Grammatical Framework

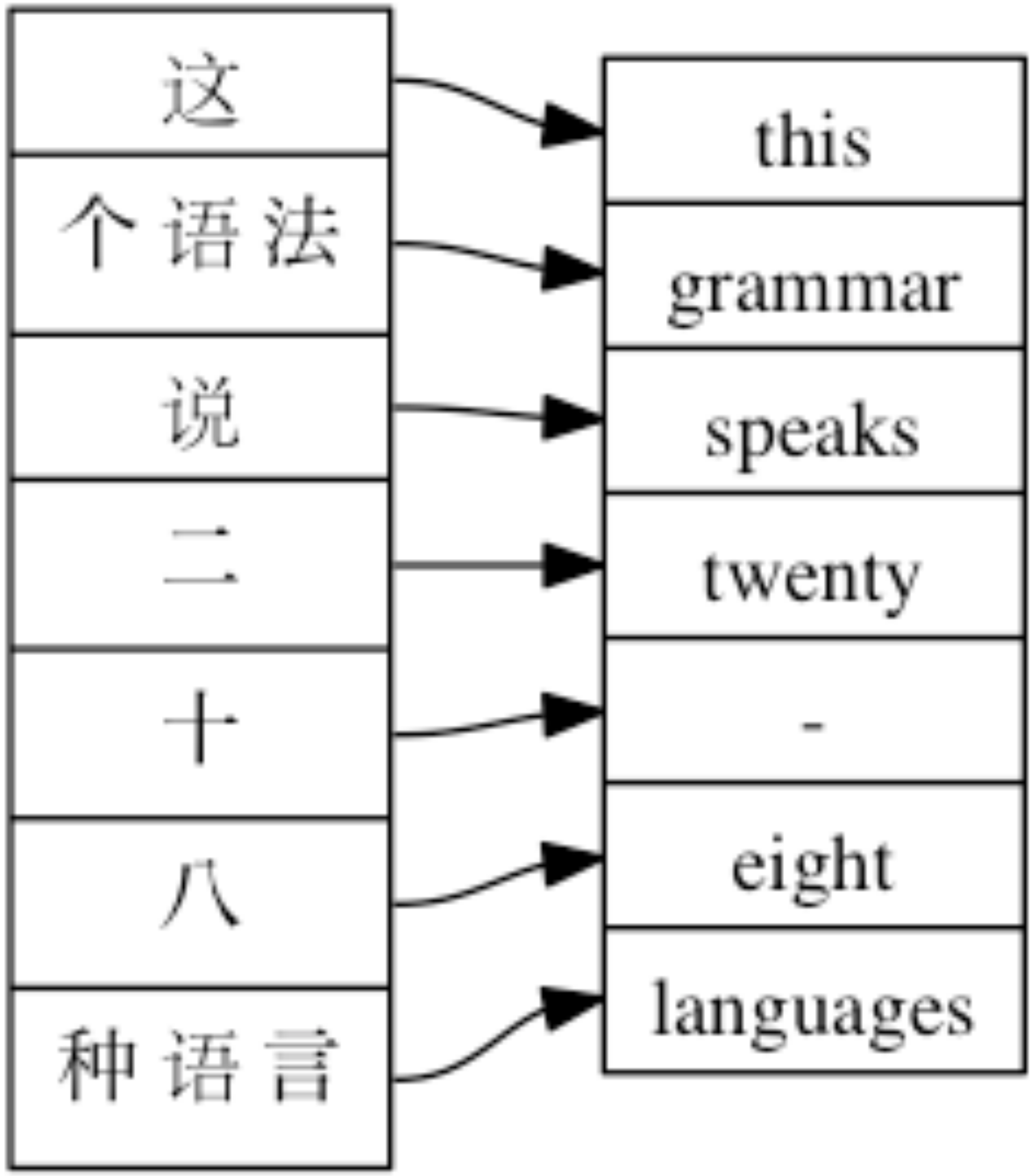
Programming with Multilingual Grammars

With a Special Focus on Chinese

@

Sun Yat-Sen University, Guangzhou
23-27 September 2013

Aarne Ranta
http://www.cse.chalmers.se/~aarne/


这
个 语 法
说
二
十
八
种 语 言
this
grammar
speaks
twenty
-
eight
languages

The goals of this course

- writing multilingual grammars
- using them for
    - translation
    - human-computer interaction
- running them
    - in web applications
    - on mobile phones (Android, iOS)

A specific goal

- Improve the grammar for Chinese
    - ★ Enable English-Chinese translation
    - ★ Build language learning applications
- Currently 6,000 words in Chinese
    - ★ Can we reach 20,000 words?

Course assignment

- *Either*
  - add 500 new Chinese words
- *Or*
  - build a GF application

Schedule

Monday	lecture 13-15 (19:00 - 21:35)
Tuesday	lab 1-2 (8:00- 9:40), lecture 7-8 (13:30 - 15:10)
Wednesday	lab1-2 (8:00- 9:40)
Thursday	lecture 1-2 (8:00- 9:40), lab 13-15 (19:00 - 21:35)
Friday	lab 10-11 (16:15 - 17:55)

GF = Grammatical Framework

- a programming language for grammars
- so are YACC and Bison
    - but GF is also for natural languages
    - **compiling natural language**

