Grammatical Framework: A Hands-On Introduction

Aarne Ranta

CADE-23, Wrocław 1 August 2011

Preamble

Whom is this tutorial for

Interest in some of

- natural languages
- formal languages
- natural language interfaces
- translation

Assumed background: programming, some mathematics, some logic

Not assumed: linguistics

What you will learn

Build multilingual translation systems on the web

Build natural language interfaces and reversible converters

Explore the structure of any of 20 languages in the GF library

Get motivated to build libraries for other languages

Demo: the MOLTO Phrasebook

http://www.grammaticalframework.org/demos/phrasebook/

Traveller's phrases for 15 languages

High-quality translation via a **semantic interlingua**

Incremental parsing and **disambiguation**

Built on a declarative GF grammar + generic components

Available on the web and for Android phones (**Phrasedroid** on the Market)

The history of GF

1988: type-theoretical grammar = Montague grammar in type theory

1992: natural language interface to ALF proof system

1998: multilingual document authoring at Xerox Research, Grenoble

2010: MOLTO (= Multilingual On-Line Translation), EU-Strep

More on GF

CADE lecture on Thursday at 9:00

http://www.grammaticalframework.org

Second GF Summer School in Barcelona, 15-26 August

Book by AR, *Grammatical Framework: Programming with Multilingual Grammars*, CSLI, Stanford, 2011.

Schedule for today

14.00-15.30

- GF on the map of linguistics, computer science, and logic.
- Building a simple translation system and its web interface.
- Scaling up a translation system: problems and tools.
- Using the GF Resource Grammar Library.
- Specifying the translation system for the hands-on session.

16.00-17.30

- Hands-on session: porting translation to a new language.
- More advanced GF: grammars and reasoning.
- More advanced GF: computational grammars for the world.

GF on the map

Linguistics: a **grammar formalism**

- equivalent to PMCFG, polynomial parsing
- multilingual grammars related by interlingua

Computer science: a **compiler framework**

- formalizes the idea of abstract syntax + concrete syntax
- framework for multi-source multi-target compiler/decompilers
- a special-purpose functional programming language

Logic: a **logical framework**

- based on Martin-Löf type theory and ALF
- GF = LF + concrete syntax rules

Simple GF grammars

The basic modules of GF

Abstract syntax: **categories** and **functions**

```
abstract Cade = {
  cat
    Term ;
  fun
    var_x : Term ;
    Abs   : Term -> Term ;
  flags startcat = Term ;
}
```

Concrete syntax: **linearization types** and **linearizations**

```
concrete CadeSymb of Cade = {
  lincat
    Term = Str ;
  lin
    var_x = "x" ;
    Abs n = "|" ++ n ++ "|" ;
}
```

GF and context-free grammars

The above GF grammar,

```
abstract Cade = {
  cat
    Term ;
  fun
    var_x : Term ;
    Abs   : Term -> Term ;
}
concrete CadeSymb of Cade = {
  lincat
    Term = Str ;
  lin
    var_x = "x" ;
    Abs n = "|" ++ n ++ "|" ;
}
```

is in fact equivalent to a **labelled BNF grammar**,

```
var_x. Term ::= "x"
Abs.   Term ::= "|" x "|"
```

So why bother? Why make it so verbose?

Reason 1: multilingual grammars

One abstract + many concretes

```
concrete CadeEng of Cade = {
  lincat
    Term = Str ;
  lin
    var_x = "x" ;
    Abs n = "the absolute value of" ++ n ;
}

concrete CadeFre of Cade = {
  lincat
    Term = Str ;
  lin
    var_x = "x" ;
    Abs n = "la valeur absolue de" ++ n ;
}
```

Compilers as multilingual grammars

Source and target language related by abstract syntax

```
                                                          iconst_2
                                                          iload_0
2 * x + 1  <----->  plus (times 2 x) 1  <------>  imul
                                                          iconst_1
                                                          iadd
```

