Grammatical Framework: Programming with Multilingual Grammars

Aarne Ranta

Slides for the GF book (CSLI, Stanford, 2011)

CSLI Studies in Computational Linguistics

Grammatical Framework

Programming with Multilingual Grammars

Aarne Ranta

Preamble

GF, Grammatical Framework

A special-purpose programming language for writing grammars.

Supports the complexities found in different natural languages.

Provides engineering tools for large projects involving many programmers.

Supports abstractions and linguistic generalizations,

Works for single languages and across multiple languages.

Uses of GF

Multilingual translation systems

Language-based human-computer interaction

Creation of computational linguistic resources.

Target users

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

Particularly designed for programmers and computer scientists.

But also for linguists interested in multilinguality and generalizations.

Web resources

The book's website, http://www.grammaticalframework.org/gf-book

- most of the code examples contained in the book
- clickable links to the URL's given in the book
- these slides

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://www.grammaticalframework.org/demos/gfse/

- grammar editor in the cloud

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

- binary packages for Linux, Mac OS, and Windows (GF version 3.2)

Many many clicks: latest developer source code

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

What the book is about

Computer programs that process natural language.

Main focus: multilingual systems

- translation systems
- applications with one language at a time
  - natural language interfaces
  - spoken dialogue systems
  - language learning aids
  - software localization

Background fields

Computational linguistics: the **purpose**

Functional programming: the **method**

None of these is presupposed!

What is in the slides

Tutorial: Chapters 1 to 6

Larger grammars and applications: Chapters 7 to 10

Not covered: reference manual

Suggestions of projects

One-week level: the "Foods" grammar (Chapter 3) for a new language

Three-week level: one of

- the "miniature resource" (Chapter 9)
- a query system with an embedded grammar (Chapter 7)
- an interface to a formal language (Chapter 8)

Ten-week level: resource grammar morphology and lexicon (Chapter 10) for a new language

Twenty-week level: complete resource grammar (Chapter 10) (with a written report, equivalent to a Mater's thesis)

For "reserved" languages: extend the library coverage or build applications.

Chapter 1: Introduction

Outline

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

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

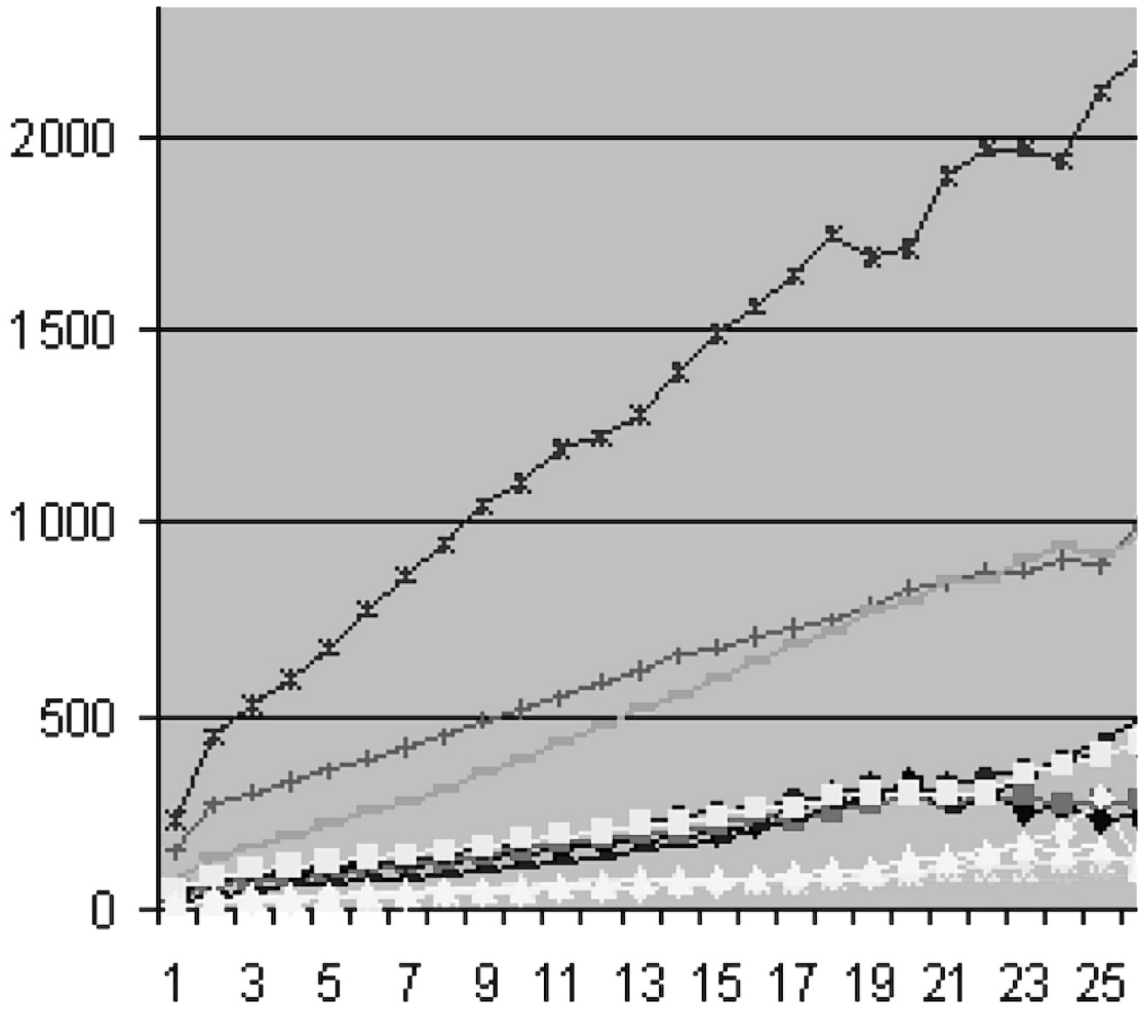
Key: optimizing compilers, algorithm development, libraries

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)

Linear parsing in the RGL



GF resource grammar parsing speed msec/token (by Krasimir Angelov). Slowest: Finnish, German, Italian; fastest: Scandinavian languages, English.

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

```
   +
  / \
 2   3
```

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

```
           parse      +      linearize      iconst_2
  2 + 3   ======>    / \    ===========>    iconst_3
                    2   3                   iadd
```

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

```
   2 + 3                  the sum of 2 and 3
         \              /
           (plus 2 3)
         /              \
iconst_2                  2:n ja 3:n summa
iconst_3
iadd
```

GF code for addition expressions

Abstract syntax

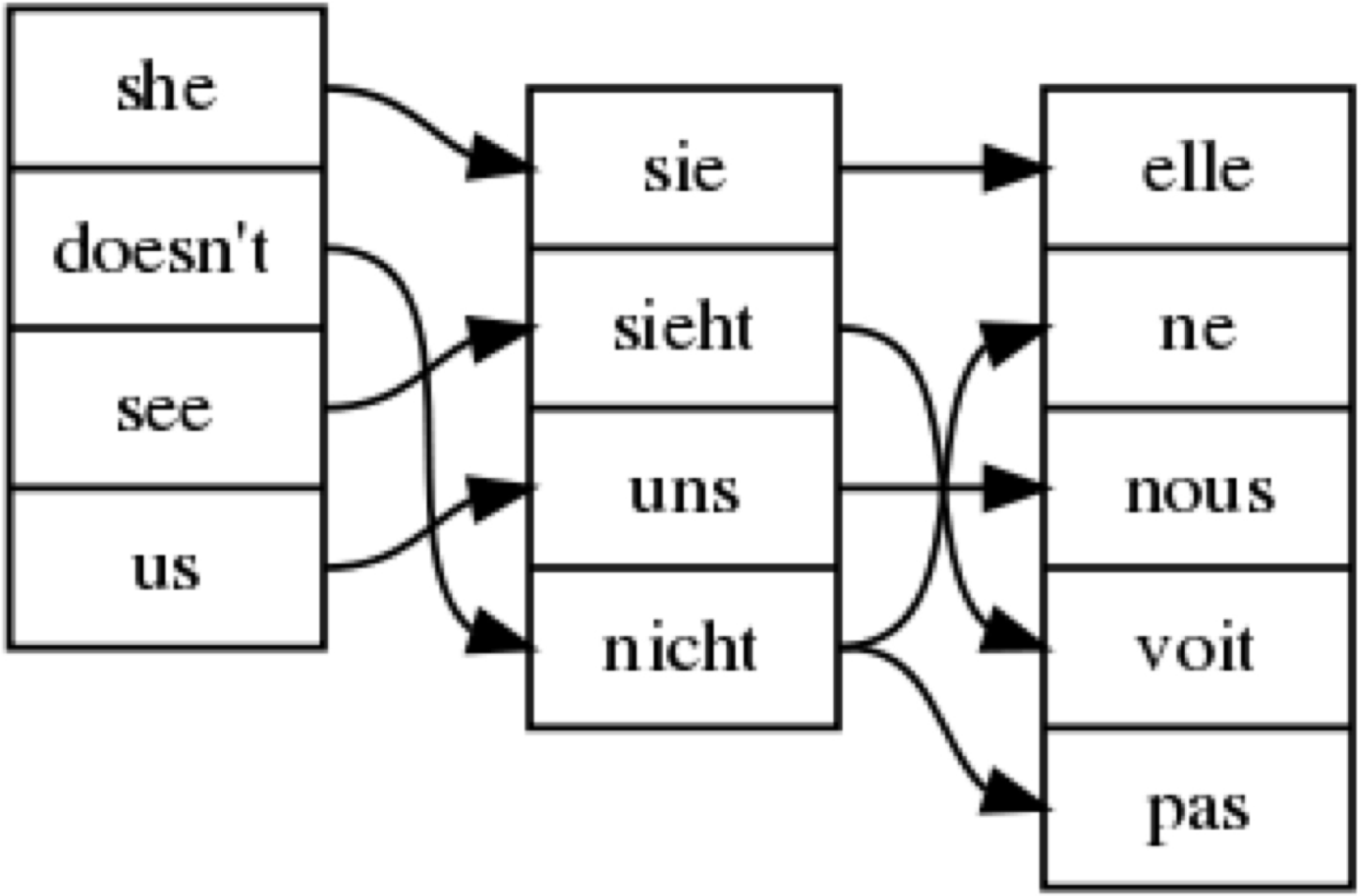
```
fun plus : Exp -> Exp -> Exp
```

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 ++ y
```

Details will follow later.

Word alignment via common abstract syntax



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*,...)

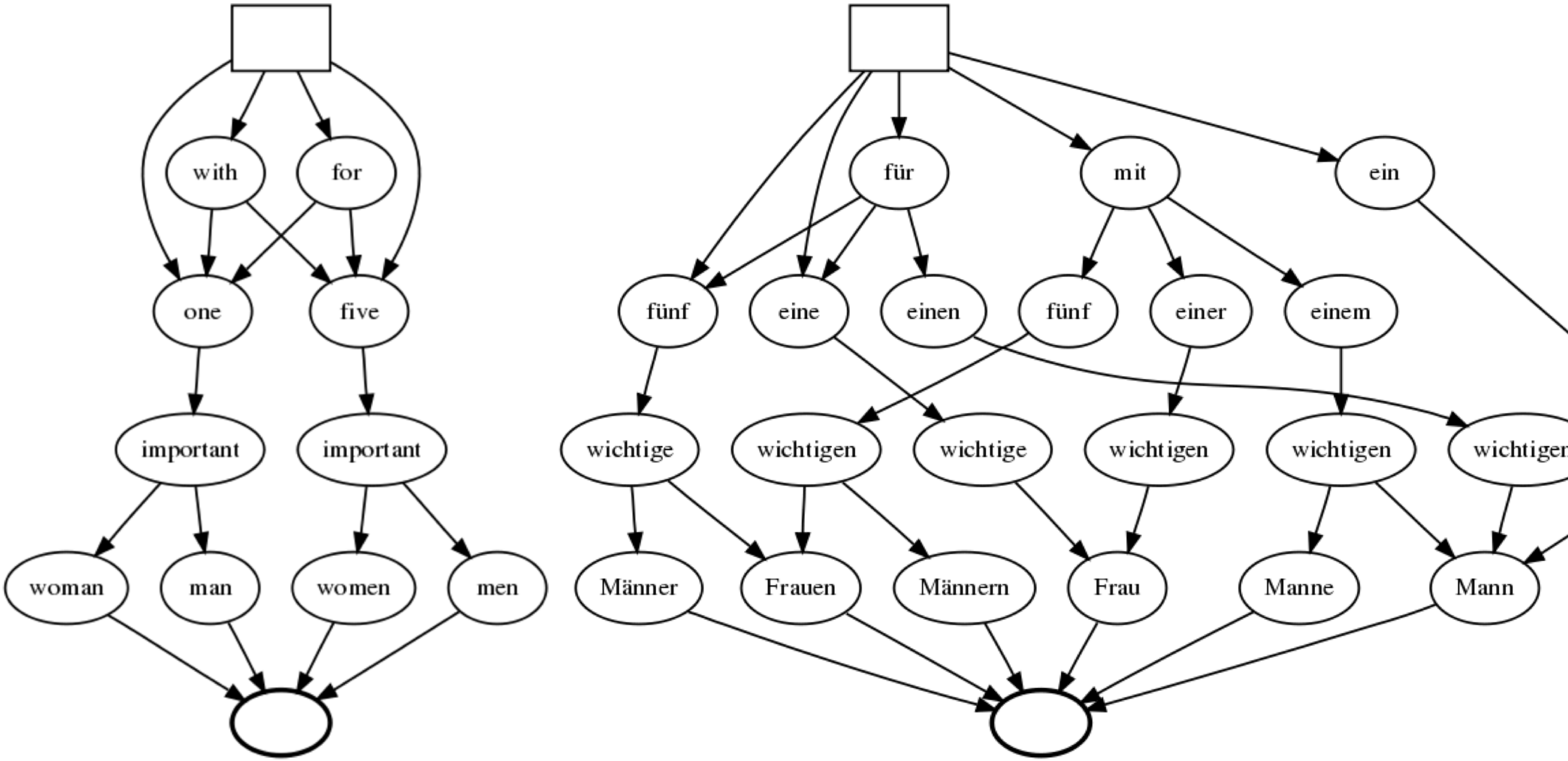
Agreement

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 and German



Semantic actions

In compilers: operations on abstract syntax trees.

In tools like YACC, this is fused with parsing rules:

```
Exp ::= Exp "+" Exp {return $1 + $3}
```

The semantic action is in curly brackets: compute the sum.

In general: any program, such as

- answer a question
- consult a database
- dialogue system

Interoperability

Semantic actions are expressed in a **host language** (Java, C, Haskell...)

GF parsing can return trees as host language objects.

This is available in many languages: Haskell, Java, JavaScript, C, Python

Application grammars and 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.

Syntax is complicated

Example: German word order

- main clause: *x ist unteibar*
- inverted clause: *ist x unteilbar*
- subordinate clause: *x unteilbar ist*

wenn x unteilbar ist, dann ist x unteilbar

if x is prime, then x is prime

Moreover: agreement in number and person, mood, tense,...

An approximative rule

```
lin Prime x = \\ord,mod =>
  let
    ist = case <mod,x.n> of {
      <Ind, Sg> => "ist" ;
      <Ind, Pl> => "sind" ;
      <Conj,Sg> => "sei" ;
      <Conj,Pl> => "seien"
      }
  in case ord of {
       Main => x.s ! Nom ++ ist ++ "unteilbar" ;
       Sub  => x.s ! Nom ++ "unteilbar" ++ ist ;
       Inv => ist ++ x.s ! Nom ++ "unteilbar"
       }
```

A precise rule using the resource grammar

In full linguistic detail.

