Concurrent Constraint Programming in Oz for Natural Language Processing

Denys Duchier
Claire Gardent
Joachim Niehren
Credits

Mozart logo by Christian Lindig

License Agreement

This software and its documentation are copyrighted by the German Research Center for Artificial Intelligence (DFKI), the Swedish Institute of Computer Science (SICS), and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE AND ITS DOCUMENTATION ARE PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
Contents

I Introduction 1

1 Abstract 3

2 What is Concurrent Constraint Programming? 5
  2.1 General Ideas ................................. 5
    2.1.1 Applications .......................... 5
    2.1.2 The Problem: Combinatorial Explosion 6
    2.1.3 Interleaving Generation and Checking 7
    2.1.4 The Method: Propagate and Distribute 8
    2.1.5 What is Oz and who is Mozart? ......... 10
  2.2 Getting Started with Oz ........................ 11
    2.2.1 Installation ................................ 11
    2.2.2 Programming Environment ................ 12
  2.3 Solving a Combinatorial Problem ................ 14
    2.3.1 Bits of a Constraint Solver ............... 14
    2.3.2 Observing Propagation ................... 15
    2.3.3 Composing the Solver .................... 16
    2.3.4 Was this a good Example? ............... 17
    2.3.5 Questions ................................ 17
  2.4 Exercise ..................................... 18
  2.5 Summary ................................... 18
  2.6 Program Collection ................................ 19

3 What is Computational Linguistics? 23
  3.1 What are we doing in Syntax? ................. 23
  3.2 What are we doing in Semantics? .............. 24
  3.3 Demo Systems ................................ 24
## II Functional Programming and Context-Free Parsing

### 4 Basic Data Structures, Expressions and Statements

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Statements versus Expressions</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Values and Types</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Expressions for Values</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>Variables, Scope, and Binding</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Scoping in the Interactive User Interface</td>
<td>30</td>
</tr>
<tr>
<td>4.6</td>
<td>Inspecting Values and Types</td>
<td>30</td>
</tr>
<tr>
<td>4.7</td>
<td>Type Checks</td>
<td>31</td>
</tr>
<tr>
<td>4.8</td>
<td>Testing Equality</td>
<td>31</td>
</tr>
<tr>
<td>4.9</td>
<td>Procedures</td>
<td>32</td>
</tr>
<tr>
<td>4.10</td>
<td>Functions as Relations</td>
<td>34</td>
</tr>
<tr>
<td>4.11</td>
<td>Boolean Conditionals</td>
<td>34</td>
</tr>
<tr>
<td>4.12</td>
<td>Records</td>
<td>35</td>
</tr>
<tr>
<td>4.13</td>
<td>Tuples</td>
<td>36</td>
</tr>
<tr>
<td>4.14</td>
<td>Lists</td>
<td>36</td>
</tr>
<tr>
<td>4.15</td>
<td>Characters, Strings, and Virtual Strings</td>
<td>38</td>
</tr>
<tr>
<td>4.16</td>
<td>Loops</td>
<td>38</td>
</tr>
<tr>
<td>4.17</td>
<td>Pattern Matching and Conditionals</td>
<td>39</td>
</tr>
<tr>
<td>4.18</td>
<td>Towards State and Abstract Data Structures</td>
<td>40</td>
</tr>
<tr>
<td>4.19</td>
<td>Modules</td>
<td>40</td>
</tr>
<tr>
<td>4.20</td>
<td>Exercises</td>
<td>41</td>
</tr>
<tr>
<td>4.21</td>
<td>Program Collection</td>
<td>42</td>
</tr>
</tbody>
</table>

### 5 A Naive Parser for Context-Free Grammars

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Natural Language, Syntax and Context-Free Grammar</td>
<td>49</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Parsing</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>Bottom-up Recognition/Parsing</td>
<td>50</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Recognition as Inferential Closure</td>
<td>51</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Inferential Closure Algorithm</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>A Bottom-Up Recognizer</td>
<td>52</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Grammar Rules</td>
<td>52</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Agenda</td>
<td>53</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Bag</td>
<td>54</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Parse</td>
<td>55</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>12</td>
<td>Unification and Encapsulation</td>
<td>139</td>
</tr>
<tr>
<td>12.1</td>
<td>Encapsulating Failure</td>
<td>139</td>
</tr>
<tr>
<td>12.2</td>
<td>How to treat a Constraint as a Value?</td>
<td>139</td>
</tr>
<tr>
<td>12.3</td>
<td>Constraints as Predicates</td>
<td>140</td>
</tr>
<tr>
<td>12.4</td>
<td>Unifying Attribute Value Matrices</td>
<td>142</td>
</tr>
<tr>
<td>12.5</td>
<td>Exercises</td>
<td>142</td>
</tr>
<tr>
<td>12.6</td>
<td>Program Collection</td>
<td>144</td>
</tr>
<tr>
<td>13</td>
<td>Chart Parsing for Unification Grammars</td>
<td>147</td>
</tr>
<tr>
<td>13.1</td>
<td>Unification-based Grammars</td>
<td>147</td>
</tr>
<tr>
<td>13.1.1</td>
<td>Feature Terms</td>
<td>147</td>
</tr>
<tr>
<td>13.1.2</td>
<td>Path equations</td>
<td>149</td>
</tr>
<tr>
<td>13.1.3</td>
<td>Organising the lexicon</td>
<td>150</td>
</tr>
<tr>
<td>13.2</td>
<td>Unification-based Parsing</td>
<td>152</td>
</tr>
<tr>
<td>13.2.1</td>
<td>Structured.categories</td>
<td>152</td>
</tr>
<tr>
<td>13.2.2</td>
<td>Is Context Free Grammar Enough?</td>
<td>153</td>
</tr>
<tr>
<td>13.2.3</td>
<td>Rule Formats</td>
<td>153</td>
</tr>
<tr>
<td>13.2.4</td>
<td>Chart Parsing</td>
<td>154</td>
</tr>
<tr>
<td>13.2.5</td>
<td>A Simplistic Example</td>
<td>155</td>
</tr>
<tr>
<td>13.3</td>
<td>The Complete Package</td>
<td>158</td>
</tr>
<tr>
<td>13.4</td>
<td>Exercises</td>
<td>158</td>
</tr>
<tr>
<td>14</td>
<td>Active Chart Parsing</td>
<td>161</td>
</tr>
<tr>
<td>14.1</td>
<td>Non concurrent active chart parsing</td>
<td>161</td>
</tr>
<tr>
<td>14.2</td>
<td>Exercise</td>
<td>163</td>
</tr>
<tr>
<td>IV</td>
<td>Concurrent Constraint Programming and Semantic Under-specification</td>
<td>165</td>
</tr>
<tr>
<td>15</td>
<td>Concurrent Constraint Programming</td>
<td>167</td>
</tr>
<tr>
<td>15.1</td>
<td>General Ideas (Recalled)</td>
<td>167</td>
</tr>
<tr>
<td>15.1.1</td>
<td>Combinatoric Explosion</td>
<td>167</td>
</tr>
<tr>
<td>15.1.2</td>
<td>Constraint Store and Propagators</td>
<td>167</td>
</tr>
<tr>
<td>15.1.3</td>
<td>Distribution</td>
<td>168</td>
</tr>
<tr>
<td>15.1.4</td>
<td>Ambiguities versus Disjunction</td>
<td>168</td>
</tr>
<tr>
<td>15.2</td>
<td>Finite Domain Constraints</td>
<td>168</td>
</tr>
</tbody>
</table>
16.1 Syntax-Semantics Interface ........................................ 194
16.1.7 Exercises .. ..................................................... 195

16.2 Complexity of Dominance Constraints ......................... 195
16.2.1 Dominance Constraints ........................................ 196
16.2.2 Constraints as Graphs ......................................... 196
16.2.3 Partitioning Trees ............................................. 196
16.2.4 A Generate and Test Algorithm .............................. 197
16.2.5 Complexity of Dominance Constraints .................... 197

16.3 Solving Dominance Constraints Efficiently ................... 198
16.3.1 Sets of Nodes .................................................. 198
16.3.2 Encoding Nodes as Records .................................. 198
16.3.3 Translation to Set Constraints ............................... 199
16.3.4 Treeness Condition ........................................... 200
16.3.5 Encoding Dominance Constraints .......................... 201
16.3.6 The Solver as a Module ...................................... 202
16.3.7 Solution Predicate ............................................ 203
16.3.8 Graphical Output with DaVinci ............................. 204
16.3.9 Exercises ....................................................... 207
16.3.10 Complete Solver ............................................. 209

16.4 Summary .......................................................... 213

V Dependency Parsing with Finite Set Constraints ............... 215

17 General Ideas ....................................................... 217
17.1 What are we Doing? .............................................. 217
17.2 Standard Principles .............................................. 217
17.3 Constraining the Word Order .................................. 218
17.4 Is a Chart still Appropriate? ................................... 218
17.5 Constraining the Constituent Structure ....................... 218

18 Word-Order and Dependency Structure ......................... 221
18.1 Topological Fields ................................................ 221
18.2 Non-Projectivity .................................................. 222
18.3 Dependency Structures ......................................... 222
18.4 Word Order Constraints ........................................ 223
18.5 Role Dependency Constraints .................................. 223
18.6 Putting it together in an example ............................. 223
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Constraint-Based Dependency Parsing</td>
<td>225</td>
</tr>
<tr>
<td>19.1</td>
<td>Introduction</td>
<td>225</td>
</tr>
<tr>
<td>19.2</td>
<td>The Big Picture</td>
<td>227</td>
</tr>
<tr>
<td>19.2.1</td>
<td>Model Generation vs Model Elimination</td>
<td>227</td>
</tr>
<tr>
<td>19.2.2</td>
<td>Finite Formulations</td>
<td>228</td>
</tr>
<tr>
<td>19.2.3</td>
<td>Sets Are Wonderful</td>
<td>228</td>
</tr>
<tr>
<td>19.2.4</td>
<td>Modelization Techniques</td>
<td>228</td>
</tr>
<tr>
<td>19.3</td>
<td>Formal Framework</td>
<td>228</td>
</tr>
<tr>
<td>19.3.1</td>
<td>Dependency Grammar</td>
<td>229</td>
</tr>
<tr>
<td>19.3.2</td>
<td>Lexicon and Lexical Entries</td>
<td>229</td>
</tr>
<tr>
<td>20</td>
<td>Constraint-Based Dependency Parser: Implementation</td>
<td>231</td>
</tr>
<tr>
<td>20.1</td>
<td>DG_Parser.oz</td>
<td>231</td>
</tr>
<tr>
<td>20.1.1</td>
<td>MakeScript</td>
<td>231</td>
</tr>
<tr>
<td>20.1.2</td>
<td>Word2Node</td>
<td>234</td>
</tr>
<tr>
<td>20.1.3</td>
<td>Gamma</td>
<td>238</td>
</tr>
<tr>
<td>20.1.4</td>
<td>CheckWord</td>
<td>242</td>
</tr>
<tr>
<td>20.2</td>
<td>DG_Lexicon.oz</td>
<td>242</td>
</tr>
<tr>
<td>20.3</td>
<td>DG_Lattice.oz</td>
<td>261</td>
</tr>
<tr>
<td>20.4</td>
<td>DG_Weight.oz</td>
<td>266</td>
</tr>
<tr>
<td>20.5</td>
<td>DG_Selection.oz</td>
<td>267</td>
</tr>
<tr>
<td>20.6</td>
<td>DG_DTreeFrame.oz</td>
<td>267</td>
</tr>
<tr>
<td>20.7</td>
<td>DG_Demo.oz</td>
<td>270</td>
</tr>
<tr>
<td>VI</td>
<td>Concurrent Programming</td>
<td>273</td>
</tr>
<tr>
<td>21</td>
<td>Concurrent Programming</td>
<td>275</td>
</tr>
<tr>
<td>21.1</td>
<td>What is Concurrency</td>
<td>275</td>
</tr>
<tr>
<td>21.2</td>
<td>Threads</td>
<td>275</td>
</tr>
<tr>
<td>21.3</td>
<td>Streams and Ports</td>
<td>276</td>
</tr>
<tr>
<td>21.4</td>
<td>State and Indeterminism</td>
<td>277</td>
</tr>
<tr>
<td>21.5</td>
<td>Exception Handling</td>
<td>278</td>
</tr>
<tr>
<td>21.6</td>
<td>Mail Box Model</td>
<td>278</td>
</tr>
<tr>
<td>21.7</td>
<td>Atomicity of State Change</td>
<td>279</td>
</tr>
<tr>
<td>21.8</td>
<td>Locks</td>
<td>280</td>
</tr>
<tr>
<td>21.9</td>
<td>Exercises</td>
<td>281</td>
</tr>
</tbody>
</table>
Part I

Introduction
Abstract

This script was developed jointly with a lecture held at the departments of computer science and computational linguistics at the University of Saarbrücken. Since 1998 the course was given regularly once per year. It will be offered again starting from October 2002. As usual, we will use the next lecture to improve on what we have. Comments and proposals for improvements are still very welcome.

The lecture introduces concurrent constraint programming in Mozart/Oz and shows how to model natural language processing on basis of this paradigm. The lecture addresses students of computer science who are interested in constraint programming or Mozart/Oz and students of computational linguistics who want to learn natural language processing with modern programming technology beyond Prolog and LISP.

The lecture introduces the following topics, all of which are presented with running programs and lots of practical exercises:

- functional programming and data structures in Mozart/Oz,
- chart parsing for context free grammars,
- unification based parsing,
- concurrent constraint programming,
- constraint solving for scope underspecification in natural language semantics,
- constraint solving for parsing with dependency grammar.

The lectures presupposes as few knowledge as possible, so that it can address students of computer science and computational linguistics simultaneously.
What is Concurrent Constraint Programming?

Concurrent constraint programming is a form of constraint programming. Constraint programming provides a method for solving combinatorial problems, which comes with a well-developed technology. Combinatorial problems are traditionally formulated as logical formulas that are called constraints. Solving combinatorial problems is inherently difficult because of the disjunctive character of combinatorics.

2.1 General Ideas

Concurrent constraint programming is a technology to solve combinatorial problems. In this section, we first sketch typical applications, the underlying methodology, and the tools available in practice.

2.1.1 Applications

Typical applications of constraint programming include optimization problems of industrial relevance such as:

- scheduling,
- time tabling,
- configuration.

Recently, many new challenging applications have been investigated at universities:

- deduction and reasoning
- knowledge representation
- processing of natural language
2.1.2 The Problem: Combinatorial Explosion

The naive way of solving combinatorial problems can be paraphrased as ‘generate and test’: In a first step one enumerates all combinations from which one selects all solutions in the second step. In most cases however, ‘generate and test’ is simply not feasible. This is obvious if the set of combinations is infinite. But even if it is finite then it is usually very large, i.e. exponentially large in size of the problem description. In this case, the generation step runs into a combinatorial explosion (from which it usually returns only several billions of years later).

2.1.2.1 Constraint Satisfaction Problems

A typical example is a constraint satisfaction problem: it consists of variables $V_1, \ldots, V_n$ which respectively take values from finite domains $D_1, \ldots, D_n$, where $D_i$ is a finite set of values such as an interval of integers. The problem is to find assignments of values to the variables such that a constraint $C(V_1, \ldots, V_n)$ is satisfied.

Let’s consider the following example of 15 variables $V_1, \ldots, V_{15}$, all taking values in the domain $\{1, \ldots, 15\}$ and for which we want to find all solutions that satisfy the constraints:

\[
V_1 < V_2 \\
V_2 < V_3 \\
\ldots \\
V_{14} < V_{15}
\]

Clearly, there is only 1 solution to this problem, namely:

\[
V_1=1, \ V_2=2, \ldots, \ V_{15}=15
\]

Let’s see how various general problem solving techniques perform on this example.

2.1.2.2 Generate and Test

We can enumerate the possible assignments by picking a non-assigned variable, non-deterministically choosing a value in its domain as its assignment, and repeating until all variables are assigned values. This process spawns a tree: each inner node of this tree corresponds to a non-deterministic choice of value to assign to a variable, and the leaves are all possible complete assignments.

The leaves which satisfy the problem’s constraint are said to be solutions. Those which violate this constraint are said to be failures. We will often display a search tree graphically as shown below, where blue circles represents choice points, red squares failures, and green diamonds solutions. For convenience, a subtree whose leaves are all failures (resp. solutions) is usually abbreviated by a red (resp. green) triangle.
2.1. General Ideas

For our problem, there are 15 variables, each taking one of 15 possible values: this means there are $15^{15} = 437,893,890,380,859,375$ possible assignments. Let’s be optimistic and suppose we have a fast computer able to check $10^9$ assignments per second to decide whether each is a solution or a failure: checking all possibilities would still take approximately 14 years.

For a concrete example, let’s consider only 6 variables $V_1, \ldots, V_6$ taking values in $\{1, \ldots, 6\}$ and such that they must satisfy $V_1 < \ldots < V_n$. The generate and test method produces the following search tree with 93311 nodes:

2.1.3 Interleaving Generation and Checking

The ‘generate and test’ method generates complete assignments and only then checks whether they are solutions or failures. We can usually improve on it by interleaving generation and checking: after each choice we can check if any constraint is obviously
violated. For example, in our problem the constraints are \( V_1 < V_2, \ V_2 < V_3, \ldots, \ V_{14} < V_{15} \). Thus, as soon as we have guessed values for \( V_1 \) and \( V_2 \), we can immediately check whether \( V_1 < V_2 \) is violated or not; we don’t have to wait until values have been guessed for all other variables too. As a result, we can decide failure much earlier and the search tree become much smaller. For the example with only 6 variables, it now contains 631 nodes instead of 93311:

This strategy, often implemented using some notion of coroutines, is traditionally called ‘test and generate’. The tests are posted first, but each remains suspended until its variables have all been assigned. Generation then proceeds as before, except that a test becomes active as soon as its variables have been assigned and thus may cause failure before further levels of the search tree are generated.

2.1.4 The Method: Propagate and Distribute

By now, you should be convinced that it is simply not practical to generate the full search tree in usual combinatorial problems. Even interleaving generate and test will not scale: in the case of 15 variables it results in a search tree with 917477 nodes; now suppose we had 100 variables. Clearly this just won’t do. So what can we do instead?

The idea is to take the interleaving idea even further: we don’t want the tests to simply be passive and check whether they are satisfied or violated; rather we want them to be active and propagate constraints. In our example both \( V_1 \) and \( V_2 \) take values in \( \{1, \ldots, 15\} \). Furthermore we have the constraint that:

\[ V_1 \leq V_2 \]

This means that the value of \( V_2 \) must be at least 1 greater than that of \( V_1 \). Therefore \( V_1 \) must actually take values in \( \{1, \ldots, 14\} \) and \( V_2 \) in \( \{2, \ldots, 15\} \). We can repeat this reasoning with \( V_2 \) and \( V_3 \), etc, until \( V_{14} \) and \( V_{15} \), at which point we obtain the conclusion that \( V_{15} \) must take values in \( \{15\} \). In other words the only possible value for \( V_{15} \) is 15. By iterating this process we deterministically arrive at the conclusion that \( V_1=1, \ldots, V_{15}=15 \). Thus the search tree now contains a single node:
In general, however, we may still have to perform search, but the idea is to first derive as much as possible through deterministic inference using the available constraints and only then make a non-deterministic choice if still necessary. This is the general method of constraint programming which can be paraphrased as ‘propagate and distribute’. A propagation step restricts the set of possible solutions using simple, deterministic inference. A distribution step performs a non-deterministic case distinction and should only be considered when no further inferences are possible through propagation alone. In this fashion, the search tree requires much fewer choice points: propagation is said to ‘prune’ the search tree.

### 2.1.4.1 Constraints as concurrent agents

In concurrent constraint programming, each constraint is implemented by a concurrent agent, also called a ‘propagator’, which observes what is currently known and attempts to derive and contribute new conclusions. This common pool of information is called the ‘constraint store’ and only contains simple statements of the form \( x \) must take values in \( \{1, 2, 7\} \). Whenever a propagator can make an inference, it updates the constraint store: the effect of an inference is to remove possible values for one or more variables. This update may in turn cause another propagator to be able to make an inference and so on. One can imagine a constraint store with its propagators as follows:
2.1.5 What is Oz and who is Mozart?

A concurrent constraint programming system provides a set of procedures for defining propagators and all machinery for running propagation and distribution. The programmer simply models his problem by defining sets of propagators and a strategy for distribution. The rest is done by the compiler and emulator of the programming system.

Oz is a concurrent constraint programming language which has been developed by the Programming Systems Lab in Saarbrücken led by Gert Smolka. The most recent Oz system is Mozart 1.2.0\(^1\). The Mozart system was developed by the Mozart consortium which comprises the programming systems lab in Saarbrücken, the programming systems lab at SICS (Swedish Institute of Computer Science) led by Seif Haridi, and Peter Van Roy’s group at the Universite catholique de Louvain. The Mozart system is freely available, extensively documented, and fully operational.

Oz unifies ideas originating from logic programming in Prolog and functional programming in Lisp or SML. Mozart provides the most innovative technology compared to other constraint programming languages on the market (ILOG, CHIP). This makes Mozart a good foundation for building innovative applications in computational linguistics and artificial intelligence.

Beyond concurrent constraint programming, Mozart supports Internet programming similar to Java. Mozart is also well-suited for building multi agent systems and sophisticated graphical user interfaces.

\(^1\)http://www.mozart-oz.org/
2.2 Getting Started with Oz

2.2.1 Installation

Mozart is available for both Unix- and Windows-based systems. If you use Windows, you need Windows 95 or NT, if you prefer Unix, you can choose between binary tarballs, source tarballs and RPM packages for Linux. To install Mozart, you have to download the newest version from the Mozart Homepage\(^2\).

The Oz Programming Environment is based on the Emacs editor. So make sure you have GNU Emacs version 19.28 or better before installing Mozart! If you aren’t using Emacs yet, you can download it for free for both Unix- and Windows-based systems from GNU Emacs\(^3\).

For a detailed description of the installation process refer to the installation manual\(^4\). This manual also tells you how to make your browser ‘Mozart application enabled’.

2.2.1.1 Common Problems

As Mozart is constantly being improved, maybe you will encounter some problems during the installation. If there is a problem you cannot solve, don’t hesitate to contact the Oz-Support-Newsgroup ps.oz.support located at news.ps.uni-sb.de.

\(^2\)http://www.mozart-oz.org/
\(^3\)http://www.gnu.org/software/emacs/emacs.html
\(^4\)http://www.mozart-oz.org/documentation/install/index.html
• A common problem on Windows is that something goes wrong with the socket. So make sure you have the network installed properly and you have an up-to-date version of winsock.dll. If this file is missing, you will not be able to run Oz, so download it from www.microsoft.com[5].

• There is a bad interaction between Oz-Applets and several programs on Windows NT. Sometimes the applets will not start, so if you encounter this problem, you should try to kill some background programs and most probably everything will work fine afterwards.

• On Unix systems a common problem is that Oz fails to establish a network connection. Of course Mozart does not need a permanent connection to the network! But it tries to contact localhost, and if there is an error message, this means that your local box is illconfigured. So try to add

    127.0.0.1 localhost

    to the file etc/hosts.

    If you encounter problems not mentioned above, please contact the newsgroup to get help. This also enables the developers to improve Mozart!

2.2.2 Programming Environment

If everything worked out fine, the Oz Programming Interface (OPI) should look like this:

The screen is split in two parts: The upper window - called Oz - is used to write small pieces of code that you want to feed to the compiler. If you open an existing file with Oz code, you should do it here too. The lower window shows the Oz compiler.

To see how all of this works, let’s try it in a little example. In the upper window, type:

{Show 'Hello World!'}

Notice that Oz is case-sensitive! It won’t work if you write ‘show’ or ‘SHOW’.

Now you have to feed this line to the compiler. You can do this either by choosing ‘Feed Line’ out of the menu ‘Oz’ or by using the shortcut ‘C-. C-l’. This shortcut looks terrible at the beginning, but you will get used to it easily. This notation means you have to press the key ‘control’ (or ‘strg’) and ‘.’ at the same time and then ‘control’ and ‘l’. There are only a few shortcuts you should remember:

- C-. C-l: Feed line
- C-. C-p: Feed paragraph
- C-. C-b: Feed whole buffer
- C-. C-r: Feed selected region

So if you feed \{Show 'Hello World!'\} into the compiler, it will output:

But to see ‘Hello World!’ shown somewhere, you have to switch to the Oz Emulator. You do this by selecting the buffer *Oz Emulator* from the menu buffers. And there you see the result of the command:
Now try a more interesting example! Switch back to the Oz-buffer (by selecting ‘Oz’ from the buffer menu) and type:

\{Browse ‘Hello World!’\}

and feed it to the compiler. A new window then appears:

![Oz Browser](image)

This time the result is not printed in the *Oz Emulator* buffer, but in this new window which is called *Oz Browser*.