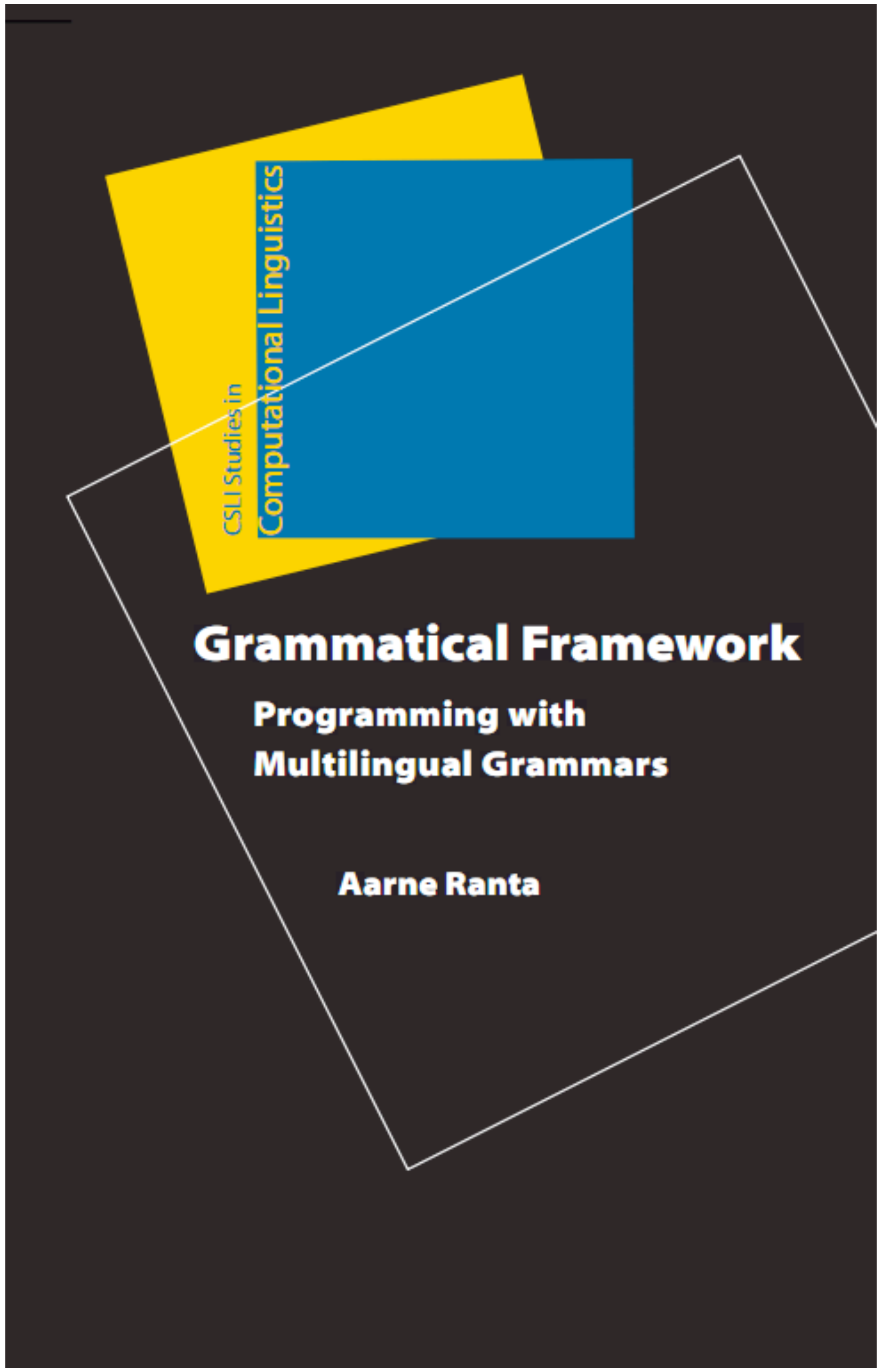
Focus of GF

- Like compilers, unlike e.g. Google translate
    - precision
    - high quality
    - grammatical correctness
    - meaning preservation

History of GF

- Started at Xerox Research in 1998
- now open-source, 100+ developers
- European projects: WebALT, MOLTO, Monnet
- Companies: Ontotext (Bulgaria), Be Informed (Holland), Lingsoft (Finland), Galois (USA)
- 100+ publications, 10+ PhD theses

The GF Book



- CSLI, Stanford, 2011
- Soon available in Chinese translation by Prof. Yan Tian of Jiao Tong University, Shanghai

语法框架

为多种自然语言语法编程

Grammatical Framework

Programming with Multilingual Grammars

[瑞典] Aarne Ranta 著

田艳译

上海交通大学出版社

Shanghai Jiao Tong University Press Co., Limited

上海 ·SHANGHAI

内容简介

语法框架是一种计算机编程语言，专为编写自然语言的语法而设计，它有能力并行处理多种自然语言。本书全面介绍和展示了如何用语法框架编写自然语言的语法，以及如何在一些实用系统，比如常用语系统，口语对话系统以及自然语言界面系统中加以应用。书中的例子和练习涉及多种自然语言。读者可以从中知晓如何从计算语言学的视角看待自己的母语。

阅读本书不需要语言学的预备知识，因此，特别适合计算机科学家和程序员。同样，语言学家也会对本书产生兴趣，因为本书从计算机程序语言的视角启示了处理多种自然语言语法的新途径。

How to translate these?

I ate bread with butter

I ate bread with you

What I get from Google translate

I ate bread with butter.

我吃面包，黄油。

I ate bread with you.

我吃了面包与你同在。

What I get from GF

I ate bread with butter

我吃了和黄油一起的面包

I ate bread with you

我在和你一起吃了面包

Actually I could also get

I ate bread with butter

我在和黄油一起吃了面包

I ate bread with you

我吃了和你一起的面包

Structural ambiguity

I ate (bread (with butter))

I ((ate bread) (with friends))