```
lin Prime x = UseCl
  (TTAnt TPres ASimul) PPos
  (PredVP x (UseComp (CompAP (PositA unteilbar_A))))
```

With the high-level resource grammar API.

```
lin Prime x = mkS (mkCl x unteilbar_A)
```

The GF Resource Grammar Library

Syntactic structure and morphology.

For 20 languages (in August 2011).

Designed to be usable by domain experts without linguistic training.

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

The Resource Grammar API

Syntax: common for all languages,

```
lin Prime x = mkS (mkCl x prime_A)
lin Prime x = mkS (mkCl x unteilbar_A)
lin Prime x = mkS (mkCl x alkuluku_N)
```

Morphology: separate (but similar) for each language:

```
prime_A     = mkA "prime"
alkuluku_N  = mkN "alkuluku"
unteilbar_A = mkA "unteilbar"
```

In the book

Simple application grammars: Chapters 2-4

Applications using resource grammar: Chapters 5-8

Resource grammars: Chapters 9-10

Translation equivalence

Resource grammar does *not* guarantee sameness of meaning!

Application grammars are (usually) *designed* to guarantee this.

The problem can be reliably solved only on restricted domains.

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, ”old notation”

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

2002: v 1.0, revised syntax more like a functional language, still used

2004: parsing complexity solved (Ljunglöf)

2004: v 2.0, module system

2009: incremental parsing (Angelov)

2009: v 3.0, separate run-time format (PGF) and binary object files

Some application projects

1998: Multilingual Document Authoring at Xerox: phrasebook, medical drug descriptions, query language (Dymetman, Lux, Ranta)

2000: Extensible Proof Text Editor GF-Alfa (Hallgren and Ranta)

2002: Software specifications in KeY (Hähnle, Johannisson, Ranta)

2004: Multimodal dialogue systems (TALK project: Ljunglöf, Bringert, Lemon)

2005: Mathematical exercises (WebALT project: Saludes, Casanellas, Caprotti)

2010: MOLTO project (Multilingual On-Line Translation)

Related work

GF is rooted in at least four research traditions:

- logic: type theory and logical frameworks
- formal linguistic syntax
- compiler construction
- functional programming

Abstract and concrete syntax in linguistics

Tectogrammatical and phenogrammatical structure (Curry 1961)

Montague grammar (Montague 1974)

Abstract Categorial Grammar (de Groote 2001)

HOG (Higher-Order Grammar, Pollard 2004)

Lambda Grammar (Muskens 2001)

Compiler construction

Abstract and concrete syntax (McCarthy 1962, Landin 1967, Appel 1998)

Multi-source multi-target compiler: GCC (GNU Compiler Collection, Stallman 2004).

Linguistic grammar formalisms

DCG (Definite Clause Grammars, Pereira and Warren 1980)

LFG (Lexical-Functional Grammars, Bresnan 1982)

HPSG (Head-Driven Phrase Structure Grammars, Pollard and Sag 1994),

TAG (Tree-Adjoining Grammars, Joshi 1985)

CCG (Combinatory Categorial Grammar, Steedman 1988)

Core Language Engine (Alshawi 1992)

MRS (Minimal Recursion Semantics, Copestake & al. 2001)

Embedded grammars

DCG embedded Prolog: Pereira and Shieber (1987), Gazdar and Mellish (1990), and Blackburn and Bos (2003)

The Zen toolkit (Huet 2005): library for morphology and lexicon implementations in the OCAML programming language.

NLTK (Natural Language Toolkit, Bird & al. 2009): tools for language processing in Python.

Grammar formalism implementations

NL-YACC (Ishii & al. 1994)

LKB for HPSG (Lexical Knowledge Builder, Copestake 2002)

XLE for LFG (Xerox Linguistic Environment)

Multilingual resource grammars

CLE in DCG (Core Language Engine, Alshawi & al. 1992, Rayner & al. 2000)

Regulus (Rayner & al. 2006)

LinGO Matrix in HPSG (Bender and Flickinger 2004, later renamed to DELPH-IN)

Pargram in LFG (Butt 2003)

Chapter 2: Basic concepts of multilingual grammars

Outline

- BNF grammars and their use in the GF system
- GF functionalities: parsing, generation, translation
- abstract vs. concrete syntax
- abstract syntax trees vs. parse trees
- limitations of the BNF format
- string-based GF grammars as a generalization of BNF
- the module structure of GF
- free variation
- limitations of string-based GF
- visualization of trees and word alignments
- lexing, unlexing, and 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 general 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

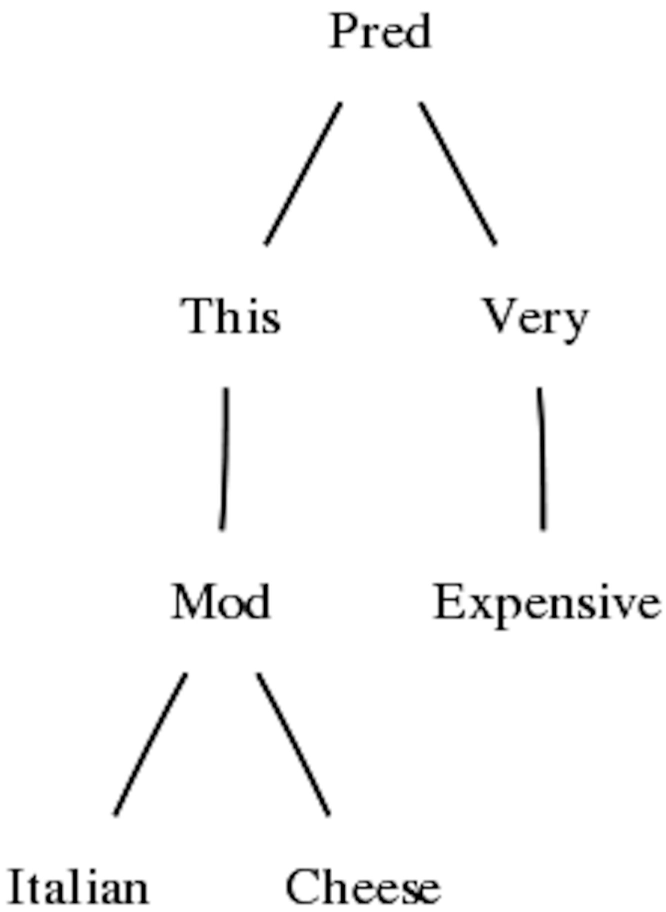
this Italian cheese is expensive

is **parsed** to the tree

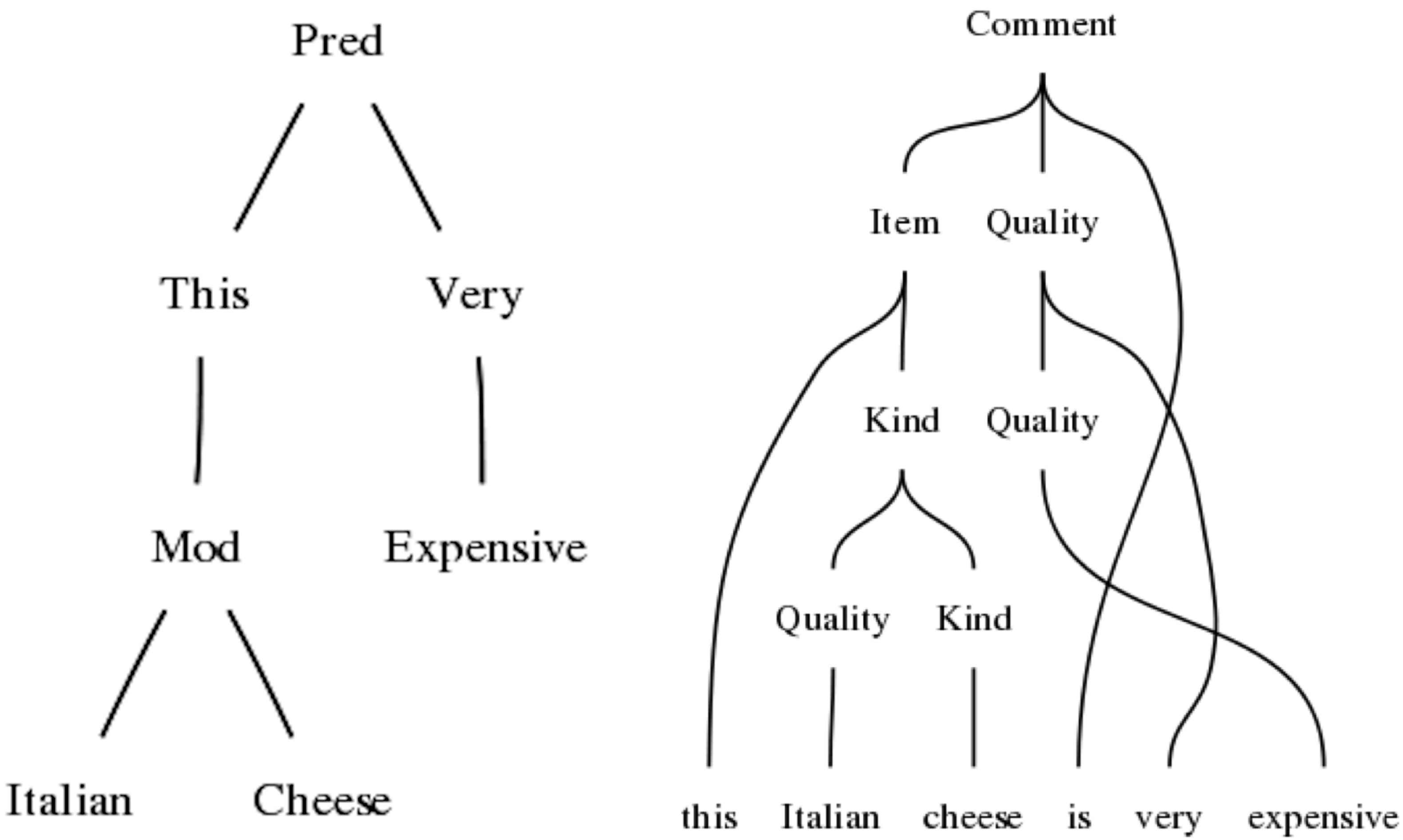
```
Pred (This (Mod Italian Cheese)) Expensive
```

This tree is **linearized** to thev 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 3.2.
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.

A BNF grammar for Italian

```
Pred.      Comment ::= Item "è" Quality
This.      Item    ::= "questo" Kind
That.      Item    ::= "quel" Kind
Mod.       Kind    ::= Kind Quality
Wine.      Kind    ::= "vino"
Cheese.    Kind    ::= "formaggio"
Fish.      Kind    ::= "pesce"
Very.      Quality ::= "molto" Quality
Fresh.     Quality ::= "fresco"
Warm.      Quality ::= "caldo"
Italian.   Quality ::= "italiano"
Expensive. Quality ::= "caro"
Delicious. Quality ::= "delizioso"
Boring.    Quality ::= "noioso"
```

Same abstract syntax

```
Pred (This Cheese) (Very Expensive)
```

this Italian cheese is very expensive

questo formaggio italiano è molto caro

So can we translate via the abstract syntax?

Different abstract syntax

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

this Italian cheese is very expensive

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

questo formaggio italiano è molto caro

Alas, the order of arguments is different in modification!

Translation with BNF grammars

Do it by hand:

```
> import foodEng.cf

> parse "this cheese is expensive"
Pred (This Cheese) Expensive

> empty

> import foodIta.cf

> linearize Pred (This Cheese) Expensive
questo formaggio è caro
```

You must `empty` the grammar environment because the abstract syntax has changed...

What we would like to do

Use a pipe:

```
> parse "this cheese is expensive" | linearize
questo formaggio è caro
```

But this is not possible:

- the BNF grammars are unrelated
- they have distinct abstract syntaxes

Solution: proceed from BNF to the full GF.

Category skeleton

Ignore terminals:

```
Comment ::= Item "is" Quality
Comment ::= Item "è" Quality
```

This gives the **category skeleton**

```
Comment ::= Item Quality
```

And from

```
Quality ::= "expensive"
Quality ::= "caro"
```

What is the category skeleton?

Category skeleton, cont'd

In the rules

```
Quality ::= "expensive"
Quality ::= "caro"
```

there are no nonterminals on the right hand side, so we get

```
Quality ::=
```

But modification rules

```
Kind ::= Quality Kind
Kind ::= Kind Quality
```

have different category skeletons.

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 type

The category skeleton

```
Comment ::= Item Quality
```

gives the **function type**

```
Item -> Quality -> Comment
```

Value type: `Comment`

Argument types: `Item` and `Quality`

Right associativity:

$$A \text{ -> } B \text{ -> } C \;\equiv\; A \text{ -> } (B \text{ -> } C)$$

Linearization rules

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

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

Concatenation: `++`,

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

Sharing abstract syntax

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, either on the same line or separately,

```
> 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 book for details.

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 Chapters 3 and 6.)

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 Pizza (With Fish 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

We can

- define some non-context-free languages
- ignore word order in abstract syntax

We can't

- ignore language-dependent morphological features
- deal with discontinuos constituents

Example: morphological features

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.

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.

Visualization of abstract syntax trees

`vt = visualize_tree` prints

```
> parse "this cheese is very expensive" | vt
```

GF displays a few lines of **graphviz** code.

How to see the tree?

