Constraint Programming for Natural Language Processing

Denys Duchier

ESSLLI 2000

Abstract

This course demonstrates how constraint programming can be used effectively in practice, for linguistic applications. It shows how many forms of ambiguity arising in computational linguistic can be represented compactly and elegantly, and processed efficiently with constraints. A key idea to derive the most benefit from constraint propagation is that intended models should be characterized as solutions of *Constraint Satisfaction Problems* (CSPs) rather than defined inductively or in a generative fashion.

We examine several topics in detail: encodings of finite domains, tree descriptions using dominance constraints, and parsing with dependency grammars. In each case, we present a formal characterization of the problem as a CSP and illustrate how to derive a corresponding constraint program. The course includes 4 complete interactive applications written in Oz, with full code supplied.

Through these programmatic vignettes the reader is exposed to the practice of constraint programming with *finite domain* and *finite set* variables, and introduced to some of the more powerful types of constraints available today, such as reified constraints, disjunctive propagators, and selection constraints.

Universität des Saarlandes, Saarbrücken, Germany
duchier@ps.uni-sb.de

This course is also available online:

http://www.ps.uni-sb.de/~duchier/esslli-2000/index.html

Contents

1

Introduction

Constraints play an important role in the formulation of modern linguistic theories. For example HPSG and LFG are overtly constraint-based: they are primarily concerned with the formulation of general structural principles that determine the class of syntactically well-formed entities, typically represented as hierarchical, possibly typed, feature structures.

Yet, practical implementations are usually not driven but merely aided by constraints. For example implementations of HPSG typically only use constraints in the form of type hierarchies and unification, but the backbone remains generative (e.g. CFG).

The purpose of this course is to illustrate how constraints can be put into the driver's seat. We will shun merely constraint-aided approaches: these have been adequately presented by others in a Logic Programming setting. Instead we will focus on purely constraint-based approaches, where computation is reduced to constraint satisfaction.

While Prolog has proven to be a versatile implementation platform for computational linguistics, it encourages a view of computation as a non-deterministic generative process: typically a Prolog program specifies how to generate models. Regrettably, this often leads to difficult problems of combinatorial explosion.

In this course we wish to promote an alternative approach: replace model generation by model elimination. We will look at problems where the class of models is known before-hand and where constraints can be used to precisely characterize those which are admissible solutions: i.e. constraints serve to eliminate non-admissible candidates.

This course is by no means anti-Prolog: it is pro-constraints. Indeed, many of the techniques presented here are applicable to modern implementations of Prolog that support constraint libraries.

Some techniques, however, go beyond the capabilities of any Prolog system. This is primarily because Prolog does not support encapsulation or concurrency. In particular in does not support encapsulated speculative computations. Such notions where explored in AKL [**?**], in the form e.g. of deep guards, but only reached their full potential in Oz, in the form of computation spaces [**?**] [**?**]. We will use Oz as our programming vehicle [**?**] [**?**].

The computational paradigm explored in this course, namely concurrent constraint programming, is the result of a historical progression that granted constraints an increasingly important role in computations:

Generate and Test: generate a candidate model and then verify that it solves the problem. Constraints are used only as post-hoc filters. Combinatorial explosion is fearsomely uncontrolled.

Test and Generate: generation and filtering are interleaved. Constraints typically remain passive but are successful in pruning entire branches of the search. This technique is often based on coroutined constraints using `freeze` (aka `geler`) introduced by Prolog II.

Propagate and Distribute: constraints are active and perform as much deterministic inference as possible. The generative aspect is reduced to the minimum: the distribution strategy is invoked to make a choice only when propagation is not sufficient to resolve all ambiguities.

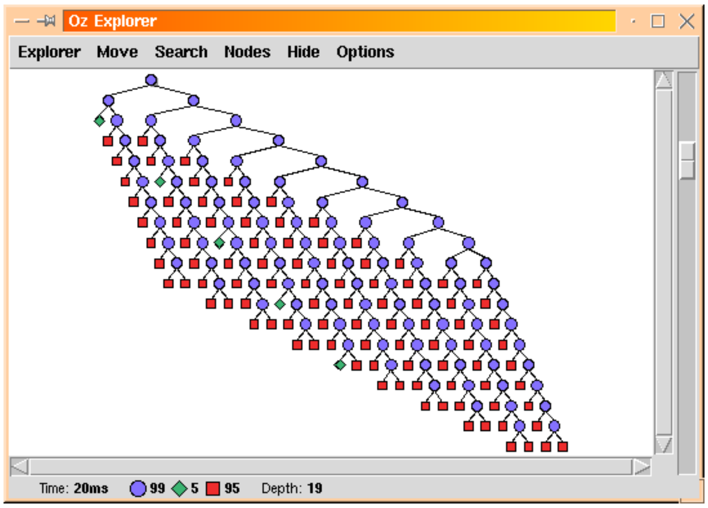
This progression results in increasingly smaller search trees. We can give an example that illustrates the improvements. Suppose that we are searching for solutions of the equation `2*A=B` where `A` and `B` denote integers between 0 and 9 inclusive. Let us first consider a *generate and test* solver written in Oz: it picks values for variables `A` and `B` and only then checks whether these values satisfy the equation:

2a ⟨**Equation Solver: generate and test** 2a⟩≡

```
proc {EquationSolver1 Solution}
   [A B] = Solution
in
   Solution:::0#9
   {FD.distribute naive Solution}
   2*A=B
end
```

What you see above is a *named code chunk*. This form of code presentation, in small named chunks that are individually explained in the text, was pioneered by Donald Knuth under the name of *Literate Programming*. We will use it constantly throughout this course.

The generate and test approach produces the search tree of size 199 shown below. Blue circles represent choice points, red squares failed nodes, and green diamonds solutions.

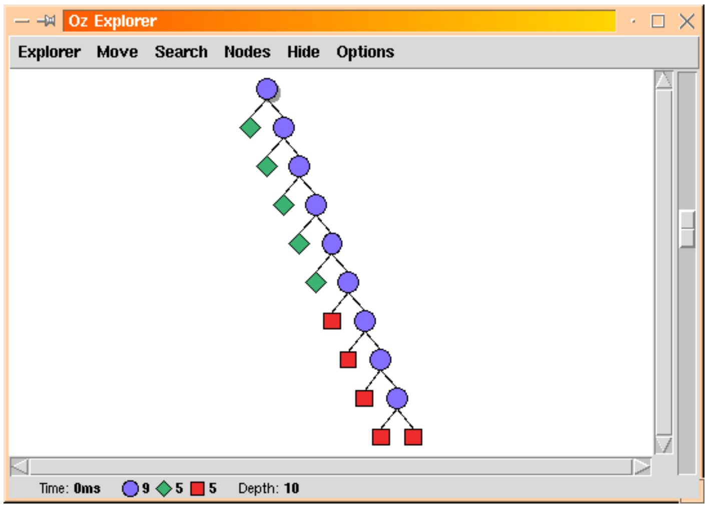


Now let's turn to the *test and generate* approach: the equation is posted in its own thread (i.e. the test is issued first) which suspends until a value has been chosen for `A`, at which the corresponding value for `B` can be computed.

3a ⟨**Equation Solver: test and generate** 3a⟩≡

```
proc {EquationSolver2 Solution}
   [A B] = Solution
in
   Solution:::0#9
   thread 2*A=B end
   {FD.distribute naive Solution}
end
```

The test and generate approach produces the search tree of size 19 shown below:

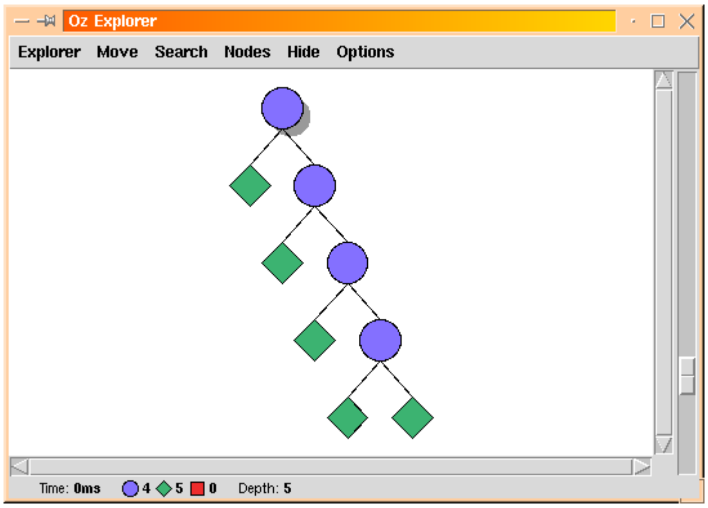


Finally, here is the real constraint programming way of solving the problem, using the *propagate and distribute* method. The equation is represented by a constraint:

3b ⟨**Equation Solver: propagate and distribute** 3b⟩≡

```
proc {EquationSolver3 Solution}
   [A B] = Solution
in
   Solution:::0#9
   2*A=:B
   {FD.distribute naive Solution}
end
```

The propagate and distribute approach produces the search tree of size 9 below. Note that this search tree contains no failed node.



When writing this course, my first priority was to demonstrate how constraints can be used effectively in practice, for linguistic applications. As a result, for each topic, in addition to the formal presentation, I also provide the corresponding code. In each case, you get a complete interactive application that you can run, and whose code you can study. For reasons of space, the printed version of the course does not contain absolutely all the code, but almost. The online version contains absolutely everything and is available at:

`http://www.ps.uni-sb.de/~duchier/esslli-2000/index.html`

The online version also allows you to start the applications by clicking on appropriate links. Since Oz code forms a large part of the diet in this course, the reader is urgently advised to become marginally familiar with Oz. We highly recommend that you read at least *The Oz Tutorial*[1] [**?**]. The Mozart system (implementing Oz) comes with a lot of online documentation[2], which is also available in Postscript or PDF[3] for printing.

[1] `http://www.mozart-oz.org/documentation/tutorial/index.html`
[2] `http://www.mozart-oz.org/documentation/`
[3] `http://www.mozart-oz.org/download/print.cgi`

2

Constraint Satisfaction Problems

Where constraint programming really shines is in solving constraint satisfaction problems where constraint propagation very effectively helps pruning the search space. In this chapter, we explain how constraint satisfaction problems (henceforth CSPs) are solved by means of constraint programming.

Much more is possible with constraint programming; in particular, using the record constraints (aka open feature structures) traditionally favored by computational linguists. However, record constraints do not lead to formulations with especially strong constraint propagation. For 2 reasons: (1) record constraints cannot represent negative information, (2) they only take effect when record structure appears (i.e. they are passive). We entirely shun them in this course and emphasize instead encodings using integers and sets of integers that provide very strong and efficient constraint propagation.

2.1 What is a CSP?

A CSP is specified by a finite number of variables and a finite number of constraints on these variables:

$$\text{CSP} \quad \equiv \quad \text{Variables} + \text{Constraints}$$

A solution to a CSP is an assignment of values to the variables such that the constraints are satisfied.

The constraint programming approach to solving a CSP rests of the following ideas:

- devise an explicit representation of the partial information known about the desired assignment of values to variables
- improve this information through constraint propagation

2.2 Constraint-based Representation of Assignments

The types of problems that constraint programming is really good at are all about integers. This is where constraints are able to provide strong and efficient propagation.

Constraints involving *Finite Domains* have become a fairly standard tool of the trade. More recently, constraints on *Finite Sets* (of integers) have appeared [**?**] [**?**] [**?**] and proven to be extremely expressive, especially for applications in computational linguistics [?] [?]. In this course, we will see many applications of set constraints.

Thus, we will consider CSPs involving integer variables (written I) and set variables (written S). The fundamental idea of the constraint programming approach is to explicitly represent the partial information about the sought assignment. For example, we may not precisely know the value to be assigned to I, but we may know that it must be either 2, 3, or 7. Thus we would write $I \in \{1, 2, 7\}$. The general form of the partial information about the assignment to an integer variable is thus:

$$I \in D$$

where D is a domain, i.e. a given finite set of integers.

Similarly, we may not know precisely which set value to assign to a set variable S, but we may have some information about its lower and upper bounds (i.e. the integers at least, resp. at most, in S). Thus we write:

$$D_1 \subseteq S \subseteq D_2$$

2.2.1 Basic Constraints

This intuitive way, presented above, of capturing partial information about an assignment can be formally presented as a logical system of *Basic Constraints*. Its abstract syntax is given by:

$$\mathcal{B} \quad ::= \quad I \in D \;\mid\; D \subseteq S \;\mid\; S \subseteq D \;\mid\; \textbf{false} \;\mid\; \mathcal{B}_1 \wedge \mathcal{B}_2$$

It is equipped with the following inference rules.

Weakening			
$I \in D$	$\rightarrow$	$I \in D'$	where $D' \supseteq D$
$D \subseteq S$	$\rightarrow$	$D' \subseteq S$	where $D' \subseteq D$
$S \subseteq D$	$\rightarrow$	$S \subseteq D'$	where $D' \supseteq D$
Strengthening			
$I \in D_1 \wedge I \in D_2$	$\rightarrow$	$I \in D_1 \cap D_2$	
$D_1 \subseteq S \wedge D_2 \subseteq S$	$\rightarrow$	$D_1 \cup D_2 \subseteq S$	
$S \subseteq D_1 \wedge S \subseteq D_2$	$\rightarrow$	$S \subseteq D_1 \cap D_2$	
Contradiction			
$I \in \emptyset$	$\rightarrow$	**false**	
$D_1 \subseteq S \wedge S \subseteq D_2$	$\rightarrow$	**false**	where $D_1 \not\subseteq D_2$

Of course, after saturation with the above rules, for each variable there is always a most specific approximation of its assignment. For an integer variable I, this is the smallest D such that $I \in D$. For a set variable S there is a largest lower bound $D_1 \subseteq S$ and a smallest upper bound $S \subseteq D_2$. In practice, these most specific basic constraints are the only ones that the system needs to represent (all others can be derived by weakening).

Constraint Store A constraint programming system contains a *constraint store* which is the (saturated) set of basic constraints representing the partial information currently known about the assignment.

store of basic constraints

$I \in \{1, 6, 7\}$
$\{1, 5\} \subseteq S$

Determined Variables We say that a variable is *determined* when its assignment has been decided. For an integer variable $I \in D$ this happens when its domain D becomes a singleton. For a set variable $D_1 \subseteq S \subseteq D_2$ this happens when its lower bound becomes equal to its upper bound.

2.3 Improving partial info by constraint propagation

Now we want the constraints of the CSP to take an active role and improve the partial information concerning the assignment. In particular, whenever possible, they should eliminate from consideration values that cannot lead to a solution.

2.3.1 Non-Basic Constraints

A constraint programming system typically provides a rich set of non-basic constraints, such as:

Equality	
$I_1 = I_2$ or $S_1 = S_2$	
Ordering, e.g.	
$I_1 < I_2$	
Arithmetic, e.g.	
$I_1 = 2 * I_2$	
Set, e.g.	
$S_1 \subseteq S_2$	subset
$S_1 \parallel S_2$	disjointness
$S_3 = S_1 \cup S_2$	union
Membership, e.g.	
$I \in S$	

and many more. The operational semantics of each non-basic constraint is specified by a collection of inference rules.

For example, the disjointness constraint $S_1 \parallel S_2$ corresponds to the two inference rules below:

$$\begin{array}{lcl} S_1 \parallel S_2 \wedge S_1 \subseteq D_1 \wedge D_2 \subseteq S_2 & \rightarrow & S_1 \subseteq D_1 \setminus D_2 \\ S_1 \parallel S_2 \wedge D_1 \subseteq S_1 \wedge S_2 \subseteq D_2 & \rightarrow & S_2 \subseteq D_2 \setminus D_1 \end{array}$$

i.e. all elements known to be in S_2 cannot possibly be in S_1 and vice versa.

A challenge that must be faced by every user of a constraint programming system is then to express a CSP's constraints in terms of non-basic constraints available in the system. Fortunately, Mozart has a very rich set of constraints (see [**?**]) that facilitate the task.

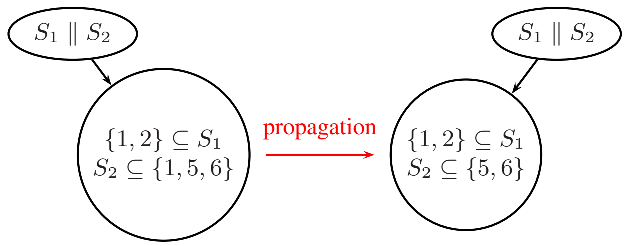
2.4 Searching For Solutions

We find solutions of a CSP (i.e. assignments satisfying the constraints) by alternating steps of *propagation* and *distribution*.

2.4.1 Propagation

Propagation is saturation under deterministic inference. The basic inference rules as well the inference rules corresponding to the non-basic constraints are applied repeatedly until a fix point is reached, i.e. until the partial information about the assignment can no longer be improved. At this point, there are 3 possibilities:

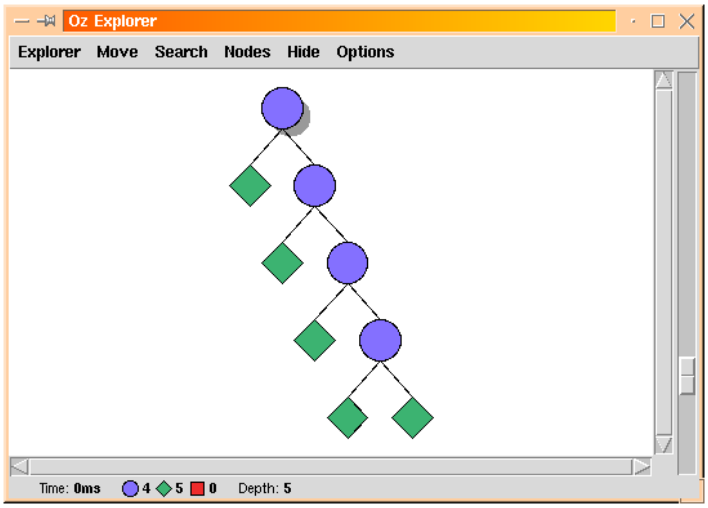
1. **false** was derived, i.e. we have arrived at a contradiction and this branch of the search must be abandoned since it cannot lead to a solution.
2. All variables are determined: the assignment is fully known and satisfies the constraints. This is a solution.
3. At least one variable is not yet determined. At this point we need search and non-deterministically apply a distribution rule.



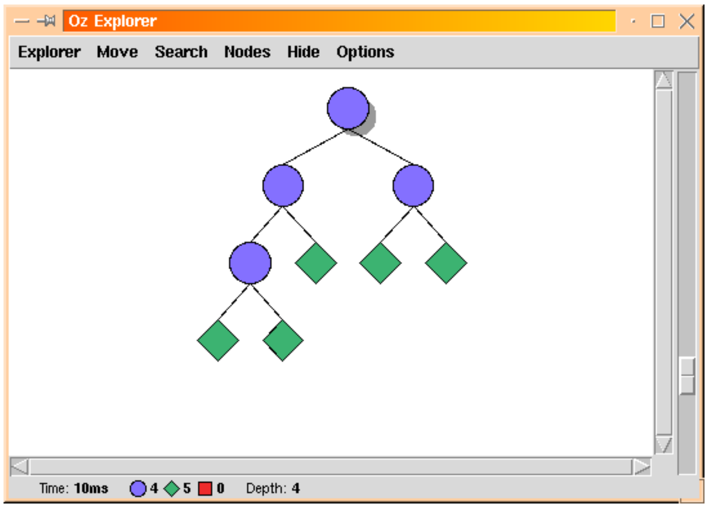
2.4.2 Distribution

The purpose of a distribution step is to allow the search to proceed by making a non-deterministic choice.

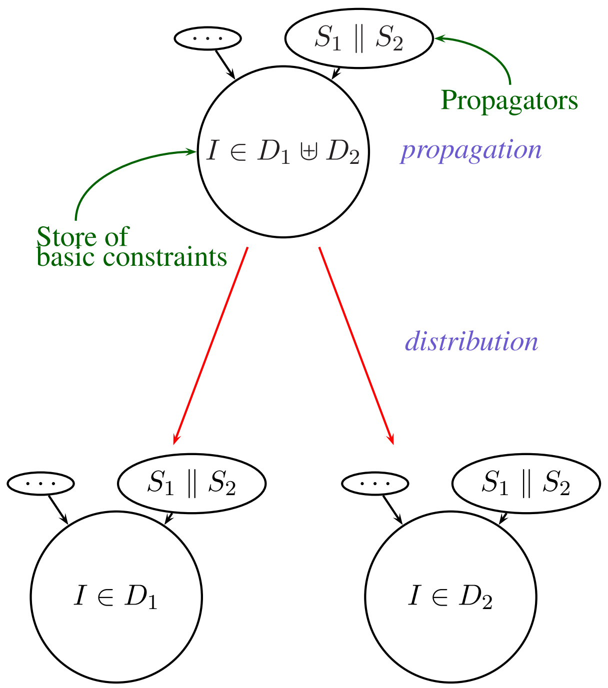
Naive Strategy The naive distribution strategy for integer variables picks a non-determined $I \in D$ and non-deterministically infers either $I = \min(D)$ or $I \neq \min(D)$. For finding solutions of `2*A=B` with `A` and `B` integers between 0 and 9 inclusive, the strategy produces the search tree below:



Domain Splitting Another strategy for integer variables is known as *domain splitting*. It picks a non-determined integer variable $I \in D_1 \uplus D_2$, where $D_1, D_2 \neq \emptyset$ and non-deterministically infers either $I \in D_1$ or $I \in D_2$. For the same problem `2*A=B` this strategy produces the search tree below:



Alternating Steps of Propagation and Distribution The picture below shows a propagation step followed by a distribution step that creates a branching point in the search tree. The distribution strategy used is *domain splitting* on variable I: each branch investigates one half of I's domain:


$S_1 \parallel S_2$
Propagators
$I \in D_1 \uplus D_2$
propagation
Store of
basic constraints
distribution
$S_1 \parallel S_2$
$S_1 \parallel S_2$
$I \in D_1$
$I \in D_2$

3

Efficient Encodings of Finite Domains

Ambiguity (lexical ambiguity, attachment ambiguity) is a major source of combinatorial complexity. It is typically addressed by resorting to some form of disjunctive representation. In this chapter, we are going to explain how constraint programming using *finite domains* allows the compact representation and efficient processing of common types of disjunctive information.

