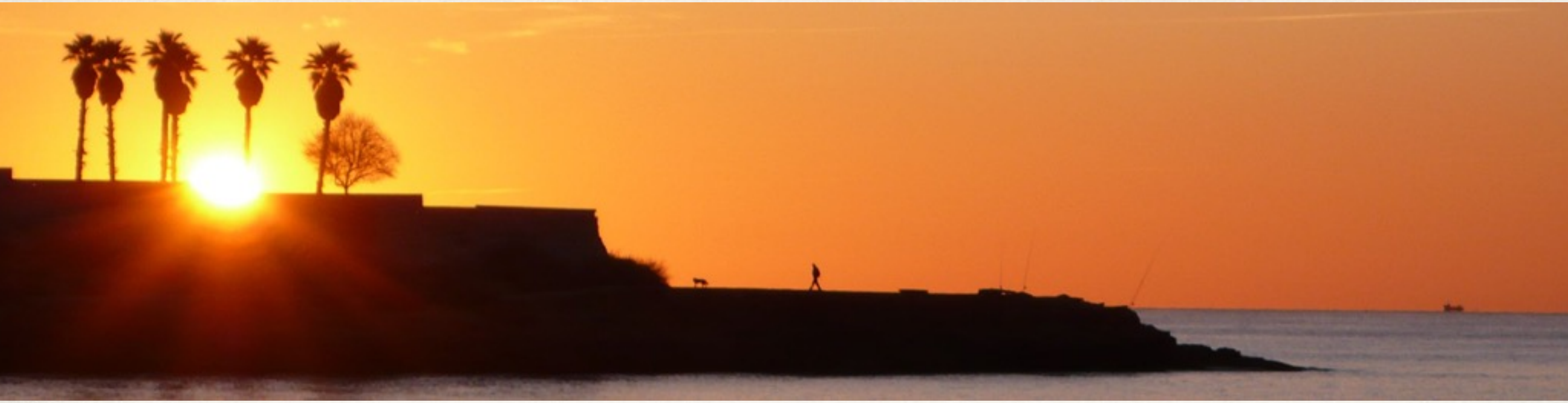


Language processing through logic grammars and constraints

Veronica Dahl (SFU, Canada) and Henning Christiansen (Roskilde University, Denmark)

Iclp16 Tutorial– October 2016

Who are we?

- **HENNING**: Danish computer scientist; works since 1980 comp.sci. disciplines and a range of interdisciplinary topics, including (constraint) logic programming technology and applications for AI and language processing, database theory, knowledge representation and reasoning, bioinformatics, and recently interactive installations in art museums.
- Developed several high-level logic programming systems, including CHR Grammars, and has shown an important, direct correspondence between abductive reasoning and constraint logic program with CHR.
- Received best paper award (with John Gallagher) at ICLP 2009
- **VERONICA**: Argentine/Canadian computer scientist- one of the 15 founders of the field of logic programming, most notably for her work on logic grammars and natural language processing.
- Pioneered extensions and uses of logic programming in the fields of computational linguistics, deductive knowledge bases, computational molecular biology and web based virtual worlds.
- Received numerous scientific awards -- such as the Calouste Gulbenkian Award for Science and Technology

A few remarks before we start

- All example programs available on the website *(TBA)*
  - Tested in SICStus 4; should be compatible with SWI
- No theorems (find them in the references), just programming :)
- No time for exercises during the course :(
- Please feel free to ask questions, to disagree even.

Introduction

- What is **Computational Linguistics**?
  - The art of *simulating* language understanding by computers

- **Language in a General Sense**:
  - Spoken human languages: speech, *text, dialogue.*
  - Molecular Biology languages
  - Rhythmic, dance, poetic, musical, …

- What are **Logic Grammars**?
  - *Symbol rewriting formalisms* that view language descriptions as executable programs. The symbols they rewrite are *logic grammar symbols* (i.e., identifiers plus logic terms: variables, constants or functional expressions)

Understanding Spoken Language

A literary and an everyday example:

- *Love's heralds should be thoughts, which ten times faster glide that the sun's beams, driving back shadows over lowering hills.*
  (Juliet's monologue, scene V, Romeo & Juliet)

- *I was caught speeding. He gave me a ticket.*

What language can express encompasses no less than the entire range of human experience

Therefore, the ***central issues are the same as in AI***: domain knowledge, problem solving, reasoning, non-monotonicity, belief revision, metaphor, planning, learning… Plus a few of its own.

Language Analysis

Levels

- *Prosody*: rhythm and intonation
- *Phonology*: sounds
- *Morphology*: components of words (morphemes)
- *Syntax*: rules for combining words into sentences and phrases
- *Semantics*: meaning of words, sentences
- *Pragmatics*: ways of using language
- *World Knowledge*: physical, human, knowledge

Stages

- *Parsing*: analyses syntactic structure: verifies well-formedness, produces parse tree
- *Semantic Interpretation*: produces some meaning representation of the utterance.
- *Expanded Interpretation*: adds structures from the knowledge base

Ex. 1: a logic grammar for the formal language $\{a^n\ b^m\}$

```
s - -> as, bs.

as- -> [a].            bs - -> [b].
as- -> [a], as.        bs - -> [b], bs.
```

Parse (analyse) ?- s([a,b],[]).

Generate (synthesize) ?-s(String,[]).

Derivation:

```
          s
    as          bs
[a]  as       [b] bs
[a] [a]       [b] [b]
```

compiles into the logic program:

```
s(P1,P):- as(P1,P2), bs(P2,P).

as([a|P],P).
as([a|P1],P):- as(P1,P).      (bs/2 is symmetric)
```

======================================

Trace: ?- s([a,a,b,b],[])

?- as([a,a,b,b],P2), bs(P2,[])

?- as([a,b,b],P2), bs(P2,[]) (P2=[b,b])

?- bs([b,b],[]) …and so on.

Ex. 2: a logic grammar for a human language sentence

- grammar symbols can be logic terms, and Prolog calls are allowed (between brackets), e.g.

```
s - -> noun (X), verb (Y), {agree(X,Y}.

noun(plural) - -> [lions].
verb(plural) - -> [sleep].

agree(Number,Number).
```

compiles into

```
s(P1,P2):- noun(X,P1,P), verb(Y,P,P2), agree(X,Y).

noun(plural,[lions|X],X).
verb(plural,[sleep|X],X).

agree(Number,Number).
```

Ex. 3: Obtaining a parse tree as a side effect of parsing

```
sent --> name, verb.

  name -->[john].
  name -->[mary].

  verb --> [laughs].
  verb --> [sings].
==================================
sent(sent(TN,TV)) --> name(TN), verb(TV).

  name(name(john)) -->[john].
  name(name(mary)) -->[mary].

  verb(verb(laughs)) --> [laughs].
  verb(verb(sings)) --> [sings].
```

Ex. 4: Front ends to Prolog Databases

Aim: to recover, as the internal representation a sentence parses into, the corresponding Prolog call to a database predicate/s

Successive parsers:

- simple sentences constructed around proper names, verbs and prepositions.
- natural language quantifiers (the, a, some, no, few, …)
- Modularity: complements
- Interrogative and relative clauses

We want to consult in English the database:

```
shines(helios).

reflects(selene,helios).
reflects_upon(selene,helios,gaia).
```

Step 1. Characterize the English Subset

Consider sample pairs input/desired output, e.g.:

Selene shines --> shines(selene)

Helios shines upon Gaia --> *shines_upon(helios,gaia)*

Names --> *constants* (e.g. selene)

Content Verbs --> *predicate names*; they can contribute a structure to be completed (e.g. reflects(X,Y))

Linking Verbs (is, has) --> *serve to link with* a noun or adjective) *which will induce the predicate name* (e.g. charming/1, father_of/2)