Save output in files

You can save any output by `wf` = `write_file`:

```
> parse "this..." | vt | wf -file=tree.dot
```

Now process this in the Unix shell,

```
$ dot -Tpng tree.dot >tree.png
```

Finally, view the result,

```
$ open tree.png  -- in MacOS
$ eog  tree.png  -- in Ubuntu Linux
```

Shell escapes

Without leaving GF, prefix command with `!`

```
> ! dot -Tpng tree.dot >tree.png
> ! open tree.png
```

Another handy one:

```
> ! clear
```

Put it all together

Use `vt` with a flag for viewer program:

```
> parse "this cheese is boring" | vt -view=open
```

Similarly, visualize parse trees by `vp` = `visualize_parse`

```
> parse "this cheese is boring" | vp -view=open
```

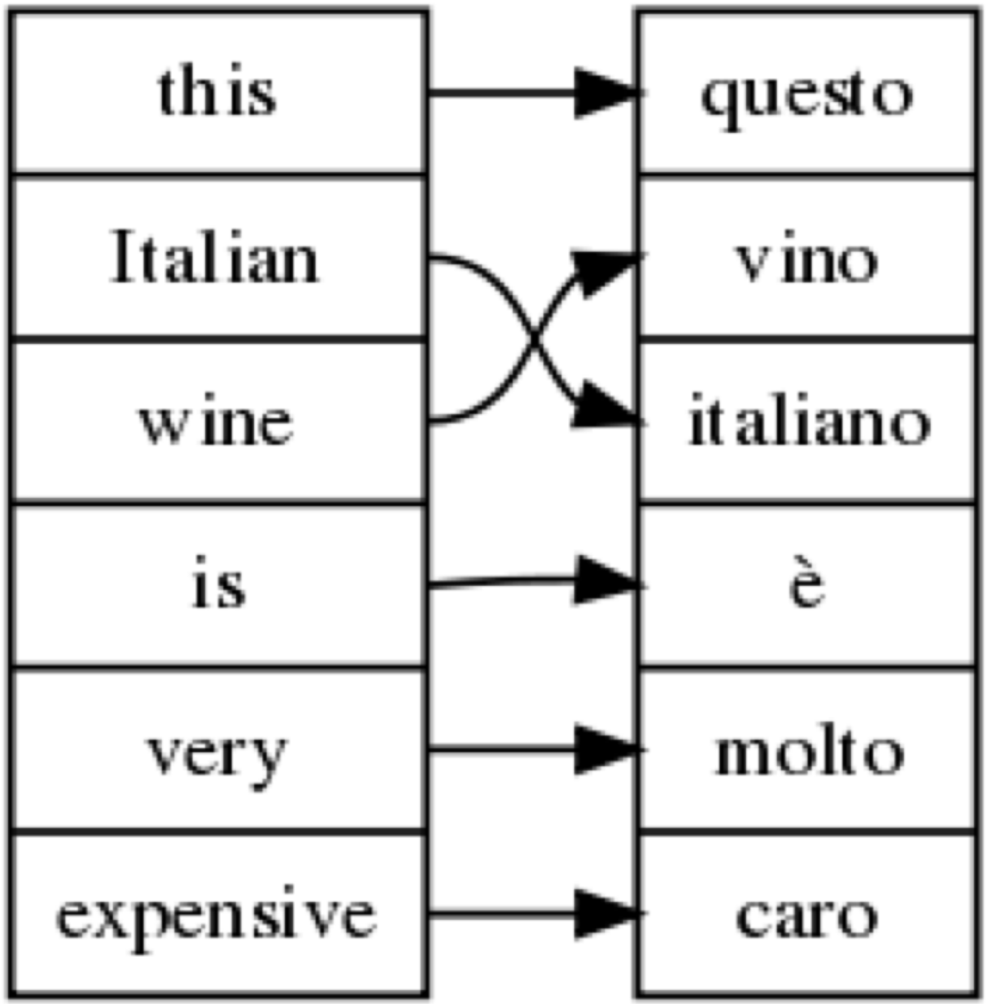
or dependency trees by `vd` = `visualize_dependency`

```
> parse "this cheese is boring" | vd -view=open
```

or show word alignment, `aw` = `align_words`:

```
> p "this Italian wine is very expensive" | aw -view=open
```

Word alignment



System pipes

Escape `?`: the escaped command receives its input from a GF pipe

```
> gr | l | ? wc -c
```

counts characters, whereas

```
> gr | l | ? espeak -f
```

sends the string to the speech synthesizer `espeak`.

Lexing and unlexing

`Str` is not strings of characters, but **token lists**.

Default representation: tokens separated by spaces

Example: the string

```
"this wine is delicious"
```

represents the token list

```
"this", "wine", "is", "delicious"
```

which the parser can recognize

Lexing problems

The string

```
"(12 + (3 * 4))"
```

by default represents the token list

```
"(12", "+", "(3", "*", "4))"
```

But we want

```
"(", "12", "+", "(", "3", "*", "4", ")", ")"
```

For this, we need a **lexer**.

String processing

Generic string processing command `ps` = `put_string` can take a lexer as an option:

```
> put_string -lexcode "(12 + (3 * 4))"
( 12 + ( 3 * 4 ) )
```

```
> put_string -lexcode "(12 + (3 * 4))" | parse
EPlus (EInt 12) (ETimes (EInt 3) (EInt 4))
```

Similarly, **unlexer**

```
> put_string -unlexcode "( 12 + ( 3 * 4 ) )"
(12 + (3 * 4))
```

Some lexers and unlexers

lexer	description
`words`	(default) tokens as separated by spaces or newlines
`chars`	treat each character as a token
`lexcode`	lex as program code (uses Haskell's lex)
`lextext`	use conventions on punctuation and capital letters

unlexer	description
`unwords`	(default) separate tokens by spaces
`unchars`	glue tokens together without spaces
`unlexcode`	format as code (spacing, indentation)
`unlextext`	format as text: punctuation, capitals

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" ;
}
```

Transliterations

Use ASCII instead of Unicode

Can be handy in editing grammar files

Also in shells missing fonts, etc.

More on this in Chapter 10.

Chapter 3: Parameters, tables, and records

Outline

- morphological variation
- variable vs. inherent features
- agreement
- parameters, tables, and records
- pattern matching
- data structures in linearization types
- functional programming in GF by operation definitions
- discontinuous constituents

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 term

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

of type

```
Number => Str
```

Selection

To access a value in a table,

```
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
- X passes Y to F

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

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

(Italian *zia*, "aunt").

Linearization types

Now generalized from `Str` to parameters, tables, and records of any complexity.

Thus in Italian,

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

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`.

English Foods: types

Linearization types

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

It's a good habit to use records `{s : Str}` instead of plain `Str`.

Then it is easier 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,

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

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

$$\backslash x_1 \texttt{,} \ \ldots \texttt{,} \ x_n \ \texttt{->} \ t \ \equiv \ \backslash x_1 \ \texttt{->} \ \ldots \ \backslash \ x_n \ \texttt{->} \ t$$

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.

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.

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*

Words are datastructures rather than strings!

Datastructures for Arabic

Roots are records of strings.

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

Patterns are functions from roots to strings.

```
Pattern : Type = Root -> Str ;
```

A special case is filling: a record of strings filling the four slots in a root.

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

This is enough for everything except middle consonant duplication (e.g. *FaCCaLa*).

Applying a pattern

A pattern obtained from a filling intertwines the records:

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

Middle consonant duplication also uses a filling but duplicates the *C* consonant of the root:

```
dfill : Filling -> Pattern = \p,r ->
  p.F + r.F + p.FC + r.C + r.C + p.CL + r.L + p.L ;
```

Arabic lexicon

Possible although tedious

```
yaktubu = fill
  {F = "ya" ; FC = "" ; CL = "u" ; L = "u"}
  {F = "k" ; C = "t" ; L = "b"}
kuttiba = dfill
  {F = "" ; FC = "u" ; CL = "i" ; L = "a"}
  {F = "k" ; C = "t" ; L = "b"}
```

We would like to write something like

```
yaktubu = word "yaFCuLu" "ktb"
kuttiba = word "FuCCiLa" "ktb"
```

Next chapter!

Chapter 4: Modular and scalable grammar writing

Outline

- reusable resource modules
- data abstraction
- smart paradigms
- pattern matching over strings
- operation overloading
- module extension and inheritance
- algebraic datatypes
- record extension, subtyping, and tuples
- prefix-dependent choices
- compile-time vs. run-time string operations

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
}
```

Example: an Italian resource

```
resource ResIta = open Prelude in {
  param
    Number = Sg | Pl ;
    Gender = Masc | Fem ;
  oper
    NounPhrase : Type =
      {s : Str ; g : Gender ; n : Number} ;
    Noun : Type = {s : Number => Str ; g : Gender} ;
    Adjective : Type = {s : Gender => Number => Str} ;

    det : Number -> Str -> Str -> Noun -> NounPhrase =
      \n,m,f,cn -> {
        s = table {Masc => m ; Fem => f} ! cn.g ++
            cn.s ! n ;
        g = cn.g ;
```

```
    n = n
  } ;
noun : Str -> Str -> Gender -> Noun =
  \vino,vini,g -> {
    s = table {Sg => vino ; Pl => vini} ;
    g = g
  } ;
adjective : (nero,nera,neri,nere : Str) -> Adjective =
  \nero,nera,neri,nere -> {
    s = table {
      Masc => table {Sg => nero ; Pl => neri} ;
      Fem => table {Sg => nera ; Pl => nere}
      }
    } ;
regAdj : Str -> Adjective = \nero ->
  let ner : Str = init nero
  in
```

```
    adjective nero (ner+"a") (ner+"i") (ner+"e") ;
  copula : Number => Str =
    table {Sg => "è" ; Pl => "sono"} ;
}
```

Local definitions

Syntax:

`let` c `:` t `=` d `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

```
concrete FoodsIta of Foods = open ResIta in {
  lincat
    Comment = {s : Str} ;
    Quality = Adjective ;
    Kind = Noun ;
    Item = NounPhrase ;
  lin
    Pred item quality =
      {s = item.s ++ copula ! item.n ++
           quality.s ! item.g ! item.n} ;
    This  = det Sg "questo" "questa" ;
    That  = det Sg "quel"   "quella" ;
    These = det Pl "questi" "queste" ;
    Those = det Pl "quei"   "quelle" ;
    Mod quality kind = {
```

```
      s = \\n => kind.s ! n ++ quality.s ! kind.g ! n ;
      g = kind.g
      } ;
    Wine = noun "vino" "vini" Masc ;
    Cheese = noun "formaggio" "formaggi" Masc ;
    Fish = noun "pesce" "pesci" Masc ;
    Pizza = noun "pizza" "pizze" Fem ;
    Very qual = {s = \\g,n => "molto" ++ qual.s ! g ! n} ;
    Fresh =
      adjective "fresco" "fresca" "freschi" "fresche" ;
    Warm = regAdj "caldo" ;
    Italian = regAdj "italiano" ;
    Expensive = regAdj "caro" ;
    Delicious = regAdj "delizioso" ;
    Boring = regAdj "noioso" ;
}
```

Data abstraction

Problem: implementing inflection paradigms.

Goals

- easy to use for all words, not just regular
- easy to modify with different sets of forms

Solution:

- abstract data types (hiding records and tables)
- constructor operations

Type synonym

Define the type `Noun` and use it consistently

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

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
    }
  } ;
```

Regular and irregular nouns

The regular oper is defined by using the constructor

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

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 will retain its type signature:

```
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"
        }
      }
    } ;
```

Old definitions are still valid:

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

Case expressions and string matching

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

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

There are several patterns, some corresponding to **regular expressions**.

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**

  index as

  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:

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

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: *bus*-*buses*, *leech*-*leeches*

Special paradigms

The first solution

```
noun_y : Str -> Noun = \fly ->
  mkNoun fly (init fly + "ies") ;
noun_s : Str -> Noun = \bus ->
  mkNoun bus (bus + "es") ;
```

But this solution has some drawbacks:

- it can be difficult to select the correct paradigm
- it can be difficult to remember the names of all different paradigms

Smart paradigms

A better solution: let GF select the paradigm

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

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*.

Arabic morphology revisited: encoding roots by strings

This is just for the ease of programming and writing lexica.

F = first letter, C = second letter, L = the rest.

```
getRoot : Str -> Root = \s -> case s of {
  F@? + C@? + L => {F = F ; C = C ; L = L} ;
  _ => Predef.error ("cannot get root from" ++ s)
  } ;
```

The **as-pattern** `x@p` matches `p` and binds `x`.

The **error function** `Predef.error` stops computation and displays the string. It is a typical catch-all value.

Encoding patterns by strings

Patterns are coded by using the letters `F`, `C`, `L`.

```
getPattern : Str -> Pattern = \s -> case s of {
  F + "F" + FC + "CC" + CL + "L" + L =>
    dfill {F = F ; FC = FC ; CL = CL ; L = L} ;
  F + "F" + FC + "C" + CL + "L" + L =>
    fill {F = F ; FC = FC ; CL = CL ; L = L} ;
  _ => Predef.error ("cannot get pattern from" ++ s)
  } ;
```

A high-level lexicon building function

Dictionary entry: root + pattern.

```
getWord : Str -> Str -> Str = \r,p ->
  getPattern p (getRoot r) ;
```

Now we can try:

```
> cc getWord "ktb" "yaFCuLu"
"yaktubu"
> cc getWord "ktb" "muFaCCiLu"
"mukattibu"
```

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.

This is formalized in interface/instance modules (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.

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

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

Inheritance: same type of module, inherit contents

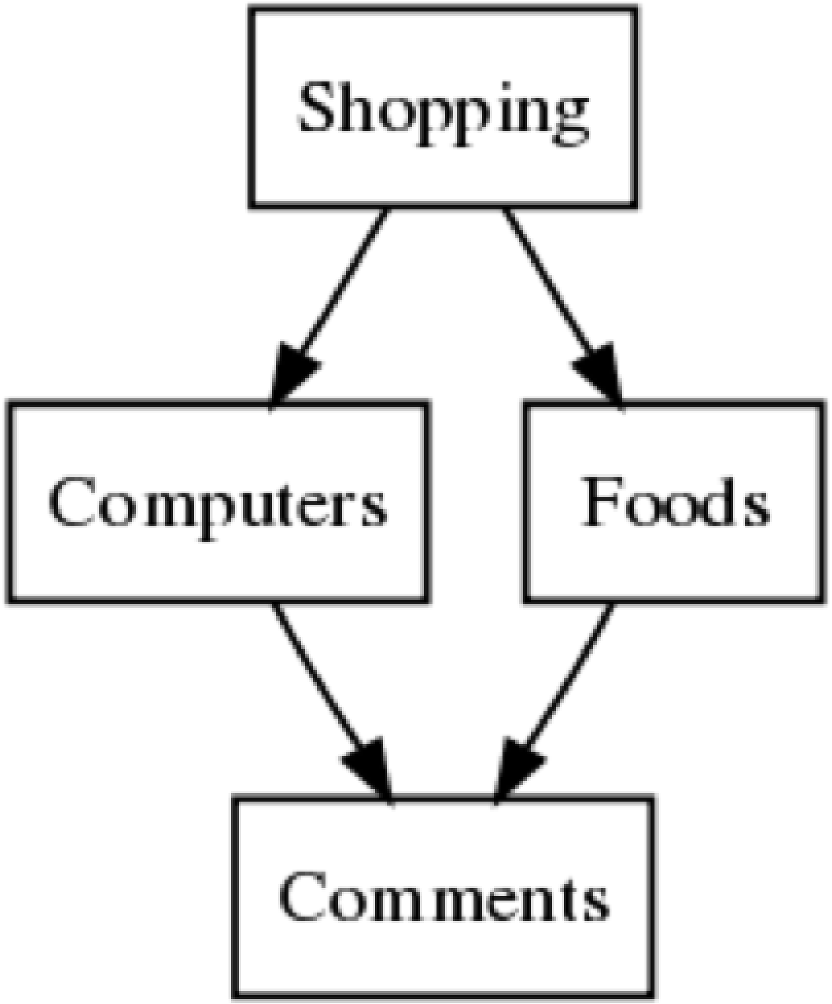
Opening: resource modules, just use its contents

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



How to produce dependency graphs

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

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.