3.1 Finite Domain Constraints

Finite domain (FD) constraints have become quite popular and are widely supported. A finite domain variable I denotes an integer. However, we may not know exactly which integer. Typically we only know that it must be one of a finite number of possibilities, e.g. one of 1,2 or 7. This would be represented by a basic constraint:

$$I \in \{1, 2, 7\}$$

Suppose, we have a second FD variable $J \in \{2, 3, 4\}$. If we unify them $I = J$, their domains are intersected thus resulting in the conclusion $I = J = 2$. This technique underlies the efficient treatment of agreement: if I encodes the agreement information on one word and J on another and the 2 words must agree, then the constraint $I = J$ enforces this agreement.

3.2 Simple Example

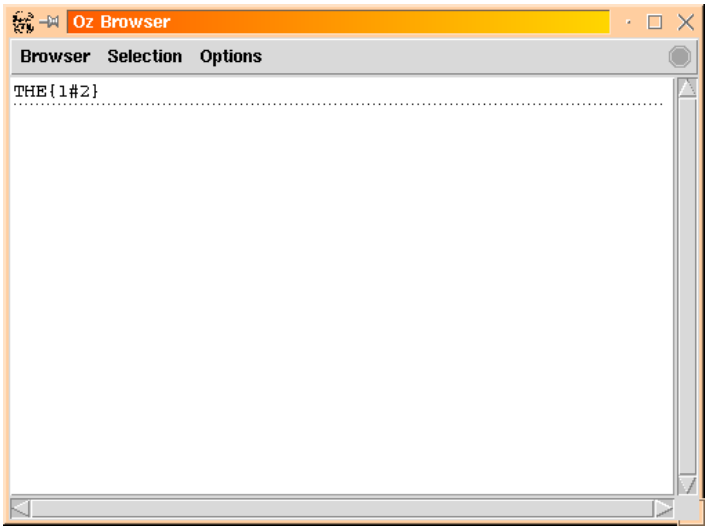
As a simple example, lets consider a minuscule English lexicon consisting of the words: `a`, `the`, `man`, `men` and just the information about whether the word is singular or plural. We are going to encode 'singular' as the integer 1 and 'plural' as the integer 2.

```
declare
Sing = 1
Plur = 2
Lexicon = o(a   : [Sing]
            the : [Sing Plur]
            man : [Sing]
            men : [Plur])
```

Now, we can represent the 'number' information of a word `W` by introducing a FD variable `I` and constraining its domain using the information in the lexicon. For example for an occurrence of article `the`, we could write:

```
declare THE
THE::Lexicon.the
```

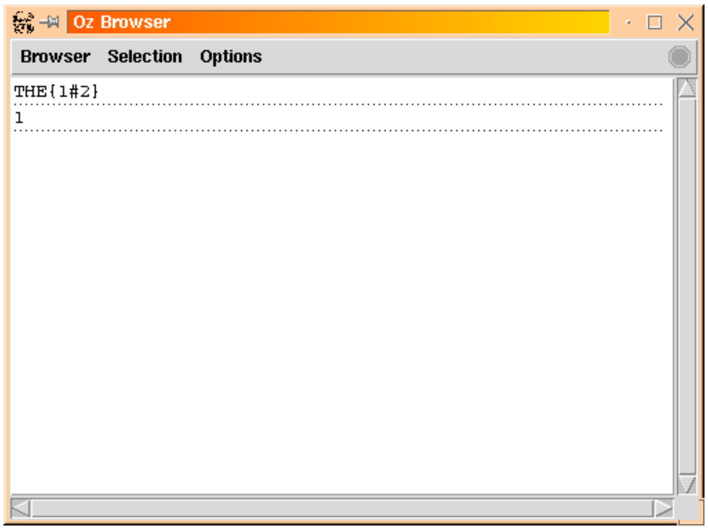
We can observe this variable in the browser [?] by invoking `{Browse THE}`, and we see:



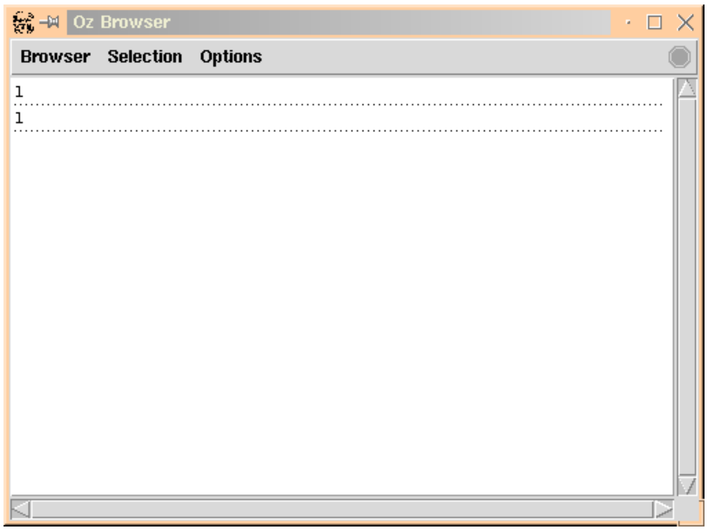
I.e. `THE` is bound to a FD variable whose domain just contains the integers 1 and 2. We can do the same for an occurrence of the word `man`:

```
declare MAN
MAN::Lexicon.man
{Browse MAN}
```

and we see:



Finally we can impose the constraint that the determiner must agree with the noun, i.e. `THE=MAN`, and the display is updated as follows:



The ambiguous number information of the determiner has been disambiguated.

3.3 Encoding Products of Finite Domains

Our simple example hardly touched on the power of the technique: it was not sufficiently ambiguous. In a real application, agreement involves several features and not just 'number'. For example, German agreement involves 5 features:

```
gender     -> masculine,feminine,neutral
number     -> singular,plural
person     -> 1,2,3
case       -> nominative,accusative,dative,genitive
quantifier -> definite,indefinite,none
```

However these features do not vary independently. For example the determiner `ein` is `singular`, but may be either `masculine` or `neutral`. If it is `masculine`, then it has `nominative` case. If it is `neutral` then it may have case `nominative` or `accusative`.

An elegant way to address the issue is, instead of insisting that the program preserve the distinction of features,[1] to merge them together into a compound 'agreement' tuple that takes values in the cartesian product:

```
gender * number * person * case * quantifier
```

Next we notice that, since each dimension of the cartesian product can take only finitely many values, the cartesian product itself has finitely many values. This means that we can encode each tuple by a distinct integer and we can represent a disjunction of tuples by means of a finite domain.

[1] actually we can recover this distinction easily as explained in Section 3.6.

3.3.1 Establishing a bijection between tuples and integers

Consider p domains D_1 through D_p. Each domain D_ℓ has cardinality n_ℓ:

$$D_\ell = \{v_1^\ell, \ldots, v_{n_\ell}^\ell\}$$

The cardinality of the cartesian product $D_1 \times \cdots \times D_p$ is $N = n_1 \times \cdots \times n_p$. We are going to establish a bijection between $D_1 \times \cdots \times D_p$ and $[1..N]$.

The idea is to partition the interval $[1..N]$ into n_1 equal subintervals: one for each value in D_1. Then to partition each subinterval into n_2 subsubintervals: one for each value in D_2. Etc recursively.

It is easy to capture this idea in a formula. Consider a tuple $(v_{i_1}^1, v_{i_2}^2, \ldots, v_{i_p}^p)$. According to the recursive algorithm outlined above, it is assigned to the following integer:

$$\begin{array}{ll} & (i_1 - 1) \times (n_2 \times n_3 \times \cdots \times n_p) \\ + & (i_2 - 1) \times (n_3 \times \cdots \times n_p) \\ \vdots & \\ + & (i_{p-1} - 1) \times n_p \\ + & (i_p - 1) \\ + & 1 \end{array}$$

Thus, given an index I in the range $[1..N]$, we can recover the corresponding tuple by calling `{DecodeInt I-1 Divs Doms}`, where `Divs` is the list:

$$\begin{array}{r} [n_2 \times n_3 \times \cdots \times n_p \\ n_3 \times \cdots \times n_p \\ \vdots \\ n_p \\ 1] \end{array}$$

and `Doms` is the list of domains, each one consisting of a list of values. The function `DecodeInt` is implemented as follows:

14a ⟨**DomainProduct DecodeInt function** 14a⟩≡
```
fun {DecodeInt I Divs Doms}
   case Divs#Doms
   of nil#nil then nil
   [] (Div|Divs)#(Dom|Doms) then
      Q = I div Div
      R = I mod Div
   in
      {Nth Dom Q+1}|{DecodeInt R Divs Doms}
   end
end
```

3.3.2 Implementation

In this section, we provide an Oz implementation of the encoding technique described above. It is provided as class `DomainProduct` exported by functor `Encode` (file `Encode.oz`).

15a ⟨**Encode.oz** 15a⟩≡
```
functor
import FD FS
export DomainProduct
define
   ⟨DomainProduct DecodeInt function 14a⟩
   class DomainProduct
      ⟨DomainProduct features 15b⟩
      ⟨DomainProduct attributes 15c⟩
      ⟨DomainProduct init method 15d⟩
      ⟨DomainProduct encode method 17a⟩
      ⟨DomainProduct decode method 17b⟩
   end
end
```

3.3.2.1 Features and Attributes

Every `DomainProduct` object is equipped with feature `range` giving the range $[1..N]$, feature `empty` for the empty set, and feature `full` for the set of all the integers in the range:

15b ⟨**DomainProduct features** 15b⟩≡ 18a▷
```
feat range empty full
```

and also with attributes `divisors` and `domains` which are as described for function `DecodeInts`. Attribute `value2set` is a dictionary mapping each value v_i^ℓ, from some domain D_ℓ, to the set of integers encoding the tuples in which this value occurs.

15c ⟨**DomainProduct attributes** 15c⟩≡
```
attr divisors domains value2set
```

3.3.2.2 Init Method

The `init` method is given a list of domains. Each domain is represented by the list of its values (these should be atoms or integers). A `DomainProduct` object is created as follows:

```
declare O = {New DomainProduct init([D1 D2 ... Dp])}
```

where each `Di` is a list of values representing a domain.

The initialization code constructs the map `value2set` by iterating through the integers in the range $[1..N]$. For each `I` in this range, `DecodeInt` is invoked to obtain the corresponding tuple (as a list of values). For each value `V` in this tuple, `I` is added to the list of indices for `V`. When we are done, these lists of indices are turned into sets of indices.

15d ⟨**DomainProduct init method** 15d⟩≡

```
meth init(Domains)
   Sizes = {Map Domains Length}
   L1    = {Map Sizes fun {$ Size} _#Size end}
   N     = {FoldR L1 fun {$ M#N Accu} M=Accu N*Accu end 1}
   Divs  = {Map L1 fun {$ M#_} M end}
   Val2Ints = {Dictionary.new}
in
   for I in 1..N do
      Tuple = {DecodeInt I-1 Divs Domains}
   in
      for V in Tuple do
         {Dictionary.put Val2Ints V
          I|{Dictionary.condGet Val2Ints V nil}}
      end
   end
   divisors <- Divs
   domains  <- Domains
   ⟨DomainProduct init method, niceties 18b⟩
   for K in {Dictionary.keys Val2Ints} do
      Val2Ints.K := {FS.value.make Val2Ints.K}
   end
   self.range = 1#N
   self.empty = FS.value.empty
   self.full  = {FS.value.make self.range}
   value2set <- Val2Ints
end
```

3.3.2.3 Encode Method

One we have created a `DomainProduct` object `O`, we can use it to turn a *specification* into a set of integers encoding the tuples corresponding to this specification. We want to allow the user to write specifications with the following abstract syntax:

$$\phi \quad ::= \quad v_i^\ell \mid \phi \wedge \phi' \mid \phi \vee \phi'$$

A tuple satisfies specification v_i^ℓ if it contains value v_i^ℓ. It satisfies $\phi \wedge \phi'$ if it satisfies both ϕ and ϕ'. It satisfies $\phi \vee \phi'$ if it satisfies either ϕ or ϕ'.

Instead of explicit connectives, we will simply allow a specification to consist of arbitrarily nested lists, eventually bottoming out with domain values. The outer level is interpreted disjunctively, and each nesting switches the interpretation of the connective: thus the 2nd level is interpreted conjunctively, the 3rd disjunctively, etc. For example, let us consider agreement information limited to just gender and person. The specification:

```
[[masc [1 3]] [fem 2]]
```

denotes the 3 tuples `[masc 1]`, `[masc 3]`, `[fem 2]`. However:

```
[[masc 1 3] [fem 2]]
```

just denotes `[fem 2]` since a tuple cannot contain both `1` and `2`. The spec:

```
[[masc 1] fem]
```

denotes the 4 tuples `[masc 1]`, `[fem 1]`, `[fem 2]` and `[fem 3]`.

17a ⟨**DomainProduct encode method** 17a⟩≡

```
meth encode(Desc $)
   {self Disj(Desc $)}
end
meth Disj(Desc $)
   case Desc
   of _|_ then {FoldL Desc
                 fun {$ Accu Desc}
                    {FS.union Accu
                     {self Conj(Desc $)}}
                 end self.empty}
   [] nil then self.empty
   else @value2set.Desc end
end
meth Conj(Desc $)
   case Desc
   of _|_ then {FoldL Desc
                 fun {$ Accu Desc}
                    {FS.intersect Accu
                     {self Disj(Desc $)}}
                 end self.full}
   [] nil then self.full
   else @value2set.Desc end
end
```

3.3.2.4 Decode Method

A `DomainProduct` object `O` makes available 3 main decoding methods:

`{O decode(I L)}`

returns the list `L` of tuples corresponding to the integers in the domain of FD variable `I`.

`{O decodeLo(S L)}`

return the list `L` of tuples corresponding to the integers in the lower bound of FS variable `S`.

`{O decodeHi(S L)}`

idem for the upper bound.

17b ⟨**DomainProduct decode method** 17b⟩≡

```
meth decodeInt(I $)
   {DecodeInt I-1 @divisors @domains}
end
meth decodeInts(L $)
   {Map L fun {$ I} {self decodeInt(I $)} end}
end
meth decode(I $)
   {self decodeInts({FD.reflect.domList I} $)}
end
meth decodeLo(S $)
   {self decodeInts({FS.reflect.lowerBoundList S} $)}
end
meth decodeHi(S $)
   {self decodeInts({FS.reflect.upperBoundList S} $)}
end
```

3.3.2.5 Niceties

Note that we can also use our abstraction in the degenerate case where we compute the product of just 1 domain. In that case, however, there is a bijection between the elements of the domain and the integers 1 to n of the encoding. It turns out to be often convenient to be able to map an element of the domain to its corresponding integer rather than to encode it into the singleton containing that integer. For this reason, we add a feature `toint` to the class:

18a ⟨**DomainProduct features** 15b⟩+≡ ◁15b

```
toint
```

For products of 2 or more domains, this feature is not used and is simply set to **`unit`**, but for a 1-product it is a dictionary mapping each domain element to its corresponding integer. Here is how we initialize the feature:

18b ⟨**DomainProduct init method, niceties** 18b⟩≡

```
case Domains of [Dom] then
   ToInt = {NewDictionary}
in
   self.toint = ToInt
   for K in Dom do
      case Val2Ints.K of [I] then
         ToInt.K := I
      end
   end
else self.toint=unit end
```

3.4 Application to German Agreement

In this section we illustrate the technique with an application to agreement of determiner/adjective/noun in German. Agreement depends of course on gender, number and person, but also on case, and on the determiner type (definite, indefinite, none). This leads us to define agreement information as a tuple in the cartesian product:

```
Gender * Number * Person * Case * Quantifier
```

We are going to develop a very small lexicon mapping words to sets of tuples. Each tuple will be encoded by an integer.

3.4.1 Lexicon

Our small lexicon is made available as functor `SmallLexicon` (file `SmallLexicon.oz`), exporting `Get` and `Agreement`. `Get` takes a word as an argument and returns a finite domain variable whose domain consists of the agreement tuples for that word (or more precisely, the integers encoding those tuples). `Agreement` is a `DomainProduct` object for the cartesian product of agreement information.

19a ⟨**SmallLexicon.oz** 19a⟩≡

```
functor
import Encode FS
export Get Agreement
define
   Agreement = {New Encode.domainProduct
                init([[masc fem neut]
                      [sing plur]
                      [1 2 3]
                      [nom acc dat gen]
                      [def indef none]])}

   Lexicon = {Dictionary.new}

   proc {Enter W Desc}
      Lexicon.W := {Agreement encode(Desc $)}
   end

   proc {Get W I}
      I::Agreement.range
      {FS.include I Lexicon.W}
   end

   ⟨SmallLexicon nouns 19b⟩
   ⟨SmallLexicon definite articles 20a⟩
   ⟨SmallLexicon indefinite articles 20b⟩
   ⟨SmallLexicon no article 20c⟩
   ⟨SmallLexicon adjectives 20d⟩
end
```

We just enter the 4 forms of *Mann* in the lexicon:

19b ⟨**SmallLexicon nouns** 19b⟩≡

```
{Enter mann     [[masc 3 sing [nom acc dat]]]}
{Enter mannes   [[masc 3 sing gen]]}
{Enter männer   [[masc 3 plur [nom acc gen]]]}
{Enter männern  [[masc 3 plur dat]]}
```

Now all forms of the definite article *der/die/das*:

20a ⟨**SmallLexicon definite articles** 20a⟩≡

```
{Enter der [[def [[masc sing 3 nom]
                  [fem  sing 3 [dat gen]]
                  [     plur 3 gen]]]]}
{Enter den [[def [[masc sing 3 acc]
                  [     plur 3 dat]]]]}
{Enter dem [[def [[[masc neut] sing 3 dat]]]]}
{Enter des [[def [[[masc neut] sing 3 gen]]]]}
{Enter die [[def [[fem sing 3 [nom acc]]
                  [    plur 3 [nom acc]]]]]}
{Enter das [[def [[neut sing 3 [nom acc]]]]]}
```

All forms of the indefinite article *ein*:

20b ⟨**SmallLexicon indefinite articles** 20b⟩≡

```
{Enter ein [[indef [[masc sing 3 nom]
                    [neut sing 3 [nom acc]]]]]}
{Enter einen [[indef [[masc sing 3 acc]]]]}
{Enter einem [[indef [[[masc neut] sing 3 dat]]]]}
{Enter eines [[indef [[[masc neut] sing 3 gen]]]]}
{Enter eine  [[indef [[fem sing 3 [nom acc]]]]]}
{Enter einer [[indef [[fem sing 3 [dat gen]]]]]}
```

A dummy entry for the absence of article:

20c ⟨**SmallLexicon no article** 20c⟩≡

```
{Enter '*no determiner*' none}
```

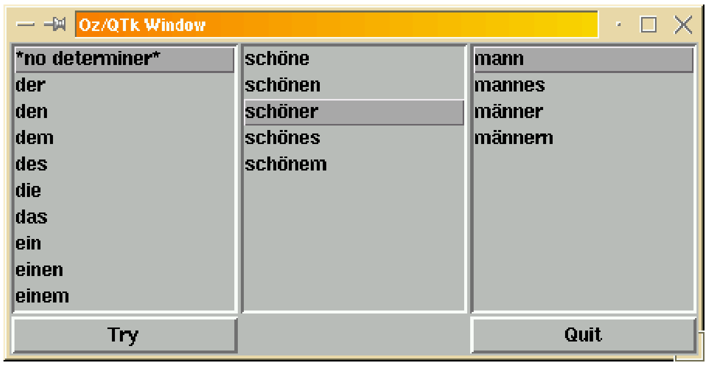
And all forms of adjective *schön*:

20d ⟨**SmallLexicon adjectives** 20d⟩≡

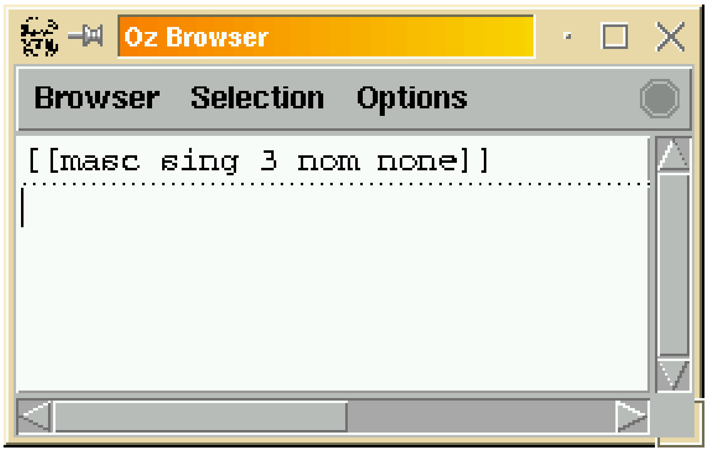
```
{Enter schöne  [[none  [nom acc] [fem plur]]
                [def   sing [nom [acc [neut fem]]]]
                [indef plur [nom acc]]]}
{Enter schönen [[none  [[masc sing [acc gen]]
                        [fem  sing gen]
                        [plur dat]]]
                [def   [plur dat gen [masc sing acc]]]
                [indef sing [dat gen [masc acc]]]]}
{Enter schöner [[none  [[masc sing nom]
                        [fem sing [dat gen]]
                        [plur gen]]]
                [indef sing masc nom]]}
{Enter schönes [[neut sing [nom acc] [indef none]]]}
{Enter schönem [[[masc neut] sing dat none]]}
```

3.5 Application

In this section, we include a small interactive application[2] to allow you to test our implementation of agreement. A window will appear as shown below, and you must select one entry in each column, i.e. one article (or none, represented by the `*no determiner*` entry), one adjective, and one noun:



Then you click on the `Try` button and a browser window pops up listing the possible tuples for this agreement.