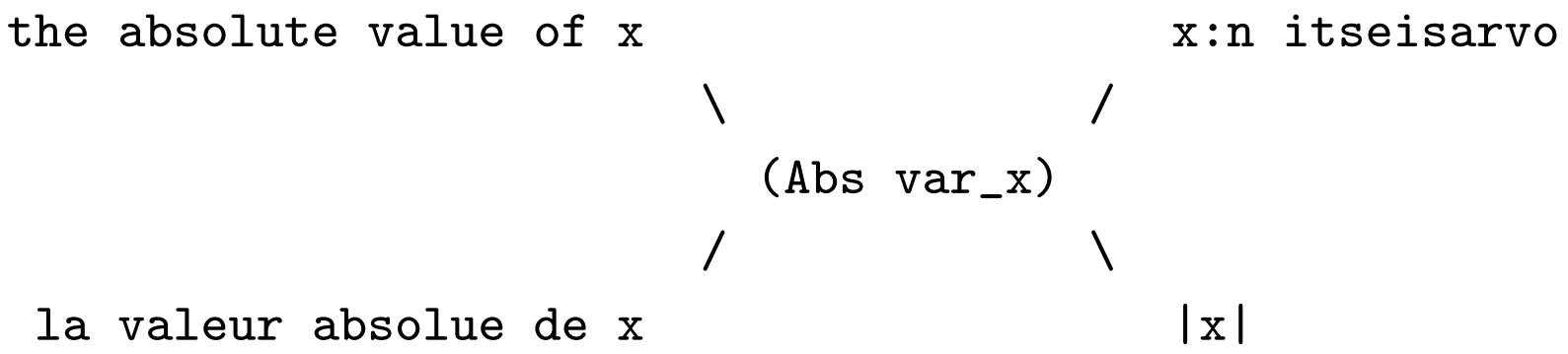
A GF grammar for Java and JVM

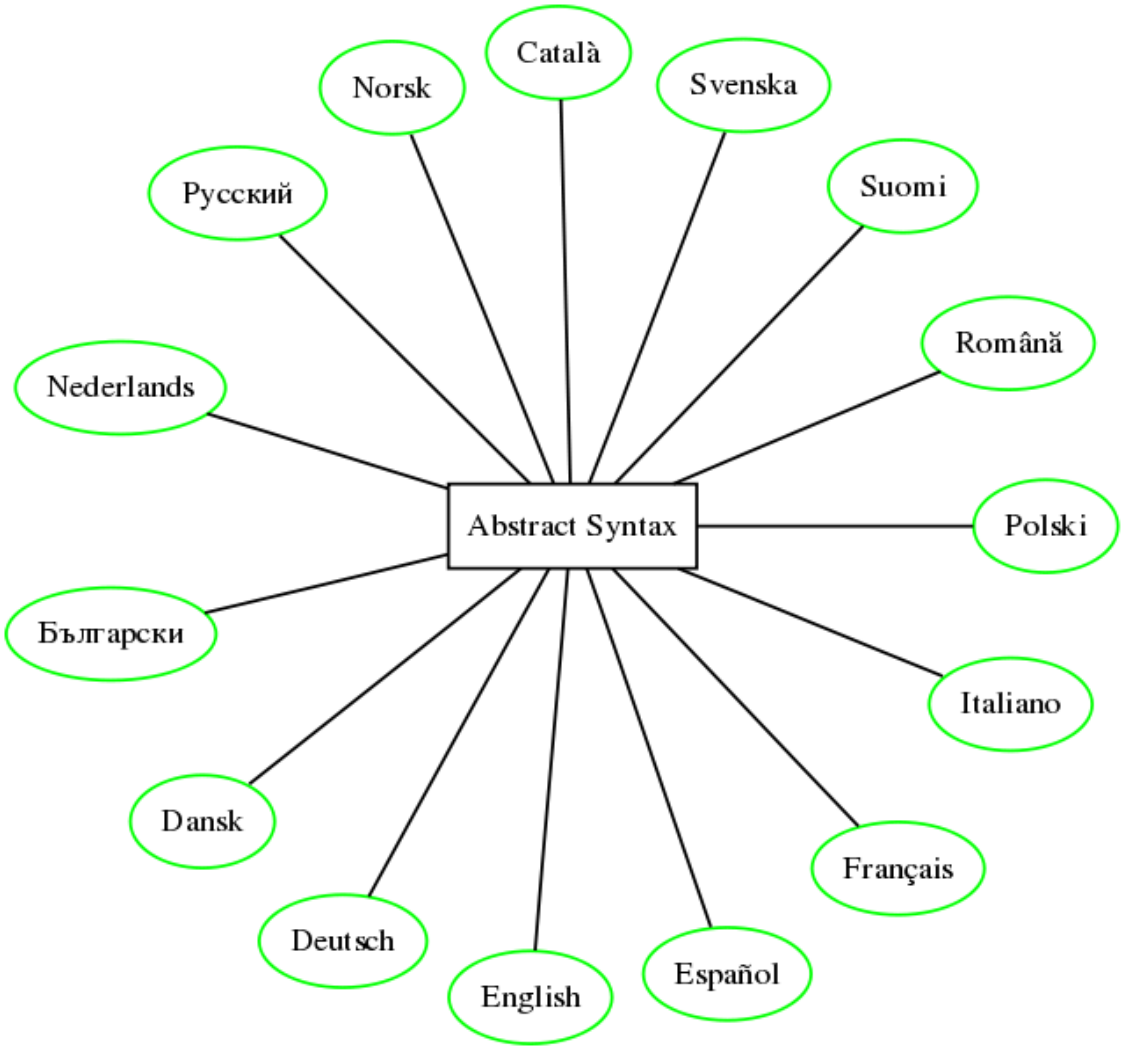
```
abstract Expr = {
  cat Exp ;
  fun plus : Exp -> Exp -> Exp ;
  fun times : Exp -> Exp -> Exp ;
  fun one, two : Exp ;
  }
```

```
concrete ExprJava of Expr = {
  lincat Exp = Str ;
  lin plus x y = x ++ "+" ++ y ;
  lin times x y = x ++ "*" ++ y ;
  lin one = "1" ;
  lin two = "2" ;
  }
```

