Grammatical Framework Tutorial

Aarne Ranta

Third GF Summer School, 2013

Day 1: getting familiar with GF (GF book chapters 1 to 4)

Day 2: the GF Resource Grammar Library (GF book chapters 5,9,10)

Day 3: best practices for GF applications

Day 1

Lesson 1: Introduction

Corresponds to book chapter 1

Outline

- the purpose of GF
- obtaining GF
- grammars vs. statistics
- the cost of grammars
- multilinguality
- semantic actions
- application grammars vs. resource grammars
- history of GF
- related work

GF, Grammatical Framework

A special-purpose programming language for writing grammars.

- support for the **complexities of natural languages**
- **engineering tools** for large projects involving many programmers
- **abstractions** and linguistic generalizations

GF works for single languages and **across multiple languages**

Some uses of GF

Multilingual translation systems

Language-based human-computer interaction

Creation of computational linguistic resources

Target users

Particularly designed for programmers and computer scientists

- GF is a typed functional programming language, like Haskell and ML

But also for linguists interested in multilinguality and generalizations

- a different approach to grammars

Web resources

GF website, http://www.grammaticalframework.org

- an updated reference manual with hyperlinks
- the full Resource Grammar Library API
- the source code for the GF system and the Resource Grammar Library
- executable binaries for the GF system

Download and install

Zero-click: http://cloud.grammaticalframework.org/gfse/

- grammar editor in the cloud

One-click: http://www.grammaticalframework.org/download/

- binary packages for Linux, Mac OS, and Windows (GF version 3.5)

Many many clicks: latest developer source code

```
darcs get --lazy http://www.grammaticalframework.org/ GF
cd GF
cabal install
```

Or use the GitHub mirror, https://github.com/GrammaticalFramework/GF/

Background fields

Computational linguistics: the **purpose**

Functional programming: the **method**

None of these is presupposed!

The GF book

Aarne Ranta, *Grammatical Framework: Programming with Multilingual Grammars*, CSLI Publications, Stanford, 2011, 340 pp.

Not necessary, but useful: the slides will roughly follow the book.

The role of grammars in language processing

How can we make computers process human language?

Symbolic approach: write processing rules, such as **grammars**

Statistical approach: learn from **data** by statistics and machine learning

The role of grammars in human language skills

Traditional school: learn the rules of grammar.

- explicit knowledge
- you know *why* you say in a certain way
- not how you learn your first language

More recent school: learn by hearing, reading, using.

- implicit knowledge
- first language

Grammars of programming languages

An important part of a **compiler**

The grammar is the **definition** of the programming language

Grammars of natural languages

A **research problem** - not a definition.

A theory formed by observing an already existing system.

The system is maybe not entirely coherent:

> All grammars leak.

(Sapir 1921).

Thus: either **incomplete** (not covering all of the language) or **over-generating** (covering expressions that in reality are "ungrammatical").

Still useful

For a human, a grammar provides a *shortcut*: its general rules replace a vast amount of training material.

Grammar usually improves the *quality* of the language produced by a human.

The same applies to computers: statistical models usually suffer from **sparse data**.

Sparse data

Inflection forms: French verbs have 51 forms easily defined by grammar; but maybe only a few of them appear in a corpus.

Word sequences: n-grams of words are sparse for large n.

- direct consequences for **long-distance dependencies**

Long-distance dependencies: agreement

Example: French adjectives agree in gender with nouns, which can be far apart.

English:

> *my father immediately became very worried*
> *my mother immediately became very worried*

French:

> *mon père est immédiatement devenu très inquiet*
> **ma** *mère est immédiatement* **devenue** *très* **inquiète**

Google translate for the latter (August 2010):

> *ma mère est immédiatement devenu très inquiet*

Long-distance dependencies: discontinuous constituents

Example: German compound verbs, e.g. *um+bringen* "kill" (literally, "bring around")

German

er bringt mich um

er bringt seinen besten Freund um

English

he kills me

he kills his best friend

Google translate for the latter (August 2011):

he brings to his best friend

Grammars vs. statistics

Don't guess if you know: if there's a grammar, use it.

- e.g. basic facts of inflection, agreement, and word order

All grammars leak: you may need more than just the grammar.

- the input may be out of grammar
- but **smoothing** may be used to guarantee **robustness**

Hybrid systems combine statistical and grammar-based methods.

The cost of grammars

Expensive to develop

- high skills
- a lot of work (PhD thesis or more)

Expensive to run

- parsing worse than linear (cubic for context-free, exponential for context-sensitive)

GF aims to tackle both of these problems.

Reducing the development cost: software engineering

Static type system detects many programming errors automatically

Module system supports division of labour

Functional programming enables powerful abstractions

Libraries enable building new grammars on earlier ones

Compilers convert GF grammars to other formats

Information extraction converts resources from other formats to GF

Improving run-time performance

Key: optimizing compilers, algorithm development, libraries

GF is equivalent to **PMCFG** (**Parallel Multiple Context-Free Grammars**)

Theoretically, parsing in GF and PMCFG is polynomial ($O(n^k)$) and the exponent k depends on the grammar.

Practical grammars are often linear: e.g. the Resource Grammar Library (RGL)

Multilinguality

A GF grammar can deal with several languages at the same time.

GF grammar = **abstract syntax + concrete syntaxes**

Abstract syntax: **trees** that capture semantically relevant structure

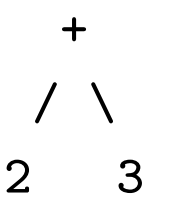
Concrete syntax relates trees with linear **strings**

Cf. compilers of programming languages

- programmers write strings
- the parser converts strings to trees
- the rest of the compiler manipulates trees

Varying the concrete syntax

Tree



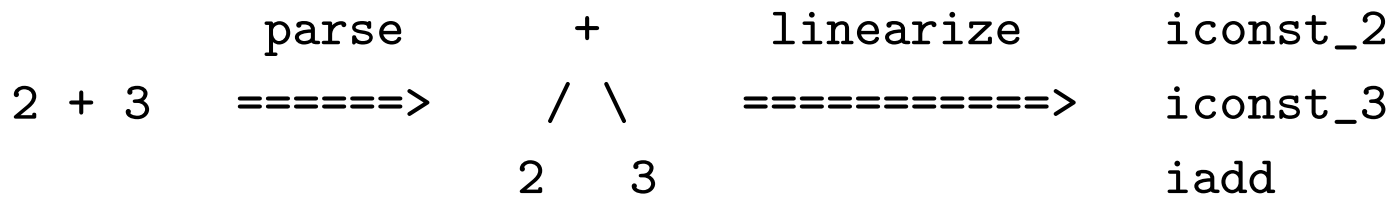
Strings

```
2 + 3                              -- infix (Java, C)
(+ 2 3)                            -- prefix (LISP)
iconst_2 ; iconst_3 ; iadd         -- postfix (Java Virtual Machine assembly)
0000 0101 0000 0110 0110 0000      -- postfix (Java Virtual Machine binary)
the sum of 2 and 3                 -- English
la somme de 2 et de 3              -- French
2:n ja 3:n summa                   -- Finnish
```

Compilation via abstract syntax

Parse Java string to tree

Linearize tree to JVM string



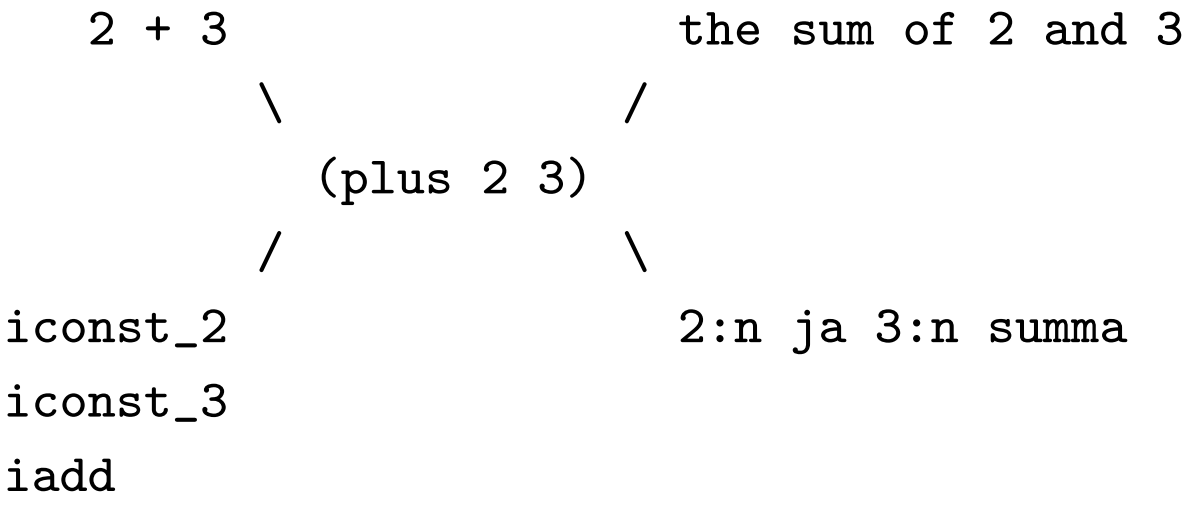
Compilers vs. GF

GF is **more powerful** (PMCFG, not just context-free)

GF is **reversible**: the same grammar defines both parsing and linearization

GF is **multilingual**: one abstract + several concrete

Compiling natural language



GF code for addition expressions

Abstract syntax

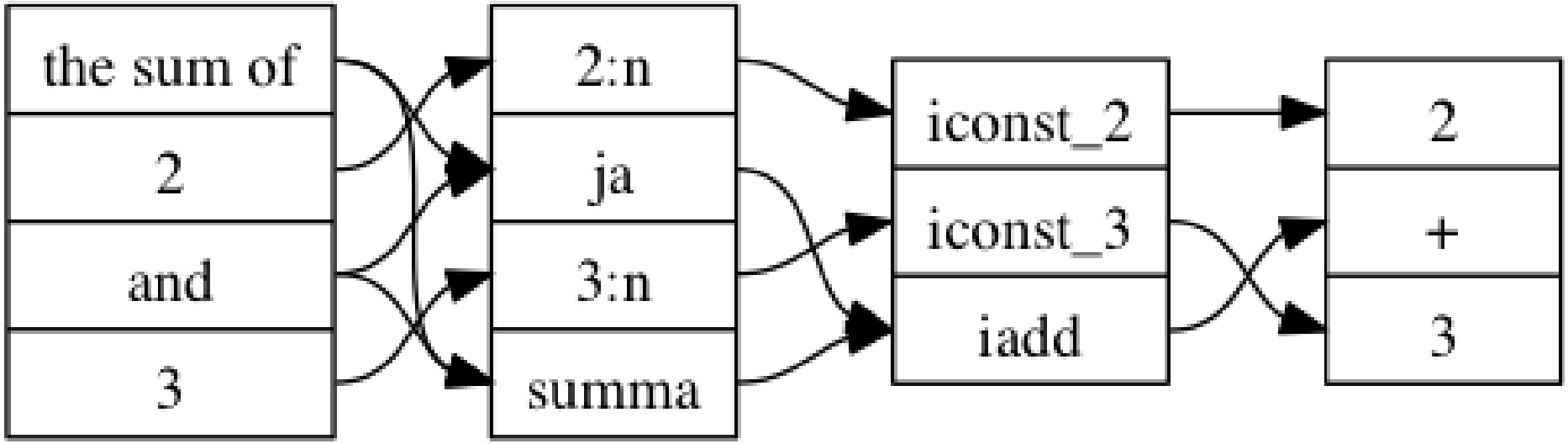
```
fun plus : Exp -> Exp -> Exp
```

Concrete syntaxes (Java, JVM, and English)