If no agreement is possible, the atom `error` is displayed instead.

3.5.1 Implementation

The implementation is provided as functor `SmallTest` (file `SmallTest.oz`).

21a ⟨**SmallTest.oz** 21a⟩≡

```
functor
import
   QTk at 'http://www.info.ucl.ac.be/people/ned/qtk/QTk.ozf'
   Application
   SmallLexicon
   Browser(browse:Browse)
```

[2] http://www.ps.uni-sb.de/~duchier/esslli-2000/SmallTest.oza

```
define
   DetBox AdjBox NounBox
   DetWords = ['*no determiner*'
               der den dem des die das
               ein einen einem eines eine einer]
   AdjWords = [schöne schönen schöner schönes schönem]
   NounWords= [mann mannes männer männern]
   proc {Try}
      {Browse
       try
          D = {Nth DetWords  {DetBox get(firstselection:$)}}
          A = {Nth AdjWords  {AdjBox get(firstselection:$)}}
          N = {Nth NounWords {NounBox get(firstselection:$)}}
       in
          {SmallLexicon.agreement
           decode(
              {SmallLexicon.get D}=
              {SmallLexicon.get A}=
              {SmallLexicon.get N} $)}
       catch _ then error end}
   end
   Interface =
   lr(listbox(init:DetWords  glue:we exportselection:false handle:DetBox)
      listbox(init:AdjWords  glue:we exportselection:false handle:AdjBox)
      listbox(init:NounWords glue:we exportselection:false handle:NounBox)
      newline
      button(text:'Try' glue:we action:Try)
      empty
      button(text:'Quit' glue:we
             action:proc {$} {Application.exit 0} end))
   Window = {QTk.build Interface}
   {Window show}
end
```

3.6 Projections

Consider now the problem of agreement between noun and relative pronoun: they must agree in gender and number, but nothing else. How can we express such agreement condition, when gender and number have been combined with other features into one `DomainProduct`? What we need is to project a `DomainProduct` onto a subset of its dimensions. In this section, we illustrate how this can be achieved using the *selection* constraint.

The idea is that to each tuple `[G N P C Q]` in the cartesian product:

```
Gender * Number * Person * Case * Quantifier
```

we can associate a corresponding tuple `[G N]` in the projection product `Gender * Number`. In other words, there is a finite map from the integers encoding the 1st product into the integers encoding the 2nd product.

3.6.1 Selection Constraint

In this section, we very briefly introduce the idea of the selection constraint for finite domains. It has the form:

```
I={Select.fd [I1 I2 ... In] K}
```

where `I`, `I1`, ..., `In`, `K` are all FD variables (possibly determined, i.e. integers). Its declarative semantics is that `I` $=$ $\mathtt{I}_{\mathtt{K}}$. Constraint propagation can affect both `I` and `K`: if `Ip` cannot be equal to `I` (i.e. their domains are disjoint), then `p` is removed from the domain of `K`. Furthermore, the domain of `I` must be a subset of the union of the domains of `Ip` for `p` in the domain of `K`. To learn more about the selection constraint, see Section 6.9 and also the treatment of dependency parsing in Chapter 5.

3.6.2 Partial Agreement

Consider now the selection constraint:

```
ProjectedAgreement={Select.fd [I1 I2 ... In] Agreement}
```

when `p` encodes agreement tuple `[G N P C Q]` and `Ip` encodes the projected tuple `[G N]`. The constraint above precisely implements the mapping from agreement tuples to projected tuples.

We make this particular projection facility available in functor `SmallPartial` (file `SmallPartial.oz`), which exports `PartialAgreement` (the projected product involving only gender and number) and `GetPartialAgreement` which is a function taking 2 input arguments that must partially agree and returning said partial agreement.

23a ⟨**SmallPartial.oz** 23a⟩≡

```
functor
import
   Select at 'x-ozlib://duchier/cp/Select.ozf'
   SmallLexicon
   Encode FS
export
   PartialAgreement
   GetPartialAgreement
define
   PartialAgreement =
   {New Encode.domainProduct
    init([[masc fem neut]
          [sing plur]])}
   1#N = SmallLexicon.agreement.range
   Projection = {Tuple.make o N}
   for I in 1..N do
      case {SmallLexicon.agreement decode(I $)}
      of [[Gender Number _ _ _]] then
         S = {PartialAgreement encode([[Gender Number]] $)}
         [J] = {FS.reflect.lowerBoundList S}
```

```
          in
             Projection.I = J
          end
       end
       proc {GetPartialAgreement A1 A2 P}
          P = {Select.fd Projection A1}
          P = {Select.fd Projection A2}
       end
    end
```

3.6.3 Application

Now we provide an application[3] similar to the previous one, but where only partial agreement is required (file SmallPartialTest.oz).

24a ⟨SmallPartialTest.oz 24a⟩≡

```
functor
import
   QTk at 'http://www.info.ucl.ac.be/people/ned/qtk/QTk.ozf'
   Application
   SmallLexicon
   SmallPartial
   Browser(browse:Browse)
define
   Box1 Box2
   Words1 = [der den dem des die das]
   Words2 = [mann mannes männer männern
             schöne schönen schöner schönes schönem]
   proc {Try}
      {Browse
       try
          W1 = {Nth Words1 {Box1 get(firstselection:$)}}
          W2 = {Nth Words2 {Box2 get(firstselection:$)}}
       in
          {SmallPartial.partialAgreement
           decode(
              {SmallPartial.getPartialAgreement
               {SmallLexicon.get W1}
               {SmallLexicon.get W2}} $)}
       catch _ then error end}
   end
   Interface =
   lr(listbox(init:Words1 glue:we exportselection:false handle:Box1)
      listbox(init:Words2 glue:we exportselection:false handle:Box2)
      newline
      button(text:'Try' glue:we action:Try)
      button(text:'Quit' glue:we
             action:proc {$} {Application.exit 0} end))
```

[3] http://www.ps.uni-sb.de/~duchier/esslli-2000/SmallPartialTest.oza

```
   Window = {QTk.build Interface}
   {Window show}
end
```

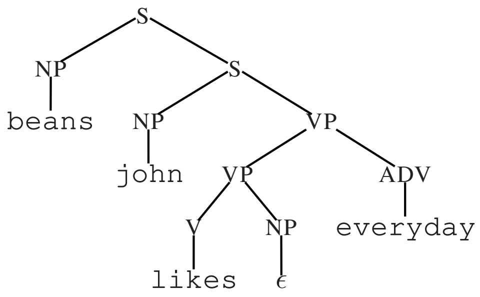
4

Dominance Constraints

Trees are widely used for representing hierarchically organized information, such as syntax trees, first-order terms, or formulae representing meanings. A tree can be regarded as a particular type of directed graph. A directed graph is a pair (V, E) where V is a set of vertices and E a multiset of directed edges between them, i.e. a subset of $V \times V$. A forest is an acyclic graph where all vertices have in-degree at most 1. A tree is a forest where there is precisely one vertex, called the root, with in-degree 0; all others have in-degree 1.

First-order terms, or finite constructor trees, are characterized by a further restriction: each node is labeled by some constructor f of a signature Σ, and its out-edges are labeled by integers from 1 to n, where n is the arity of the constructor. This can be formalized as follows: we assume a signature Σ of function symbols $f, g, \ldots$, each equipped with an arity $\mathsf{ar}(f) \geq 0$. A finite constructor tree is then a triple (V, E, L) where (V, E) defines a finite tree, and $L : V \to \Sigma$ and $L : E \to \mathbb{N}$ are labelings, and such that for any vertex $v \in V$ there is exactly one outgoing edge with label k for each $1 \leq k \leq \mathsf{ar}(L(v))$ and no other outgoing edges.

Trees are often used to represent syntactic structure. For example, here is a possible analysis of *"beans, John likes everyday"*.



or logical formulae representing meanings. For example, here are the simplified representations of the two meanings of the ambiguous sentence *"every yogi has a guru"*:

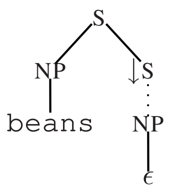
4.1 Descriptions of Trees

Since trees are a basic food group in the computational linguist's diet, it is often very convenient to be able to describe classes of trees with certain general structural properties. As mentioned earlier, a tree is typically specified extensionally by listing its edges, i.e. by providing a relation of immediate dominance between vertices. This is often too specific, hence the idea of also permitting the more relaxed relation of (general) dominance, which is simply the transitive closure of immediate dominance.

This makes it possible to express that some vertex v_1 should dominate another vertex v_2, which we write $v_1 \triangleleft^* v_2$, without having to commit to any specific detail. This merely requires that v_2 must occur below v_1, and allows arbitrary material to be inserted on the path between them. In other words, math/v_1\LE v_2/ is satisfied in every model (tree) in which v_1 denotes a proper ancestor of v_2.

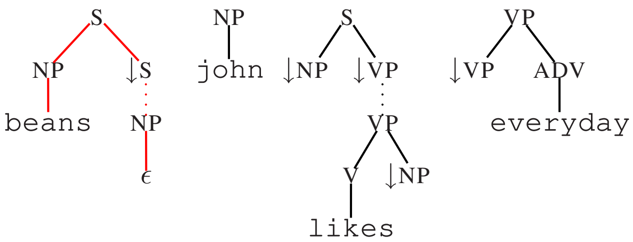
4.1.1 DTree Grammar

One application of tree descriptions is in the class of grammars which here we will, a bit loosely, generically call DTree grammars (DTGs) [**?**] [**?**] [**?**] [**?**]. Where Lexicalized Tree Adjoining Grammars (LTAGs) require the non-monotonic operation of adjunction, DTGs instead support the monotonic insertion into *holes* provided by dominance. For example, here is an elementary tree description for a topicalized NP:

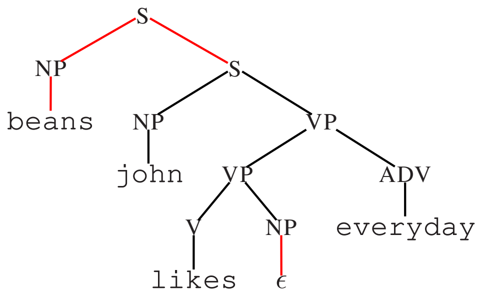


Note how the description introduces a dominance *hole* (graphically represented by a dotted line) between a sentence node S and a trace NP to be positioned somewhere below (i.e. to stand for the missing NP that was topicalized).

One main attraction of tree descriptions, is that it should be possible to put together several descriptions and then look for solution trees that simultaneously satisfy all of them. Parsing is one application: we would like to select in the lexicon one description for each word and combine them into a syntax tree. Consider our example sentence *"beans, John likes everyday"* and the following descriptions for each word:



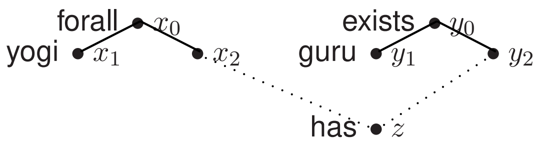
A corresponding solution tree is shown below where the parts originating with the description for the topicalized NP *"beans"* are shown in red.



4.1.2 Underspecified Semantic Representation

Semantic representations are typically expressed in the form of higher-order logical formulae [**?**] [**?**]. The semantic representation of a phrase is built-up from the semantic contributions of its parts. One advantage of tree descriptions in this context is that they permit a certain amount of underspecification sufficient to account for scope ambiguity [**?**].

Consider again the sentence *"every yogi has a guru"*. It is ambiguous in the scope of the existential quantifier associated with *guru*. If this quantifier takes wide scope, then all yogis have the same guru. If it takes narrow scope, different yogis may have different gurus. This ambiguity can be succinctly expressed with the description shown below:



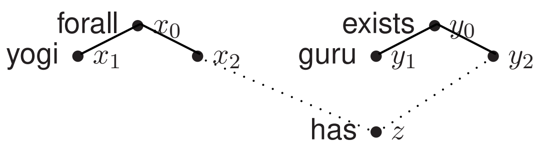
4.2 Dominance Constraints

We now present a formal language with which we can write tree descriptions. This is the language of *Dominance Constraints with Set Operators* as described in [**?**]. It has the following abstract syntax:

$$\phi \quad ::= \quad x\,R\,y \;\mid\; x : f(x_1 \ldots x_n) \;\mid\; \phi \wedge \phi'$$

where variables x, y denote nodes and $R \subseteq \{=, \lhd^+, \rhd^+, \perp\}$. $\lhd^+$ represents proper dominance and $\perp$ disjointness. The constraint $x\,R\,y$ is satisfied when the relationship that holds between x and y is one in R. Thus $x\,\{=, \perp\}\,y$ is satisfied either when x and y are equal, or when they denote nodes in disjoint subtrees. We write $x \lhd^* y$ instead of $x\,\{=, \lhd^+\}\,y$ and generally omit braces when we can write a single symbol.

Consider again the scope ambiguity example:



It can be expressed as the following dominance constraint:

$$\begin{array}{ll} & x_0 : \mathsf{forall}(x_1, x_2) \\ \wedge & y_0 : \mathsf{exists}(y_1, y_2) \\ \wedge & x_1 : \mathsf{yogi} \wedge x_2 \lhd^* z \\ \wedge & y_1 : \mathsf{guru} \wedge y_2 \lhd^* z \\ \wedge & z : \mathsf{has} \end{array}$$

4.2.1 Models of Dominance Constraints

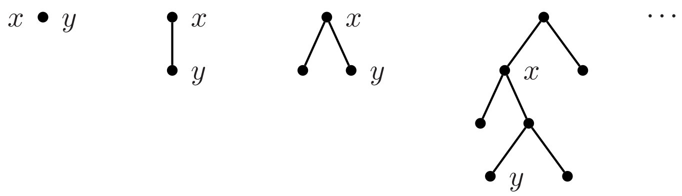
A model of a dominance constraint ϕ is a pair (T, I) of a tree T and an interpretation of I mapping variables of ϕ to nodes in T, and such that ϕ is satisfied. We write $(T, I) \models \phi$ to say that ϕ is satisfied by (T, I) and define it in the usual Tarskian way as follows:

$$\begin{array}{ll} (T, I) \models \phi \wedge \phi' & \text{if } (T, I) \models \phi \text{ and } (T, I) \models \phi' \\ (T, I) \models x \, R \, y & \text{if } I(x) \text{ is in relation } r \text{ to } I(y) \text{ in } T, \text{ for some } r \in R \end{array}$$

Solving dominance constraints is NP-hard. This was shown e.g. in [**?**] by encoding boolean satisfiability as a dominance problem. However, the constraint-based technique first outlined in [**?**] has proven quite successful and solves practical problems of arising e.g. in semantic underspecification very efficiently. The technique is formally studied and proven sound and complete in [**?**]. An extension of this technique to handle general descriptions with arbitrary Boolean connectives was presented in [**?**].

4.2.2 Solved Forms of Dominance Constraints

If a dominance constraint is satisfiable, it has infinitely many models; in particular, if T is a solution of ϕ, then any tree that contains T is also a solution. For example, here are a few models of $x \lhd^* y$:



The problem is similar when solving equations on first-order terms. The equation $f(a, X_1, X_2) = f(Y_1, b, Y_2)$ has infinitely many models, namely all first-order terms of the form $f(a, b, T)$ where T is a first-order term. Instead of attempting to enumerate all models, a solver returns a *most general unifier*:

$$X_1 = b, Y_1 = a, X_2 = Y_2$$

A unifier is also known as a *solved form*: it represents a non-empty family of solutions.

For a dominance constraint ϕ, we are going to proceed similarly and search for solved forms rather than for solution trees. The idea is that we simply want to arrange the nodes interpreting the variables of ϕ into a tree shape.

For our purpose, a solved form will make explicit the relationship $x \ r \ y$ that holds between every two variables x and y of the description: i.e. in a solved form $r \in$

$\{=, \lhd^+, \rhd^+, \perp\}$. In [**?**], we show that it is possible to use less explicit solved forms while preserving completeness: the relationship between x and y need not be fully decided, but merely one of $x \lhd^* y$ or $x \neg\lhd^* y$. For simplicity of presentation, we will only consider fully explicit solved forms, but, in practice, less explicit solved forms are to be preferred since they require less search. In [**?**], we show that the less explicit solved forms may avoid an exponential number of choice points.

4.3 Solving Dominance Constraints

A naïve approach to search for solutions of a description ϕ is to non-deterministically fix the relationship xry between any two variables of ϕ and then (1) verify that there are trees corresponding to this configuration (i.e. essentially that the solved form has a tree-shape), (2) verify that some of these trees also satisfy ϕ. This algorithm is exponential. But, many configurations do not correspond to trees, and/or cannot satisfy ϕ. This is where constraint propagation can prove very effective: it can deterministically rule out configurations that cannot lead to admissible solutions.

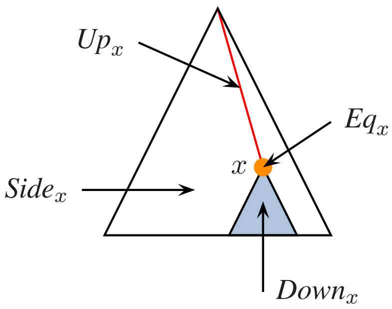
Thus our approach will consist in formulating 2 sets of criteria:

Well-formedness Constraints: these guarantee that a solved form has a tree-shape, i.e. that it has tree solutions.

Problem-specific Constraints: these guarantee that a solved form actually satisfies the description

4.3.1 Well-Formedness Constraints

Consider a solution tree for a description ϕ. Further consider the node $Node_x$, in that tree, interpreting variable x occurring in ϕ. When observed from the vantage point of this node, the nodes of the tree (hence the variables that they interpret) are partitioned into 4 regions: $Node_x$ itself, all nodes above, all nodes below, and all nodes to the side (i.e. in disjoint subtrees).



The variables of ϕ are correspondingly partitioned: all variables that are also interpreted by $Node_x$, all variables interpreted by the nodes above, resp. below or to the side. We introduce 4 variables to denote these sets:

$$Eq_x, Up_x, Down_x, Side_x$$

Clearly, x is one of the variables interpreted by Node_x:

$$x \in \mathit{Eq}_x$$

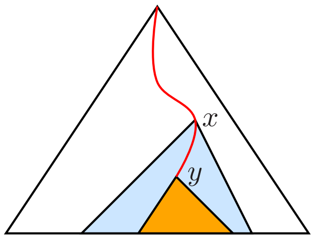
Furthermore, as described above, these sets must form a partition of the variables occurring in ϕ:

$$\mathsf{Vars}(\phi) = \mathit{Eq}_x \uplus \mathit{Up}_x \uplus \mathit{Down}_x \uplus \mathit{Side}_x$$

4.3.1.1 Characteristic Set Constraints

In order to formulate the constraints that will only license tree-shaped solved forms, we must first consider each individual case $x\ r\ y$ for $r \in \{=, \lhd^+, \rhd^+, \perp\}$. For each case $x\,r\,y$ and its negation $x\,\neg r\,y$, we will formulate characteristic constraints involving the set variables that we introduced above.