Information hiding

Mimicking "private" and "public"

```
resource Auxiliary = {oper aux ...}

resource Library = open Auxiliary in {oper foo = aux ...}

concrete Application of A = open Library in {
  lin f = foo ... ;  -- CORRECT
  lin g = aux ...    -- INCORRECT
}
```

Qualified names

Problem: a constant appears in two opened modules.

Solution: use **qualified name**

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

One can also qualify all names in opening,

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

Algebraic datatypes for parameters

Parameter constructors with arguments (like datatypes in Haskell and ML).

Create parameter hierarchies.

Example: German determiners have genders only in the singular. Don't write

```
param Gender = Masc | Fem | Neutr
param Case = Nom | Acc | Dat | Gen

lincat Det = Number => Gender => Case => Str
```

but (getting 3*4+4=16 distinct values instead of 2*3*4=24)

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

lincat Det = DetForm => Str
```

German definite article

```
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.

Abstraction via parameter types

The definition

```
lincat N = {s : Number => Case => Str}
```

leaks the information that nouns have two variable features.

```
param NForm = NF Number Case ;
lincat N = {s : NForm => Str}
```

gives more robust code.

For instance

```
lin Mod adj noun = {s = \\f => adj.s ++ noun.s ! f}
```

now works independently of whether nouns have a case.

Parameter records

An alternative to `NForm`

```
lincat N = {s : {n : Number ; c : Case} => Str}
```

Cf. **feature structures** in unification grammars.

Pattern matching with partial patterns (for nouns like *fish*):

```
oper invarPluralN :
  Str -> {s : {n : Number ; c : Case} => Str} = \s -> {
    s = table {
      {c = Gen} => s + "'s" ;
      _ => s
      }
    }
```

Record extension and subtyping

Two-place verbs as verbs with a complement case (in German):

```
lincat V2 = V ** {c : Case} ;

lin Follow = regV "folgen" ** {c = Dative} ;
```

Now V2 becomes a **subtype** of V: V2 has all fields of V.

Tuples and product types

Product types and tuples are syntactic sugar for record types and records:

$$T1 * \ldots * Tn \equiv \{\mathtt{p1}\ :\ T1\ ;\ \ldots\ ;\ \mathtt{pn}\ :\ Tn\}$$

$$<t1, \ldots, tn> \equiv \{\mathtt{p1} = T1\ ;\ \ldots\ ;\ \mathtt{pn} = Tn\}$$

The labels `p1, p2,...` are hard-coded.

Partial patterns - logically but slightly surprisingly,

```
case <g,n,p> of {
  <Fem> => t
  ...
  }
```

Prefix-dependent choices and pattern macros

Problem: English indefinite article is

- *an* if the next token begins with a vowel
- *a* otherwise

Solution: **prefix-dependent choice expression**:

```
indefArt : Str =
  pre {
    "a" | "e" | "i" | "o" | "u" => "an" ;
    _ => "a"
    } ;
```

Not really a solution in English

```
pre {
  "eu"  => "a" ;     -- a euphemism
  "uni" => "a" ;     -- a university
  "un"  => "an" ;    -- an uncle
  "u"   => "a" ;     -- a user
  "a" | "e" | "i" | "o" => "an" ;
  _     => "a"
  }
```

Problem: the article depends on *pronunciation*.

Pattern macros

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

indefArt : Str =
  pre {
    #vowel => "an" ;
    _ => "a"
    }
```

Strings at compile time vs. run time

Summary of tokens:

- quoted string: `"foo"`
- gluing : `t + s`
- predefined operations: `init, tail, tk, dp`
- pattern matching over strings: `"y" => "ies"`
- prefix-dependent choices: `pre {...}`

Principle: *all tokens must be known at compile time*.

Corollary: above operations may not be applied to **run-time variables**.

Example

Using the + operator "to eliminate the space":

```
lin Question p = {s = p + "?"} ; -- INCORRECT!
```

Solution: use the lexer `lextext`

OR: use the Prelude operation

```
glue : Str -> Str -> Str
 = \x,y -> x ++ "&+" ++ y
```

which uses a special token `&+` that lexers and unlexers may handle.

Chapter 5: Using the resource grammar library

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

Summer 2011: 20 languages complete API

`Afrikaans`	`Bulgarian`	`Catalan`	`Danish`
`Dutch`	`English`	`Finnish`	`French`
`German`	`Italian`	`Nepalese`	`Norwegian`
`Persian`	`Polish`	`Punjabi`	`Romanian`
`Russian`	`Spanish`	`Swedish`	`Urdu`

Complete inflection (and some syntax): Amharic, Arabic, Latin, Turkish

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. "this pizza is 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. "this pizza is 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"
```

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 rules

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`

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
```

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_Quant kind ;
    That kind = mkNP that_Quant kind ;
    These kind = mkNP this_Quant plNum kind ;
    Those kind = mkNP that_Quant plNum 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_Quant kind ;
    That kind = mkNP that_Quant kind ;
    These kind = mkNP this_Quant plNum kind ;
    Those kind = mkNP that_Quant plNum 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
...
```

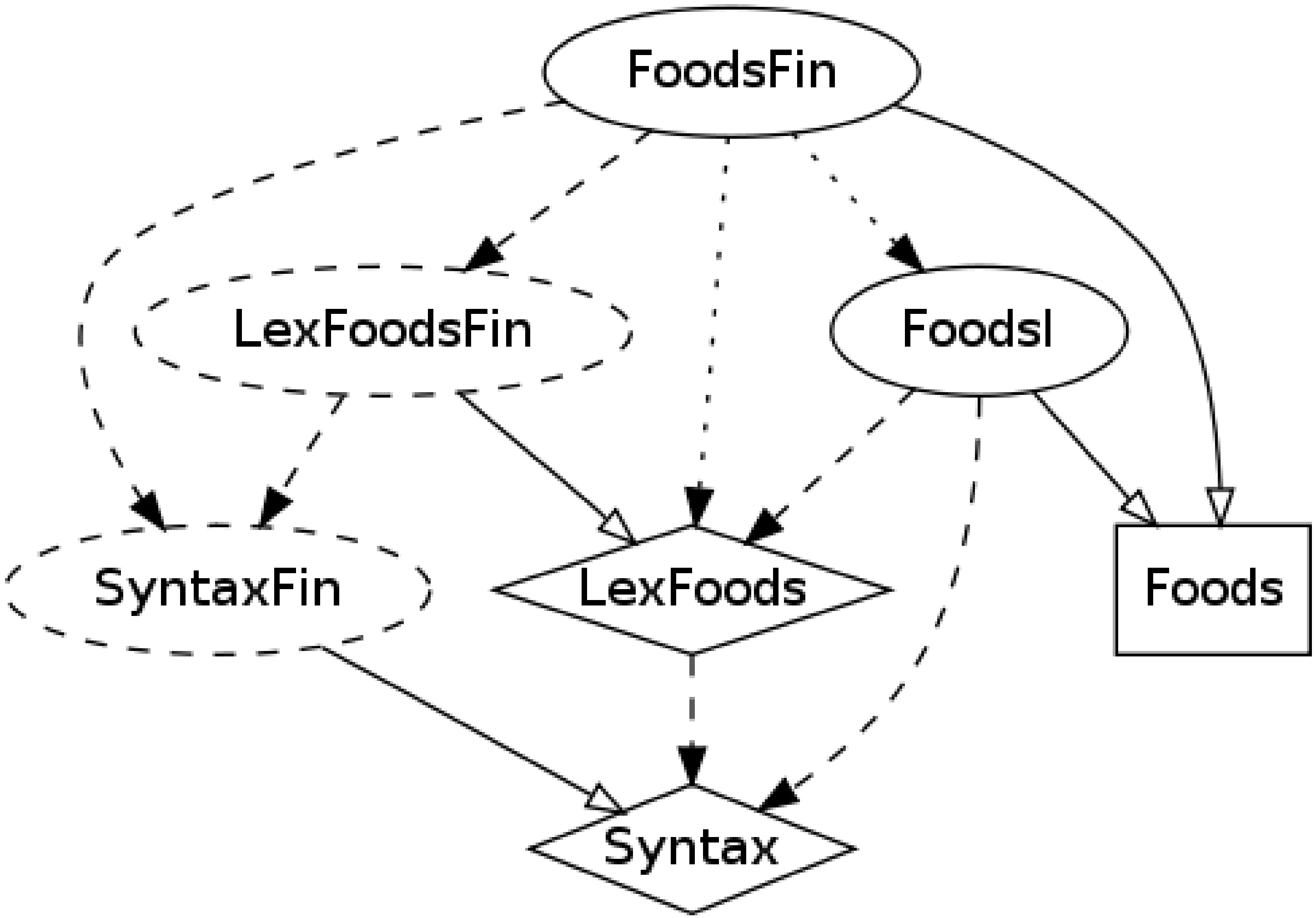
Functor instantiations

```
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 yo
```

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
```

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 as explained in Section **??**. 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. ...

Chapter 6: Semantic actions and conditions in abstract syntax

Outline

- GF as a logical framework
- dependent types
- selection restrictions
- polymorphism
- proof objects and proof-carrying documents
- variable binding and higher-order abstract syntax
- semantic definitions

Type theory

These concepts are inherited from **type theory** (more precisely: constructive type theory, or Martin-Löf type theory).

Type theory is the basis **logical frameworks**.

GF = logical framework + concrete syntax.

Dependent types

Dependent type = type depending on an object of another type.

Example: all natural numbers, numbers up to *n*

```
  cat
    Nat ;
    Nats Nat ;
```

Example usage: guarantee that *m* - *n* is well formed (i.e. *m* >= *n*)

```
  fun minus : (m : Nat) -> (n : Nats m) -> Nat
```

Dependent function type: `(x : A) -> B x` i.e. value type depends on argument type.

Another example

Vectors of *n* elements; index within bounds; append:

```
cat
  Vector Nat ;
fun
  index  : (n : Nat) -> Vector n -> Nats n -> Nat ;
  append : (m,n : Nat) -> Vector m -> Vector n -> Vector (plus m n) ;
```

Yet another example

Well-formed postal addresses

```
cat
  Address ;
  Country ;
  City Country ;
  Street (x : Country) (City x) ;
fun
  MkAddress : (x : Country) -> (y : City x) -> Street x y -> Address ;
```

Notice **progressive context**: `(x : Country) (City x)` where variable must be bound (cf. argument list of function type).

Dependent types in grammar

Problem: to express **conditions of semantic well-formedness**.

Example: a voice command system for a ''smart house'' wants to eliminate meaningless commands.

Thus we want to restrict particular actions to particular devices - we can *dim a light*, but we cannot *dim a fan*.

The following example is borrowed from the Regulus Book (Rayner & al. 2006).

A simple example is a ''smart house'' system, which defines voice commands for household appliances.

A dependent type system

Ontology:

- there are commands and device kinds
- for each kind of device, there are devices and actions
- a command concerns an action of some kind on a device of the same kind

Abstract syntax formalizing this:

```
cat
  Command ;
  Kind ;
  Device Kind ; -- argument type Kind
  Action Kind ;
fun
  CAction : (k : Kind) -> Action k -> Device k -> Command ;
```

`Device` and `Action` are both dependent types.

Examples of devices and actions

Assume the kinds `light` and `fan`,

```
light, fan : Kind ;
dim : Action light ;
```

Given a kind, *k*, you can form the device *the k*.

```
DKindOne  : (k : Kind) -> Device k ;  -- the light
```

Now we can form the syntax tree

```
CAction light dim (DKindOne light)
```

but we cannot form the trees

```
CAction light dim (DKindOne fan)
CAction fan   dim (DKindOne light)
CAction fan   dim (DKindOne fan)
```

Linearization and parsing with dependent types

Concrete syntax does not know if a category is a dependent type.

```
lincat Action = {s : Str} ;
lin CAction _ act dev = {s = act.s ++ dev.s} ;
```

Notice that the `Kind` argument is suppressed in linearization.

Parsing with dependent types is performed in two phases:

1. context-free parsing
2. filtering through type checker

Parsing with suppression

By just doing the first phase, the `kind` argument is not found:

```
> parse "dim the light"
CAction ? dim (DKindOne light)
```

Moreover, type-incorrect commands are not rejected:

```
> parse "dim the fan"
CAction ? dim (DKindOne fan)
```

The term `?` is a **metavariable**, returned by the parser for any subtree that is suppressed by a linearization rule.

(NB in GF 3.2, the parser actually tries to solve the metavariables.)

Solving metavariables

GF parser tries to solve the metavariables:

```
> parse "dim the light"
CAction light dim (DKindOne light)
```

The type checking process may fail, in which case an error message is shown and no tree is returned:

```
> parse "dim the fan"

Error in tree UCommand (CAction ? 0 dim (DKindOne fan)) :
  (? 0 <> fan) (? 0 <> light)
```

Polymorphism

Sometimes an action can be performed on all kinds of devices.

This is represented as a function that takes a `Kind` as an argument and produce an `Action` for that `Kind`:

```
fun switchOn, switchOff : (k : Kind) -> Action k ;
```

Functions of this kind are called **polymorphic**.

Polymorphism in concrete syntax

We can use this kind of polymorphism in concrete syntax oper's as well, to express Haskell-type library functions:

```
oper const :(a,b : Type) -> a -> b -> a =
  \_,_,c,_ -> c ;

oper flip : (a,b,c : Type) -> (a -> b ->c) -> b -> a -> c =
  \_,_,_,f,x,y -> f y x ;
```

Proof objects

Curry-Howard isomorphism = propositions as types principle: a proposition is a type of proofs (= proof objects).

Example: define the *less than* proposition for natural numbers,

```
cat Nat ;
fun Zero : Nat ;
fun Succ : Nat -> Nat ;
```

Define inductively what it means for a number x to be *less than* a number y:

- `Zero` is less than `Succ` y for any y.
- If x is less than y, then `Succ` x is less than `Succ` y.

The axioms in type theory

Expressing these axioms in type theory with a dependent type `Less` x y and two functions constructing its objects:

```
cat Less Nat Nat ;
fun lessZ : (y : Nat) -> Less Zero (Succ y) ;
fun lessS : (x,y : Nat) -> Less x y -> Less (Succ x) (Succ y) ;
```

Example: the fact that 2 is less that 4 has the proof object

```
lessS (Succ Zero) (Succ (Succ (Succ Zero)))
      (lessS Zero (Succ (Succ Zero)) (lessZ (Succ Zero)))
 : Less (Succ (Succ Zero)) (Succ (Succ (Succ (Succ Zero))))
```

Proof-carrying documents

Idea: to be semantically well-formed, the abstract syntax of a document must contain a proof of some property, although the proof is not shown in the concrete document.

Example: documents describing flight connections:

To fly from Gothenburg to Prague, first take LH3043 to Frankfurt, then OK0537 to Prague.

The well-formedness of this text is partly expressible by dependent typing:

```
cat
  City ;
  Flight City City ;
```

```
fun
  Gothenburg, Frankfurt, Prague : City ;
  LH3043 : Flight Gothenburg Frankfurt ;
  OK0537 : Flight Frankfurt Prague ;
```

Proving that the connection is possible

To extend the conditions to flight connections, we introduce a category of proofs that a change is possible:

```
cat IsPossible (x,y,z : City)(Flight x y)(Flight y z) ;
```

A legal connection is formed by the function

```
fun Connect : (x,y,z : City) ->
  (u : Flight x y) -> (v : Flight y z) ->
    IsPossible x y z u v -> Flight x z ;
```

Restricted polymorphism

Above, all Actions were either of

- **monomorphic**: defined for one Kind
- **polymorphic**: defined for all Kinds

To make this scale up for new Kinds, we can refine this to **restricted polymorphism**: defined for Kinds of a certain **class**

The notion of class uses the Curry-Howard isomorphism as follows:

- a class is a **predicate** of Kinds — i.e. a type depending of Kinds
- a Kind is in a class if there is a proof object of this type

Example: classes for switching and dimming

We modify the smart house grammar:

```
cat
  Switchable Kind ;
  Dimmable   Kind ;
fun
  switchable_light : Switchable light ;
  switchable_fan   : Switchable fan ;
  dimmable_light   : Dimmable light ;

  switchOn : (k : Kind) -> Switchable k -> Action k ;
  dim      : (k : Kind) -> Dimmable k -> Action k ;
```

Classes for new actions can be added incrementally.

Variable bindings

Mathematical notation and programming languages have expressions that **bind** variables.

Example: universal quantifier formula

```
(All x)B(x)
```

The variable `x` has a **binding** `(All x)`, and occurs **bound** in the **body** `B(x)`.

Examples from informal mathematical language:

```
for all x, x is equal to x

the function that for any numbers x and y returns the maximum of x+y
and x*y

Let x be a natural number. Assume that x is even. Then x + 3 is odd.
```

Higher-order abstract syntax

Abstract syntax can use functions as arguments:

```
cat Ind ; Prop ;
fun All : (Ind -> Prop) -> Prop
```

where `Ind` is the type of individuals and `Prop`, the type of propositions.

Let us add an equality predicate

```
fun Eq : Ind -> Ind -> Prop
```

Now we can form the tree

```
All (\x -> Eq x x)
```

which we want to relate to the ordinary notation

```
(All x)(x = x)
```

In **higher-order abstract syntax** (HOAS), all variable bindings are expressed using higher-order syntactic constructors.

Higher-order abstract syntax: linearization

HOAS has proved to be useful in the semantics and computer implementation of variable-binding expressions.

How do we relate HOAS to the concrete syntax?

In GF, we write

```
fun All : (Ind -> Prop) -> Prop
lin All B = {s = "(" ++ "All" ++ B.$0 ++ ")" ++ B.s}
```

General rule: if an argument type of a `fun` function is a function type `A -> C`, the linearization type of this argument is the linearization type of `C` together with a new field `$0 : Str`.

The argument `B` thus has the linearization type

```
{s : Str ; $0 : Str},
```

If there are more bindings, we add `$1`, `$2`, etc.

Eta expansion

To make sense of linearization, syntax trees must be **eta-expanded**: for any function of type

```
A -> B
```

an eta-expanded syntax tree has the form

```
\x -> b
```

where `b : B` under the assumption `x : A`.

Linearization needs eta expansion

Given the linearization rule

```
lin Eq a b = {s = "(" ++ a.s ++ "=" ++ b.s ++ ")"}
```

the linearization of the tree

```
\x -> Eq x x
```

is the record

```
{$0 = "x", s = "( x = x )"}
```

Then we can compute the linearization of the formula,

```
All (\x -> Eq x x)  --> {s = "( All x ) ( x = x )"}.
```

The linearization of the variable `x` is, "automagically", the string `"x"`.

Parsing variable bindings

GF can treat any one-word string as a variable symbol.

```
> p -cat=Prop "( All x ) ( x = x )"
All (\x -> Eq x x)
```

Variables must be bound if they are used:

```
> p -cat=Prop "( All x ) ( x = y )"
no tree found
```

Semantic definitions

The `fun` judgements of GF are declarations of functions, giving their types.

Can we **compute** `fun` functions?

Mostly we are not interested, since functions are seen as constructors, i.e. data forms - as usual with

```
fun Zero : Nat ;
fun Succ : Nat -> Nat ;
```

But it is also possible to give **semantic definitions** to functions. The key word is `def`:

```
fun one : Nat ;
```

```
def one = Succ Zero ;

fun twice : Nat -> Nat ;
def twice x = plus x x ;

fun plus : Nat -> Nat -> Nat ;
def
  plus x Zero = x ;
  plus x (Succ y) = Succ (Sum x y) ;
```

Computing a tree

Computation: follow a chain of definition until no definition can be applied,

```
plus one one -->
plus (Succ Zero) (Succ Zero) -->
Succ (plus (Succ Zero) Zero) -->
Succ (Succ Zero)
```

Computation in GF is performed with the `put_term` command and the `compute` transformation, e.g.

```
> parse -tr "1 + 1" | put_term -compute -tr | l
plus one one
Succ (Succ Zero)
s(s(0))
```

Definitional equality

Two trees are definitionally equal if they compute into the same tree.

Definitional equality does not guarantee sameness of linearization:

```
plus one one      ===> 1 + 1
Succ (Succ Zero) ===> s(s(0))
```

The main use of this concept is in type checking: sameness of types.

Thus e.g. the following types are equal

```
Less Zero one
Less Zero (Succ Zero))
```

so that an object of one also is an object of the other.

Judgement forms for constructors

The judgement form `data` tells that a function is a data constructor:

```
data
  Zero : Nat ;
  Succ : Nat -> Nat ;
```

Notice: in `def` definitions, identifier patterns not marked as `data` will be treated as variables.

Hence `data` must be used instead of `fun`, to make patterns in `def` definitions work correctly.

Exercises on semantic definitions

1. Implement an interpreter of a small functional programming language with natural numbers, lists, pairs, lambdas, etc. Use higher-order abstract syntax with semantic definitions. As concrete syntax, use your favourite programming language.

2. There is no termination checking for `def` definitions. Construct an example that makes type checking loop.

Chapter 7: Embedded grammars and code generation

Outline

- PGF, a portable format for multilingual GF grammars
- host-language API's for PGF
- manipulation of abstract syntax trees in the host language
- stand-alone translation programs
- question-answering systems
- multilingual syntax editors
- web services
- mobile phone applications
- language models for speech recognition

Functionalities of an embedded grammar format

GF grammars can be used as parts of programs written in other programming languages, to be called **host languages**.

This facility is based on several components:

- PGF: a portable format for multilingual GF grammars
- a PGF interpreter written in the host language
- a library in the host language that enables calling the interpreter
- a way to manipulate abstract syntax trees in the host language

The portable grammar format

The portable format is called PGF, "Portable Grammar Format".

This format is produced by using GF as batch compiler, with the option `-make`, from the operative system shell:

```
% gf -make SOURCE.gf
```

PGF is the recommended format in which final grammar products are distributed

- stripped from superfluous information
- can be started and faster than sets of separate modules

Application programmers have never any need to read or modify PGF files.

PGF thus plays the same role as machine code in general-purpose programming (or bytecode in Java).

Haskell: the EmbedAPI module

The Haskell API contains (among other things) the following types and functions:

```
readPGF   :: FilePath -> IO PGF

linearize :: PGF -> Language -> Tree -> String
parse     :: PGF -> Language -> Category -> String -> [Tree]

linearizeAll     :: PGF -> Tree -> [String]
linearizeAllLang :: PGF -> Tree -> [(Language,String)]

parseAll     :: PGF -> Category -> String -> [[Tree]]
parseAllLang :: PGF -> Category -> String -> [(Language,[Tree])]

languages    :: PGF -> [Language]
categories   :: PGF -> [Category]
startCat     :: PGF -> Category
```

First application: a translator

Let us first build a stand-alone translator, which can translate in any multilingual grammar between any languages in the grammar.

```
module Main where

import PGF
import System (getArgs)

main :: IO ()
main = do
  file:_ <- getArgs
  gr     <- readPGF file
  interact (translate gr)

translate :: PGF -> String -> String
```

```
translate gr s = case parseAllLang gr (startCat gr) s of
  (lg,t:_):_ -> unlines [linearize gr l t | l <- languages gr, l /= lg]
  _ -> "NO PARSE"
```

To run the translator, first compile it by

```
  % ghc -make -o trans Translator.hs
```

For this, you need the Haskell compiler GHC.

Producing PGF for the translator

Then produce a PGF file. For instance, the `Food` grammar set can be compiled as follows:

```
% gf -make FoodEng.gf FoodIta.gf
```

This produces the file `Food.pgf` (its name comes from the abstract syntax).

The Haskell library function `interact` makes the `trans` program work like a Unix filter, which reads from standard input and writes to standard output. Therefore it can be a part of a pipe and read and write files. The simplest way to translate is to `echo` input to the program:

```
% echo "this wine is delicious" | ./trans Food.pgf
questo vino è delizioso
```

The result is given in all languages except the input language.

A translator loop

To avoid starting the translator over and over again: change `interact` in the main function to `loop`, defined as follows:

```
loop :: (String -> String) -> IO ()
loop trans = do
  s <- getLine
  if s == "quit" then putStrLn "bye" else do
    putStrLn $ trans s
    loop trans
```

The loop keeps on translating line by line until the input line is `quit`.

A question-answer system

The next application is also a translator, but it adds a **transfer** component - a function that transforms syntax trees.

The transfer function we use is one that computes a question into an answer.

The program accepts simple questions about arithmetic and answers "yes" or "no" in the language in which the question was made:

```
Is 123 prime?
No.
77 est impair ?
Oui.
```

We change the pure translator by giving the `translate` function the transfer as an extra argument:

```
translate :: (Tree -> Tree) -> PGF -> String -> String
```

Ordinary translation as a special case where transfer is the identity function (`id` in Haskell).

To reply in the *same* language as the question:

```
translate tr gr = case parseAllLang gr (startCat gr) s of
  (lg,t:_):_ -> linearize gr lg (tr t)
  _ -> "NO PARSE"
```

Abstract syntax of the query system

Input: abstract syntax judgements

```
abstract Query = {

  flags startcat=Question ;

  cat
    Answer ; Question ; Object ;

  fun
    Even   : Object -> Question ;
    Odd    : Object -> Question ;
    Prime  : Object -> Question ;
    Number : Int -> Object ;

    Yes : Answer ;
    No  : Answer ;
}
```

Exporting GF datatypes to Haskell

To make it easy to define a transfer function, we export the abstract syntax to a system of Haskell datatypes:

```
% gf --output-format=haskell Query.pgf
```

It is also possible to produce the Haskell file together with PGF, by

```
% gf -make --output-format=haskell QueryEng.gf
```

The result is a file named `Query.hs`, containing a module named `Query`.

Output: Haskell definitions

```
module Query where
import PGF

data GAnswer =
   GYes
 | GNo

data GObject = GNumber GInt

data GQuestion =
   GPrime GObject
 | GOdd GObject
 | GEven GObject

newtype GInt = GInt Integer
```

All type and constructor names are prefixed with a `G` to prevent clashes.

The Haskell module name is the same as the abstract syntax name.

The question-answer function

Haskell's type checker guarantees that the functions are well-typed also with respect to GF.

```
answer :: GQuestion -> GAnswer
answer p = case p of
  GOdd x   -> test odd x
  GEven x  -> test even x
  GPrime x -> test prime x

value :: GObject -> Int
value e = case e of
  GNumber (GInt i) -> fromInteger i

test :: (Int -> Bool) -> GObject -> GAnswer
test f x = if f (value x) then GYes else GNo
```

Converting between Haskell and GF trees

The generated Haskell module also contains

```
class Gf a where
  gf :: a -> Tree
  fg :: Tree -> a

instance Gf GQuestion where
  gf (GEven x1) = DTr [] (AC (CId "Even")) [gf x1]
  gf (GOdd x1) = DTr [] (AC (CId "Odd")) [gf x1]
  gf (GPrime x1) = DTr [] (AC (CId "Prime")) [gf x1]
  fg t =
    case t of
      DTr [] (AC (CId "Even")) [x1] -> GEven (fg x1)
      DTr [] (AC (CId "Odd")) [x1] -> GOdd (fg x1)
      DTr [] (AC (CId "Prime")) [x1] -> GPrime (fg x1)
```

```
    _ -> error ("no Question " ++ show t)
```

For the programmer, it is enougo to know:

- all GF names are in Haskell prefixed with `G`
- `gf` translates from Haskell objects to GF trees
- `fg` translates from GF trees to Haskell objects

Putting it all together: the transfer definition

```
module TransferDef where

import PGF (Tree)
import Query   -- generated from GF

transfer :: Tree -> Tree
transfer = gf . answer . fg

answer :: GQuestion -> GAnswer
answer p = case p of
  GOdd x   -> test odd x
  GEven x  -> test even x
  GPrime x -> test prime x

value :: GObject -> Int
```

```
value e = case e of
  GNumber (GInt i) -> fromInteger i

test :: (Int -> Bool) -> GObject -> GAnswer
test f x = if f (value x) then GYes else GNo

prime :: Int -> Bool
prime x = elem x primes where
  primes = sieve [2 .. x]
  sieve (p:xs) = p : sieve [ n | n <- xs, n `mod` p > 0 ]
  sieve [] = []
```

Putting it all together: the Main module

Here is the complete code in the Haskell file `TransferLoop.hs`.