Prepositions --> *become part of a predicate name* (e.g. reflects-upon(X,Y))

Step 2: Building the desired meaning

```
name(gaia) --> [gaia].
name(helios) --> [helios].
name(selene) --> [selene].

 np(X) --> name(X).

iv(X,shines(X)) --> [shines].
tv(X,Y,reflects(X,Y)) --> [reflects].

pp(X)  --> prep, np(X).

prep --> [upon].
```

Tying the noun phrase with the vp:

```
s(M) --> np(X), vp(X,M).
```

Step 3: Adding Quantified Noun Phrases

```
np(X,VP,M) --> d(X,NP,VP,M),
   n(X,NP).

d(X,NP,VP,the(X,NP,VP)) --> [the].
d(X,NP,VP,a(X,NP,VP)) --> [a].
d(X,NP,VP,no(X,NP,VP)) --> [no].

n(X,sun(X)) --> [sun].

vp(X,M)  --> tv(X,Y,Sk), np(Y,Sk,M).


s(M) --> np(X,VP,M), vp(X,VP).
```

TESTS

```
i([helios,illuminates,gaia]).
i([selene,reflects,helios,upon,gaia]).
i([selene,reflects,helios]).
…

go:- i(Input), write(Input), nl,
   s(M,Input,[]), write(M), nl, fail.

% Also try generating from known M
```

Step 4: A more uniform treatment of verbs

```
comps([],M,M) --> [].
comps([C1|L],M1,M]) --> comp(C1,M1,M2), comps(L,M2,M).

comp([P,X],M1,M) --> prep(P), np(X,M1,M).

verb(X,shines(X),[]) --> [shines].

verb(X,reflects(X,Y),[[dir(Y)]]) --> [reflects].

verb(X,reflects_upon(X,Y,Z),[[dir(Y)],[upon,Z]])
 --> [reflects].
```

A single rule now suffices for all verb phrases:

```
vp(X,M)  --> verb(X,M1,L), comps(L,M1,M).
```

Using Complements for other constructions

Peter is happy with math ~~> happy_with(peter,math).

adj(X,happy_with(X,Y),[[with,Y]]) --> [happy].

adj_phrase(X,M1,M) --> adj(X,P,L), comps(L,M1,M).

Non-classical reasoning for computational linguistics

- Assumptions (linear and intuitionistic logic inspired)
- Abduction
- Constraint-Based

- Beyond classical logic: “what if”, “possible cause” scenarios.

- **Assumptions**: evolved from Girard’s work on linear logic, where *linear implication* serves to represent state change, in the form of resources that are ***consumed*** (exactly once) to produce other resources, e.g.

```
cream -> butter
```

Affine linear implication (or ***linear assumption***): resources can be consumed at most once.

Assumptions in logic grammars

- Assumptions developed by [Dahl & al., 1997] refined by [Christiansen, Dahl, 2004, ...]
- Included in the HYPROLOG and CHRG systems (more later)

$+A$	Assert linear assumption A for subsequent proof steps. Linear means "can be used once".
$*A$	Assert intuitionistic assumption A for subsequent proof steps. Intuitionistic means "can be used any number of times".
$-A$	Expectation: consume/apply existing intuitionistic assumption in the state which unifies with A.
$=+A$, $=*A$, $=-A$	Timeless versions of the above, meaning that order of assertion of assumptions and their application or consumption can be arbitrary.

Linear assumptions: noted "+", consumed with "-"

- A variant of append/3:

```
append(L1,L2,L):- +note(L2), app(L1,L).

    app([ ], L):- -note(L).
    app([H|T], [H|L1]) :- app(T, L1).
```

Exercise: Complete the following program with clauses for s/0, so that it describes the language $\{a^n\ b^n\ c^n\}$ scrambled.

```
assumptions a/0, b/0, c/0.

input:- +a, +b, +b, +c, +a, +c, s.

all_consumed:- -P, !, fail.
all_consumed.
```

Relative Clauses through Linear Assumptions

Example: "the house that Jack built" ("Jack built *the house*")

np(X,M,M) --> name(X).
np(X,VP,M) --> d(X,NP,VP,M), n(X,N), rel(X,N,NP). % X represents the antecedent
np(X,NP,NP) --> {-missing(X)}. % recovers X as the value of the non-overt noun phrase

rel(X,N,and(N,R)) --> [that], {+missing(X)}, s(R). % records that X will be missing in R
rel(_,N,N) --> []. % there is no relative clause

Intuitionistic Assumptions: noted "*", consumed with "-"

ANAPHORA: Resolving pronouns to their antecedents

```
np(X,Gender) --> name(X,Gender), {*acting(X,Gender)}.
np(X,Gender) --> {-acting(X,Gender)}, pronoun(Gender).

sentence(s(A,V,B)) -->  np(A,_), verb(V), np(B,_).

sentences((S1,S2)) --> sentence(S1),sentences(S2).
sentences(nil) --> [].

pronoun(fem) --> [her].
```

Sample Test:

```
?- phrase(sentences(S), [peter,likes,martha, mary,hates,her]).

 S = (s(peter,like,martha),s(mary,hate,mary),nil) ? ;
 S = (s(peter,like,martha),s(mary,hate,martha),nil) ? ;
  no
```

Timeless Assumptions: noted “=*”, consumed with “=-“

<u>ELISION: Reconstructing missing elements</u>

Peter likes and Mary hates Martha

sentence(s(A,V,B)) --> np(A,_), verb(V), np(B,_), {=*obj(B)}.
(timeless assumption: can be consumed either before or after being assumed)

sentence(s(A,V,B)) --> np(A,_), verb(V), [and], {=-obj(B)}.
(timeless consumption)

Constraint-Based Reasoning

- Constraint store as a ***knowledge base***
- CHR rules as “business logic” or “integrity constraints” ≈ rules about knowledge
- Prolog or additional CHR rules as “driver algorithm”

A motivating example . . .

A motivation example (1:3)

Consider the following Prolog program:

```
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

What is it supposed to mean?

Let's try it:

```
| ?- happy(henning).
! Existence error in user:rich/1
! procedure user:rich/1 does not exist
! goal:  user:rich(henning)
```

Another way of saying **no** :(

The problem: Prolog's ***closed world*** assumption

A motivation example (2:3)

Let's try with a little help from CHR:

```
:- use_module(library(chr)).
:- chr_constraint rich/1, professor/1, has/2.

happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

Intuition: Make certain predicates "*open world*".

Let's try it:

```
| ?- happy(henning).
rich(henning) ? ;
professor(henning),
has(henning,nice_students) ? ;
no
```

Looks more like it, but still not perfect . . .

A motivation example (3:3)

Adding a bit of "universal knowledge" in terms of a CHR rule:

```
:- use_module(library(chr)).
:- chr_constraint rich/1, professor/1, has/2.

professor(X), rich(X) ==> fail.
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

Let's try it:

```
| ?- happy(henning), professor(henning).
professor(henning),
has(henning,nice_students) ? ;
no
```

Thus:

- CHR constraints represent ***concrete facts*** about a given world.
- CHR rules represent ***universal knowledge*** valid in any world.

Abduction and CHR for Computational Linguistics

- **Abductive Reasoning with CHR**
  - Definition, implementation in CHR, applications
- **Language Analysis 1: With DCGs (= Prolog) plus CHR**
- **Language Analysis 2: CHR Grammars**
- **Probabilistic Abductive Reasoning with CHR (not included in this tutorial)**
  - Each branch of computation represented as a CHR constraint
  - Allows for best-first computations

Abduction????

A term due to C.S.Pierce (1839-1914); the trilogy:

- ***Deduction***
  - Reason "forward" in a sound way from what we know already; finding its logic consequences; i.e., nothing really new
- ***Induction***
  - Creating rules from example, so we can use these rules in new situations
- ***Abduction***
  - Figure out which currently unknown facts that can explain an observation; unsound from logical point of view ;-)

Abduction with CHR

You've seen it already!