Let's consider the case $x \lhd^+ y$ for which a solution looks as shown below:



For convenience, we define, for each variable x, the additional set variables Eqdown_x and Equp_x as follows:

$$\begin{aligned} \mathit{Eqdown}_x &= \mathit{Eq}_x \uplus \mathit{Down}_x \\ \mathit{Equp}_x &= \mathit{Eq}_x \uplus \mathit{Up}_x \end{aligned}$$

We write $[\![x \lhd^+ y]\!]$ for the constraint characteristic of case $x \lhd^+ y$ and define it as follows:

$$\begin{array}{lcll} [\![x \lhd^+ y]\!] & \equiv & & \mathit{Eqdown}_y \subseteq \mathit{Down}_x \\ & & \wedge & \mathit{Equp}_x \quad\ \subseteq \mathit{Up}_y \\ & & \wedge & \mathit{Side}_x \quad\ \subseteq \mathit{Side}_y \end{array}$$

I.e. all variables equal or below y are below x, all variables equal or above x are above y, and all variables disjoint from x are also disjoint from y. This illustrates how set constraints permit to succinctly express certain patterns of inference. Namely $[\![x \lhd^+ y]\!]$ precisely expresses:

$$\begin{array}{ll} \forall z & y \lhd^* z \rightarrow x \lhd^+ z \\ \forall z & z \lhd^* x \rightarrow z \lhd^+ y \\ \forall z & z \perp x \ \rightarrow \ z \perp y \end{array}$$

The negation is somewhat simpler and states that no variable equal to x is above y, and no variable equal to y is below x. Remember that $S_1 \parallel S_2$ expresses that S_1 and S_2 are disjoint.

$$\begin{array}{lcll} [\![x \neg\lhd^+ y]\!] & \equiv & & \mathit{Eq}_x \parallel \mathit{Up}_y \\ & & \wedge & \mathit{Eq}_y \parallel \mathit{Down}_x \end{array}$$

We can define the other cases similarly. Thus $[\![x \perp y]\!]$:

$$\begin{array}{lcll} [\![x \perp y]\!] & \equiv & & \mathit{Eqdown}_x \subseteq \mathit{Side}_y \\ & & \wedge & \mathit{Eqdown}_y \subseteq \mathit{Side}_x \end{array}$$

and its negation $[\![x \neg\bot\, y]\!]$:

$$\begin{array}{rcl} [\![x \perp y]\!] & \equiv & \ \ \mathit{Eq}_x \parallel \mathit{Side}_y \\ & & \wedge\ \ \mathit{Eq}_y \parallel \mathit{Side}_x \end{array}$$

For the case $[\![x = y]\!]$ we first introduce notation. We write Node_x for the tuple defined as follows:

$$\mathit{Node}_x \quad \equiv \quad \langle \mathit{Eq}_x, \mathit{Up}_x, \mathit{Down}_x, \mathit{Side}_x, \mathit{Equp}_x, \mathit{Eqdown}_x, \mathit{Daughters}_x \rangle$$

where $\mathit{Daughters}_x = f(\mathit{Node}_{x_1}, \ldots, \mathit{Node}_{x_n})$ when the constraint $x : f(x_1, \ldots, x_n)$ occurs in ϕ (more about this when presenting the problem-specific constraints). Now we can simply define $[\![x = y]\!]$ as:

$$[\![x = y]\!] \quad \equiv \quad \mathit{Node}_x = \mathit{Node}_y$$

and its negation $[\![x \neg\!= y]\!]$ as:

$$[\![x \neg\!= y]\!] \quad \equiv \quad \mathit{Eq}_x \parallel \mathit{Eq}_y$$

4.3.1.2 Well-Formedness Clauses

For every pair of variables x and y, we introduce a finite domain variable R_{xy} to denote the relationship $x\ R_{xy}\ y$ that obtains between them. $R_{xy} \in \{=, \lhd^+, \rhd^+, \perp\}$ and we freely identify $\{=, \lhd^+, \rhd^+, \perp\}$ with $\{1, 2, 3, 4\}$. In a solved form, every R_{xy} must be determined.

In order to guarantee that a solved form is tree-shaped, for each pair of variables x and y, we consider the 4 mutually exclusive possibilities. For each possible relation $r \in \{=, \lhd^+, \rhd^+, \perp\}$ we state that either $R_{xy} = r$ and the corresponding characteristic constraints $[\![x\ r\ y]\!]$ hold, or $R_{xy} \neq r$ and the constraints $[\![x \neg r\ y]\!]$ characteristic of the negation hold.

Thus for each pair of variables x and y, we stipulate that the following 4 *Well-Formedness Clauses* hold:

$$\begin{array}{c} [\![x = y]\!] \ \wedge R_{xy} = = \ \ \textbf{or}\ R_{xy} \neq = \ \ \wedge\ [\![x \neg\!= y]\!] \\ [\![x \lhd^+ y]\!] \wedge R_{xy} = \lhd^+ \textbf{ or } R_{xy} \neq \lhd^+ \wedge [\![x \neg\lhd^+ y]\!] \\ [\![x \rhd^+ y]\!] \wedge R_{xy} = \rhd^+ \textbf{ or } R_{xy} \neq \rhd^+ \wedge [\![x \neg\rhd^+ y]\!] \\ [\![x \perp y]\!] \ \wedge R_{xy} = \perp \ \ \textbf{or}\ R_{xy} \neq \perp \ \ \wedge\ [\![x \neg\perp y]\!] \end{array}$$

These clauses are all of the form C_1 **or** C_2. This denotes a *disjunctive propagator* and is explained in the next section.

4.3.1.3 Disjunctive Propagator

In Logic Programming, the only method available for dealing with complex disjunctions is non-determinism. Thus, in Prolog, you would write:

```
C1 ; C2
```

and this would have the effect to first try `C1`, and then on backtracking to try `C2`. In other words, in Prolog you must commit very early to exploring either one alternative or the other.

Early commitment is a poor strategy in general. In many cases, it would be preferable to delay this choice until e.g. sufficient information is known to reject one alternative altogether. This is the intuition behind the disjunctive propagator:

```
C1 or C2
```

It is a propagator, not a choice point! `C1` and `C2` are arbitrary constraints: when one of them becomes inconsistent with what is currently known (i.e. with the store of basic constraints), then the propagator reduces to the other one. Thus, a disjunctive propagator has the declarative semantics of sound logical disjunction (unlike Prolog's `;` operator which depends on *negation as failure*) and the operational semantics given by the rules below:

$$\frac{\mathcal{B} \wedge C_1 \quad \rightarrow^* \quad \text{false}}{\mathcal{B} \wedge (C_1 \text{ **or** } C_2) \quad \rightarrow \quad \mathcal{B} \wedge C_2}$$

$$\frac{\mathcal{B} \wedge C_2 \quad \rightarrow^* \quad \text{false}}{\mathcal{B} \wedge (C_1 \text{ **or** } C_2) \quad \rightarrow \quad \mathcal{B} \wedge C_1}$$

How is this possible? We have already explained that all computations operate over a *constraint store*. We can go one better and allow *nested* constraint stores. This idea is indeed supported by Oz under the name of *Computation Spaces* (see Section 6.4). The disjunctive propagator `C1 or C2` creates 2 nested computation spaces - one in which to execute `C1` and one in which to (concurrently) execute `C2` - and it monitors both spaces. If an inconsistency is derived in the space where, say, `C2` executes, then the space where `C1` executes is simply merged with the current space, thus *committing* to alternative `C1`.

A disjunctive propagator `C1 or C2` allows to delay the choice between the two alternative in the hope that constraint propagation alone will be able to decide it. However, this may fail to happen, in which case, to ensure completeness, we may have to non-deterministically force the choice anyway. How can we achieve this?

The usual technique is to introduce a *Control Variable*, i.e. a finite domain variable whose purpose is simply to allow to choose the alternative. For example, you might write:

```
X=1 C1  or  X=2 C2
```

Thus, if constraint propagation is unable to decide the disjunction either way, you can non-deterministically either try `X=1` or try `X=2`. Variable `X` allows you to *control* the disjunction.

In the case of the well-formedness clauses, we already have the variables R_{xy} which can serve this purpose.

4.3.2 Problem-Specific Constraints

The well-formedness constraints guarantee that our solved forms correspond to trees. We now need additional constraints to guarantee that these trees actually satisfy our description ϕ. We shall achieve this by translating each literal in ϕ into a constraint as explained below.

If the constraint is of the form $x\ R\ y$, then the translation is trivial in terms of the variable R_{xy} that we introduced earlier to denote the relationship between x and y:

$$R_{xy} \in R$$

If the constraint is $x : f(x_1, \ldots, x_n)$, then it is translated into the constraints below:

$$\begin{aligned} \mathit{Down}_x &= \mathit{Eqdown}_{x_1} \uplus \ldots \uplus \mathit{Eqdown}_{x_n} \\ \mathit{Equp}_x &= \mathit{Up}_{x_i} \qquad \text{forall } 1 \leq i \leq n \\ \mathit{Daughters}_x &= f(\mathit{Node}_{x_1}, \ldots, \mathit{Node}_{x_n}) \end{aligned}$$

Particularly important is the first constraint which states that the trees rooted at the daughters are pairwise disjoint and that they exhaustively account for the variables below x.

4.3.3 Searching for Solved Forms

Given a description ϕ, we transform it into the constraint satisfaction problem given by the conjunction of the well-formedness constraints and the problem specific constraints. The CSP can be solved by searching for assignments to the variables R_{xy} consistent with these constraints.

4.4 Implementation

In this section, we present the Oz functor `Dominance` (file `Dominance.oz`) implementing a solver as described in the preceding section. It exports `solutionPredicate` which can be used as follows:

```
{ExploreAll {Dominance.solutionPredicate Desc}}
```

where `Desc` is a tree description in the form of a list where each element is of one of the forms described below:

`dom(X R Y)`

where `X` and `Y` are both atoms naming variables, and `R` is an atom or list of atoms from the set `eq`, `above`, `below`, `side`.

`lab(X f(X1 ... Xn))`

where `X` and `Xi` are atoms naming variables.

`labeled(X)`

where `X` is an atom naming a variable. This is a new constraint not mentioned in our abstract syntax but very convenient in practice: it simply states that `X` must be identified with some variable that is explicitly labeled in the input description.

36a ⟨**Dominance.oz** 36a⟩≡

```
functor
import FD FS
export SolutionPredicate
define
   ⟨Dominance: SolutionPredicate 36b⟩
   ⟨Dominance: MakeNode 36c⟩
   ⟨Dominance: characteristic constraints 37a⟩
   ⟨Dominance: well-formedness clauses 37b⟩
   ⟨Dominance: Solver class 37c⟩
   ⟨Dominance: utilities 39a⟩
end
```

`SolutionPredicate` takes a tree description as argument and returns a procedure appropriate for encapsulated search. This procedure creates a solver object. This object is only needed for converting the description into constraints and starting the search.

36b ⟨**Dominance: SolutionPredicate** 36b⟩≡

```
fun {SolutionPredicate Desc}
   proc {$ Info}
      {New Solver init(Desc Info) _}
   end
end
```

4.4.1 MakeNode

For each variable x in the description, we must create all the corresponding set variables required by the encoding. This is the purpose of function `MakeNode`. It is invoked as `{MakeNode I Vars}` where `I` is the unique integer used to represent variable x and `Vars` is a set variable representing the set of all variables occurring in the description. The function returns a representation of $Node_x$ in the form of a record.

36c ⟨**Dominance: MakeNode** 36c⟩≡

```
fun {MakeNode I Vars}
   [Eq Down Up Side] = {FS.var.list.decl 4}
   EqDown = {FS.union Eq Down}
   EqUp   = {FS.union Eq Up}
in
   {FS.partition [Eq Down Up Side] Vars}
   {FS.include I Eq}
   node(
      eq        : Eq
      down      : Down
      up        : Up
      side      : Side
      eqdown    : EqDown
      equp      : EqUp
      daughters : _)
end
```

4.4.2 Characteristic Constraints

The constraints characteristic of $x\ r\ y$ or $x\ \neg r\ y$ for each $r \in \{=, \lhd^+, \rhd^+, \perp\}$ can be easily expressed in terms of the node representations $Node_x$ and $Node_y$.

37a ⟨Dominance: characteristic constraints 37a⟩≡

```
proc {Equal N1 N2} N1=N2 end
proc {NotEqual N1 N2}
   {FS.disjoint N1.eq N2.eq}
end
proc {Above N1 N2}
   {FS.subset N2.eqdown N1.down}
   {FS.subset N1.equp N2.up}
   {FS.subset N1.side N2.side}
end
proc {NotAbove N1 N2}
   {FS.disjoint N1.eq N2.up}
   {FS.disjoint N2.eq N1.down}
end
proc {Disjoint N1 N2}
   {FS.subset N1.eqdown N2.side}
   {FS.subset N2.eqdown N1.side}
end
proc {NotDisjoint N1 N2}
   {FS.disjoint N1.eq N2.side}
   {FS.disjoint N2.eq N1.side}
end
```

4.4.3 Well-Formedness Clauses

Procedure `Clauses` creates the well-formedness clauses for the pair of variables x and y, where `N1` denotes $Node_x$, `N2` denotes $Node_y$, and `R` denotes R_{xy}. Note that we identify $=$ with 1, $\lhd^+$ with 2, $\rhd^+$ with 3, and $\perp$ with 4.

37b ⟨Dominance: well-formedness clauses 37b⟩≡

```
proc {Clauses N1 N2 C}
   thread or {Equal    N1 N2} C=1 [] C\=:1 {NotEqual    N1 N2} end end
   thread or {Above    N1 N2} C=2 [] C\=:2 {NotAbove    N1 N2} end end
   thread or {Above    N2 N1} C=3 [] C\=:3 {NotAbove    N2 N1} end end
   thread or {Disjoint N1 N2} C=4 [] C\=:4 {NotDisjoint N1 N2} end end
end
```

4.4.4 `Solver` class

We use an object of class `Solver` to turn a description into a CSP. There is no overwhelming reason to use an object: the author simply finds that an OO-idiom is in this case rather pleasant. In particular, it makes it easier to manipulate state (see e.g. `@counter` attribute later).

37c ⟨Dominance: Solver class 37c⟩≡

```
class Solver
   ⟨Dominance: Solver class, attributes 38a⟩
   ⟨Dominance: Solver class, init method 38b⟩
   ⟨Dominance: Solver class, var2node method 39b⟩
   ⟨Dominance: Solver class, lab method 40a⟩
   ⟨Dominance: Solver class, dom method 40b⟩
   ⟨Dominance: Solver class, labeled method 41a⟩
   ⟨Dominance: Solver class, info method 41b⟩
end
```

4.4.4.1 Attributes

Each solver object has several attributes. `@counter` allows us to give each variable occurring in the description a distinct integer to encode it. `@var2int` is a dictionary mapping each atom naming a variable to the corresponding integer encoding it. `@int2node` is a dictionary mapping an integer encoding a variable x to the record representing $Node_x$. `@vars` is a set variable representing the set of all variables occurring in the description. `@labs` represents the subset of these variables that are explicitly labeled in the description. `@choices` is a dictionary that allows us to map a pair of variables x, y to the corresponding R_{xy} representing the relationship between them. Since R_{yx} is the inverse of R_{xy}, we only need to represent one of them: we only represent R_{xy} when x, y are respectively encoded by integers `I,J` and `I` is larger than `J`. For simplicity, we assume that there are fewer than 1000 variables in the description and use index `I*1000+J` to retrieve R_{xy} from `@choices`.

38a ⟨**Dominance: Solver class, attributes** 38a⟩≡

```
attr counter:0 var2int int2node vars labs choices
```

4.4.4.2 `init` method

The `init` method initializes the attributes and then processes each literal of the description, creating the corresponding problem-specific constraints. A literal is processed simply by invoking it as method on the solver object. The first time that variable x is encountered, it is assigned a new integer to encode it and its corresponding $Node_x$ is created. The nested loops create all well-formedness clauses for the variables encountered in the description. Finally, the *first-fail* distribution strategy is invoked to search for consistent assignments to the variables R_{xy}.

38b ⟨**Dominance: Solver class, init method** 38b⟩≡

```
meth init(Desc Info)
   var2int  <- {NewDictionary}
   int2node <- {NewDictionary}
   vars     <- {FS.var.decl}
   labs     <- {FS.var.decl}
   choices  <- {NewDictionary}
   for D in Desc do {self D} end
   {CloseSet @vars}
   {CloseSet @labs}
   {self info(Info)}
   for I in 1..@counter do
```

```
            for J in 1..(I-1) do
               {Clauses @int2node.I @int2node.J @choices.(I*1000+J)}
            end
         end
         {FD.distribute ff {Dictionary.items @choices}}
      end
```

Note that @vars and @labs are initially completely underspecified. Whenever a new variable is encountered, it is stated to be an element of @vars. Whenever a labeling constraint is encountered, the labeled variable is stated to be an element of @labs. When all literals have been processed, then all variables are known, and all explicitly labeled variables as well: CloseSet is invoked on both @vars and @labs to state that whatever is known to be in these sets sofar is really all there is (i.e. the lower bound is also the upper bound).

39a ⟨**Dominance: utilities** 39a⟩≡ 40c▷

```
proc {CloseSet S}
   {FS.var.upperBound {FS.reflect.lowerBound S} S}
end
```

4.4.4.3 var2node method

Whenever, in a literal, we encounter an atom naming a variable, we invoke method var2node to make sure that this variable is already encoded and to retrieve the corresponding node representation. If this is the first time we encounter this variable x, we allocate for it a new integer and create a node representation $Node_x$ for it. Further, we also create R_{xy} for all variables y that were known sofar (i.e. all variables encoded by integers smaller than the one encoding x).

39b ⟨**Dominance: Solver class, var2node method** 39b⟩≡

```
meth var2node(X Node)
   I = {Dictionary.condGet @var2int X unit}
in
   if I==unit then
      I=(counter<-I)+1
   in
      {FS.include I @vars}
      Node={MakeNode I @vars}
      @var2int.X   := I
      @int2node.I  := Node
      for J in 1..(I-1) do
         @choices.(I*1000+J) := {FD.int 1#4}
      end
   else Node=@int2node.I end
end
```

4.4.4.4 lab method

This method translates a labeling literal lab(X f(X1 ... Xn)) into the corresponding problem-specific constraints (as described in Section 4.3.2). The last line states that the variable named by X is an element of the set of labeled variables.

40a ⟨**Dominance: Solver class, lab method** 40a⟩≡

```
meth lab(X R)
   N = {self var2node(X $)}
in
   N.daughters =
   {Record.map R fun {$ Xi} {self var2node(Xi $)} end}
   {FS.partition
    {Record.map N.daughters
     fun {$ Ni}
        Ni.up = N.equp
        Ni.eqdown
     end}
    N.down}
   {FS.include @var2int.X @labs}
end
```

4.4.4.5 `dom` method

This method translates a dominance literal `dom(X R Y)` into the corresponding problem-specific constraint (as described in Section 4.3.2). Remember that since R_{yx} is the inverse of R_{xy}, we only represent one of them: we represent R_{xy} when x, y are encoded by `I,J` and `I>J`.

40b ⟨**Dominance: Solver class, dom method** 40b⟩≡

```
meth dom(X R Y)
   {self var2node(X _)}
   {self var2node(Y _)}
   I = @var2int.X
   J = @var2int.Y
in
   if I==J then 1::{Encode R}
   elseif I>J then
      @choices.(I*1000+J)::{Encode R}
   else
      @choices.(J*1000+I)::{Encode {Inverse R}}
   end
end
```

Here is how to inverse and encode the symbolic representation of a dominance specification.

40c ⟨**Dominance: utilities** 39a⟩+≡ ◁39a

```
fun {Encode R}
   case R
   of eq    then 1
   [] above then 2
   [] below then 3
   [] side  then 4
   [] _|_   then {Map R Encode}
   end
```

```
end
fun {Inverse R}
   case R
   of eq    then eq
   [] above then below
   [] below then above
   [] side  then side
   [] _|_   then {Map R Inverse}
   end
end
```

4.4.4.6 `labeled` method

This method translates a literal `labeled(X)` and forces the variable named by `X` to be eventually identified with one of the explicitly labeled variables. We introduce `I` to denote this explicitly labeled variable.

41a ⟨**Dominance: Solver class, labeled method** 41a⟩≡

```
meth labeled(X)
   N = {self var2node(X $)}
   I = {FD.decl}
in
   {FS.include I @labs}
   {FS.include I N.eq}
end
```

4.4.4.7 `labeled` method

The sole purpose of this method is to assemble a representation of a solution. A solution is represented by a tuple mapping integer `I` to $Node_x$ for the variable x that `I` encodes. Actually, not quite, as we also augment $Node_x$ with an indication of the name x of the variable (on feature `var`).

41b ⟨**Dominance: Solver class, info method** 41b⟩≡

```
meth info($)
   Int2var = {NewDictionary}
in
   {ForAll {Dictionary.entries @var2int}
    proc {$ V#I} Int2var.I := V end}
   {Record.mapInd {Dictionary.toRecord o @int2node}
    fun {$ I N}
       {AdjoinAt N var Int2var.I}
    end}
end
```

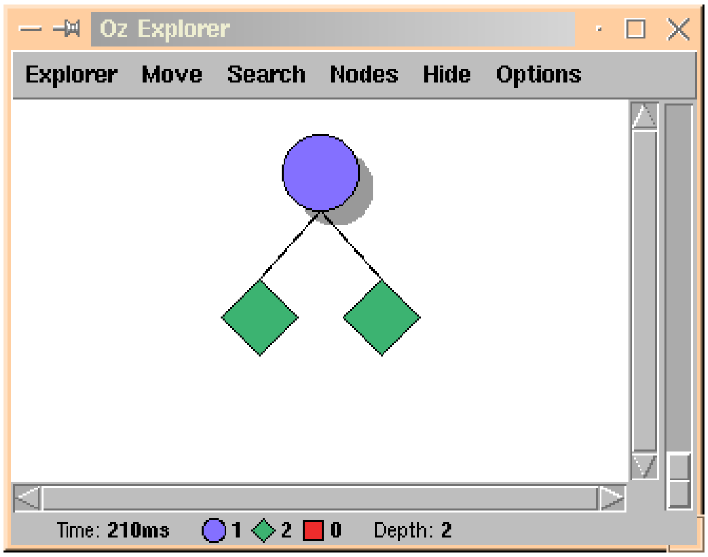
4.5 Application

In this section, we provide a small interactive application[1] to test our solver for dominance constraints. A window will appear as shown below and you can type the literals

[1] `http://www.ps.uni-sb.de/~duchier/esslli-2000/DomDemo.oza`

of a description in the text area. The literals shown in the picture correspond to the example *"every yogi has a guru"*.