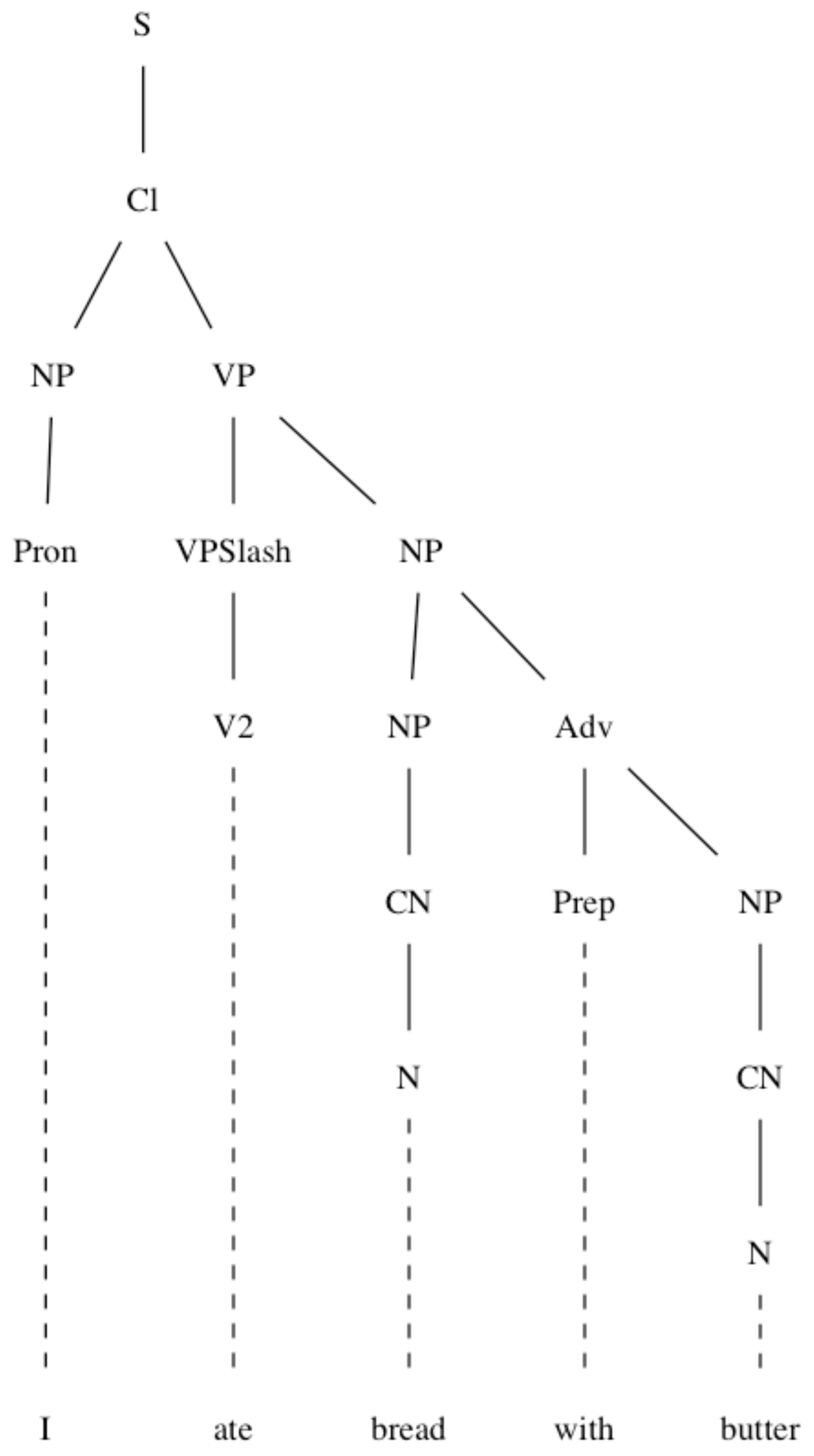
Familiar from mathematics

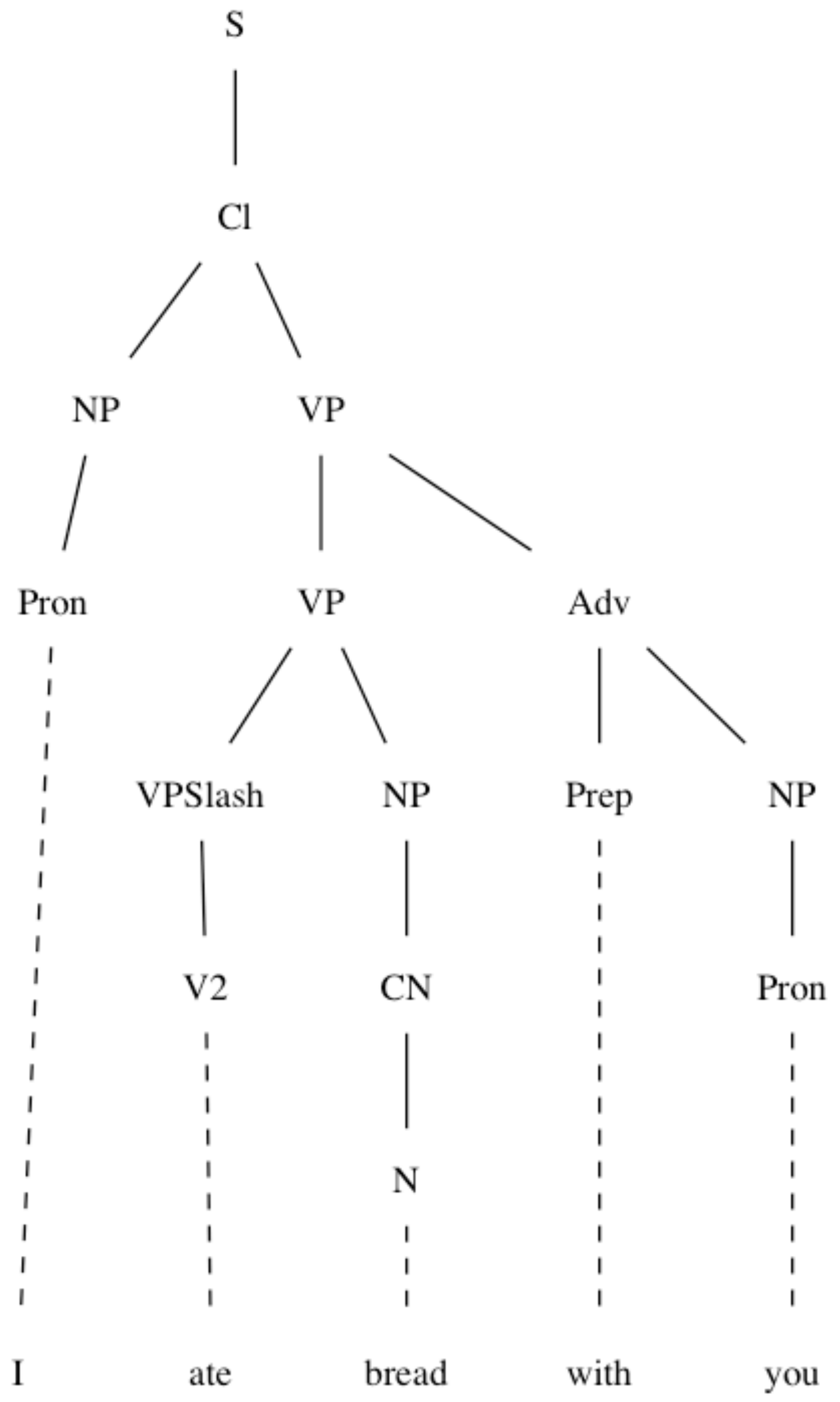
two plus three times four

2 + 3 * 4 = 14   // = 2 + (3 * 4)

