Based on the course named "Natural Language Technology", talks and tutorials given by Aarne Ranta (aarne [at] cs.chalmers.se)
Natural Language Technology and state of the art Technologies

- 6000-8000 living languages in the world
- Translation Systems: either limited or of low quality
- Dialogue Systems: both limited and of low quality
- Teaching: not used as much as it could be
- Web search: advanced for some languages but unknown
- Error messages: bad quality through "canned text": you have 1 message(s)
- Software localization: lists of canned text sentences
Natural vs programming languages

- Generally, **Grammar = Syntax + Semantics**

- For a programming language, the grammar is part of its specification and implementation

- Natural language is not defined by a grammar. Grammar of a NL is a research problem

- A part of a language technology application is often to solve a part of this research problem!
The Objective of these Tutorials

- Building some applications in three sub disciplines of NLT using GF
  - Morphology: theory of words and their forms
  - Syntax: theory of text and sentence structure
  - Semantics: theory of meaning
- Understanding what is needed for high-quality translation, dialogues etc and their specific solutions in GF
What will we cover

- Tutorial 1: Morphology & Lexicon
- Tutorial 2: Syntax and Translation systems
- Tutorial 3: Syntax, Translation and Formal Proofs
- Tutorial 4: Syntax and Semantics
What is GF?

- Grammar formalism based on type theory
- Special-purpose functional programming language having a powerful type system
- Fundamental structure:
  
  grammar = abstract syntax + concrete syntaxes

- Abstract syntax = semantic conditions (correct syntactic structures/trees of a language)
- Concrete syntax = mapping abstract syntax into strings along-with the grammatical features for a language (and back, by reversibility)
Morphology

- Part of speech or word class (Nouns, Verbs, Adjectives, Adverbs etc)
- GF follows word-and-paradigm model of morphology in which word forms are created by combining different morphs
- Inflection tables = Display all forms of a word.

Example: English regular nouns

<table>
<thead>
<tr>
<th></th>
<th>Singular</th>
<th>Plural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominative</td>
<td>rat</td>
<td>rats</td>
</tr>
<tr>
<td>Genitive</td>
<td>rat's</td>
<td>rats'</td>
</tr>
</tbody>
</table>
Stems, endings, morphs, morphemes

A word form can often be analysed to parts:
- Undressed --- un- (prefix), dress (stem), -ed (suffix)
- Carelessness --- care (stem), -less (suffix), -ness (suffix)

All these significant parts are called morphs.

A morpheme is an abstraction over different morphs that have the same function. For instance, s and es are variants of the plural morpheme:
- boy + s, kiss + es

Morphological analysis = analysis into morphemes (in the abstract sense of parameter description)
- boys --> boy +Nom +Pl
- babies' --> baby +Gen +Pl

Thus analysis = lemma + tags
Parameters

The different form descriptions are grouped into types.
- Examples of such types and their values:
  - **number**: singular, plural (Arabic also: dual)
  - **gender**: masculine, feminine (French, Arabic, Urdu/Hindi) / masculine, feminine, neuter (German, Latin, English)
  - **case**: nominative, genitive (Swedish) / nominative, accusative, dative, genitive (German)

Heavily dependent on language!

A word class is morphologically defined by telling what type of parameters its forms depend on.

**Parametric vs. inherent features**

- To define a word class, we should also tell what inherent features attach to it. Cf. class in Java:
  - **method**: inflection for different combinations of parameters
  - **attributes**: inherent features
Defining morphology of a language

- **Type system**: define parameter types and word classes
- **Inflection engine**: define all paradigms for all word classes
- **Lexicon**: list all words with their word classes and paradigms.

The definitions can be made with stg like 100 + 1000 + 10000 lines of code, for a "medium hard" language like French.

- **English** needs less types and paradigms, but more lexicon.
Uses of a morphology

- **Synthesis**: given a dictionary word, generate inflection table.
- **Analysis**: given a word form, return lemma, word class, and form description (which can be ambiguous)

Implementing morphology

- General-purpose programming languages: Haskell, Caml, Java, C,... need to define the types and data structures of the type system, the inflection engine, and the lexicon. And also an analysis program! ex. **Functional Morphology** (a Haskell library for morphology development).
- Special-purpose morphology languages. The most well-known: **XFST**, based on regular expressions.
- **GF**, Further it extends seamlessly from morphology to syntax and semantics.
Implementation
Defining parameter types in GF