When you have typed your literals, you can click on `Solve` and The Explorer window pops up and displays the search tree for all solutions to your description. For our example, we get the expected 2 readings:



4.5.1 Implementation

The implementation is provided as functor `DomDemo` (file `DomDemo.oz`).

42a ⟨**DomDemo.oz** 42a⟩≡

```
functor
import
   QTk at 'http://www.info.ucl.ac.be/people/ned/qtk/QTk.ozf'
```

```
   TkTools
   Application
   Compiler
   Explorer
   Dominance
define
   TextBox
   proc {Clear} {TextBox delete("1.0" "end")} end
   proc {Solve}
      Text = {TextBox getText("1.0" "end" $)}
   in
      try
         Desc = {Compiler.virtualStringToValue '['#Text#']'}
         Pred = {Dominance.solutionPredicate Desc}
      in
         {Explorer.object all(Pred)}
      catch _ then
         {New TkTools.error
          tkInit(master:Window text:'error') _}
      end
   end
   proc {Quit } {Application.exit 0} end
   Window={QTk.build
           lr(text(height:20 width:50 bg:white handle:TextBox)
              continue continue newline
              button(text:'Solve' glue:ew action:Solve)
              button(text:'Clear' glue:ew action:Clear)
              button(text:'Quit'  glue:ew action:Quit))}
   {Window show}
end
```

5

Dependency Parsing

In this chapter, we will take a look at parsing in the framework of dependency grammar. The chapter is called *"Dependency Parsing"* rather than, say, *"Parsing with Dependency Grammars"* because, for reasons of space/time, it is concerned only with the creation of dependency trees for input sentences and not with other issues, such as word-order, which are also essential for determining grammaticality. Also coverage will be minimal and is only meant to illustrate the application of the techniques presented here.

Dependency parsing is particularly interesting because it exhibits, in a very simple way, 2 fundamental forms of ambiguity that commonly arise in parsing: lexical ambiguity and structural ambiguity, and allows to showcase convincingly constraint-based techniques to effectively handle such ambiguities.

Maruyama [**?**] was the first to propose a complete treatment of dependency grammar as a CSP and described parsing as a process of incremental disambiguation. Harper [**?**] continues this line of research and proposed several algorithmic improvements within the MUSE CSP framework [**?**]. Menzel [**?**] [**?**] [**?**] advocates the use of soft *graded* constraints for robustness in e.g. parsing spoken language. His proposal turns parsing into a more expensive optimization problem, but adapts gracefully to constraint violations.

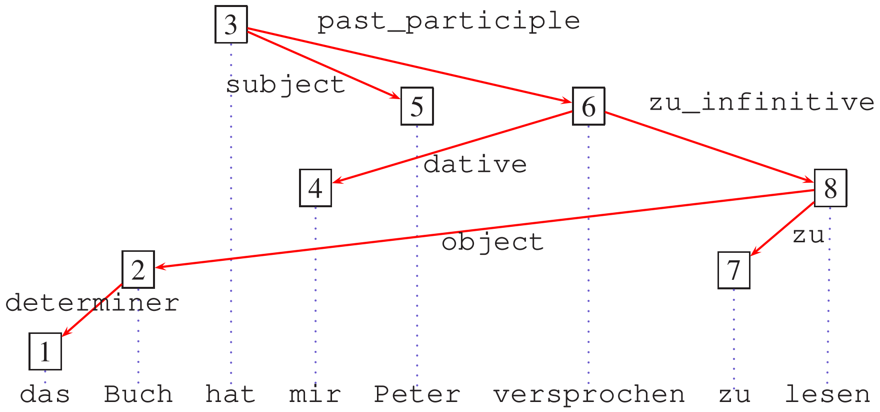
The material in this chapter is based on our paper [**?**]. Our presentation has one noticeable advantage over Maruyama's in that it follows modern linguistic practice: the grammar is specified by a lexicon and a collection of principles. The formulation in this chapter is also a little simpler than the one in [**?**] because it takes advantage of the new *selection union constraint* (see Section 5.4.6 and Section 6.9.4).

5.1 Overview

We hope that the fundamentals of DG are known to the reader. We review the basic ideas only very briefly.

Contrary to phrase structure grammar, where parse trees consist mostly of non-terminal nodes and words appear only as leaves, dependency grammar postulates no non-terminals: words are in bijection with the nodes of the dependency tree. In other words, edges are drawn directly between words. Thus a finite verb has typically an edge directed to its subject, and another to its object.

Unlike traditional phrase structure trees, dependency trees usually allow crossing branches. This makes dependency trees rather attractive for languages with free word order (e.g. German), and for the representation of long distance dependencies. As an illustration, consider the dependency tree shown below:



Each box represents a node in the dependency tree. For ease of reading, the words are written at the bottom and for each one the corresponding node is indicated by connecting it to the word by a vertical dotted blue line. Also each box contains an integer indicating the position of the word in the input. The directed edges of the dependency tree are represented by red arrows. Each arrow is labeled to indicate the type of the dependency, for example by `subject` or `zu_infinitive`. We call such labels *roles*. These roles are purely syntactic and not to be confused e.g. with thematic roles.

5.2 Formal Presentation

In this section, we briefly outline the formal presentation of dependency grammar presented in [**?**]. The basic idea is that each node of the dependency tree must be assigned a lexical entry from the lexicon, and that certain principles of well-formedness must be verified. For example, the lexical entry stipulates a valency, and the node must have precisely the outgoing dependency edges that realize this valency.

A dependency grammar is a 7-tuple

$$\langle Words, Cats, Args, Comps, Mods, Lexicon, Rules \rangle$$

Words a finite set of strings notating fully inflected forms of words.

Cats a finite set of categories such as `n` (noun), `det` (determiner), or `vfin` (finite verb).

Args a finite set of (what we call) agreement tuples such as `<masc sing 3 nom>`.

Comps a finite set of *complement* roles, such as `subject` or `zu_infinitive`.

Mods a finite set of *modifier* roles, such as `adj` (adjectives), disjoint from *Comps*. We write $Roles = Comps \uplus Mods$ for the set of all types of roles.

Lexicon a finite set of lexical entries (see below).

Rules a finite family of binary predicates, indexed by role labels, expressing local grammatical principles: for each $\rho \in \textit{Roles}$, there is $\Gamma_\rho \in \textit{Roles}$ such that $\Gamma_\rho(w_1, w_2)$ characterizes the grammatical admissibility of a dependency edge labeled ρ from mother w_1 to daughter w_2.

A lexical entry is an attribute value matrix (AVM) with signature:

$$\begin{bmatrix} \textsf{string} & : & \textit{Words} \\ \textsf{cat} & : & \textit{Cats} \\ \textsf{agr} & : & \textit{Agrs} \\ \textsf{comps} & : & 2^{\textit{Comps}} \end{bmatrix}$$

and we write attribute access in functional notation. If e is a lexical entry, then $\textsf{string}(e)$ is the full form of the corresponding word, $\textsf{cat}(e)$ is the category, $\textsf{agr}(e)$ is the agreement tuple, and $\textsf{comps}(e)$ the valency expressed as a set of complement roles.

5.2.1 Dependency Tree

We assume given a set of nodes $\mathcal{V}$ which for simplicity we identify with the set of integers $\{1, \ldots, n\}$, and a set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \textit{Roles}$ of labeled directed edges between these node. $(\mathcal{V}, \mathcal{E})$ is a directed graph, in the classical sense, with labeled edges. We restrict our attention to finite graphs that are also trees.

A dependency tree is then defined as a pair $\mathcal{T} \equiv ((\mathcal{V}, \mathcal{E}), \textsf{entry})$ of a tree as stipulated above and a function $\textsf{entry}$ mapping each node in $\mathcal{V}$ to a lexical entry in *Lexicon*.

5.2.2 Well-Formedness Principles

Not all dependency trees as described above are grammatically admissible. We now describe the conditions of admissibility, aka well-formedness.

While we have identified nodes with integers, we still prefer to write w_i (and often just w or w') instead of i to remind the reader that they correspond to words and to distinguish them from other occurrences of integers that have no such interpretation. We also write $w \xrightarrow{\rho} w'$ to represent an edge labeled ρ from mother w to daughter w'.

First, any complement required by a node's valency must be realized precisely once:

$$\forall w \in \mathcal{V},\ \forall \rho \in \textsf{entry}(w),\ \exists! w' \in \mathcal{V},\ w \xrightarrow{\rho} w' \in \mathcal{E}$$

Second, if there is an outgoing edge from w, then it must be labeled by a modifier role or by a complement role in w's valency:

$$\forall w \xrightarrow{\rho} w' \in \mathcal{E},\ \rho \in \textit{Mods} \cup \textsf{comps}(\textsf{entry}(w))$$

Third, whenever there is an edge $w \xrightarrow{\rho} w'$, then the grammatical condition $\Gamma_\rho(w, w')$ for $\Gamma_\rho \in \textit{Rules}$ must be satisfied in $\mathcal{T}$:

$$\forall w \xrightarrow{\rho} w' \in \mathcal{E},\ \mathcal{T} \models \Gamma_\rho(w, w')$$

5.3 Improving Lexical Economy

Typically the same full form of a word corresponds to several distinct agreement tuples. The preceding formal presentation simply assumed that there would be a distinct lexical entry for each one of these tuples. In practice, this would not be very realistic. Therefore, we are going to define licensed lexical entries in terms of more economical *lexicon entries*.

Thus, instead of a feature `agr` mapping to a single agreement tuple, a lexicon entry has a feature `agrs` mapping to a set of agreement tuples.

Although this is much less useful, we shall do the same for category information and replace feature `cat` mapping to a single category by a feature `cats` mapping to a set of categories.

Optional complements are another source of considerable redundancy in the lexicon. Therefore, instead of modeling the valency by a single feature `comps` mapping to a set of complement roles, we shall have 2 features: `comps_req` mapping to a set of required complement roles, and `comps_opt` mapping to a set of optional complement roles.

We now define the lexicon as a finite set of *lexicon entries*, where a lexicon entry is an AVM of the form:

$$\begin{bmatrix} \textsf{word} & : & W \\ \textsf{cats} & : & C \\ \textsf{agrs} & : & A \\ \textsf{comps\_req} & : & R \\ \textsf{comps\_opt} & : & O \end{bmatrix}$$

and the *lexical entries* licensed by the lexicon entry above are all AVMs of the form:

$$\begin{bmatrix} \textsf{word} & : & W \\ \textsf{cat} & : & c \\ \textsf{agr} & : & a \\ \textsf{comps} & : & S \end{bmatrix}$$

where

$$\begin{array}{l} c \in C \\ a \in A \\ R \subseteq S \subseteq R \cup O \end{array}$$

This simple formulation demonstrates how constraints can be used to produce compact representations of certain forms of lexical ambiguity. Note that lexicon entries as presented here do not support covariation of features: in such a case, you still need to expand into multiple lexicon entries. Covariation could easily be added and supported using the selection constraint, but I have never found the need for it: the most common application for covariation is agreement, and we have already elegantly taken care of it by means of a product of finite domains.

5.4 Constraint Model

In this section, we develop a constraint model for the formal framework of Section 5.2. In essence, the constraint model is an axiomatization of admissible dependency trees, but also has a reading as a constraint program. It is carefully formulated to take effective advantage of modern technological support for constraint programming.

The approach is, as usual, a reduction to a CSP. We introduce variables to represent the quantities and mappings mentioned in the formal model, and formulate constraints on them that precisely capture the conditions that the formal model stipulates.

Our approach takes essential advantage of the intrinsic *finiteness* of the problem: (1) a dependency tree has precisely one node for each word, (2) there are finitely many edges labeled with roles that can be drawn between n nodes, (3) for each word, there are finitely many possible lexical entries. Thus the problem is to pick a set of edges and, for each node, a lexical entry, such that (a) the result is a tree, (b) none of the grammatical conditions are violated. Viewed in this light, dependency parsing is a configuration problem, and therefore it is no surprise that constraint programming should be able to address it very effectively.

5.4.1 Representation

We now introduce the most important variables in our constraint model. Other variables will be introduced later in the course of developing our axiomatization. As in the formal model, a node is identified with the integer representing the position of the corresponding word in the input sentence.

5.4.1.1 Lexical Attributes

The formal model posited a function entry to map a node to a lexical entry. Here, we shall seek to eliminate the use of entry and use lexical attribute functions that operate directly on the node rather than on the entry which it is assigned:

$$\begin{array}{rcl} \mathsf{word}(w) & \equiv & \mathsf{word}(\mathsf{entry}(w)) \\ \mathsf{cat}(w) & \equiv & \mathsf{cat}(\mathsf{entry}(w)) \\ \mathsf{agr}(w) & \equiv & \mathsf{agr}(\mathsf{entry}(w)) \\ \mathsf{comps}(w) & \equiv & \mathsf{comps}(\mathsf{entry}(w)) \end{array}$$

In the constraint model, $\mathsf{cat}(w)$ is a variable denoting the category of the lexical entry assigned to node w (similarly for the other attributes).

5.4.1.2 Daughter Sets

We need to represent labeled edges between nodes. Traditionally, this is realized by means of feature structures: each node is a feature structure and an edge $w \xrightarrow{\rho} w'$ is represented by the presence of a feature ρ on w, such that $w.\rho = w'$. This, however, is a lousy idea when the goal is to take advantage of active constraint propagation. The problem is that, in the traditional view, features are partial functions: i.e. ρ is a partial function from nodes to nodes (from now on, we will write $\rho(w)$ instead of $w.\rho$). It is rather inconvenient to try to express constraints on $\rho(w)$ when it is not always defined!

However, a slight change of representation allows us to turn ρ into a total function. Instead of $\rho(w)$ being either undefined or defined and denoting a node, we let it denote a set of nodes. Now instead of being undefined $\rho(w)$ is simply empty. In the case where it was originally defined, it now denotes a singleton set.

Thus $\mathsf{subject}(w)$ denotes the set of subjects of w: empty except when w is a finite verb, in which case it is a singleton. We say that $\mathsf{subject}(w)$ is a *daughter set* of w, i.e. a set of daughters. This idea has the second advantage that, in addition to complements (like subject), it also naturally accommodates modifiers (like adjectives): $\mathsf{adj}(w)$ is the set of adjectives of w. The difference is that a modifier daughter set may have any number of elements instead of at most 1 for a complement daughter set.

Formally, for each role $\rho \in \mathit{Roles}$, we introduce a function ρ such that $\rho(w)$ is the set of immediate daughters of w whose dependency edge is labeled with ρ:

$$\rho(w) = \{w' \mid w \xrightarrow{\rho} w' \in \mathcal{E}\}$$

in the constraint model $\rho(w)$ is a finite set variable.

5.4.1.3 Lexicon

We consider a function lex mapping a full form of a word to the corresponding set of lexicon entries, or rather, without loss of generality, we assume that lex returns a sequence rather than a set, which allows us to identify lexicon entries by position in this sequence:

$$\mathsf{lex}(s) = \langle e \in \mathit{Lexicon} \mid \mathsf{word}(e) = s \rangle$$

We will have to pick one entry in this sequence. For this purpose we introduce $\mathsf{entryindex}(w)$ to denote the position of the selected entry in the sequence obtained from the lexicon.

5.4.2 Lexical Constraints

In this section, we define precisely the constraints governing assignment of lexical attributes. Consider the sequence of lexicon entries obtained for w from the lexicon:

$$\langle e_1, \ldots, e_n \rangle = \mathsf{lex}(\mathsf{word}(w))$$

let's write I for the position of the one that is selected out of this sequence:

$$I = \mathsf{entryindex}(w)$$

Abstractly, we can write E to denote the selected entry and define it thus:

$$E = \langle e_1, \ldots, e_n \rangle[I]$$

The lexical attributes assigned to w are then obtained as explained in Section 5.3:

$$\begin{gathered}\mathsf{cat}(w) \in \mathsf{cats}(E) \\ \mathsf{agr}(w) \in \mathsf{agrs}(E) \\ \mathsf{comps\_req}(E) \subseteq \mathsf{comps}(w) \subseteq \mathsf{comps\_req}(E) \cup \mathsf{comps\_opt}(E)\end{gathered}$$

However, for practical reasons of implementation, the selection constraint cannot operate on arbitrary AVMs, but is only provided for finite domains and finite sets. This means that we cannot use the selection constraint directly on the sequence of lexicon entries to obtain E. However, we only need E to access its attributes, and we overcome the limitation we pointed out by pushing attribute access into the selection:

$$\begin{array}{r}\mathsf{cats}(w) = \langle \mathsf{cats}(e_1), \ldots, \mathsf{cats}(e_n)\rangle[I] \\ \mathsf{agrs}(w) = \langle \mathsf{agrs}(e_1), \ldots, \mathsf{agrs}(e_n)\rangle[I] \\ \mathsf{comps\_req}(w) = \langle \mathsf{comps\_req}(e_1), \ldots, \mathsf{comps\_req}(e_n)\rangle[I] \\ \mathsf{comps\_opt}(w) = \langle \mathsf{comps\_opt}(e_1), \ldots, \mathsf{comps\_opt}(e_n)\rangle[I]\end{array}$$

$$\begin{array}{c}\mathsf{cat}(w) \in \mathsf{cats}(w) \\ \mathsf{agr}(w) \in \mathsf{agrs}(w) \\ \mathsf{comps\_req}(w) \subseteq \mathsf{comps}(w) \subseteq \mathsf{comps\_req}(w) \cup \mathsf{comps\_opt}(w)\end{array}$$

5.4.3 Valency Constraints

Every daughter set is a finite set of nodes in the tree:

$$\forall w \in \mathcal{V},\ \forall \rho \in \mathit{Roles}, \quad \rho(w) \subseteq \mathcal{V}$$

The second principle of well-formedness requires that a complement daughter set $\rho(w)$ be non-empty only when ρ appears in w's valency. Additionally, the first principle states that, when it is non-empty, the complement daughter set $\rho(w)$ must be a singleton:

$$\begin{array}{l}\forall \rho \in \mathit{Comps} \\ \qquad\quad |\rho(w)| \leq 1 \\ \qquad \wedge \quad |\rho(w)| = 1 \quad \equiv \quad \rho \in \mathsf{comps}(w)\end{array}$$

In practice, the equivalence above will be enforced using *reified constraints* which are explained in Section 6.8.

5.4.4 Role Constraints

For each role $\rho \in \mathit{Roles}$ there is a corresponding binary predicate Γ_ρ. The third principle of well-formedness requires that whenever the dependency tree contains an edge $w \xrightarrow{\rho} w'$, then the grammatical condition $\Gamma_\rho(w, w')$ must hold in the tree. Therefore the tree must satisfy the proposition below:

$$\forall w, w' \in \mathcal{V},\ \forall \rho \in \mathit{Roles}, \quad w' \in \rho(w) \Rightarrow \Gamma_\rho(w, w')$$

In practice, the proposition will be enforced by creating a disjunctive propagator for each triple (w, w', ρ):

$$w' \in \rho(w) \wedge \Gamma_\rho(w, w') \quad \textbf{or} \quad w' \notin \rho(w)$$

For illustration, let's consider some examples of $\Gamma_\rho(w, w')$.

Subject. The subject of a finite verb must be either a noun or a pronoun, it must agree with the verb, and must have nominative case. We write NOM for the set of agreement tuples with nominative case:

$$\begin{array}{rl}\Gamma_{\mathsf{subject}}(w, w') \equiv & \mathsf{cat}(w') \in \{\mathtt{n}, \mathtt{pro}\} \\ \wedge & \mathsf{agr}(w) = \mathsf{agr}(w') \\ \wedge & \mathsf{agr}(w') \in \textsc{nom}\end{array}$$

Adjective. An adjective may modify a noun and must agree with it:

$$\Gamma_{\mathsf{adj}}(w,w') \equiv \begin{array}{ll} & \mathsf{cat}(w) = \mathtt{n} \\ \wedge & \mathsf{cat}(w') = \mathtt{adj} \\ \wedge & \mathsf{agr}(w) = \mathsf{agr}(w') \end{array}$$

5.4.5 Treeness Constraints

Our formal framework simply stipulated that models should be trees. In the constraint model, we provide an explicit axiomatization of this notion. A tree is a directed acyclic graph, where every node has a unique incoming edge, except for a distinguished node, called the root, which has none.

To support our axiomatization, we introduce two new variables $\mathsf{mother}(w)$ and $\mathsf{daughters}(w)$ for each node w. Again to avoid the problem that mother might be undefined at the root, we make it denote a set:

$$\begin{array}{r} \mathsf{mother}(w) \subseteq \mathcal{V} \\ \mathsf{daughters}(w) \subseteq \mathcal{V} \end{array}$$

$\mathsf{mother}(w)$ denotes the set of mothers of w and $\mathsf{daughters}(w)$ the set of its immediate daughters. To enforce that a node has at most one mother, we pose:

$$|\mathsf{mother}(w)| \leq 1$$

The set of immediate daughters of w is simply defined as the union of its daughter sets:

$$\mathsf{daughters}(w) = \bigcup_{\rho \in \mathit{Roles}} \rho(w)$$

w is a mother of w' iff w' is a daughter of w:

$$w \in \mathsf{mother}(w') \quad \equiv \quad w' \in \mathsf{daughters}(w)$$

We could introduce a variable to denote the root, but instead we introduce the variable ROOTSET to denote the singleton containing the root.

$$\begin{array}{c} \textsc{rootset} \subseteq \mathcal{V} \\ |\textsc{rootset}| = 1 \end{array}$$

Now, w is root iff it has no mother:

$$w \in \textsc{rootset} \quad \equiv \quad |\mathsf{mother}(w)| = 0$$

So far, we have enforced that every node (except the root) has a unique mother, but it could still have more than one incoming edge from the same mother. We can exclude this case by stating that the daughter sets for all nodes together with the root set form a partition of the input words:

$$\mathcal{V} = \textsc{rootset} \uplus \biguplus_{\substack{w \in \mathcal{V} \\ \rho \in \mathit{Roles}}} \rho(w)$$

In order to guarantee well-formedness, we still need to enforce acyclicity.