```
:- use_module(library(chr)).
:- chr_constraint rich/1, professor/1, has/2.
prof(X), rich(X) ==> fail.
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

```
| ?- happy(henning), professor(henning).
professor(henning),
has(henning,nice_students) ? ;
no
```

In logic programming terms:

Figure out which facts should be added to the program to make a the given goal succeed

Traditional definition of Abductive Logic Programming (ALP)

- An ***abductive logic program*** consist of
  - A number of ***predicates***, some of which are called ***abducibles,*** *Abd*
  - A usual ***logic program***, *P*, in which abducibles do not occur in the head of rules
  - A set of ***integrity constraints***, *IC*, which are formulas that must always be true
- An abductive answer to a query *Q* is a set of abducible atoms *Ans* such that
  - $P \cup Ans \models Q$ and $P \cup Ans \models IC$
- (It is also possible to include an answer substitution, but we ignore that)

Translating ALP into Prolog+CHR

Abducible predicates	→	CHR constraints
Integrity constraints	→	CHR rules

Let us inspect our sample program:

```
:- use_module(library(chr)).
:- chr_constraint rich/1, professor/1, has/2.

prof(X), rich(X) ==> fail.

happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

Compare with “traditional” ALP

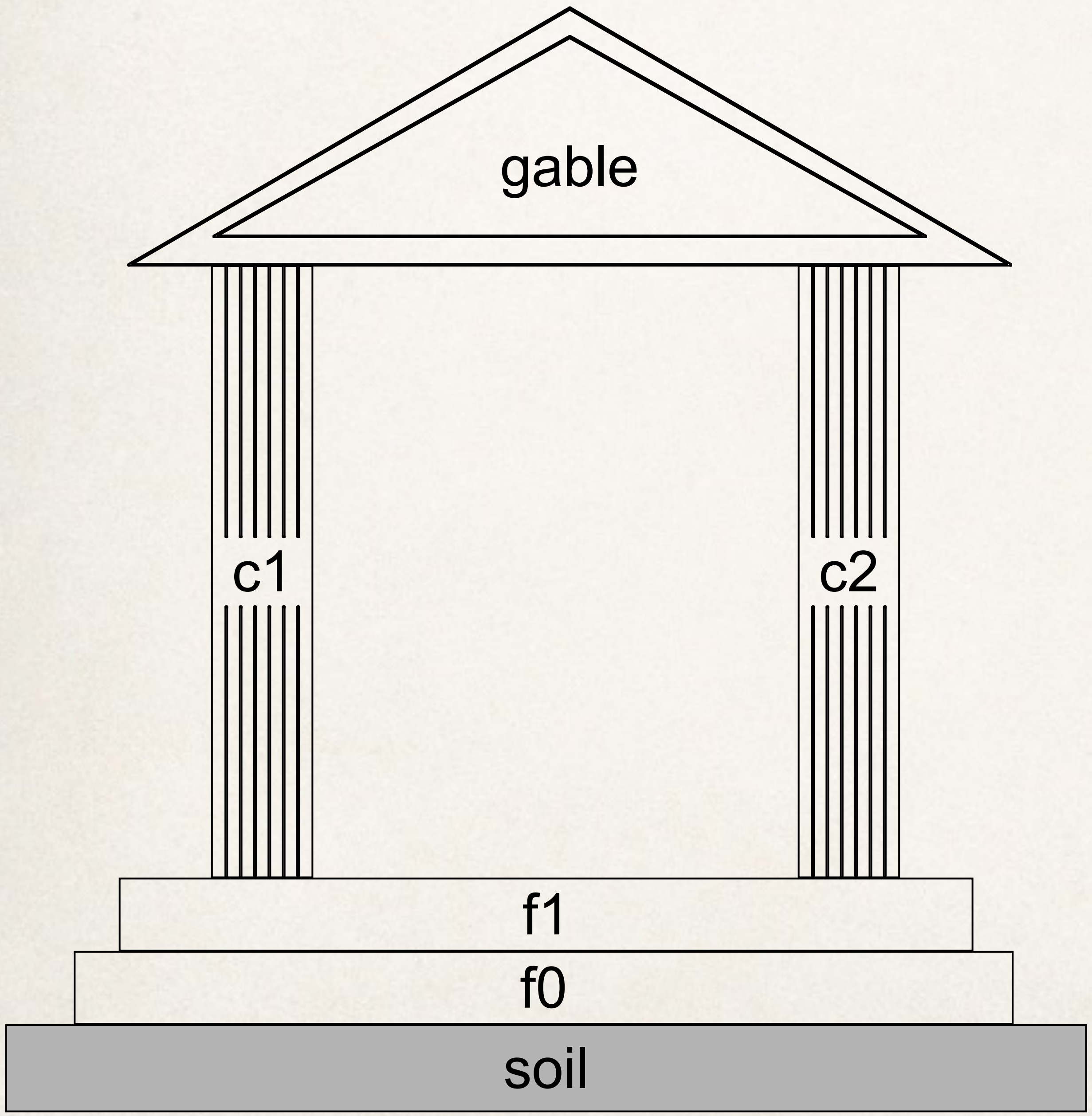
- Usually defined by difficult algorithms and implemented with complicated meta-interpreters; see references to work by Kowalski, Kakas & al, Decker, ...
- Our approach employs existing technology
  - in the most efficient way
  - with no meta-level overhead
  - and we can use all of Prolog and CHR (libraries, all sorts of dirty tricks)
- To our knowledge, by far the most efficient implementation of ALP
- The cost? Only very limited use of negation (you can read about that)

Planning as Abduction

- ***Problem:*** Given a number of tasks + restrictions on the order in which they can be done.
- ***Solution:*** An assignment of a time point to each task so the restrictions are obeyed.
- ***In our terms***
  - Abducibles (CHR constraints): Assignment of a time point to a task
  - Integrity constraints (CHR ***rules***): The restrictions
  - The goal (≈ desired observation): *"The work has been done."*

Planning as Abduction, example

Architect's drawing:



CHR rules:

```
mount(P0,Time0), mount(P1,Time1) ==>
    supports(P0,P1), Time0 > Time1
    | fail.
mount(P,Time0), mount(P,Time1) ==>
    Time0 \= Time1
    | fail.
```

Prolog facts:

```
part(gable).
part(c1).
...
supports(soil,f0).
supports(f0,f1).
```

Driver algorithm in Prolog: *next slide*

CHR rules:

```
mount(P0,Time0), mount(P1,Time1) ==>
  supports(P0,P1), Time0 > Time1
  | fail.