### 2.3 Solving a Combinatorial Problem

Our next goal is to build a constraint solver for the following problem which is given by an equation system with variables denoting integers.

\[
\begin{align*}
X, Y, Z &\in \{1, \ldots, 7\} \\
X + Y &= 3 \cdot Z \\
X - Y &= Z
\end{align*}
\]

A solution of this problem is an assignment of variables \(X, Y, Z\) to natural numbers which satisfies the given arithmetic constraints.

#### 2.3.1 Bits of a Constraint Solver

Let’s now illustrate how to solve this problem in Oz. We define the following constraint which can be added directly to the constraint store:

\[
[X Y Z] :::: 1#7
\]

Here we make use of Oz-variables whose syntax is given by words with leading capital letters. The line above states that \(X, Y, Z\) are so called finite domain variables, i.e. variables taking values in a finite set of integers (here, between 1 and 7). We define the following set of propagators involving these variables:
2.3. Solving a Combinatorial Problem

\[
\begin{align*}
X + Y &= 3 \cdot Z \\
X - Y &= Z
\end{align*}
\]

Note the trailing colon in \(=:\). The trailing colon is characteristic of operators denoting finite domain constraints.

Next, we invoke a predefined distribution strategy to enumerate the assignments to \(X, Y, Z\) consistent with our constraints.

\[
\{ \text{FD.distribute naive \{X Y Z\}} \}
\]

We represent a solution as a record (called feature tree in computational linguistics):

\[
\text{solution}(x:X \ y:Y \ z:Z)
\]

This record has label \(\text{solution}\) and three features \(x, y, z\) (Oz atoms, i.e. words starting with a lowercase letter). \(x:X\) indicates that the value of feature \(x\) is given by variable \(X\).

2.3.2 Observing Propagation

It might be instructive to observe propagation independently from distribution. Propagation relies on the concept of a constraint store which is simply a set of simple constraints on values of variables. New information can be added to the constraint store by propagation. Propagation is done by propagators. These are agents observing the constraint store and getting active whenever they are able to add information. The Oz programmer can observe the constraint store by using the \(\text{Oz Browser}\). For instance, feed the following Oz-code into the Oz-compiler:

\[
\text{declare X Y Z in \[X Y Z\] \tiny \{Browse \[X Y Z\]\}}
\]

This declares three new variables \(X, Y, Z\) for integers in the domain 1, ..., 10 and browses whatever the constraint store knows about their values. When new information is added the browser updates its output. For instance, you may feed the propagator:

\[
2 \cdot Y =: Z
\]

This propagator tells the constraint store new information on upper and lower bounds of \(Y\) and \(Z\) whenever possible. For example, it adds the information that \(Y\) must be at most 5 and \(Z\) must be at least 2 to the constraint store. However, it cannot tell the constraint store to remove odd numbers from the interior of the domain of \(Z\). We next might feed a new propagator stating that \(X\) is strictly smaller than \(Y\):

\[
X <: Y
\]

One of the effect of this propagator is that 1 is removed from the lower bound of \(Y\). This reactivates the observing propagators \(2 \cdot Y =: Z\) which excludes 2 and 3 from the domain of \(Z\).
2.3.3 Composing the Solver

As in Prolog, you need only specify your problem and let the search facilities of the
language derive its solutions. In contrast to Prolog, search in Oz is *encapsulated*: in
particular this means that the search facilities are not implicit as in Prolog, but rather
that search engines are explicit objects in the language, that you can create as many as
you want, and that they are all independent and may coexist peacefully.

Each search engine can solve a different problem. Consequently a search problem must
be encapsulated into a predicate (a procedure) which is passed explicitly to the search
engine. We often call such a procedure a *search predicate* and it consists of 2 parts:

1. a description of what constitutes a solution
2. a specification of the distribution strategy

More precisely, the search predicate takes one argument, intended to denote a solution,
and typically follows the design pattern below:

1. introduce the local variables of the problem
2. define a solution term using these variables
3. post the constraints which the variables must satisfy
4. apply a distribution strategy to the variables

For example, search predicate *Equations* describes exactly the solutions of the prob-
lem considered above.

```oz
declare
proc [Equations Sol]
  X Y Z
in
  Sol = solution(x:X y:Y z:Z)
% Propagate
  [X Y Z] ::: 1#7
  X + Y =: 3*Z
  X - Y =: Z
% Distribute
  {FD.distribute naive [X Y Z]}
end
```

The definition of *Equations* in Oz not only specifies a set of objects but also describes
how these objects can be searched by propagation and distribution. For computing its
solutions in Oz, it is sufficient to pass the definition of *Equations* to the Oz-Explorer.

```oz
{Explorer.all Equations}
{Explorer.one Equations}
```
2.3.4 Was this a good Example?

- Yes, because it was so simple.
- No, since there are much better solvers in this case (Gauss elimination algorithm).

Constraint programming yields good solvers only if no direct algorithm for solving your problem is available.

2.3.5 Questions

- Why are there three colons in the statement \([X \ Y \ Z]:::1\#7\)?
  
  If you want to restrict the domain of a single FD variable then you write \(X:::1\#7\) with two colons. But if you want to restrict the domains of all variables of some list like \([X \ Y \ Z]\), then you need to write three colons.

- Is the name `solution` in the example program `Equations` arbitrary?
  
  Yes, you may choose whatever Oz-atom instead. You could also use a list rather than a record to represent a solution.

- What is the difference between the statements \(X+Y=:3*Z\) and \(X+Y=3*Z\)?
  
  Be careful, this is very different! The first statement \(X+Y=:3*Z\) creates a (concurrent) propagator for equation \(X + Y = 3 \times Z\). The second statement \(X+Y=3*Z\) blocks until the exact values of the variables are known, then evaluates terms \(X+Y\) and \(3*Z\) and finally unifies these two results.

- Why does the Explorer come up with a yellow diamond in the following program instead of searching for a solution?

```oz
declare Z
proc (Equations Sol)
    X Y
    in
        Sol = solution(x:X y:Y z:Z)
        [X Y Z]:::1\#7
        X + Y =: 3*Z
        X - Y =: Z
        (FD.distribute naive [X Y Z])
    end

    (Explorer.one Equations)
end
```

The problem is that the search predicate contains the free global variable \(Z\). Search blocks in an unstable state since the value of \(Z\) is not yet determined (but may eventually become determined, e.g. as an effect of another, concurrent computation).

- Why does the Explorer come up with a yellow square in the following program instead of searching for the solutions?
The problem is that the distributor \{\text{FD.distribute naive [X Y Z]}\} blocks the execution of all subsequent statements. The distributor waits until the variables X, Y, Z have to denote integers in a finite domain. This will never happen since the execution of the statement [X Y Z] :::: 1#7 is blocked by the distributor itself. So we have a deadlock.

The yellow diamond displayed by the Explorer means that the search process is blocked forever.

You can resolve the problem running the distributor in its own thread, i.e. by replacing \{\text{FD.distribute naive [X Y Z]}\} by \text{thread} \{\text{FD.distribute naive [X Y Z]}\} end.

- I found the following call of the explorer in some document. What’s wrong with this?

\{Explorer.one(Equations)\}

This is the old syntax of DFKI Oz 2.0 which is no longer valid in Mozart. There the syntax for calling the Explorer is slightly different. You have to use the more consistent notation \{Explorer.one Equations\} instead.

2.4 Exercise

Write a solver for the equation SEND + MORE = MONEY, where every letter stands for a distinct digit between 0 and 9 and such that leading digits are distinct from 0.

2.5 Summary

- The main problem of constraint solving is the danger of combinatoric explosion.

- The basic method of concurrent constraint programming is ‘propagate and distribute’, in contrast to ‘generate and test’.

- Propagation is an efficient concurrent process. Propagation is typically incomplete from a logical point of view. Completeness can be obtained by adding distribution to propagation.
2.6 Program Collection

%%% V1,...,V6 in {1,...,6}
%%% V1< V2, ... , V5< V6

```
declare
N=6
fun {Tail _|L} L end
proc {GenerateAndTest L}
  L=(MakeList N)
  L ::: 1#N
  {FD.distribute naive L}
  for
    X in L
    Y in {Tail L}
    do
      (X<Y)=true
    end
  end
end
proc {TestAndGenerate L}
  L=(MakeList N)
  L ::: 1#N
  for
    X in L
    Y in {Tail L}
    do
      thread (X<Y)=true end
  end
  {FD.distribute naive L}
end
proc {PropagateAndDistribute L}
  L=(MakeList N)
  L ::: 1#N
  for
    X in L
    Y in {Tail L}
    do
      X<:Y
    end
  end
  {FD.distribute naive L}
end

{ExploreAll GenerateAndTest}
{ExploreAll TestAndGenerate}
{ExploreAll PropagateAndDistribute}
```

%%% X,Y,Y,U,V,W in {1,...,7}
%%% X+Y = 3*Z
%%% X-Y = Z
%%% U+W = 2*Z
%%% V+W = U+V

%%% propagate and distribute
declare
proc {Equations Sol}
    X Y Z U V W
in
    Sol = solution(x:X y:Y z:Z u:U v:V w:W)
    [X Y Z U V W] :: 1#7
% Propagate
    X + Y ::= 3*Z
    X - Y ::= Z
    U + W ::= 2*Z
    V + W ::= U+V
% Distribute
    {FD.distribute naive [X Y Z U V W]}
end

(Explorer.all Equations)

%%% test and generate
declare
proc {Equations Sol}
    X Y Z U V W
in
    Sol = solution(x:X y:Y z:Z u:U v:V w:W)
    [X Y Z U V W] :: 1#7
% Test
    thread X + Y = 3*Z end
    thread X - Y = Z end
    thread U + W = 2*Z end
    thread V + W = U+V end
% Distribute
    {FD.distribute naive [X Y Z U V W]}
end

(Explorer.all Equations)

%%% generate and test
declare
proc {Equations Sol}
    X Y Z U V W
in
    Sol = solution(x:X y:Y z:Z u:U v:V w:W)
    [X Y Z U V W] :: 1#7
% Distribute
  {FD.distribute naive [X Y Z U V W]}
% Test
  X + Y = 3*Z
  X - Y = Z
  U + W = 2*Z
  V + W = U+V
end

(Explorer.all Equations)
What is Computational Linguistics?

- Computational Linguistics is a large discipline like computer science or mathematics.

- The central question is: How can we make a computer understand natural language?

- We will not answer this question during this course. We will even not try to do so. But perhaps, we will succeed in giving you some insights about the state of the art in computational linguistics today.

- There are many subdisciplines of computational linguistics:
  - Syntax
  - Semantics
  - Pragmatics
  - Discourse
  - Morphology
  - Speech
  - ...

We will only care about syntax and semantics of written language.

3.1 What are we doing in Syntax?

- We assume that everybody knows about context-free grammars. For example:

  % Rules:
  
  s  ->  np  vp
  np  ->  det  n
  vp  ->  v  np

  % Lexicon:
  
  n  ->  man  |  woman
  v  ->  likes  |  knows
  det  ->  the  |  a  |  every

We will soon build a bottom-up parser for a context free grammar in Oz.
• Does everybody know what a chart-parser is? If you don’t know it then your are probably not studying computational linguistics. Don’t worry, we will explain it to you and even built a chart-parser in Oz.

• Unification Grammar = Context-free Grammar + Feature constraints

• We will show to you how to built a chart parser for unification grammars in Oz. This will turn out to be quite simple even though some advanced features of Oz are needed (which are not available in any other language).

• In the last part of this lecture, we will built a parser for a dependency grammar in Oz (if there is enough time). Beside of what you need for unification grammar, this parser makes heavily use of constraint programming with finite domain constraints.

3.2 What are we doing in Semantics?

• What is the semantics of the following sentence?

    Tomorrow, every student will write an Oz-program.

Its semantics may be given by the following logic formula:

    tomorrow(every x (student(x) -> exists y (oz_program(y) & write(x,y))))

• If there is time than we will built a underspecified semantics for a unification grammar.

• We need an implementation of the lambda-calculus in Oz and also a solver for dominance constraints.

3.3 Demo Systems

We will demostrate some larger systems built in different projects of the SFB 378 where the technology presented in this lecture is used:

• Chorus Demo System[1]: A unification grammar with chart parser with underspecified semantics based on dominance constraints.

• A solver for dominance constraints is needed in Chorus and Lisa and built by in a Chorus/Negra/Nep cooperation.

• Negra Demo Apple[2]: A dependency grammar with a parser whose implementation is based on finite domain constraints and finite set constraints in Oz.

---

Part II

Functional Programming and Context-Free Parsing
We first introduce the Oz’s basic data structures and the Oz’s expressions and statements to program with them. We concentrate on concepts rather than completeness.

A data-structure allows to store values of some (data) type and provides the standard procedures for compute with these values. Even though data structures are provided by all programming languages, they are not always provided the same way. All we say has to be complemented by The Oz Base Environment\(^1\) of the Mozart documentation\(^2\).

We take the viewpoint of functional programming as in SML or Lisp which is quite distinct from that of constraint programming. However, functional programming provides a good platform for constraint programming. The idea of functional programming is to organize computation in terms of values, types, and functions between values of some types. Functions are interpreted as procedures computing the functional value for given arguments.

### 4.1 Statements versus Expressions

A functional program in Oz is sequence of statements. Statements describe actions such as showing a value or assigning a value to a variable. The following program, for instance, consists of two statements:

```
{Inspect welcome}
{Inspect 7*8-14}
```

Values are described by expressions that may be nested such as \(3+4\times25\). Expressions are to be evaluated to values. Note that Oz’s statements do not have values themselves in contrast to Oz’s expressions.

Later on we will also see expressions that contain statements. The actions of these statements are to be executed during evaluation.

\(^1\)http://www.mozart-oz.org/documentation/base
\(^2\)http://www.mozart-oz.org/documentation
4.2 Values and Types

Up to now we have seen several values used in Oz: numbers, atoms, records, and lists. There are more values and types in Oz. A still incomplete list of values and types is the following:

- A number is either an integer or a float (rational number).
- An atom is a word.
- A Boolean value is either true or false.
- The unit is a constant value without particular meaning (a dummy).
- A record is an atom or a term of the form Lab(F1:V1 ... Fn:Vn) where:
  - \( n \geq 0 \).
  - the label Lab is an atom, the unit, or a Boolean.
  - the features F1, ..., Fn are pairwise distinct atoms or integers.
  - the fields V1, ..., Vn are arbitrary values.
- A tuple is a record with only integer features.
- A list is a tuple which is either the atom nil or a tuple \((1:V 2:L)\) where \(1\) is an atom, \(V\) a value, and \(L\) a list. The atom \(1\) is sometimes called ‘cons’.
- A string "hiho" is a list of ASCII characters, i.e integers between 0 and 128.
- A procedure is a value.

4.3 Expressions for Values

Oz provides expressions to describe values. There may be many alternative expressions describing the same value. We start with variable free expressions that determine some value completely.

- Integers are described as \(0, 1, -1, 2, 3\) etc and floats by \(0.0, 1.0, -1.1\) etc.
- Atoms are described by words starting with lower case letter like `thisIsAnAtom` or by a word in backwards quotes like `‘case’, ‘true’` and `‘ThisIsAnAtom’`.
- The Booleans and the unit are described by the keywords `true, false, unit`.
- Typical description for tuples and records are described by the keywords `true, false, unit`.

```
plus(5 times(5 ~10))
address(street:'Talstrasse'
    name:unit(first:hans
        second:kamp))
det(phon:a number:singular)
```
In the first tuple, we have left out the features; it’s a syntactically sugared version of `plus(1:5 2:times(1:5 2:-10)).`

The values of a record at some feature can be selected by using the selection function that is denoted by a dot. For instance, the atom `singular` is described by the expression

```
    det(phon:a number:singular).number
```

- Typical descriptions of lists are: `1|2|3|nil`, `[1 2 3]`, and `nil`. Note however that `[ ]` does not describe the empty list!

- A description of a procedure computing the square function is:

```
    fun($ X} X*X end
```

The symbol `$` simply means that this procedure is anonymous, i.e. is not yet given a name. The syntax for the application of procedures uses curly brackets. For instance, the number 9 is described by the following application whose evaluation computes the square of 3:

```
    {fun($ X} X*X end 3)
```

### 4.4 Variables, Scope, and Binding

An Oz variable describes a value of an arbitrary type. Variables in Oz are logic variable whose value cannot cannot be changed.

The Oz programming interface comes with a lot of predefined global variables such as `List` and `Number`. The values of both variables are records containing the standard functions for lists and numbers. For instance, a procedure for multiplication `Number.*` can be selected from the record `Number` at feature `'*'`. The expression `X*X` in turn is nothing else than syntactic sugar for the application `{Number.* X X}`.

Local variables can be introduced in Oz by using expression of the form `local ... in ... end`. The following expressions describes a record which contains two numbers, the squares of 3 and 4.

```
    local
    Square = fun($ X} X*X end
    in
    record(s3:{Square 3} s4:{Square 4})
    end
```

The scope of a local variable is restricted by the local-end-expression in which its is introduced. For instance, the local variable `Square` cannot be accessed any further.

Note that `local` constructs can not only be used to form statments rather then expressions:

```
    local
    X = 3*4
    in
```
30 Chapter 4. Basic Data Structures, Expressions and Statements

\[
\{\text{Browse } X*X+5\} \\
\text{end}
\]

This statement shows a value rather but does not describe any.

### 4.5 Scoping in the Interactive User Interface

There is also a way for introducing new global variables in the programming interface. The can be done by using the construct `declare` which introduces new variables and binds them to their value. For instance we can declare the variable \( x \) and assign the value 2 to \( x \) by the following statement:

\[
\text{declare } x=2
\]

The scope of declared variables is open until the end of the session in the interactive user interface. It is like a local expression but without an explicit end. Note that `declare` can only be used within the interactive user interface.

### 4.6 Inspecting Values and Types

The Oz Inspector is a output tool provided by the Oz programming interface. The Oz Inspector is written in Oz itself and bound to the global variable `Inspect`. For instance, we can inspect the value of the term \( 2*4+3 \) by executing:

\[
\text{declare } x=2*4+3 \\
\{\text{Inspect } x\}
\]

Evaluating the statement \( \{\text{Inspect } x\} \) evokes the side effect of inspecting that value of \( x \) which is 11. Note that \( \{\text{Inspect } x\} \) is a statement and not an expression, so that it does not have any value itself.

The Inspector allows you to observe the values of expressions. For instance, feed the following lines to the emulator.

\[
\begin{align*}
\text{declare } A & = \text{'Maria'} \\
\text{declare } L & = [A \text{'Klaus'}] \\
\text{declare } R & = \text{address(street:'Talstrasse' name:unit(first:hans second:kamp) kids:L)} \\
\text{declare } T & = \text{tuple(A L R)} \\
\{\text{Inspect } T\}
\end{align*}
\]
4.7 Type Checks

The types of values can be checked in Oz dynamically, as illustrated by the following examples.

```
{Inspect {IsRecord R}}
{Inspect {IsRecord F}}

{Inspect {Or {IsRecord ~100}
    {IsBool ~100} }}
{Inspect {And {And
    {IsNumber ~100}
    {IsInt ~100}}
    {IsFloat ~100} }}

{Inspect {Not {IsRecord false}}}  
{Inspect {IsRecord {IsRecord false}}} 
{Inspect {And
    {And
        {IsList L}
        {IsTuple L}}
    {IsRecord L}}} 
```

It might be surprising that the Unit, the Booleans, and atoms are records. The reason is that a record need not to have subrecords.

There also exists a predefined procedure in Oz which computes the type of a given value. This is the procedure \texttt{Value.status}. When applied, it return not only the type of its input argument but also its actual status which may be either determined, kinded, or free.

```
{Inspect 
    [ {Value.status R} 
    {Value.status T} 
    {Value.status L} 
    {Value.status F} ] }
```

For functional programming, we’d better deal only with values of status ‘determined’, in order to avoid suspensions (blocking computations).

4.8 Testing Equality

The equality test \texttt{==} checks whether two Oz values are equal. It is a function of type

\[ == : \text{value} \times \text{value} \rightarrow \text{bool} \]

. For instance, a record without subrecords is identified with its label.
The function == for testing equality can be used to built expressions. Note the == is very much different from the operator = in assignment statements. The latter operator = binds a variable (usually on the left) to a value (usually on the right). The former equality test == returns a Boolean value, but only if both input arguments are known. Otherwise, it waits until they become known. The programmer must be careful because all followup up commmands are suspended too.

%% introduce a free variable

\textbf{declare} \textit{X}

%% the equality test waits until the value of \textit{X} is known.
%% the following browse statement therefore waits too.

(Browse \textit{X}==2)

%% feed the next line independently.

\textit{X}=3

%% now the Browser should browse false

The equality test proceeds only once \textit{X} gets bound to its value. The \textbf{Browse} can only become active once this has happened.

The above example may go wrong if you feed the above lines in a single block, since everything in the same block that succeeds a blocking statement blocks. The statement that assign a value to \textit{X} must thus be fed in another block.

### 4.9 Procedures

A \textit{functional procedure} is a procedure which computes a function. Given a value of the functions domain, it computes the result of applying the function to this value. If this computation terminates then this output value is returned as output.

As an example, we consider a the \textit{square} function of type

\[ \text{square : number} \rightarrow \text{number} \]

which maps a number to its square. The corresponding functional procedure can be applied to an integer input and the returns the square of the input as its output. It can be defined as follows:

\textbf{declare} \textit{Square} = \textbf{fun}($\texttt{N}$) \textit{N*N} \textbf{end}
There exists a nicer syntax where dollar symbols are non needed. With this syntax, the above definition can be rewritten equivalently to:

\[
\text{declare fun} \ (\text{Square} \ N) \ N * N \ \text{end}
\]

We can now build expressions that describe the value to which the function maps another value. For instance, the value of the following application is 16.

\[
\{\text{Inspect} \ \{\text{Square} \ 4\}\}
\]

So far, we have seen only functions with a single input and a single output argument. It is also easily possible to define functional procedures for functions with several inputs. As an example, we consider the ternary Boolean valued function which tests whether the sum of the squares of the first two arguments is greater than the square of the third argument.

%% an arithmetic test
\[
declare \text{Square} = \text{fun} \ (\$ \ X) \ X * X \ \text{end}
declare \text{fun} \ (\text{Test} \ X \ Y \ Z)
\{\text{Square} \ X\} + \{\text{Square} \ Y\} == \{\text{Square} \ Z\}
\ \text{end}
\{\text{Inspect} \ \{\text{Test} \ 3 \ 4 \ 5\}\}
\]

This function can be given two different types since the comparison function in Oz can either compare floats or integers.

\[
\text{maximum} : \text{int} \times \text{int} \times \text{int} \rightarrow \text{int} \\
\text{maximum} : \text{float} \times \text{float} \times \text{float} \rightarrow \text{float}
\]

Note that floats cannot directly be compared with numbers and vice versa. Usually, one calls functions polymorphic if they can be applied to different types.

Functions without output argument are simply relations and the corresponding procedures relational procedures. Oz provides extra syntax to write relational procedures. For instance, we can define a own output procedures (which output by side effect instead of returning values):

\[
declare \text{proc} \ (\text{MyInspect} \ X) \\
\{\text{Show} \ X\} \\
\{\text{Inspect} \ ['\text{MyInspect}: ' \ X]\} \\
\text{end}
\{\text{MyInspect} \ hiho\}
\]

The application of a relational procedure is a statement which does not describe a value in contrast to an expression. This explains also why \{\text{Browse} \ X\} does not have a value.
4.10 Functions as Relations

Internally, Oz supports only a single kind of procedures which are all relational. The idea is that \( n \)-ary functions are treated as \( n + 1 \)-ary relations. This can be observed in the Browser:

\[
\text{declare } \text{Square} = \text{fun}($ \text{X} \text{X}^2 \text{end}
\]

\[
\{\text{Inspect } ['Browsing fun($ \text{X} \text{X}^2 \text{end yields <P/2>ʼ Square}])
\}
\]

When browsing the value of procedure named \text{Square} a string is displayed meaning that the value of \text{Square} is relational procedure with 2 arguments rather than a functional procedure with a single argument. The additional argument is used for the output.