- Judgements identified by the keyword `param`.
- Examples:
  ```
  param Number = Sg | Pl ;
  param Gender = Masc | Fem | Neutr ;
  param Case = Nom | Acc | Gen | Dat ;
  ```
Tables in GF

- Finite functions on parameter types. Have a type of the form $P \Rightarrow A$.
- Example:
  ```
  table {
    Sg => "mouse" ;
    Pl => "mice"
  } : Number => Str
  ```

- Tables of tables. e.g: an English regular noun
  ```
  table {
    Sg => table {
      Nom => "rat" ;
      Gen => "rat's"
    } ;
    Pl => table {
      Nom => "rats" ;
      Gen => "rats"
    }
  } : Number => Case => Str
  ```
Wild cards in tables

Example: German "weak masculine" noun, singular forms

```
<table>
<thead>
<tr>
<th>Case</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nom</td>
<td>&quot;Kommunist&quot;</td>
</tr>
<tr>
<td>Acc</td>
<td>&quot;Kommunisten&quot;</td>
</tr>
<tr>
<td>Gen</td>
<td>&quot;Kommunisten&quot;</td>
</tr>
<tr>
<td>Dat</td>
<td>&quot;Kommunisten&quot;</td>
</tr>
</tbody>
</table>
```

This expands to a table with values for every case:

```
<table>
<thead>
<tr>
<th>Case</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nom</td>
<td>&quot;Kommunist&quot;</td>
</tr>
<tr>
<td>Acc</td>
<td>&quot;Kommunisten&quot;</td>
</tr>
<tr>
<td>Gen</td>
<td>&quot;Kommunisten&quot;</td>
</tr>
<tr>
<td>Dat</td>
<td>&quot;Kommunisten&quot;</td>
</tr>
</tbody>
</table>
```
Selection from a table

- Application of the finite function. Notation: !

Examples:

- table {Sg => "mouse" ; Pl => "mice"}! Sg
  ---> "mouse"

- table {Nom => "Kommunist" ; _ => "Kommunisten"}! Dat
  ---> "Kommunisten"

- table {
    Sg => table {Nom => "Kommunist" ; _ => "Kommunisten"} ;
    Pl => table { _ => "Kommunisten"}
  }! Pl
  ---> table { _ => "Kommunisten"}
Operation definitions in GF

- The keyword `oper` is used to introduce a constant of any type with any definition. Similar to function definitions in Haskell:

```haskell
oper fish : Str = "fish" ;
oper genitive : Case = Gen ;
oper Mouse : Num => Str = table {
  Sg => "mouse" ;
  Pl => "mice"
} ;
```
Functions in GF

- The argument type can be any type, not just a parameter.
- Syntax is like in Haskell, including lambda abstraction.
- Example: English regular noun inflection

```
oper regNoun : Str -> Number => Case => Str = \rat ->
table {
  Sg => table {
    Nom => rat ;
    Gen => rat + "s"
  } ;
  Pl => table {
    Nom => rat + "s" ;
    Gen => rat + "s"
  }
}
```

Notice: strings are written with quotes, variables without quotes.
Function application

Syntax like in Haskell, by juxtaposition:

```plaintext
regNoun "rat" --->
  table {
    Sg => table {
      Nom => "rat" ;
      Gen => "rat's"
    } ;
    Pl => table {
      Nom => "rats" ;
      Gen => "rats''"
    }
  } : Number => Case => Str
```
Parametric vs. inherent features in GF

- We use records and record types to collect all information. Example: English nouns:
  
  ```
  oper Noun : Type = { s : Number => Case => Str ;
                     g : Gender
                   } ;
  ```

- Example object of this type
  
  ```
  {s = table {
    Sg => table { Nom => "rat" ; Gen => "rat's" } ;
    Pl => table { Nom => "rats" ; Gen => "rats'" }
  };
  g = Masc
  }
  ```

- Record projection (.) gives access to fields:
  
  ```
  {s = ... ; g = Masc}.g ---> Masc
  ```
Morphology in GF

Type system: define parameter types and word classes. e.g.

\[
\text{param } \text{Number} = \text{Sg} | \text{Pl} ; \\
\text{param } \text{Case} = \text{Nom} | \text{Gen} ;
\]

\[
\text{oper } \text{Noun} : \text{Type} = \{s : \text{Number} => \text{Case} => \text{Str}\} ; \text{-- simplified!}
\]

\[
\text{oper } \text{Verb} : \text{Type} = \{s : \text{Number} => \text{Str}\} ; \text{-- simplified!}
\]