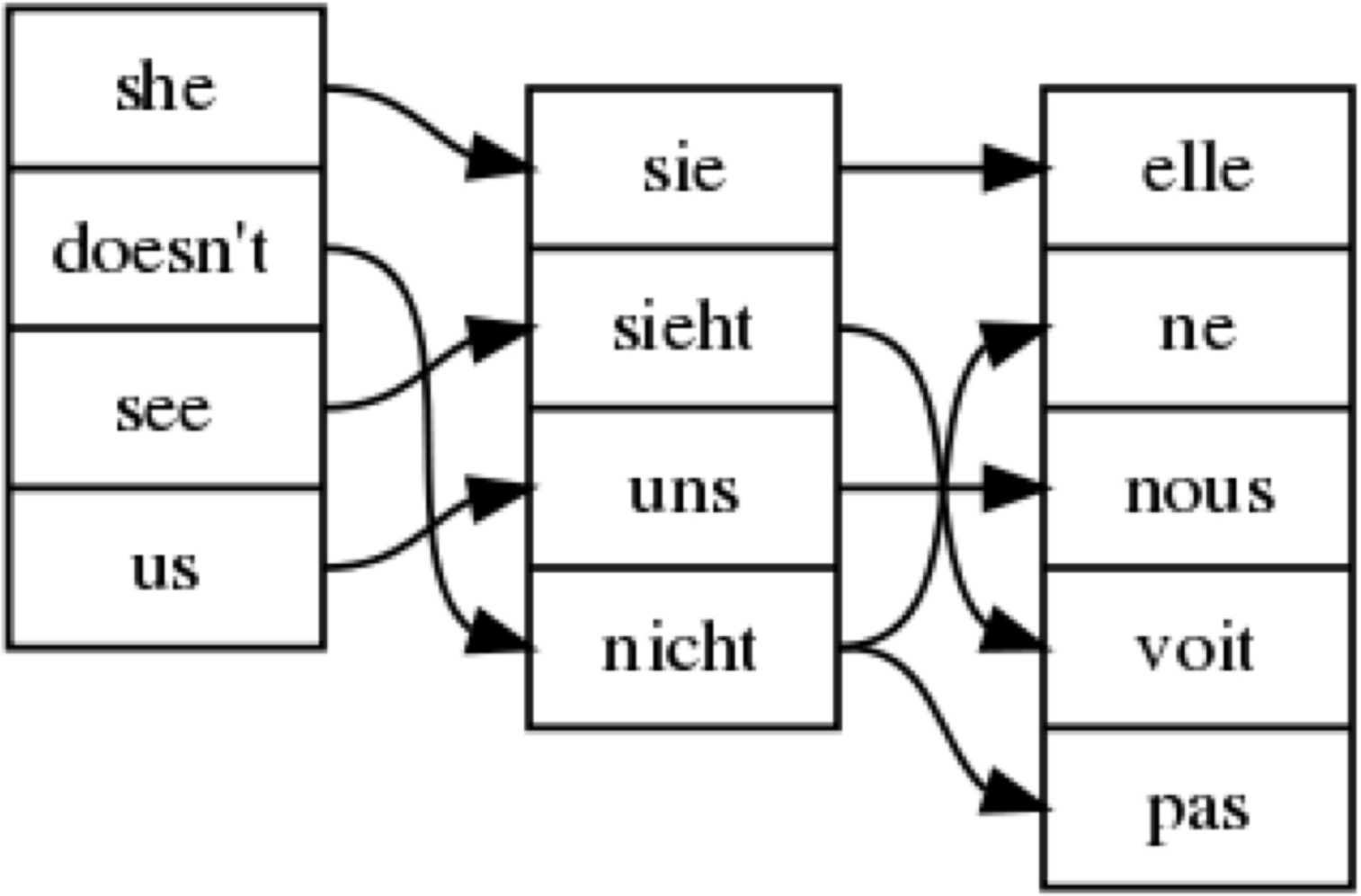
```
lin plus x y = x ++ "+" ++ y
lin plus x y = x ++ ";" ++ y ++ ";" ++ "iadd"
lin plus x y = "the sum of" ++ x ++ "and" ++ y
```

Details will follow later.

Word alignment via common abstract syntax



Word alignment: a more familiar case



Typical differences in concrete syntax

Words

Inflectional morphology

Word order

Discontinuous constituents

Morphology

English nouns have four forms (*house*, *houses*, *house's*, *houses'*)

French nouns have two forms (*maison*, *maisons*)

Finnish nouns have 26 forms (*talo*, *talon*, *taloa*, *taloksi*, *talona*, *talossa*, *talosta*, *taloon*, *talolla*, *talolta*, *talolle*, *talotta*, *talot*, *talojen*, *taloja*, *taloiksi*, *taloina*, *taloissa*, *taloista*, *taloihin*, *taloilla*, *taloilta*, *taloille*, *taloitta*, *taloine*, *taloin*) plus up to 3,000 more (*taloiksenikohan*,...)

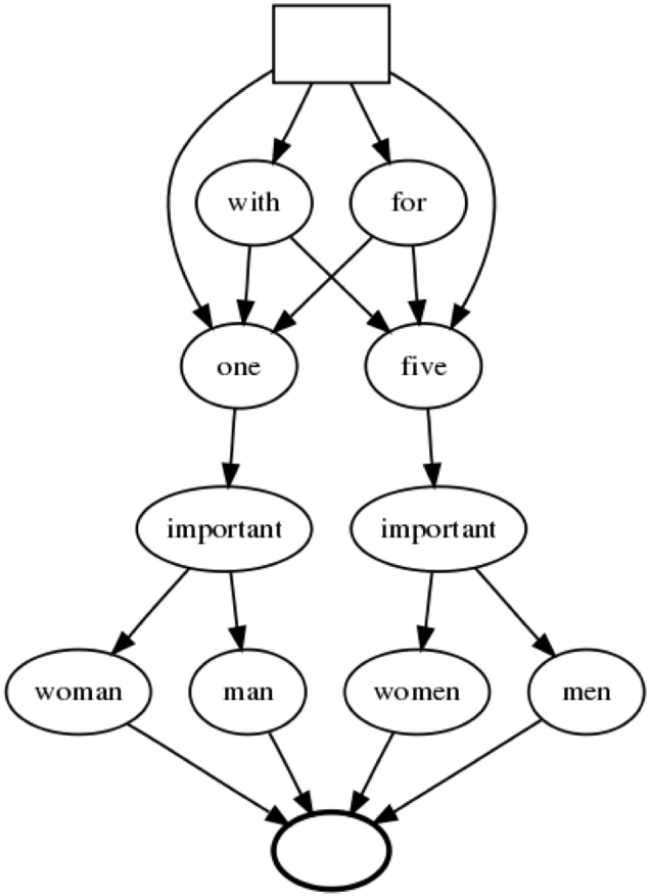
Agreement

The choice of a word form depends on other words occurring in the sentence.

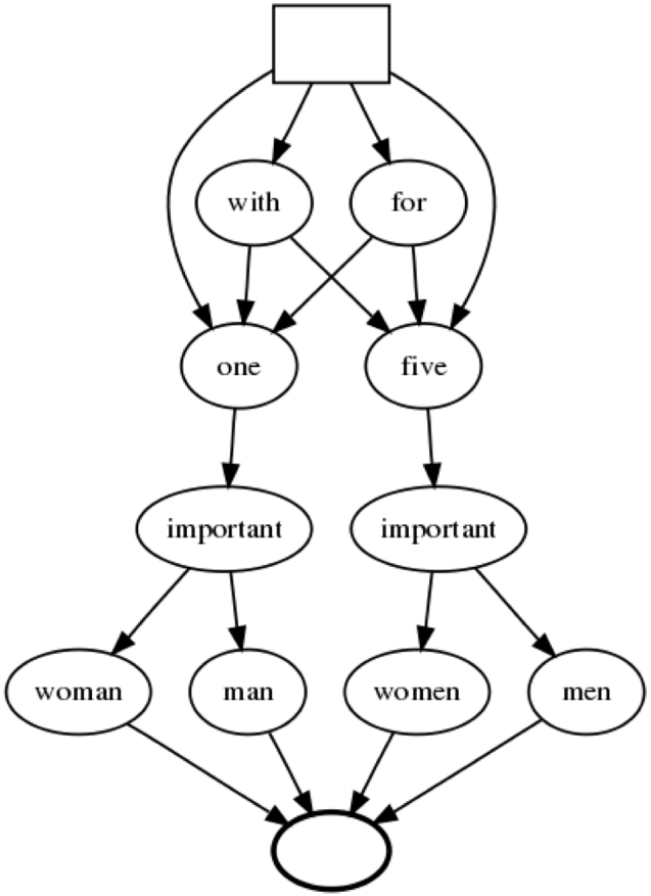
English nouns vary in number, in agreement to e.g. a numeral (*one man* vs. *five men*).

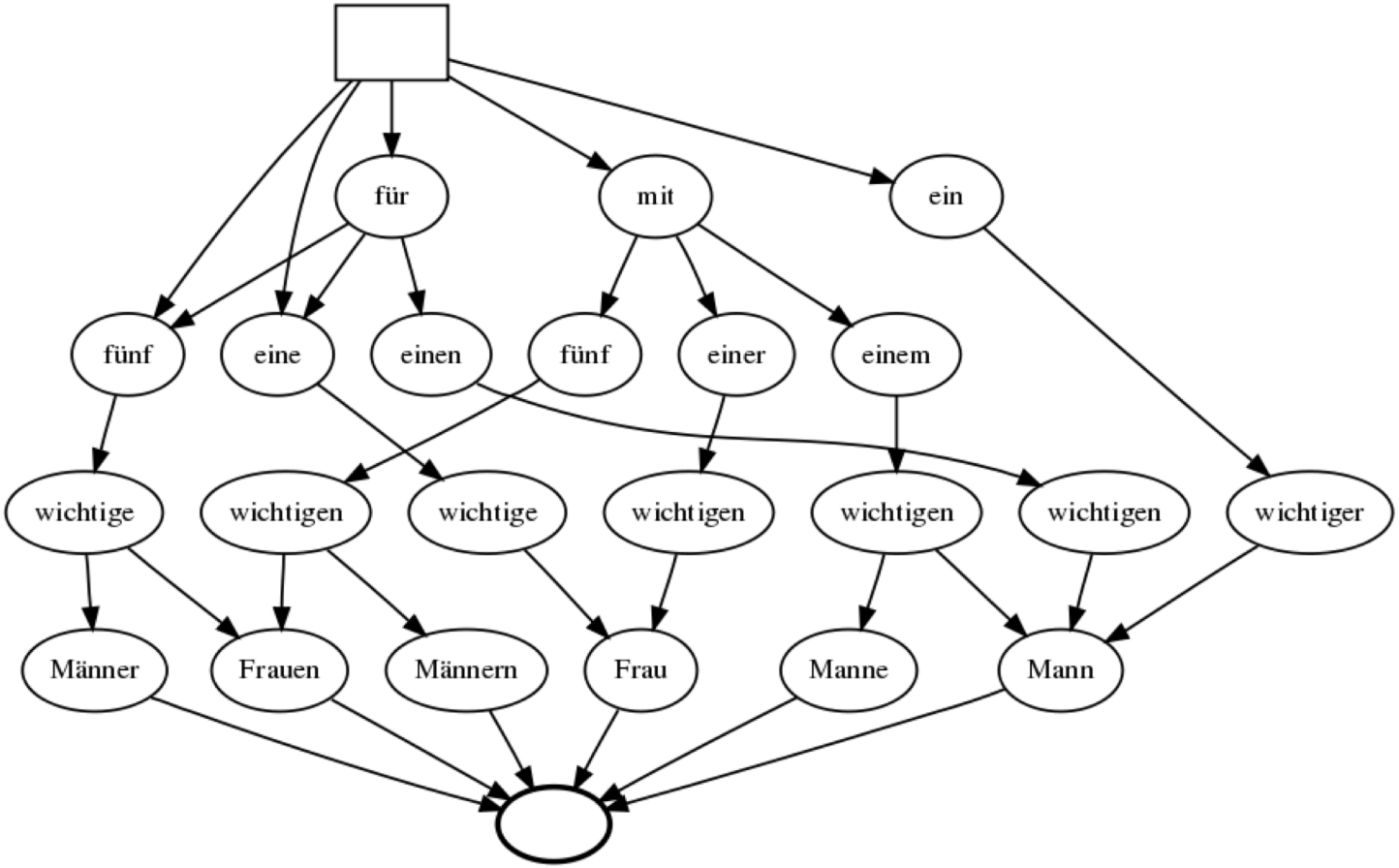
German nouns also vary in case, in agreement to e.g. a preposition, and adjectives agree in gender as well.