mount(P,Time0), mount(P,Time1) ==>
  Time0 \= Time1
  | fail.
```

Prolog facts:

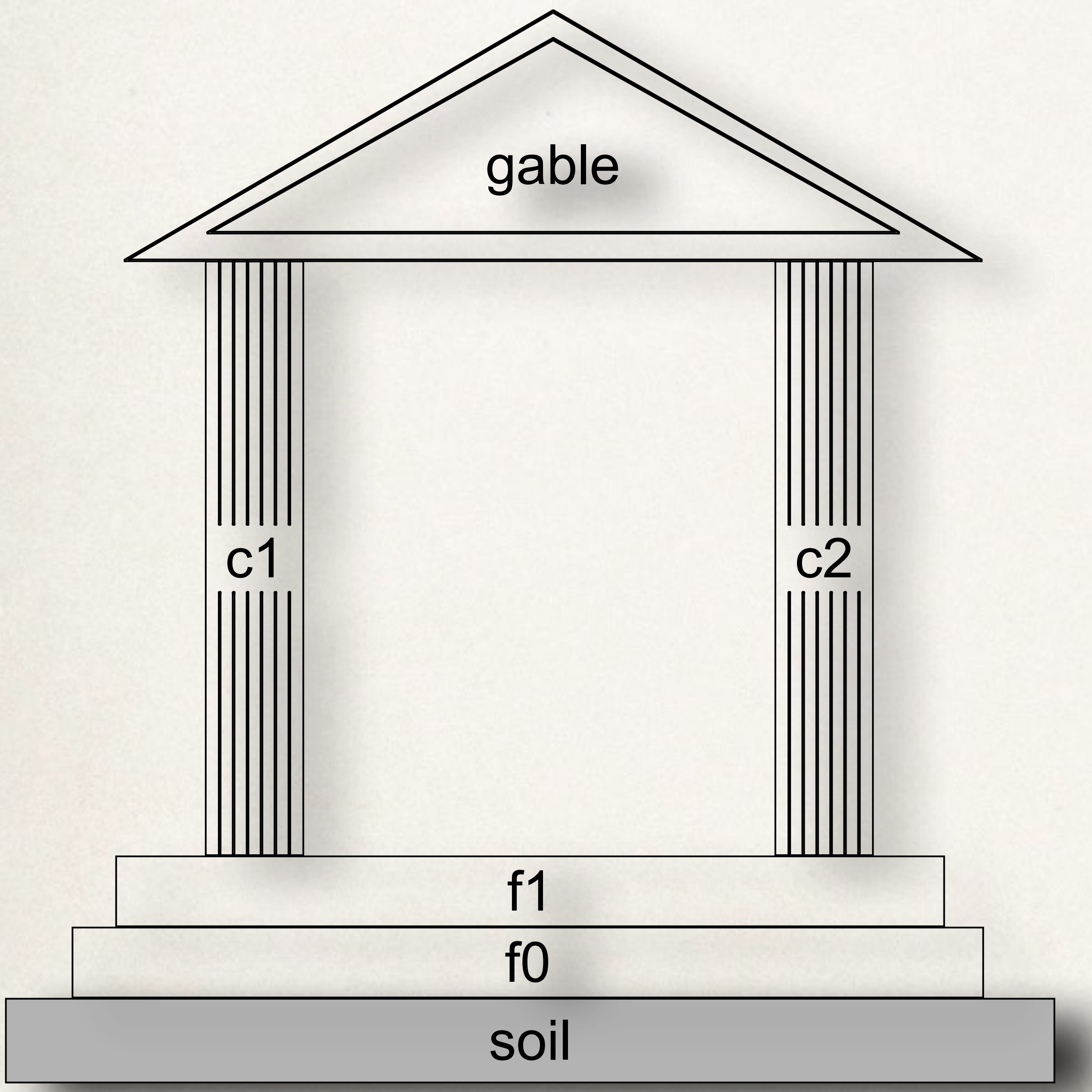
```
part(gable).
part(c1).
...
supports(soil,f0).
supports(f0,f1).
```

Driver algorithm in Prolog:

```
built:-  mount(soil,0), build(1).

build(6):- !.

build(Time):-
   part(P),
   mount(P,Time),
   Time1 is Time+1,
   build(Time1).
```

```
| ?- build.
mount(gable,5),
mount(c2,4),
mount(c1,3),
mount(f1,2),
mount(f0,1),
mount(soil,0) ? ;

mount(gable,5),
mount(c1,4),
mount(c2,3),
mount(f1,2),
mount(f0,1),
mount(soil,0) ? ;

no
```

Wanna see an animation of the first solution?

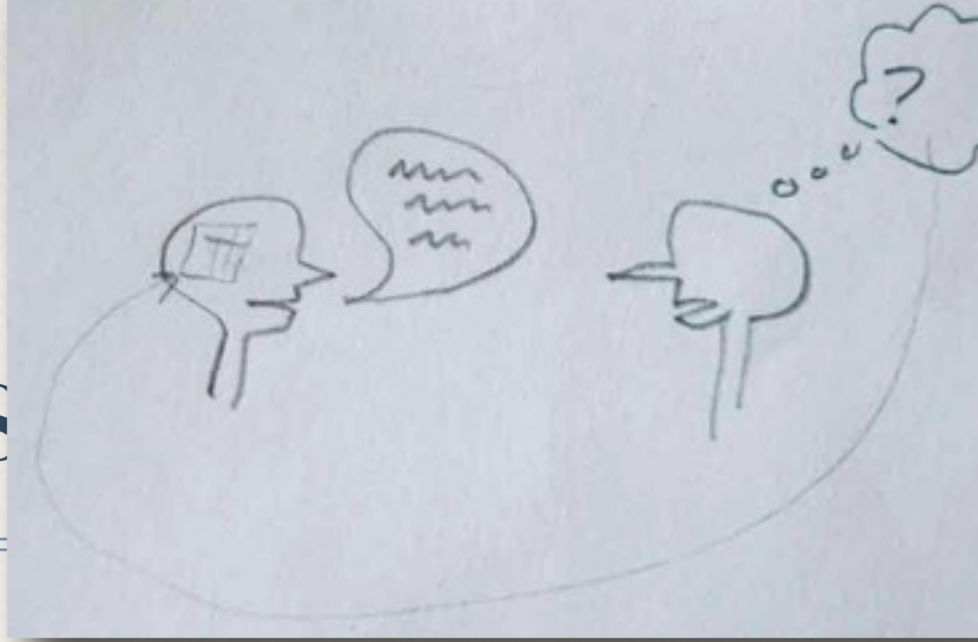



```
| ?- build.
mount(gable,5),
mount(c2,4),
mount(c1,3),
mount(f1,2),
mount(f0,1),
mount(soil,0) ?

mount(gable,5),
mount(c1,4),
mount(c2,3),
mount(f1,2),
mount(f0,1),
mount(soil,0) ? ;

no
```

More on planning

- With the same technique, we can extend with
  - ***Duration***, e.g., it takes 8 hours to mount a column
  - ***Resources***, e.g., to mount a column, we need 1 crane and 12 workers
  - ***Restrictions***+= At any time, the resources in use must not exceed the maximum available (say, 2 cranes and 30 workers)
- ***Your exercise (voluntary!):*** Extend the example and implement the scheme above
- ***Your next exercise (difficult & voluntary):*** Extend your program so it tries to find a solution that minimizes the no. of unoccupied workers — or, alternatively, the solution that finishes the building as early as possible.

Integrating DCG, CHR and Assumptions



- My favourite metaphor: ***"Interpretation as abduction"***
  - Jerry R. Hobbs, Mark E. Stickel, Douglas E. Appelt, Paul A. Martin: Interpretation as Abduction. Artif. Intell. 63(1-2): 69-142 (1993)
  - Also Charniac, McDermott (1985), Gabbay & al (1997), Christiansen (1993)
- We use Prolog's Definite Clause Grammars (DCGs) extended with CHR
- Resulting method:
  - Integrates semantic and pragmatic analysis (in contrast to tradition methods)
  - A great experimental tool for students and researcher in linguistics; easy to approach and "advanced" analyses can be specified in very short time.

A short historical note

- Basic idea comes from CHR Grammars (Christiansen, 2001-2005), that we will look at later in the course
- Idea of using DCGs emerged through our joint work, 2002, and onwards....
  - Lead to the ***Hyprolog*** system (Christiansen, Dahl, ICLP, 2005)
  - adds a thing layer of syntactic sugar upon Prolog+CHR that supports ***abduction***
  - and ***assumptions***,
- Here we show things first expressed directly in Prolog(DCG)+CHR

Adding semantics/pragmatics

- Traditionally:
  - ***"Semantics"*** = context-independent (lambda) terms
  - ***"Pragmatics"*** = relating "Semantics" to context, e.g., mapping variables to (identifiers of) "real worlds"
- The present approach ***blurs this distinction,*** which suits much better my intuition about how humans process language
- You may see this in the examples

A DGC with CHR for sem/pragm

First version: Only noting facts