Inflection engine: define inflection paradigms. e.g.

\[
\text{oper } \text{regNoun} : \text{Str} -> \text{Noun} = \lambda \text{rat} -> \\
\{s = \text{table} \\
\text{Sg} => \text{table} \\
\text{Nom} => \text{rat} ;
\text{Gen} => \text{rat} + "'s"
\} ; \\
\text{Pl} => \text{table} \\
\text{Nom} => \text{rat} + "s" ;
\text{Gen} => \text{rat} + "s'"
\}
\]

\[
\text{oper } \text{regVerb} : \text{Str} -> \text{Verb} = \lambda \text{walk} -> \\
\{s = \text{table} \\
\text{Sg} => \text{walk} + "s" ;
\text{Pl} => \text{walk}
\}
\]

\[
\lambda x -> t \text{ -- lambda abstract}
\]

\[
\lambda x,y -> t \text{ -- same as } \lambda x -> \lambda y -> t
\]
Morphology in GF continued.

Lexicon: define lemma types, lemmas, and their mappings to inflection tables.
This requires four new judgment forms:

- $\text{fun Rat : N ;}$
- $\text{fun House : N ;}$
- $\text{fun Look : V ;}$
- $\text{fun Resubmit : V ;}$

$\text{lin Rat = regNoun "rat" ;}$
$\text{lin House = regNoun "house" ;}$
$\text{lin Look = regVerb "look" ;}$
$\text{lin Resubmit = regVerb "resubmit" ;}$
Using GF

- We will use GF as an interpreter. Start it by `gf` in the OS shell, and you enter into a GF shell, where several commands are available.
- Here are some GF commands that are useful in morphology:
  - import: `i File` - load new grammar to environment
  - empty: `e` - reset environment to empty
  - show table: `l -table Fun` - show full inflection table
  - morphological analysis: `ma String` - give morphological analysis
  - morphological quiz: `mq -cat=Cat` - generate morphological exercises
- Let's run our developed morphology module.
The Module system of GF

Abstract syntax:
EngBasicAbs.gf

abstract EngBasicAbs = {
  cat N ;
  ...
  fun Rat, House : N;
  ...
}

Concrete syntax:
EngBasic.gf

concrete EngBasic of EngBasicAbs =
  open ResEngBasic in {
    lincat N = Noun ;
    ..... 
    lin Rat = regNoun "rat" ;
    ..... 
  }

Resource File:
ResEngBasic.gf

resource ResEngBasic = {
  param Number = Sg | Pl ;
  ....
  oper
    regNoun : Str -> Noun =
    ....
  };
  ....
  ....
}
Steps for developing a morphology from scratch for a bigger implementation

- Starting with the type system for the major word classes, in this order:
  1. nouns
  2. adjectives
  3. verbs
- Identifying what different forms are needed for nouns.
- It is often the case that all noun forms are also adjective forms, and that adjective forms are also verb forms (participles).
- In the inflection engine, starting with a worst-case function for each word class, and continue by special cases of it.
- Leaving the closed classes last.
Running example: English Morphology

A type system and worst-case function for English common nouns

**param**

Number = Sg | Pl ;
Case = Nom | Gen ;

**oper**

CommonNoun : Type = \{s : Number => Case => Str ; g : Gender \} ;
ProperName : Type = \{s : Case => Str ; g : Gender \} ;

mkNoun : (_,_,_,_ : Str) -> \{s : Number => Case => Str \} =
\{man,men,mans,mens ->
{s = table {
Sg => table \{Nom => man ; Gen => mans\} ;
Pl => table \{Nom => men ; Gen => mens\}
}
}
"}
Running example: English Morphology

- Regular paradigms for English common nouns

  \[
  \text{nounReg} : \text{Str} \rightarrow \{ s : \text{Number} \Rightarrow \text{Case} \Rightarrow \text{Str} \} = \lambda \text{dog} \rightarrow \\
  \text{mkNoun} \text{dog} (\text{dog} + "s") (\text{dog} + "'s") (\text{dog} + "s'") ;
  \]

- These paradigms are "almost regular":

  -- nouns with plural ending "es"

  \[
  \text{nounS} : \text{Str} \rightarrow \{ s : \text{Number} \Rightarrow \text{Case} \Rightarrow \text{Str} \} = \lambda \text{kiss} \rightarrow \\
  \text{mkNoun} \text{kiss} (\text{kiss} + "es") (\text{kiss} + "'s") (\text{kiss} + "es'") ;
  \]