Note also that the expression \{\text{Square 111}\} denotes a value, whereas \{\text{Browse 111}\} has none. The reason is that \text{Browse} denotes a relational procedure which in contrast to a functional procedure does not return a output value.

As said already, the output behaviour of a functional procedure can be simulated by a relational procedure which raises a side effect on a logic variable (see section unification). In fact, Oz supports functional procedure through relational procedures. A functional procedure with \( n \) arguments is translated into a relational procedure with \( n + 1 \) arguments, where the last arguments serves as an output argument. For instance, the functional procedure \text{fun}\{\text{Square X} \text{X}^2 \text{end}\} and its application \( Y = \{\text{Square 3}\} \) are translated as follows:

\[
\text{fun}\{\text{Square X} \text{X}^2 \text{end} \Rightarrow \text{proc}\{\text{Square X Out} \text{Out=X}^2 \text{end}\}
\]

\[
\text{declare } Y = \{\text{Square 3}\} \Rightarrow \text{declare } \text{in } \{\text{Square 3 Y}\}
\]

The translation introduces a free logic variable \( Y \) that is passed as an additional argument to the application \{\text{Square 3 Y}\}. The application is then executed; once it finishes it computation, it binds the logical variable \( Y \) to the functional result \( 9 \). The result is thus passed by a side effect on a logic variables.

4.11 Boolean Conditionals

Boolean conditionals distinguish cases on basis of a Boolean value. Their syntax has the form \text{if ... then ... else ... end}. We can use Boolean conditionals for instance to define functions for type coercion. The following procedure coerces a number (an integer or a float) into a float.

\[
\text{declare fun}\{\text{NumberToFloat N}\}
\]

\[
\text{if } \{\text{IsFloat N}\}
\]

\[
\text{then } N
\]

\[
\text{else } \{\text{IntToFloat N}\}
\]

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{%% summation of floats to integers requires type conversion}
\]

\[
\{\text{Inspect } \{\text{NumberToFloat 3.3} +\{\text{NumberToFloat 2}\}\}
\]
4.12 Records

Boolean conditionals to define recursive functions on integers. Consider for instance the factorial function. We implement it by using the equality tester == of type value × value → bool.

```%
(Factorial N) = N * ... * 1
```

```declare fun(Factorial N)
  if N==0
  then 1
  else N*(Factorial N-1)
end
end```

```Inspect {Factorial 3})
{Factorial 1111})```

### 4.12 Records

Records are the central data structure in Oz. Records are equally important in computational linguistics, where they are called feature trees. For instance, one might wish to represent the English word `girl` and its features as the following record:

```word(cat:noun phon:[girl] subcat:determiner)```

The main operation on records is feature selection which allows to access a field belonging to some feature. Feature selection is denoted by a dot. For instance:

```{Inspect word(cat:noun phon:[girl] subcat:determiner).phon}
{Inspect word(cat:noun phon:[girl] subcat:determiner).phon.1}```

Note that feature selection is a very efficient operation in Oz which can be done in constant time. A record is implemented as a `hash table` whose keys are the features of the record.

The base environment of Oz is provided by a set of records that are also called `modules`. Global variables denoting modules `Number`, `Record`, `List`, `FD`, and many more. For instance if you want to see the functionality provided for finite domains or records in Oz then simply browse the modules `FD` and `Record`.

```{Inspect FD}
{Inspect Record}```

This also explains the syntax of `FD.distribute` in our introductory example: a procedure for distribution is selected from the record `FD`.

For further information on records, we refer to ‘The Oz Base Environment’.
4.13 Tuples

Tuples are special records with integers as for instance \( f(X_1 \ X_2 \ X_3) \) which is equal to \( f(1:X_1 \ 2:X_2 \ 3:X_3) \). The values of the features of a tuple can be arbitrary. The functionality of tuples is mostly inherited from that for record.

The Oz syntax supports the special tuple constructor \( \# \) that can be written in infix notation. It can be used to construct tuples like \( b\#c \) which is equal to \( \#(b \ c) \). More generally, it holds that

\[
X_1\#X_2\#...\#X_n = \#(X_1 \ X_2 \ ... \ X_n)
\]

We can also use this tuple constructor to build nested tuples. In this case we have to write parenthesis explicitly. For instance:

\[
X_1\#(X_2\#X_3) = \#(X_1 \ #(X_2 \ X_3))
\]

4.14 Lists

Lists are another important data structure in Oz similarly to Lisp. Therefore, much functionality for lists is provided in the Oz-module List. Again, we only give some examples here and refer to documentation The Oz Base Environment for further information.

Here is an example of a list which might be obtained by reading lexical information on natural language from some file:

%% a list representing a simple lexicon

\[
\text{declare WordReps=}[[\text{mary} \ \text{noun} \ \text{nil}]
\]
\[
\quad[[\text{john} \ \text{noun} \ \text{nil}]
\]
\[
\quad[[\text{girl} \ \text{noun} \ \text{determiner}]
\]
\[
\quad[[\text{nice} \ \text{adjective} \ \text{nil}]
\]
\[
\quad[[\text{pretty} \ \text{adjective} \ \text{nil}]
\]
\[
\quad[[\text{the} \ \text{determiner} \ \text{nil}]
\]
\[
\quad[[\text{laughs} \ \text{verb} \ \text{noun}]
\]
\[
\quad[[\text{meets} \ \text{verb} \ [\text{noun} \ \text{noun}]]
\]
\[
\quad[[\text{kisses} \ \text{verb} \ [\text{noun} \ \text{noun}]]
\]
\[
\quad[[\text{embarrasses} \ \text{verb} \ [\text{noun} \ \text{noun}]]
\]
\[
\quad[[\text{thinks} \ \text{verb} \ [\text{verb} \ \text{noun}]]
\]
\[
\quad[[\text{is} \ \text{verb} \ [\text{adjective} \ \text{noun}]]
\]
\[
\quad[[\text{met} \ \text{adjective} \ \text{nil}]
\]
\[
\quad[[\text{kissed} \ \text{adjective} \ \text{nil}]
\]
\[
\quad[[\text{embarrassed} \ \text{adjective} \ \text{nil}]]
\]

\{\text{Inspect WordReps}\}
\]

As proposed above, one might wish to represent the features of a word in a more accessible way by using a record rather than a list. For instance, the record \( \text{word(cat:noun phon:[mary] subcat:nil)} \) is more readable than the list \[\text{[mary noun nil]}\). More importantly, it is possible to
select a feature of a word in the record representation in constant time, whereas it takes linear time in the number of features in the list representation.

Given the list of list \texttt{WordReps} above, we can compute a list of records \texttt{Words} by converting all representations in \texttt{WordReps}. This can be done by using the functional procedure \texttt{Map}:

\begin{verbatim}
%% convert lists to records
declare fun(Convert) [P C S]
   word(phon:[P] cat:C subcat:S)
end
declare Words = {Map WordReps Convert}
{Inspect Words}
\end{verbatim}

When \texttt{Convert} is applied it matches its argument against the pattern \texttt{[P C S]} and returns the record \texttt{word(phon:[P] cat:C subcat:S)}. More on pattern matching can be found in the next section.

Note that the procedure \texttt{Map} is provided by the module \texttt{List}. Indeed, \texttt{Map} is identical to \texttt{List.map}, as shown when feeding:

\begin{verbatim}
{Inspect Map==List.map}
\end{verbatim}

Here, we apply the predefined functional procedure \texttt{==} which compares two Oz-values for equality and returns the Boolean value.

Next, we might want to filter all verbs out of the lexicon \texttt{Words}. This can be done by using the procedure \texttt{Filter} also defined in the module \texttt{List}:

\begin{verbatim}
declare Verbs = {Filter Words fun W | W.cat == verb}
{Inspect Verbs}
\end{verbatim}

Or else, you might want to test, whether all words in the lexicon are verbs. This can be done by using the function \texttt{FoldL} (or alternatively \texttt{FoldR}).

\begin{verbatim}
declare B1 = {FoldL Words fun B W | And B W.cat==verb}
   true
declare B2 = {FoldR Words fun W B | And B W.cat==verb}
   true
{Inspect [B1 B2]}
\end{verbatim}

The value of the term \{FoldL \{X1 \ldots \ Xn\} P Z\} is the same as \{P \ldots \{P \{P \{X1 \ X2\} \ X2\} \ldots \ Xn\}\}
i.e. where grouping goes from left to right (hence the \texttt{L} in \texttt{FoldL}). Correspondingly, \{FoldR \{X1 \ldots \ Xn\} P Z\} evaluates like \{P \ X1 \{P \ X2 \ldots \{P \ Xn \ Z\} \ldots\}\} where grouping goes from right to left (hence the \texttt{R} in \texttt{FoldR}).
4.15 Characters, Strings, and Virtual Strings

Characters in Oz are modelled by integers between 0 and 128 as specified by the ASCII table. In Oz we can write

\&a

to obtain the ASCII number of the letter a. Strings are lists of characters. For instance

\&a

A virtual string is either a string, an atom, or a composition

V1#...#Vn

of virtual strings. When you want to pass strings from Oz to the outside you can freely use virtual strings instead of strings. For instance, we can define URL compositionally:

\begin{verbatim}
declare AuthorURL = 'http://www.ps.uni-sb.de/~niehren'
declare CourseDir = 'Web/Vorlesungen/Oz-NL-SS01/vorlesung'
declare CourseURL = AuthorURL#'/'#CourseDir
\end{verbatim}

4.16 Loops

Oz provides a nice syntax for loops even though these can be easily expressed by recursion. The full beauty of loops will be discuss in Chapter on Loops (page 63). Here we only give a simple example related to working with lists.

Loops are useful if one wants to do something for all elements of a list. For instance, we might want to show all elements of a list:

\begin{verbatim}
declare proc(InspectAll List)
    for X in List do {Inspect X} end
end
{InspectAll [5 4 3 2 1]}
\end{verbatim}

We leave it as an exercise to the reader to write a recurse program that does the same but without any loop. This becomes quite easy with using pattern matching conditionals that we discuss in the next section.
Pattern matching is a very convenient way of decomposing records. A pattern is simply a term with variables such as \( \text{unit}(\text{hi}:X \text{ ho}: \text{unit}(Y)) \). Matching a pattern against a value means to bind the variable in the pattern to the corresponding subrecords. In Oz, we can write pattern matching as follows:

```plaintext
%% decomposing data structures
declare unit (hi:X ho:unit (Y)) = unit (hi:’blue’(’tain’) ho:unit (jazz))
(Inspect [X Y])
```

The `declare` is needed to declare the variables in the pattern \( X, Y \) as new.

Pattern matching is also highly useful in conditionals. Oz provides pattern matching conditionals denoted by `case ... end` beside Boolean conditionals.

Pattern matching conditionals are most typically used for defining recursive functions on recursive data structures such as lists. As an example, we consider the functional procedure `SquareList` which when applied inputs a list of numbers and outputs the list its squares.

```plaintext
%% pattern matching conditionals and recursion
declare fun {SquareList Ints}
    case Ints
        of I|Is then I*I | {SquareList Is}
        elseof nil then nil
    end
end
(Inspect {SquareList [1 2 3 4 5]})
(Inspect {SquareList {SquareList [1 2 3 4 5]}})
```

One the one hand side, pattern matching selects the matching condition and on the other hand side, it selects the matching data.

An implicit form of pattern matching is supported in procedure definitions. For instance, the definition of the converter function

```plaintext
fun {Convert [P C S]}
    word (phon:[P] cat:C subcat:S)
end
```

we have seen before is equivalent to

```plaintext
fun {Convert Arg}
    case Arg of [P C S] then word (phon:[P] cat:C subcat:S) end
end
```
4.18 Towards State and Abstract Data Structures

Stateful programming is a useful concept known from imperative programming languages. It is also supported by many functional languages, where state is typically used to implement abstract data structures (ADS). These are called objects in object-oriented programming. Typical examples for abstract data structures are stacks, queues, and dictionaries.

The primitives needed for stateful programming are stateful references, i.e. references to mutable values. Logic variables are not stateful in that sense since their value can never be changed. Only the mode of a logic variable is mutable if the logic variable gets bound to its value.

Throughout this book, we will frequently use stateful programming with abstract data structures. For now, we will only work with predefined abstract data structures and show how to use them. How to write your own abstract data structures will be discussed in the chapter on Stateful Data Structures (page 75).

The nice thing with abstract data structures is that they abstract away from their implementation. Therefore, one can also easily exchange the implementation of an abstract data structure without changing its usage.

Abstract data structures (that are called objects in object oriented programming) can only be accessed by a set of functions (that called methods in object oriented programming). It is thus sufficient to know these functions (and not their implementation) to use them. A record of such functions is returned when creating an abstract data structure. The types of these functions describe the interface to the abstract data structure. The creation is done by calling a creator function (that is called class in object oriented programming).

4.19 Modules

Mozart supports (predefined) abstract data structures through a module system. Modules can be loaded from files or URLs. These files contain precompiled Oz-programs and therefore carry the suffix \texttt{.ozf} instead of \texttt{.oz}.

We now illustrate all these concepts at the example of an ordinary stack which we will need already in the parsing example of the next Chapter. A stack is simply a list of values that can be muted by methods push and pop and be tested for emptiness. Push adds a new value in front of the list. Pop deletes the first value of the list and returns it. It raises an error if there is the stack is empty.

Every stack is a record of procedures of the following type:

\begin{verbatim}
unit {
  push : Value ->
  pop  : -> Value
  isEmpty: -> Bool
  ...
}
\end{verbatim}
Those kinds of abstract data structures that we will frequently use in the lecture are made available in the module `ADS_Module` that is located at the url `ADS_URL` shown below. This module is a record of functional procedures (classes) by which to create the different kinds of abstract data structure (objects). The creator for stacks, for instance, is available as `ADS_Module.newStack`.

%% define the ADS_URL
%% maybe change the ADS_URL to an appropriate local filename
declare AuthorURL = 'http://www.ps.uni-sb.de/~niehren'
declare CourseDir = 'Web/Vorlesungen/Oz-NL-SS01/vorlesung'
declare CourseURL = AuthorURL#/'#CourseDir
declare ADS_URL= URL#/'/Functors/Version.3.2/Abstract.ozf'

%% load the module and use it to create a stack
declare [ADS_Module] = {Module.link [ADS_URL]}
declare NewStack = ADS_Module.newStack % (class)
declare Stack = {NewStack} % (object)

%% now we can use this stack
{Inspect {Stack.isEmpty}}
{Stack.push 5}
{Stack.push 27}
{Inspect {Stack.isEmpty}}
{Inspect {Stack.pop}}

The function `Module.link` maps a list of urls or filenames to the corresponding list of modules. This is the interactive way of loading and linking modules. Later we will discuss how to write your own module programs called functors: functors have import declarations which don’t need to make any explicit calls to `Module.link` or anything like that, and are much simpler and more intuitive to use.

If you prefer, to download the module `ADS_Module` and to save it locally on your file system, you should change `ADS_URL` to the appropriate filename.

### 4.20 Exercises

- Define a functional procedure `Append` computing the concatenation of two lists. Is your procedure tail recursive?

- Define a functional procedure `Product` which computes the product of a list of integers. For instance, `{Product [1 2 3]}` should return 6. Is your procedure tail recursive? Do you know of a procedure `ProductShort` computing the same function than `Product` but whose code is less than 40 letters long? Does the execution of your procedure `ProductShort` require constant space?

- Some people A, B, C, D, E order wine together. Below you find a table which contains the prices per bottels and the number of bottels ordered per person. How much has everybody to pay?
Chapter 4. Basic Data Structures, Expressions and Statements

Prices = [19.3 27.25 29.75 26.5 27.5 46.5 15.32 5]
A = [4 3 0 3 0 6 3 3 1]
B = [0 2 0 4 2 0 0 0 0]
C = [0 0 3 1 1 3 0 0 1]
D = [0 3 3 3 3 3 3 3 2]
E = [2 3 3 1 0 0 0 0 0]

- Write a functional procedure `AppendLists` based on `FoldL` (or `FoldR`) and `Append` which transforms the list `[[a] [b c [d]] nil [e f]]` to a list `[a b c [d] e f]`. Which variant does need less time, when using `FoldL` or `FoldR`? Why?

- Write a functional procedure `Reverse` which reverses a list. What is the worst case complexity for the runtime of your program? If it is not linear then write a linear one `SmartReverse`, too.

- Write an evaluator `Eval` for arithmetic expressions represented as Oz-terms (without variables). For instance, `(Eval times(plus(2 4) 3))` should return 18.

- Boolean conditionals and record selection can be used instead of pattern matching even though this is much less convenient. To see this, write an functional procedure `SquareList` that does not use a pattern matching conditional.

- Write a procedure that inspects all elements of a list without using loops.

- Write a functional procedure `ToList` of type `stack -> list(value)` that converts a stack to its actual contents.

4.21 Program Collection

```
-:: Program collection

-:: browse values

declare A = 'Maria'
declare L = [A 'Klaus']
declare R = address(street:'Talstrasse'
  name:unit(first:hans
  second:kamp)
  kids:L)
declare T = tuple(A L R)
{Inspect T}

-:: browse types

{Inspect {IsRecord R}}
{Inspect {IsRecord F}}
{Inspect {Or {IsRecord ~100}}
```
{IsBool ~100})
{Inspect (And (And
    {IsNumber ~100}
    {IsInt ~100})
    {IsFloat ~100}))}

{Inspect (Not (IsRecord false))}
{Inspect (IsRecord (IsRecord false))}

{Inspect (And
    (And
        {IsList L}
        {IsTuple L})
    {IsRecord L})}

{Inspect 

% browse status

{Inspect [(Value . status R)
    (Value . status T)
    (Value . status L)
    (Value . status F)]}

% test equality

{Inspect atom == atom()}
{Inspect (IsRecord atom)}

{Inspect (And true == true()
    (And unit == unit()
        false == false()))}

{Inspect "hiho" == [104 105 104 111]}

%%%%% procedures

declare Square = fun($ N) N*N end

% dollar free syntax

declare fun(Square N) N*N end

% application of functions

{Inspect (Square 4)}
% functions with several inputs

%% an arithmetic test
declare Square = fun($ X) X*X end
declare fun(Test X Y Z)
   {Square X} + {Square Y} == {Square Z}
end
{Inspect {Test 3 4 5}}

%% relational procedures
declare proc(MyInspect X)
   {Show X}
   {Inspect ['MyInspect: ' X]}
end
{MyInspect hiho}

%% browsing functional procedures
declare Square = fun($ X) X*X end
{Inspect ['Browsing fun($ X) X*X end yields <P/2>' Square]}
{Inspect {Square 111}}

%% functions for type coercion
declare fun(NumberToFloat N)
   if {IsFloat N}
      then N
   else {IntToFloat N}
   end
end
%% summation of floats to integers requires type conversion
{Inspect {NumberToFloat 3.3}+{NumberToFloat 2}}

%% recursive functions on integers
%% (Factorial N) = N * ... * 1
declare fun(Factorial N)
   if N==0
      then 1
   else N*{Factorial N-1}
   end
end
{Inspect {Factorial 3}}
{Inspect {Factorial 1111}}

$$$$ records $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$_$$

word(cat:noun phon:[girl] subcat:determiner)
4.21. Program Collection

---

(Inspect word(cat:noun phon:[girl] subcat:determiner).phon)
(Inspect word(cat:noun phon:[girl] subcat:determiner).phon.1)

(Inspect FD)
(Inspect Record)

%%% lists %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%#####

%% a list representing a simple lexicon
declare WordReps=[[mary noun nil]
[john noun nil]
[girl noun determiner]
[nice adjective nil]
[pretty adjective nil]
[the determiner nil]
[laughs verb noun]
[meets verb [noun noun]]
[kisses verb [noun noun]]
[embarrasses verb [noun noun]]
[thinks verb [verb noun]]
is verb [adjective noun]]
[met adjective nil]
[kissed adjective nil]
[embarrassed adjective nil]]

(Inspect WordReps)

%% convert lists to records
declare fun(Convert [P C S])
    word(phon:[P] cat:C subcat:S)
end
declare Words = {Map WordReps Convert}
(Inspect Words)

declare Verbs = {Filter Words fun($ W)
    W.cat == verb
    end}
(Inspect Verbs)

%%% pattern matching %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% decomposing data structures
declare unit(hi:X ho:unit(Y)) = unit(hi:'blue'('tain') ho:unit(jazz))
(Inspect [X Y])

%% pattern matching conditionals and recursion
declare fun(SquareList Ints)
    case Ints
    of I|Is then I*I | {SquareList Is}
Chapter 4. Basic Data Structures, Expressions and Statements

```oz

elseof nil then nil
derend
{Inspect {SquareList [1 2 3 4 5]}}
{Inspect {SquareList {SquareList [1 2 3 4 5]}}}

%%% loops %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

declare proc {InspectAll List}
for X in List do {Inspect X} end
derr {InspectAll [5 4 3 2 1]}

% % % % free variables % % % % % % % % % % % % % %
%
% % % Free, Kinded, and Determined Variables % % % %
declare X Y Z=1
Y :: 1#7
{Browse status(x:{Value.status X})
 y:{Value.status Y}
 z:{Value.status Z})}

{Wait X} {Browse 'will browse if you feed the line X=1 separately'}

X=1

{Browse newStatus(x:{Value.status X})
 y:{Value.status Y}
 z:{Value.status Z})}

%%% abstract data types, modules, and stacks % % % % %
%
%% maybe change the ADS_URL to an appropriate local filename
declare AuthorURL = 'http://www.ps.uni-sb.de/~niehren'
declare CourseDir = 'Web/Vorlesungen/Oz-NL-SS01/vorlesung'
declare CourseURL = AuthorURL#'CourseDir
declare ADS_URL = URL#'LoadURL&pickle.version/Abstract.ozf'

% % load the module with the abstract data structures
declare [ADS_Module] = {Module.link [ADS_URL]}
% % select the abstract data structure for the stack (class)
declare NewStack = ADS_Module.newStack
% % create a new stack (object)
declare Stack = {NewStack}

% % now we can use this stack
{Inspect {Stack.isEmpty})
{Stack.push 5}
{Stack.push 27)
{Inspect {Stack.isEmpty})
{Inspect {Stack.pop})
5

A Naive Parser for Context-Free Grammars

5.1 Natural Language, Syntax and Context-Free Grammar

Natural language has an underlying structure usually referred to under the heading of Syntax. The fundamental idea of syntax is that words group together to form so-called constituents i.e. groups of words or phrases which behave as a single unit. These constituents can combine together to form bigger constituents and eventually sentences. So for instance, *John, the man, the man with a hat* and *almost every man* are constituents (called Noun Phrases or NP for short) because they all can appear in the same syntactic context (they can all function as the subject or the object of a verb for instance). Moreover, the NP constituent *the man with a hat* can combine with the VP (Verb Phrase) constituent *run* to form a S (sentence) constituent.

A commonly used mathematical system for modelling constituent structure in Natural Language is Context-Free Grammar (CFG) which was first defined for Natural Language in (Chomsky 1957) and was independently discovered for the description of the Algol programming language by Backus (backus 1959) and Naur (Naur et al. 1960).

Context-Free grammars belong to the realm of formal language theory (cf. Hopcroft and Ullman 1974 for a detailed overview) where a language (formal or natural) is viewed as a set of sentences; a sentence as a string of one or more words from the vocabulary of the language and a grammar as a finite, formal specification of the (possibly infinite) set of sentences composing the language under study. Specifically, a CFG (also sometimes called Phrase-Structure Grammar) consists of four components:

- T, the terminal vocabulary: the words of the language being defined
- N, the non-terminal vocabulary: a set of symbols disjoint from T
- P, a set of productions of the form $a \rightarrow b$, where $a$ is a non-terminal and $b$ is a sequence of one or more symbols from $T \cup V$
- S, the start symbol, a member from N

A language is then defined via the concept of derivation and the basic operation is that of rewriting one sequence of symbols into another. If $a \rightarrow b$ is a production, we can rewrite any sequence of symbols which contains the symbol $a$, replacing $a$ by $b$. 
We denote this rewrite operation by the symbol \( \Rightarrow \) and read \( u \ a \ v \ \Rightarrow \Rightarrow \ u \ b \ v \) as: \( u \ b \ v \) directly derives from \( u \ a \ v \) or conversely, \( u \ a \ v \) directly produces/generates \( u \ b \ v \). For instance, given the grammar G1

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow PN \\
VP & \rightarrow Vi \\
PN & \rightarrow John \\
Vi & \rightarrow runs
\end{align*}
\]

the following rewrite steps are possible:

\[(D1) \ S \ \Rightarrow \ NP \ VP \ \Rightarrow \ PN \ VP \ \Rightarrow \ PN \ Vi \ \Rightarrow \ John \ Vi \ \Rightarrow \ John \ runs\]

which can be captured in a tree representation called a parse tree e.g.,:

ADD TREE

So \( John \ runs \) is a sentence of the language defined by G1. More generally, the language defined by a CFG is the set of terminal strings (sequences composed entirely of terminal symbols) which can be derived by a sequence of rewrite steps from the start symbol S.

