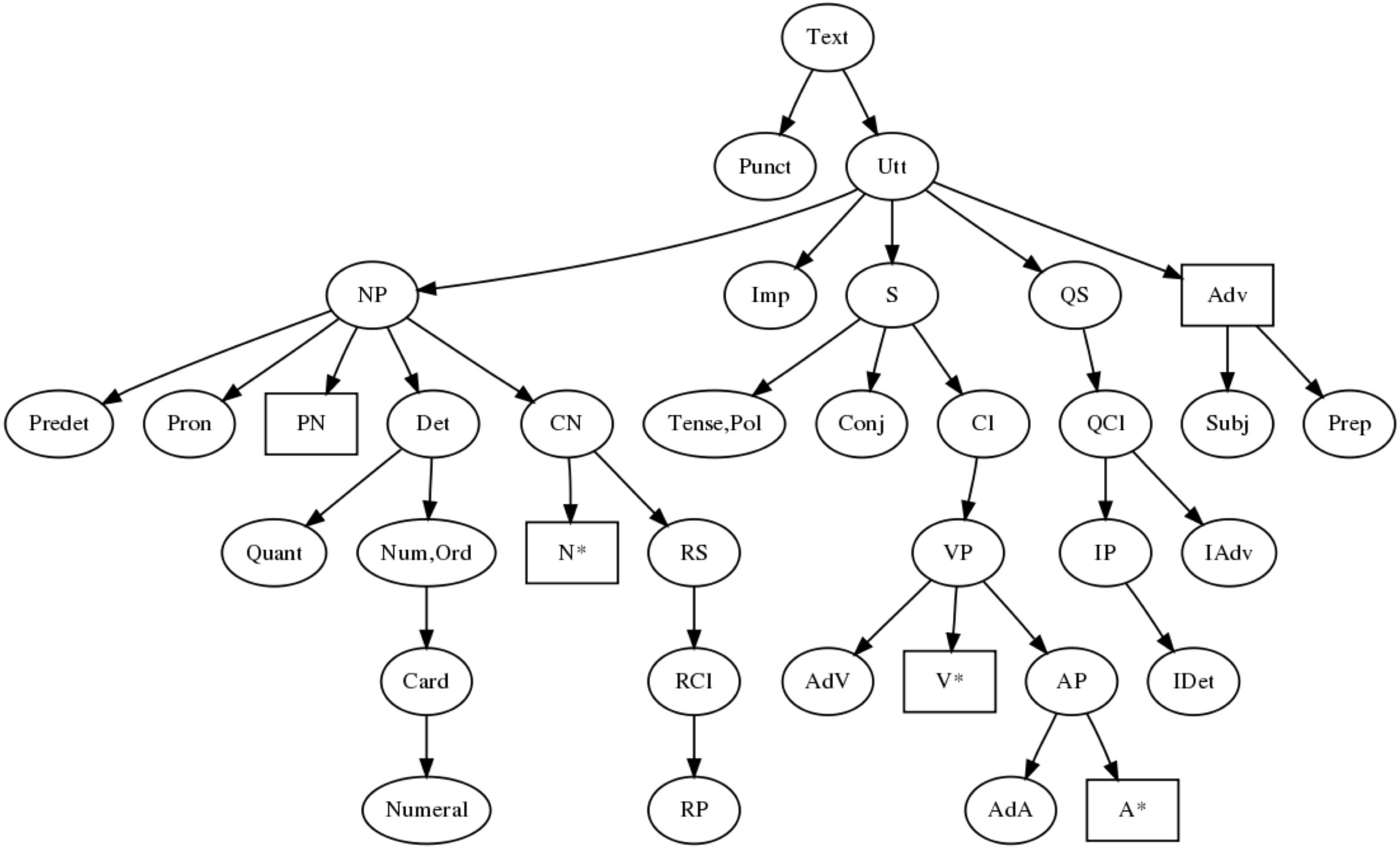
For the full story, see the **resource grammar synopsis** in

`grammaticalframework.org/lib/doc/synopsis.html`

Main division:

- `Syntax`, common to all languages
- `Paradigms`*L*, specific to language *L*

Main categories and their dependencies


Text
Punct
Utt
NP
Imp
S
QS
Adv
Predet
Pron
PN
Det
CN
Tense,Pol
Conj
Cl
QCl
Subj
Prep
Quant
Num,Ord
N*
RS
VP
IP
IAdv
Card
RCl
AdV
V*
AP
IDet
Numeral
RP
AdA
A*

Categories of complex phrases

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

Lexical categories for building predicates

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

Functions for building predication clauses

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

Noun phrases and common nouns

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

Questions and interrogatives

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

Sentence formation, tense, and polarity

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

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

Utterances and imperatives

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

More

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

Relative clauses: *man who owns a donkey*

Adverbs: *in the house*

Subjunction: *if a man owns a donkey*

Coordination: *John and Mary are English or American*

Exercises

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

2. Compile and test the grammars `face/Face`*L* (available in course source files).

3. Write a concrete syntax of `Face` for some other resource language by adding a domain lexicon and a functor instantiation.

4. Add functions to `Face` and write their concrete syntax for at least some language.

5. Design your own domain grammar and implement it for some languages.