5.4.6 Yield and Acyclicity

The yield of a node is the set of nodes reachable through the transitive closure of complement and modifier edges, i.e. by traversing 0 or more dependency edges. We distinguish between yield and *strict* yield and introduce the corresponding variables $\mathsf{yield}(w)$ and $\mathsf{yieldS}(w)$. The strict yield of w is the set of all descendents of w that can be reached by traversing 1 or more dependency edges. The yield of w is obtained by adding w to its strict yield. In order to enforce acyclicity, we must require that w does not occur in its strict yield. Therefore it suffices to define the yield of w as being partitioned by $\{w\}$ and its strict yield:

$$\mathsf{yield}(w) = \{w\} \uplus \mathsf{yieldS}(w)$$

It remains to define the strict yield. For this purpose, we introduce a new member of the family of selection constraints: the *selection union constraint*:

$$S = \cup\langle S_1, \ldots, S_n\rangle[\mathit{SI}]$$

where all of S, S_i, SI are finite set variables. Its declarative semantics are:

$$S = \bigcup_{i \in \mathit{SI}} S_i$$

i.e. from the sequence $\langle S_1, \ldots, S_n\rangle$, the sets occurring at all positions indicated by SI are selected and their union is returned. See also Section 6.9.4 for further discussion of the selection union constraint.

The strict yield of w is simply the union of the yields of its daughters:

$$\mathsf{yieldS}(w) = \cup\langle \mathsf{yield}(w_1), \ldots, \mathsf{yield}(w_n)\rangle[\mathsf{daughters}(w)]$$

The use of the selection union constraint improves and simplifies the formulation in [**?**].

5.4.7 Solving the CSP

We have now completely specified our constraint model. The CSP is obtained by collecting the constraints that its stipulates and then solving in terms of its variables: we need to look for assignments to the variables of the constraint model that satisfy its constraints. Here is the simple strategy that we follow:

- First, determine assignments to the daughter set variables. In other words, we apply a distribution strategy on:

$$\{\rho(w) \mid \rho \in \mathit{Roles},\ w \in \mathcal{V}\}$$

- Second, we make sure that a lexicon entry has really been selected for each word. I.e. we apply a distribution strategy on:

$$\{\mathsf{entryindex}(w) \mid w \in \mathcal{V}\}$$

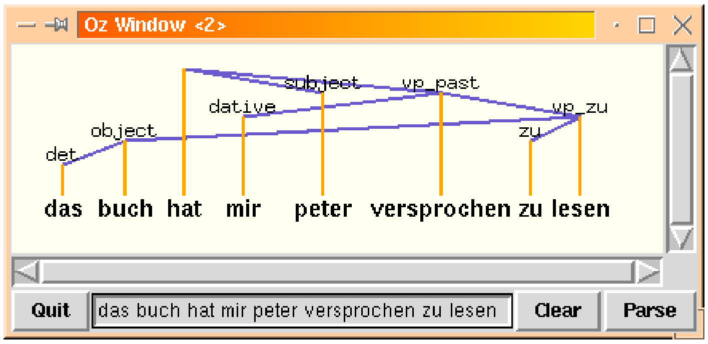
This is sufficient to determine assignments to all variables.

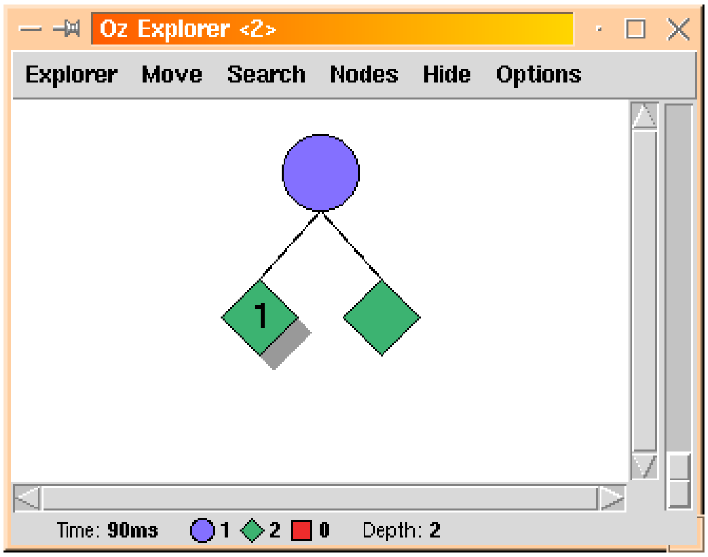
5.5 Implementation

In this section, we develop an Oz implementation of the constraint model presented in the preceding section. It consists of several functors and reuses functor `Encode` developed in Chapter 3.

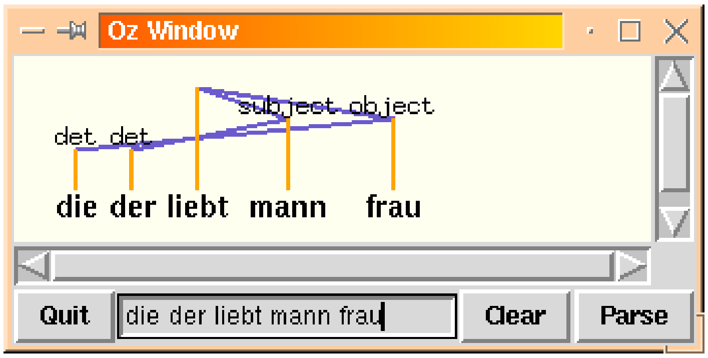
5.5.1 Demo Session

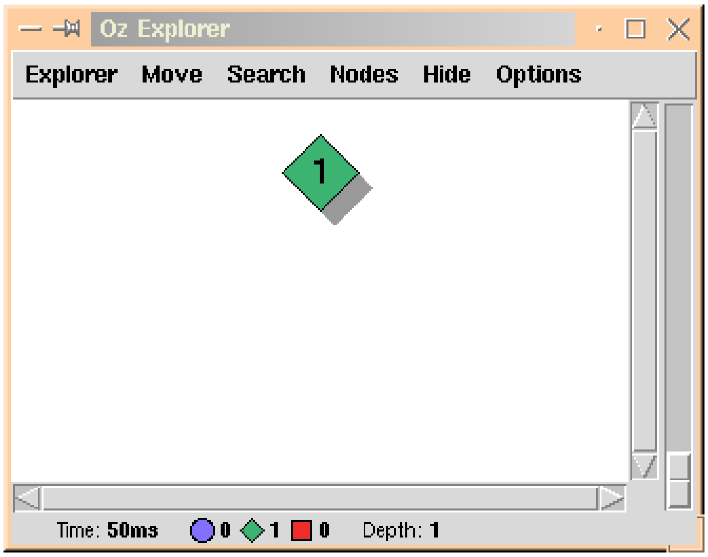
You can try the demo application at `http://www.ps.uni-sb.de/~duchier/esslli-2000/DG`. Note that in order to run the demo application, you will need to have package `mogul:/duchier/select` installed at your site (see Section 5.5.5). When the demo application starts, it pops up a window with an entry box where you can type in a your input. Let's illustrate this with the example sentence which which we started this chapter: *das Buch hat mir Peter versprochen zu lesen*. After you type in the sentence, click the `Parse` button and an explorer window pops up showing you the search tree for all possible parses. You can click on any node in this search tree to see a graphical display of the current state of the corresponding, possibly partial, dependency tree.



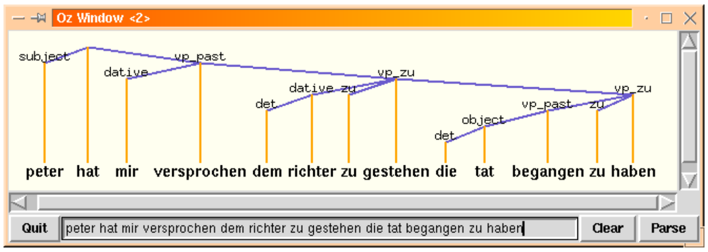


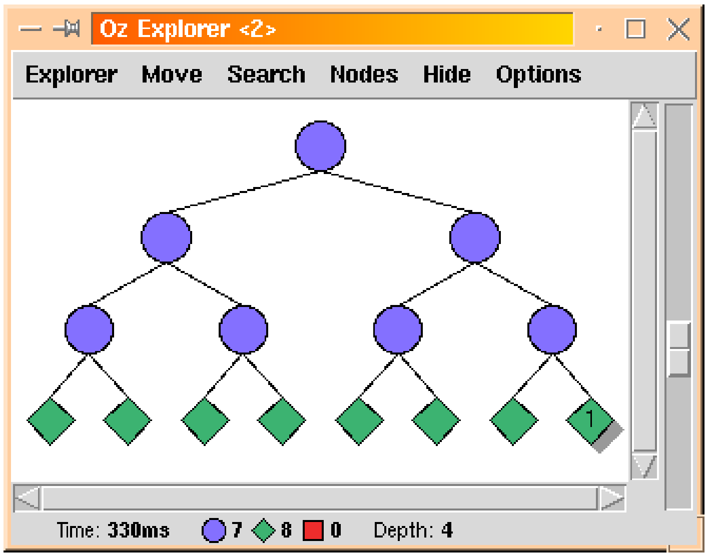
There are more possible solutions than you might expect, simply because we have no word order restrictions. This can be seen clearly if we deliberately mix up the words as in *die der liebt Mann Frau*:





Here is another example inspired by Reape's article [?]: *Peter hat mir versprochen, dem Richter zu gestehen, die Tat begangen zu haben.*





5.5.2 `Entry.oz`

In this section, we develop module `Entry` (file `Entry.oz`) whose purpose is to encode lexicon entries. It imports module `Encode` which we developed in Section 3.3.2.

56a ⟨**Entry.oz** 56a⟩≡
```
functor
import
   Encode FS
export
   ⟨Entry exports 56b⟩
define
   ⟨Entry domain products 56c⟩
   ⟨Entry encoder of lexicon entry 58f⟩
end
```

5.5.2.1 Agreement

This is the same notion of agreement as introduced in Section 3.3. It involves gender, number, person, case, and definiteness of quantifier.

56b ⟨**Entry exports** 56b⟩≡ 57a▷
```
Agreement
```

56c ⟨**Entry domain products** 56c⟩≡ 57b▷
```
Agreement = {New Encode.domainProduct
                 init([[masc fem neut]
                       [sing plur]
                       [1 2 3]
                       [nom acc dat gen]
                       [def indef none]])}
```

5.5.2.2 Category

A word may have one of the following categories: noun (`n`), pronoun (`pro`), infinitive verb (`vinf`), finite verb (`vfin`), past participle (`vpast`), determiner (`det`), particle (`part`, e.g. 'zu' in 'zu lesen'), separable verb prefix (`vpref`, e.g. 'ein' in 'einkaufen', i.e 'ich kaufe ein'), adjective (`adj`), adverb (`adv`), preposition (`prep`, e.g. 'mit').

57a ⟨**Entry exports** 56b⟩+≡ ◁56b 57c▷

```
Category
```

57b ⟨**Entry domain products** 56c⟩+≡ ◁56c 57d▷

```
Category = {New Encode.domainProduct
                init([[n pro vinf vfin vpast det part vpref adj adv prep]])}
```

5.5.2.3 Roles

We support the following types of complements: determiner (`det`), subject (`subject`), nominative (`nominative`, e.g. 'er ist ein Mann'), object (`object`), dative (`dative`), zu particle (`zu`, e.g. 'zu lesen'), separable verb prefix (`vpref`, e.g. 'ein' in 'ich kaufe ein'), infinitive verb phrase with zu (`vp_zu`, e.g. 'ich verspreche zu lesen'), past participle (`vp_past`, e.g. 'ich habe gelesen'), infinitive verb phrase without zu (`vp_inf`, e.g. 'ich will lesen').

57c ⟨**Entry exports** 56b⟩+≡ ◁57a 57e▷

```
ComplementRoles
```

57d ⟨**Entry domain products** 56c⟩+≡ ◁57b 57f▷

```
ComplementRoles = [det subject nominative object dative
                   zu vpref vp_zu vp_past vp_inf]
```

We also support the following types of modifiers: adjective (`adj`), adverb (`adv`), prepositional noun phrase (`pp_np`, e.g. 'mit dem Buch').

57e ⟨**Entry exports** 56b⟩+≡ ◁57c 57g▷

```
ModifierRoles AllRoles
```

57f ⟨**Entry domain products** 56c⟩+≡ ◁57d 57h▷

```
ModifierRoles = [adj adv pp_np]
AllRoles      = {Append ComplementRoles ModifierRoles}
```

We can now express the domain of all roles, as well as compute the sets of complement roles and of modifier roles:

57g ⟨**Entry exports** 56b⟩+≡ ◁57e 58a▷

```
Roles Complements Modifiers
```

57h ⟨**Entry domain products** 56c⟩+≡ ◁57f 58b▷

```
Roles       = {New Encode.domainProduct init([AllRoles])}
Complements = {Roles encode(ComplementRoles $)}
Modifiers   = {Roles encode(ModifierRoles $)}
```

5.5.2.4 Verb Prefixes And Marks

In German, the same base verb can be modified by many possible verb prefixes, e.g. ‘einkaufen’, ‘abkaufen’, ‘auskaufen’, etc. In order to get good propagation, we also need to encode the set of possible verb prefixes. Here we only consider ‘ein’, but you could add as many as you want. The treatment of separable verb prefixes is an extension to the constraint model we presented earlier, but it is fairly straightforward.

58a ⟨**Entry exports** 56b⟩+≡ ◁57g 58c▷

```
Vprefixes
```

58b ⟨**Entry domain products** 56c⟩+≡ ◁57h 58d▷

```
Vprefixes = {New Encode.domainProduct init([[ein]])}
```

Here are other aspects of verbs which we have not considered earlier: (1) a verb may require either ‘haben’ or ‘sein’ as an auxiliary. (2) a separable verb prefix is not always separated (e.g. ‘einkaufen’). (3) the zu particle is not always separated from the verb (e.g. ‘einzukaufen’). In order to be able to represent these details in the lexicon, we introduce a domain of marks:

58c ⟨**Entry exports** 56b⟩+≡ ◁58a 58e▷

```
Marks
```

58d ⟨**Entry domain products** 56c⟩+≡ ◁58b

```
Marks = {New Encode.domainProduct init([[zu vpref haben sein]])}
```

5.5.2.5 Encoder of Lexicon Entry

A lexicon entry is specified by a record where each feature maps to a descriptor for the corresponding domain product. A lexicon entry must have at least feature `cats`. All other features are optional and the encoder provides the obvious default value.

Feature `agrs` provides a descriptor for a set of agreement tuples. The default value is the set of all agreement tuples. Feature `comps_req` is a list of required complement roles (default empty). Feature `comps_opt` is a list of optional complement roles (default empty). `vpref` is a list of verb prefixes (default empty) and indicates that the full form of the verb has one of these prefixes. `aux` is a list (default empty) of at most one of `haben` or `sein` and indicates that the word is a form of one of these auxiliaries. `marks` is a list of marks (default empty): this list contains `zu` if the particle is part of the word, `vpref` if the separable prefix is not separated, `haben` (resp. `sein`) if it requires ‘haben’ (resp. ‘sein’) as an auxiliary.

function `LexEncode` takes a specifier for a lexicon entry and returns the corresponding (encoded) lexicon entry. For simplicity in the parser, the lexicon entry contains lower and upper bounds for the set of complements rather than the sets of required and optional complements.

58e ⟨**Entry exports** 56b⟩+≡ ◁58c

```
LexEncode
```

58f ⟨**Entry encoder of lexicon entry** 58f⟩≡

```
fun {LexEncode Desc}
   DescCat = Desc.cats
   DescAgr = {CondSelect Desc agrs [nil]}
   DescReq = {CondSelect Desc comps_req nil}
   DescOpt = {CondSelect Desc comps_opt nil}
   DescVpf = {CondSelect Desc vpref nil}
   DescMrk = {CondSelect Desc marks nil}
   DescAux = {CondSelect Desc aux   nil}
   %%
   Cats    = {Category  encode(DescCat $)}
   Agrs    = {Agreement encode(DescAgr $)}
   Reqs    = {Roles     encode(DescReq $)}
   Opts    = {Roles     encode(DescOpt $)}
   Vpref   = {Vprefixes encode(DescVpf $)}
   Zmrks   = {Marks     encode(DescMrk $)}
   Aux     = {Marks     encode(DescAux $)}
   %%
   CompsLo = Reqs
   CompsHi = {FS.union Reqs Opts}
in
   lex(cats     : Cats
       agrs     : Agrs
       comps_lo : CompsLo
       comps_hi : CompsHi
       vpref    : Vpref
       marks    : Zmrks
       aux      : Aux)
end
```

5.5.3 Lexicon.oz

Functor `Lexicon` (file `Lexicon.oz`) imports module `Entry` defined in the previous section, and exports function `Get` which takes an atom as an argument (representing the full form of a word) and returns the corresponding list of lexicon entries, or raises exception `unknownword(Word)` if the word does not appear in the lexicon.

59a ⟨**Lexicon.oz** 59a⟩≡

```
functor
import Entry
export Get
define
   Lexicon = {Dictionary.new}
   proc {PUT Word Spec}
      {Dictionary.put Lexicon Word
       {Entry.lexEncode Spec}|{Dictionary.condGet Lexicon Word nil}}
   end
   fun {Get Word}
      try Lexicon.Word
      catch _ then
```

```
                raise unknownword(Word) end
             end
          end
          ⟨Lexicon entries 60a⟩
       end
```

5.5.3.1 Proper Names

In German, proper names may take a determiner: 'der Peter', 'die Maria'.

60a ⟨Lexicon entries 60a⟩≡ 60b▷

```
{PUT peter
 lex(cats            : [n]
     agrs            : [[masc sing 3 [nom acc dat]]]
     comps_opt       : [det])}
{PUT peters
 lex(cats            : [n]
     agrs            : [[masc sing 3 gen]]
     comps_opt       : [det])}
{PUT maria
 lex(cats            : [n]
     agrs            : [[fem sing 3 [nom acc dat]]]
     comps_opt       : [det])}
{PUT marias
 lex(cats            : [n]
     agrs            : [[fem sing 3 gen]]
     comps_opt       : [det])}
```

5.5.3.2 Pronouns

Here is a sample of pronoun entries:

60b ⟨Lexicon entries 60a⟩+≡ ◁60a 61a▷

```
{PUT ich
 lex(cats            : [pro]
     agrs            : [[sing 1 nom]])}
{PUT mich
 lex(cats            : [pro]
     agrs            : [[sing 1 acc]])}
{PUT mir
 lex(cats            : [pro]
     agrs            : [[sing 1 dat]])}
{PUT du
 lex(cats            : [pro]
     agrs            : [[sing 2 nom]])}
{PUT dich
 lex(cats            : [pro]
     agrs            : [[sing 2 acc]])}
{PUT dir
 lex(cats            : [pro]
     agrs            : [[sing 2 dat]])}
```

5.5.3.3 Common Nouns

We only show ‘Mann’ for illustration.