```
module Main where

import PGF
import TransferDef (transfer)

main :: IO ()
main = do
  gr <- readPGF "Query.pgf"
  loop (translate transfer gr)

loop :: (String -> String) -> IO ()
loop trans = do
  s <- getLine
```

```
  if s == "quit" then putStrLn "bye" else do
    putStrLn $ trans s
    loop trans

translate :: (Tree -> Tree) -> PGF -> String -> String
translate tr gr s = case parseAllLang gr (startCat gr) s of
  (lg,t:_):_ -> linearize gr lg (tr t)
  _ -> "NO PARSE"
```

Putting it all together: the Makefile

To automate the production of the system, we write a `Makefile` as follows:

```
all:
	gf -make --output-format=haskell QueryEng
	ghc --make -o ./math TransferLoop.hs
	strip math
```

(The empty segments starting the command lines in a Makefile must be tabs.) Now we can compile the whole system by just typing

```
make
```

Then you can run it by typing

```
./math
```

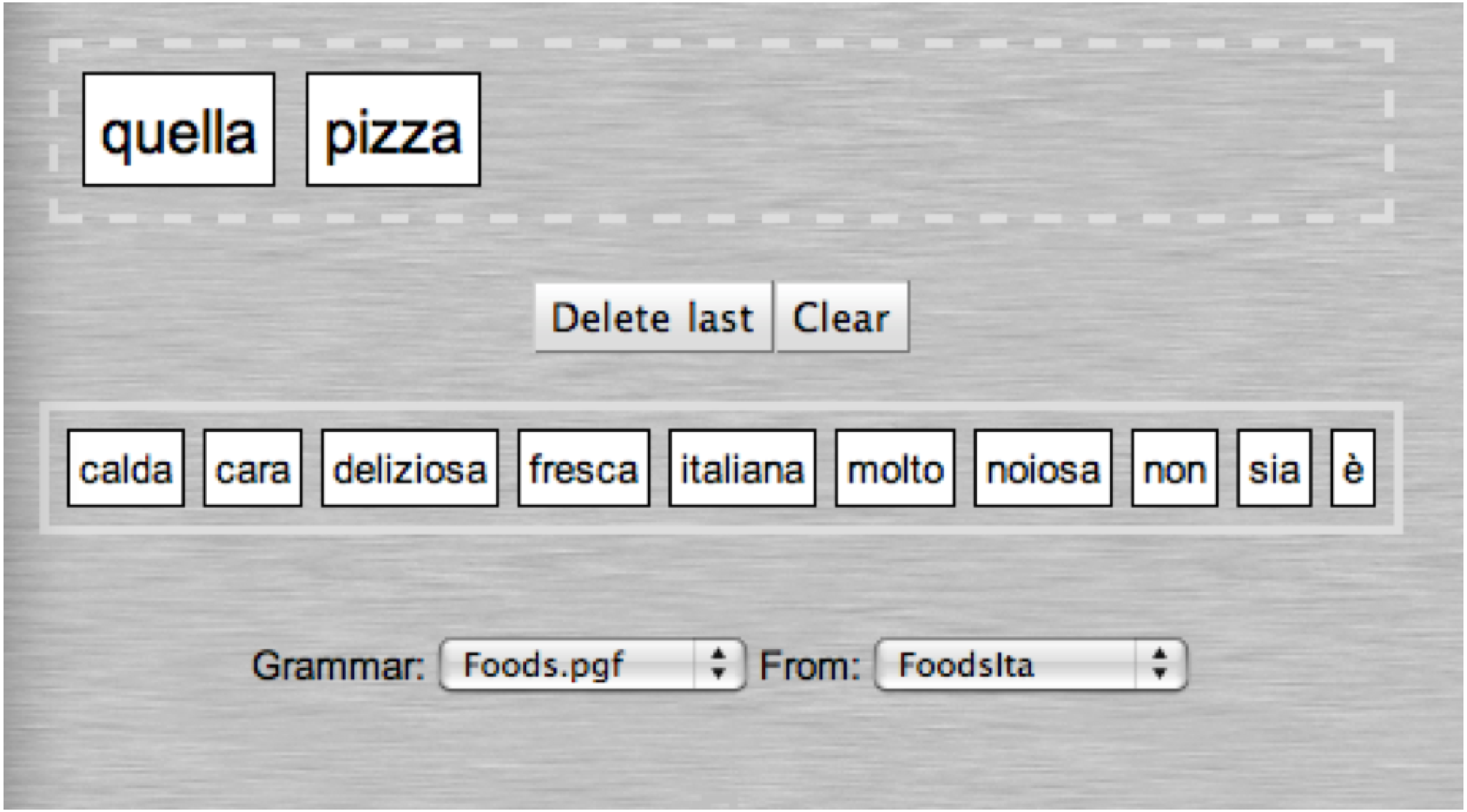
Just to summarize, the source of the application consists of the following files:

```
Makefile         -- a makefile
Math.gf          -- abstract syntax
Math???.gf       -- concrete syntaxes
TransferDef.hs   -- definition of question-to-answer function
TransferLoop.hs  -- Haskell Main module
```

Web server applications

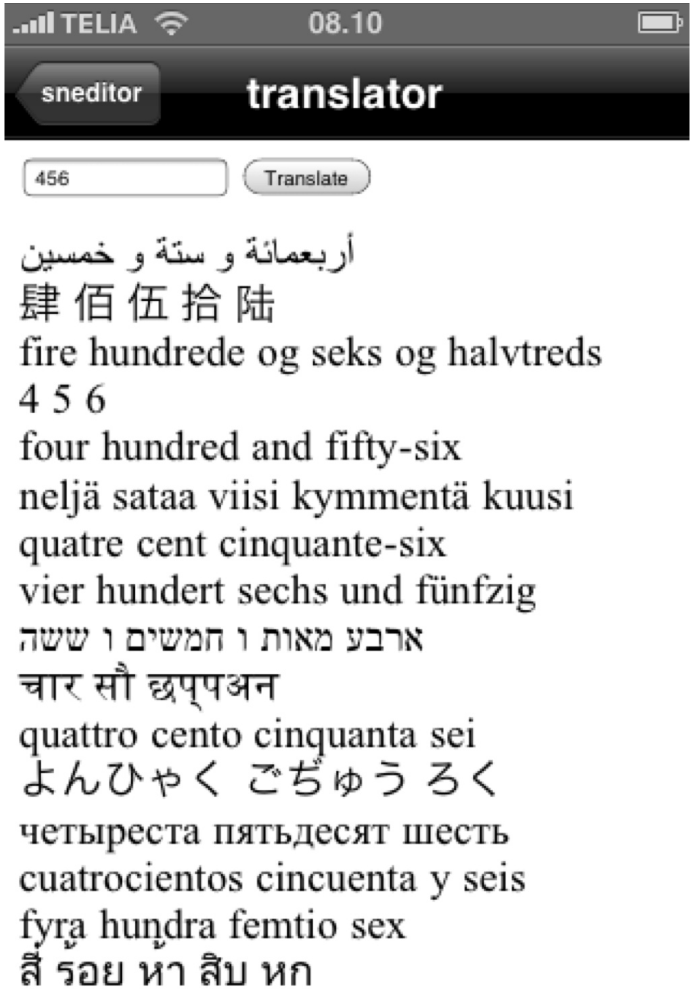
PGF files can be used in web servers, for which there is a Haskell library. How to launch a server is explained in

http://www.grammaticalframework.org/doc/gf-quickstart.html


quella
pizza
Delete last
Clear
calda
cara
deliziosa
fresca
italiana
molto
noiosa
non
sia
è
Grammar:
Foods.pgf
From:
FoodsIta

JavaScript applications

JavaScript is a programming language that has interpreters built in in most web browsers. It is therefore usable for client side web programs, which can even be run without access to the internet. The following figure shows a JavaScript program compiled from GF grammars as run on an iPhone.


TELIA
08.10
sneditor
translator
456
Translate
أربعمائة و ستة و خمسين
肆 佰 伍 拾 陆
fire hundrede og seks og halvtreds
4 5 6
four hundred and fifty-six
neljä sataa viisi kymmentä kuusi
quatre cent cinquante-six
vier hundert sechs und fünfzig
ארבע מאות ו חמשים ו ששה
चार सौ छप्पअन
quattro cento cinquanta sei
よんひゃく ごぢゅう ろく
четыреста пятьдесят шесть
cuatrocientos cincuenta y seis
fyra hundra femtio sex
สี่ ร้อย ห้า สิบ หก

Compiling to JavaScript

JavaScript is one of the output formats of the GF batch compiler. Thus the following command generates a JavaScript file from two `Food` grammars.

```
% gf -make --output-format=js FoodEng.gf FoodIta.gf
```

The name of the generated file is `Food.js`, derived from the top-most abstract syntax name. This file contains the multilingual grammar as a JavaScript object.

Language models for speech recognition

The standard way of using GF in speech recognition is by building **grammar-based language models**.

GF supports several formats, including GSL, the formatused in the Nuance speech recognizer.

GSL is produced from GF by running `gf` with the flag `--output-format=gsl`.

GSL generated from “FoodsEng.gf“

```
% gf -make --output-format=gsl FoodsEng.gf
% more FoodsEng.gsl

;GSL2.0
; Nuance speech recognition grammar for FoodsEng
; Generated by GF

.MAIN Phrase_cat

Item_1 [("that" Kind_1) ("this" Kind_1)]
Item_2 [("these" Kind_2) ("those" Kind_2)]
Item_cat [Item_1 Item_2]
Kind_1 ["cheese" "fish" "pizza" (Quality_1 Kind_1)
        "wine"]
Kind_2 ["cheeses" "fish" "pizzas"
```

```
        (Quality_1 Kind_2) "wines"]
Kind_cat [Kind_1 Kind_2]
Phrase_1 [(Item_1 "is" Quality_1)
          (Item_2 "are" Quality_1)]
Phrase_cat Phrase_1

Quality_1 ["boring" "delicious" "expensive"
           "fresh" "italian" ("very" Quality_1) "warm"]
Quality_cat Quality_1
```

More speech recognition grammar formats

Other formats available via the `--output-format` flag include:

Format	Description
`gsl`	Nuance GSL speech recognition grammar
`jsgf`	Java Speech Grammar Format (JSGF)
`jsgf_sisr_old`	JSGF with semantic tags in SISR WD 20030401 format
`srgs_abnf`	SRGS ABNF format
`srgs_xml`	SRGS XML format
`srgs_xml_prob`	SRGS XML format, with weights
`slf`	finite automaton in the HTK SLF format
`slf_sub`	finite automaton with sub-automata in HTK SLF

All currently available formats can be seen with `help print_grammar`.

Chapter 8: Interfacing formal and natural languages

Outline

- arithmetic expressions and precedences
- code generation as linearization
- strict vs. liberal abstract syntax
- natural language generation from logic
- logical semantics of natural language
- graftals: grammars for fractals

Arithmetic expressions

We construct a calculator with addition, subtraction, multiplication, and division of integers.

```
abstract Calculator = {

cat Exp ;

fun
  EPlus, EMinus, ETimes, EDiv : Exp -> Exp -> Exp ;
  EInt : Int -> Exp ;
}
```

The category `Int` is a built-in category of integers. Its syntax trees **integer literals**, i.e. sequences of digits:

```
5457455814608954681 : Int
```

These are the only objects of type `Int`: grammars are not allowed to declare functions with `Int` as value type.

Concrete syntax: a simple approach

We begin with a concrete syntax that always uses parentheses around binary operator applications:

```
concrete CalculatorP of Calculator = {

lincat
  Exp = SS ;
lin
  EPlus  = infix "+" ;
  EMinus = infix "-" ;
  ETimes = infix "*" ;
  EDiv   = infix "/" ;
  EInt i = i ;

oper
```

```
    infix : Str -> SS -> SS -> SS = \f,x,y ->
      ss ("(" ++ x.s ++ f ++ y.s ++ ")") ;
  }
```

Now we have

```
  > linearize EPlus (EInt 2) (ETimes (EInt 3) (EInt 4))
  ( 2 + ( 3 * 4 ) )
```

First problems:

- to get rid of superfluous spaces and
- to recognize integer literals in the parser

Precedence and fixity

Arithmetic expressions should be unambiguous. If we write

```
2 + 3 * 4
```

it should be parsed as one, but not both, of

```
EPlus (EInt 2) (ETimes (EInt 3) (EInt 4))
ETimes (EPlus (EInt 2) (EInt 3)) (EInt 4)
```

We choose the former tree, because multiplication has **higher precedence** than addition.

To express the latter tree, we have to use parentheses:

```
(2 + 3) * 4
```

The usual precedence rules

- Integer constants and expressions in parentheses have the highest precedence.
- Multiplication and division have equal precedence, lower than the highest but higher than addition and subtraction, which are again equal.
- All the four binary operations are **left-associative**: `1 + 2 + 3` means the same as `(1 + 2) + 3`.

Precedence as a parameter

Precedence can be made into an inherent feature of expressions:

```
oper
  Prec : PType = Ints 2 ;
  TermPrec : Type = {s : Str ; p : Prec} ;

  mkPrec : Prec -> Str -> TermPrec = \p,s -> {s = s ; p = p} ;

lincat
  Exp = TermPrec ;
```

Notice `Ints 2`: a parameter type, whose values are the integers `0,1,2`.

Using precedence levels

Compare the inherent precedence of an expression with the expected precedence.

- if the inherent precedence is lower than the expected precedence, use parentheses
- otherwise, no parentheses are needed

This idea is encoded in the operation

```
oper usePrec : TermPrec -> Prec -> Str = \x,p ->
  case lessPrec x.p p of {
    True  => "(" x.s ")" ;
    False => x.s
  } ;
```

(We use `lessPrec` from `lib/prelude/Formal`.)

Fixities

We can define left-associative infix expressions:

```
infixl : Prec -> Str -> (_,_ : TermPrec) -> TermPrec = \p,f,x,y ->
  mkPrec p (usePrec x p ++ f ++ usePrec y (nextPrec p)) ;
```

Constant-like expressions (the highest level):

```
constant : Str -> TermPrec = mkPrec 2 ;
```

All these operations can be found in `lib/prelude/Formal`, which has 5 levels.

Calculator compactly

```
concrete CalculatorC of Calculator = open Formal, Prelude in {

flags lexer = codelit ; unlexer = code ; startcat = Exp ;

lincat Exp = TermPrec ;

lin
  EPlus  = infixl 0 "+" ;
  EMinus = infixl 0 "-" ;
  ETimes = infixl 1 "*" ;
  EDiv   = infixl 1 "/" ;
  EInt i = constant i.s ;
}
```

Code generation as linearization

Translate arithmetic (infix) to JVM (postfix):

```
  2 + 3 * 4

    ===>

  iconst 2 : iconst 3 ; iconst 4 ; imul ; iadd
```

Just give linearization rules for JVM:

```
  lin
    EPlus  = postfix "iadd" ;
    EMinus = postfix "isub" ;
    ETimes = postfix "imul" ;
    EDiv   = postfix "idiv" ;
```

```
  EInt i = ss ("iconst" ++ i.s) ;
oper
  postfix : Str -> SS -> SS -> SS = \op,x,y ->
    ss (x.s ++ ";" ++ y.s ++ ";" ++ op) ;
```

Programs with variables

A **straight code** programming language, with **initializations** and **assignments**:

```
int x = 2 + 3 ;
int y = x + 1 ;
x = x + 9 * y ;
```

We define programs by the following constructors:

```
fun
  PEmpty : Prog ;
  PInit  : Exp -> (Var -> Prog) -> Prog ;
  PAss   : Var -> Exp  -> Prog  -> Prog ;
```

`PInit` uses higher-order abstract syntax for making the initialized variable available in the **continuation** of the program.

Example

Program code

```
int x = 2 + 3 ;
int y = x + 1 ;
x = x + 9 * y ;
```

Abstract syntax tree

```
PInit (EPlus (EInt 2) (EInt 3)) (\x ->
  PInit (EPlus (EVar x) (EInt 1)) (\y ->
    PAss x (EPlus (EVar x) (ETimes (EInt 9) (EVar y)))
      PEmpty))
```

Binding checked by HOAS

No uninitialized variables are allowed - there are no constructors for `Var`! But we do have the rule