(2 + 3) * 4 = 20

Different syntax trees

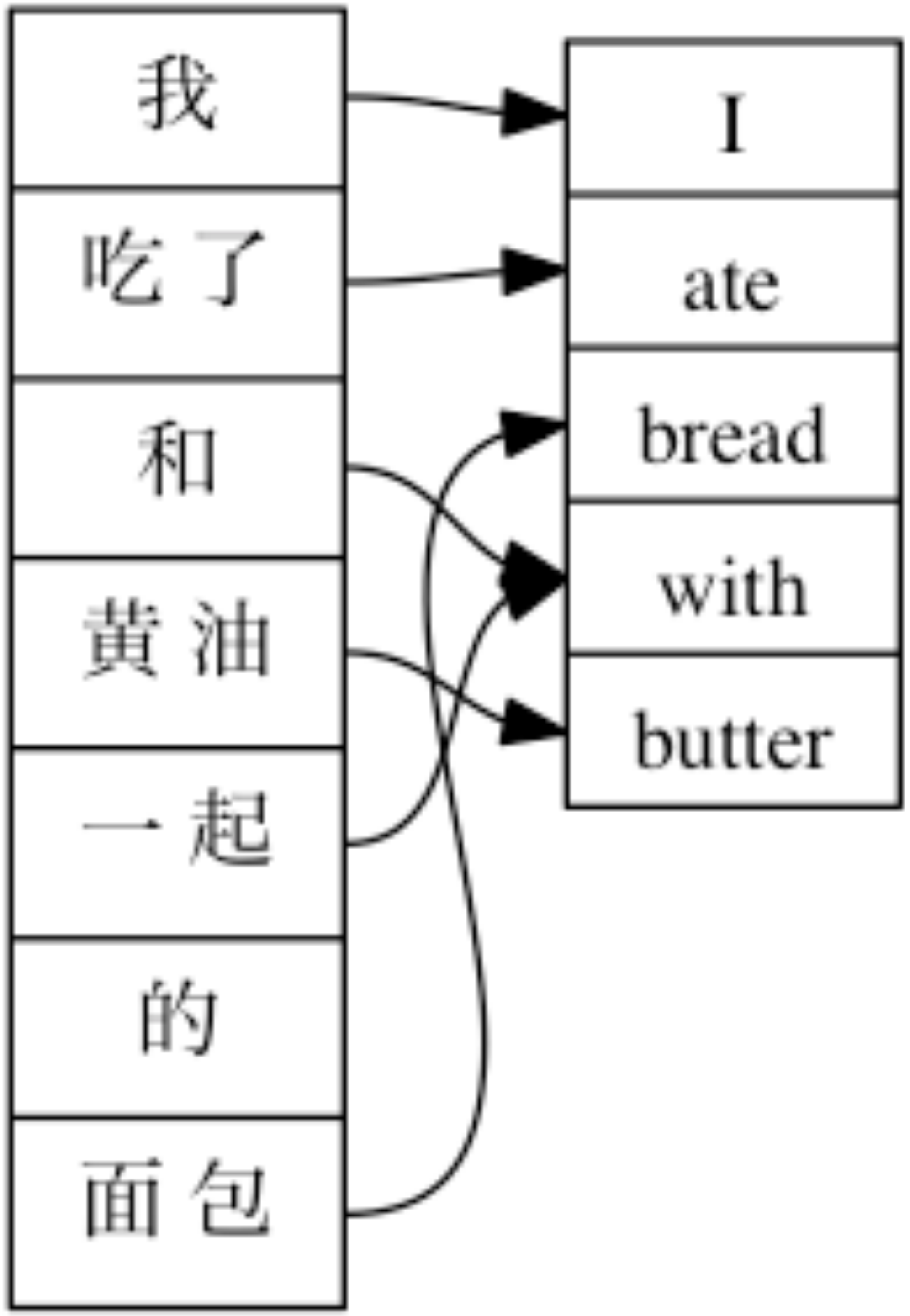


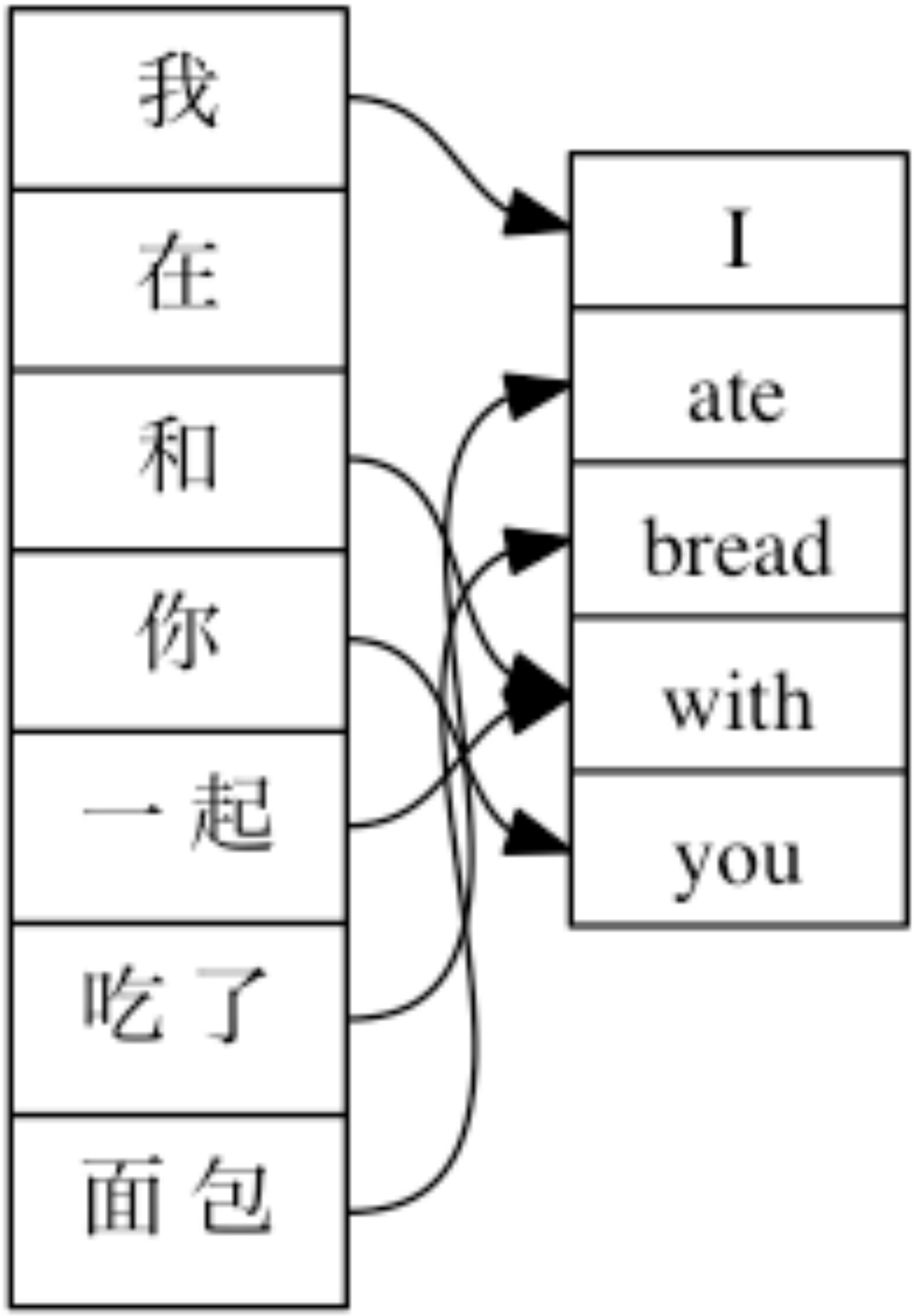


Order vs. structure

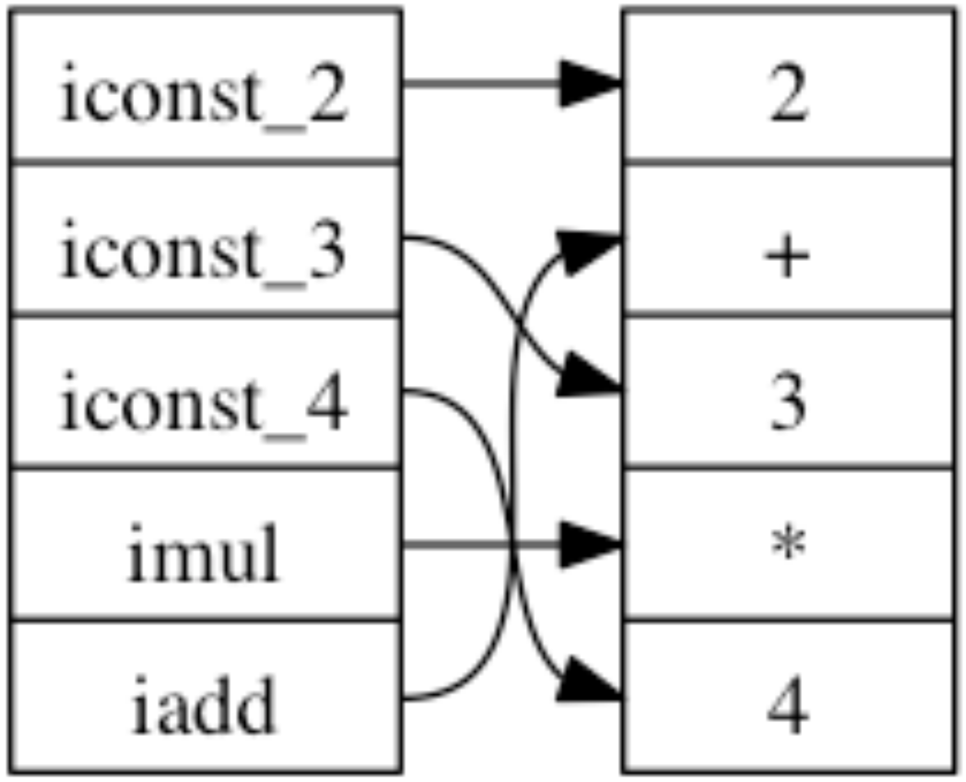
- English:
  - Subject Verb (Object Adverb) -> Subject Verb Object Adverb
  - Subject (Verb Object Adverb) -> Subject Verb Object Adverb
- Chinese:
  - Subject Verb (Object Adverb) -> Subject Verb Adverb Object
  - Subject (Verb Object Adverb) -> Subject Adverb Verb Object