```
:- chr_constraint at/2, see/2.

story --> [] ; s, ['.'], story.

s --> np(X), [sees], np(Y),
      {see(X,Y)}.

s --> np(X), [is,at], np(E),
      {at(E,X)}.

s --> np(X), [is,on,vacation],
      {at(vacation,X)}.

np(peter)    --> [peter].
np(mary)     --> [mary].
np(jane)     --> [jane].

np(chr_fall_school)
       --> [the,iclp,fall,school].

np(our_course)
       --> [our,course].

np(vacation) --> [vacation].
```

```
:- phrase(story,
      [peter,sees,mary,'.',
       peter,sees,jane,'.',
       peter,is,at,the,
             iclp,fall,school,'.',
       mary,is,at,our,course, '.',
       jane,is,on,vacation,'.']).

at(vacation,jane),
at(our_course,mary),
at(iclp_fall_school,peter),
see(peter,jane),
see(peter,mary) ?
```

2nd version: Adding world knowledge

```
:- chr_constraint at/2, in/2, see/2, skypes/2.

at(iclp_fall_school,X) ==> in(nyc,X).

in(Loc1,X) \ in(Loc2,X) <=> Loc1=Loc2.

at(our_course,X)    ==> at(iclp_fall_school,X).

at(vacation,X)           ==> in(Loc,X), diff(Loc,nyc).

see(X,Y) ==> true |
    (in(L,X), in(L,Y)
     ; in(Lx,X), in(Ly,Y), diff(Lx,Ly), skypes(X,Y)).

diff(...) <=> ... . % Homemade version of dif/1 for nicer output

% Grammar rules: Exactly the same as before
```

```
| ?- phrase(story, [mary,is,at,our,course,'.']).

at(iclp_fall_school,mary),
at(our_course,mary),
in(nyc,mary) ?
```

```
| :- phrase(story,
      [peter,sees,mary,'.',
       peter,sees,jane,'.',
       peter,is,at,the,
             iclp,fall,school,'.',
       mary,is,at,our,course, '.',
       jane,is,on,vacation,'.']).
```

```
at(vacation,jane),

at(iclp_fall_school,mary),

at(our_course,mary),

at(iclp_fall_school,peter),

in(_A,jane),

in(nyc,mary),

in(nyc,peter),

see(peter,jane),

see(peter,mary),

skypes(peter,jane),

diff(nyc,_A) ?
```

What is HYPROLOG, btw.?

- A system that adds a thin layer of syntactic sugar on top of Prolog+CHR
  - Special syntax for declaring abducibles (as you have seen)
  - Utilities and options for abductive reasoning (not shown here)
  - Assumptions implemented as you have just seen
- Implementation principles
  - Using same facilities as DCGs and CHR: `term_expansion`
  - Operator declarations in Prolog.

A realistic example: Extracting UML diagrams from Use Cases

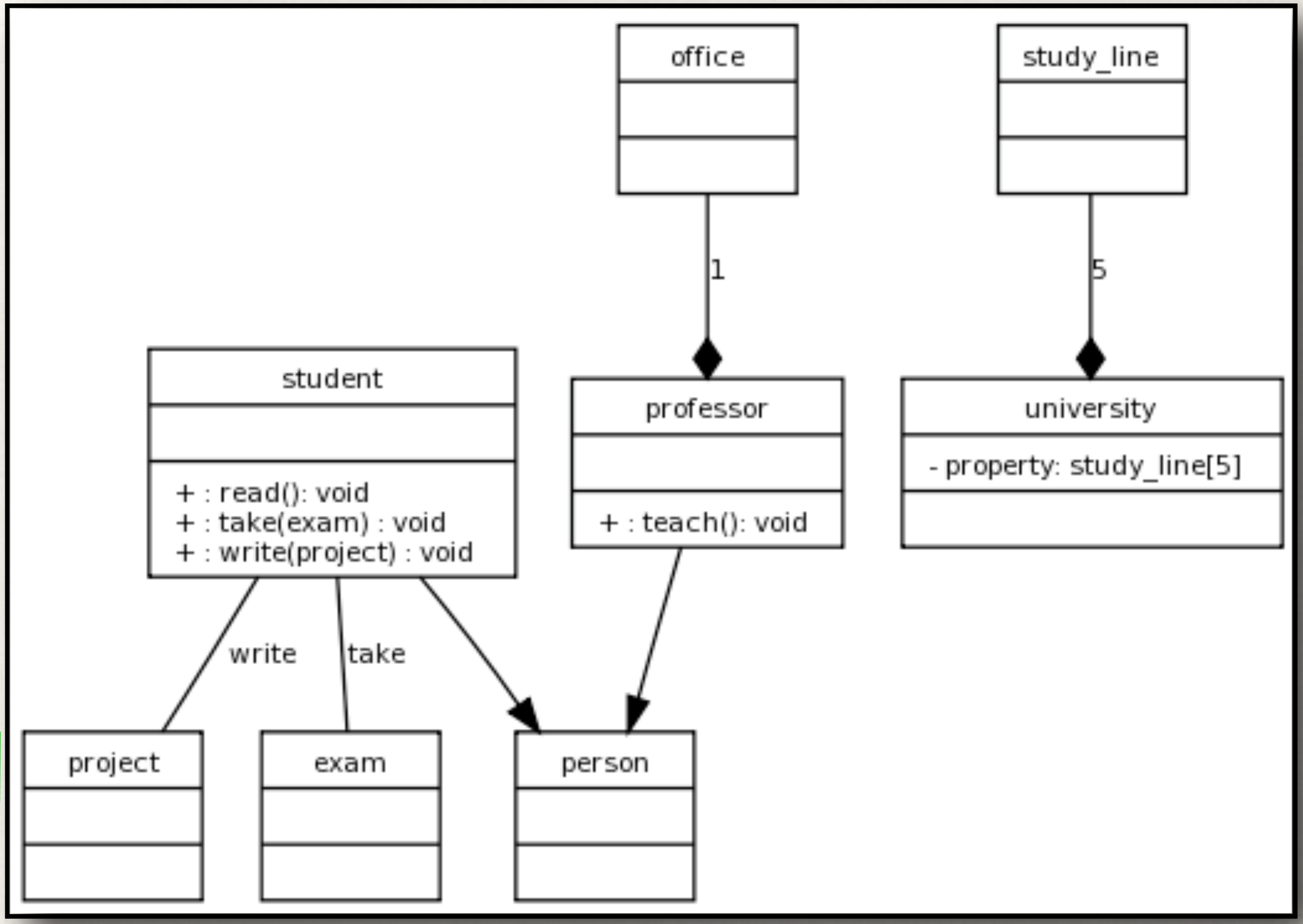
- Based on 4 week project work with two students [Christiansen, Have, Tveitane, 2007 a+b]
- Only a brief sketch; here using the full power of CHR without caring about formal details ;-)
- Use cases?? In the OOA/OOP tradition, small *stories* about the world which the system to be developed will fit it.
- According to OOA principles, UML diagrams describing classes and their property, etc., are produced manually from use cases...
- But why not do it automatically, when we have a tool such as Prolog +CHR which is perfectly suited for semantic/pragmatic analysis

Example of input and output

From uses cases:

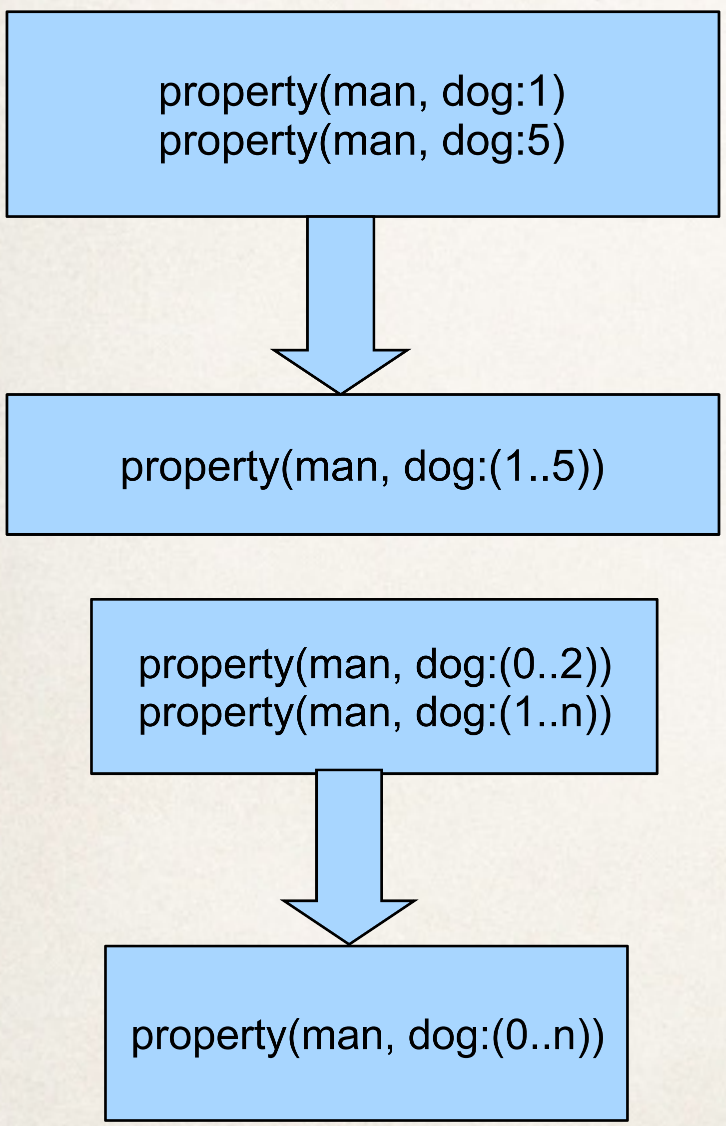
- *The professor teaches. A student reads, writes projects and takes exams. Henning is a professor. He has an office. The university has five study lines. Students and professors are persons.*

... extract info and produce



Examples of CHR rules for knowledge extraction (1:2)

Merging cardinalities, e.g.:




```
property(C,P:N), property(C,P:M) <=>
   count(N), count(M), N=<M
| property(C,P:(N..M)).
```

```
property(C,P:(N1..M1)),property(C,P:(N2..M2)) <=>
   min(N1,N2,N), max(M1,M2,M),
   property(C,P:(N..M)).
```

(NB: “`n`” is a special symbol meaning “many”)

Examples of CHR rules for knowledge extraction (2:2)

Pronoun resolution, e.g.,

> *Jack and John are teachers. Jack teaches music. John teaches computer science. Mary is a student.* ***He*** *has many students.*

Our heuristics: Take most recent referent that matches gender and when no ambiguity arises; in case of ambiguity, we call it an error

> ~~*Jack and John are teachers. He ....*~~

```
sentence_no(Now), referent(No,G,Id,T) \  expect_referent(No,G,X) <=>
   T < Now,  there is no other relevant referent with Timestamp > T
|
   if there is another relevant referent with Timestamp = T then
      X = errorcode(ambiguous)
   else
      X = Id.
```

Summary: Language analysis with DGC+CHR+ Assumptions

- Natural and straightforward integration of semantic/pragmatic analysis with parsing
- 10^6 times easier for this purpose than any other, known tools
- DCGs (i.e., Prolog) provide parsing plus auxiliary predicates
- CHR constraint store as knowledge base; CHR rules for world knowledge
- We showed
  - Direct use of DCG+Prolog
  - HYPROLOG which provided syntactic sugar, Assumptions and various auxiliaries
  - A realistic example with pronoun resolution and semantic reasoning

CHR Grammars

CHR Grammars, background

- Around 2000, Henning noticed that it was easy to write bottom-up parsers with CHR
- Experiments showed that there was much more power in this principle than expected:
  - very flexible context-dependent rules, gaps, parallel matching, ...
  - interesting treatment of ambiguity
  - having parsing to depend on “semantics”, and a lot of other stuff
- 2002: CHR Grammar system released; update to recent version of SWI Prolog
  Available at **`http://www.ruc.dk/~henning/chrg/`**
- Main publication 2005 [JLP]
- Applications: The full power of CHR Grammars still needs to be discovered

CHR Grammars, overview

- Bottom-up parsing with CHR, our principle
- A grammar notation and its translation into CHR
- What we can do in CHR Grammars, derived from the translation into CHR
  - We have squeezed as much power as possible out of CHR without caring whether it is useful *(our preferred design methodology ;-)*
- Example: a biological application

Bottom-up parsing with CHR

Encode the string as a set of constraints with ***word boundaries***

"Peter likes Mary"

```
token(0,1,peter),token(1,2,likes),token(2,3,mary).
```

A bottom-parser that checks word/phrase boundaries

```
:- chr_constraint np/2, verb/2,
   sentence/2, token/3.

token(N0,N1,peter) ==> np(N0,N1).
token(N0,N1,mary)  ==> np(N0,N1).
token(N0,N1,likes) ==> verb(N0,N1).

np(N0,N1), verb(N1,N2), np(N2,N3)
                 ==> sentence(N0,N3).
```

```
?-  ... .
np(0,1),
verb(1,2),
np(2,3),
sentence(0,3),
token(0,1,peter),
token(1,2,likes),
token(2,3,mary) ?
```

A grammar notation upon CHR

Why write this?

```
:- chr_constraint np/2, verb/2,
   sentence/2, token/3.
token(N0,N1,peter) ==> np(N0,N1).
token(N0,N1,mary)  ==> np(N0,N1).
token(N0,N1,likes) ==> verb(N0,N1).
np(N0,N1), verb(N1,N2), np(N2,N3)
                   ==> sentence(N0,N3).
```

```
?- token(0,1,peter),
   token(1,2,likes),
   token(2,3,mary).
```

When we would like to write this:

```
:- grammar_symbol np/0, verb/0,
                  sentence/0.

[peter] ::> np.
[mary] ::> np.
[likes] ::> verb.

np, verb, np ::> sentence.