Noun phrases with prepositions in English

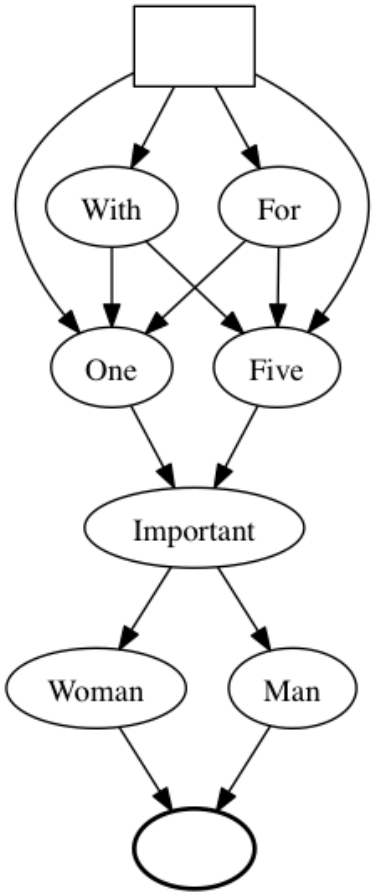


Noun phrases with prepositions in English and German

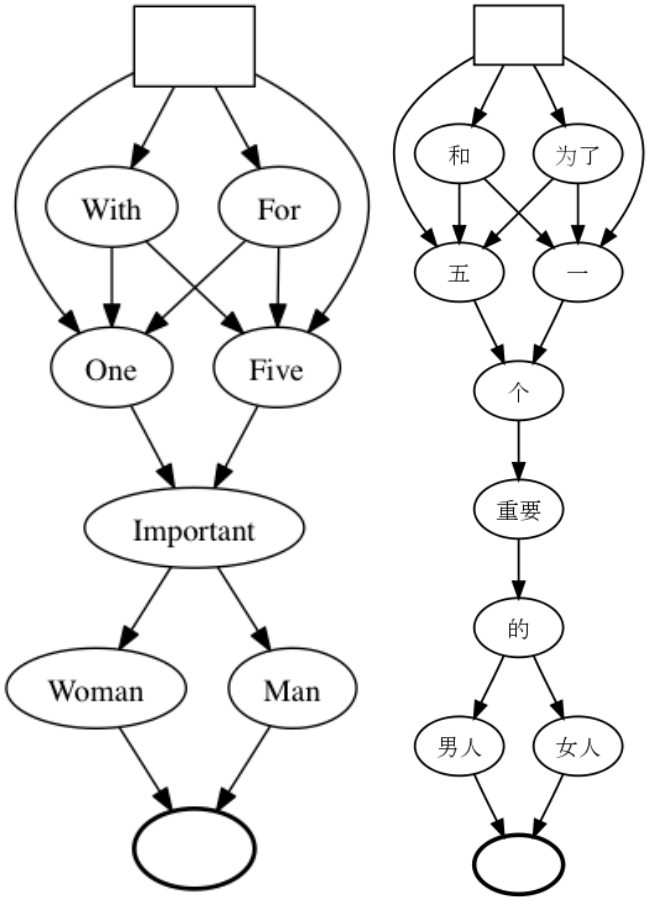




Noun phrases with prepositions in abstract syntax



Noun phrases with prepositions in abstract syntax and Chinese



Application grammars vs. resource grammars

Application grammar

- **semantic grammar**: abstract syntax reflects semantics
- built by domain expert
- typically: small, restricted domain

Resource grammar

- **syntactic grammar**: abstract syntax follows linguistic structure
- built by linguist
- unrestricted domain, "the whole language"

Two components of quality

Domain semantics: *odd* in French is *impair* rather than *bizarre*, *étrange*, *dépareillé*, in mathematics.

Linguistics: *impair* is inflected *impair*, *impairs*, *impaire*, *impaires* and appears after the noun (*nombre impair*).

Two kinds of trees

the sum of 2 and 3 is prime

Maths application grammar tree: predicate and its arguments

```
Prime (Sum (Num 2) (Num 3))
```

Resource grammar tree: sentence structure with tense etc.

```
UseCl
  (TTAnt TPres ASimul) PPos
  (PredVP
     (AdvNP
        (DetCN (DetQuant DefArt NumSg) (UseN sum_N))
           (PrepNP of_Prep
             (ConjNP and_Conj
               (BaseNP (UsePN n2_PN) (UsePN n3_PN)))))
     (UseComp (CompAP (PositA prime_A))))
```

Abstractions in semantic trees

The predicate `Prime` can be expressed with an adjective, as in English and German

x is prime

x ist unteilbar

or also with a noun, both in English and Finnish

x is a prime number

x on alkuluku

The resource trees are different, but the application tree is the same.

The GF Resource Grammar Library

Syntactic structure and morphology.

For 28 languages (in August 2013).

Designed to be usable by domain experts without linguistic training.

Has been used for mathematics, dialogue systems, tourist phrasebooks, museum object descriptions, pharmaceutical patents,...

Also extended for large-coverage translation with some languages

- *this is one of the goals of the summer school*

Early history of GF

Type-theoretical grammar (Ranta 1991, 1994)

- **Montague grammar** (1974) extended to **constructive type theory** (Martin-Löf 1984)
- implemented in ALF (Another Logical Framework), as a natural language interface to proof systems
- generation of six languages written in SML and later in Haskell

Grammatical Framework as a language of its own

- first implemented 1998 at Xerox Research, Grenoble
- generic grammar formalism, reversible grammars
- abstract syntax formalism = Logical Framework

Some milestones

1998: v 0.1, first release

2001: Resource Grammar Library started (English, Swedish, Russian)

2004: parsing complexity solved (Peter Ljunglöf)

2008: incremental parsing (Krasimir Angelov), web interfaces (Björn Bringert)

2009: v 3.0, separate run-time format (PGF)

2010: the MOLTO project, Multilingual On Line Translation

2013: robust statistical parsing, open-domain translation

Related work

GF is rooted in four research traditions:

- logic: type theory and logical frameworks
- linguistics: syntax, semantics, morphology
- compiler construction
- functional programming

Lesson 2: Basic concepts of multilingual grammars

Corresponds to book chapter 2

Outline

- BNF grammars and their use in the GF system
- GF functionalities: parsing, generation, translation
- abstract vs. concrete syntax
- limitations of the BNF format
- the module structure of GF
- free variation
- limitations of string-based GF
- character encoding

The BNF grammar format

BNF = Backus-Naur Format = context-free grammars

The most widely known grammar formalism.

In computer science: to specify programming languages

In linguistics: as pedagogical tool, but also e.g. speech recognition systems

Full GF is more powerful, but BNF is an interesting subset.

The GF system supports a BNF notation.

Example: foodEng.cf

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

BNF notation

Each line is a **labelled rule**.

The form of a rule is

```
Label . Category ::= Production
```

The production consists of

- **categories** (unquoted identifiers) a.k.a. **nonterminals**
- **tokens** (quoted strings) a.k.a. **terminals**

Labels and categories are **identifiers** (letter followed by letters, digits, underscores).

Parsing and linearization

The string

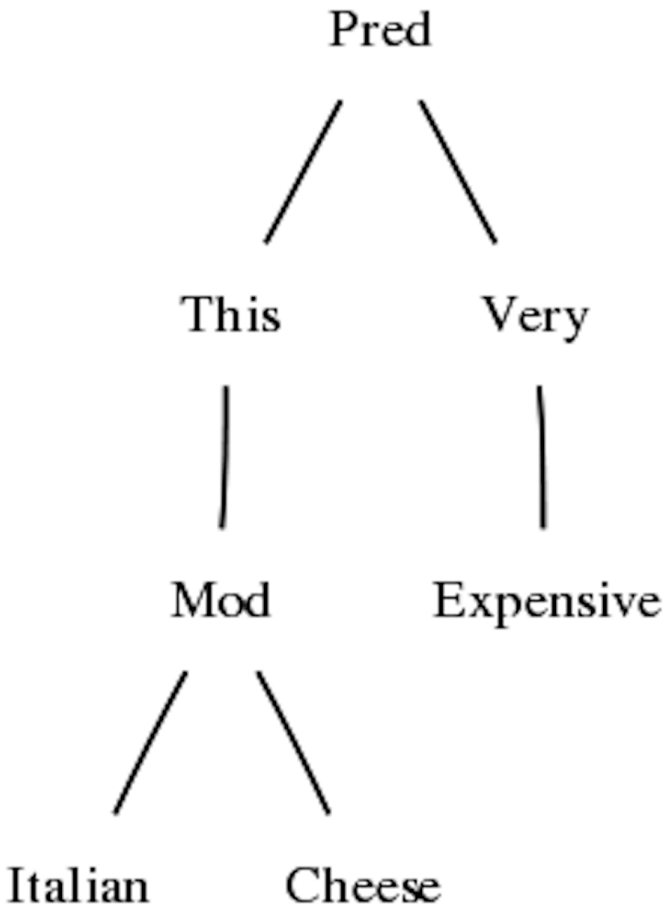
this Italian cheese is expensive

is **parsed** to the tree

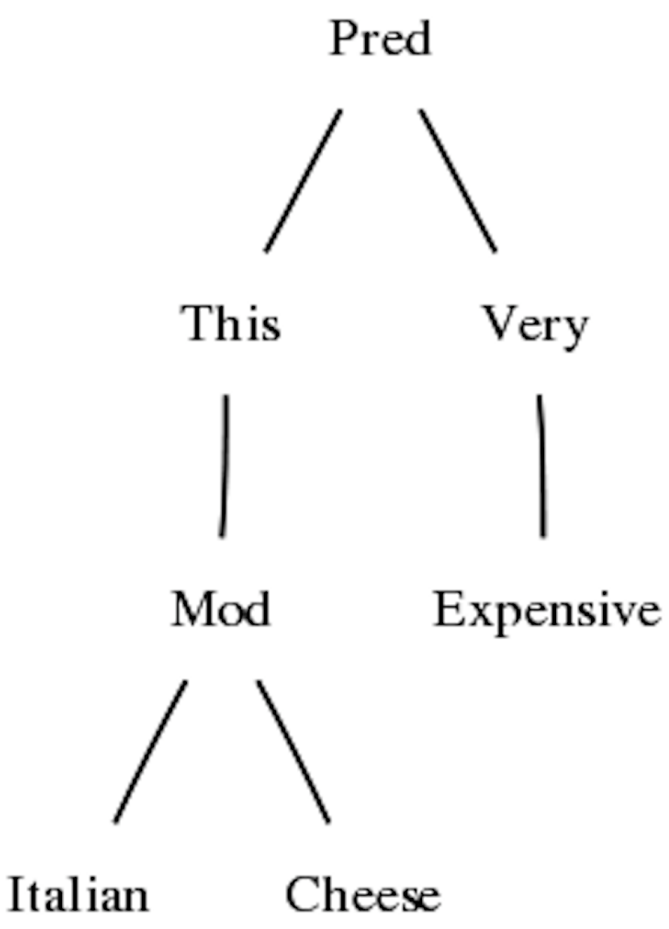
```
Pred (This (Mod Italian Cheese)) Expensive
```

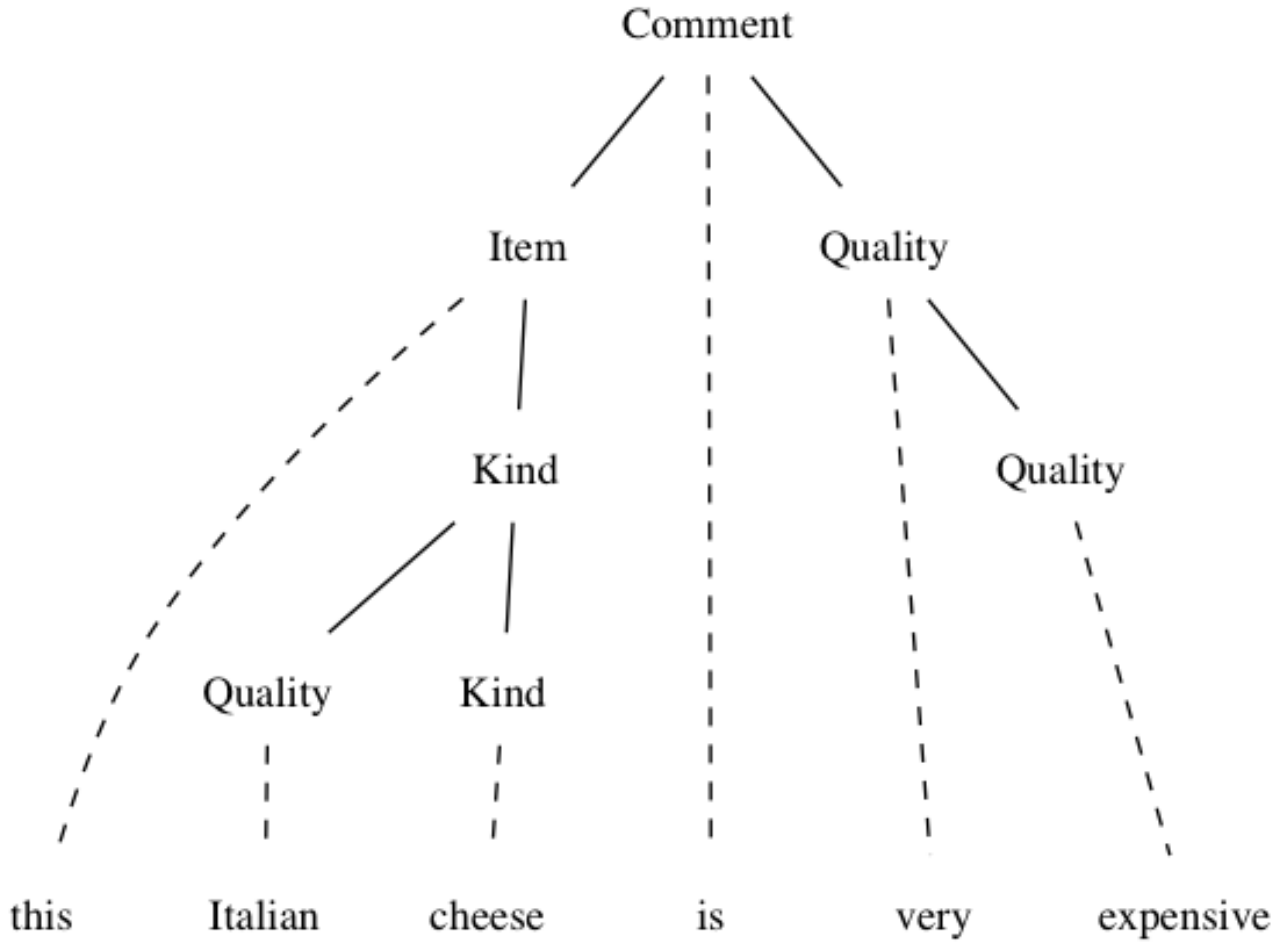
This tree is **linearized** to the string