```
fun EVar : Var -> Exp ;
```

The rest of the grammar is just the same as for arithmetic expressions #Rsecprecedence. The best way to implement it is perhaps by writing a module that extends the expression module. The most natural start category of the extension is `Prog`.

Exercises on code generation

1. Define a C-like concrete syntax of the straight-code language.

2. Extend the straight-code language to expressions of type `float`. To guarantee type safety, you can define a category `Typ` of types, and make `Exp` and `Var` dependent on `Typ`. Basic floating point expressions can be formed from literal of the built-in GF type `Float`. The arithmetic operations should be made polymorphic (as #Rsecpolymorphic).

3. Extend JVM generation to the straight-code language, using two more instructions

- `iload` x, which loads the value of the variable x
- `istore` x which stores a value to the variable x

Thus the code for the example in the previous section is

```
iconst 2 ; iconst 3 ; iadd ; istore x ;
iload x ; iconst 1 ; iadd ; istore y ;
iload x ; iconst 9 ; iload y ; imul ; iadd ; istore x ;
```

4. If you made the exercise of adding floating point numbers to the language, you can now cash out the main advantage of type checking for code generation: selecting type-correct JVM instructions. The floating point instructions are precisely the same as the integer one, except that the prefix is `f` instead of `i`, and that `fconst` takes floating point literals as arguments.

Generating graphics via graftals

Abstract syntax

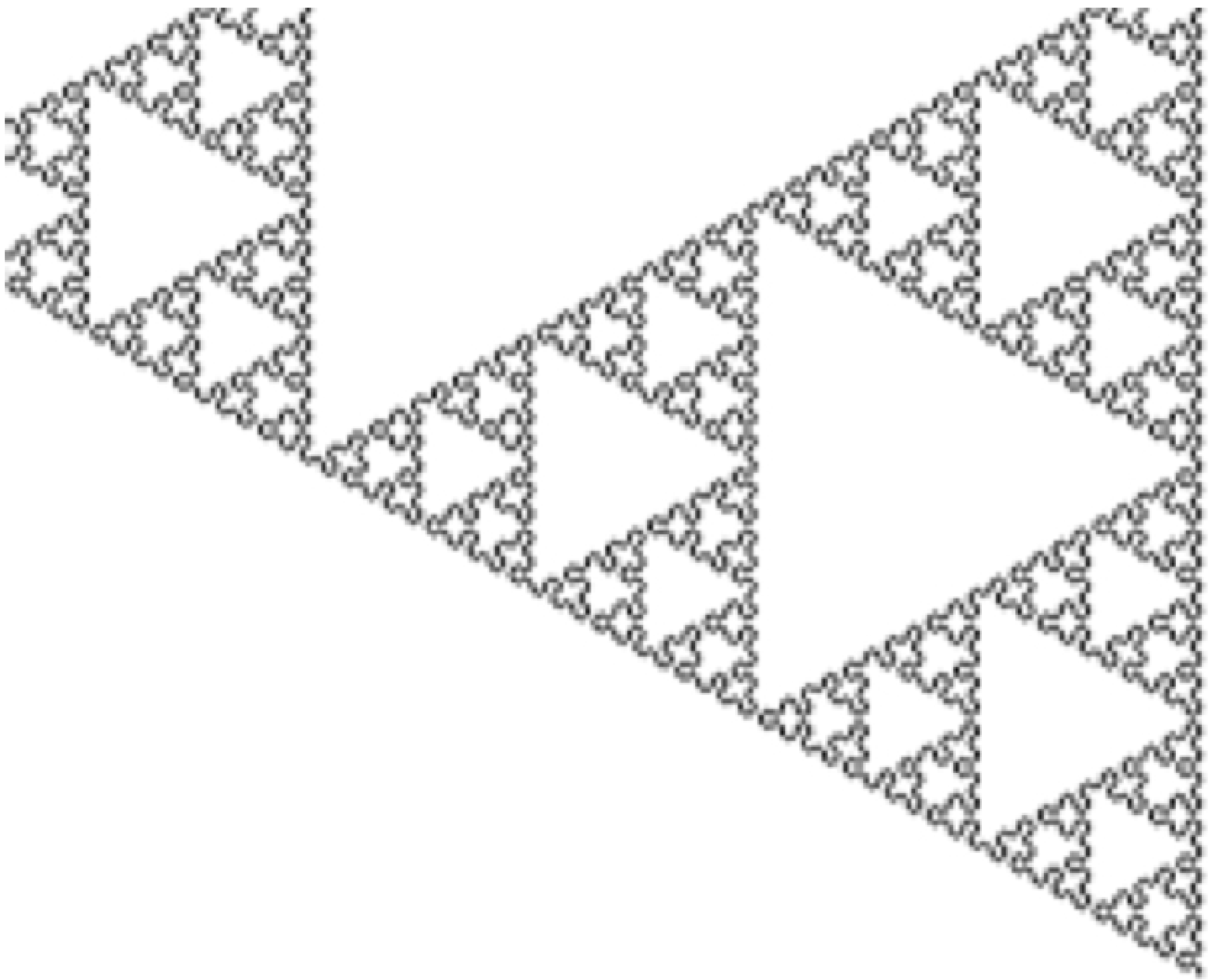
```
-- (c) Krasimir Angelov 2009
abstract Graftal = {
  cat N; S;
  fun z : N ;
      s : N -> N ;
      c : N -> S ;
  }
```

Concrete syntax: Sierpinski in PostScript

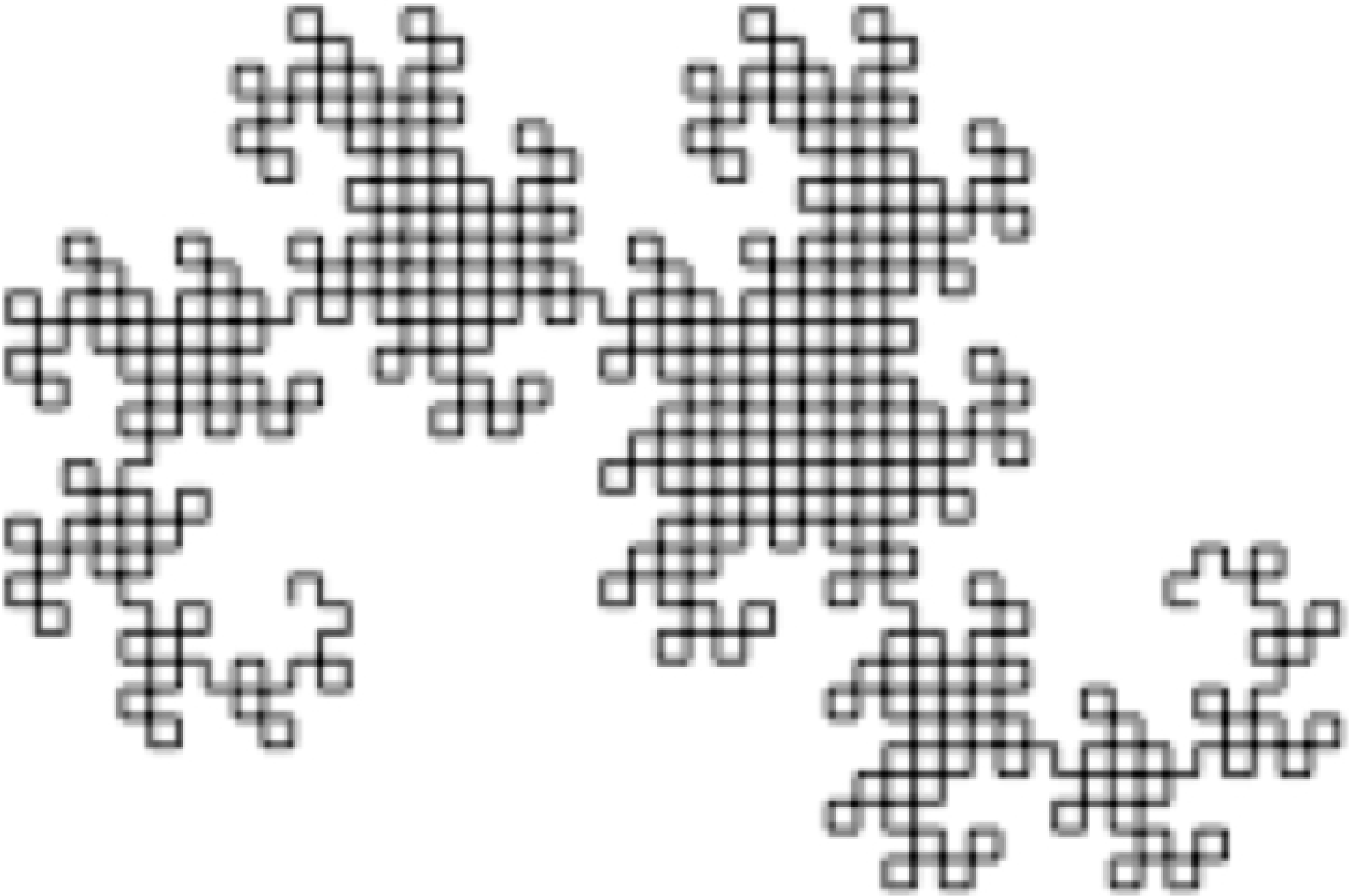
```
concrete Sierpinski of Graftal = {
  lincat N = {a : Str; b : Str} ;
  lincat S = {s : Str} ;

  lin z = {a = A; b = B} ;
  lin s x = {
    a = x.b ++ R ++ x.a ++ R ++ x.b ;
    b = x.a ++ L ++ x.b ++ L ++ x.a
    } ;
  lin c x = {s = "newpath 300 550 moveto" ++ x.a ++ "stroke showpage"} ;

  oper A : Str = "0 2 rlineto" ;
  oper B : Str = "0 2 rlineto" ;
  oper L : Str = "+60 rotate" ;
  oper R : Str = "-60 rotate" ;
}
```

An example as Sierpinski

```
c (s (s (s (s (s (s (s (s (s (s z))))))))))
```



An example as Dragon



Chapter 9: Getting started with resource grammar programming

Contents

The key categories and rules

Morphology-syntax interface

Examples and variations in English, Italian, French, Finnish, Swedish, German, Hindi

A miniature resource grammar: Italian

Module extension and dependency graphs

Ergativity in Hindi/Urdu

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.

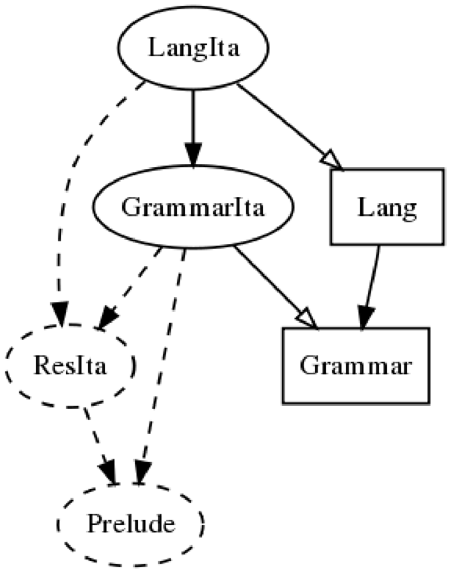
Linearization types are often specific and inflection are irregular.

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 Lang = Grammar **...                   -- test lexicon added to Grammar

resource ResIta                                 -- resource for Italian

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

concrete LangIta of Lang = GrammarIta ** open ResIta in... -- It. lexicon
```

Module dependencies



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

The module Grammar

```
abstract Grammar = {
  cat
    Cl ; NP ; VP ; AP ; CN ; Det ; N ; A ; V ; V2 ;
  fun
    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 ;
}
```

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 ;
```

Italian verb phrases

Tense and agreement of a verb phrase, in syntax

`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
    }
  } ;
```

To do

Full details of the core resource grammar are in `ResIta` (150 loc) and `GrammarIta` (80 loc).

One thing is not yet done correctly: agreement of participle to accusative clitic object: now it gives *io la ho amato*, and not *io la ho amata*.

This is left as an exercise!

Ergativity in Hindi/Urdu

Normally, the subject is nominative and the verb agrees to the subject.

However, in the perfective tense:

- the subject of a transitive verb is in an ergative "case" (particle *ne*)
- the verb agrees to the object

Example: "the boy/girl eats the apple/bread"

subj	**obj**	**gen. present**	**perfective**
Masc	Masc	*ladka: seb Ka:ta: hai*	*ladke ne seb Ka:ya:*
Masc	Fem	*ladka: roTi: Ka:ta: hai*	*ladke ne roTi: Ka:yi:*
Fem	Masc	*ladki: seb Ka:ti: hai*	*ladki: ne seb Ka:ya:*
Fem	Fem	*ladki: roTi: Ka:ti: hai*	*ladki: ne roTi: Ka:yi:*

A Hindi clause in different tenses

VPGenPres True	लड़की सेब खाती है
VPGenPres False	लड़की सेब नहीं खाती है
VPImpPast True	लड़की सेब खाती थी
VPImpPast False	लड़की सेब नहीं खाती थी
VPContPres True	लड़की सेब खा रही है
VPContPres False	लड़की सेब नहीं खा रही है
VPContPast True	लड़की सेब खा रही थी
VPContPast False	लड़की सेब नहीं खा रही थी
VPPerf True	लड़की ने सेब खाया
VPPerf False	लड़की ने सेब नहीं खाया
VPPerfPres True	लड़की सेब खायी है
VPPerfPres False	लड़की सेब नहीं खायी है
VPPerfPast True	लड़की सेब खायी थी
VPPerfPast False	लड़की सेब नहीं खायी थी
VPSubj True	लड़की सेब खाये
VPSubj False	लड़की सेब न खाये
VPFut True	लड़की सेब खायेगी
VPFut False	लड़की सेब न खायेगी

Exercises

1. Learn the commands `dependency_graph`, `print_grammar`, system escape `!`, and system pipe `?`.

2. Write tables of examples of the key syntactic functions for your target languages, trying to include all possible forms.

3. Implement `Grammar` and `Lang` for your target language.

4. Even if you don't know Italian, you *may* try this: add a parameter or something in `GrammarIta` to implement the rule that the participle in the perfect tense agrees in gender and number with an accusative clitic. Test this with the sentences *lei la ha amata* and *lei ci ha amati* (where the current grammar now gives *amato* in both cases).

5. Learn some linguistics! My favourite book is *Introduction to Theoretical Linguistics* by John Lyons (Cambridge 1968, at least 14 editions).