  -- nouns ending with "y"

  \[
  \text{nounY} : \text{Str} \rightarrow \{ s : \text{Number} \Rightarrow \text{Case} \Rightarrow \text{Str} \} = \lambda \text{fl} \rightarrow \\
  \text{mkNoun} (\text{fl} + "y") (\text{fl} + "ies") (\text{fl} + "y's") (\text{fl} + "ies'") ;
  \]
Running example: English Morphology
Analysing the stem

Very typically, the paradigm can be predicted from the stem. Here is general common noun inflection operation for English.

\[
nounGen : \text{Str} \rightarrow \{ \text{s : Number} \Rightarrow \text{Case} \Rightarrow \text{Str} \} = \text{\$} \rightarrow \text{case last s of}
\]

\[
\{ \\
"s" \Rightarrow \text{nounS s} ; \\
"y" \Rightarrow \text{nounY (init s)} ; \\
_ \Rightarrow \text{nounReg s}
\}
\]

But observe: nounGen does not always predict right.

auxiliary functions:

last : \text{Str} \rightarrow \text{Str} = \text{Predef.dp 1} ; -- last char
init : \text{Str} \rightarrow \text{Str} = \text{Predef.tk 1} ; -- drop last char

Predef library functions:

tk : \text{Int} \rightarrow \text{Tok} \rightarrow \text{Tok} ; -- take all but the n last chars
dp : \text{Int} \rightarrow \text{Tok} \rightarrow \text{Tok} ; -- drop all but the n last chars
Running example: English Morphology

Regular expression patterns

Instead of taking apart suffixes, it is possible to match with regular expression patterns:

```markdown
mkN : Str -> {s : Number => Case => Str } = \car -> case car of {
  _ + ("ay"|"ey"|"oy"|"uy") => -- day, valley, boy, guy
    mkNoun car (car+"s") (car+"s") (car+"s") ;
  fl + "y" => -- copy, lobby
    mkNoun car (fl+"ies") (car+"s") (fl+"ies") ;
  _ + ("s"|"x"|"sh"|"ch") => -- kiss, box, bush, bench
    mkNoun car (car+"es") (car+"s") (car+"es") ;
  _ =>
    mkNoun car (car+"s") (car+"s") (car+"s")
}
```

This function covers more cases than nounGen, because it inspects longer suffixes than just the last character.
Summary of regular expression patterns in GF

- $p + q$: token consisting of $p$ followed by $q$
- $p^*$: token $p$ repeated 0 or more times (max the length of the string to be matched)
- $-p$: matches anything that $p$ does not match
- $x@p$: bind to $x$ what $p$ matches
- $p | q$: matches what either $p$ or $q$ matches
Example of binding: Swedish second declension

This is not the whole paradigm, but just plural formation:
pojke-pojkar, nyckel-nycklar, segersegrar, bil-bilar:

plural2 : Str -> Str = \w -> case w of {
    pojk + "e" => pojk + "ar" ;
    nyck + "e" + l@("l" | "r" | "n") => nyck + l + "ar" ;
    bil => bil + "ar"
} ;
Running example: English Morphology

general functions for Noun

\[\text{regN} : \text{Str} \rightarrow \text{CommonNoun} = \lambda \text{car} \rightarrow \text{mkN} \text{car} \quad \{g = \text{Neutr}\};\]

\[\text{reg4N} : (\_,\_,\_,\_\_ : \text{Str}) \rightarrow \text{CommonNoun} = \lambda \text{man, men, mans, mens} \rightarrow \text{mkNoun man men mans mens} \quad \{g = \text{Neutr}\};\]

\[\text{genderN} : \text{Gender} \rightarrow \text{Str} \rightarrow \text{CommonNoun} = \lambda g, \text{car} \rightarrow \text{mkN} \text{car} \quad \{g = g\};\]

\[\text{gender4N} : \text{Gender} \rightarrow (\text{man, men, mans, mens} : \text{Str}) \rightarrow \text{CommonNoun} = \lambda g, \text{man, men, mans, mens} \rightarrow \text{mkNoun man men mans mens} \quad \{g = g\};\]

\[\text{e.g. } \text{regN} \ "\text{rat}"; \]

\[\text{e.g. } \text{gender4N Masc } \text{"man" } \text{"men" } \text{"man's" } \text{"men's"}; \]
Running example: English Morphology

general functions for Noun

\[ \text{regN} : (\_,\_,\_,\_,\_ : \text{Str}) \rightarrow \text{CommonNoun} = \\{ \text{man, men, mans, mens} \rightarrow \{ \text{s = table} \{ \\
\text{Sg} \rightarrow \text{table} \{ \\
\text{Nom} \rightarrow \text{man} ; \\
\text{Gen} \rightarrow \text{mans} \\
\}, \\
\text{Pl} \rightarrow \text{table} \{ \\
\text{Nom} \rightarrow \text{men} ; \\
\text{Gen} \rightarrow \text{mens} \\
\} \\
g = \text{Neutr} \} \}; \]