Tree, graphically



Abstract tree vs. parse tree





Abstract tree vs. parse tree

Abstract tree:

- nodes and leaves are labels

Parse tree:

- nodes are categories
- labels are tokens

Given a GF grammar, an abstract tree determines a parse tree, but a parse tree can correspond to many abstract trees (how?).

Using the GF shell

```
$ gf

          *  *  *
       *            *
     *                *
    *
    *
    *         * * * * * *
    *         *         *
     *        * * * *  *
       *      *       *
          *  *  *

This is GF version 3.5.
License: see help -license.
Bug reports:
  http://code.google.com/p/grammatical-framework/issues/list

Languages:
>
```

GF commands

The first command you may want to give:

```
> help
```

More help on each command:

```
> help parse
```

Short names of commands:

```
> h p
```

Testing a grammar in the GF system

`import` = `i` the grammar file:

```
> import foodEng.cf
linking ... OK
```

`parse` = `p` a string in quotes:

```
> parse "this Italian cheese is very expensive"
Pred (This (Mod Italian Cheese)) (Very Expensive)
```

Use `help p` to see options.

Incremental parsing

Use the tab key to get next words and their completions:

```
> p "<TAB>
that this
```

Later in the sentence you get

```
> p "this cheese is <TAB>
Italian boring delicious expensive fresh very warm

> p "this cheese is e<TAB>
> p "this cheese is expensive
```

Linearization

`linearize` = `l`: from tree to string

```
> linearize Pred (This Fish) Delicious
this fish is delicious
```

Use `help l` to see options.

Random generation

`generate_random` = `gr`: a random tree

```
> generate_random
Pred (That Cheese) Italian
```

Use `help gr` to see options.

Pipes

Feed the output of one command as input to a next one

```
> generate_random | linearize
that expensive delicious boring wine is expensive
```

Trace option `-tr`, to see an intermediate step

```
> generate_random -tr | linearize
Pred (That Fish) Warm
that fish is warm
```

Generate many

Many random trees

```
> generate_random -number=100 | linearize
```

All trees (up to a certain depth)

```
> generate_trees | linearize
that cheese is boring
that cheese is delicious
that cheese is expensive
that cheese is fresh
...
```

Use `help` to see more options.

Separating abstract and concrete syntax

One rule in BNF,

```
Pred. Comment ::= Item "is" Quality
```

becomes two rules in GF,

```
fun Pred : Item -> Quality -> Comment ;
lin Pred item quality = item ++ "is" ++ quality ;
```

The `fun` is a **function** for building trees.

The `lin` is its **linearization** rule.

Function rules

```
fun Pred : Item -> Quality -> Comment
```

Value type: `Comment`

Argument types: `Item` and `Quality`

Linearization rules

```
lin Pred item quality = item ++ "is" ++ quality ;
```

The **variables** `item` and `quality`: linearizations of arguments.

Concatenation: `++`,

NB. one could use other variable names, e.g. `x` and `y`.

Sharing abstract syntax: word selection

Abstract syntax

```
fun Pred : Item -> Quality -> Comment ;
```

English linearization

```
lin Pred item quality = item ++ "is" ++ quality ;
```

Italian linearization

```
lin Pred item quality = item ++ "è" ++ quality ;
```

Sharing abstract syntax: word order

Abstract syntax

```
fun Mod : Quality -> Kind -> Kind
```

English linearization

```
lin Mod quality kind = quality ++ kind
```

Italian linearization

```
lin Mod quality kind = kind ++ quality
```

The module system

Abstract syntax modules: `fun` rules

Concrete syntax modules: `lin` rules

We also need rules for categories:

- `cat` in abstract syntax (to declare a category)
- `lincat` in concrete syntax (to define the type of its linearization)

The abstract syntax Food

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

The concrete syntax FoodEng

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

The concrete syntax FoodIta

```
concrete FoodIta of Food = {
  lincat
    Comment, Item, Kind, Quality = Str ;
  lin
    Pred item quality = item ++ "è" ++ quality ;
    This kind = "questo" ++ kind ;
    That kind = "quel" ++ kind ;
    Mod quality kind = kind ++ quality ;
    Wine = "vino" ;
    Cheese = "formaggio" ;
    Fish = "pesce" ;
    Very quality = "molto" ++ quality ;
    Fresh = "fresco" ;
    Warm = "caldo" ;
    Italian = "italiano" ;
    Expensive = "caro" ;
    Delicious = "delizioso" ;
    Boring = "noioso" ;
}
```

Import a multilingual grammar

Import any number of files with the same abstract syntax:

```
> import FoodEng.gf FoodIta.gf
- compiling Food.gf... wrote file Food.gfo
- compiling FoodEng.gf... wrote file FoodEng.gfo
- compiling FoodIta.gf... wrote file FoodIta.gfo
linking ... OK
Languages: FoodEng FoodIta
>
```

Separate compilation: each module gets its `.gfo` file (GF Object file).

Translating in GF

First import the grammars:

```
> import FoodEng.gf
> import FoodIta.gf
```

Then translate by piping:

```
> p -lang=Eng "this delicious wine is Italian" | l -lang=Ita
questo vino delizioso è italiano

> p -lang=Ita "quel pesce è molto caro" | l -lang=Eng
that fish is very expensive
```

Convention:

concrete = abstract + ISO 639-3 language code

Multilingual generation

```
> gr | l
that delicious warm fish is fresh
quel pesce caldo delizioso è fresco

> gr | l -treebank
Food: Pred (That (Mod Delicious (Mod Warm Fish))) Fresh
FoodEng: that delicious warm fish is fresh
FoodIta: quel pesce caldo delizioso è fresco
```

Translation quiz

A simple "end-user application" in the shell: `tq` = `translation_quiz`.

```
> tq -from=FoodEng -to=FoodIta
Welcome to GF Translation Quiz. The quiz is over when you
have done at least 10 examples with at least 75 % success.

* that wine is very boring
quel vino è molto noioso
Yes. Score 1/1

* that cheese is very warm
questo fromage è molto caldo
No, not questo fromage è molto caldo, but
  quel formaggio è molto caldo
Score 1/2
```

The structure of grammar modules

The main parts:

- **module header** with **module type** (`abstract` or `concrete of` A) and **module name** (`Food`)
- **module body** with **judgements**

Forms of judgement:

- abstract: `cat` and `fun`
- concrete: `lincat` and `cat`
- both: `flags`

Type checking

A concrete syntax is **complete** w.r.t. an abstract syntax, if it contains

- a `lincat` for each `cat`,
- a `lin` for each `fun`.

It is **well-typed** if

- all types used in `lincat` judgements are valid linearization types,
- all linearization rules define well-typed functions.

See the GF reference manual (in the book or on the web) for the details of type checking.

Groups of judgements

Judgements are terminated by semicolons.

Keywords can be shared:

$$\texttt{cat C ; D ;} \equiv \texttt{cat C ; cat D ;}$$

Right-hand-sides cn be shared:

$$\texttt{fun f, g : A ;} \equiv \texttt{fun f : A ; g : A ;}$$

Names

Each judgement introduces a **name**, which is the first identifier in the judgement.

Names are in **scope** in the entire module and can only be introduced once.

Comments

-- *after two dashes, anything until a newline*

{- *after left brace and dash, anything until dash and right brace* -}

How GF is more expressive than BNF

The separation of concrete and abstract syntax allows

- **permutation**: changing the order of constituents
- **suppression**: omitting constituents
- **reduplication**: repeating constituents

(Even more expressive power will be introduced in the next lesson.)

The copy language

Reduplication permits the non-context-free language $\{$`x x | x <- (a|b)*`$\}$.

```
abstract CopyAbs = {
  cat S ; AB ;
  fun s   : AB -> S ;
      end : AB ;
      a,b : AB -> AB ;
}
concrete Copy of CopyAbs = {
  lincat S, AB = Str ;
  lin s x = x ++ x ;
      end = [] ;          -- empty token list
      a x = "a" ++ x ;
      b x = "b" ++ x ;
}
```

Permutation

Needed in Food for the modification rule.

Increases **strong generative capacity**, (to define relations between trees and strings).

Reduplication also increases **weak generative capacity**, (to define sets of strings)

Exercise on permutation

Exercise. * Define the `reverse` operation as a GF grammar by using one abstract syntax and two concrete syntaxes. Translation between the concrete syntaxes should read a sequence of symbols and return them in the opposite order. For instance, `a b c` is translated `c b a`.

Suppression and metavariables

Pronoun as new primitive - not so interesting semantically

```
fun Pron : Item
lin Pron = "it"
```

Pronoun as a function that hides its interpretation

```
fun Pron : Item -> Item
lin Pron r = "it"
```

Parsing a pronoun

```
> parse "it is very expensive"
Pred (Pron ?) (Very Expensive)
```

The **metavariable** `?` is sent further to **anaphora resolution**

Metavariables in testing

To control random and exhaustive generation.

```
> generate_random Pred (This ?) Italian
```

generates only trees of the form *this X is Italian* where X is a random `Kind`.

Likewise, translation quiz can be given such a term as an argument, to create focused exercises.

Free variation

One abstract syntax, several linearizations:

```
lin Delicious = "delicious" | "exquisit" | "tasty"
```

NB. this is only valid on the abstraction level chosen for this semantic grammar.

Alternative ways to order a ticket

```
lin Ticket X Y =
  ((("I" ++ ("would like" | "want") ++ "to get" |
    ("may" | "can") ++ "I get" |
    "can you give me" |
    []) ++
      "a ticket") |
   []) ++
   "from" ++ X ++ "to" ++ Y ++
   ("please" | []) ;
```

Ambiguity

A string is **ambiguous** if it parses to more than one tree.

Example rule creating ambiguity:

```
fun With : Kind -> Kind -> Kind ;
lin With kind1 kind2 = kind1 ++ "with" ++ kind2 ;

> parse "fish with cheese with wine"
With (With Fish Cheese) Wine
With Fish (With Cheese Wine)
```

Avoiding ambiguity by design

One can force right associativity of `With`:

```
fun With : Kind -> ComplexKind -> ComplexKind
```

But be careful: the ambiguity is maybe real in natural language!

Irrelevant ambiguity

The same in English and Italian

```
lin With kind1 kind2 = kind1 ++ "with" ++ kind2 ;
lin With kind1 kind2 = kind1 ++ "con"  ++ kind2 ;
```

Now irrelevant for translation (but perhaps not for deeper semantics).

Ambiguity in translation

English

do you want this wine

Italian

vuoi questo vino (singular, familiar),

vuole questo vino (singular, polite),

volete questo vino (plural, familiar), and

vogliono questo vino (plural, polite).