### 5.1.1 Parsing

A CFG only defines a language. It does not say how to determine whether a given string belongs to the language it defines. To do this, a parser can be used whose task is to map a string of words to its parse tree.

When proceeding through the search space created by the set of parse trees generated by a CFG, a parser can work either top-down or bottom-up. A top-down parser builds the parse tree from the top, working from the start symbol towards the string. The example derivation \( D1 \) given above illustrates a top-down parsing process: starting from the symbol S, productions are applied to rewrite a symbol in the generated string until the input string is derived. By contrast, a bottom-up parser starts from the input string and applies productions in reverse until the start symbol is reached. So for instance, assuming a depth-first traversal of the search space and a left-to-right processing of the input string, a bottom up derivation of \( John \ runs \) given the grammar G1 would be:

\[
John \ runs \ \Rightarrow \ PN \ runs \ \Rightarrow \ PN \ V \ \Rightarrow \ NP \ V \ \Rightarrow \ NP \ VP \ \Rightarrow \ S
\]

The parse tree of course remains the same. In the remainder of this chapter, we show how to implement a bottom-up CF-parser in Oz.

### 5.2 Bottom-up Recognition/Parsing

The primary job of a parser/recognizer is to determine whether a sequence of terminal categories can be generated by the rules of the grammar. Bottom-up parsing is based on the idea of applying rules in reverse: i.e. given a rule \( C \rightarrow C1 \ldots Cn \), we can replace
by any sequence of categories matching \( C_1 \ldots C_n \). This is a non-deterministic process (choice of rule, choice of subsequence) which we can iterate. If by repeated applications of rules in reverse, we can arrive at a sequence of length 1 containing a single category \( C \), we have recognized \( C \).

Let’s illustrate the process on the sequence *john sees the man*. Given (lexical) rules recognizing the terminals:

\[
\begin{align*}
PN & \rightarrow \text{john} \\
VT & \rightarrow \text{sees} \\
DET & \rightarrow \text{the} \\
N & \rightarrow \text{man}
\end{align*}
\]

the sequence can be rewritten \( PN \ VT \ DET \ N \). Then we can recognize the two noun phrases:

\[
\begin{align*}
NP & \rightarrow \text{DET N} \\
NP & \rightarrow \text{PN}
\end{align*}
\]

obtaining \( NP \ VT \ NP \). Then the verb phrase:

\[
\text{VP} \rightarrow \text{VT NP}
\]

obtaining \( NP \ VP \). And finally a whole sentence:

\[
S \rightarrow \text{NP VP}
\]

Arriving at the singleton sequence \( S \). Thus, we have recognized \( S \).

### 5.2.1 Recognition as Inferential Closure

Let’s now make this intuitive algorithm a little more formal and describe it as an inference-based process. We write \( \text{STRING } [A_1 \ldots A_k] \) to state that we have a string of categories \( A_1 \ldots A_k \). We write \( \text{RULE } C \rightarrow C_1 \ldots C_n \) to state that the grammar contains a rule rewriting category \( C \) to the sequence of categories \( C_1 \ldots C_n \). Now consider the backward application of a rule. It can be formalized by the inference rule (schema) below:

\[
\begin{align*}
\text{STRING } [A_1 \ldots A_p C_1 \ldots C_n B_1 \ldots B_q] \\
\text{RULE } C \rightarrow C_1 \ldots C_n \\
\hline
\text{STRING } [A_1 \ldots A_p C B_1 \ldots B_q]
\end{align*}
\]

Above the line are the two premises, and below the line is the conclusion. Now, we turn to the case of a singleton sequence. We write \( \text{RECOGNIZED } C \) to state that we have recognized category \( C \). The recognition inference rule can be formulated as follows:

\[
\begin{align*}
\text{STRING } [C] \\
\hline
\text{RECOGNIZED } C
\end{align*}
\]

Thus the question of whether the sequence of terminals \( T_1 \ldots T_k \) is in the language generated by the grammar can be reformulated as the question of whether, for some \( C \), \( \text{RECOGNIZED } C \) is in the inferential closure of \( \text{STRING } [T_1 \ldots T_k] \).
5.2.2 Inferential Closure Algorithm

We now outline at a fairly abstract level an algorithm for computing the inferential closure of a set of Literals, given a set of inference rules. Given a literal $X$ and an inference rule $R$, we write $R(X)$ for the set of literals which can be derived by applying $R$ to $X$. Our first algorithm is:

- pick an element $X$ of Literals
- pick an inference rule $R$
- add $R(X)$ to Literals
- repeat until no new literal can be added

We can improve on this algorithm by distinguishing between literals which have already been processed (i.e. cannot contribute new conclusions) and those which are yet unprocessed. Thus, we distinguish two sets of literals: DONE and TODO. Our second algorithm is:

- initially set DONE to empty and TODO to Literals.
- while TODO is not empty:
  - remove an element $X$ from TODO
  - add $X$ to DONE
  - for each rule $R$, add $R(X)$ to TODO

The datastructure used to implement TODO is often called an agenda. It is typically either a stack or a queue.

5.3 A Bottom-Up Recognizer

We now show how to implement such a recognizer in Oz. We do this by implementing the inferential closure algorithm for grammatical rules. As such, this section is not merely about parsing: it really illustrates how to write an inferential closure algorithm.

5.3.1 Grammar Rules

First, we define the grammar rules. We distinguish phrasal rules like $NP \rightarrow DET N$ involving only non-terminals, and lexical rules like $DET \rightarrow the$ which are normally stated in a lexicon.

52a (Grammar Rules 52a) ≡

```
declare RULES = [ (Phrasal Rules 53a)
                   (Lexical Rules 53a) ]
```
What you see above is the definition of a named chunk of code. The name of this chunk is Grammar Rules and it uses two other named chunks, respectively called Phrasal Rules and Lexical Rules. The idea is that wherever a named chunk is used, it stands for the corresponding chunk of code stipulated in its definition, i.e. this chunk of code is substituted in its place.

Using code chunks allows us to present and discuss a program in small bits that are easier to understand. We will use this form of program presentation throughout the remainder of the course. It was invented by Donald Knuth under the name of Literate Programming. In the HTML version of the course, wherever a named chunk is used, you can click on it to jump to its definition. In the printed version, you should use the trailing reference number to lookup the definition.

A rule \( C \rightarrow C_1 \ldots C_n \) is represented by the record:

\[
\text{o(cat:C subcat:[C_1 \ldots C_n])}
\]

In the rules below \( \text{vi} \) stands for verb intransitive, \( \text{vt} \) for verb transitive, and \( \text{vd} \) for verb ditransitive.

53a **Phrasal Rules**

\[
\begin{align*}
&\text{o(cat:s subcat:[np vp])} \\
&\text{o(cat:np subcat:[det n])} \\
&\text{o(cat:np subcat:[pn])} \\
&\text{o(cat:np subcat:[np pp])} \\
&\text{o(cat:vp subcat:[vi])} \\
&\text{o(cat:vp subcat:[vt np])} \\
&\text{o(cat:vp subcat:[vd np np])} \\
&\text{o(cat:vp subcat:[vp pp])} \\
&\text{o(cat:pp subcat:[prep np])}
\end{align*}
\]

The lexicon contains rules where the subcat list consists of a single terminal category:

53b **Lexical Rules**

\[
\begin{align*}
&\text{o(cat:pn subcat:[john])} \\
&\text{o(cat:pn subcat:[mary])} \\
&\text{o(cat:det subcat:[the])} \\
&\text{o(cat:n subcat:[man])} \\
&\text{o(cat:vi subcat:[runs])} \\
&\text{o(cat:vt subcat:[likes])} \\
&\text{o(cat:vt subcat:[sees])} \\
&\text{o(cat:prep subcat:['with'])} \\
&\text{o(cat:n subcat:[telescope])}
\end{align*}
\]

5.3.2 Agenda

For are going to implement the agenda by a stack. These are provided as abstract data structure (ADS) as already discussed in Section on Modules (page 40) in the Chapter on functional programming (page 27):

53c **declare NewStack**

\[
\text{declare NewStack}
\]
Agendas are thus stacks. These are records whose programming interface is described by the following type:

\[
\text{unit}\{
    \text{push} : \text{Value} \rightarrow \\
    \text{pop} : \rightarrow \text{Value} \\
    \text{isEmpty}: \rightarrow \text{Bool} \\
    \ldots \\
\}
\]

If \(A\) is an agenda, then \(A\).push \(X\) adds item \(X\) to the agenda, \(A\).pop removes and returns an item from the agenda, and \(A\).isEmpty returns true iff there are no items in the agenda.

Here, we used a stack, but we could just as easily have used a queue. In fact, as long as we provide the same programming interface, we can use any implementation we please.

### 5.3.3 Bag

In order to accumulate results (recognized single categories), we define a second abstract datatype representing a bag of results. For simplicity, we implement bags by stacks, even though this is not very efficient. A better implementation that relies directly on cells is given in Section on Bags (page 79) in the Chapter on Stateful Data Structures (page 75):

\[
\text{declare NewBag}  \equiv \\
\{\text{declare NewStack}\}  \equiv \\
% \text{implement bags through stacks} \\
% \text{declare NewBag = NewStack}
\]

We will now exploit that stacks do also have a function that converts their actual contents into a list, i.e. we exploit bags with the following type (that is compatible with that of a stack).
5.3. A Bottom-Up Recognizer

If B is a bag, a new item X can be added to it with \{B.push X\}, and the list of its contents can be obtained with \{B.toList\}.

5.3.4 Parse

The bottom up recognizer is implemented by function Parse which takes as arguments a sequence of (terminal) categories Cats and a list of Rules, and returns the list of (single) categories which have been recognized. Cats is in the language iff the returned list is non-empty. The implementation follows exactly our algorithm: the main loop is realized by the recursive local procedure Process.

```plaintext
55a (Bottom Up Recognizer 55a)≡

fun {Parse Cats Rules}
    Bag = {NewBag}
    Agenda = {NewAgenda}
    proc {Process}
        if {Not {Agenda.isEmpty}} then
            Cats = {Agenda.pop}
            in
            %% recognition rule
            case Cats of [Cat] then {Bag.push Cat} else skip end
            %% apply all rules
            for Rule in Rules do {Infer nil Cats Rule} end
            %% iterate
            {Process}
        end
    end

{Bottom Up Infer 55a}
in
    {Agenda.push Cats}
    {Process}
    {Bag.toList}
end
```

Infer is the function which given a sequence of categories Cats and a grammar rule Rule, produces all conclusions according to the first inference rule (see Section 5.2.1). Recall that the rule schema was:

```
STRING [A1 ... Ap C1 ... Cn B1 ... Bq]
RULE C -> C1 ... Cn
```

```
---------------------------------------------------------------------------------------------------
STRING [A1 ... Ap C B1 ... Bq]
```
Therefore, given a string of categories and a rule \( C \rightarrow C_1 \ldots C_n \), we need to find all possible subsequences \( C_1 \ldots C_n \) in our string in order to produce all possible conclusions. We do this by going through our string, and, at each step, checking if what follows matches the sequence \( C_1 \ldots C_n \). Therefore, at each step, our string is divided into a Prefix that we have already looked at, and a Suffix which is everything that follows and hasn’t been looked at yet. For reasons of convenience and efficiency, Prefix is actually in reversed order: this is why we have \( \{\text{Reverse Prefix}\} \) in the code below. In order to check whether the rule is applicable to the Suffix, we simply check whether \( C_1 \ldots C_n \) (i.e. the subcat list) is a prefix of Suffix.

\[56a\]

\[
\text{Bottom Up Infer}\begin{align*}
&\text{proc}\{\text{Infer}\ \text{Prefix}\ \text{Suffix}\ \text{Rule}\} \\
&\% \text{ does the rule apply here?} \\
&\text{if}\ \{\text{IsPrefix}\ \text{Rule.subcat}\ \text{Suffix}\}\ \text{then} \\
&\% \text{ add the conclusion to the agenda} \\
&\text{Agenda.push} \\
&\{\text{Append}\ \{\text{Reverse Prefix}\} \\
&\text{Rule.cat}\|\{\text{List.drop}\ \text{Suffix}\ \{\text{Length}\ \text{Rule.subcat}}\}\}\}
\end{align*}
\]

\% rewritings for the rest of the sequence 
\text{case } \text{Suffix} \\
\text{of } \text{nil } \text{then } \text{skip} \\
[\ ] \text{H|T then } \{\text{Infer } \text{H|Prefix T Rule}\}
\end{align*}
\]

In order to check whether \( \text{Rule.subcat} \) is a prefix of \( \text{Suffix} \), we define the function \text{IsPrefix}. The Mozart library contains the function \text{List.isPrefix} which is exactly what we need, but, for didactic purposes, we will also ask you to implement this function in an exercise.

\[56b\]

\[
\text{declare IsPrefix}\begin{align*}
&\text{declare}\ \text{IsPrefix}\ =\ \text{List.isPrefix}
\end{align*}
\]

### 5.3.5 The Complete Program

\[56c\]

\[
\text{Bottom Up Recognizer Program}\begin{align*}
&\text{declare NewAgenda}\begin{align*}
&\text{declare NewBag}\begin{align*}
&\text{declare IsPrefix}\begin{align*}
&\text{Grammar Rules}\begin{align*}
&\text{Bottom Up Recognizer}\begin{align*}
&\text{TestSuite}\begin{align*}
&\text{Parse TestSuite}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\]

The complete program also contains a simple test suite:

\[56d\]

\[
\text{TestSuite}\begin{align*}
\end{align*}
\]
5.3. A Bottom-Up Recognizer

\[
\text{local} \\
S1 = \{\text{john runs}\} \\
S2 = \{\text{man runs}\} \\
S3 = \{\text{john sees the man}\} \\
\%\% \text{the next two tests need more time to finish} \\
\%\% \text{as you would like to happen.} \\
S4 = \{\text{the man 'with' the telescope sees john}\} \\
S5 = \{\text{john sees the man 'with' the telescope}\} \\
\text{in} \\
\text{TestSuite} = [S1 \ S2 \ S3 \ S4 \ S5] \\
\text{end}
\]

The examples in the test suite can be parsed as follows. At first sight, the parser might seem to not come back for examples 3 and 4. This is not true: it returns but needs much more time you might expect (since the algorithm used is too stupid, see below).

\[
\text{Parse TestSuite} =
\text{for } \text{Sentence in TestSuite do} \\
\text{Inspect \{Parse Sentence RULES\}} \\
\text{end}
\]

Here is the complete code of the bottom up recognizer, also available in file Bottom_Up_Recognizer_Program.oz:

\[
\%\% \text{maybe you want to change this URL to} \\
\%\% \text{the appropriate local filename} \\
declare \text{URL} = 'http://www.ps.uni-sb.de/~niehren/Web/Vorlesungen/Oz-NL-SS01' \\
declare \text{ADS_URL} = \text{URL} \\
\%\% \text{load the module with the abstract data structures} \\
declare \text{ADS_Module} = \{\text{Module}\}.\text{link}\ [\text{ADS_URL}] \\
\%\% \text{select the abstract data structure for the stack} \\
declare \text{NewStack} = \text{ADS_Module}.\text{newStack} \\
\%\% \text{implement agendas through stacks} \\
declare \text{NewAgenda} = \text{NewStack} \\
\%\% \text{implement bags through stacks} \\
declare \text{NewBag} = \text{NewStack} \\
declare \text{IsPrefix} = \text{List}.\text{isPrefix} \\
declare \text{RULES} = [\text{o(cat:s subcat: [np vp])} \\
\text{o(cat:np subcat: [det n])} \\
\text{o(cat:np subcat: [pn])} \\
\text{o(cat:np subcat: [np pp])} \\
\text{o(cat:vp subcat: [vi])} \\
\text{o(cat:vp subcat: [vt np])} \\
\text{o(cat:vp subcat: [vd np np])} \\
\text{o(cat:pp subcat: [prep np])} \\
\text{o(cat:pn subcat: [john])}]
\]

\footnote{\text{code/Bottom_Up_Recognizer_Program.oz}}
fun {Parse Cats Rules}
    Bag = {NewBag}
    Agenda = {NewAgenda}
proc {Process}
    if {Not {Agenda.isEmpty}} then
        Cats = {Agenda.pop}
    in
        %% recognition rule
        case Cats of [Cat] then {Bag.push Cat} else skip end
        %% apply all rules
        for Rule in Rules do {Infer nil Cats Rule} end
        %% iterate
        {Process}
    end
end
proc {Infer Prefix Suffix Rule}
    %% does the rule apply here?
    if {IsPrefix Rule.subcat Suffix} then
        %% add the conclusion to the agenda
        {Agenda.push}
        {Append {Reverse Prefix}
         Rule.cat[(List.drop Suffix {Length Rule.subcat})]}
    end
    %% rewritings for the rest of the sequence
    case Suffix of nil then skip
        [] H | T then {Infer H | Prefix T Rule} end
    end
    in
        {Agenda.push Cats}
        {Process}
        {Bag.toList}
    end
local
    S1 = [john runs]
    S2 = [man runs]
    S3 = [john sees the man]
    %% the next two tests need more time to finish
    %% as you would like to happen.
    S4 = [the man ’with’ the telescope sees john]
5.4 Experiments, Critique, Improvements and Extensions

5.4.1 Redundant Derivations

Let's now try our parser:

\{Inspect \{Parse \[man\] RULES\}\}

This displays \[n man\]. These are all the single categories which have been recognized. Indeed, since we start with the single category \textit{man}, it is recognized, and, after application of lexical rule \textit{n} \(\rightarrow\) \textit{man}, the single category \textit{n} is also recognized. Now let's try:

\{Inspect \{Parse \[the man\] RULES\}\}

This displays \[np np\]. Indeed, according to our algorithm, there are two ways to arrive at the single category \textit{np}:

\[
\begin{align*}
\text{the man} & \Rightarrow \text{det man} \Rightarrow \text{det n} \Rightarrow \text{np} \\
\text{the man} & \Rightarrow \text{the n} \Rightarrow \text{det n} \Rightarrow \text{np}
\end{align*}
\]

Oops! One way to fix this problem is to avoid inferring twice the same conclusion. For example, in the 2nd derivation above, when we infer \textit{det n}, we should notice that we already produced this conclusion earlier in the 1st derivation and discard it instead of processing it again to redundantly infer \textit{n}. You will have to do this as an exercise.

A different and better way to address this problem is to try to share the recognition of subsequences between derivations using memoization. In our example: both derivations reduce \textit{man} to \textit{n}, \textit{the} to \textit{det}, and \textit{det n} to \textit{np}. This approach leads to the idea of \textit{Chart Parsing} and will be the subject of a later chapter.

5.4.2 Building Parse Trees

Our recognizer only returns a list of recognized categories. It would be nicer if it returned a list of parse trees. We will represent a parse tree as a record whose label is the recognized category. For example, the parse tree for \[\text{[the man]}\] can be represented by:

\[
\text{np(det (the) n(man))}
\]
and for \([\text{john sees the man}]\) by:

\[s(np(pn(john)) \ \text{vp}(vt(sees) \ np(det(the) \ n(man))))\]

In order to apply rule \(np \rightarrow \text{det} \ n\) to the sequence of trees \([\text{det(the) n(man)}]\), it is necessary to modify \texttt{IsPrefix} to check the labels of the trees against the categories of the rule's body (exercise). It is also necessary to modify \texttt{Infer} to replace the matched subsequence by a new parse tree instead of just by the recognized category:

\[
\text{proc } \{ \texttt{Infer Prefix Suffix Rule} \}
\]

\[
\% \text{ does the rule apply here?}
\]

\[
\text{if } \{ \texttt{IsPrefix Rule.subcat Suffix} \} \ \text{then} \ \text{Args Rest Tree in}
\]

\[
\{ \texttt{List.takeDrop Suffix \{Length Rule.subcat\} Args Rest} \}
\]

\[
\% \text{ assemble new parse tree}
\]

\[
\text{Tree=\{List.toTuple Rule.cat Args\}}
\]

\[
\% \text{ add the conclusion to the agenda}
\]

\[
\{ \texttt{Agenda.push \{Append \{Reverse Prefix\} Tree\Rest} \}
\]

\[
\text{end}
\]

\[
\% \text{ rewriterings for the rest of the sequence}
\]

\[
\text{case Suffix}
\]

\[
\text{of } \texttt{nil then skip}
\]

\[
\{ [] \} \texttt{H|T then } \{ \texttt{Infer H|Prefix T Rule} \}
\]

\[
\text{end}
\]

The library procedure \texttt{List.takeDrop} combines the functionality of \texttt{List.take} and \texttt{List.drop}:

\[
\{ \texttt{List.takeDrop L N PREFIX SUFFIX} \}
\]

binds \texttt{PREFIX} to the list of the \texttt{N} first elements of \texttt{L} and \texttt{SUFFIX} to the rest of \texttt{L}. A parser that avoids redundant derivations and produces parse trees will return the following two trees for \([\text{john sees the man with the telescope}]\), thus properly accounting for PP attachment ambiguity:

\[
\{ s(np(pn(john)))
\]

\[
\text{vp(vt(sees)}
\]

\[
\text{np(np(det(the) n(man)))}
\]

\[
\text{pp(prep(with) np(det(the) n(telescope))))}
\]

\[
\text{s(np(pn(john))}
\]

\[
\text{vp(vp(vt(sees) np(det(the) n(man)))}
\]

\[
\text{pp(prep(with) np(det(the) n(telescope))))}
\]

\[
\text{]}\]

\[
\text{5.5 Exercises}
\]

\[
\bullet \text{ Write a function } \texttt{IsPrefix} \text{ taking 2 list arguments and returning true iff the first is a prefix of the second. Replace the definition of } \texttt{IsPrefix} \text{ in the bottom-up parser with yours and test the parser.}
\]
• Write a function `Drop` which takes a list `L` as first argument and an integer `N` as second argument, ignores the first `N` elements of `L` and returns the remainder of the list. For example `{Drop [a b c d] 0}` should return `[a b c d]` and `{Drop [a b c d] 2}` should return `[c d]`. The Mozart library contains function `List.drop`: you should not use it in this exercise. Then use your `Drop` instead of `List.drop` in the bottom-up parser.

• Write a function `Take` which takes a list `L` as first argument and an integer `N` as second arguments and returns the first `N` elements of `L`. For example `{Take [a b c d] 0}` should return `nil` and `{Take [a b c d] 2}` should return `[a b]`. The Mozart library contains function `List.take`: you should not use it in this exercise.

• In addition to `List.drop` and `List.take`, the Mozart library contains function `List.takeDrop` that combines the two functionalities:

```
local Prefix Suffix in
    {List.takeDrop [a b c d e] 2 Prefix Suffix)
    {Inspect Prefix
    {Inspect Suffix}
end
```

As brain teaser (but not a course exercise), see if you can figure out a way to do this by defining your own `TakeDrop` procedure. The library procedure achieves it in `N` steps (where `N` is its 2nd argument): can you do as well?

• Modify the bottom-up recognizer to discard conclusions which have already been derived before. Hint: it may be helpful to take advantage of the abstract datatype provided by `NewBag`.

• Modify the bottom-up recognizer into a parser which returns a list of parse trees rather than a list of recognized categories only. See Section 5.4.2 for hints. If possible, use your own `Drop` and `Take` functions, or your `TakeDrop` procedure if you were able to define it.
Loops, Exception Handling, and Threads

We now discuss loops in some more detail. We then introduce two new control concepts: exception handling and threads. Exception handling can be used to quit loops or recursive procedures. It is also important for error handling. Threads are the most important ingredient for concurrent processes. Concurrency is ubiquitously useful: for functional, internet, and constraint programming. Concurrent threads are fully supported by Oz in contrast to many other programming languages, including high-level languages such as SML and Lisp.

6.1 Loops

We have already introduced loops to recurse over lists. We now discuss loops in some more details. We also discuss how to implement loops through recursive procedures, as done in Oz.