end_of_CHRG_source.
```

```
?- parse([peter,likes,mary]).
```

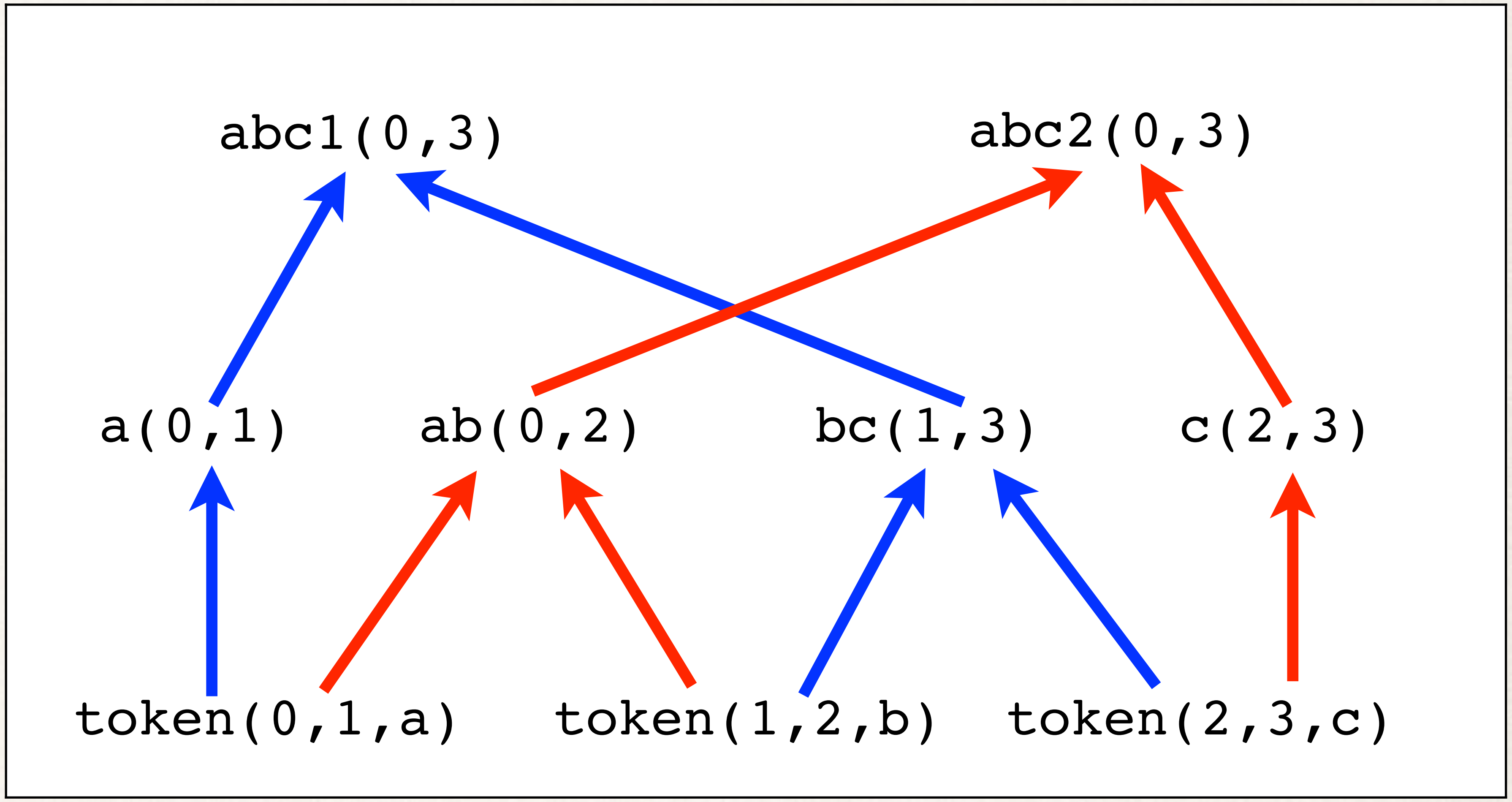
The CHR compiler

compile-on-load using `term_expansion`

Inherent handling of ambiguity

```
[a]   ::> a.
[b,c] ::> bc.
[a,b] ::> ab.
[c]   ::> c.
a, bc ::> abc1.
ab, c ::> abc2.
```

```
| ?- parse([a,b,c])
```



- I.e., all possible parses are run "in parallel"
- You can limit this by, e.g., simplification rules;
  - in the example, you would end up with only `abc1(0,3)`.
- Thus the semantics *very* procedural! (good or bad?)

What else can we put in? (1:5)

- `::>` translates into `==>`
- `<:>` translates into `<=>`
- Order independent syntax for simpagations

```
!a, b, !c <:> ac.
```

translated into

```
b(N1,N2) \ a(N0,N1), c(N2,N3) <=> ac(N0,N3).
```

What else can we put in? (2:5)

Gaps in the head

```
[blip], 7...10, [blop] ::> blipblop
```

- translated into

```
a(N0,N1),b(N2,N3) ::>
   N2-N1 >= 7, N2-N1 =< 10
| ab(N0,N3).
```

- This may be relevant for biologic applications such as RNA folding

What else can we put in? (3:5)

Left and right context

- *left-context* `-\` *core-to-be-reduced* `/-` *right-context* `::>` ....
- For example

```
c1, ..., c2 -\ c3, c4 /- ..., c5 <:> c34.
```

- translated into

```
c1(_,N1),c2(N2,N3),c3(N3,N4),c4(N4,N5),
c5(N6,_)
<=> N1=<N2, N5=<N6 | c34(N3,N5).
```

What else can we put in? (4:5)

Parallel matching

- *one-reading-of-the-text* `$$` *another-reading-of-the-text* `::>` ....
- For example: `a $$ b <:> c.`
- translates into: `a(N0,N1), a(N0,N1) <=> c(N0,N1).`
- And: `a, 5...12 $$ b, c <:>  d`
- translates into:

```
a(N0,N1), b(N0,N11), c(N11,N2)
<=> N1-N2 >= 5, N1-N2 =< 12 | d(N0,N2)
```

- ***Applications?*** *I forgot why I included it, but it is smart, isn't it?*

What else can we put in? (5:5)

- Assumptions as we have seen
- Further equipment for abduction (see paper on CHRG)
- All sorts of utilities and options (see online User's Guide)
- Extra-grammatical constraints in the head and body of rules (...)

Example: Simplification and context for disambiguation

An abstract ~~and highly ambiguous~~ grammar:

```
e, [+], e /- (['+'];[')'];[eof])   <:> e.
e, [*], e /- ([*];[+];[')'];[eof]) <:> e.
e, [^], e /- [X] <:> X \= ^ | e.
['('], e, [')'] <:> e.
[N] <:> integer(N) | e.
```

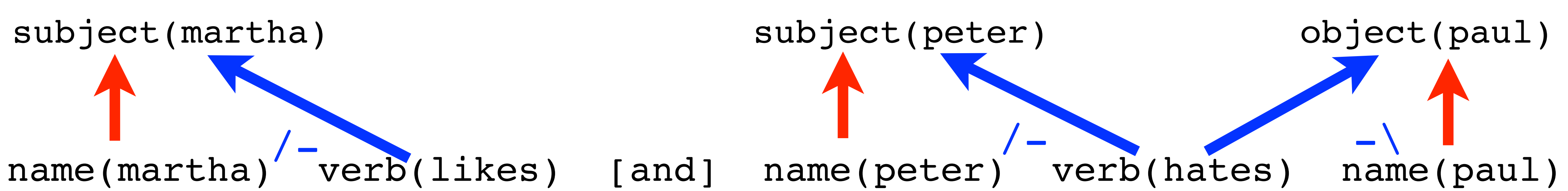
Here we used LR(1) items as right context to disambiguate...

just one special case of what we can do

Example: Context used for tagger-like rules

Classify np's according to position of the verb

```
               name(A) /- verb(_) <:>   subject(A).
verb(_) -\ name(A)                <:>    object(A).
name(A),   [and], subject(B)      <:> subject(A+B).
object(A), [and], name(B)         <:>  object(A+B).
```



Martha likes and Peter hates Paul

A little voluntary exercise

- Write the remaining rules for a grammar that may parse the entire phrase given in the previous slide.
  - to make certain terminal symbols into nonterminals such as `name(mary)`
  - to make certain terminal symbols into nonterminals `verb(likes)`
  - to parse complete sentences, i.e., that include explicit object.
  - to parse incomplete sentences that has implicit object, given by another sentence after "`and`".
- Next, add at attribute to each sentence of the form `fact(`*subject*`,`*verb*`,`*object*`)` and modify your grammar so that it generates the correct "meaning" for each sentence, also the incomplete ones.
  - For example, the first incomplete sentence in the previous example should generate the "meaning" `fact(martha,like,paul)`.
- Extend the grammar with whatever you find interesting.

CHRG for Molecular Biology

- *Structural linguistics of nucleic acids- what we know about this very expressive language for specifying the structures and processes of life:*
- *∑={g,c,a,t} where g and c tend to bind together, and so do c and a*
- *Nucleic acids are* ***not regular*** *(inverted repeats need mirror strings, of the form u v ~u^{R,} with u,v ∑*),*
- ***non-deterministic*** *(a given state doesn't determine uniquely the next state),* ***non-linear*** *(cannot be expressed by grammars that never spawn more than one nonterminal), and*
- ***ambiguous*** *(in a way that can subvert the implicit biological meaning). They are* ***not even context-free…*** *(David Searles)*

Translation: essential in both spoken and biological languages

Codons of three nucleotides are translated to amino acids.

Input:

```
?-
  parse([leucine,tryptophan,phenyl
  alanine]).
```

Parser:

```
token(tryptophan)::> codon([u,u,g]).
token(leucine) ::>codon([u,u,a]).
token(leucine) ::>codon([u,u,c]).
token(phenylalanine)==> ::>codon([u,u,u]).
```

	A	C	G	U
AA |	Lys	Asn	Lys	Asn
AC |	Thr	Thr	Thr	Thr
AG |	Arg	Ser	Arg	Ser
AU |	Ile	Ile	MET	Ile
CA |	Gln	His	Gln	His
CC |	Pro	Pro	Pro	Pro
CG |	Arg	Arg	Arg	Arg
CU |	Leu	Leu	Leu	Leu
GA |	Glu	Asp	Glu	Asp
GC |	Ala	Ala	Ala	Ala
GG |	Gly	Gly	Gly	Gly
GU |	Val	Val	Val	Val
UA |	-	Tyr	-	Tyr
UC |	Ser	Ser	Ser	Ser
UG |	-	Cys	Trp	Cys
UU |	Leu	Phe	Leu	Phe

Detecting Tandem Repeats

```
 [X], string(Y) ::> string([X| Y]).
[X] ::>string([X]).

string(X), string(X)::> tandem_repeat(X).
```

<u>TEST</u>:

```
?-  parse([a,c,c,g,t,a,c,c,g,t]).

 tandem_repeat(0,10, [a,c,c,g,t]);

tandem_repeat(1,3,[c]);

tandem_repeat(6,8,[c])
```

CHRG for grammatical inference

- CHR/CHRG promote fairly direct materializations of constraint-based linguistic theories
- But: no general concensus of what they are (Shieber: common threads such as modularity, declarativeness, partial info), no stress on space reduction through narrowing variables' domains.
- To what extent do the "constraint-based" grammar models fit into de constraint solving model per se?

- Among the candidate models, Property Grammars stands out through its aim at complete reliance on constraints, which are:

Constraints in Property Grammar

Constituency	A : S	*children must have categories in the set S, e.g.* np: {det,noun,adj,name, sup}
Obligation	A : △B	*at least one B child, e.g.* vp : △verb
Uniqueness	A : B !	*at most one B child, e.g.* np: det !
Precedence	A : B <C	*B children precede C children, e.g.* np : det <noun
Requirement	A : B⇒C	*if B is a child, then also C is, e.g.* np: noun ⇒ get
Exclusion	A : B/⇔ C	*B and C children are mutually exclusive, e.g.* *np: noun/⇔name*
Dependency	A : B ~ C	the features of C1 and C2 are the same

The Womb Grammar Model of Grammatical Inference- the intuitive idea

In 2012 I was visited by a startling idea to help linguists keep up with the rate of grammar discovery needed in our Babel-ish world:

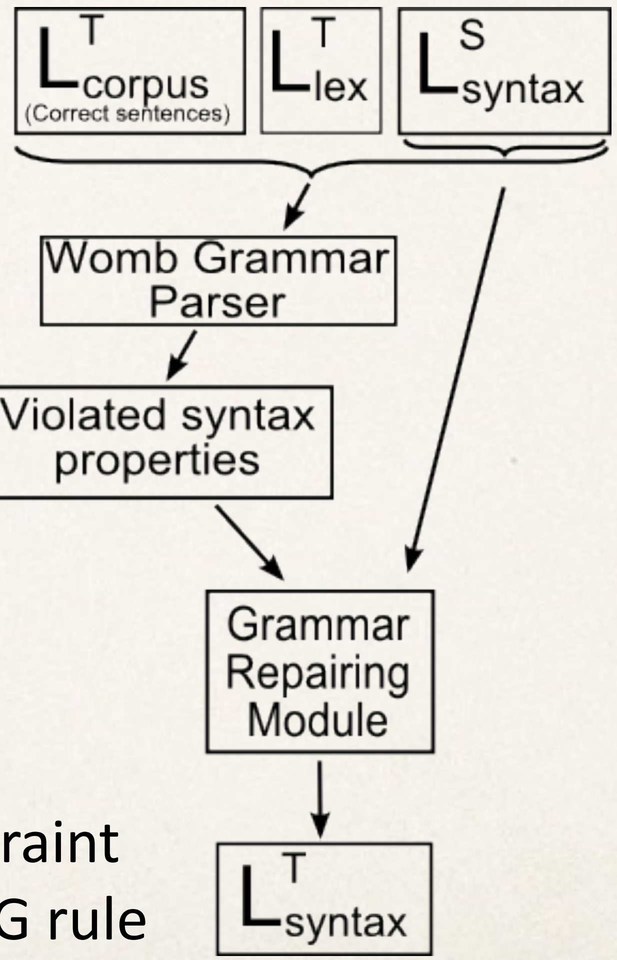
Just as human wombs can, given appropriate input, generate different races, could I devise a grammatical "womb" capable of mapping a known grammar into the grammar of a different language, given only a set of positive but representative input sentences of that language, plus its lexicon?

Womb Grammar Parsing- Hybrid Parser

- Combine target corpus and lexicon with
- source syntax

We obtain a list of violated properties

Compare source syntax with violated properties
to derive target syntax



E.g., if the source grammar contains the precedence constraint NP : N < ADJ but we get the input “the blue heron”, a CHRG rule applies to delete that precedence constraint. Thus *the constraint was checked for violation, not satisfaction*

Summary of CHRGs

- A powerful language specification language
- A powerful language processing system
- Exemplifies how you can use CHR to implement fairly advanced, knowledge-based systems
- A compile-on-load implementation technique, you can use for other purposes
- The power of CHRGs has not been explored fully; biological applications are under consideration

Summary of the tutorial

- Constraint Solving through CHR is for more than numbers, inequalities and stuff like that
- CHR is a ***powerful knowledge representation & manipulation language***
- We have shown methods for abductive and assumptive reasoning and language processing, that are
  - executed directly by the underlying CHR and Prolog systems
  - thus efficient for the right kind of problems
- We have intended that, after this course and a bit of reading, ***you can***
  - use the methods as described directly
  - invent your own ways to work with knowledge and experiment with in Prolog+CHR

Further Applications

- The Modeling Beauty of Constraint Solving (Dahl 16)
- Parsing as Semantically-Guided Constraint Satisfaction: the role of ontologies (Dahl et al 16)
- Using Womb Grammars for inducing the grammar of a subset of Yoruba sentences (Adebara 16)
- Shape Analysis as an aid to grammar induction (Adebara et al 15)
- Completing Mixed Language Grammars through Womb Grammars plus Ontologies (Adebara et al 15)
- On Second-Language Tutoring through Womb Grammars (Becerra-Bonache et al 13)
- The role of universal constraints on language acquisition ((Becerra-Bonache et al 13)
- Principle-Driven Decision Making (Dahl et al 2012)
- A dual processing scheme for both spoken and biological languages (Dahl & Maharshak 09)
- Decoding nucleic acid strings through spoken language (Dahl 10)
- An RNA-inspired analysis of poetry (Dahl, Perriquet, Jimenez-Lopez 2011)
- chrRNA (Bavarian and Dahl 06): a CHR+probability *method for RNA secondary structure design*

The End