Catalan numbers

Exercise. * How many trees are there for an expression of form *Kind with ... with Kind* for 2, 3, and 4 *with*'s? This series of numbers is known as the **Catalan numbers**, and it is a common pattern of counting in combinatorics; see `en.wikipedia.org/wiki/Catalan_number` for other examples.

We are not there yet

Add

```
fun Pizza : Kind
```

Everything works fine in English, but Italian gets

**questo pizza*

**pizza italiano*

instead of *questa pizza*, *pizza italiana*, because *pizza* is feminine and Italian has **gender agreement**.

Wanted: gender in Italian concrete syntax, without changing abstract syntax and English concrete syntax.

Character sets

English: ASCII

Many European languages: iso-latin-1

Most languages of the world: Unicode

GF uses internally 32-bit Unicode

Character encoding

Standard choice: UTF-8

The GF module body must then contain

```
flags coding = utf8 ;
```

as the default (for historical reasons) is iso-latin-1.

The generated gfo and pgf files are in UTF-8.

Example: Hindi

```
concrete FoodHin of Food = {
  flags coding = utf8 ;
  lincat Comment, Item, Kind, Quality = Str ;
  lin
    Pred item quality = item ++ quality ++ "है" ;
    This kind = "यह" ++ kind ;
    That kind = "वह" ++ kind ;
    Mod quality kind = quality ++ kind ;
    Wine = "मदिरा" ;
    Cheese = "पनीर" ;
    Fish = "मछली" ;
    Very quality = "अति" ++ quality ;
    Fresh = "ताज़ा" ;
    Warm = "गरम" ;
    Italian = "इटली" ;
    Expensive = "बहुमूल्य" ;
    Delicious = "स्वादिष्ट" ;
    Boring = "अरुचिकर" ;
}
```

A grammar-writing task

Exercise. Write a concrete syntax of `Food` for your favourite language. Use random generation to see how correct it becomes. Don't care about ungrammatical sentences due to gender and related things yet; just make a list of things that come out wrong.

Lesson 3: Parameters, tables, and records

Corresponds to chapters 3 and 4.

Outline

- morphological features and agreement
- parameters, tables, and records
- pattern matching
- functional programming in GF: operation definitions
- discontinuous constituents
- reusable resource modules
- smart paradigms
- operation overloading
- module extension and inheritance
- algebraic datatypes

The problem of morphological variation

Number of nouns in English

this wine is Italian

these wines are Italian

Context-free solution: split the relevant categories

```
Comment ::= Item_Sg "is"  Quality
Comment ::= Item_Pl "are" Quality
Item_Sg ::= "this" Kind_Sg
Item_Pl ::= "these" Kind_Pl
```

Explosion in categories

In Italian, both number and gender matter

```
Comment ::= Item_Sg_Masc "è"  Quality_Sg_Masc
Comment ::= Item_Sg_Fem  "è"  Quality_Sg_Fem
Comment ::= Item_Pl_Masc "sono"  Quality_Pl_Masc
Comment ::= Item_Pl_Fem  "sono"  Quality_Pl_Fem
```

Parametrized rules

Take out the suffixes as parameters, and introduce variables

```
Comment ::= Item(Sg,g) "è"    Quality(Sg,g)
Comment ::= Item(Pl,g) "sono" Quality(Pl,g)
```

This is the solution in **Definite Clause Grammars**.

In GF, we want parameters only in concrete syntax (since they depend on language).

Parameters and tables

New judgement form: parameter type definition

```
param Number = Sg | Pl
```

New form of type: table types

```
Number => Str
```

read, "table from numbers to strings".

Inflection tables

number	**form**
singular	*pizza*
plural	*pizze*

represented as the object

```
table {Sg => "pizza" ; Pl => "pizze"}
```

of type

```
Number => Str
```

Selection

To access a value in a table, use operator `!`

Example:

```
table {Sg => "pizza" ; Pl => "pizze"} ! Pl
```

computes to the string `"pizze"`.

Several parameters

Italian adjectives

```
Gender => Number => Str
```

where the type `Gender` is defined by

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

The inflection of the adjective *caldo* ("warm")

```
table {
  Masc => table {Sg => "caldo" ; Pl => "caldi"} ;
  Fem  => table {Sg => "calda" ; Pl => "calde"}
  }
```

Pattern matching

Variable `g`

```
table {g => table {Sg => "grave" ; Pl => "gravi"}}
```

Wildcard `_` (variable that is not used)

```
table {_ => table {Sg => "grave" ; Pl => "gravi"}}
```

Sugar for one-branch table

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

Variable vs. inherent features

Nouns in both English and Italian have both singular and plural forms.

Gender is different: Italian nouns *have* it, just one.

Cf. a dictionary entry for *pizza*:

> *pizza*, pl. *pizze*: n.f.

In other words: *pizza* is a feminine noun (n.f.) with the plural form (pl.) *pizze*.

For Italian nouns, number is **variable** and gender is **inherent**.

Agreement

For Italian adjectives, both number and gender are variable.

In modification, the gender of the adjective is determined by the noun.

Agreement, in general:

- X has inherent F
- Y has variable F
- Y receives its F from X

Records and record types

Italian nouns can be represented by **records**,

```
{s = table {Sg => "pizza" ; Pl => "pizze"} ; g = Fem}
```

This record has the **record type**

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

`s` and `g` are **labels**.

Field = label + type (in record types), or label + value (records)

Projection

To access the value in a record, the projection operator dot (`.`)

$$\{\texttt{s = "these" ; n = Pl}\}\texttt{.n} \Downarrow \texttt{Pl}$$

Thus together with selection

$$\{\texttt{s = table \{Sg => "zia" ; Pl => "zie"\} ; g = Fem}\}\texttt{.s ! Sg} \Downarrow \texttt{"zia"}$$

(Italian *zia*, "aunt").

The Foods grammar

```
abstract Foods = {
  flags startcat = Comment ;
  cat
    Comment ; Item ; Kind ; Quality ;
  fun
    Pred : Item -> Quality -> Comment ;
    This, That, These, Those : Kind -> Item ;
    Mod : Quality -> Kind -> Kind ;
    Wine, Cheese, Fish, Pizza : Kind ;
    Very : Quality -> Quality ;
    Fresh, Warm, Italian,
      Expensive, Delicious, Boring : Quality ;
}
```

We have just added `These`, `Those`, `Pizza`.

Linearization types for Foods

English:

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

Italian:

```
    Item    = {s : Str ; g : Gender ; n : Number} ;
    Kind    = {s : Number => Str ; g : Gender} ;
    Quality = {s : Gender => Number => Str} ;
```

It is a good habit to use records {`s : Str`} instead of plain `Str`. Then it is possible to add fields if needed.

English Foods: rules

Obvious:

```
lin
  This kind = {s = kind.s ! Sg ; n = Sg} ;
  Mod qual kind = {s = table {n => qual.s ++ kind.s ! n}} ;
```

Notice how `n` is passed in `Mod`.

Predication rule is slightly more complex:

```
lin Pred item qual = {
  s = item.s ++
      table {Sg => "is" ; Pl => "are"} ! item.n ++
      qual.s
  } ;
```

The middle term is in fact the verb *be*.

English Foods: words

Records and tables as dictated by the types

```
lin
  Wine = {s = table {Sg => "wine" ; Pl => "wines"}} ;
  Cheese = {s = table {Sg => "cheese" ; Pl => "cheeses"}} ;
  Fish = {s = \\_ => "fish"} ;

  Warm = {s = "warm"} ;
```

Can we make this more compact?

Functional programming in GF

The golden rule of functional programming:

> *Whenever you find yourself programming by copy and paste, define a function instead.*

Example

Instead of writing

```
Wine   = {s = table {Sg => "wine"   ; Pl => "wines"  }} ;
Cheese = {s = table {Sg => "cheese" ; Pl => "cheeses"}} ;
```

define a regular noun function `regNoun`, which factors out all shared parts:

```
Wine   = regNoun "wine" ;
Cheese = regNoun "cheese" ;
```

Operation definitions

Yet another judgement in concrete syntax,

$$\texttt{oper}\ f\ \texttt{:}\ t\ \texttt{=}\ e$$

Thus:

```
oper regNoun : Str -> {s : Number => Str} =
  \word -> {s = table {Sg => word ; Pl => word + "s"}} ;
```

using a **lambda abstract**

$$\backslash x\ \texttt{->}\ t$$

and **gluing** (+) of two tokens into one.

Gluing vs. concatenation

`"foo" + "bar"` $\Downarrow$ `"foobar"` (one token, `"foobar"`)

`"foo" ++ "bar"` $\Downarrow$ `"foo bar"` (list of two tokens, `"foo"`, `"bar"`)

Usually distinguished by a space, but this is relative to lexer.

Notations for functions

Application by juxtaposition: *f x*

Function types *A* -> *B* like in abstract syntax

Lambda with many arguments

$$\backslash x_1\text{, ..., } x_n \text{ -> } t \equiv \backslash x_1 \text{ -> ... } \backslash\ x_n \text{ -> } t$$

similarly to tables $\backslash\backslash x_1$, ..., x_n => *t*

The English Foods grammar: parameters and operations

```
concrete FoodsEng of Foods = {
  param
    Number = Sg | Pl ;
  oper
    det : Number -> Str ->
      {s : Number => Str} -> {s : Str ; n : Number} =
        \n,det,noun -> {s = det ++ noun.s ! n ; n = n} ;
    noun : Str -> Str -> {s : Number => Str} =
      \man,men -> {s = table {Sg => man ; Pl => men}} ;
    regNoun : Str -> {s : Number => Str} =
      \car -> noun car (car + "s") ;
    adj : Str -> {s : Str} =
      \cold -> {s = cold} ;
    copula : Number => Str =
      table {Sg => "is" ; Pl => "are"} ;
```

The English Foods grammar: linearization types

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

The English Foods grammar: linearizations

```
lin
  Pred item quality = {s = item.s ++ copula ! item.n ++ quality.s} ;
  This  = det Sg "this" ;
  That  = det Sg "that" ;
  These = det Pl "these" ;
  Those = det Pl "those" ;
  Mod quality kind = {s = \\n => quality.s ++ kind.s ! n} ;
  Wine = regNoun "wine" ;
  Cheese = regNoun "cheese" ;
  Fish = noun "fish" "fish" ;
  Pizza = regNoun "pizza" ;
  Very a = {s = "very" ++ a.s} ;
  Fresh = adj "fresh" ;
  Warm = adj "warm" ;
  Italian = adj "Italian" ;
```

```
Expensive = adj "expensive" ;
Delicious = adj "delicious" ;
Boring = adj "boring" ;
```

Testing inflection and operations in GF

Flags for linearization

```
Foods> linearize -table Wine
s Sg  : wine
s Pl  : wines
```

Compute concrete; you must retain oper's instead of compiling them away.

```
> import -retain FoodsEng.gf

> compute_concrete (regNoun "wine").s ! Pl
"wines"
```

Partial application

Function

```
fun This : Kind -> Item
```

Full application: expression of a ground type

```
lin This kind = det Sg "this" kind
```

Partial application: expression of a function type

```
lin This = det Sg "this"
```

Notice: you need to design the type of the oper as

```
Number -> Str -> {s : Number => Str} -> {s : Str ; n : Number}
```

rather than

```
{s : Number => Str} -> Number -> Str -> {s : Str ; n : Number}
```

Discontinuous constituents

Records with more strings than one.

Example: English verb phrase (VP) used both in declaratives and questions

John **is old**

is *John* **old**

Discontinuous in the VP.

It has a finite verb part (*is*) and a complement part (*old*).

A minimal grammar

```
cat
  S ; NP ; VP ;
fun
  Decl  : NP -> VP -> S ;
  Quest : NP -> VP -> S ;
  John  : NP ;
  IsOld : VP ;

lincat
  S, NP = Str ;
  VP = {verb,comp : Str} ;
lin
  Decl np vp  = np ++ vp.verb ++ vp.comp ;
  Quest np vp = vp.verb ++ np ++ vp.comp ;
  IsOld = {verb = "is" ; comp = "old"} ;
  John = "John" ;
```

Expressiveness of discontinuity

Exercise. * Write a grammar that generates the (non-context-free) language $a^n b^n c^n$, i.e. a language whose strings are the empty string, *a b c*, *a a b b c c*, etc, where there are always as many *a*'s as *b*'s and *c*'s.

Exercise. * Write a grammar that generates the (non-context-free) language $a^m b^n c^m d^n$, i.e. where the number of *a*'s and *c*'s is the same and so is the number of *b*'s and *d*'s. This language is well-known as a model of Swiss German, originally presented by Shieber in 1985 in his argument that Swiss German is not context-free.