6.1.1 Basic Loops

Oz has a looping construct introduced by the keyword `for`. The simplest form is to iterate over the elements of a list:

\[
\text{for } X \text{ in } [a \ b \ c] \text{ do } \text{Inspect } X \text{ end}
\]

One can think of \( X \) as a mutable variable that successively takes values of the list \([a \ b \ c]\). You can also use special syntax to write something loops over integers:

\[
\text{for } I \text{ in } 2..25 \text{ do } \text{Inspect } I \text{ end}
\]

This has the same effect as if \( I \) ranged over the integer list \([2 \ldots 25]\) except that this list is never constructed explicitly.
### 6.1.2 Loops by Recursion

Conceptually, we can think of the variable of a loop to be mutable. But of course, variables are not mutable in Oz. So how are these two statement compatible?

Another way of looking at loops is as recursive procedures. These procedures apply the body of the loop to all elements of a list the loop iterates over. For instance, the loop above becomes the following procedure:

```oz
local
proc(AboveLoop Xs)
  case Xs
  of nil the skip
  | X | Xss then {Inspect X} {AboveLoop Xss}
  end
end
in
{AboveLoop [a b c]}
end
```

Indeed, this is that way that loops are implemented in Oz. The expansion of loops into the core syntax turns them into recursive procedures.

### 6.1.3 Parallel Iteration

The loop concept of Oz also permits to iterate over several lists in parallel.

```oz
for
  X in [a b c d]
  I in 1..3
do
  {Inspect X#I}
end
```

This means that \( x \) and \( i \) are always set to the next possible value simultaneously. Executing the above loop thus prints:

```
a#1
b#2
c#3
```

Note that there are mixed pairs such as \( a#2 \) or \( b#1 \) of the full cross product in the output. Note also, that the iteration of the loop stops, once one of the two list over which the iteration runs becomes empty.
6.1.4 Quit a Loop

Sometimes it is also very convenient to exit a loop in the middle of the iteration, and to jump to the follow up statement. This can be done with loops having a break feature. This feature gives access to a Break procedure by which to jump out of the loop.

The break feature is also very useful to write while or until loops:

```plaintext
while Test do Statement ==>
   for break:Break
   do
      if Test
      then Statement
      else {Break}
   end
end
```

6.1.5 Loops and State

Loops are not only inspired by stateful programming; they are also particularly useful in the context of stateful data structures. For instance, we can use stacks to accumulate the results of a loop:

```plaintext
65a \begin{align*}
\text{reverse } A &= \langle \text{declare NewStack } 3 \rangle \\
&\text{declare fun (Reverse List)} \\
&\text{Acc = } \{\text{NewStack}\} \\
&\text{in} \\
&\text{for } X \text{ in List do } \{\text{Acc.push } X\} \text{ end} \\
&\{\text{Acc.toList}\} \\
&\{\text{Inspect } \{\text{Reverse } [1 2 3 4]\}\}
\end{align*}
```

6.2 Example: Finite Automata

Let A be a finite set whose element we call letters. A finite automaton over A consists of a finite set Q, whose elements we call states, a finite set of single letter transitions \( D \subseteq Q \times A \times Q \), a set of initial states \( I \subseteq Q \) and a set of final states \( F \subseteq Q \).

6.2.1 Membership

Every automaton \( (A, Q, D, I, F) \) accepts a subset of words over A, i.e. a set of lists with elements in A. A word \([l_1 \ldots l_p]\) \( \in A^* \) belongs to that language if the word \([l_1 \ldots l_p]\) can reach a final state when starting from a initial state.

\[
q \text{ in } \text{reach}([l_1 \ldots l_p]) \quad q \text{ in } F
\]

--------------------------------------

\[ [l_1 \ldots l_p] \text{ in Language} \]
The empty word \( \text{word} \) can reach the initial state.

\[
\begin{align*}
q \text{ in } I \\
------------------------
q \text{ in } \text{reach}(\text{nil})
\end{align*}
\]

If a word \([l_1 \ldots l_p]\) can reach state \(q_1\) and there is a transition \((q_1, l, q_2) \in D\) then \([l_1 \ldots l_p l]\) can reach \(q_2\)

\[
\begin{align*}
(q_1, l, q_2) \text{ in } D & \quad q_1 \text{ in } \text{reach}([l_1 \ldots l_p]) \\
------------------------&
q_2 \text{ in } \text{reach}([l_1 \ldots l_p l])
\end{align*}
\]

We can easily represent a finite automaton as a record. It is also not difficult to test for membership:

\[
\begin{align*}
\text{declare} & \quad \text{Automaton} = \text{unit}(\text{trans:unit}(p: \text{unit}(1: [q] 2: \text{nil})) \\
& \quad \quad \quad \quad \quad q: \text{unit}(1: [p] 2: [q r]) \\
& \quad \quad \quad \quad \quad r: \text{unit}(1: \text{nil} 2: \text{nil})) \\
& \quad \quad \quad \quad \quad \quad \text{init:}[p] \\
& \quad \quad \quad \quad \quad \quad \text{fin:}[q])
\end{align*}
\]

\[
\begin{align*}
\text{fun} \{ \text{Reach Word} \} & \% \text{ returns a set of states} \\
& \quad \text{Result} = \{\text{NewSet}\} \\
& \quad \text{in} \\
& \quad \quad \text{if} \ \text{Word} == \text{nil} \\
& \quad \quad \quad \text{then} \\
& \quad \quad \quad \quad \text{for} \\
& \quad \quad \quad \quad \quad \text{State} \ \text{in} \ \text{Automaton}.\text{init} \\
& \quad \quad \quad \quad \quad \text{do} \\
& \quad \quad \quad \quad \quad \{\text{Result}.\text{add State}\} \\
& \quad \quad \quad \quad \end{align*}
\]

\[
\begin{align*}
& \quad \quad \text{else} \\
& \quad \quad \quad \text{Prefix#LastLetter} = \{\text{Split Word}\} \\
& \quad \quad \text{in} \\
& \quad \quad \quad \text{for} \\
& \quad \quad \quad \quad \text{Statel} \ \text{in} \ \{(\text{Reach Prefix}).\text{content}\} \\
& \quad \quad \quad \quad \text{do} \\
& \quad \quad \quad \quad \quad \text{for} \\
& \quad \quad \quad \quad \quad \quad \text{State} \ \text{in} \ \text{Automaton}.\text{trans}.\text{Statel}.\text{LastLetter} \\
& \quad \quad \quad \quad \quad \quad \text{do} \\
& \quad \quad \quad \quad \quad \quad \{\text{Result}.\text{add State}\} \\
& \quad \quad \quad \quad \end{align*}
\]

\[
\begin{align*}
& \quad \quad \text{end} \\
& \quad \quad \text{end} \\
& \quad \quad \text{ResultSet} \\
& \quad \text{end} \\
\end{align*}
\]

\[
\begin{align*}
\text{fun} \{ \text{Accept Word} \}
\end{align*}
\]
6.3 Exception Handling

Exceptions can be used to quit recursive procedures or loops. They are also very useful for error handling.

6.3.1 Basic Statements

Exceptions are values that are thrown when executing a `raise` statement.

```
raise exception(4711) end
```

The effect of throwing an exception is that recursive calls are quit and that all follow up statements are ignored. Exceptions can be caught by exception handlers that are marked by the keywords `try`.

```
try ... catch Exception then ... end
```

If an exception handler catches an exception then the evaluation continues with the code behind its `then` continuation.
6.3.2 Quit Recursion

We now illustrate how to quit a recursive call by throwing and catching an exception. The example is a simple recursion that increments a counter but quits the recursion once it gets equal to 10000.

\[
\text{\texttt{\{quit recursion\}}} \equiv \begin{array}{l}
declare\ proc(P,cess\ N) \\
\quad \text{if } N==10000 \\
\quad \quad \text{then raise 'error in Process' end} \\
\quad \else \{P,cess\ N+1\} \\
\quad \end{array}
\]

\[
\text{try} \\
\quad \{P,cess\ 0\} \\
\text{catch Exception then} \\
\quad \{\text{Browse Exception}\} \\
\text{end}
\]

6.3.3 Quit Loops

The break feature of loops is implemented by using exception handling.

\[
\text{for} \\
\quad \ldots \\
\quad \text{break:Break} \\
\text{do} \\
\quad \ldots \text{\{Break\} \ldots} \\
\text{end}
\]

The above loop is replaced by an statement of the following form:

\[
\text{try} \\
\quad \text{for} \\
\quad \ldots \\
\quad \text{do} \\
\quad \ldots \text{raise break end \ldots} \\
\text{end} \\
\text{catch Exception then} \\
\quad \text{case Exception of} \text{ break} \\
\quad \quad \text{then skip} \\
\quad \quad \text{else raise Exception end} \\
\quad \text{end}
\]

6.4 Concurrent Threads and Logic Variables

Concurrency is a way to organize computation based on the notion of concurrent processes. Concurrency is well-known from operating systems like UNIX which support
multi-tasking in order to administrate multiple windows each of which runs in its own process. Oz supports concurrent computation on a high level of abstraction. The present-
ation of concurrency in this reader stays at the very surface of the phenomenon.

A process in Oz is called a *thread*. A thread is created when executing a sequences of Oz-statement sequentially. A thread may block until more information becomes available. At first sight blocking may seem to be a programming error. For instance, consider:

```oz
declare F X={F 2}
{Browse 'this thread blocks'}
{Browse variables(x:X f:F number:1)}
```

When feeding this piece of code at once, nothing is browsed. The problem is that the value of the variable `F` is unknown such that the application of `{F 2}` has to block. All followup statements of the same thread (code sequence) are also blocked until the free variable `F` gets assigned a value (i.e. gets bound).

Using the programming interface, you can easily feed another sequence of statements which then computes concurrently in its own thread.

```oz
F=fun($ Y) Y*Y end
```

Now, the value of `F` has become known. Thereby, the first thread become active again and could executed its remaining two Browse-statements.

You can also create your own threads without using the Oz-Programming-Interface. This can be done by using the command:

```oz
thread ...
```

For instance, the above example can be rewritten such that the blocking application does not block the subsequent statements.

```oz
declare X F
thread
 X={F 2}
{Browse 'this thread blocks ...'}
{Browse variables(x:X f:F number:1)}
{Browse '... but not forever'}
end
{Browse 'this thread does NOT block'}
F=fun($ Y) Y*Y end
```

This example illustrates the creation of a new thread which first blocks until the free variable `F` gets bound by the main thread which runs concurrently to its newly sporned thread.

Threads in Oz threads communicate over shared logic variables which play the same role such as channels in CML or PICT. In Oz, you can also consider a thread as a hand-
written propagator which adds information about the value of variables to a shared constraint store.
6.5 Debugger

A debugger is a tool which helps you to find programming errors. The idea of a debugger is that you can follow the programs execution interactively. At each time point you get informed about:

- the set of actual threads, one of which is selected by the debugger.
- the actual stack of procedure calls of the actual thread
- what is known about the values of the variables of the actual thread
- which statement or expression of the actual thread is actually executed.

Code has to be compiled differently to make all information accessible to the debugger. This may have a lot of consequences. For instance, it may happen that an error does no more occur or that new errors arise.

You can use the debugger from the interactive user interface. It sees all those chunks of codes that were fed after the debugger was started. This means that you may have to fed some code again in order to recompile it so that it becomes accessible to the debugger.

You can also use the debugger to observe error free programs. Try it for instance with the following program to see the stack of procedure calls that is computed.

```plaintext
%% (Factorial N) = N * ... * 1
declare fun{Factorial N}
    if N==0
    then 1
    else N*(Factorial N-1)
    end
end
{Inspect {Factorial 3}}
{Inspect {Factorial 1111}}
```

Next use the debugger to find the error in the following program:

```plaintext
declare C = {NewCell unit}
{Inspect {Access C}}
{Assign C 5}
declare Old in {Exchange C Old Old+1}
{Inspect Old}
```

6.6 Exercises

- Write a procedure for the function FoldL that uses the loop construct and state.
  Hint: You can abuse a stack to define a data structure for a cell that contains a mutable value. You may also use cells directly.
- Write another definition of \texttt{FoldL} that uses tail recursion but neither loops nor state.

- Reduce an until loop to a while loop.

- Write a functional procedure that tests whether a word is accepted by a finite automaton. Is there an algorithm that can do so with linear run time in the size of the word and the given automaton?

- Write a functional procedure that tests whether the language accepted by a finite automaton is empty or not. Can you do it in linear time in the size of the automaton?

- Something about threads, state, and indeterminism.

- Something about exception handling.

\section*{6.7 Program Collection}

Here is the collection of all the programs:

\begin{verbatim}
71a (program collection control \texttt{71a})

% % loops
% % list loop
(list loop \texttt{52a})

% % integer loop
(integer loop \texttt{52b})

% % loop by recursion
(loop by recursion \texttt{54a})

% % parallel iteration
(parallel iteration \texttt{54b})

% % accumulation in stateful data structures
(reverse \texttt{55a})

% % Exception handling

% % quit recursion
(quit recursion \texttt{58a})
\end{verbatim}

\begin{verbatim}
71a (program collection control \texttt{71a})

% % loops

% % list loop
(list loop \texttt{52a})

% % integer loop
(integer loop \texttt{52b})

% % loop by recursion
(loop by recursion \texttt{54a})

% % parallel iteration
(parallel iteration \texttt{54b})

% % accumulation in stateful data structures
(reverse \texttt{55a})

% % Exception handling

% % quit recursion
(quit recursion \texttt{58a})
\end{verbatim}
% list loop
for X in [a b c] do {Inspect X} end

% integer loop
for I in 2..25 do {Inspect I} end

% loop by recursion
local proc {AboveLoop Xs}
  case Xs of nil the skip
  | X|Xss then {Inspect X} {AboveLoop Xss} end
in {AboveLoop [a b c]}
end

% parallel iteration
for X in [a b c d] I in 1..3 do {Inspect X#I} end

% accumulation in stateful data structures
% maybe you want to change this URL to
% the appropriate local filename

declare URL = 'http://www.ps.uni-sb.de/~niehren/Web/Vorlesungen/Oz-NL-SS01'
declare ADS_URL = URL#'/vorlesung/Functors-Version.3.2/Abstract.ozf'
declare [ADS_Module] = {Module.link [ADS_URL]}
declare NewStack = ADS_Module.newStack
declare fun {Reverse List}
  Acc = {NewStack} in
    for X in List do {Acc.push X} end
  {Acc.toList} end
{Inspect {Reverse [1 2 3 4]}}

% Exception handling
%% quit recursion
declare proc(Process N)
if N==10000
then raise 'error in Process' end
else {Process N+1}
end
end
try
{Process 0}
catch Exception then
{Browse Exception}
end
Stateful Data Structures

In this chapter, we take a closer look at stateful data structures. We discuss the primitives offered by Oz and how to derive your own abstract data structures with them. These can also be seen as stateful objects that are created from classes.

7.1 State in Abstract Data Structures

One the one hand side, stateful data structures can support simpler algorithms and help to achieve better complexity. On the other hand side, stateful programming notoriously bears the dangers: It might easily lead a programmer to confuse conceptual and implementation concerns during software development: a program should never become more difficult to maintain or understand only because stateful programming is used to make it efficient.

Another independent problem with stateful programming is that it may quickly get painful in concurrent programs. The potential problem is that different orders of state changes may lead to different computation result.

Both problems can be addressed by encapsulate state manipulations into abstract data structures as we will always do in what follows. Abstract data structures separate the specification of stateful procedures from their implementation. Abstract data structure not only makes it easier to replace specific implementations of data structures. They also support top down programming development, where one first specifies the functions one needs before implementing them.

We have already seen stateful programming with abstract data structures in the previous section. There, we used predefined stacks to implement agendas and bags. Agendas gave us a simple way to administrate the intermediate results of a recursive process. Bags allowed us to collect the set of results of a process with multiple return values.

Oz offers stateful concepts through four independent primitives: cells, dictionaries, arrays, and objects. All of them are useful, even though each one were sufficient as a basis of stateful programming in principle. Stacks and dictionaries will be most important throughout this book. Cells will only be used to implement stacks and queues.

7.2 Object versus Library Style

There is no doubt that you should always try to abstract when using a new data data structure (rather than leaving it implicit in some code). So is is clearly useful to in-
troduce our own data abstract data structures if the Oz libraries do not already provide them for us.

Beside of this, we will also introduce the object style for abstract data structures, in contrast to the the Oz library style. Most importantly, both styles help us to avoid naming conflicts: One can easily reuse the same name for different procedures of different abstract data structures.

The object style is well known from object-oriented programming. The difference to the Oz library style is quite easy: In the object style, every instance of an abstract data structures owns all its functions by itself. In the library style, in contrast, there are common functions for all instances of the same type of abstract data structures.

So why do we do mostly use the object style instead of the Oz libraray style? This is mainly a matter of convenience. Let us consider an example to clarify the difference. If we have an instance History of the abstract stack data structure then it is simply shorter to write

{History.push 'what I did today'}

in object style than

{Stack.push History 'what I did today'}

in the Oz library style.

Finally, note that we will not make use of the special Oz object system. This system comes with a new syntax and its own implementation. This means that it mainly introduces yet another programming language within Oz. It does not give use any more expressiveness or convenience than we already have (even though it has some nice features too). We will give an example for how to use the Oz object syntax at the end of this section.

7.3 Cells

A cell is a container that contains a mutable value. Cells are supported by the Oz base environment. We first present the library functions for cells and then show how to turn them into an abstract data structure.

7.3.1 Library Style

We will frequently use cells to implement abstract data structures. In this case, we can use the Oz library functions directly.

You can create a new cell by calling the function Cell.New whose argument specifies the initial content of your cell. You can always look up the actual content of a cell applying the function Access. The content can be changed by using the procedure Assign.
There is a procedure `Exchange` which combines `Access` with `Assign` into an atomic operation. It is called in the form:

```
{Exchange C ?Old ?New}
```

The Old value is access before the new one is assigned. It is important that `Exchange` is atomic in the presence of concurrent processes: this means that the exchange operation cannot be interrupted by any other concurrent process. Otherwise, the old value might get corrupted while the new is computed. Such an accident could easily turn the state of your whole program inconsistent.

But how can one simultaneously access the old value and assign the new value if the new value is to be computed in dependency of the old value? The trick in Oz is that one can always describe the new value by a free logic variable whose value can then be computed later on from the old value.

### 7.3.2 Object Style

We now change from the Oz library style to that of object oriented programming. Cell objects are modelled by records of procedures that give access to a hidden library cell. The type of a cell object is:

```
unit (get : -> value
    put : value ->
    exchange: value -> value
    ...)
```

Cell objects can be created by calling a creation function `NewCellObject` (the cell’s class) while passing the initial value.
This functor defines a module which exports the function \texttt{NewCellObject} at feature \texttt{new}.

The procedures of cell objects are defined by reduction to the corresponding library functions. Essentially, they do nothing else than applying the library functions to the concrete hidden cell.

But note that we also did some more changes and less systematic changes for our convenience. We turn the binary library procedure \texttt{Cell.exchange} into a function \texttt{Exchange} that inputs a value to be put into the cell and outputs the old of the cell.

Furthermore, we have also adapted the name of the functions for convenience. We use \texttt{put} and \texttt{get} as usual in the library rather than \texttt{Access} and \texttt{Assign}. Finally, we have turned \texttt{Exchange} into a function that inputs the new value and returns the old one. Note that the input argument \texttt{New} is often left variable in applications. We can now use bags by linking a compiled version of the module \texttt{Bag.ozf} that is made available at the following URL:

```oz
exchange:fun{$ New} 
  Old 
  in 
  {Exchange C Old New} 
  Old 
end
access:Get
assign:Put)
end
end
```

This allows you to test the above implementation of cell objects.
7.4 Bags

Bags are finite multisets of values, i.e sets in which elements may occur multiply. Stated otherwise, a bag is a function from values to numbers, which maps each value to the number of its occurrences in the bag. The type of a bag object is:

```
unit{put :value=>
toList:->list
member:value=>bool)
```

7.4.1 Library Style

We can implement bag abstract data structures through cells and lists. This is more efficient than using stacks for the same purpose.

```
79a (BagLib.oz 79a)≡
functor
export
  new : NewBag
  fromList: ListToBag
  Put ToList Member
define
  fun {NewBag}{Cell.new nil} end
  fun {ListToBag L} {Cell.new L} end
  proc{Put B X} L in {Exchange B L X L} end
  fun {ToList B} {Access B} end
  fun {Member B X} {List.member X {ToList B}} end
end
```

7.4.2 Object Style

We now lift the bags in library style to the object style. This strictly follows the usual pattern.

```
79b (Bag.oz 79b)≡
functor
import
  Bag(fromList put toList member) at 'BagLib.ozf'
export
  new : NewBagObject
  fromList: ListToBagObject
define
  fun {ListToBagObject L}
     B = {Bag.fromList L}
   in
     unit{put : proc{$ X} {Bag.put B X} end
           toList: fun {$ } {Bag.toList B} end
           member: fun {$ X} {Bag.member B X} end)
  end
```
Chapter 7. Stateful Data Structures

fun {NewBagObject}
{ListToBagObject nil}
end
end

We can now use bags by linking a compiled version of the module Bag.ozf that are made available at the usual URL. The function Bag.new creates a new bag object B. Elements can be added to B by invoking B.put. A bag B is converted into a list by calling B.toList:

```
80a (Test Bag.oz 80a)≡
declare BagURL= URL#'/Bag.ozf'
declare [Bag] = {Module.link [BagURL]}
declare B = {Bag.new}
{B.put hello}
{B.put world}
{Inspect {B.toList}}
```

7.5 Dictionaries

A dictionary is a bit like a modifiable record: it associates features to values, but you can add new features and remove or modify old ones. This is by far the most useful and most frequently used stateful data structure in Oz.