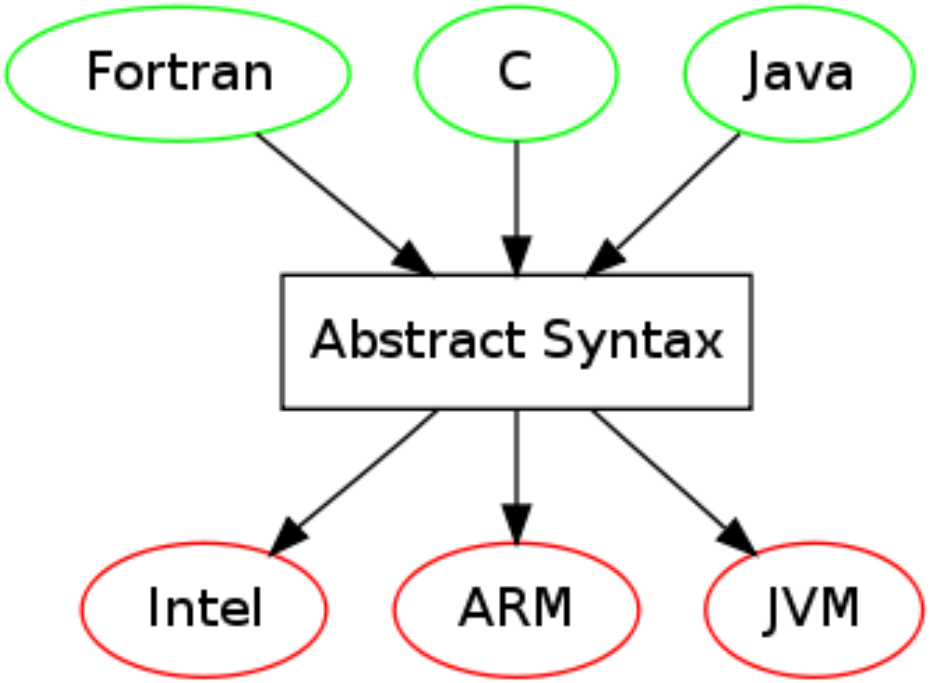
```
concrete ExprJVM of Expr= {
  lincat Expr = Str ;
  lin plus x y = x ++ y ++ "iadd" ;
  lin times x y = x ++ y ++ "imul" ;
  lin one = "iconst_1" ;
  lin two = "iconst_2" ;
  }
```

Compiling natural language




Català
Norsk
Svenska
Русский
Suomi
Română
Nederlands
Abstract Syntax
Polski
Български
Italiano
Dansk
Français
Deutsch
English
Español

Multi-source multi-target compiler-decompiler



Using GF grammars in the GF interpreter

1. Download and install GF, http://grammaticalframework.org/download/
2. Each module *foo* has to be in the file *foo*.gf
3. Start the interpreter with

   ```
   gf CadeEng.gf CadeSymb.gf
   ```

4. Commands and pipes:

   ```
   > parse -lang=Eng "the absolute value of x"
   > linearize -lang=Symb Abs var_x
   > parse -lang=Eng "the absolute value of x" | linearize -lang=Symb
   > generate_random | linearize
   ```

Using GF grammars in web applications

1. Compile the grammars into PGF = Portable Grammar Format

   ```
   $ gf -make CadeEng.gf CadeSymb.gf
   wrote Cade.pgf
   ```

2. Start the PGF server

   ```
   $ pgf-http
   Starting HTTP server, open http://localhost:41296/ in your web browser
   Options {documentRoot = "/home/aarne/.cabal/share/gf-server-1.0/www",
   ```

3. Copy `Cade.pgf` to the grammar directory under `documentRoot`

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

4. In your web browser, open `http://localhost:41296/`

Building GF grammars in the cloud

This is still experimental, but needs no software to be installed!