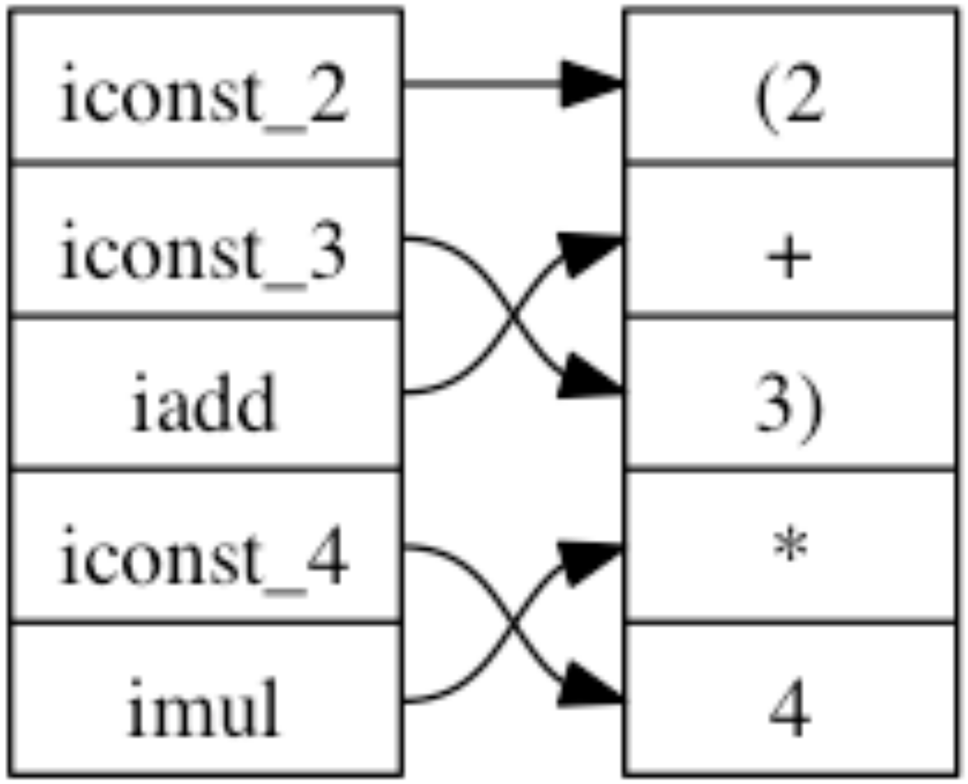
Word alignments





Compilers: aligning Java and JVM





The principle of compilers (and GF)

Translate via **abstract syntax**

- shared, underlying **tree structure**

Languages differ in **concrete syntax**

- conversion of trees into **strings**

An example of GF

Abstract syntax: tree construction function for multiplication expressions

```
fun EMul : Exp -> Exp -> Exp
```

Concrete syntax: linearization to Java

```
lin EMul x y = x ++ "*" ++ y
```

Concrete syntax: linearization to JVM

```
lin EMul x y = x ++ y ++ "imul"
```

Natural language in GF

Categories: types of expressions

S	sentence	e.g.	*I eat rice*	我吃饭
NP	noun phrase	e.g.	*I*	我
VP	verb phrase	e.g.	*eat rice*	吃饭
V	verb	e.g.	*eat*	吃

Abstract syntax functions

Predication, complementation, and a couple of words:

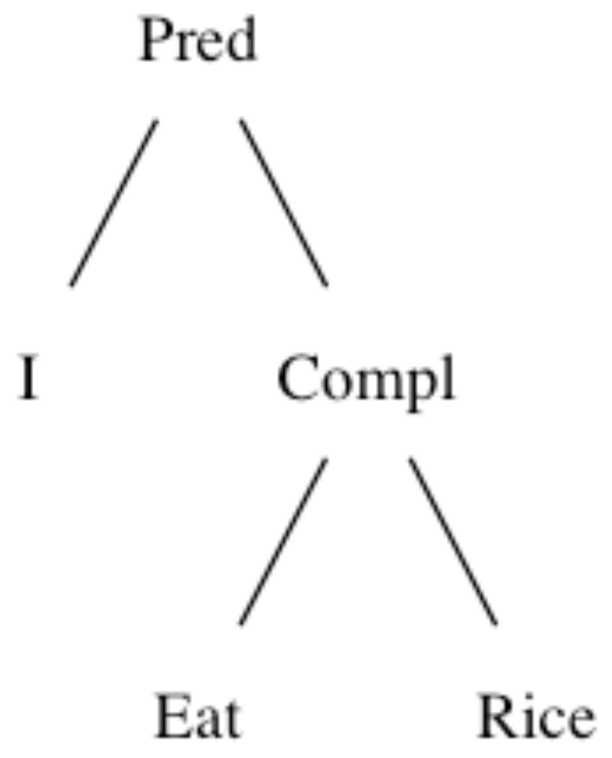
```
fun
  Pred  : NP -> VP  -> S
  Compl : V  -> NP  -> VP
  I     : NP
  Eat   : V
  Rice  : NP
```

Building an abstract syntax tree

We apply functions Compl and Pred:

```
            Compl Eat Rice : VP
Pred I (Compl Eat Rice): S
```

Graphical version of the syntax tree:



Concrete syntax: linearization to English

```
lin
  Pred  np vp  = np ++ vp
  Compl v2 np  = v2 ++ np
  I    = "I"
  Rice = "rice"
  Eat  = "eat"
```

The symbol ++ means **concatenation**.

Now we can linearize:

```
Pred I (Compl Eat Rice)

= "I" ++ ("eat" ++ "rice")

= "I eat rice"
```

Concrete syntax: linearization to Chinese

```
lin
  Pred  np vp  = np  ++ vp
  Compl v2 np  = v2  ++ np
  I = "我"
  Rice = "饭"
  Eat = "吃"
```

Now we can linearize:

```
  Pred I (Compl Eat Rice)

  = "我" ++ ("吃" ++ "饭")

  = "我 吃 饭"
```

Notice: spaces needed in English, should be removed in Chinese. We will return to this.

Using GF grammars

We want to use grammars

- in the GF shell
- in a web application

To do so, we

- write the grammars in .gf files
- compile the grammars

Abstract syntax file: `Basic.gf`

```
abstract Basic = {                -- module header

flags startcat = S ;              -- setting start category

cat                               -- categories

  S ;                             -- separated by semicolons

  NP ;

  VP ;

  V ;

fun                               -- functions

  Pred  : NP -> VP  -> S ;

  Compl : V  -> NP  -> VP ;

  I : NP ;

  Rice : NP ;

  Eat : V ;                       -- two dashes start a comment

}
```

Concrete syntax file: `BasicEng.gf`

```
concrete BasicEng of Basic = {  -- module header

lincat                          -- linearization types of cat's

  S = Str ;

  NP = Str ;

  VP = Str ;

  V = Str ;

lin                             -- linearization rules of fun's

  Pred  np vp  = np ++ vp ;

  Compl v  np  = v  ++ np ;

  I = "I" ;

  Rice = "rice" ;

  Eat = "eat" ;

}
```