7.5.1 Library Style

Dictionaries are supported directly by the Oz base library. They are implemented by hash tables that can be extended dynamically. The type of dictionary keys is

```
type key = (atom + int)
```

Given this type, we can now define the type of a dictionary object:

```
unit {toRecord : -> record
  isEmpty : -> bool
  entries : -> list(key#value)
  items : -> list(value)
  put : key x value ->
  get : key -> value
  condGet : key x value -> value
  member : key -> bool
  toKeys : -> list(key)
  remove : key ->
  removeAll: ->
  collect : key x value -> ...
}
```

A dictionary is created using NewDictionary. You can add new features using the := operator.
7.5. Dictionaries

81a (Create Dictionary from Library 81a)

```plaintext
declare D={NewDictionary}
D.foo := baz
D.hello := world
D.42 := fourtytwo
```

Now the dictionary contains the 3 features foo, hello and 42. You can look up the value associated with a feature using the dot notation and modify their association using := operator. The statement `{CondSelect D Feature Default}` performs a lookup of Feature in D much like the dot operation, but, if the Feature does not exist, it returns a Default instead. It is possible to check whether a feature exists using HasFeature:

81b (Test Dictionary from Library 81b)

```plaintext
{Inspect D.hello}
D.hello := universe(D)
{Inspect {CondSelect D foo fooNotFound}}
{Inspect {CondSelect D baz bazNotFound}}
{Inspect {HasFeature D foo}}
{Inspect {HasFeature D baz}}
```

The procedure CondSelect and HasFeature also work on records and arrays. You can find what features, items (i.e. values), and entries (feature/item pairs) are in D using respectively:

```plaintext
{Dictionary.keys D} ==> [hello 42 foo]
{Dictionary.items D} ==> [universe(D) fourtytwo baz]
{Dictionary.entries D} ==> [hello#universe(D) 42#fourtytwo foo#baz]
```

A feature can be deleted using Dictionary.remove:

```plaintext
{Dictionary.remove D foo}
```

7.5.2 Object Style

81c (Dictionary.oz 81c)

```plaintext
functor
  export
    new:NewDict
  define
    D = Dictionary
fun{NewDict}
    Dict = {D.new}
proc{Put Key Val}
    Dict.Key := Val
end
fun{Get Key}
    Dict.Key
end
```
fun[Member Key]
   {D.member Dict Key}
end
fun[Keys]
   {D.keys Dict}
end
fun[ToRecord]
   {D.toRecord unit Dict}
end
fun[ToRecordLabel Label]
   {D.toRecord Label Dict}
end
fun[CondGet Key Default]
   {CondSelect Dict Key Default}
end
proc[Remove Key]
   {D.remove Dict Key}
end
proc[RemoveAll]
   {D.removeAll Dict}
end
fun[Entries]
   {D.entries Dict}
end
fun[Items]
   {D.items Dict}
end
fun[IsEmpty]
   {D.isEmpty Dict}
end
proc[Collect Key Val]
   {Put Key Val | (CondGet Key nil)}
end

in
unit(put :Put
     get :Get
     condGet :CondGet
     toRecord :ToRecord
     isEmpty :IsEmpty
     entries :Entries
     items :Items
     dict :Dict
     toKeys :Keys
     toRecordLabel :ToRecordLabel
     condSelect :CondSelect
     member :Member
     keys :Keys
     remove :Remove
     removeAll :RemoveAll
)

7.6 Stacks

Stacks can be easily implemented by using cells and lists. Therefore, they are not directly provided by the Oz base environment. The type of a stack object is:

```oz
unit(push : value : ->
    pop : -> value
    put : value ->
    get : -> value
    top : -> value
    size : -> int
    isEmpty: -> bool
    clear : ->
toList : -> list
... )
```

7.6.1 Library Style

The file We now implement a module StackLib.oz in the library style. A pre-compiled version of StackLib.ozf is available on-line at Functors/Version.3.2/StackLib.ozf.

```
functor
export
    new : NewStack
    fromStack : ListToStack
    Get Put Top IsEmpty Clear ToList Push Pop
define
    fun {NewStack} {Cell.new nil} end
    fun {ListToStack L} {Cell.new L } end

    fun {Get C} Old New in
        (Exchange C Old New)
        case Old of nil then New=nil raise empty end
        [] H T then New=T H
        end
    end
    Pop = Get

    proc {Put C X} Old New in New=X Old {Exchange C Old New} end
    Push = Put

    fun {Top C}
```

1Functors/Version.3.2/StackLib.ozf
Chapter 7. Stateful Data Structures

7.6.2 Object Style

We now turn the library stacks into an abstract data structure $Stack.ozf Module/. A precompiled version of $Stack.ozf$ is available on-line at Functors/Version.3.2/Stack.ozf odd fun functor
import
Stack(new fromList get put top
isEmpty clear toList push pop) at 'StackLib.ozf'
export
new : NewStackObject
fromList : ListToStackObject
define
fun (NewStackObject) {ToObject (Stack.new)} end
fun (ListToStackObject L) {ToObject (Stack.fromList L)} end
fun (ToObject S)
  fun (Get ) {Stack.get S } end
  proc (Put X) {Stack.put S X} end
  fun (Top ) {Stack.top S } end
  fun (IsEmpty) {Stack.isEmpty S } end
  proc (Clear ) {Stack.clear S } end
  fun (ToList ) {Stack.toList S } end
  proc (Push X) {Stack.push S X} end
  fun (Pop ) {Stack.pop S } end
in
unit(
  get : Get
  put : Put
  top : Top
  isEmpty : IsEmpty
  clear : Clear
toList : ToList
push : Push
pop : Pop)
end
end

We can now use stacks by linking a compiled version of the module $Stack.ozf$ from the usual URL.

Functors/Version.3.2/NewStackObject.ozf
7.7 Queues

We now define queues in library style and then turn them into abstract data structures. The type of a queue object is:

```
unit (put : value ->
  get : -> value
  top : -> value
  size : -> int
  isEmpty : -> bool
  clear : ->
  toList : -> list
... )
```

7.7.1 Library Style

The module QueueLib.ozf Module/ provides queues in library style. A precompiled version of QueueLib.ozf is available on-line at Functors/Version.3.2/QueueLib.ozf

85a (QueueLib.oz 85s) =

functor export
  new : NewQueue
  fromList : ListToQueue
  Get Put Top Size IsEmpty Clear ToList Enq Deq
define
  fun (NewQueue) L in {NewCell 0#L#L} end
  fun (ListToQueue L1) L2 in {NewCell (Length L1)#{Append L1 L2}#L2} end
  proc {Put Q X}
    Old New
    in
      (Exchange Q Old New)
      case Old
        of N#H#T then T2 in T=X|T2 New=N+1#H#T2 end
    end
    Enq = Put

  fun (Get Q)
    Old New
    in
      (Exchange Q Old New)
      case Old
        of 0#_#_ then New=Old raise empty end
        [] N#(X|H)#T then New=N-1#H#T X end
    end
    Deq = Get

3Functors/Version.3.2/QueueLib.ozf
fun (Top Q)
case (Access Q)
of 0#_#_ then raise empty end
[1] _#(X) #_ then X end
end

fun (Size Q) (Access Q).1 end
fun (IsEmpty Q) (Access Q).1==0 end
proc (Clear Q) L in (Assign Q 0#L#L) end

fun (ToList Q)
case (Access Q) of N#L#_ then
  for
    I in 1..N
    X in L
    collect:C
do
  {C X}
end
end
end
end

7.7.2 Object Style

We also turn queues into an abstract data structure Queue.ozf
Module/.

A precompiled version of Queue.ozf is available on-line at Functors/Version.3.2/Queue.ozf

86a <Queue.oz 86a>≡
  functor
  import
    Queue(new fromList size get put top
      isEmpty clear toList enq deq ) at 'QueueLib.ozf'
  export
    new : NewQueueObject
    fromList : ListToQueueObject
  define
    fun (NewQueueObject) {ToObject (Queue.new)} end
    fun (ListToQueueObject L) {ToObject (Queue.fromList L)) end
    fun (ToObject Q)
      fun (Size ) {Queue.size Q } end
      fun (Get ) {Queue.get Q } end
      proc (Put X) {Queue.put Q X} end
      fun (Top ) {Queue.top Q } end
      fun (IsEmpty) {Queue.isEmpty Q } end
      proc (Clear ) {Queue.clear Q } end
      fun (ToList ) {Queue.toList Q } end
  end

4Functors/Version.3.2/QueueAbs.ozf
7.8 Arrays

An array is like a sequence of cells where each position in the sequence can be referenced by an integer index. Arrays are directly supported by the Oz library.

7.8.1 Library Style

An array is created using `NewArray`:

```overture
declare A={NewArray LowIndex HighIndex InitValue}
```

where indices will range from `LowIndex` to `HighIndex` inclusive, and the initial value of each cell is `InitValue`. You can look up the value contained in the cell at position `I` as follows:

```overture
A.I
```

where `I` is an index between `LowIndex` and `HighIndex` inclusive. Note that this is the same notation as feature access. Alternatively you can use procedure `Get`. The value contained in said cell can be changed to `foo` using:

```overture
A.I := foo
```

Note that this combines the dot notation with the new operator `:=`. Alternatively you can use procedure `Put`.

7.8.2 Object Style

to be written
7.9 Module Abstract.oz

We now collect all abstract data structures in a separate module Abstract.oz. A precompiled version of Abstract.ozf is available on-line at Functors/Version.3.2/Abstract.ozf.

```oz
functor
import
Cell( new:NewCell) at 'Cell.ozf'
Bag( new:NewBag fromList:ListToBag) at 'Bag.ozf'
Dictionary(new:NewDictionary) at 'Dictionary.ozf'
Stack( new:NewStack fromList:ListToStack) at 'Stack.ozf'
Queue( new:NewQueue fromList:ListToQueue) at 'Queue.ozf'
end
```

7.10 Oz Object System

We now illustrate the Oz object system by which we can also implement stacks and queues. For missing explanation about Oz object system, we refer to the Oz documentation.

```oz
class StackClass from BaseObject
attr stack:nil
meth init stack <- nil end
meth push(E) stack <- E|@stack end
meth pop($)
    Stack = @stack
in
    stack <- Stack.2
    Stack.1
end
meth isEmpty($)
    case @stack
    of nil then true
    else false
```

Functors/Version.3.2/Abstract.ozf


```ruby
end
end
meth toList($) @stack end end

89a 〈Stack Object 89a）≡
unit{push :proc{$ X} {Stack push(X)} end
pop :fun{$} {Stack pop($)} end
init :proc{$} {Stack init} end
isEmpty:fun{$} {Stack isEmpty($)} end
toList:fun{$} {Stack toList($)} end)

89b 〈NewStack 88b）≡
local
( StackClass 88b)
in
fun{NewStack}
Stack = {New StackClass init}
in
( Stack Object 89a)
end
end

89c 〈Test Object System Stack 89c）≡
declare
〈NewStack 88b）
Stack = {NewStack}
in
{Inspect Stack}
{Inspect {Stack.isEmpty}}
{ForAll [1 2 3 4 5] Stack.push}
{Inspect {Stack.toList}}
```

### 7.11 Exercises

- Write an abstract data-structure to represent finite sets. Your data type should at least provide the functionality of adding an element to the set and testing whether an element is in the set.

- Implement another abstract data-structure to represent finite sets. This time, however, assume the all elements of your sets belong to a given finite subset of atomic values.

- Reduce an until loop to a while loop.

- Write an abstract data structure `Counter` in object style which supports the operations `increment`, `decrement`, and `toInt`.  


- Represent the formulas of propositional logics as terms. The labels of these
terms should subsume

\[
\text{and } \text{or} \text{ implies } \text{not} \text{ var}
\]

and other atoms for propositional variables. Then write a function that maps
all propositional variables occurring in propositional formula to a unique number
you assign to it. Hint: dictionaries and counters might help.

### 7.12 Program Collection

Here is the collection of all the programs:
7.12. Program Collection

%%% cell library function
%%%cell library function

\begin{verbatim}
declare C = (Cell.new unit)

{Inspect {Access C}}
{Assign C hi}
{Inspect old#{Access C}}
{Inspect old#{Exchange C $ _}}
{Inspect new#{Access C}}
{Access C} = ‘the new value of C was bound late’
\end{verbatim}

%%%cell objects
%%%cell objects

\begin{verbatim}
declare AuthorURL = ‘http://www.ps.uni-sb.de/~niehren’
CourseDir = ‘/Web/Vorlesungen/Oz-NL-SS01’
CourseURL = AuthorURL#CourseDir
PickleVersion = ‘Version.3.2’
DataDir = ‘/vorlesung/Functors’#PickleVersion
URL = CourseURL#DataDir

declare CellURL = URL#/‘Cell.ozf’
declare [CellMod] = {Module.link [CellURL]}
declare C={CellMod.new 2002}

{Inspect {C.get}}
{C.put 4711}
{Inspect {C.put}}
declare NewVal OldVal = {C.exchange NewVal}

{Inspect OldVal}
NewVal=OldVal-42
{Inspect {C.get}}
\end{verbatim}

%%% bags
%%% bags

\begin{verbatim}
declare BagURL = URL#/‘Bag.ozf’
declare [Bag] = {Module.link [BagURL]}
declare B = (Bag.new}
{B.put hello}
{B.put world}
{Inspect {B.toList}}
\end{verbatim}

%%% dictionary
%%% dictionary

\begin{verbatim}
declare D={NewDictionary}
D.foo := baz
\end{verbatim}
D.hello := world
D.42 := fortytwo
{Inspect D.hello}
D.hello := universe(D)
{Inspect {CondSelect D foo fooNotFound}}
{Inspect {CondSelect D baz bazNotFound}}
{Inspect {HasFeature D foo}}
{Inspect {HasFeature D baz}}

declare Abstract = functor
import
Cell( new:NewCell) at 'Cell.ozf'
Bag( new:NewBag fromList:ListToBag) at 'Bag.ozf'
Dictionary(new:NewDictionary) at 'Dictionary.ozf'
Stack( new:NewStack fromList:ListToStack) at 'Stack.ozf'
Queue( new:NewQueue fromList:ListToQueue) at 'Queue.ozf'
export
newStack : NewStack
listToStack : ListToStack
newQueue : NewQueue
listToQueue : ListToQueue
newAgenda : NewStack
listToAgenda : ListToStack
newBag : NewBag
listToBag : ListToBag
newCell : NewCell
newDictionary: NewDictionary
end

local
class StackClass from BaseObject
attr stack:nil
meth init stack <- nil end
meth push(E) stack <- E|@stack end
meth pop($)
  Stack = @stack
in
  stack <- Stack.2
  Stack.1
end

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% collection of object style data structures
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Oz object system: stacks and queues
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

declare


meth isEmpty($)
  case @stack
  of nil then true
  else false
  end
end
meth toList($) @stack end
end

in
fun{NewStack}
  Stack = {New StackClass init}
  in
    unit(push : proc {$ X} {Stack push(X)} end
    pop : fun{$} {Stack pop($)} end
    init : proc{$} {Stack init} end
    isEmpty: fun{$} {Stack isEmpty($)} end
    toList : fun{$} {Stack toList($)} end)
  end
end
Stack = {NewStack}

in

{Inspect Stack}
{Inspect {Stack.isEmpty})
(ForAll [1 2 3 4 5] Stack.push)
{Inspect {Stack.toList})
Functors and Modular Programming

The Mozart system makes data types and their operations available through modules. For example module `Dictionary` exports all the library functions for creating and operating on dictionaries. A module is simply a record, e.g.:

- `Dictionary.new` creates a new dictionary
- `Dictionary.get` returns the value associated with a key in the dictionary, etc...

Some modules such as `List` and `Dictionary` are always available: they are part of the Base Environment. Others need to be explicitly loaded; we have seen an example of this earlier in the course using `Module.link` (see Section 5.3.2).

In this chapter, we will show you how to create your own modules and what the advantages are to do so. We will introduce the notions of functors and of modular programming, and as an illustration, we will rewrite, in a modular fashion, the context-free parsers of Chapter 5.

8.1 Functors as values

A functor is data-structure that describes a module. In the same way that `proc {Foo} ... end` constructs a procedure, `functor Foo ... end` constructs a functor. For example:

```plaintext
declare
functor Foo
export
  greetings : Greetings
define
  Greetings = 'hello world'
end
```

Let’s take a look at this value by invoking `{Inspect Foo}`:
What we see is that a functor is an Oz value known as a *chunk*. A chunk is like a record in that it has features, but unlike records which have structural equality, chunks have token equality; that is, each chunk is unique and distinct from any other chunk, even if they have the same features and values.

### 8.1.1 Features of functors

Feature `import` is a record with label `import` describing the module’s imports: we will see later what these are, but here there none, so the record is empty.

Feature `export` is a record with label `export` describing what is exported by the module. Here, there is only one feature called `greetings` and the value associated with it in the export record is `atom`, which is simply some type information inferred by the compiler.

Feature `apply` contains a function to create the module’s exports given its imports. In the case of our functor above, the function is essentially the following:

```
declare
fun { $ Imports}
  Greetings = ‘hello world’
in
  ‘exports’(greeting:Greetings)
end
```

### 8.1.2 Functors as abstract datatypes

Functors are abstract datatypes that are concretely realized using chunks. In order to be able to safely distinguish functors from other chunks, a special feature is added to the representation: a chunk is a functor iff it has this feature. The feature, displayed here as `<N:functorID>` is an Oz name. This is a common technique for implementing safe abstract datatypes, i.e. datatypes which cannot be forged/faked.

For example, we have seen before an implementation of a bag:

```
fun {NewBag}
  C={NewCell nil}
in
  bag(put : proc { $ X} L in (Exchange C L X|L) end
  tolist : fun { $} {Access C} end)
end
```

We might test whether a given value is a bag by looking at the label of the record, but that’s a test which is easy to fool. Instead, we can create a private name, known only to the implementation of the abstract datatype:

```
local
  CELL_ID = {NewName}
in
  fun (NewBag)
    C={NewCell nil}
```
NewChunk is a library function which takes a record as an argument and returns a chunk with the same features. Why doesn’t NewBag return the record directly? Here is why:

```
declare R=foo(a:1) {Inspect {Arity R}}
```

By invoking Arity, anyone could obtain all the features of the record, including CELL_ID which was supposed to be private. By design, it is not possible to obtain the arity of a chunk, which makes chunk ideal for implementing type-safe abstract datatypes.

### 8.1.3 Module creation = functor linking

A functor is a description of a module. In order to actually obtain this module, the functor must be linked: this is the job of a module manager. We can create a module from the Foo functor we defined earlier as follows:

```
declare [FooModule] = {Module.apply [Foo]}
(Inspect FooModule)
```

### 8.2 Imports

So far, we have either used modules directly available in the interactive environment or explicitly obtained new ones by invoking Module.link or Module.apply. When writing functors, a better alternative is available: using import declarations. Let’s write a module that exports a Greet procedure which prints `hello world`:

```
declare
functor Foo
import
```
Chapter 8. Functors and Modular Programming

System at ‘x-oz://system/System.ozf’
export
greet : Greet
define
   proc {Greet}
       {System.showInfo "hello world!"}
   end
end

Let’s have a look at this functor value with \{Inspect Foo\}:

As we can see, the functor indicates that the module it describes exports a nullary procedure on feature greet and that it imports one module from URI ‘x-oz://system/System.ozf’. Let’s try it out immediately thus:

\declare [M]={Module.apply [Foo]}
{M.greet}

The import section begins with the keyword import and contains one or more import specifications. Here there is only one:

System at ‘x-oz://system/System.ozf’

it states that variable System should be bound to the module which can be constructed from the functor at the URI identified by atom ‘x-oz://system/System.ozf’. Instead of an abstract URI, you could also have a file name or a URL. For libraries which are part of the Mozart system, a shorter form of imports is also supported:

import System

is exactly equivalent to:

import System at ‘x-oz://system/System.ozf’

That is: the name of the variable is automatically used to construct the appropriate URI. For modules not part of the Mozart library you must use the long form. Also for exports, there is a similar kind of short form:

export Greet
is exactly equivalent to:

```plaintext
export greet : Greet
```

The name of the variable is used to infer the name of the export feature by downcasing its first letter. Thus our functor can be more simply written:

```plaintext
declare functor Foo
import System
export Greet
define
  proc {Greet}
    {System.showInfo "hello world!"}
  end
end
```

## 8.3 Persistent Functors

Not only can you create functors interactively, but you can also save them in files. We call this ‘making functors persistent’. Consider our earlier functor Foo. We can save it to file /tmp/Foo.ozf as follows:

```plaintext
{Pickle.save Foo "/tmp/Foo.ozf"}
```

A persistent value is also called a pickle, hence the name Pickle for the module providing the API for persistent values. Note the extension ozf for the file in which we saved our functor: this is the extension conventionally chosen for persistent functors. The advantage of a persistent functor is that it can be reused:

```plaintext
declare [M]={Module.link ["/tmp/Foo.ozf"]}
{M.greet}
```

Even more interestingly, it can be reused as an import in an other module:

```plaintext
declare functor Baz
import GreetModule at "/tmp/Foo.ozf"
export GreetTwice
define
  proc {GreetTwice}
    {GreetModule.greet}
    {GreetModule.greet}
  end
end

declare [M]={Module.apply [Baz]}
{M.greetTwice}
```
8.4 Functor Programs

so far, all our functors were created interactively. The better way to develop software with Mozart is to write one functor per file. In this fashion, each file describes one module in your application. We will call a file that contains the code of a functor a ‘functor program’. The syntax of a functor program omits the variable following the functor keyword. A functor program for our greeting module is:

```oz
functor
import System
export Greet
define
  proc {Greet}
    {System.showInfo "hello world!"}
  end
end
```

Let us write it to a file called Greet.oz. The functor value corresponding to this program is obtained by compiling the file. In general compilation of Oz code is the job of the Mozart compiler ozc. However, we will instead promote a simpler and more general tool called ozmake. It is invoked from the command line:

```
ozmake Greet.ozf
```

automagically ‘builds’ Greet.ozf by compiling Greet.oz: Greet.oz contains a functor program and after compilation Greet.ozf contains the corresponding persistent functor value.

8.4.1 ozmake

You will have to download and install ozmake on your machine. While ozmake is available in source form from The MOGUL Archive, for your convenience prebuilt executables are also provided:

- ozmake for Unix
- ozmake.exe for Windows

Also, it might help to have a look at the ozmake documentation page.

---

2. [http://www.ps.uni-sb.de/~duchier/mogul/pub/pkg/ozmake](http://www.ps.uni-sb.de/~duchier/mogul/pub/pkg/ozmake)
8.5 Functors at Abstract URIs

If my compiled functor is stored in file /home/denys/oz-kurs/Greet.ozf then I must also import it from this file, i.e. explicitly using this pathname, and my programs will not work for you when you download them to your computer. Hardwiring filenames is a bad idea. A much better idea is to use an abstract name rather than a specific concrete one, for example:

\[ \text{x-ozlib://oz-kurs/examples/Greet.ozf} \]

Such an abstract name is called a URI (Uniform Resource Identifier), and the mapping from URI to specific files (or URLs) is performed automatically by the Mozart resolver. We will not go into the details of the resolver, but rather, we will show you how to take advantage of it by installing packages and by developing your own projects as packages.

All the Mozart system modules use URIs of the form x-oz://... For the modules developed in this course, we will use URIs of the form x-ozlib://oz-kurs/... 

8.5.1 A Simple Makefile

Let us turn our greeting library into a package. To this end, we need to provide ozmake with a description of our package. Such a description is called a makefile and is conventionally stored in a file called makefile.oz. This contains a record describing the package. For our greeting library we write the following makefile:

\[
\text{makefile}(
\begin{array}{l}
\text{lib} : ['Greet.ozf'] \\
\text{uri} : 'x-ozlib://oz-kurs/examples' \\
\text{mogul} : 'mogul:/oz-kurs/denys/greet'
\end{array}
)\]

The \text{lib} feature lists the compiled functors provided by the package. The \text{uri} feature gives the base URI for our libraries: thus our Greet.ozf functor will be available at URI:

\[ \text{x-ozlib://oz-kurs/examples/Greet.ozf} \]

The \text{mogul} feature uniquely identifies this package. Why do we need both feature \text{uri} and \text{mogul}? Because it is often useful to have several packages that install libraries under the same URI: feature \text{mogul} identifies the package, while feature \text{uri} gives the namespace for its libraries. Now, simply invoking:

\[
\text{ozmake}
\]

consults the makefile and builds Greet.ozf if necessary, but more importantly:

\[
\text{ozmake --install}
\]
installs our package so that the greeting module can be imported from the URI mentioned above. Let’s try this command now:

```bash
% ozmake --install
ozc -c Greet.oz -o Greet.ozf
mkdir /home/denys/.oz/cache/x-ozlib/oz-kurs
mkdir /home/denys/.oz/cache/x-ozlib/oz-kurs/examples
cp Greet.ozf /home/denys/.oz/cache/x-ozlib/oz-kurs/examples/Greet.ozf
```

ozmake echoes the commands that it executes. Here, it compiled Greet.oz to produce Greet.ozf, then created the directories to install the libraries in my personal Oz area (each user has their own), and finally copied Greet.ozf to its final destination as an installed library. You too can install this package and we all can use the greeting library it provides by importing it from the same abstract URI:

```
declare [M]={Module.link ['x-ozlib://oz-kurs/examples/Greet.ozf']}
(M.greet)
```

### 8.6 Creating a Package

With packages, you can easily take advantage of libraries offered by other people: it suffices to download and install the package on your own computer; ozmake was primarily designed to simplify this process. Conversely, you can create your own packages that others can download and install. In this section, we illustrate the creation and installation of packages using our greeting library as an example.

#### 8.6.1 Package Creation

File `makefile.oz` entirely describes our greeting package. It is also a good idea to state who the author of the package is, so let’s update `makefile.oz` as follows:

```oz
makefile(
  lib : [ 'Greet.ozf' ]
  uri : 'x-ozlib://oz-kurs/examples'
  mogul : 'mogul:/oz-kurs/denys/greet'
  author: 'Denys Duchier'
)
```

Now we can create a `package file` as follows:

```bash
% ozmake --create --package=denys-greet.pkg
writing denys-greet.pkg
```

A new file called `denys-greet.pkg` was written and it contains everything needed for the package.

---

5 actually ozmake echoes an idealization of the commands that it executes. This idealization shows the Unix commands that would typically be executed to obtain the desired effect. The actual implementation is designed to be portable across platforms: in particular, it needs to work on Windows.
8.6.2 Package Installation

Any one can now use file denys-greet.pkg in order to install our package. Let’s try it now:

```
% ozmake --upgrade --package=denys-greet.pkg
mkdir /tmp/fileXc39F9
writing /tmp/fileXc39F9/Greet.oz
ozc -c /tmp/fileXc39F9/Greet.oz -o /tmp/fileXc39F9/Greet.ozf
cp /tmp/fileXc39F9/Greet.ozf /home/denys/.oz/cache/x-ozlib/oz-kurs/examples/Greet.ozf
rm -R /tmp/fileXc39F9
```

I gave the option argument `--upgrade` rather than `--install` because we already installed the package once before (see Section 8.5.1). ozmake creates a temporary directory in which to extract the contents of the package, build it and install it. Finally, it removes the temporary directory when it is done.