Nonconcatenative morphology: Arabic

Semitic languages, e.g. Arabic: *kataba* has forms *kaAtib*, *yaktubu*, ...

Traditional analysis:

- word = **root** + **pattern**
- root = three consonants (**radicals**)
- pattern = function from root to string (notation: string with variables F, C, L for the radicals)

Example: *yaktubu* = *ktb* + *yaFCuLu*

Roots and patterns are discontinuous!

Datastructures for Arabic

Roots are records of strings.

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

A filling pattern is a record of strings filling the four slots in a root.

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

Builfing words by filling a pattern

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

Now we can!

Exercise. + Now we have defined a part of GF that is *complete* in the sense that pretty much any GF grammar can be written in it. So you can try and write a concrete syntax of `Foods` for any language you please, and make it correct.

Reusable resource modules

New module type: `resource`

Judgements contained: `param`, `oper`

Can be reused in different `concrete` modules by **opening**:

```
resource MorphoEng = {
  oper regNoun ...
  }

concrete FoodsEng of Foods = open MorphoEng in {
  lin Wine = regNoun "wine" ;
  }
```

The Prelude

A resource module useful for many languages, containing things like Boolean and string operations

```
resource Prelude = {
param
  Bool = True | False ;
oper
  init : Str -> Str = ...  -- all characters except the last
}
```

Local definitions

Syntax:

$$\texttt{let } c \texttt{ : } t \texttt{ = } d \texttt{ in } e$$

Example from `resIta`:

```
regAdj : Str -> Adjective = \nero ->
  let ner : Str = init nero
  in
  adjective nero (ner+"a") (ner+"i") (ner+"e") ;
```

Many-argument function types

Arguments of the same type can be shared, by using variables

```
(nero,nera,neri,nere : Str) -> Adjective
(_,_,_,_ : Str)             -> Adjective
```

are actually the same as

```
Str -> Str -> Str -> Str -> Adjective
```

The Italian Foods

There is a resource module

```
resource ResIta = open Prelude in {...}
```

and a concrete module using it

```
concrete FoodsIta of Foods = open ResIta in {...}
```

See the code in the book for the details:

- http://www.grammaticalframework.org/gf-book/examples/chapter3/ResIta.gf
- http://www.grammaticalframework.org/gf-book/examples/chapter3/FoodsIta.gf

Data abstraction

Task: implementing inflection paradigms suitable for reuse

Goals

- easy to use for all words, not just regular
- easy to change, e.g. add or remove forms

Solution:

- **abstract data types** instead of explicit recorda and table types
- **constructor operations** instead of explicit tables and records

Abstract types and constructors

1. Define the type `Noun`

```
oper Noun : Type = {s : Number => Str} ;
```

This is the only place where you see the explicit type!

2. Define a constructor operation, the **worst-case function** that covers all possible nouns

```
oper mkNoun : Str -> Str -> Noun = \x,y -> {
  s = table {
    Sg => x ;
    Pl => y
    }
  } ;
```

This is the only place where you see the record and the table!

Regular and irregular nouns

The regular oper is defined by using the constructor

```
oper regNoun : Str -> Noun =
  \word -> mkNoun word (word + "s") ;
```

Lexical items use either this or the worst-case oper:

```
lin House = regNoun "house" ;

lin Mouse = mkNoun "mouse" "mice" ;
```

Changing the internal representation

Suppose we want to add case (nominative and genitive) to English nouns:

```
param Case = Nom | Gen ;
oper Noun : Type = {s : Number => Case => Str} ;
```

The worst-case function must be redefined but has the same type as before:

```
oper mkNoun : Str -> Str -> Noun = \x,y -> {
  s = table {
    Sg => table {
      Nom => x ;
      Gen => x + "'s"
      } ;
    Pl => table {
      Nom => y ;
      Gen => case y of {
        _ + "s" => y + "'" ;
        _       => y + "'s"
      }
    }
  } ;
```

All other old definitions are still valid:

```
oper regNoun : Str -> Noun = \x -> mkNoun x (x + "s") ;

lin House = regNoun "house" ;

lin Mouse = mkNoun "mouse" "mice" ;
```

Case expressions and string matching

Inside `mkNoun`, we used a **case expression**,

```
case y of {
  _ + "s" => y + "'" ;
  _       => y + "'s"
  }
```

This performs **pattern matching** using **regular expressions**.

- `_ + "s"` is a pattern that matches any string followed by an *s*
- `_` is a pattern that matches anything
- the patterns are matched in the written order

String matching patterns

- the **disjunctive pattern**
  $P \mid Q$, matches everything that P or Q matches
- the **concatenation pattern**
  P + Q, matches any string of form st where P matches s and Q matches t
- the **variable pattern**
  `x`, matches anything and binds the variable `x` to this
- the **wildcard pattern**
  `_`, matches anything
- the **alias pattern** x`@`P, matches anything that P matches and binds the variable x to this
- the **string pattern**
  `"foo"`, matches just the string `"foo"`
- the **one-character pattern**
  `?`, matches any string whose lenght is exactly one (Unicode) character

Case expressions as table selections

Case expressions for parameter types are in fact syntactic sugar:

$$\texttt{case}\ e\ \texttt{of}\ \{\ldots\} \equiv \texttt{table}\ \{\ldots\}\ \texttt{!}\ e$$

Predictable variations

Between the completely regular *dog*-*dogs* and the completely irregular *mouse*-*mice*, we have

- nouns ending with *y*: *fly*-*flies*, except if a vowel precedes the *y*: *boy*-*boys*
- nouns ending with *s*, *ch*, and a number of other endings: *kiss*-*kisses*, *leech*-*leeches*

Smart paradigms

Use regular expressions to select the inflection

```
regNoun : Str -> Noun =
  \w -> case w of {
    _ + ("a" | "e" | "i" | "o") + "o"   => mkNoun w (w + "s") ;
    _ + ("s" | "x" | "sh" | "o" | "ch") => mkNoun w (w + "es") ;
    _ + ("a" | "e" | "o" | "u") + "y"   => mkNoun w (w + "s") ;
    x + "y"                             => mkNoun w (x + "ies") ;
    _                                   => mkNoun w (w + "s")
    }
```

German Umlaut

Exercise. Implement the German **Umlaut** operation on word stems. The operation has the type `Str -> Str`. It changes the vowel of the stressed stem syllable as follows: *a* to *ä*, *au* to *äu*, *o* to *ö*, and *u* to *ü*. You can assume that the operation only takes syllables as arguments. Test the operation to verify that it correctly changes *Arzt* to *Ärzt*, *Baum* to *Bäum*, *Topf* to *Töpf*, and *Kuh* to *Küh*.

Separating operation types and definitions

Instead of

```
oper regNoun : Str -> Noun =
  \s -> mkNoun s (s + "s") ;
```

one can have "two judgements"

```
oper regNoun : Str -> Noun ;
oper regNoun s = mkNoun s (s + "s") ;
```

and only display the first to the library user.

They can even appear in different modules (interface + instance, see Chapter 5).

Overloading of operations

Operations that have different types can be given the same name.

The type checker performs **overload resolution**.

The oper's have to be grouped together:

```
oper mkN = overload {
  mkN : (dog : Str) -> Noun = regNoun ;
  mkN : (mouse,mice : Str) -> Noun = mkNoun ;
}
```

This is used all the time in the GF Resource Grammar Library.

Lexicon with overloaded smart paradigms

We only need to know the part of speech and the characteristic forms.

We can guess the oper names: `mk`C where C is the category.

```
mkN "house"
mkN "mouse" "mouses"
mkN "leech"
mkV "walk"
mkV "write" "wrote" "written"
mkV "spy"
```

Such lexican can often be automatically extracted from different sources.

Module extension and inheritance

A module can **extend** another and **inherit** its contents.

This creates **module hierarchies**.

Example:

- base module: `Comments`
- two extensions: `Foods` and `Clothes`
- putting the extensions together: `Shopping`

The base module

General syntax and vocabulary for comments

```
abstract Comments = {
  flags startcat = Comment ;
  cat
    Comment ; Item ; Kind ; Quality ;
  fun
    Pred : Item -> Quality -> Comment ;
    This, That, These, Those : Kind -> Item ;
    Mod : Quality -> Kind -> Kind ;
    Very : Quality -> Quality ;
}
```

Comments on different kinds of things

```
abstract Foods = Comments ** {
  fun
    Wine, Cheese, Fish, Pizza : Kind ;
    Fresh, Warm, Italian,
      Expensive, Delicious, Boring : Quality ;
}

abstract Clothes = Comments ** {
  fun
    Shirt, Jacket : Kind ;
    Comfortable, Elegant : Quality ;
}
```

Both kinds of shopping

```
abstract Shopping = Foods, Clothes ;
```

Inheritance vs. opening

The general syntax of module headers

$$\textit{moduletype}\ \textit{name} = \textit{extends}\ \texttt{**}\ \textit{opens}\ \texttt{in}\ \textit{body}$$

Inheritance: same type of module, inherit contents

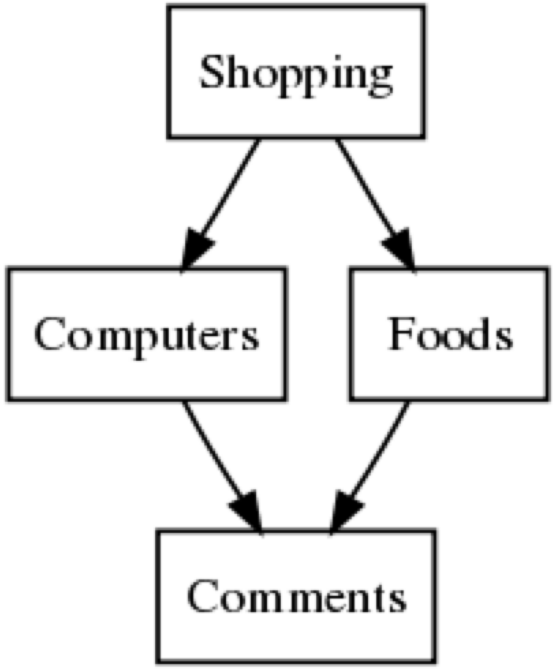
Opening: resource modules, just use its contents

Both cases enjoy separate compilation (to .gfo files).

Parallel inheritance in concrete

```
concrete FoodsIta of Comments = open ResIta in {...}

concrete FoodsIta of Foods = CommentsIta ** open ResIta in {...}

concrete FoodsIta of Foods = CommentsIta ** open ResIta in {...}

concrete ShoppingIta of Shopping = FoodsIta, ClothesIta ;
```

Visualising module dependencies



See GF shell command `dependency_graph`

Multiple inheritance

Inheritance of several modules, as in

```
abstract Shopping = Foods, Clothes ;
```

Diamond property: what happens when the same constant is inherited twice from an underlying module, via two extensions of it?

No problem in GF, since the intermediate modules may not change the inherited constant.

Restricted inheritance

```
abstract SmallShopping =
  Foods - [Wine],                          -- all except Wine
  Clothes [Kind,Quality,Shirt,Elegant] ;   -- only these
```

Redefining a constant

Possible only after restricted inheritance - but blocks later multiple inheritance.

```
abstract Comments = ...
abstract Foods = Comments ** {...}
abstract Clothes = Comments - [Very] ** {fun Very ...}
abstract Shopping = Foods, Clothes ; -- ERROR!
```

Rule: the same constant can be inherited twice only if it comes from the same source.

Qualified names

Problem: a constant appears in two opened modules.

```
resource Prelude = { ... init : ...}
resource Morpho  = { ... init : ...}
```

Solution: use **qualified name**

```
concrete C of A = open Prelude, Morpho in {
  lin c = Morpho.init (Prelude.init x)
  }
```

Alias qualifiers

One can also force the qualification of all names of an opened module, and introduce an **alias qualifier**

```
concrete C of A = open (P = Prelude), (Morpho = Morpho) in {
  lin c = init (P.init x)
  }
```

(Unfortunately, there is no qualification for inheritance.)

Need for hierarch datatypes for parameters

Example: German determiners have genders only in the singular.

form	**Sg Masc**	**Sg Fem**	**Sg Neutr**	**Pl**
Nom	*der*	*die*	*das*	*die*
Acc	*den*	*die*	*das*	*die*
Dat	*dem*	*der*	*dem*	*den*
Gen	*des*	*der*	*des*	*der*

Thus we don't want to write

```
param Gender = Masc | Fem | Neutr
param Case = Nom | Acc | Dat | Gen
lincat Det = Number => Gender => Case => Str
```

How to avoid this?