Concrete syntax: `BasicChi.gf`

```
concrete BasicChi of Basic = {

flags coding = utf8 ;           -- set non-latin character encoding

lincat                          -- the rest is exactly as in English...

  S = Str ;

  NP = Str ;

  VP = Str ;

  V  = Str ;

lin

  Pred  np vp  = np ++ vp ;

  Compl v  np  = v  ++ np ;

  I = "我" ;                    -- ...except the words

  Rice = "饭" ;

  Eat = "吃" ;

}
```

Using the GF shell

1. Go to GF Download page to install GF:

   `http://www.grammaticalframework.org/download`

2. Install GF (Linux, Mac OS, Windows)

3. Open the GF shell in your OS shell:

   ```
   gf
   ```

4. Import the files you want

   ```
   import BasicChi.gf BasicEng.gf
   ```

5. Parse, linearize, generate, translate,...

Example GF shell session

- Import grammars

```
> import BasicChi.gf BasicEng.gf
linking ... OK
Languages: BasicChi BasicEng
```

- Parse English string into tree

```
Basic> parse -lang=Eng "I eat rice"

Pred I (Compl Eat Rice)
```

- Linearize tree into Chinese

```
Basic> linearize -lang=Chi Pred I (Compl Eat Rice)

我 吃 饭
```

- Parse Chinese, linearize to English

```
Basic> parse -lang=Chi "我 吃 饭" | linearize -lang=Eng
I eat rice
```

- Generate random tree, linearize to both languages

```
Basic> generate_random | linearize

饭 吃 我
rice eats I
```

About GF shell commands

- `parse` maps strings to trees
- `linearize` maps trees to strings
- `-lang=XXX` sets language (default: all)
- **pipe** `|` sends output to next command
  - translate: `parse | linearize`
- `generate_random` builds random trees
  - test: `generate_random | linearize`

Testing a grammar

We already found an error in English:

```
Basic> generate_random | linearize

饭 吃 我

rice eats I
```

This should be

```
rice eats me
```

We will return to this.

Web applications

1. Compile to pgf = Portable Grammar Format

```
aarne$ gf -make BasicChi.gf BasicEng.gf
linking ... OK
Writing Basic.pgf...
```

2. Start GF in server mode

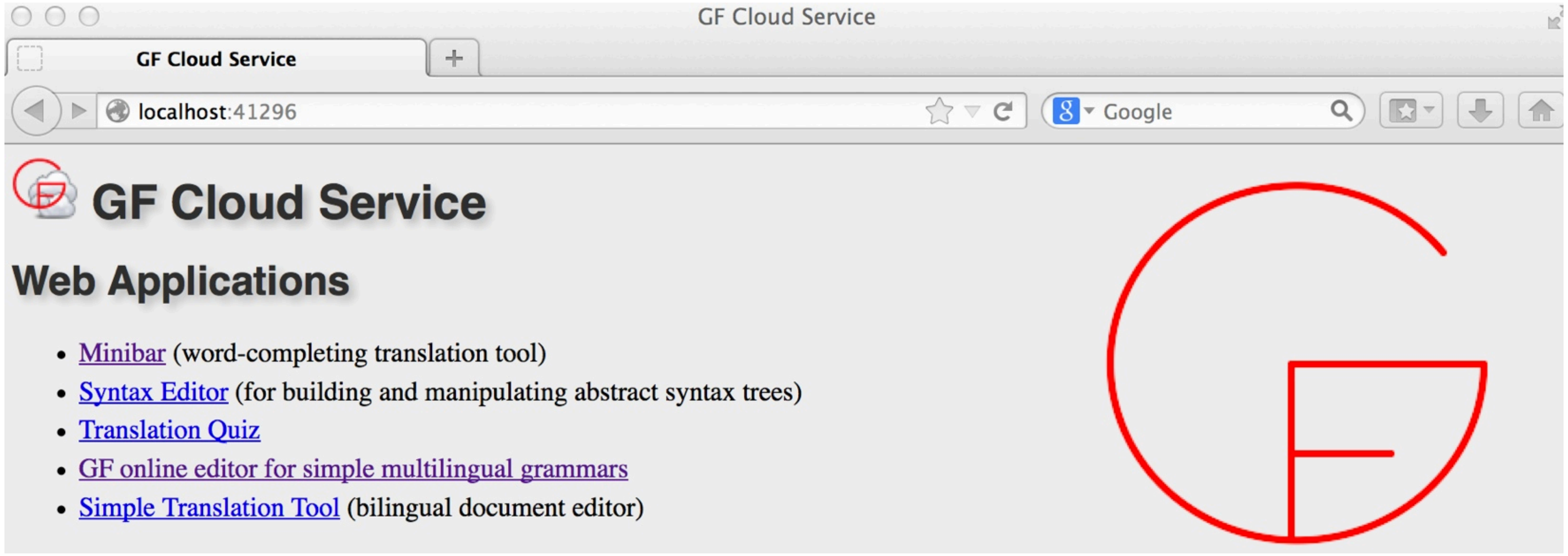
```
aarne$ gf -server

Document root = /Users/aarne/Library/Haskell/ghc-7.4.2/lib/gf-3.5/
share/www

Starting HTTP server, open http://localhost:41296/ in your web browser.
```