8.6.3 Package Download

Having created a package, we can make it available to others by putting it on the web. Let me illustrate this by copying the package to my web area on my computer:

```
cp denys-greet.pkg ~/public_html/denys-greet.pkg
```

Now, others can download and install this package. In fact, ozmake also simplifies such a task because it allows the user to supply a URL as the package argument to be installed:

```
% ozmake --upgrade --package http://fox.ps.uni-sb.de/~denys/denys-greet.pkg
mkdir /tmp/file7C8Bs2
downloading package http://fox.ps.uni-sb.de/~denys/denys-greet.pkg
writing /tmp/file7C8Bs2/Greet.oz
ozc -c /tmp/file7C8Bs2/Greet.oz -o /tmp/file7C8Bs2/Greet.ozf
cp /tmp/file7C8Bs2/Greet.ozf /home/denys/.oz/cache/x-ozlib/oz-kurs/examples/Greet.ozf
rm -R /tmp/file7C8Bs2
```

8.7 Refactoring the Naïve Parser

The recommended way to develop a Mozart application is to organize it as a collection of functors. This is generally known as modular programming. In this section, we are going to illustrate this style of software development by revisiting the naïve parser of Chapter 5.

A considerable advantage of modular programming is that it permits and encourages the development of reusable components. Collections of reusable software components are often called libraries. In Chapter 5, we have seen that both the trivial recognizer and the more interesting parser that builds parse trees actually share most of their code. We are going to write functors that provide the common functionality and abstract away from application specific processing.
8.7.1 Course Libraries

For your convenience, we make available libraries developed specifically for this course. In particular, the support for stacks, queues, agendas, and their versions as abstract datatypes, which was first introduced in Chapter 7. These libraries are provided as package `oz-kurs.pkg` which you should install using `ozmake`. For example, download this package into a local file `oz-kurs.pkg`, then invoke at a command prompt:

```plaintext
ozmake --install --package=oz-kurs.pkg
```

Currently the package contains modules `Stack.ozf` (page 83), `Queue.ozf` (page 85), `Stack.ozf` (page 84), and `Queue.ozf` (page 86), as well as the more convenient `Abstract.ozf` (page 88).

8.7.2 NaiveClosure.ozf: Inferential Closure

The primary engine of our naïve parser is the computation of the inferential closure of an initial set of literals under application of a set of inference rules. There is nothing specific to parsing here and the question is then: how can we usefully abstract the notion of inference rule?

For our present purpose it suffices to model an inference rule as a function which takes a premise as argument and returns a list, possibly empty, of conclusions. Function `Closure` takes as inputs a list of `Literals` and a list of `Rules` where each element of the latter is a function as just described. `Closure` returns the corresponding inferential closure in the form of a list of literals.

The literals of the inferential closure are incrementally accumulated in a bag called `Inferred`. Notice also that we import agenda and bag functionality from the course library identified by URI `x-ozlib://oz-kurs/Abstract.ozf`.

```plaintext
104a (NaiveClosure.ozf) =
  functor
  export Closure
  import
    LIB(newAgendaFromList : NewAgendaFromList
        newBag    : NewBag)
    at 'x-ozlib://oz-kurs/Abstract.ozf'
  define
    fun (Closure Literals Rules)
      Agenda   = (newAgendaFromList Literals)
      Inferred = (newBag)
      in
      for break:Break do
        if (Agenda.isEmpty) then {Break}
        else Lit = {Agenda.get} in
          for Rule in Rules do
            for Conc in {Rule Lit} do
```

```
"oz-kurs.pkg"
The code above illustrate another aspect of the syntax of functors. So far, we had seen import specification of the form:

```oz
import LIB at 'x-ozlib://oz-kurs/Abstract.ozf'
```

We can be more precise, and state precisely what we wish to import from module `LIB`:

```oz
import LIB(newAgendaFromList newBag) at 'x-ozlib://oz-kurs/Abstract.ozf'
```

which says that we intend to use only features `newAgendaFromList` and `newBag` from module `LIB`. An advantage of being so specific is that the compiler will produce an error if we attempt to use another feature of `LIB`. Why is this useful? Because it catches many typos, for example in the following code:

```oz
{LIB.newAgendaFrmList L}
```

where I typed `Frm` instead of `From`. For further convenience, we can also write:

```oz
import LIB(newAgendaFromList : NewAgendaFromList
newBag : NewBag) at 'x-ozlib://oz-kurs/Abstract.ozf'
```

where we additionally introduce variables for some or all features. Thus we can use `NewBag` instead of `LIB.newBag`.

### 8.7.3 NaiveParser.ozf: Grammatical Processing

The naïve recognizer processes sequences of categories and a rule replaces a subsequence by a single category. The naïve parser processes sequences of parser trees and replaces a subsequence of parse tree by a new parse tree constructed using the elements of this subsequence. In terms of processing, the recognizer and the parser differ in only two operations:

- First, in the way a rule determines whether it matches a subsequence. We will abstract this operation as the function `IsPrefix`.  
Second, in the function that maps a subsequence to its replacement. We will abstract this operation as the function `MakeTree`.

The standard technique for abstracting away from specific operations is to pass them as arguments. Functor `NaiveParser.oz` exports function `MakeParser` which takes 3 arguments: `IsPrefix` and `MakeTree` as described above, and `Rules` which is a list of grammar rules in the format:

\[
o(cat:s subcat:[np vp])
\]

`MakeParser` returns a parser, i.e. a function which takes a list of `Words` (a sequence of terminal categories) as input and returns the categories or parse trees (depending on `MakeTree`) that have been recognized.

First, the `Rules` are converted into inference `RULES` appropriate for module `NaiveClosure.ozf`. Then `Parse` computes the infential closure of the initial literal `Words` under this set of `RULES`. The result of `Parse` is the list of conclusions which are sequences consisting of a single element (a category for the recognizer, a parser tree for the parser).

Now, we must transform a grammar `Rule` of the form `rule(cat:s subcat:[np vp])` into an inference rule, i.e., a function that takes a sequence of `Cats` as input and returns a list where subsequences matching the `Rule.subcat` list have been replaced according to `Rule.cat`. This list will be accumulated in a `Bag` and the job of recognizing matching subsequences is performed by the auxiliary procedure `Infer`. Notice that `Infer` is passed `Bag.put` in order to accumulate new conclusions.
8.7. Refactoring the Naïve Parser

(107a) \( \text{(NaiveParser MakeInferenceRule)} \)

\[
\text{fun } \{ \text{MakeInferenceRule Rule} \}
\]

\[
\text{fun } \{ \text{($ Cats)} \}
\]

\[
\text{Bag=}{\text{NewBag)}
\]

\[
\text{in}
\]

\[
\{ \text{Infer nil Cats Rule Bag.put)}
\]

\[
\{ \text{Bag.toList)}
\]

\[
\text{end}
\]

\[
\text{end}
\]

Finally, \( \text{Infer} \) is exactly as we defined it in Section 5.4.2, except that we invoke the more abstract \( \text{MakeTree} \) rather than the specific \( \text{List.toTuple} \).

(107b) \( \text{(NaiveParser Infer)} \)

\[
\text{proc } \{ \text{Infer Prefix Suffix Rule PutConclusion}}
\]

\[
\text{if } \{ \text{(IsPrefix Rule.subcat Suffix)}} \text{ then Args Rest Tree in}
\]

\[
\{ \text{List.takeDrop Suffix \{Length Rule.subcat\} Args Rest)}
\]

\[
\text{Tree } = \{ \text{MakeTree Rule Args)}
\]

\[
\{ \text{PutConclusion \{Append \{Reverse Prefix\} Tree\}Rest(}\}
\]

\[
\text{end}
\]

\[
\text{case Suffix}
\]

\[
\text{of nil then skip}
\]

\[
[] \H T \text{ then } \{ \text{Infer H \{Prefix T Rule PutConclusion}}\}
\]

\[
\text{end}
\]

end

8.7.4 \( \text{naive-makefile.oz} \)

Let us now take advantage of \texttt{ozmake} and turn \texttt{NaiveClosure.ozf} and \texttt{NaiveParser.ozf} into libraries available through a package. To this end, we must write a makefile which we will place in a file called \texttt{naive-makefile.oz}.

(107c) \( \text{(naive-makefile.oz)} \)

\[
\text{makefile(}
\]

\[
\text{lib} : [’NaiveClosure.ozf’ ’NaiveParser.ozf’]
\]

\[
\text{uri} : ’x-ozlib://oz-kurs/naive’
\]

\[
\text{mogul} : ’mogul:/oz-kurs/denys/naive’
\]

\[
\text{author} : ’Denys Duchier’)
\]

We can now install our libraries as follows:

\[
\text{ozmake --install --makefile=naive-makefile.oz}
\]

which illustrates another feature of \texttt{ozmake}, namely the possibility to specify a makefile other than the default \texttt{makefile.oz} using a \texttt{--makefile} option argument. We can also create a package \texttt{denys-naive.pkg} as follows:

\[
\text{ozname --create --makefile=naive-makefile.oz --package=denys-naive.pkg}
\]
Using the function `MakeParser` offered by our new library, we are going to implement both the naive recognizer and the naive parser producing parse trees. First, we acquire the library module from its abstract URI:

\[\text{(Using the refactored library \(108a\)) = }\]

```oz
\begin{verbatim}
declare
[Parser] = (Module.link ['x-ozlib://oz-kurs/naive/NaiveParser.ozf'])
\end{verbatim}
```

\[\text{(Grammar Rules \(108a\))} \]

\[\text{(Instantiating the naive recognizer \(108b\))} \]

\[\text{(Instantiating the naive parser \(108c\))} \]

For the recognizer, `IsPrefix` is the library function for lists and `MakeTree` simply returns the `Rule`’s `cat`.

\[\text{(Instantiating the naive recognizer \(108b\)) = }\]

```oz
\begin{verbatim}
%% recognizer
declare
IsPrefix1 = List.isPrefix
fun {MakeTree1 Rule Cats} Rule.cat end
NaiveRecognizer = {Parser.makeParser IsPrefix1 MakeTree1 RULES}
\end{verbatim}
```

\[\text{(Inspect \{NaiveRecognizer \[john sees the man with the telescope\]\})} \]

For the parser, `IsPrefix` must look at the labels of the record representing the parse trees. You already did this as an exercise. `MakeTree` simply packs the recognized subsequence into a new tree using `List.toTuple`:

\[\text{(Instantiating the naive parser \(108c\)) = }\]

```oz
\begin{verbatim}
%% parser
declare
fun {IsPrefix2 Cats Trees} case Cats of nil then true
[] Cat|Cats then case Trees of Tree|Trees then Cat=={Label Tree} andthen {IsPrefix2 Cats Trees} else false end
else false end end
fun {MakeTree2 Rule Trees} {List.toTuple Rule.cat Trees} end
NaiveParser = {Parser.makeParser IsPrefix2 MakeTree2 RULES}
\end{verbatim}
```

\[\text{(Inspect \{NaiveParser \[john sees the man with the telescope\]\})} \]
8.8 Exercises

- Write a file \texttt{SetAbs.oz} containing a functor program for the set abstract datatype that you previously developed in an exercise (see Section 7.11). The functor should export the constructor procedure on feature \texttt{new}. Install \texttt{ozmake} on your computer and use it to compile \texttt{SetAbs.oz} to produce \texttt{SetAbs.ozf}. The shell command to achieve this is:

\begin{verbatim}
ozmake --build SetAbs.ozf
\end{verbatim}

or more simply:

\begin{verbatim}
ozmake SetAbs.ozf
\end{verbatim}

Verify that this worked by obtaining the \texttt{SetAbs} module in the OPI from file \texttt{SetAbs.ozf} using \texttt{Module.link}. Create a set. Add some elements. Check for membership, etc.

- Use \texttt{ozmake} to remove the compiled file \texttt{SetAbs.ozf} from your project directory. This can be achieved with the command:

\begin{verbatim}
ozmake --clean
\end{verbatim}

Now write a makefile in file \texttt{makefile.oz} to describe this library. The library \texttt{SetAbs.ozf} should be installed at the abstract URI \texttt{x-ozlib://oz-kurs/exercises/SetAbs.ozf} (think what is the value of the makefile’s \texttt{uri} feature needed in order to achieve this). The package offering this library should be identified by \texttt{mogul://oz-kurs/exercises/modprog} (note that \texttt{mogul:} here is part of this abstract identifier, just like \texttt{x-ozlib:} above is part of the librarian’s abstract URI). Invoke \texttt{ozmake} in order to build and install your package. The command for this is:

\begin{verbatim}
ozmake --install
\end{verbatim}

You should now list the contents of your database of installed packages as follows:

\begin{verbatim}
ozmake --list
\end{verbatim}

Now test that this worked by obtaining the \texttt{SetAbs} module in the OPI from its abstract URI:

\begin{verbatim}
declare [MySetModule] = {Module.link [‘x-ozlib://oz-kurs/exercises/SetAbs.ozf]}
\end{verbatim}

Create a set. Add some elements, etc.

- Rewrite the finite automaton code of Section 6.2 into a functor program \texttt{FiniteAutomaton.oz} which exports on feature \texttt{accept} one function:

\begin{verbatim}
fun {Accept Automaton Word} ... end
\end{verbatim}

This function takes an \texttt{Automaton} as 1st argument and a \texttt{Word} as 2nd argument and returns true iff the \texttt{Word} is accepted by the \texttt{Automaton}. Your functor should import the library \texttt{x-ozlib://oz-kurs/exercises/SetAbs.ozf} that you installed in the previous exercise and use this abstraction in its code. Extend \texttt{makefile.oz} to state that \texttt{FiniteAutomaton.ozf} is also part of your package. Compile and install your package using the command:
ozmake --upgrade

what happens if instead you invoke ozmake -install and why? Test that your library works by getting your new FiniteAutomaton module with Module.link in the OPI. What is its abstract URI? Use it in the argument to Module.link. Test FiniteAutomaton.accept with some examples.

• Create a package file modprog.pkg for this little project. This is achieved with command:

  ozmake --create --package=modprog.pkg

Verify that this worked by installing the package you just created:

  ozmake --upgrade --package=modprog.pkg

Why do we use option -upgrade rather than -install? Now uninstall the package:

  ozmake --uninstall --package=mogul:/oz-kurs/exercises/modprog

Why is the package argument here mogul:/oz-kurs/exercises/modprog rather than modprog.pkg? Verify that the library provided by the package is no longer available by attempting to obtain and use it in the OPI. Install it again from modprog.pkg and very that it is again available and works.

8.9 Exercises

• Implement your own Map function by using a for loop with a collect feature.

• Write a function Add1ToInts adds 1 to all numbers in a list of numbers. The challenge is to write your function such that it never blocks even if some of the numbers in the list are still unknown (i.e. a free logic variable). For instance:

  {Inspect {Add1ToInts [1 _ 3 _ 4]}}

should return [2 _ 4 _ 5] in the inspector.

• (Extrapunkt) Reduce an until loop to a for loop.

• Write a function that tests whether a record contains a node which is labeled with the atom got_it. Use an exception to quit the recursion once successful.

• The following functor automaton.oz exports a function Reach that should compute the set of states that can be reached by a given finite automaton when reading a given word. But there is a bug in the functors definition. Find the bug and correct it.

  – Install the packages Abstract.ozf,
  – safe the functor automaton.oz in a file of the same name and write an Oz-Makefile to compile it with the debugger option -g,
  – debug the test program for automaton.oz for the functor below, and correct the bug once you found it.
Here comes the functor `automaton.oz`.

```
functor import
    Abstract at 'x-ozlib://oz-kurs/Abstract.ozf'
export
    Reach
define

    %% ReachByLetter computes the set of states that the
    %% Automaton can reach from States via a single Letter
    fun{ReachByLetter Automaton States Letter}
        Reached = {Abstract.newBag}
        for State in States do
            for NextState in Automaton.trans.State.Letter do
                Reached.put NextState
            end
        end
        States = {Reached.toList}
in
    end

    %% ReachByWord computes the set of states that the
    %% Automaton can reach from States via a given Word
    fun{ReachByWord Automaton States Word}
        case Word
            of nil then States
                in
                    [] Letter|RestWord then
                        NextStates = {ReachByLetter Automaton States Letter}
in
                        {ReachByWord Automaton NextStates RestWord}
                    end
            end
        end

    %% Reach computes the set of states that the
    %% Automaton can reach via a given Word
    fun{Reach Automaton Word}
        {ReachByWord Automaton Automaton.init Word}
    end
end
```

You can use the following program to test the above functor from the OPI (from where it is easy to run the debugger).

```
declare [Auto] = {Module.link ['automaton.ozf']}
declare Automaton = unit(trans:unit(p:unit(1:[q] 2:nil)
    q:unit(1:[p] 2:[q r])
    r:unit(1:nil 2:nil))
    init:[p] 0x28
```
A transducer is a finite automaton where transitions additionally produce output. For the purpose of this exercise, we suppose that each transition from state \( S \) to state \( S' \) has 1 input symbol \( I \) and 1 output symbol \( O \):

\[
\begin{align*}
&I \\
S &\quad \rightarrow \quad S' \\
&O
\end{align*}
\]

- You must first extend the representation of an automaton to allow the specification of an output symbol for each transition.
- With a transducer, we don’t simply reach a state: we reach it together with an output produced so far. For this reason, a word is not simply accepted or rejected: when it is accepted, it is accepted with a corresponding output. We call the pair of a state that is reached together with the output produced so far a *token*. You must modify the code for `Reach` provided in the previous exercise to collect tokens.
- Then you must abstract over what gets collected so that your code can serve either to collect the reachable states or the reachable tokens.
- You should then package your `Reach` function into a library and use it to instantiate a function that collects reachable states and one that collects reachable tokens.
9.1 Chart Parsing

9.1.1 What's Wrong with the Naive Parser?

The parser we developed in Chapter 5 was dumb because it had no memory: it did not recall which constituents it had already built and often did reparse a constituent it had parsed at some previous processing step. When a big grammar is used, this re-parsing of constituents can become very costly.

This problem is neither specific to our recogniser, nor to the specific (bottom-up) parsing strategy it uses. It also affects top-down parsers. As a simple example suppose the string to be analysed is the verb phrase "saw the man talk with the actress" and the grammar contains the following rules:

\[
\begin{align*}
\text{VP} & \rightarrow \text{V} \ \text{NP} \\
\text{VP} & \rightarrow \text{V} \ \text{NP} \ \text{PP} \\
\text{VP} & \rightarrow \text{V} \ \text{NP} \ \text{VP}
\end{align*}
\]

And suppose that we use a top-down parser. Then assuming the rules are treated sequentially and in the order given above, a chartless parser applies the first rule thereby parsing the verb "saw" and the NP "the man"; as the first rule fails, the second rule is tried out: again the verb "saw" and the NP "the man" are parsed. Finally the third rule is tried out and the parser succeeds. In total, both the verb "saw" and the NP "the man" are parsed three times which obviously is a waste of precious time.

To remedy this shortcoming, chart parsers use a table or "chart" in which parsed constituents are stored. The parsing strategy then ensures that no constituent is added to the chart which is already in it.

9.1.2 Chart Parsing as Inferential Closure

In chapter Chapter 5, we described bottom-up parsing in terms of an inference process. Here, we are going to improve on this description to arrive at chart parsing. One notion that we introduce is that of span: we write \text{SPAN} \ C_{i,j} \] to state that we have...
recognized category \( C \) and that it corresponds to the subsequence of the input starting at position \( i \) (included) and ending at position \( j \) (excluded).

The second important notion is that of a partially recognized category (or equivalently a partially applied rule). We write \( \text{ITEM } C [i, j] C_1 \ldots C_n \) to state that we have partially recognized \( C \): the portion that has been recognized spans \([i, j]\) and what remains to be recognized is the sequence \( C_1 \ldots C_n \).

Finally, as before, we write \( \text{RULE } C \rightarrow C_1 \ldots C_n \) to state that the grammar contains a rule \( C \rightarrow C_1 \ldots C_n \). We also write \( \text{INPUT } C_1 \ldots C_n \) to state that the input consists of the sequence \( C_1 \ldots C_n \).

Let’s consider first the rules that initialize the inference process. If \( C_i \) is in the input then it spans itself:

\[
\text{INPUT } C_1 \ldots C_n \\
\text{-------------} \\
\text{SPAN } C_i[i, i+1]
\]

In the following, we will write \( N \) for the size of the input. At every position \( i \), we have partially recognized every rule with an empty span \([i, i]\):

\[
\text{RULE } C \rightarrow C_1 \ldots C_k \\
\text{----------------------} \\
\text{ITEM } C [i, i] C_1 \ldots C_k
\]

Now the rule which extends a partial recognition:

\[
\text{ITEM } C [i, j] C_1 C_2 \ldots C_p \\
\text{SPAN } C_1 [j, k] \\
\text{-----------------------------} \\
\text{ITEM } C [i, k] C_2 \ldots C_p
\]

and the rule which notices a fully recognized category:

\[
\text{ITEM } C [i, j] \\
\text{-------------} \\
\text{SPAN } C [i, j]
\]

and finally a rule which notices that a category covers the entire input. We write \( \text{ACCEPT } C \) to state that the entire input has been recognized as category \( C \).

\[
\text{SPAN } C [1, N] \\
\text{-------------} \\
\text{ACCEPT } C
\]

Thus the question of whether \( C_1 \ldots C_n \) is in the language generated by the grammar can be replaced by the question of whether, for some \( C, \text{ACCEPT } C \) is in the inferential closure of \( \text{INPUT } C_1 \ldots C_n \).
9.1.3 What's a chart? What is it good for?

What's a chart? A chart is a directed graph consisting of (numbered) nodes and (labelled) edges. Each edge represents a constituent: the label of an edge indicates the category of that constituent whereas the number of the initial and final nodes give the position of that constituent in the input string. For instance, an edge labelled NP and stretching from node 1 to node 5 represents an NP stretching from position 1 to position 5 in the input string.

A chart represents the set of all parse trees for a sentence. Whereas a parse tree represents a structure that can be assigned to a sentence by the underlying grammar, the chart stores all constituents which may or may not be part of the a parse tree.

9.1.4 Chart Parsing

How is a chart used by a parser? This is well-known in computational linguistics but also in computer science, where chart parsing is usually called the CYK-algorithm.

The idea is as follows: We start from the chart which edges for all words. We then apply the rules of the grammar to subsequences of adjacent edges. For all binary rule, say $C \rightarrow C_1 C_2$, and all adjacent edges of categories $C_1$ and $C_2$, we add a combined edge of category $C$.

An example for "a man runs" to be written

To problem now is that we have to determine when chart is complete. This means that no further edges can be added to the chart. A simple way to check for the completeness of a chart si provide an additional data structure which contains all those edges, which have to be considered for further rule applications. We call this data structure an agenda which could be realized either as a stack or as a heap.

We call an agenda-chart pair complete if it satisfies the following property:

If two edges in a chart can be combined then the resulting edge belongs either to the chart or to the agenda.

If an agenda-chart pair is complete and the agenda empty, then the chart is complete in the previous sense. Chart parsing now looks as follows: Pick and delete an edge from the agenda, add all combinations of this edge to the agenda, and the edge itself to the chart.

This algorithm terminates since only a quadratic number of edges can be added to a chart. Adding an edge costs linear time such that the all over costs for chart parsing are cubic time.

9.1.5 Looking to the Right is Enough

So far, edges from the agenda have to be combined to the left and to the right with edges from the chart. But if we impose an ordering on adding edges then it is enough to look to one side only. Suppose we want to look to the right only. Then we have to built the chart from the right to the left. This means that an edge at position $P$ is added only if the chart is complete between position $P + 1$ and the right most position. Such an ordering can be obtained by organising the agenda as a stack.
An example has to be added here.

At the beginning, the edges for all words such that the right most word is on top of the stack.

Then each edge in the agenda is processed as follows. Let edge(Cat,Begin,End) be the edge to be processed, then:

- For each unary rule of the form $\text{Cat'} \rightarrow \text{Cat}$, add the edge edge(Cat',Begin,End) to the agenda.
- For each edge in the chart of the form edge(RightCat,End,NewEnd) and for each rule of the form $\text{Cat'} \rightarrow \text{Cat RightCat}$, add the edge edge(Cat',Begin,NewEnd) to the agenda.

What this does is process the string from right-to-left, processing one word at a time and recursively adding all possible right-hand expansions of this word to the chart. In this way, all possible constituents are parsed and each constituent is only parsed once.

### 9.1.6 Exercise

- Proof by induction that the above stack-based algorithm terminates with a complete chart.

### 9.2 Implementing a bottom-up chart parser

The main difference between the parser we now present and the one we saw in the previous lecture is the use of a chart. The first question we therefore need to address is the question of how to represent this chart.

#### 9.2.1 Charts as arrays

Recall that the chart is a graph whose nodes are numbered. Generally, we need to be able to refer to positions in the chart: we want to add edges to the chart between two given position or we might want to search the chart for all edges starting/ending at a given position. One simple way to model such a structure is to use an array. How exactly do we represent a chart using an array? First, we set the length of the array to the number of words contained in the input string. Initially, each position in the array is bound to the empty list but as the chart is built, they become bound to lists of edges representation. Specifically, an edge stretching from Begin to End and labelled with category Cat is represented at position Begin in the array by the record

```plaintext
edge(cat:Cat begin:Begin 'end':End)
```

```plaintext
edge.oz:116a
functor
import
export

%% type cat = atom
%% type edge = unit(cat:cat begin:int 'end':int)
make:MakeEdge % cat x int x int -> edge
```
define
fun {MakeEdge C B E}
  edge (cat:C begin:B end:E)
end
end

Next we need to be able (i) to add edges to the chart and (ii) to search the chart for edges ending/starting at a given position. To do this, we use the predefined procedures Put and Get.