61a ⟨**Lexicon entries** 60a⟩+≡ ◁60b 61b▷

```
{PUT mann
 lex(cats              : [n]
     agrs              : [[masc 3 sing [nom acc dat]]]
     comps_opt         : [det])}
{PUT mannes
 lex(cats              : [n]
     agrs              : [[masc 3 sing gen]]
     comps_opt         : [det])}
{PUT männer
 lex(cats              : [n]
     agrs              : [[masc 3 plur [nom acc gen]]]
     comps_opt         : [det])}
{PUT männern
 lex(cats              : [n]
     agrs              : [[masc 3 plur dat]]
     comps_opt         : [det])}
```

5.5.3.4 Determiners

We only show 4 entries for illustration:

61b ⟨**Lexicon entries** 60a⟩+≡ ◁61a 61c▷

```
{PUT der
 lex(cats              : [det]
     agrs              : [[def [[masc sing 3 nom]
                                [fem  sing 3 [dat gen]]
                                [     plur 3 gen]]]]])}
{PUT den
 lex(cats              : [det]
     agrs              : [[def [[masc sing 3 acc]
                                [     plur 3 dat]]]]])}

{PUT ein
 lex(cats              : [det]
     agrs              : [[indef [[masc sing 3 nom]
                                  [neut sing 3 [nom acc]]]]]])}
{PUT einen
 lex(cats              : [det]
     agrs              : [[indef [[masc sing 3 acc]]]]])}
```

5.5.3.5 Verbs

Here is a selection of verbal entries:

61c ⟨**Lexicon entries** 60a⟩+≡ ◁61b 63a▷

```
{PUT lieben
 lex(cats            : [vinf]
     comps_opt       : [zu object])}
{PUT liebe
 lex(cats            : [vfin]
     agrs            : [[1 sing nom]]
     comps_req       : [subject]
     comps_opt       : [object])}
{PUT geliebt
 lex(cats            : [vpast]
     comps_opt       : [object]
     marks           : [haben])}

{PUT laufen
 lex(cats            : [vinf]
     comps_opt       : [zu])}
{PUT laufe
 lex(cats            : [vfin]
     agrs            : [[1 sing nom]])}
{PUT gelaufen
 lex(cats            : [vpast]
     marks           : [sein])}


{PUT verspreche
 lex(cats            : [vfin]
     agrs            : [[1 sing nom]]
     comps_req       : [subject]
     comps_opt       : [object dative])}
{PUT verspreche
 lex(cats            : [vfin]
     agrs            : [[1 sing nom]]
     comps_req       : [subject vp_zu]
     comps_opt       : [dative])}

{PUT kaufen
 lex(cats            : [vinf]
     comps_opt       : [zu object dative])}
{PUT kaufe
 lex(cats            : [vfin]
     agrs            : [[1 sing nom]]
     comps_req       : [subject]
     comps_opt       : [object dative])}
{PUT gekauft
 lex(cats            : [vpast]
     comps_opt       : [object dative]
     marks           : [haben])}

{PUT einkaufen
```

```
 lex(cats              : [vinf]
     comps_opt         : [zu object dative]
     marks             : [vpref])}
{PUT einzukaufen
 lex(cats              : [vinf]
     comps_opt         : [object dative]
     marks             : [vpref zu])}
{PUT kaufe
 lex(cats              : [vfin]
     agrs              : [[1 sing nom]]
     vpref             : [ein]
     comps_req         : [subject vpref]
     comps_opt         : [object dative])}
{PUT einkaufe
 lex(cats              : [vfin]
     agrs              : [[1 sing nom]]
     comps_req         : [subject]
     comps_opt         : [object dative]
     marks             : [vpref])}
{PUT eingekauft
 lex(cats              : [vpast]
     comps_opt         : [object dative]
     marks             : [vpref])}
```

5.5.3.6 Auxiliaries

Auxiliaries can also be used as normal verbs:

63a ⟨**Lexicon entries** 60a⟩+≡ ◁61c 64a▷

```
{PUT haben
 lex(cats              : [vinf]
     comps_opt         : [zu object])}
{PUT habe
 lex(cats              : [vfin]
     agrs              : [[1 sing nom]]
     comps_req         : [subject]
     comps_opt         : [object])}

{PUT haben
 lex(cats              : [vinf]
     aux               : [haben]
     comps_opt         : [zu]
     comps_req         : [vp_past])}
{PUT habe
 lex(cats              : [vfin]
     agrs              : [[1 sing nom]]
     aux               : [haben]
     comps_req         : [subject vp_past])}

{PUT sein
```

```
 lex(cats              : [vinf]
     comps_opt         : [zu nominative])}
{PUT bin
 lex(cats              : [vfin]
     agrs              : [[1 sing nom]]
     comps_req         : [subject]
     comps_opt         : [nominative])}

{PUT sein
 lex(cats              : [vinf]
     aux               : [sein]
     comps_opt         : [zu]
     comps_req         : [vp_past])}
{PUT bin
 lex(cats              : [vfin]
     agrs              : [[1 sing nom]]
     aux               : [sein]
     comps_req         : [subject vp_past])}
```

5.5.3.7 Modals

A taste of modal verbs.

64a ⟨**Lexicon entries** 60a⟩+≡ ◁63a 64b▷

```
{PUT wollen
 lex(cats              : [vinf]
     comps_opt         : [zu object])}
{PUT wollen
 lex(cats              : [vinf]
     comps_opt         : [zu]
     comps_req         : [vp_inf])}
{PUT will
 lex(cats              : [vfin]
     agrs              : [[[1 3] sing nom]]
     comps_req         : [subject]
     comps_opt         : [object])}
{PUT will
 lex(cats              : [vfin]
     agrs              : [[[1 3] sing nom]]
     comps_req         : [subject vp_inf])}
```

5.5.3.8 Adjectives

A sample of adjective entries:

64b ⟨**Lexicon entries** 60a⟩+≡ ◁64a 65a▷

```
{PUT schöne
 lex(cats              : [adj]
     agrs              : [[none  [nom acc] [fem plur]]
                          [def   sing [nom [acc [neut fem]]]]
```

```
                                [indef plur [nom acc]]])}
        {PUT schönen
         lex(cats              : [adj]
             agrs              : [[none   [[masc sing [acc gen]]
                                           [fem  sing gen]
                                           [plur dat]]]
                                  [def    [plur dat gen [masc sing acc]]]
                                  [indef sing [dat gen [masc acc]]]]])}
```

5.5.3.9 Miscellaneous

Here: 1 verb prefix, 1 particle, 1 preposition, and 1 adverb:

65a ⟨Lexicon entries 60a⟩+≡ ◁64b

```
{PUT ein
 lex(cats              : [vpref])}
{PUT zu
 lex(cats              : [part])}
{PUT mit
 lex(cats              : [prep]
     comps_req         : [dative])}
{PUT heute
 lex(cats              : [adv])}
```

5.5.4 Gamma.oz

Functor Gamma (file Gamma.oz) imports module Entry and exports a binary predicate for each possible role.

65b ⟨Gamma.oz 65b⟩≡

```
functor
import FS
   Entry(
      category   : Category
      agreement  : Agreement
      marks      : Marks
      vprefixes  : Vprefixes
      )
export
   ⟨Gamma exports 66b⟩
define
   ⟨Gamma variables 66a⟩
   ⟨Gamma predicates 66c⟩
end
```

5.5.4.1 Variables

Here, we define some variables that will be used when writing roles predicates. For example variable CATS_NP denotes the encoding of the set of categories [n pro], and CAT_DET denotes the integer encoding category det.

66a ⟨Gamma variables 66a⟩≡

```
CATS_NP     = {Category encode([n pro] $)}
CATS_V      = {Category encode([vfin vinf vpast] $)}
CATS_NPV    = {FS.union CATS_NP CATS_V}
CAT_DET     = Category.toint.det
CAT_PART    = Category.toint.part
CAT_VPREF   = Category.toint.vpref
CAT_VFIN    = Category.toint.vfin
CAT_VINF    = Category.toint.vinf
CAT_VPAST   = Category.toint.vpast
CAT_N       = Category.toint.n
CAT_ADJ     = Category.toint.adj
CAT_ADV     = Category.toint.adv
CAT_PREP    = Category.toint.prep

AGRS_NOM     = {Agreement encode([nom]  $)}
AGRS_ACC     = {Agreement encode([acc]  $)}
AGRS_DAT     = {Agreement encode([dat]  $)}

MARK_ZU     = Marks.toint.zu
MARK_VPREF  = Marks.toint.vpref
```

5.5.4.2 Predicates

We will only show a few of the role predicates. For example, a subject must be a noun or pronoun, must agree with its mother (the finite verb), and must have nominative case.

66b ⟨Gamma exports 66b⟩≡ 66d▷

```
Subject
```

66c ⟨Gamma predicates 66c⟩≡ 66e▷

```
proc {Subject Mother Daughter}
   {FS.include Daughter.cat CATS_NP}
   Mother.agr = Daughter.agr
   {FS.include Daughter.agr AGRS_NOM}
end
```

The ‘zu’ particle must have category `part`, it must be the word `zu`, its mother should not already contain a particle (as in ‘einkaufen’, i.e. ‘zu einzukaufen’ is illegal), its mother should also not contain a verb prefix (i.e. ‘zu einkaufen’ is illegal).

66d ⟨Gamma exports 66b⟩+≡ ◁66b 67a▷

```
Zu
```

66e ⟨Gamma predicates 66c⟩+≡ ◁66c 67b▷

```
proc {Zu Mother Daughter}
   Daughter.cat=CAT_PART
   Daughter.word = 'zu'
   {FS.exclude MARK_ZU    Mother.marks}
```

```
    {FS.exclude MARK_VPREF Mother.marks}
end
```

Now let's consider a complement that is an infinitive VP with zu particle. The predicate simply states that the daughter must be an infinitive verb and that its `haszu` feature must be true (i.e. equal to 1, as opposed to false, i.e. equal to 0). We use this feature to conveniently cover both the case when the daughter is of the form 'einzukaufen' (i.e. zu attached) and 'zu kaufen' (i.e. zu separated). The `haszu` feature is defined in module `Parser`.

67a ⟨Gamma exports 66b⟩+≡ ◁66d 67c▷

```
Vp_zu
```

67b ⟨Gamma predicates 66c⟩+≡ ◁66e 67d▷

```
proc {Vp_zu Mother Daughter}
   Daughter.cat=CAT_VINF
   Daughter.haszu=1
end
```

For and adjective edge, the mother must be a noun, the daughter an adjective and they must agree.

67c ⟨Gamma exports 66b⟩+≡ ◁67a

```
Adj
```

67d ⟨Gamma predicates 66c⟩+≡ ◁67b

```
proc {Adj Mother Daughter}
   Mother.cat=CAT_N
   Daughter.cat=CAT_ADJ
   Mother.agr=Daughter.agr
end
```

5.5.5 `Parser.oz`

Functor `Parser` (file `Parser.oz`), imports modules `Entry`, `Lexicon`, and `Gamma` defined previously, as well as module `Select`. Module `Select` is neither part of this application nor of the Mozart distribution. Rather it is a 3rd party package provided by me. Like all 3rd party packages, it is available through the MOGUL repository[1]. In order to run this application you need to download and install the package known in MOGUL by the id `mogul:/duchier/select`. See Section 6.9.1 for further details.

The module exports function `ParsePredicate` which takes a list of words (input phrase) as argument and returns a predicate appropriate for encapsulated search (i.e. as an argument to e.g. `ExploreAll`).

67e ⟨`Parser.oz` 67e⟩≡

[1] http://www.mozart-oz.org/mogul/

```
functor
import
   FD FS
   Entry(
      category   : Category
      agreement  : Agreement
      roles      : Roles
      marks      : Marks
      allRoles   : AllRoles
      complementRoles:ComplementRoles)
   Lexicon(get)
   Gamma
   Select(fs fd union) at 'x-ozlib://duchier/cp/Select.ozf'
export
   ParsePredicate
define
   ⟨Parser variables 68a⟩
   ⟨Parser helper functions 68b⟩
   ⟨Parser MakeNode 69a⟩
   ⟨Parser ParsePredicate 71a⟩
end
```

5.5.5.1 Variables And Helper Functions

A couple of variables and obvious helper functions to be used in the rest of the parser.

68a ⟨Parser variables 68a⟩≡

```
MARK_ZU  = Marks.toint.zu
CAT_VINF = Category.toint.vinf
```

68b ⟨Parser helper functions 68b⟩≡

```
fun {GetYield      R} R.yield      end
fun {GetCats       R} R.cats       end
fun {GetAgrs       R} R.agrs       end
fun {GetCompsLo    R} R.comps_lo   end
fun {GetCompsHi    R} R.comps_hi   end
fun {GetVpref      R} R.vpref      end
fun {GetMarks      R} R.marks      end
fun {GetAux        R} R.aux        end
fun {GetEntryIndex R} R.entryindex end
```

5.5.5.2 MakeNode

Function `MakeNode` constructs all the representational support for a node. In our constraint model we introduced a lot of functions that map a node to a constrained variable. Here, we will simply represent a node as record with a feature for each function introduced in the constraint model (and also some additional features for convenience).

The function is invoked as

```
{MakeNode Word I AllIndices Entries RootSet}
```

where `Word` is an atom representing the full form of the word, `I` is its position in the input, `Positions` is the set of all word positions in the input (1 to n), `Entries` is the list of lexicon entries for the word, and `RootSet` is a variable denoting the singleton set containing the root (position) of the sentence.

69a ⟨**Parser MakeNode** 69a⟩≡
```
fun {MakeNode Word I Positions Entries RootSet}
   ⟨Parser MakeNode, attributes of selected lexicon entry 69b⟩
   ⟨Parser MakeNode, attributes of word 70a⟩
in
   node(
      isroot     : IS_ROOT
      word       : Word
      index      : I
      entryindex : EntryIndex
      cat        : CAT
      agr        : AGR
      comps      : COMPS
      vpref      : E_VPREF
      marks      : E_MARKS
      aux        : E_AUX
      yieldS     : YIELDS
      yield      : YIELD
      dtrsets    : DTRSETS
      daughters  : DAUGHTERS
      mother     : MOTHER
      haszu      : HAS_ZU
      role       : _
      )
end
```

We initialize `EntryIndex` to range over the possible positions in the list of `Entries`, and then we use the selection constraint repeatedly to obtain the various attributes of the selected lexicon entry.

69b ⟨**Parser MakeNode, attributes of selected lexicon entry** 69b⟩≡
```
EntryIndex EntryIndex::1#{Length Entries}

E_CATS     = {Select.fs {Map Entries GetCats   } EntryIndex}
E_AGRS     = {Select.fs {Map Entries GetAgrs   } EntryIndex}
E_COMPS_LO = {Select.fs {Map Entries GetCompsLo} EntryIndex}
E_COMPS_HI = {Select.fs {Map Entries GetCompsHi} EntryIndex}
E_VPREF    = {Select.fs {Map Entries GetVpref  } EntryIndex}
E_MARKS    = {Select.fs {Map Entries GetMarks  } EntryIndex}
E_AUX      = {Select.fs {Map Entries GetAux    } EntryIndex}
```

We define category, agreement, and complement roles just as explained in the constraint model:

70a ⟨Parser MakeNode, attributes of word 70a⟩≡ 70b▷

```
CAT CAT::Category.range  {FS.include CAT E_CATS}
AGR AGR::Agreement.range {FS.include AGR E_AGRS}
COMPS {FS.subset COMPS Roles.full}
      {FS.subset COMPS E_COMPS_HI}
      {FS.subset E_COMPS_LO COMPS}
```

For the daughter sets, we create a subrecord. Daughter set for role `R` on node `W` will be accessible as `W.dtrsets.R`. Furthermore, if `R` is a complement role, the cardinality of `W.dtrsets.R` must be 0 or 1, and it is 1 iff `R` is in the set of complement roles stipulated by the valency:

70b ⟨Parser MakeNode, attributes of word 70a⟩+≡ ◁70a 70c▷

```
DTRSETS = {List.toRecord o
           {Map AllRoles
            fun {$ R} R#{FS.subset $ Positions} end}}

for R in ComplementRoles do
   {FS.reified.include Roles.toint.R COMPS}={FS.card DTRSETS.R}
end
```

The set of immediate daughters can be computed as the union of the daughter sets. The yield can also be defined here, but the strict yield can only be initialized; it will be properly constrained in `ParsePredicate` after representations for all nodes have been constructed.

70c ⟨Parser MakeNode, attributes of word 70a⟩+≡ ◁70b 70d▷

```
DAUGHTERS = {FS.unionN DTRSETS}
YIELDS    = {FS.subset $ Positions}
YIELD     = {FS.partition [{FS.value.singl I} YIELDS]}
```

The mother set has cardinality at most 1, and the node is root precisely when the cardinality of the mother set is 0.

70d ⟨Parser MakeNode, attributes of word 70a⟩+≡ ◁70c 70e▷

```
MOTHER = {FS.subset $ Positions} {FS.cardRange 0 1 MOTHER}
IS_ROOT=({FS.card MOTHER}=:0)
{FS.reified.include I RootSet}=IS_ROOT
```

Attribute `W.haszu` is true iff the word has a zu particle attached or as a complement (this is an exclusive or). Furthermore, if this attribute is true, the word must be an infinitive verb.

70e ⟨Parser MakeNode, attributes of word 70a⟩+≡ ◁70d

```
HAS_ZU HAS_ZU::0#1
{FD.exor
 {FS.reified.include MARK_ZU E_MARKS}
 {FS.card DTRSETS.zu}
 HAS_ZU}
{FD.impl HAS_ZU CAT=:CAT_VINF 1}
```

5.5.5.3 ParsePredicate

Function `ParsePredicate` takes a list of atoms as argument representing the input words and returns a predicate appropriate for encapsulated search (e.g. as an argument to `ExploreAll`).

71a ⟨Parser ParsePredicate 71a⟩≡

```
fun {ParsePredicate Words}
   N = {Length Words}
   WordEntriesPairs
   = {Map Words fun {$ W} W#{Lexicon.get W} end}
   Positions = {FS.value.make 1#N}
   ⟨Parser ParsePredicate, ParseTree 71b⟩
in
   ParseTree
end
```

The search predicate applies exactly the distribution strategies that we described.

71b ⟨Parser ParsePredicate, ParseTree 71b⟩≡

```
proc {ParseTree Nodes}
   ⟨ParseTree create root set 71c⟩
   ⟨ParseTree create nodes 71d⟩
   ⟨ParseTree yields and role constraints 71e⟩
   ⟨ParseTree global partition 72a⟩
in
   {FS.distribute naive AllDtrSets}
   {FD.distribute ff {Map Nodes GetEntryIndex}}
end
```

The root set (the set of roots) is a singleton:

71c ⟨ParseTree create root set 71c⟩≡

```
RootSet={FS.subset $ Positions}
{FS.cardRange 1 1 RootSet}
```

Nodes for each word are created by invoking function `MakeNode`.

71d ⟨ParseTree create nodes 71d⟩≡

```
!Nodes = {List.mapInd WordEntriesPairs
          fun {$ I Word#Entries}
             {MakeNode Word I Positions Entries RootSet}
          end}
```

We can now collect the list of yields of all nodes, which allows us to properly constrain the strict yield of each node. For each node `N`, we consider every possible node `M` as a potential mother, and express the constraint that states that `M` is a mother or `N` iff `N` is a daughter of `M`. Furthermore, for each triple `(N,M,R)` where `R` is an arbitrary role, we impose a role constraint in the form of disjunctive propagator.

71e ⟨ParseTree yields and role constraints 71e⟩≡

```
Yields = {Map Nodes GetYield}
for N in Nodes do
   N.yieldS = {Select.union Yields N.daughters}
   for M in Nodes do
      {FS.reified.include M.index N.mother}=
      {FS.reified.include N.index M.daughters}
      for R in AllRoles do
         thread
            or {FS.include N.index M.dtrsets.R}
               N.role=R {Gamma.R M N}
            [] {FS.exclude N.index M.dtrsets.R}
            end
         end
      end
   end
end
```

Finally, we impose the condition that the root set together with the daughter sets of all nodes form a partition of the input.

72a ⟨**ParseTree global partition** 72a⟩≡

```
AllDtrSets =
RootSet|
{FoldL Nodes
 fun {$ L N}
    {Append {Record.toList N.dtrsets} L}
 end nil}
{FS.partition AllDtrSets Positions}
```

6

Constraint Programming Bestiary

This chapter is intended to serve as a very short reference to the concepts of constraint programming in general and the programmatic idioms of Oz in particular. Readers are encouraged to consult the Mozart documentation[1] for further reading material and tutorial about the Oz language and the Mozart system.

6.1 Constraint Store

Logic Programming (henceforth LP) has the notion of a logic variable which is either free or bound to a value. Constraint Programming (henceforth CP) has the more general notion of a constrained variable, i.e. a variable that is no longer free, but whose value is not yet fully determined. For example, we may have the following partial information about an integer variable I:

$$I \in \{1, 2, 7\}$$

meaning that I may take only one of these 3 values. Such a piece of information is called a *basic constraint*. Thus, the LP idea of a set of bindings is replaced in CP by a set of basic constraints which we call the *constraint store*.

In LP, when a variable becomes bound to a value, this binding cannot subsequently be changed (except on backtracking): only new bindings may be added. Similarly in CP, the constraint store grows and the information therein improves monotonically. For example, $I \in \{1, 2, 7\}$ might improve to become $I \in \{1, 7\}$. When only one possible value remains, e.g. $I \in \{7\}$, we say that the variable is *determined* and its value is 7, which we also write $I = 7$.

In (concurrent) CP, computations operate over a shared constraint store. The basic operations are *ask* and *tell*. A computation can *ask* whether a basic constraint is entailed (which happens when it or a stronger version of it is *told* to the store) or disentailed (i.e. its negation is entailed). The *ask* operation blocks until sufficient information has been concurrently accumulated in the store to answer the question either way. A computation can *tell* a basic constraint into the store (i.e. extend the store or improve the information already in the store, or derive a contradiction). The semantics of *ask* give us dataflow synchronization for free. Consider the statement:

[1] `http://www.mozart-oz.org/documentation/`

```
case L
of nil then {P}
[] H|T then {Q H T}
end
```

it asks whether `L=nil` is entailed. As result, the statement blocks until there is sufficient information to answer the question one way or the other. In practice, this means that the statement blocks until `L` becomes bound.

6.2 Constraint Variables

Of particular relevance to this course are finite domain (FD) variables and finite set (FS) variables.

6.2.1 FD Variables

A FD variable I denotes an integer out of a finite domain of possible values. In the store, it is represented by a basic constraint of the form e.g.:

$$I \in \{1, 2, 7\}$$

This is essentially a form of disjunction. Its declarative semantics are simply:

$$I = 1 \quad \vee \quad I = 2 \quad \vee \quad I = 7$$

FD variables are a very economical and effective way to address certain forms of ambiguity arising in computational linguistics. Any finite collection of values/objects can be encoded as a finite domain: therefore an underspecified element of this collection can be represented by a FD variable. In Chapter 3, we illustrate this idea for agreement in German.