Chapter 10: Extending the Resource Grammar Library

Outline

Module structure

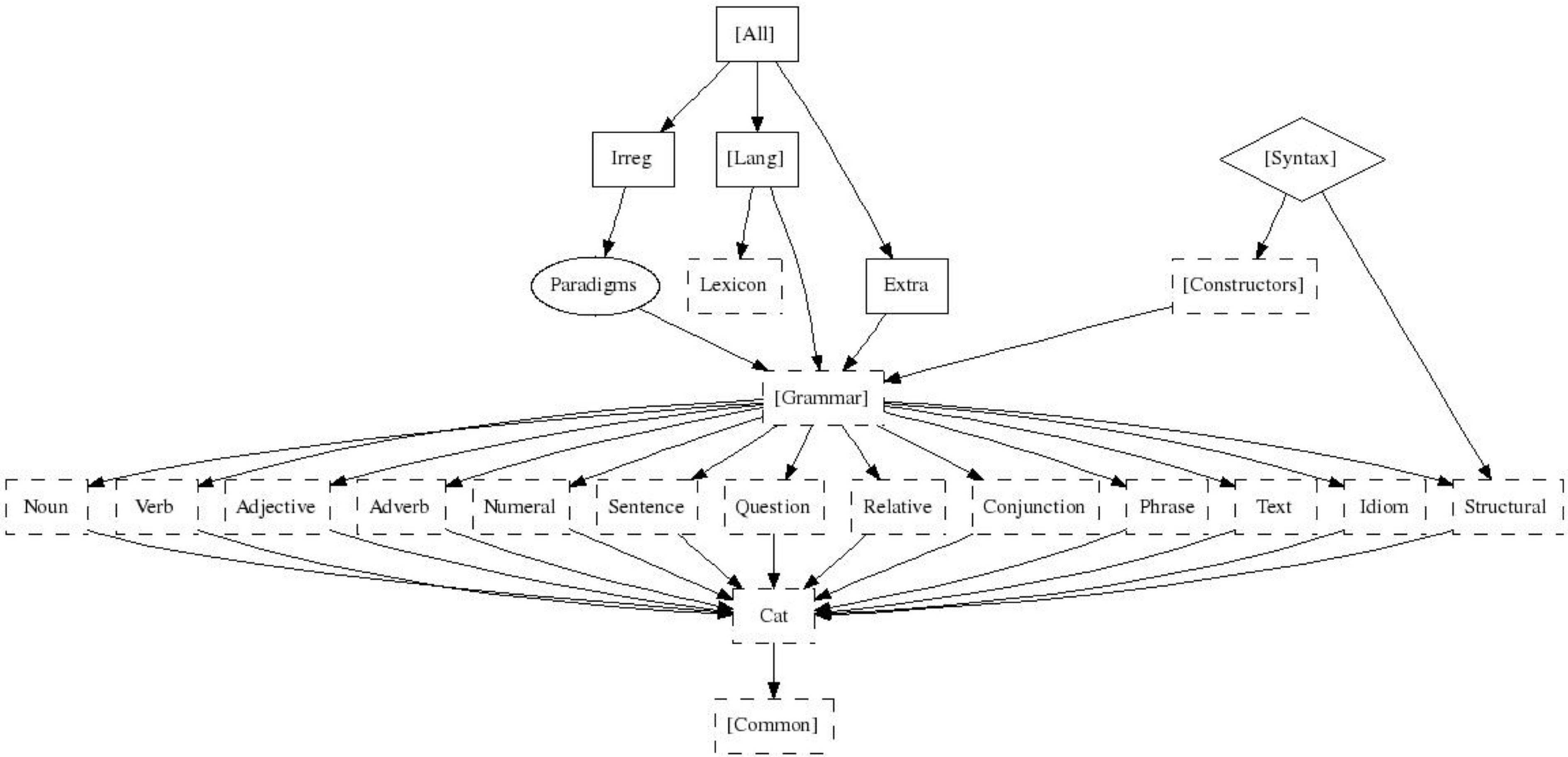
Statistics

How to start building a new language

How to test a resource grammar

The Assignment

The principal module structure



solid = API, dashed = internal, ellipse = abstract+concrete, rectangle = resource/instance, diamond = interface, green = given, blue = mechanical, red = to do

Division of labour

Written by the resource grammarian:

- concrete of the row from `Structural` to `Verb`
- concrete of `Cat` and `Lexicon`
- `Paradigms`
- abstract and concrete of `Extra`, `Irreg`

Already given or derived mechanically:

- all abstract modules except `Extra`, `Irreg`
- concrete of `Common`, `Grammar`, `Lang`, `All`
- `Constructors`, `Syntax`, `Try`

Roles of modules: Library API

`Syntax`: syntactic combinations and structural words

`Paradigms`: morphological paradigms

`Try`: (almost) everything put together

`Constructors`: syntactic combinations only

`Irreg`: irregularly inflected words (mostly verbs)

Roles of modules: Top-level grammar

`Lang`: common syntax and lexicon

`All`: common grammar plus language-dependent extensions

`Grammar`: common syntax

`Structural`: lexicon of structural words

`Lexicon`: test lexicon of 300 content words

`Cat`: the common type system

`Common`: concrete syntax mostly common to languages

Roles of modules: phrase categories

module	**scope**	**value categories**
`Adjective`	adjectives	`AP`
`Adverb`	adverbial phrases	`AdN, Adv`
`Conjunction`	coordination	`Adv, AP, NP, RS, S`
`Idiom`	idiomatic expressions	`Cl, QCl, VP, Utt`
`Noun`	noun phrases and nouns	`Card, CN, Det, NP, Num, Ord`
`Numeral`	cardinals and ordinals	`Digit, Numeral`
`Phrase`	suprasentential phrases	`PConj, Phr, Utt, Voc`
`Question`	questions and interrogatives	`IAdv, IComp, IDet, IP, QCl`
`Relative`	relat. clauses and pronouns	`RCl, RP`
`Sentence`	clauses and sentences	`Cl, Imp, QS, RS, S, SC, SSlash`
`Text`	many-phrase texts	`Text`
`Verb`	verb phrases	`Comp, VP, VPSlash`

Type discipline and consistency

Producers: each phrase category module is the producer of value categories listed on previous slide.

Consumers: all modules may use any categories as argument types.

Contract: the module `Cat` defines the type system common for both consumers and producers.

Different grammarians may safely work on different producers.

This works even for mutual dependencies of categories:

```
Sentence.UseCl  : Temp -> Pol -> Cl -> S  -- S uses Cl
Sentence.PredVP : VP -> NP -> Cl          --        uses VP
Verb.ComplVS    : VS -> S -> VP           --             uses S
```

Auxiliary modules

`resource` modules provided by the library:

- `Prelude` and `Predef`: string operations, booleans
- `Coordination`: generic formation of list conjunctions
- `ParamX`: commonly used parameter, such as `Number = Sg | Pl`

`resource` modules up to the grammarian to write:

- `Res`: language specific parameter types, morphology, VP formation
- `Morpho`, `Phono`,...: possible division of `Res` to more modules

Dependencies

Most phrase category modules:

```
concrete VerbGer of Verb = CatGer ** open ResGer, Prelude in ...
```

Conjunction:

```
concrete ConjunctionGer of Conjunction = CatGer **
  open Coordination, ResGer, Prelude in ...
```

Lexicon:

```
concrete LexiconGer of Lexicon = CatGer **
  open ParadigmsGer, IrregGer in {
```

Functional programming style

The Golden Rule: *Whenever you find yourself programming by copy and paste, write a function instead!*

- Repetition inside one definition: use a `let` expression

- Repetition inside one module: define an `oper` in the same module

- Repetition in many modules: define an `oper` in the `Res` module

- Repetition of an entire module: write a functor

Functors in the Resource Grammar Library

Used in families of languages

- Romance: Catalan, French, Italian, Spanish
- Scandinavian: Danish, Norwegian, Swedish

Structure:

- `Common`, a common resource for the family
- `Diff`, a minimal interface extended by interface `Res`
- `Cat` and phrase structure modules are functors over `Res`
- `Idiom`, `Structural`, `Lexicon`, `Paradigms` are ordinary modules

Example: DiffRomance

Words and morphology are of course different, in ways we haven't tried to formalize.

In syntax, there are just eight parameters that fundamentally make the difference:

Prepositions that fuse with the article (Fre, Spa *de*, *a*; Ita also *con*, *da*, *in*, *su*).

```
param Prepos ;
```

Which types of verbs exist, in terms of auxiliaries. (Fre, Ita *avoir*, *être*, and refl; Spa only *haber* and refl).

```
param VType ;
```

Derivatively, if/when the participle agrees to the subject. (Fre *elle est partie*, Ita *lei è partita*, Spa not)

```
oper partAgr : VType -> VPAgr ;
```

Whether participle agrees to foregoing clitic. (Fre *je l'ai vue*, Spa *yo la he visto*)

```
oper vpAgrClit : Agr -> VPAgr ;
```

Whether a preposition is repeated in conjunction (Fre *la somme de 3 et de 4*, Ita *la somma di 3 e 4*).

```
oper conjunctCase : NPForm -> NPForm ;
```

How infinitives and clitics are placed relative to each other (Fre *la voir*, Ita *vederla*). The `Bool` is used for indicating if there are any clitics.

```
oper clitInf : Bool -> Str -> Str -> Str ;
```

To render pronominal arguments as clitics and/or ordinary complements. Returns `True` if there are any clitics.

```
oper pronArg : Number -> Person -> CAgr -> CAgr -> Str * Str * Bool ;
```

To render imperatives (with their clitics etc).

```
oper mkImperative : Bool -> Person -> VPC -> {s : Polarity => AAgr => Str} ;
```

Pros and cons of functors

+ intellectual satisfaction: linguistic generalizations

+ code can be shared: of syntax code, 75% in Romance and 85% in Scandinavian

+ bug fixes and maintenance can often be shared as well

+ adding a new language of the same family can be very easy

– difficult to get started with proper abstractions

– new languages may require extensions of interfaces

Workflow: don't start with a functor, but do one language normally, and refactor it to an interface, functor, and instance.

Suggestions about functors for new languages

Romance: Portuguese probably using functor, Romanian probably independent

Germanic: Dutch maybe by functor from German, Icelandic probably independent

Slavic: Bulgarian and Russian are not functors, maybe one for Western Slavic (Czech, Slovak, Polish) and Southern Slavic (Bulgarian)

Fenno-Ugric: Estonian maybe by functor from Finnish

Indo-Aryan: Hindi and Urdu most certainly via a functor

Semitic: Arabic, Hebrew, Maltese probably independent

Effort statistics, completed languages

language	syntax	morpho	lex	total	months	started
common	413	-	-	413	2	2001
abstract	729	-	468	1197	24	2001
Bulgarian	1200	2329	502	4031	3	2008
English	1025	772	506	2303	6	2001
Finnish	1471	1490	703	3664	6	2003
German	1337	604	492	2433	6	2002
Russian	1492	3668	534	5694	18	2002
Romance	1346	-	-	1346	10	2003
Catalan	521	*9000	518	*10039	4	2006
French	468	1789	514	2771	6	2002
Italian	423	*7423	500	*8346	3	2003
Spanish	417	*6549	516	*7482	3	2004
Scandinavian	1293	-	-	1293	4	2005
Danish	262	683	486	1431	2	2005
Norwegian	281	676	488	1445	2	2005
Swedish	280	717	491	1488	4	2001
total	12545	*36700	6718	*55963	103	2001

Lines of source code in April 2009, rough estimates of person months. * = generated code.

How to start building a language, e.g. Marathi

1. Create a directory `GF/lib/src/marathi`

2. Check out the ISO 639-3 language code: `Mar`

3. Copy over the files from the closest related language, e.g. `hindi`

4. Rename files `marathi/*Hin.gf` to `marathi/*Mar.gf`

5. Change imports of `Hin` modules to imports of `Mar` modules

6. Comment out every line between *header* { and the final }

7. Now you can import your (empty) grammar: `i marathi/LangMar.gf`

Suggested order for proceeding with a language

1. `ResMar`: parameter types needed for nouns
2. `CatMar`: `lincat N`
3. `ParadigmsMar`: some regular noun paradigms
4. `LexiconMar`: some words that the new paradigms cover
5. (1.-4.) for `V`, maybe with just present tense
6. `ResMar`: parameter types needed for `Cl, CN, Det, NP, Quant, VP`
7. `CatMar`: `lincat Cl, CN, Det, NP, Quant, VP`
8. `NounMar`: `lin DetCN, DetQuant`
9. `VerbMar`: `lin UseV`
10. `SentenceMar`: `lin PredVP`

Character encoding for non-ASCII languages

GF internally: 32-bit unicode

Generated files (`.gfo`, `.pgf`): UTF-8

Source files: whatever you want, but use a flag if not isolatin-1.

UTF-8 and cp1251 (Cyrillic) are possible in strings, but not in identifiers. The module must contain

```
flags coding = utf8 ;  -- OR coding = cp1251
```

Transliterations are available for many alphabets (see `help unicode_table`).

Using transliteration

This is what you have to add in `GF/src/GF/Text/Transliterations.hs`

```
transHebrew :: Transliteration
transHebrew = mkTransliteration allTrans allCodes where
  allTrans = words $
    "A  b  g  d  h  w  z  H  T  y  K  k  l  M  m  N " ++
    "n  S  O  P  p  Z. Z  q  r  s  t  -  -  -  -  - " ++
    "w2 w3 y2 g1 g2"
  allCodes = [0x05d0..0x05f4]
```

Also edit a couple of places in `GF/src/GF/Command/Commands.hs`.

You can later convert the file to UTF-8 (see `help put_string`).

Diagnosis methods along the way

Make sure you have a compilable `LangMar` at all times!

Use the GF command `pg -missing` to check which functions are missing.

Use the GF command `gr -cat=C | l -table` to test category C

Regression testing with a treebank

Build and maintain a **treebank**: a set of trees with their linearizations:

1. Create a file `test.trees` with just trees, one by line.

2. Linearize each tree to all forms, possibly with English for comparison.

```
> i english/LangEng.gf
> i marathi/LangMar.gf
> rf -lines -tree -file=test.trees |
    l -all -treebank | wf -file=test.treebank
```

3. Create a **gold standard** `gold.treebank` from `test.treebank` by manually correcting the Marathi linearizations.

4. Compare with the Unix command `diff test.treebank gold.treebank`

5. Rerun (2.) and (4.) after every change in concrete syntax; extend the tree set and the gold standard after every new implemented function.

Sources

A *good* grammar book

- lots of inflection paradigms
- reasonable chapter on syntax
- traditional terminology for grammatical concepts

A *good* dictionary

- inflection information about words
- verb subcategorization (i.e. case and preposition of complements)

Wikipedia article on the language

Google as "gold standard": is it *rucola* or *ruccola*?

Google translation for suggestions (can't be trusted, though!)

Compiling the library

The current development library sources are in `GF/lib/src`.

Use `make` in this directory to compile the libraries.

Use `runghc Make lang api langs=Mar` to compile just the language `Mar`.

Assignment: a good start

1. Build a directory and a set of files for your target language.

2. Implement some categories, morphological paradigms, and syntax rules.

3. Give the `lin` rules of at least 100 entries in `Lexicon`.

4. Send us: your source files and a treebank of 100 trees with linearizations in English and your target language. These linearizations should be correct, and directly generated from your grammar implementation.