Running example: English Morphology
Adjectives

Adjectives have comparison forms: positive (small), comparative (smaller), superlative (smallest).

There is a productive formation of adverbials from adjectives, regularly with the ending ly (free- freely).

\[
\begin{align*}
\text{param Degree} &= \text{Posit} \mid \text{Compar} \mid \text{Superl} ; \\
\text{param AForm} &= \text{AAdj Degree} \mid \text{AAdv} ; \\
\text{oper Adjective} : \text{Type} &= \{s : \text{AForm} \Rightarrow \text{Str}\} ;
\end{align*}
\]

\[
\text{oper mkADeg} : (\_,\_,\_,\_,\_ : \text{Str}) \Rightarrow \text{Adjective} = \text{\{good, better, best, well \}} \Rightarrow \\
\{s = \text{table} \{
    \text{AAdj Posit} \Rightarrow \text{good} ; \\
    \text{AAdj Compar} \Rightarrow \text{better} ; \\
    \text{AAdj Superl} \Rightarrow \text{best} ; \\
    \text{AAdv} \Rightarrow \text{well}
\}\} ;
\]
Running example: English Morphology

Adjectives

```haskell
regADeg : Str -> Adjective =
    let
        happ = init happy ;
        y = last happy ;
        happie = case y of {
            "y" => happ +"ie" ; -- happy, heavy
            "e" => happy ; -- close
            _ => "e" --clever
        } ;
        happily = case y of {
            "y" => happ +"ily" ; -- happy
            _ => happy +"ly" -- closely, cleverly
        } ;
    in mkADeg happy (happie + "r") (happie + "st") happily ;
```
e.g. regADeg "happy" ;

E.g. regADeg "happy" ;
Running example: English Morphology
Adjectives

-- adverbials from adjectives
-- free-freely

e.g. regA "free" ;

regA : Str -> Adjective = \free ->
  let
    e   = last free ;
    fre = init free ;
    freely = case e of {
      "y" => fre + "ily" ;
      _   => free + "ly"
    }
  in mkADeg free [] [] freely ;

compoundADeg : Str -> Adjective =\beautiful ->
  let
    beautifully =case last beautiful of {
      "y" => init beautiful +"ily" ;
      _   => beautiful +"ly"
    } ;
  in mkADeg beautiful
    ("more" ++ beautiful)
    ("most" ++ beautiful)
    (beautifully) ;
Running example: English Morphology

Verb types in English

- The worst-case variation is shown by the six forms of the verb be:
  (I) am, (we,you,they) are, (he,she) is,
  (I,he,she) was, (we,you,they) were,
  been,
  be

- We choose to put 'be' into a special category, and then need only five verb forms:

  ```
  param VForm = Pres Number | Past | PastPart | PresPart ;
  oper Verb : Type = {s : VForm => Str} ;
  ```

- Notice the constructor Pres of VForm, which takes a Number as argument. The possible values are thus:

  Pres Sg, Pres Pl, Past, PastPart, PresPart
Running example: English Morphology

Verb inflection in English: worst case

\[
\text{mkVerb : (_,_,_,_,_,: Str) -> Verb = \{\text{go,goes,went,gone,going} ->}
\{\text{s = table} \{
    \text{Pres Pl} \Rightarrow \text{go} ;
    \text{Pres Sg} \Rightarrow \text{goes} ;
    \text{Past} \Rightarrow \text{went} ;
    \text{PastPart} \Rightarrow \text{gone} ;
    \text{PresPart} \Rightarrow \text{going}
\}}
\]
Running example: English Morphology
Verb inflection in English: regular verbs

regV : Str -> Verb = \cry ->

let
    crie = init ((regN cry).s ! Pl ! Nom) ;

cried : Str = case crie of {
    _ + "e"   => crie + "d" ;      -- use-d
    _     => crie + "ed"          -- talk-ed
}

    crying : Str = case cry of {
    _ + "ee" => cry + "ing" ;     -- free-ing
    us + "e" => us + "ing" ;      -- us-ing
    _     => cry + "ing"         -- cry-ing
}

     in mkVerb cry (crie + "s") cried cried crying ;
Running example: English Morphology