Algebraic datatypes

Parameter constructors with arguments (like datatypes in Haskell and ML), to create parameter hierarchies.

```
param DetForm = DSg Gender Case | DPl Case

lincat Det = DetForm => Str
```

getting 3*4+4 = 16 entries, instead of 3*4*2 = 24.

German definite article

Compactly, using pattern matching over algebraic datatypes:

```
oper artDef : DetForm => Str = table {
  DSg Masc Acc | DPl Dat => "den" ;
  DSg (Masc | Neutr) Dat => "dem" ;
  DSg (Masc | Neutr) Gen => "des" ;
  DSg Neutr _ => "das" ;
  DSg Fem (Nom | Acc) | DPl (Nom | Acc) => "die" ;
  _ => "der"
  }
```

form	**Sg Masc**	**Sg Fem**	**Sg Neutr**	**Pl**
Nom	*der*	*die*	*das*	*die*
Acc	*den*	*die*	*das*	*die*
Dat	*dem*	*der*	*dem*	*den*
Gen	*des*	*der*	*des*	*der*

Syncretism

Different parameters produce the same value, e.g. (SgFem Nom) and (Pl Nom) in German articles.

Not always clear to tell from parameter hierarchies, e.g. German Acc only matters in SgMasc.

Day 2

Lesson 4: Using the resource grammar library

Corresponds to Chapter 5

Outline

- the coverage of the Resource Grammar Library
- the structure and presentation of the library
- lexical vs. phrasal categories
- the resource grammar API (Application Programmer's Interface)
- reimplementing the `Foods` grammar and porting it to new languages
- interfaces, instances, and functors
- the division of labour between resource and application grammars
- functor overriding and compile-time transfer
- resource grammars as a linguistic ontology
- a tour of the resource grammar library
- browsing the library

The purpose of the library

The main grammar rules of different languages:

- the low-level details of morphology and syntax
- define grammatically correct language (not: semantically, pragmatically, stylistically...)

For application grammarians,

grammar checking becomes type checking

that is, whatever is type-correct in the resource grammar is also grammatically correct.

Required of application grammarians: just practical knowledge of the target language.

The library languages

Currently (August 2013) 28 languages: Afrikaans, Bulgarian, Catalan, Chinese, Danish, Dutch, English, Finnish, French, German, Greek (Modern), Hindi, Italian, Japanese, Latvian, Maltese, Nepali, Norwegian (bokmål), Persian, Polish, Punjabi, Romanian, Russian, Sindhi, Spanish, Swedish, Thai, Urdu.

Under construction for more languages: Amharic, Arabic, Estonian, Greek (Ancient), Hebrew, Latin, Lithuanian, Mongol, Swahili, Turkish, Yiddish.

40+ contributors, 3-6 person months per language.

See also: http://grammaticalframework.org/lib/doc/status.html

Lexical vs. phrasal rules

Linguistically:

- lexical: to define words and their properties
  - lexical categories
  - lexical rules

- phrasal (combinatorial, syntactic): phrases of arbitrary size
  - phrasal categories
  - phrasal rules

Formally: lexical = zero-place abstract syntax functions

Lexical across languages

What is one word in one language can be zero or more in another

- English *that*, Swedish *den där*, French *ce-là*
- English *the*, Swedish inflection form, Finnish nothing

Lexical in the library

For each language *L*

- `Syntax`*L* with phrasal rules - shared API
- `Paradigms`*L* with morphological paradigms - distinct API's

Closed vs. open lexical categories

Closed (structural words, function words) - given in `Syntax`

```
Det ;  -- determiner   e.g. "this"
AdA ;  -- adadjective  e.g. "very"
```

Open (content words) - constructed with `Paradigms`

```
N ;    -- noun         e.g. "cheese"
A ;    -- adjective    e.g. "warm"
```

Phrasal categories and rules

Five phrasal categories needed in the `Foods` grammar:

```
Utt ;  -- utterance          e.g. "is this pizza warm"
Cl ;   -- clause             e.g. "this pizza is warm"
NP ;   -- noun phrase        e.g. "this warm pizza"
CN ;   -- common noun        e.g. "warm pizza"
AP ;   -- adjectival phrase  e.g. "very warm"
```

Syntactic combinations

The syntactic combinations we need are the following:

```
mkUtt : Cl  -> Utt ;        -- e.g. "is this pizza warm"
mkCl  : NP  -> AP -> Cl ;   -- e.g. "this pizza is warm"
mkNP  : Det -> CN -> NP ;   -- e.g. "this pizza"
mkCN  : AP  -> CN -> CN ;   -- e.g. "warm pizza"
mkAP  : AdA -> AP -> AP ;   -- e.g. "very warm"
```

Principle: every way of combining categories is available as an overloaded oper.

Lexical insertion rules

Form phrases from single words:

```
mkCN : N -> CN ;
mkAP : A -> AP ;
```

Naming convention

Opers producing *C* have name `mk`*C*, e.g. `mkNP`

Not always possible (why?)

Words: *word*`_`*C*, e.g. `wine_N`

Other things: *description*C, e.g. `presentTense`

Example

these very warm pizzas are Italian

Resource grammar expression:

```
mkUtt
  (mkCl
    (mkNP these_Det
       (mkCN (mkAP very_AdA (mkAP warm_A)) (mkCN pizza_N)))
    (mkAP italian_A))
```

Application grammar syntax

```
Pred (These (Mod (Very Warm) Pizza)) Italian
```

The resource API: categories

Category	Explanation	Example
`Utt`	utterance (sentence, question,...)	*who are you*
`Cl`	clause, 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*
`Det`	determiner	*this*
`A`	one-place adjective	*warm*
`N`	common noun	*house*

The resource API: combination rules

Function	Type	Example
`mkUtt`	`Cl -> Utt`	*John is very old*
`mkCl`	`NP -> AP -> Cl`	*John is very old*
`mkNP`	`Det -> 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*

The resource API: structural words

Function	Type	In English
`this_Det`	`Det`	*this*
`that_Det`	`Det`	*that*
`these_Det`	`Det`	*this*
`those_Det`	`Det`	*that*
`very_AdA`	`AdA`	*very*

The resource API: lexical paradigms

English:

Function	Type
mkN	(dog : Str) -> N
mkN	(man,men : Str) -> N
mkA	(cold : Str) -> A

Italian:

Function	Type
mkN	(vino : Str) -> N
mkA	(caro : Str) -> A

German:

Function	Type
Gender	Type
masculine	Gender
feminine	Gender
neuter	Gender
mkN	(Stufe : Str) -> N
mkN	(Bild,Bilder : Str) -> Gender -> N
mkA	(klein : Str) -> A
mkA	(gut,besser,beste : Str) -> A

Finnish:

Function	Type
mkN	(talo : Str) -> N
mkA	(hieno : Str) -> A

English and Italian Foods with the resource

See the next two slides as an animation!

```
concrete FoodsEng of Foods = open SyntaxEng, ParadigmsEng in {
  lincat
    Comment = Utt ;
    Item = NP ;
    Kind = CN ;
    Quality = AP ;
  lin
    Pred item quality = mkUtt (mkCl item quality) ;
    This kind = mkNP this_Det kind ;
    That kind = mkNP that_Det kind ;
    These kind = mkNP these_Det kind ;
    Those kind = mkNP those_Det kind ;
    Mod quality kind = mkCN quality kind ;
    Very quality = mkAP very_AdA quality ;
    Wine = mkCN (mkN "wine") ;
    Pizza = mkCN (mkN "pizza") ;
    Cheese = mkCN (mkN "cheese") ;
    Fish = mkCN (mkN "fish" "fish") ;
    Fresh = mkAP (mkA "fresh") ;
    Warm = mkAP (mkA "warm") ;
    Italian = mkAP (mkA "Italian") ;
    Expensive = mkAP (mkA "expensive") ;
    Delicious = mkAP (mkA "delicious") ;
    Boring = mkAP (mkA "boring") ;
}
```

```
concrete FoodsIta of Foods = open SyntaxIta, ParadigmsIta in {
  lincat
    Comment = Utt ;
    Item = NP ;
    Kind = CN ;
    Quality = AP ;
  lin
    Pred item quality = mkUtt (mkCl item quality) ;
    This kind = mkNP this_Det kind ;
    That kind = mkNP that_Det kind ;
    These kind = mkNP these_Det kind ;
    Those kind = mkNP those_Det kind ;
    Mod quality kind = mkCN quality kind ;
    Very quality = mkAP very_AdA quality ;
    Wine = mkCN (mkN "vino") ;
    Pizza = mkCN (mkN "pizza") ;
    Cheese = mkCN (mkN "formaggio") ;
    Fish = mkCN (mkN "pesce") ;
    Fresh = mkAP (mkA "fresco") ;
    Warm = mkAP (mkA "caldo") ;
    Italian = mkAP (mkA "italiano") ;
    Expensive = mkAP (mkA "caro") ;
    Delicious = mkAP (mkA "delizioso") ;
    Boring = mkAP (mkA "noioso") ;
}
```

Now everyone can do it!

Exercise. Write a concrete syntax of `Foods` for some other language included in the resource library. You can compare the results with the hand-written grammars presented earlier in this tutorial.

Functor implementation of multilingual grammars

As shown in the animation, a new language is added easily:

1. copy the concrete syntax of an already given language
2. change the words (strings and inflection paradigms)

But how to avoid this copy and paste?

Answer: write a **functor**: a function that produces a module.

Functor = **parametrized module**

Instance and interface

Interface: declarations of oper's

```
interface LexFoods = open Syntax in {
  oper
    wine_N : N ;
    pizza_N : N ;
    -- etc
}
```

Instance: definitions of oper's

```
instance LexFoodsEng of LexFoods = open SyntaxEng, ParadigmsEng in {
  oper
    wine_N = mkN "wine" ;
    pizza_N = mkN "pizza" ;
    -- etc
}
```

The Foods functor

```
incomplete concrete FoodsI of Foods = open Syntax, LexFoods in {
  lincat
    Comment = Utt ; Item = NP ; Kind = CN ; Quality = AP ;
  lin
    Pred item quality = mkUtt (mkCl item quality) ;
    This kind = mkNP this_Det kind ;
    That kind = mkNP that_Det kind ;
    These kind = mkNP these_Det kind ;
    Those kind = mkNP those_Det kind ;
    Mod quality kind = mkCN quality kind ;
    Very quality = mkAP very_AdA quality ;
    Wine = mkCN wine_N ;
    Pizza = mkCN pizza_N ;
    Cheese = mkCN cheese_N ;
    Fish = mkCN fish_N ;
    Fresh = mkAP fresh_A ;
    Warm = mkAP warm_A ;
    Italian = mkAP italian_A ;
    Expensive = mkAP expensive_A ;
    Delicious = mkAP delicious_A ;
    Boring = mkAP boring_A ;
}
```

The Syntax interface and instances

Given in the resource grammar library:

```
interface Syntax
instance SyntaxEng of Syntax
instance SyntaxIta of Syntax
...
```