3. Copy `Basic.pgf` to <document root>`/grammars/`

4. Open the link in Firefox - you get to GF Cloud

5. Select Minibar -> Grammar:Basic.pgf

There are many cloud services available for GF grammars



GF Cloud Service

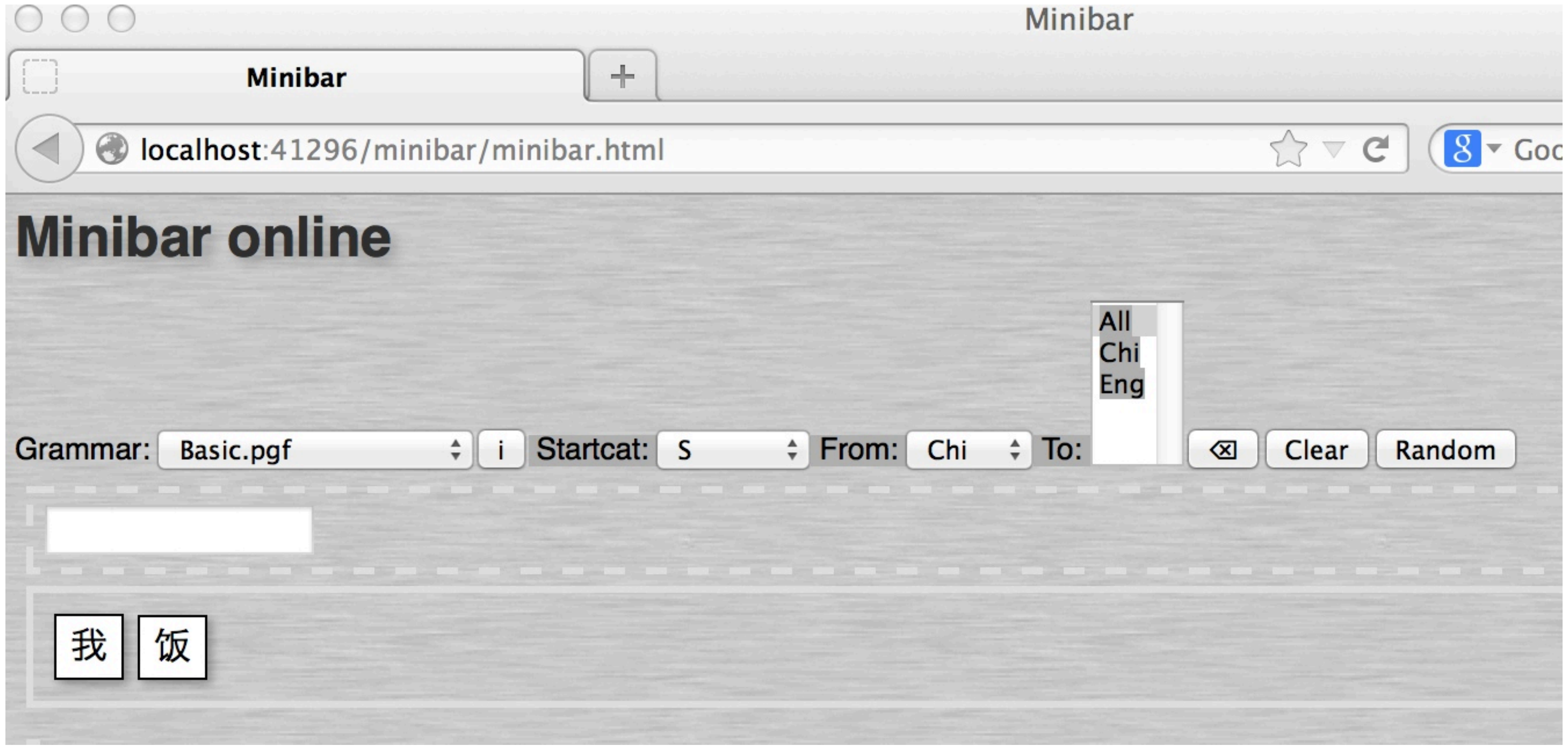
localhost:41296

GF Cloud Service

Web Applications

- Minibar (word-completing translation tool)
- Syntax Editor (for building and manipulating abstract syntax trees)
- Translation Quiz
- GF online editor for simple multilingual grammars
- Simple Translation Tool (bilingual document editor)

The Minibar is a "fridge magnet" based editor



Minibar

localhost:41296/minibar/minibar.html

Minibar online

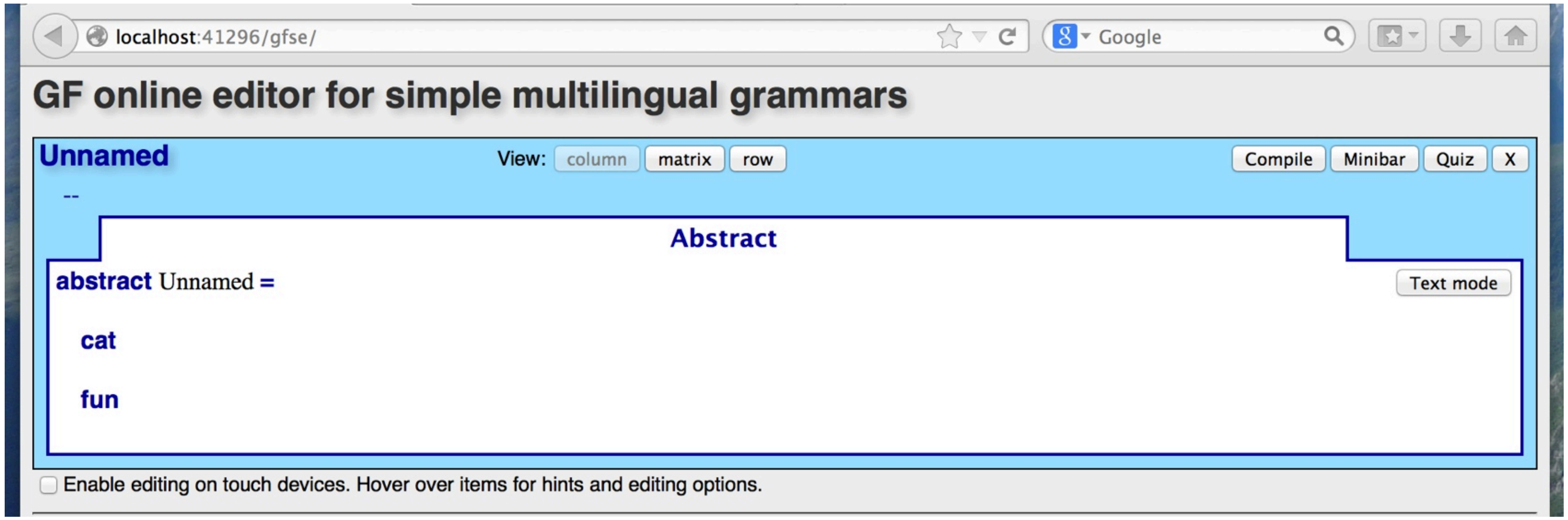
Grammar: Basic.pgf Startcat: S From: Chi To: All Chi Eng

The easiest way to use GF

Cloud service on GF server:

http://cloud.grammaticalframework.org/

Select GF online editor for simple multilingual grammars to work in GF without installing anything!


localhost:41296/gfse/

GF online editor for simple multilingual grammars

Unnamed

--

View: column matrix row

Compile Minibar Quiz X

Abstract

Text mode

```
abstract Unnamed =

  cat

  fun
```

Enable editing on touch devices. Hover over items for hints and editing options.

★ At this point, we will do some cloud work on translating, generating, grammar testing, and language training.

The power of GF

- shared abstract syntax
- different words
- different word orders
- translate A to B = parse A | linearize B
- works for any number of languages

The problem with English

- Western languages have **inflection**: words have many forms
  - Chi 吃 ; Eng *eat, eats, ate, eaten, eating*
- Inflection is used in **agreement**: the form chosen depends on other words
  - Chi 我 吃, 他吃 ; Eng *I* <u>*eat*</u>*, he* <u>*eats*</u>
- How can we do this in GF?

Parameters and tables

We linearized V (and VP) as strings:

```
lincat V = Str
```

We can change this to a **table**, a.k.a. a **finite function**:

```
lincat V = Number => Str
```

It depends on a **parameter**:

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

Now every verb has two forms:

```
lin Eat =

table {Sg => "eats"; Pl => "eat"}
```

Records

The form of a verb is determined by the subject NP.

Therefore, an NP has an **inherent** number. It is stored in a **record**:

```
lincat NP = {s : Str ;     n : Number}
lin      I = {s = "I" ;     n = Pl}
lin   Rice = {s = "rice" ; n = Sg}
```

Sg = Singular, Pl = Plural.

Agreement

The inherent number of NP is passed to the VP, to select the proper form:

```
lin Pred np vp = np.s ++ vp ! np.n
```

Read this:

The string of the NP followed by the form of the VP selected for the number of the NP.

The complete code

```
param
  Number = Sg | Pl ;
lincat
  S = Str ;
  NP = {s : Str ; n : Number} ;
  VP = Number => Str ;
  V = Number => Str ;
lin
  Pred  np vp  = np.s ++ vp ! np.n ;
  Compl v  np  = table {n => v ! n ++ np.s} ;
  I = {s = "I" ; n = Pl} ;
  Rice = {s = "rice" ; n = Sg} ;
  Eat = table {Sg => "eats" ; Pl => "eat"} ;
```

One problem solved

The form of the verb gets right now:

```
  Pred Rice (Compl Eat I)

-> rice ++

   (table {Sg => "eats" ; Pl => "eat"} ! Sg ++ "I")

-> "rice eats I"
```

But how to get *rice eats* <u>*me*</u> ?