Verb inflection in English: irregular verbs

\[
\text{irregV} : (x1, _, x3 : \text{Str}) \rightarrow \text{Verb} = \{\text{sing}, \text{sang}, \text{sung} \rightarrow \\
\text{let } \text{sings} = (\text{regV sing}).s \\
\text{in} \\
\text{mkVerb} \\
\text{(sings} \! \text{ Pres Pl)} \\
\text{(sings} \! \text{ Pres Sg)} \\
\text{sang} \\
\text{sung} \\
\text{(sings} \! \text{ PresPart}) \\
\text{);}
\]

- Do we cover all verbs with \(\text{regV}\) and \(\text{irregV}\) ?

\(\text{e.g. irregV } \text{"come" } \text{"came" } \text{"come"} \);
Running example: English Morphology

Final consonant duplication: problem

We don't cover all verbs yet:

> cc regV "fit"

... PastPart => "fited" ;
PresPart => "fiting"

■ The proper forms are fitted, fitting.
■ Can we make a case for final t?
■ No! Compare omit - omitting and vomit - vomiting.
■ The problem is that we need to know if the final syllable is stressed, and the English orthography does not tell us this.
■ This is a very common problem in the morphology of written language.
Running example: English Morphology

Final consonant duplication: solution

def regDuplV : Str -> Verb = \fit ->
  case last fit of {
    ("a" | "e" | "i" | "o" | "u" | "y") =>
      Predef.error (["final duplication makes no sense for"] ++ fit) ;
    t =>
      let fitt = fit + t in
      mkVerb fit (fit + "s") (fitt + "ed") (fitt + "ed") (fitt + "ing")
  } ;

def irregDuplV : (_,_,_ : Str) -> Verb = \fit,y,z ->
  let fitting = (regDuplV fit).s ! VPresPart
  in mkVerb fit (fit+"s") y z fitting ;

e.g. regDuplV "stop" ;
e.g. irregDuplV "run" "ran" "run" ;
The lexicon writer then has to choose

\[
\begin{align*}
\text{omit}_V &= \text{duplRegV} \ "omit" ; \\
\text{vomit}_V &= \text{regV} \ "vomit" ; \\
\text{get}_V &= \text{duplIrregV} \ "get" \ "got" \ "got" ; \\
\text{sing}_V &= \text{irregV} \ "sing" \ "sang" \ "sung" ;
\end{align*}
\]
Running example: English Morphology
some other word classes

- **Personal pronouns**: I/me/my/mine
- **Demonstrative pronouns**: it/its, this
- **Relative pronouns**: who/whom/whose, which, that
- **Interrogative pronouns**: who/whom/whose, which, what
- **Numerals**: three, third; inflected like proper names in nominative and genitive case.

- The other classes are not inflected: prepositions, conjunctions,...
Let's examine and execute the English morphology resource
The Lexicon Extraction

- group word-forms that belong to a paradigm.
  - For instance: if we find *galumph*, *galumphs*, *galumphed*, *galumphing*, then introduce *galumph* as a regular verb.

Starting point:
- word list compiled from corpus
- a morphological type system and set of paradigms

Method:
- for each word in the list, find out which paradigms it fits
- for each such paradigm, test if the other forms are also in the list
- if successful, add a lexical entry and test each entry (usually a good idea)
The Lexicon Extraction

- The tool **extract** can be used to automatically extract the Lexicon with minimal human effort.

- The tool requires a **paradigm file** and a **corpus**

```plaintext
regexp Char = letter;
regexp Word = letter+;

-- rat
paradigm noun =
  x { x & (x+"s" | x+"s") };

-- party
paradigm nounY =
  x+"y" { x+"y" & x+"ies" & (x+"y's" | x+"ies'") };

-- beg
paradigm verbRep [x:Word, y:Char] =
  x+y { x+y & ((x+y+y+"ed" & x+y+"s") | x+y+y+"ing") };
```

**Result**

- noun rat
- noun cat
- ...
- nounY party
- nounY
- ...
- verbRep beg
- ..
Questions?

Next Tutorial:
Syntax and Translation systems

GF Tutorials Homepage

http://www.lama.univ-savoie.fr/~humayoun/tutorial-GF/