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

Solving linguistic problems

Reason 2: scaling up the grammar

Let's extend the grammar a bit:

```
abstract Cade = {
  cat
    Proposition ; Term ;
  fun
    var_x : Term ;
    Abs   : Term -> Term ;
    Positive : Term -> Proposition ;
  flags startcat = Proposition ;
}
concrete CadeEng of Cade = {
  lincat
    Term, Proposition = Str ;
  lin
    var_x = "x" ;
    Abs n = "the absolute value of" ++ x ;
    Positive n = n ++ "is positive" ;
}
```

The problem of agreement

```
concrete CadeFre of Cade = {
  lincat
    Term, Proposition = Str ;
  lin
    var_x = "x" ;
    Abs n = "la valeur absolue de" ++ x ;
    Positive n = n ++ "est positif" ;
}
```

But in French, adjectives have **gender agreement**

- masculine: *x est positif*
- feminine: *la valeur absolue de x est positive*

How can we use the same abstract syntax?

Caution: some linguistic torture

The next few slides contain linguistic details. The serve a triple purpose

- show how GF deals with linguistic problems
- maybe, make you interested in contributing to their solution
- but more probably, make you motivated to use the libraries

So don't panic - you can program in GF with little attention on or knowledge of low-level linguistic details.

Solution: parametric variation

We introduce a **parameter type** of genders and change the linearization type of numbers into a **record** with a gender:

```
concrete CadeFre of Cade = {
  param
    Gen = Masc | Fem ;
  lincat
    Proposition = Str ; Term = {s : Str ; g : Gen} ;
  lin
    var_x = {s = "x" ; g = Masc} ;
    Abs n = {
      s = "la valeur absolue de" ++ x.s ;
      g = Fem
      } ;
    Positive n = n.s ++ "est" ++ case n.g of {
      Masc => "positif" ;
      Fem  => "positive"
      } ;
}
```

Inflection tables

Context-free grammar has **strings** and **concatenation**.

GF also has **parameters** and **records**.

We need one more thing: **tables**, which are functions on parameters:

```
table {                    -- a table
  Masc => "positif" ;
  Fem  => "positive"
  }
   : Gen => Str            -- the type of the table
```

Adjectives

We want to use **adjectives** for both

- **predication**: *x is positive*
- **modification**: *a positive integer*

We reformulate the abstract syntax once more

```
cat
  Proposition ; Term ; Adjective ; Noun ;
fun
  Pred     : Term -> Adjective -> Proposition ;
  Mod      : Adjective -> Noun -> Noun ;
  Positive : Adjective ;
  Integer  : Noun ;
```

Combinations with adjectives

parameters	**English**	**French**
Sg, Masc	*positive integer*	*entier positif*
Sg, Fem	*positive value*	*valeur positive*
Pl, Masc	*positive integers*	*entiers positifs*
Pl, Fem	*positive values*	*valeurs positives*

Moreover, French adjectives have a position:

- **prefix**: *bon vin* "good wine"
- **postfix**: *vin intéressant* "interesting wine"

The syntax of French adjectives

```
param
  Num = Sg | Pl ;
  Gen = Masc | Fem ;
  Pos = Pref | Postf ;
lincat
  Adjective = {s : Gen => Num => Str ; p : Pos} ;
  Noun      = {s : Num => Str ; g : Gen} ;
lin
  Mod adj noun = {
    s = table {n =>
      let
        adjs  = adj.s ! noun.g ! n ;
        nouns = noun.s ! n
      in case adj.p of {
        Pref  => adjs ++ nouns ;
        Postf => nouns ++ adjs
        }
      } ;
    g = noun.g
    } ;
```

The morphology of French adjectives

To linearize an adjective, we *could write*

```
lin Positive = {
  s = table {
    Masc => table {Sg => "positif"  ; Pl => "positifs"} ;
    Fem  => table {Sg => "positive" ; Pl => "positives"}
    } ;
  p = Postf
  } ;
```

But this is tedious to repeat for all adjectives.