Just another parameter

Noun phrase forms depend on **case**:

```
param Case = Nom | Acc

lincat NP = {s : Case => Str ; n : Number}

lin I = {s = table {Nom => "I" ; Acc => "me"} ; n = Pl}

lin Rice = {s = table {_ => "rice"} ; n = Sg}
```

Nom = Nominative, Acc = Accusative.

`_ =>` means "for all values of the parameter".

Putting the case parameter in place

```
concrete BasicEng of Basic = {


param
  Number = Sg | Pl ;
  Case = Nom | Acc ;
lincat
  S = Str ;
  NP = {s : Case => Str ; n : Number} ;
  VP = Number => Str ;
  V = Number => Str ;
lin
  Pred  np vp  = np.s ! Nom ++ vp ! np.n ;
  Compl v  np  = \\n => v ! n ++ np.s ! Acc ;


  I = {s = table {Nom => "I" ; Acc => "me"} ; n = Pl} ;
  Rice = {s = table {_ => "rice"} ; n = Sg} ;
  Eat = table {Sg => "eats" ; Pl => "eat"} ;
}
```

This is the final, correct version of BasicEng.gf.

Adverbs

Adverbs: modifiers of verbs (and other expressions as well:

with butter, in the house, tomorrow

In Chinese, adverbs come **before** the modified expression.

In English, they come after.

Chinese also has to add some extra words.

Adding adverbs to Basic

Abstract syntax

```
AdvVP : VP -> Adv -> VP
AdvNP : NP -> Adv -> NP
```

Concrete syntax, English

```
AdvVP vp adv = table {n => vp ! n ++ adv}
AdvNP np adv =
  {s = table {c => np.s ! c ++ adv} ; n = np.n}
```

Concrete syntax, Chinese

```
AdvVP vp adv = "在" ++ adv ++ vp

AdvNP np adv = adv ++ "的" ++ np
```

Forming adverbs

The most productive way: preposition + NP

```
With    : NP -> Adv
Without : NP -> Adv
```

English:

```
With np = "with" ++ np.s ! Acc
Without np = "without" ++ np.s ! Acc
```

Chinese:

```
With np = "和" ++ np ++ "一 起"
Without np = "没 有" ++ np
```

NB. there are many other translations of *with*.

Trying out word order with adverbs

```
Basic> p -lang=Eng "I eat bread with butter" | l -treebank
1. Basic: Pred I (AdvVP (Compl Eat Bread) (With Butter))
BasicChi: 我 在 和 黄 油 一 起 吃 面 包

BasicEng: I eat bread with butter
2. Basic: Pred I (Compl Eat (AdvNP Bread (With Butter)))
BasicChi: 我 吃 和 黄 油 一 起 的 面 包

BasicEng: I eat bread with butter


Basic> p -lang=Eng "I eat bread with you" | l -treebank
1. Basic: Pred I (AdvVP (Compl Eat Bread) (With You))
BasicChi: 我 在 和 你 一 起 吃 面 包

BasicEng: I eat bread with you
2. Basic: Pred I (Compl Eat (AdvNP Bread (With You)))
BasicChi: 我 吃 和 你 一 起 的 面 包

BasicEng: I eat bread with you
```

Ambiguity

- A string that parses to many trees is **ambiguous**
- This is one of the main problems of NLP (=Natural Language Processing)
- Example: translation from English to Chinese needs **disambiguation**.
  - **syntactic ambiguity**: whether the Adv modifies NP or VP
  - **lexical ambiguity**: many senses of *with*

Addressing lexical ambiguity

1. Define a separate abstract syntax function for each sense

```
With         : NP -> Adv
With_company : NP -> Adv
```

2. Linearizations of each sense may or may not be different

```
Chinese:  With np         = "和" ++ np ++ "一 起"

          With_company np = "跟" ++ np

English:  With_company np = "with" ++ np
```

3. Try to pick the tree with the right sense

```
Basic> p -lang=Eng "I eat rice with you" | l -treebank

1. Basic: Pred I (AdvVP (Compl Eat Rice) (With You))
BasicChi: 我 在 和 你 一 起 吃 饭
BasicEng: I eat rice with you

2. Basic: Pred I (AdvVP (Compl Eat Rice) (With_company You))
BasicChi: 我 在 跟 你 吃 饭
BasicEng: I eat rice with you

3. Basic: Pred I (Compl Eat (AdvNP Rice (With You)))
BasicChi: 我 吃 和 你 一 起 的 饭
BasicEng: I eat rice with you

4. Basic: Pred I (Compl Eat (AdvNP Rice (With_company You)))
BasicChi: 我 吃 跟 你 的 饭
BasicEng: I eat rice with you
```

Morphology

- the study of different forms of words
- in Western languages, a word can have *thousands* of forms
    - ★ in English, at most 5; in Finnish, up to 10k
- we don't want to write all forms in the grammar, but define **functions** that produce them
- morphological function = **paradigm**

English regular verb paradigm

```
resource English = {

param
  VForm = Inf | Pres | Past | PastPart | PresPart ;

oper
  regV : Str -> VForm => Str = \walk -> table {
    Inf => walk ;
    Pres => walk + "s" ;
    Past | PastPart => walk + "ed" ;
    PresPart => walk + "ing"
    } ;

-- examples
  Walk = regV "walk" ;
  Annoy = regV "annoy" ;
  Reject = regV "reject" ;
}
```

Almost regular verbs

Ending with *s*, *sh*, *x*, *z*,...:

kiss, kisses, kissed, kissed, kissing

Ending with *e*:

use, uses, used, used, using

Ending with *y*:

cry, cries, cried, cried, crying

Except if preceded by a vowel:

play, plays, played, played, playing

Ending with vowel + consonant:

wrap, wraps, wrapped, wrapped, wrapping

Smart paradigms

Functions with **regular expression pattern matching**:

```
smartV : Str -> VForm => Str = \s -> case s of {
  _ + ("s"|"sh"|"x") =>
     table VForm [s ; s+"es"  ; s+"ed"  ; s+"ed"  ; s+"ing"] ;
  x + "e" =>
     table VForm [s ; s+"s"   ; s+"d"   ; s+"d"   ; x+"ing"] ;
  _ + ("a"|"e"|"o"|"u") + "y"
     => regV s ;
  x + "y" =>
     table VForm [s ; x+"ies" ; x+"ied" ; x+"ied" ; s+"ing"] ;
  _  =>
     regV s
  }
```

What we have shown so far

- GF can define grammatical correctness
- GF can relate many languages via abstract syntax
- GF can be run in the shell, to develop and test grammars
- GF can be run on the web, to build end-used applications

What we have not shown yet

- GF supports big grammars
- GF supports disambiguation
- GF grammars can be written without knowing the target language
- GF grammars can be combined with statistics
- GF grammars can be run on mobile devices
- GF grammars support speech and dialogue applications

Plan for tomorrow

Lab 8-9.40

- get everyone started with GF installation
- complete the Basic grammar with some useful things

Lecture 13:30-15:10

- introduce the GF Resource Grammar Library (RGL)
- show how RGL helps language learning
- show how to write big grammars, even without knowing the target language