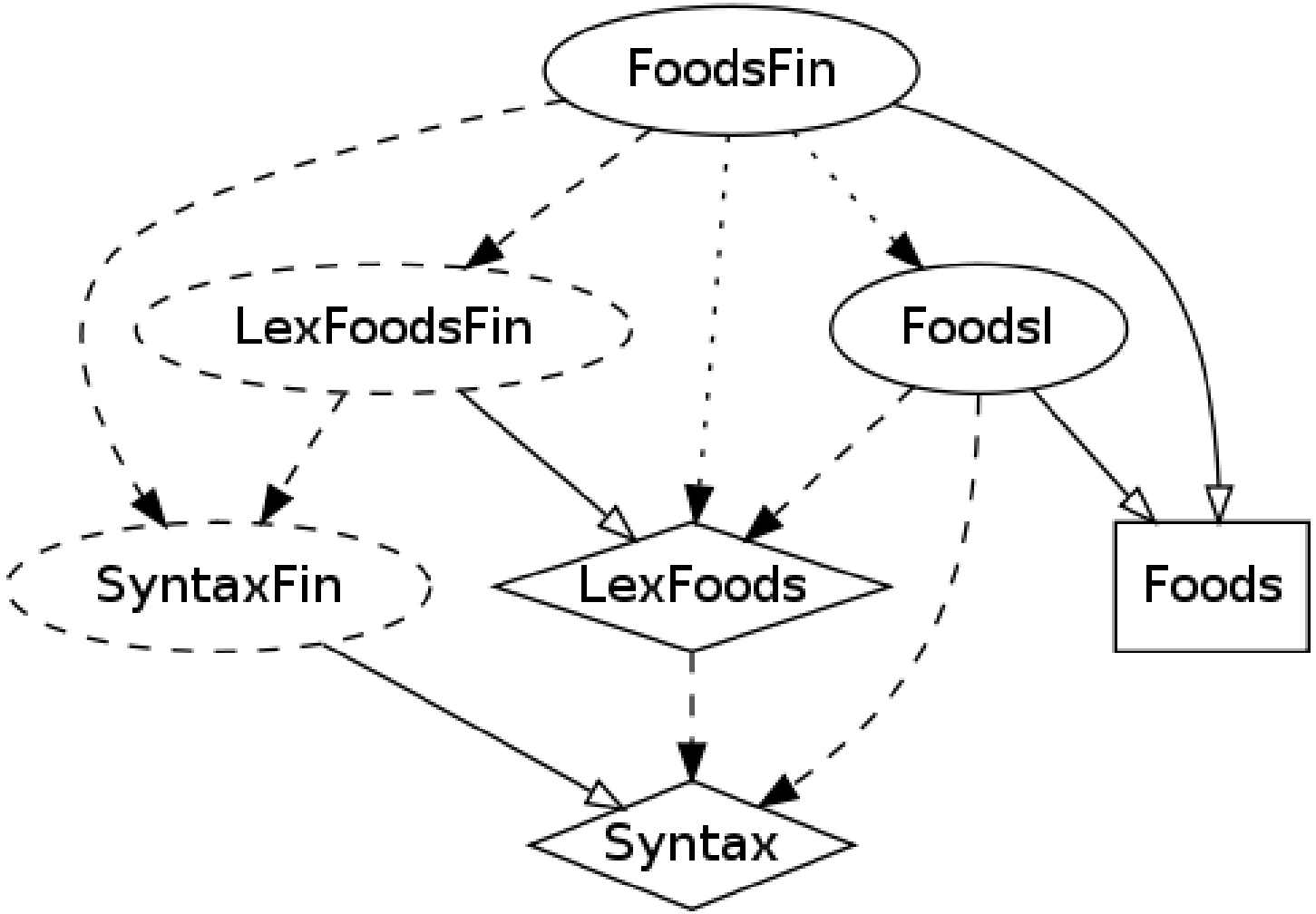
Functor instantiations

```
concrete FoodsEng of Foods = FoodsI with
  (Syntax = SyntaxEng),
  (LexFoods = LexFoodsEng)

concrete FoodsIta of Foods = FoodsI with
  (Syntax = SyntaxIta),
  (LexFoods = LexFoodsIta)
```

All you need to add a new language

```
instance LexFoodsGer of LexFoods = open SyntaxGer, ParadigmsGer in {
  oper
    wine_N = mkN "Wein" ;
    pizza_N = mkN "Pizza" "Pizzen" feminine ;
    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" ;
}

concrete FoodsGer of Foods = FoodsI with
  (Syntax = SyntaxGer),
  (LexFoods = LexFoodsGer)
```

A design pattern for multilingual grammars



When does a functor work

A functor using the resource `Syntax` interface works when the concepts are expressed by using the same structures in all languages.

When they don't, their linearizations can be expressed by parameters in the domain lexicon interface.

Problem: when new languages are added, more things may have to be moved to the interface.

Overriding a functor

We can use restricted inheritance.

Contrived example:

```
concrete FoodsEng of Foods = FoodsI - [Pizza] with
  (Syntax = SyntaxEng),
  (LexFoods = LexFoodsEng) **
    open SyntaxEng, ParadigmsEng in {

  lin Pizza = mkCN (mkA "Italian") (mkN "pie") ;
}
```

Transfer in translation

Translation by **transfer**: change the syntactic structure.

John likes Mary -> Maria piace a Giovanni (Italian)

```
mkCl x like_V2 y  -> mkCl y piacere_V2 x
```

What is your name? -> Wie heißt du? (German)

```
mkQCl what_IP (mkNP you_Pron name_N) -> mkQCl how_IAdv (mkCl you_Pron hei
```

How old are you? -> Quanti anni hai? (Italian)

```
mkQCl (how_IAdA old_A) you_Pron -> mkQCl (how_many_IDet year_N) have_V2 y
```

Compile-time transfer

The system can still be built with **interlingua**, the application abtract syntax.

Only the way the resource grammar is used varies

```
fun Like : Person -> Item -> Comment

lin Like x y = mkCl x like_V2 y

lin Like x y = mkCl y piacere_V2 x
```

Three ways:

- no functor, separate concrete syntaxes (more copy and paste)
- functor with this rule as a parameter (can be unstable)
- functor with an exception (usually the most practical solution)

The resource grammar as a linguistic ontology

Domain: **linguistic objects** - nouns, verbs, predication, modification...

Cf. grammar books:

- chapters on nouns, verbs, sentence formation...
- sections on gender, cases, agreement...

The chapters become the resource grammar abstract syntax.

The sections become the concrete syntax.

Advantages of common linguistic ontology

Foreign language learners are helped by familiar conceots.

Resource grammar implementation can exploit previous work.

- abstract syntax gives a "check list"
- concrete syntax code may be reusable via opening, inheritance, functors

Application grammar writing gets easy

- learn the library for one language, learn it for all
- maybe use functors

Language typology gets precise concepts to talk about the similarities and differences of languages.

A tour of the resource API

Go through the resource API in

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

Flattening of constructions

Core resource: minimal set of rules, maximally general.

This creates deep resource grammar trees.

Predication in core resource

```
mkCl : NP -> VP -> Cl
mkVP : VPSlash -> NP -> VP
mkVPSlash : V2 -> VPSlash
```

V2-predication in the API

```
mkCl : NP -> V2 -> NP -> Cl

mkCl x v y = mkCl x (mkVP (mkVPSlash v) y)
```

Tense and polarity

A **clause** is a sentence with variable tense and polarity

```
mkS : (Temp) -> (Pol) -> Cl -> S
```

Default sentence: present tense, positive polarity

```
mkS : Cl -> S
```

N.B. Parentheses in API documentation are used for optionality of arguments

- implemented by overloading
- not significant for GF compiler

Full tense and polarity inflection

Form	**English**	**Italian**
Sim Pres Pos	*I sleep*	*dormo*
Sim Pres Neg	*I don't sleep*	*non dormo*
Sim Past Pos	*I slept*	*dormivo*
Sim Past Neg	*I didn't sleep*	*non dormivo*
Sim Fut Pos	*I will sleep*	*dormirò*
Sim Fut Neg	*I won't sleep*	*non dormirò*
Sim Cond Pos	*I would sleep*	*dormirei*
Sim Cond Neg	*I wouldn't sleep*	*non dormirei*
Ant Pres Pos	*I have slept*	*ho dormito*
Ant Pres Neg	*I haven't slept*	*non ho dormito*
Ant Past Pos	*I had slept*	*avevo dormito*
Ant Past Neg	*I hadn't slept*	*non avevo dormito*
Ant Fut Pos	*I will have slept*	*avrò dormito*
Ant Fut Neg	*I won't have slept*	*non avrò dormito*
Ant Cond Pos	*I would have slept*	*avrei dormito*
Ant Cond Neg	*I wouldn't have slept*	*non avrei dormito*

Trying out tenses

Lang*L*.gf: the resource grammar as concrete syntax (rather than resource)

```
> import alltenses/LangEng.gfo
> parse -cat=Cl "I sleep" | linearize -table
```

The library path

The compiled libraries will be in some directory, such as `/usr/local/lib/gf` in Unix-like environments.

GF uses the environment variable `GF_LIB_PATH` to locate this library. To see if it is set, try

```
$ echo $GF_LIB_PATH
```

If the variable is not set, do

```
$ export GF_LIB_PATH=/usr/local/lib/gf
```

in Bash, maybe `setenv` in another shell.

Even better: put this in your `.bashrc`.

Two versions of libraries

In two directories of `GF_LIB_PATH`

- `alltenses`, containing all tense forms
- `present`, containing only the present tense forms, infinitives, and participles.

The same modules, but in two versions, e.g. `alltenses/SyntaxEng.gfo` and `present/SyntaxEng.gfo`.

Produced from the same source.

Testing the library

If you have `GF_LIB_PATH` set correctly,

```
> import -retain present/ParadigmsGer.gfo
> compute_concrete -table mkN "Farbe"
```

The path flag

List of directories to search GF source and object files

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

Either in the source file or the import command.

```
> import -path=.:present FoodsREng.gf
```

Browsing the library

The core concrete syntax

```
Lang> p "this wine is good"
PhrUtt NoPConj (UttS (UseCl (TTAnt TPres ASimul) PPos
  (PredVP (DetCN (DetQuant this_Quant NumSg) (UseN wine_N))
  (UseComp (CompAP (PositA good_A))))) NoVoc
```

The derived resource module

```
> i -retain alltenses/TryEng.gfo
> cc -all mkUtt (mkCl this_NP (mkA "cool"))
this is cool
```

Learn to use the resource library

Exercise. Construct some expressions and their translations by parsing and linearizing in the resource library:

- *is this wine good*
- *I (don't) like this wine*, *do you like this wine*
- *I want wine*, *I would like to have wine*
- *I know that this wine is bad*
- *can you give me wine*
- *give me some wine*
- *two apples and wine*
- *he says that this wine is good*
- *she asked which wine was the best*

Exercise. + Extend the `Foods` grammar with new forms of expressions, corresponding to the examples of the previous exercise. First extend

the abstract syntax, then implement it by using the resource grammar and a functor. You can also try to minimize the size of the abstract syntax by using free variation. For instance, *I would like to have X*, *give me X*, *can you give me X*, and *X* can be variant expressions for one and the same order.

Your own resource grammar application project

Exercise. + Design a small grammar that can be used for controlling an MP3 player. The grammar should be able to recognize commands such as *play this song*, with the following variations:

- objects: *song*, *artist*
- modifiers: *this*, *the next*, *the previous*
- verbs with complements: *play*, *remove*
- verbs without complements: *stop*, *pause*

The implementation goes in the following phases:

1. abstract syntax
2. functor and lexicon interface
3. lexicon instance for the first language
4. functor instantiation for the first language
5. lexicon instance for the second language
6. functor instantiation for the second language
7. ...

Lesson 5: resource grammar internals

Corresponds to Chapter 9.

Outline

- The key categories and rules
- Morphology-syntax interface
- Examples and variations in English, Italian, French, Finnish, Swedish, German, Hindi

Don't worry if the details of this lecture feel difficult! Syntax **is** *difficult and this is why resource grammars are so useful!*

Syntax in the resource grammar

"Linguistic ontology": syntactic structures common to languages

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

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

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

A lot of work, easy to get wrong!

The key categories 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*

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")

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.

The miniature resource

```
abstract Grammar = {
  flags startcat = S ;
  cat
    S ; Cl ; NP ; VP ; AP ; CN ;
    Det ; N ; A ; V ; V2 ; AdA ;
    Tense ; Pol ;
    Conj ;
  data
    UseCl   : Tense -> Pol -> Cl -> S ;
    PredVP  : NP -> VP -> Cl ;
    ComplV2 : V2 -> NP -> VP ;
    DetCN   : Det -> CN -> NP ;
    ModCN   : AP -> CN -> CN ;
    CompAP  : AP -> VP ;
    AdAP    : AdA -> AP -> AP ;
    ConjS   : Conj -> S  -> S  -> S ;
    ConjNP  : Conj -> NP -> NP -> NP ;
    UseV    : V -> VP ;
    UseN    : N -> CN ;
    UseA    : A -> AP ;
```

```
    a_Det, the_Det, every_Det : Det ;
    this_Det, these_Det : Det ;
    that_Det, those_Det : Det ;
    i_NP, she_NP, we_NP : NP ;
    very_AdA : AdA ;
    and_Conj, or_Conj : Conj ;

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

abstract Test = Grammar ** {
fun
  man_N, woman_N, house_N, tree_N : N ;
  big_A, small_A, green_A : A ;
  walk_V, arrive_V : V ;
  love_V2, please_V2 : V2 ;
}
```

Exercise

Implement the miniature resource grammar for your language.

Add some paradigms for nouns and adjectives, as well as an instance of the miniature `Syntax` interface.

Implement a `Foods` grammar by using the miniature resource rather than the standard one.

Code to get started:

- http://www.grammaticalframework.org/gf-book/examples/chapter9/

Also in http://cloud.grammaticalframework.org/gfse/ as "Miniresource"