We eliminate the repetition by some **functional programming**.

An auxiliary operation for French adjectives

An `oper` definition defines a reusable concrete-syntax function:

```
oper mkAdjective :
  Str -> Str -> Str -> Str -> Pos -> Adjective =
  \msg,mpl,fsg,fpl,p -> {
    s = table {
      Masc => table {Sg => msg ; Pl => mpl} ;
      Fem  => table {Sg => fsg ; Pl => fpl}
      } ;
    p = p
    } ;
```

The notation `\msg,mpl,fsg,fpl,p -> ...` is **lambda abstraction**.

Now we can write

```
lin Positive = mkAdjective "positif" "positifs"
                           "positive" "positives" Postf ;
```

A smart paradigm for French adjectives

We use **pattern matching** to define the regular variations of French adjectives, and also assume most adjectives are postfix.

```
oper regAdjective : Str -> Adjective =
  \msg -> case msg of {
    v  + "if" => mkAdjective msg (msg + "s") (v  + "ive") (v  + "ives") Postf ;
    ch + "er" => mkAdjective msg (msg + "s") (ch + "ère") (ch + "ères") Postf ;
    _         => mkAdjective msg (msg + "s") (msg + "e")  (msg + "es")  Postf
    } ;
```

Now we can write really concisely

```
lin
  Positive = regAdjective "positif" ;
  Prime    = regAdjective "premier" ;
  Even     = regAdjective "pair" ;
```

The GF Resource Grammar Library

End of linguistic torture

Software libraries are a key to efficient programming. They

- encapsulate expert knowledge
- hide low-level details
- enable productive programming without expert knowledge

Grammars are an obvious object for such libraries. They should give

- morphological inflection
- syntactic combinations (word order, agreement)

The grammar library API

Types

Cl	clause	*John loves Mary*
NP	noun phrase	*John*
V2	two-place verb	*love*
A	adjective	*old*
CN	common noun	*man*

Syntax functions (overloaded, `mk`C for value of type C)

mkCl	NP -> V2 -> NP -> Cl	*John loves Mary*
mkCl	NP -> A -> Cl	*John is old*
mkCN	A -> CN -> CN	*old man*

Morphology functions (overloaded, `mk`C for value of type C)

mkA	Str -> A	*cher, chers, chère, chères*
mkA	Str -> Str -> A	*frais, frais, fraîche, fraîches*

Using the library in concrete syntax

```
concrete CadeFre of Cade = open SyntaxFre, ParadigmsFre in {
  lincat
    Proposition = Cl ;
    Adjective = A ;
    Term = NP ;
    Noun = CN ;
  lin
    Pred term adj = mkCl term adj ;
    Mod adj noun = mkCN adj noun ;
    Positive = mkA "positif" ;
}
```

The GF Resource Grammar Library

Complete morphology + comprehensive syntax

20 languages: Afrikaans, Bulgarian, Catalan, Danish, Dutch, English, Finnish, French, German, Italian, Nepali, Norwegian, Persian, Polish, Punjabi, Romanian, Russian, Spanish, Swedish, Urdu

Under construction, but available: Amharic, Arabic, Hindi, Latin, Latvian, Thai, Turkish.

Effort: 3-5 kLOC of GF code, 3-9 person months per language.

Exploring the library

Browse the **synopsis**,

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

Try inflection, parsing, generation, and translation in the GF shell

Hands-on: a grammar for logic and mathematics

The grammar

Available in

http://www.grammaticalframework.org/gf-tutorial-cade-2011/code/

- Cade.gf
- CadeEng.gf
- CadeFre.gf

Either copy this code to your computer, or follow the session on the screen.

The task

1. Go through the code
2. Make some experiments in the shell
3. Port the code to a new language available in the Resource Grammar Library

Later work

Functors: share concrete syntax code across language (GF book, Chapter 5)

Logic in natural language: improving the style (CADE lecture on Thursday)

More advanced GF: grammars and reasoning

Type theory and abstract syntax

Dependent types

Higher-order abstract syntax

Semantic definitions

Transfer functions

See the GF book, Chapters 6 and 8, and also CADE lecture on Thursday

More advanced GF: computational grammars for the world

Extending the resource grammar

There are 6,000-26 languages left!

Are they all possible?

Are some languages more complex than others?

See the GF book, Chapters 9 and 10

Extending the coverage of grammars

Building large-scale lexica

Using statistical language processing

- to recover from errors (**robustness**)
- to generate grammars semi-automatically (**bootstrapping**)

See http://www.molto-project.eu