6.2.2 FS variables

A FS variable S denotes a finite set of integers. In the store it is represented by a basic constraint providing information about lower and upper bounds:

$$\{1, 7\} \subseteq S \subseteq \{1, 4, 5, 7\}$$

Again, this a form of disjunction: S may take as value any set that contains at least 1 and 7, and at most 1, 4, 5 and 7. Sets are incredibly useful in computational linguistic applications: (1) they permit elegant and succinct axiomatizations, (2) these axiomatizations are also efficient constraint programs.

6.3 Constraints And Propagators

A constraint which is more complicated than basic constraints cannot be directly represented in the store. Instead it is represented by a propagator. A propagator is a concurrent agent that observes the store and tries to improve the information in it according to the semantics of the constraint it represents. A propagator has the declarative semantics of the constraint it represents and operational semantics that are described by inference rules.

As an example, consider the *disjointness* constraint $S_1 \parallel S_2$. Its declarative semantics is that it is satisfied only in models where S_1 and S_2 denote disjoint sets. The operational semantics of the propagator can be specified by the following two inference rules:

$$\begin{array}{lcl} S_1 \parallel S_2 \;\wedge\; i \in S_1 & \rightarrow & i \notin S_2 \\ S_1 \parallel S_2 \;\wedge\; i \in S_2 & \rightarrow & i \notin S_1 \end{array}$$

i.e. if an integer i is known to be in S_1 then it must be excluded from S_2, and reciprocally. A propagator is supposed to notice as soon as possible whether its constraint is entailed by the store. For example, if the upper bounds of S_1 and S_2 are disjoint, then the sets are necessarily disjoint. In such a case, the propagator disappears since its constraint is satisfied and it will no longer be able to improve the information in the store.

6.4 Encapsulation And Computation Spaces

In traditional LP and CP, new bindings and constraints appear and take effect globally. Oz supports a much more general idea: encapsulation. In Oz, the constraint store is an explicit 1st-class object and is called a *computation space* [**?**] [**?**] [**?**] [**?**]. A computation space is a place where computations run and their constraints are accumulated. Every computation (also called a thread) is *situated*: it takes place (is encapsulated) in a specific computation space. You can have multiple computation spaces, each with different computations and different constraints.

Oz also supports an even more powerful idea: computation spaces can be nested. Since a computation space is a 1st class object in Oz, it is possible for a thread running in space `S1` to create a new computation space `S2`. In this case, `S2` is situated, and therefore nested, in `S1`. The semantics of nested computation spaces requires that if `S2` is nested in `S1`, then all basic constraints of `S1` are inherited, i.e. visible, in `S2`. However, whenever a thread running in `S2` *tells* a basic constraint, this only takes effect in `S2` but does not affect its parent `S1`.

Furthermore, it is possible to ask whether the space `S2` is failed (i.e. an inconsistency was derived), or entailed (i.e. all its basic constraints are also present in `S1`). Thus, by creating a nested space `S2` to run a constraint `C`, and then by asking whether `S2` is entailed or disentailed, you can discover whether a non-basic constraint `C` is entailed or contradicted. For example, the condition statement below is implemented in terms of this facility:

```
cond C then {P} else {Q} end
```

A new space `S2` is created to run `C`, and the statement simply asks whether `S2` is entailed or disentailed. The statement blocks until there is sufficient information to answer the question. Nested computation spaces are used in a similar fashion to implement the disjunctive propagator.

The fact that computation spaces are 1st class values makes it possible to program search in Oz itself (see next section) and also to write tools, like the Explorer [**?**]

[?], that allow you to interactively explore the search tree and inspect the state of the constraints at any node in this tree.

6.5 Search

A major application domain for CP, and the only one that we look at in this course, is to solve *constraint satisfaction problems* (CSPs); i.e. to find assignment of values to variables such that a collection of constraints is satisfied.

The whole advantage of CP for solving CSPs rests on constraint propagation: let propagators improve the information in the store until no further improvement is possible (i.e. a fix point is reached). To improve the information in the constraint store means to reduce the number of possible values for each variable. This effectively reduces the search tree since after propagation there are fewer possible assignments to consider.

In fact, to solve a CSP, we proceed by alternating steps of propagation and distribution. If propagation is not sufficient to determine values for all variables, we may pick a variable `X` that is not yet determined, non-deterministically pick a value `v` from the set of those that remain after propagation, and assign it to the variable, i.e. tell the basic constraint `X=v`. `X=v` is new information, and again we let propagation derive as many improvements from it as possible. The non-deterministic cycle is repeated until all variables are determined.

In the preceding explanation, we postulated a non-deterministic process. This is standard in most LP and CP systems: a basic constraint `X=v` is non-deterministically chosen and added to the same global store. It may be removed and another one chosen on backtracking. Oz, however, offers an alternative. Since computation spaces are 1st class, we can make a copy and add `X=v` in the copy. This way, we don't have to backtrack on failure, we can just throw away the failed copy; we still have the original, and we can make another copy to try a different constraint, say `X=a`.

Thanks to this powerful idea, search in Oz is not built-in as in Prolog; instead it is programmed in Oz itself. The reader interested in how various search algorithms can be programmed in Oz is advised to read [?]. Search in Oz is guided by a *distribution strategy*. A distribution strategy is invoked when it is necessary to perform a distribution step. It is responsible for choosing a constraint C (for example `X=v` above). The search engine then creates 2 copies of the current space: in the first copy it adds constraint C and in the second copy it adds $\neg C$. Thus, for above example, one branch of the search would explore the consequences of $X = v$ and the other of $X \neq v$.

The Mozart system offers rich libraries with predefined search engines and distribution strategies that cover most of the usual cases. The reader is strongly advised to read [?] which is an introduction to constraint programming in Oz. For some advanced topics, see also [?] [?] [?] [?].

6.6 Search Predicates

A typical way to solve a CSP in Oz, is to express this CSP as a predicate and invoke the Explorer [?] [?] with this predicate as an argument.

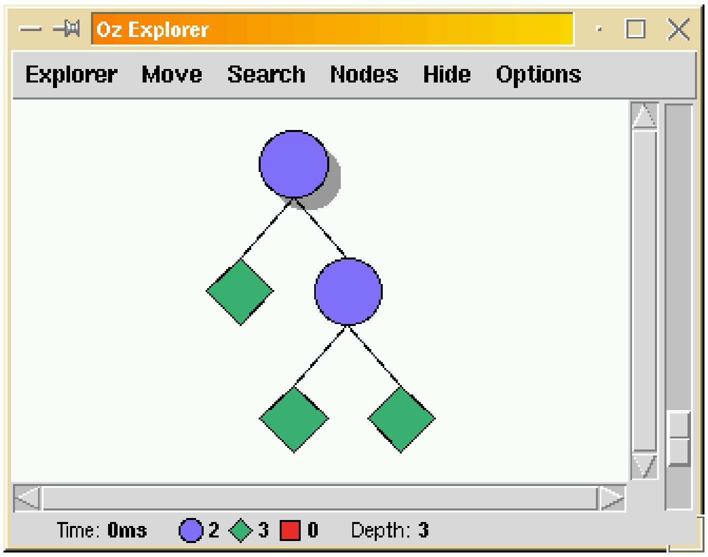
```
{ExploreAll SimpleProblem}
```

We explain now how to write such a predicate. It always follows the same pattern.

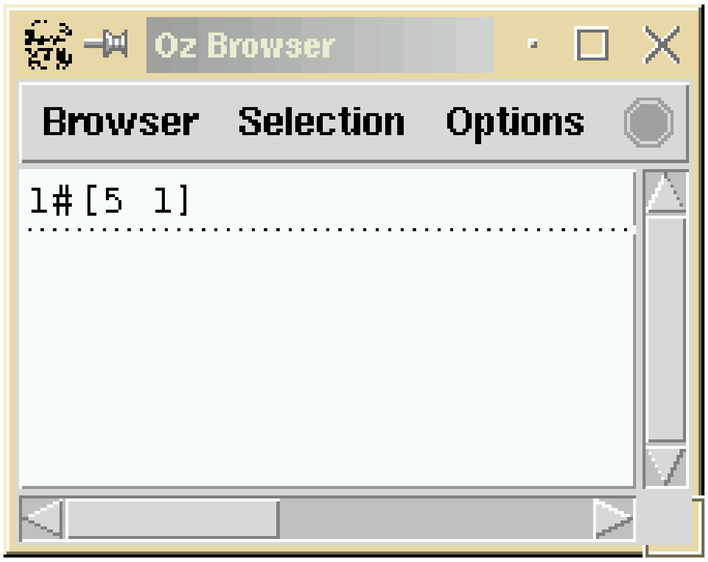
```
declare
proc {SimpleProblem Solution}
   %% declare variables
   X Y
in
   %% representation of solution
   Solution = [X Y]

   %% pose constraints of CSP
   X::1#9
   Y::1#9
   X+2*Y =: 7

   %% distribution strategy
   {FD.distribute ff [X Y]}
end
```

`SimpleProblem` is a predicate that defines what is a `Solution`: a solution is a list `[X Y]` of two integers satisfying the equation `X+2*Y =: 7`. It invokes the predefined distribution strategy `{FD.distribute ff [X Y]}`, where `ff` means *first-fail* i.e. 'choose the variable with the fewer remaining possible values, and then try each one of these values in increasing order'.

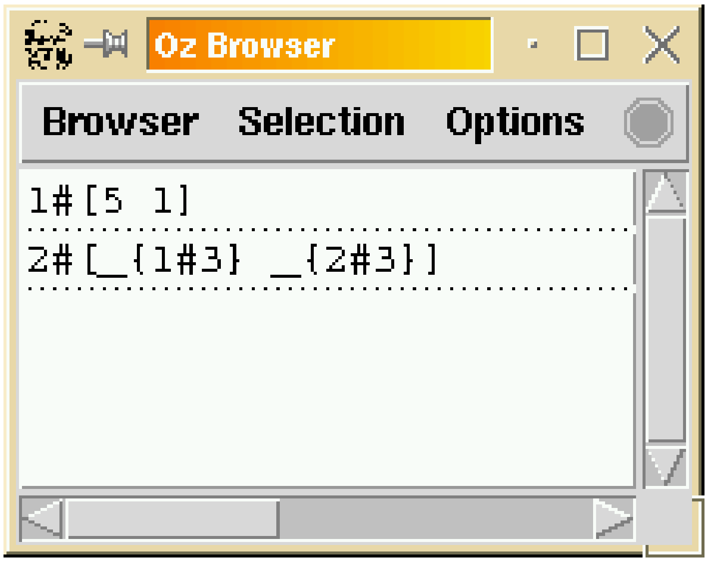
`{ExploreAll SimpleProblem}` produces the following search tree in the Explorer:



clicking on the first green diamond (a solution) shows the following in the browser:



We can also click on the second blue circle (a choice point) to observe the partial information at that point:



For a thorough tutorial on constraint programming in Oz, we recommend again to read [?].

6.7 Disjunctive Propagator

In Oz, a disjunctive propagator has the form:

```
or C1 [] C2 end
```

and its declarative semantics is simply that of disjunction. In LP, the only method available for dealing with complex disjunctions of that kind is non-determinism. Thus in Prolog, you would would write:

```
C1  ;  C2
```

Operationally, this means: first try `C1` and if that fails backtrack and try `C2` instead. This has several drawbacks: (1) it is not sound (failure to prove C is not the same as proving $\neg C$), (2) it forces the computation to commit immediately to exploring either one alternative or the other.

Early commitment is a poor strategy. It is often preferable to delay a choice until sufficient information is available to reject one of the alternatives. This is the intuition underlying the disjunctive propagator: `or C1 [] C2 end` is a propagator not a choice point. It blocks until either `C1` or `C2` becomes inconsistent with respect to the current store of basic constraints: at that point, the propagator *commits*, i.e. *reduces to*, the remaining alternative. In this way, a disjunctive propagator has the declarative semantics of sound logical disjunction, unlike Prolog's `;` operator which implements merely *negation as failure*. The operational semantics are given by the rules below, where $\mathcal{B}$ represents the current basic constraints:

$$\frac{\mathcal{B} \wedge C_1 \quad \to^* \quad \mathsf{false}}{\mathcal{B} \wedge (C_1 \;\mathbf{or}\; C_2) \quad \to \quad \mathcal{B} \wedge C_2}$$

$$\frac{\mathcal{B} \wedge C_2 \quad \to^* \quad \mathsf{false}}{\mathcal{B} \wedge (C_1 \;\mathbf{or}\; C_2) \quad \to \quad \mathcal{B} \wedge C_1}$$

This is realized by taking advantage of nested computation spaces. A disjunctive propagator `or C1 [] C2 end` creates 1 nested space to run `C1` and another to run `C2` and constantly monitors their status (either entailed or failed). When for example the space running `C1` is discovered to be failed, then the disjunctive propagator commits to the other space, i.e. it *merges* the contents of the space running `C2` with the current space. When this is done the propagator disappears (we also say that it *reduces*).

A disjunctive propagator blocks until it reduces. This is why, in Oz programs, disjunctive propagators are usually spawned in their own thread to allow the rest of the computation to proceed. In effect, this makes a disjunctive propagator into a concurrent agent:

```
thread or C1 [] C2 end end
```

A disjunctive propagator commits to one alternative only when the other becomes inconsistent. But, when neither becomes willingly inconsistent, it is often necessary, in the interest of completeness, to non-deterministically enumerate the alternatives. This can easily be achieved by introducing a *control variable* `X`:

```
or  X=1 C1  []  X=2 C2  end
```

We can now apply a distribution strategy on FD variable `X` to force exploration of the alternatives. In a typical program, you might often invoke first a distribution strategy on the variables of the CSP, and then a second distribution strategy on the control variables to make sure for the sake of completeness that all disjunctions have been decided:

```
%% distribute on the variables of the CSP
{FD.distribute ff [I1 I2 I3]}
%% distribute on the control variables
{FD.distribute ff [X1 X2]}
```

6.8 Reified Constraints

There are often cases when, instead of imposing a constraint C, we want to speak (and possibly constrain) its truth value. Let's take an example from Chapter 5: the cardinality of a daughter set $\rho(w)$ is at most 1, and it is 1 iff ρ is required by w's valency, i.e if $\rho \in \mathsf{comps}(w)$. Let b_1 stand for the truth value of $|\rho(w)| = 1$ and b_2 stand for the truth value of $\rho \in \mathsf{comps}(w)$:

$$\begin{array}{rcl} b_1 & \equiv & |\rho(w)| = 1 \\ b_2 & \equiv & \rho \in \mathsf{comps}(w) \end{array}$$

The well-formedness condition mentioned above is expressed simply by $b_1 = b_2$.

For the purpose of this section, we will write $B \equiv C$ to represent a reified constraint, where B is a FD variable representing the truth value of constraint C: 0 means false, 1 means true. The operational semantics are as follows:

- if C is entailed then $B = 1$ is inferred
- if C is inconsistent then $B = 0$ is inferred
- if $B = 1$ is entailed, then C is imposed
- if $B = 0$ is entailed, then $\neg C$ is imposed

For example, here is how reified constraints can express that exactly one of C_1, C_2, and C_3 must hold:

$$\begin{array}{l} B_1 \equiv C_1 \\ B_2 \equiv C_2 \\ B_3 \equiv C_3 \\ B_1 + B_2 + B_3 = 1 \end{array}$$

Similarly, here is how to express $C_1 \Rightarrow C_2$:

$$\begin{array}{l} B_1 \equiv C_1 \\ B_2 \equiv C_2 \\ B_1 \leq B_2 \end{array}$$

The astute reader may wonder 'why do we need a new concept? can't we express reified constraints in terms of disjunctive propagators?'. Indeed, $B \equiv C$ can also be written:

```
or B=1 C [] B=0 ~C end
```

where somehow `~C` is intended to represent the negation of `C`. What makes reified constraints attractive is that they are much more efficient. A disjunctive propagator needs to create 2 nested spaces, but a reified constraint doesn't need any.

In the libraries of the Mozart system, many constraints are also available as reified constraints (but not all). For example:

```
{FS.reified.include I S B}
```

`B` represents the truth value of `{FS.include I S}`. If `B=0`, the reified constraint reduces to `{FS.exclude I S}`.

6.9 Selection Constraints

The selection constraint is based on the intuitive notion of selecting the Ith element out of a sequence of values. We write it abstractly as follows:

$$X = \langle Y_1, \ldots, Y_n \rangle [I]$$

The notation $\langle Y_1, \ldots, Y_n \rangle [I]$ was chosen to be reminiscent of array access, i.e. subscripting. The declarative semantics is simply:

$$X = Y_I$$

that is: X is equated with the Ith variable in the sequence $\langle Y_1, \ldots, Y_n \rangle$. However, unlike functional selection which would block until I is known and then select the appropriate element, the above is a constraint that affects both X and I. If $X = Y_k$ is inconsistent, then k is removed from the domain of I. Conversely, the information about X can be improved by lifting the information common to all Y_i at positions that are still in the domain of I. We will explain this in more detail later.

The idea of the selection constraint was first introduced by [**?**], but the sequence was restricted to integer values. In [**?**], I introduced a more general form that accepts homogeneous sequences of either FD variables or FS variables and has a very efficient implementation.

Of particular interest to the linguist is the fact that the selection constraint can express covariant assignments:

$$\begin{aligned} X &= \langle X_1, \ldots, X_n \rangle [I] \\ Y &= \langle Y_1, \ldots, Y_n \rangle [I] \end{aligned}$$

These two selection constraints share the same selector I and can be regarded as realizing the *dependent* (or *named*) disjunctions shown below ([**?**], [**?**], [**?**], [**?**]) both labeled with name I:

$$\begin{array}{lcl} (X = X_1 & \vee \ldots \vee & X = X_n)_I \\ (Y = Y_1 & \vee \ldots \vee & Y = Y_n)_I \end{array}$$

Notational variants of dependent disjunctions have been used to concisely express covariant assignment of values to different features in feature structures. The selection constraint provides the same elegance, but additionally affords the benefits of efficient and effective constraint propagation.

6.9.1 `Select` Package

The selection constraint is not part of the Mozart distribution, rather it is a 3rd party package provided by me. Like all Mozart 3rd party packages, it is available from the MOGUL repository[2] in which it is known by id `mogul:/duchier/select`. In order to be able to use it, you must download and install it: this is made very easy. From the repository, you will be able to download `duchier-select-1.4.1.tgz` or whatever the name is for the most recent version of the package. To install it, simply execute the following sequence of instructions at a shell:

[2] `http://www.mozart-oz.org/mogul/`

```
tar zxf duchier-select-1.4.1.tgz
cd duchier-select-1.4.1
./configure
make
make install
```

Once installed, the package is available through URI `x-ozlib://duchier/cp/Select.ozf`. For example, in a functor you can import it as follows:

```
functor
import Select at 'x-ozlib://duchier/cp/Select.ozf'
...
end
```

and in the interactive OPI, you might obtain it as follows:

```
declare [Select] = {Module.link ['x-ozlib://duchier/cp/Select.ozf']}
```

Unfortunately, the package needs at least Mozart version 1.1.1 which is not yet officially released. You can get the current development version of Mozart from our anonymous CVS server (see the Mozart site[3] for information on how to do this). Supposing you have obtained a copy of the mozart sources from the CVS server and that this copy resides in `~/mozart`, here is what to do to build and install the Mozart system. Execute the following sequence of steps in a shell:

```
cd ~
mkdir build-mozart
cd build-mozart
~/mozart/configure
make
make install
```

You can install in a non-standard place either by using an option at configuration time:

```
~/mozart/configure --prefix=$OTHERDIR
```

or by using an argument at installation time:

```
make install PREFIX=$OTHERDIR
```

6.9.2 FD Selection Constraint

The selection constraint exists in a version where the sequence contains only FD variables. It is exported on feature `fd` of module `Select`:

```
X={Select.fd [X1 ... Xn] I}
```

[3] `http://www.mozart-oz.org/`

Here is how the information about X can be improved by this constraint. Since X must ultimately be identified with one Xi, its domain is at most the union of their domains (it cannot take other values than they can). More precisely, its domain is at most the union of the domains of the Xis at positions that are still in the domain of I (i.e. that may still be selected). In other words, if $I \in D$, and $X_i \in D_i$, then

$$X \in \bigcup_{i \in D} D_i$$

6.9.3 FS Selection Constraint

The selection constraint exists also in a version where the sequence contains only FS variables. It is exported on feature fs of module Select:

```
X={Select.fs [X1 ... Xn] I}
```

Again the information about X can be improved by propagation. The upper bound of X is at most the union of the upper bounds of the Xis for i in the domain of I. The lower bound of X is at least the intersection of the lower bounds of the Xis for i in the domain of I. In other words, if $I \in D$ and $D_i \subseteq X_i \subseteq D_i'$, then:

$$\bigcap_{i \in D} D_i \subseteq X \subseteq \bigcup_{i \in D} D_i'$$

6.9.4 FS Selection Union Constraint

The selection union constraint is the latest arrival in this family of constraints. It is exported on feature union of module Select. All its variables are sets and we write it abstractly as follows:

$$S = \cup\langle S_1, \ldots, S_n \rangle[\mathit{SI}]$$

and concretely thus:

```
S={Select.union [S1 ... Sn] SI}
```

Its declarative semantics is simply:

$$S = \bigcup_{i \in \mathit{SI}} S_i$$

Again the information about S can be improved as follows. If $D_i \subseteq S_i \subseteq D_i'$ and $D \subseteq \mathit{SI} \subseteq D'$, then:

$$\bigcup_{i \in D} D_i \subseteq S \subseteq \bigcup_{i \in D'} D_i'$$