```
117a %chart.oz [117b]
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% NewChart creates a new chart of the following type:
%% type edge = 'a
%% type chart(a) = unit(add:edge ->
%% get:int -> list(edge)
%% min:int
%% max:int
%% words:list(atom)
%% toList: -> list(edge))
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
functor
export
  new:NewChart
define

  fun {NewChart Sentence}
    MinPos = 0
    MaxPos = {Length Sentence}
    Chart = {Array.new MinPos MaxPos nil}
  fun {GetChart Pos}
    {Array.get Chart Pos}
  end
  proc {Add Edge}
    {Array.put Chart Edge.begin Edge}{Array.get Chart Edge.begin}}
  end
  fun {ToList}
    for
    Pos in MinPos..MaxPos
    collect:Collect
    do
      for Edge in {GetChart Pos} do
        {Collect Edge}
      end
    end
  end
in
  unit {add:Add % edge ->
```
The function `NewChart` binds the variable `Chart` to an array with key range `MinPosition` (1) to `MaxPosition` (the number of words in the input string plus one) with all items initialised to the empty list `nil`. It returns a record with two features add and get whose values are operations on `Chart`. GetC simply retrieves the content of `Chart` (an array) at a given position. Add takes an edge as input (i.e. a record of the form `cat:C begin:B 'end':E) and modify the array `Chart` as follows: at position `End` in `Chart` it adds to the current list value `{Get Chart End}` the element `left(Cat Begin)` and at position `Begin` in `Chart` it adds to the current list value `{Get Chart Begin}` the element `right(Cat End)`.

### 9.2.2 Parsing with a chart

We now define the top level loop of the parsing function as follows:

```oz
functor
import
C(new:NewChart) at 'chart.ozf'
E(make:MakeEdge) at 'edge.ozf'
A(newStack:NewStack) at 'x-ozlib://oz-kurs/Abstract.ozf'
export
new:NewParser %% grammar x lexicon -> parser

%% type word = atom
%% type cat = atom
%% type rule = unit(left:cat right:list(cat))
%% type grammar = list(rules)
%% type lexicon = record(word:cat)
%% type parser = list(word) -> chart(edge)

define
fun{NewParser Rules Lexicon}
fun{Parser Words}
  Chart = {NewChart Words}
  Agenda = {NewStack}
```
We now an example for testing. We have to define a grammar, a lexicon, and sentences for testing.

**Lexicon:***

```oz
functor
import
export
toCat:WordToCat
define
  LexList = [john#pn runs#v likes#v sees#v the#det man#n 'with'#prep
telescope#n]
  LexRec = (List.toRecord unit LexList)
fun{WordToCat Word}
  LexRec.Word
end
define
```

**Grammar:***

```oz
functor
import
export

%% type cat = atom
Rules %% unit(left:cat right:list(cat))
define
  fun{MakeRule L R} unit(left:L right:R) end
  fun{R C C1} {MakeRule C [C1]} end
  fun{R2 C C1 C2} {MakeRule C [C1 C2]} end

Rules = [{R2 s np vp}
  {R2 np det n}
  {R2 np np pp}
  {R2 vp v np}
  {R2 vp vp pp}]
```
Chapter 9. A Chart Parser for Context Free Grammars

\{R2 \text{ pp prep np}\}
\{R1 \text{ np pn}\}
\{R1 \text{ vp v}\}

\text{end}

\text{Parse} \ takes \ a \ list \ of \ words \ as \ input, \ initialises \ an \ agenda \ and \ a \ chart, \ adds \ lexical \ edges \ to \ the \ agenda (\text{Words to Agenda}), \ process \ the \ agenda \ and \ when \ finished, \ displays \ the \ content \ of \ the \ resulting \ chart \ at \ position \ 1 \ that \ is, \ the \ list \ of \ edges \ starting \ in \ position \ 1.

\subsection{9.2.3 Initialising the agenda}

Adding lexical edges to the agenda is defined as follows.

\begin{verbatim}
120a (Words to Agenda) ≡
  %% create an edge for all words in the input sentence
  %% and add it to the agenda.
  for %% iterate in parallel
    Word in Words
    Pos in Chart.min..Chart.max
    do
      Edge = (MakeEdge {Lexicon.toCat Word} Pos Pos+1)
      in
        {Agenda.push Edge}
    end

Words to Agenda recursively walks down the list (Phons) of input words, and for each word encountered, it add a lexical edge of the form Cat(Begin End) to the agenda where Cat is the category assigned by the lexicon to that word and Begin and End gives the position of that word in the input string.
\end{verbatim}

\subsection{9.2.4 Processing the agenda}

Next, the processing of the agenda is defined: if the agenda is empty, nothing is done (\text{skip}). Else the first element in the agenda-stack is retrieved, processed and then added to the chart. When this is done, the recursive call to WorkOnAgenda ensures that the rest of the agenda is processed.

\begin{verbatim}
120b (Process Edges) ≡
  %% process the agenda
  for break:Break do
    if {Agenda.isEmpty}
      (Break)
    else
      Edge = {Agenda.pop}
      in
        {Process Edge}
        {Chart.add Edge}
    end
  end
\end{verbatim}
9.2.5 Processing an edge

Finally, the main processing step, the processing of an edge taken out of the agenda. As discussed in the introduction, processing an edge consists in producing all edges resulting from:

- The successful application of some unary rule to the category labelling this edge,
- The successful application of some binary rule to the category labelling this edge followed by the category of some right adjacent edge.
- The recursive combination of the edges resulting from the previous step with their right-adjacent constituents.

This is what the procedure `ProcessEdge` does. It proceeds as follows. First, all edges resulting from the application of a unary rule to `Edge` are added to the agenda `ProcessUnary`. Second, right-adjacent edges are collected into a list and binary rules are tried out: each right-adjacent edge is tried for reduction with the edge being processed and in case of success, the resulting edge is added to the agenda.

```
121a (procedure Process [121a]a)≡
  proc {Match Edge Rule Begin}
  Cat = Edge.cat
  in
  case Rule.right
  of nil then
   raise error('right hand sides of rules must be nonempty') end
  [] ![Cat] then
   {Agenda.push {MakeEdge Rule.left Begin Edge.'end'}}
  [] ![Cat|NextCats then
   NextRule = unit(left:Rule.left right:NextCats)
   in
     for NextEdge in {Chart.get Edge.'end'} do
       {Match NextEdge NextRule Begin}
     end
   else skip % rule does not apply
  end
end
```

```
%% Process adds all Edges to the Agenda
%% that start with the edge `Edge`
proc {Process Edge}
  for Rule in Rules do
    {Match Edge Rule Edge.begin}
  end
end
```
9.2.6 A Test Interface

Here is a small interface: it binds a variable (Chart) to the result of parsing a string i.e. the final chart. The function Result then takes this chart as input, filters from the list of edges listed in the minimal position of the chart those edges which end in the maximal position and returns their category.

```
declare Modules = ['parser.ozf' 'sentences.ozf' 'grammar.ozf' 'lexicon.ozf' 'ChartToWindow.ozf']
decide [P S G Lexicon Ch2Win] = \{Link Modules\}
decclare Sentences = S.testSuite
decclare MakeParser = P.new
decclare Grammar = G.rules
decclare fun(Result Chart)
  \{Map \{Filter \{Chart.get Chart.min\}
    \{fun($ E) E.'end' == Chart.max end\}
    \{fun($ E) E.cat == s\}\end\}
end
decclare Parser = \{MakeParser Grammar Lexicon\}
decclare C2W = Ch2Win.chartToWindow

%% Test
for Sentence in Sentences do
  \{Inspect Sentence\}
  Chart = \{Parser Sentence\}
  Edges = \{Chart.toList\}
  Res = \{Filter Edges \{fun($ E)
    \{And \{And
      E.begin==Chart.min
      E.'end'==Chart.max\}
      E.cat==s\}\end\} end\}
end
  \{Inspect 'number of parses:'\#{Length Res}\}
end
```

9.2.7 The Complete Package

`oz-kurs-chart.pkg` is a package containing the entire code for the passive chart parsing library. You can install this package as follows:

```
ozmake --install --package=oz-kurs-chart.pkg
```

`'oz-kurs-chart.pkg`
All functors in this library are made available at URI `x-ozlib://oz-kurs/chart`. For the exercise, you will also need the sources which you can extract from the package file as follows:

```
ozmake --extract --package=oz-kurs-chart.pkg
```

For your browsing convenience, here is the source of the parser:

```ozf
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% a passive chart parser for context free grammars
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
functor
import
    C(new:NewChart) at 'chart.ozf'
    E(make:MakeEdge) at 'edge.ozf'
    A(newStack:NewStack) at 'x-ozlib://oz-kurs/Abstract.ozf'
export
    new:NewParser %% grammar x lexicon -> parser
define
    fun(NewParser Rules Lexicon)
    fun(Parser Words)
        Chart = {NewChart Words}
        Agenda = {NewStack}
        proc(Match Edge Rule Begin)
            Cat = Edge.cat
            in
                case Rule.right
                of nil then
                    raise error('right hand sides of rules must be nonempty') end
                [] ![Cat] then
                    (Agenda.push {MakeEdge Rule.left Begin Edge.'end'})
                [] !Cat|NextCats then
```

NextRule = unit(left:Rule.left right:NextCats)
in
for NextEdge in [Chart.get Edge.'end'] do
    {Match NextEdge NextRule Begin}
end
else skip  %% rule does not apply
end

%% Process addes all Edges to the Agenda
%% that start with the edge Edge
proc{Process Edge}
    for Rule in Rules do
        {Match Edge Rule Edge.begin}
    end
end

%% create an edge for all words in the input sentence
%% and add it to the agenda.
for  %% iterate in parallel
    Word in Words
    Pos in Chart.min..Chart.max
    do
        Edge = {MakeEdge {Lexicon.toCat Word} Pos Pos+1}
in
        {Agenda.push Edge}
    end

%% process the agenda
for break:Break do
    if {Agenda.isEmpty} then {Break}
    else
        Edge = {Agenda.pop}
in
        {Process Edge}
        {Chart.add Edge}
    end
end
in
Chart
in
Parser
end
end
9.2.8 Exercises

- Modify the chart parser such that all its edges are named and have reference to the names of their childs. For instance, the chart for the sentence \textit{john runs} should contain edges of the following form:

  \begin{verbatim}
  (Edges: 125a)
  [edge(begin:1 cat:s 'end':3 id:17 ids:[16 12])
   edge(begin:1 cat:s 'end':8 id:18 ids:[16 13])
   edge(begin:1 cat:s 'end':8 id:19 ids:[16 15])
   edge(begin:1 cat:s 'end':5 id:20 ids:[16 14])
   edge(begin:1 cat:np 'end':2 id:16 ids:[1])
   edge(begin:1 cat:pn 'end':2 id:1 ids:nil)
   edge(begin:2 cat:vp 'end':3 id:12 ids:[2])
   edge(begin:2 cat:vp 'end':8 id:13 ids:[2 11])
   edge(begin:2 cat:vp 'end':8 id:15 ids:[14 9])
   edge(begin:2 cat:vp 'end':5 id:14 ids:[2 10])
   edge(begin:2 cat:v 'end':3 id:2 ids:nil)
   edge(begin:3 cat:np 'end':8 id:11 ids:[10 9])
   edge(begin:3 cat:np 'end':5 id:10 ids:[3 4])
   edge(begin:3 cat:det 'end':4 id:3 ids:nil)
   edge(begin:4 cat:n 'end':5 id:4 ids:nil)
   edge(begin:5 cat:pp 'end':8 id:9 ids:[5 8])
   edge(begin:5 cat:prep 'end':6 id:5 ids:nil)
   edge(begin:6 cat:np 'end':8 id:8 ids:[6 7])
   edge(begin:6 cat:det 'end':7 id:6 ids:nil)
   edge(begin:7 cat:n 'end':8 id:7 ids:nil)]
  \end{verbatim}

  The name (identity) of an edge can be an arbitrary Oz value, for instance the integers 10 11 12 13 14 as above.

- Write a functional procedure \texttt{ChartToParseTrees} that computes the set of parse trees from a chart with named edges. If you have not solved the previous exercise then simply compute the parse trees constructable from the above list of edges for the sentence \texttt{[john runs]}.

- Define a stack that inspects everything that is pushed onto it. Hint: Use the function \texttt{Record.adjoin}.

- Write a chart parser with window output in Oz. Use the chart parser functor of the previous exercise and the functor at the URL:

  \url{http://www.ps.uni-sb.de/~niehren/oz-course.html/Functors/Version.3.2/ChartToWindow.ozf}

- What is the meaning of the symbol ! in pattern matching statements, as used in \texttt{procedure Process}?
Part III

Unification and Parsing
Unification

10.1 What is Unification?

The idea of unification is to describe values by logical equations which can be resolved automatically by some unification algorithm. First-order unification is a form of unification whose values are trees. There are several kinds of first-order unification which differ in the choice of the notion of trees. Trees may be ground terms (tuples) or feature trees (records), they may be finite or infinite. First-order unification for feature trees is called feature unification.

Oz provides first-order unification for tuples and records. Feature unification is a prerequisite for building a parser for a unification grammar. In order to write unification-based chart parser in a high-level manner, something beyond feature unification is needed. Is is essential to treated solved forms of equations as first-class values which then can be associated with the edge of a chart. Indeed, Oz supports constraints as first-class values by so called encapsulated search. Oz is unique in this respect.

10.2 Finite and Infinite Trees

A finite tree is an atom or a tuple of finite trees. For instance \( \text{f(f(a b) g(a f(c)))} \) is a finite tree. A finite tree is uniquely determined by its tree domain and a labeling function. The tree domain of a tree is the set of path leading from its root to its nodes. The labeling function maps a path to the label at the node reachable by this path. In the above example, the tree domain \( D=\{\text{epsilon,1,11,12,2,21,22,221}\} \); the labeling function \( L \) is given by: \( L(\text{epsilon})=\text{f}, L(1)=\text{f}, L(11)=\text{a}, L(12)=\text{b}, L(2)=\text{g}, L(21)=\text{a}, L(22)=\text{f}, \) and \( L(221)=\text{c} \).

In other words, an infinite tree can be defined by an infinite tree domain and a labeling function. For instance, \( \text{f(f(f(\ldots)))} \) is an infinite tree whose tree domain \( D \) is the set \( \{\text{epsilon,1,11,111,\ldots}\} \). Infinite trees can be considered as a unwinding of rooted graphs. Infinite trees matter for programming since they allow to model cyclic data structures for constraint programming.

10.3 Equations

The variable in Oz are logic variable. A logic variable can be understood as placeholder for a value which can be filled in when needed. At run time, a logic variable can either be free, kinded, or determined, depending on how much is known about its value.
Logic variables can be used for writing equations. Solving equations possibly determines their values. Solving equations interpreted over trees can be done by a standard unification algorithm. In Oz, unification for infinite trees is built in.

Suppose for instance, that you want to unify the terms \( f(X, X) \) and \( f(g(Y, Z), Y) \) where \( X, Y, Z \) are logic variables denoting some possibly infinite tree. In order to do so, it is sufficient to solve the equation \( f(X, X) = f(g(Y, Z), Y) \) which can be done simply by feeding it into the Oz-emulator.

\[
\text{declare} \\
X \ Y \ Z \\
\text{in} \\
f(X, X) = f(g(Y, Z), Y) \\
\text{Browse \{X Y Z\}}
\]

In the Browser, you can observe the result of the unification process. The variable \( Z \) is still free; the variables \( X \) and \( Y \) are bound to a term \( g(g(... Z) Z) Z \) which can be solved by an infinite tree depending on the value of \( Z \). Note that the equation \( X.2 = X.1.2 \) is valid independently of the choice of \( Z \).

For observing variable assignment into infinite trees in Oz, the Browser provides two extra options for representation, either as graphs or as minimal graphs. The difference between the graph and minimal graph option can be observed at the following example:

\[
\text{declare} \\
X \ Y \ Z \\
\text{in} \\
X=f(Y, X) \\
Y=f(Y, X) \\
\text{Browse X}
\]

Note that you can rebrowse a term by selecting the field Rebrowse from the Browser’s menu Select.

Unification in Oz terminates even though equations interpreted over infinite trees are solved. The reason is that a solved form of the equations which cycles can be stored in the Oz constraint store. However, the programmer has to care about termination, in particular, when applying a recursive procedure to an infinite tree. For instance, the following program does not terminate:

\[
\text{declare} \\
L=\text{infty} | L \\
\text{Browse \{Length L\}}
\]

Browsing infinite trees terminates in Oz. When choosing the tree option for representation then there is a fixed bound which limits the depth up to which an (infinite) tree is displayed.
10.4 Efficiency

First-order unification for infinite trees can be implemented in quasi-linear time. The implementation in Oz is quadratic in worst case and linear for mostly all important cases.

A classical example where Robinson’s original unification algorithm has exponential runtime is the following:

131a \( \text{UnifBinTree} \) ≡

\[
\text{proc\{UnifBinTree X N\}}
\quad \text{if} \quad N==0 \quad \text{then} \quad X=a
\quad \text{else}
\quad \text{Y}
\quad \text{in}
\quad X = f(Y Y)
\quad \text{(UnifBinTree Y N=1)}
\quad \text{end}
\quad \text{end}
\]

/*
declare X = {UnifBinTree $ 1000000}{Browse done}
*/

If you browse \( X \) then you have to wait for a while since \( X \) is a bound to the complete infinite binary tree. Even when the display depth is restricted to 15 (as by default in Oz) then the Browser output has size \( O(2^{15}) \).

Next we wish to measure the time that unification needs.

131b \( \text{Property time} \) ≡

/*
declare \{Browse \{Property.get time\}\}
*/

There are many time measures explained in the documentation. We now write a procedure that measures the run time of a procedure \( P \) without arguments.

131c \( \text{Time} \) ≡

\[
\text{fun\{Time P\}}
\quad T1 = \{Property.get time\}.user
\quad \{P\}
\quad T2 = \{Property.get time\}.user
\quad \text{in}
\quad T2-T1
\quad \text{end}
\]

Next, we measure the time for building a representation of the complete binary tree of depth \( N \).

131d \( \text{TimeUnif} \) ≡
fun(TimeUnif N)
    {Time proc($) {UnifBinTree _ N} end}
end
/

{Browse {TimeUnif 10000}}
{Browse {TimeUnif 100000}}
{Browse {TimeUnif 200000}}
{Browse {TimeUnif 400000}}
*/

For administrative reasons, we collect all the chunks presented so far into a chunk.

132a (Test TimeUnif 132a) =
    declare
    (UnifBinTree 132a)
    (Property time 132a)
    (Time 132a)
    (TimeUnif 132a)

10.5 Feature Constraints

A feature tree is a ground record whose sub-values are also records. Oz provides feature constraints, i.e partial (underspecified) descriptions of feature trees. Feature unification is the process of solving feature constraints.

For instance, you can tell the following feature constraint to the Oz constraint store:

    declare X AVM1 in
    AVM1^dtrs^2^1 = unit(cat:vp nb:sg)
    {Browse AVM1}

The feature selector “hat” ^ allows to describe a fields of a feature tree without specifying the set of all features of this tree a priori. The hat selector provides for open feature (tree) descriptions. Apart from this, the two feature selectors “hat” ^ and “dot” . do not differ.

Next you can add the following additional constraint for fixing the arity of AVM1:

    AVM1 = unit(cat:s
        nb:X
        dtrs:[unit(cat:vp
            nb:X)
            unit(cat:vp
            nb:X)]

Note also, how the coreferences due to the variable X are treated correctly by unification.
10.6 Feature Structures versus Feature Trees

A feature structure is a rooted graph, which is built similar to a feature tree. Every feature tree is a feature structure. Every feature structure can be unfolded into a feature tree. When doing so, node equality is lost but structural equality is preserved.

Feature constraints can be given an alternative semantics. Rather than interpreting a variable as a feature trees, one may interpreted a variable as a node in a feature structure. This alternative interpretation could be used without affecting the notions of solvability, solved forms, or unification.

In other words, for a feature structure only the information it represents matters but how structural equality is expressed: either by node equality or by the structural equality of trees. Node equality is an implementation detail which can be used to express structural equality. Only structural equality matters. We built our logics on the notion of feature trees rather than on feature structures in order to not use node equality (called “structure sharing” in Pollard/Sag) as a basic metaphor in our explanations.

10.7 Exercises

- Consider the following representation of terms:

  \[
  \text{term ::= var | f(term ... term)}
  \]

  where a variable \( \text{var} \) is represented by an integer. Write a function \( \text{Decompose:term -> var* [basic]} \) that decomposes a term into a pair of a variable denoting the root of the term and a list of basic constraints where the latter have the following abstract syntax:

  \[
  \text{basic ::= eq(var var) | eq(var f(var ... var))}
  \]

  for example \( \text{f(1 g(a 1 2))} \) should decompose into:

  \[
  3\#[\text{eq}(3 \text{ f(1 4)}) \text{ eq}(4 \text{ g(5 1 2)}) \text{ eq}(5 \text{ a})]
  \]

- Write a function \( \text{Substitute:var*[basic] -> term} \) that takes a pair of a variable and a list of basic constraints and returns the corresponding term representation. Apply the function to:

  \[
  1\#[\text{eq}(1 \text{ f(2 2)}) \\
  \text{ eq}(2 \text{ f(3 3)}) \\
  \text{ eq}(3 \text{ f(4 4)}) \\
  \text{ eq}(4 \text{ f(5 5)}) \\
  \text{ eq}(5 \text{ f(6 6)}) \\
  \text{ eq}(6 \text{ f(7 7)}) \\
  \text{ eq}(7 \text{ f(8 8)}) \\
  \text{ eq}(8 \text{ f(9 9)})]
  \]

  Then to
Chapter 10. Unification

1. \[
\begin{align*}
1 & \equiv (1 \ f(2 \ 2 \ 2 \ 2)) \\
2 & \equiv (2 \ f(3 \ 3 \ 3 \ 3)) \\
3 & \equiv (3 \ f(4 \ 4 \ 4 \ 4)) \\
4 & \equiv (4 \ f(5 \ 5 \ 5 \ 5)) \\
5 & \equiv (5 \ f(6 \ 6 \ 6 \ 6)) \\
6 & \equiv (6 \ f(7 \ 7 \ 7 \ 7)) \\
7 & \equiv (7 \ f(8 \ 8 \ 8 \ 8)) \\
8 & \equiv (8 \ f(9 \ 9 \ 9 \ 9)) \\
\end{align*}
\]

If it gets very slow, can you identify the cause of the inefficiency and modify your implementation to avoid it?

- Write a function `Unify:term->bool` that tests whether two terms are unifiable. Suppose that terms are represented as above. Hint: Replace integers by Oz variables in a first step and use exception handling to catch failure.

- Feature unification: Give a closure algorithm that unifies record descriptions `rec ::= f(f_1:rec ... f_n:rec) | var` where all atoms `f_1 ... f_n` are pairwise distinct (on paper). Hint: adapt the unification algorithm presented in the lecture and reproduced below.

- Write a function `Unify:rec->bool` that tests whether two records are unifiable. Suppose again that variables are represented by integers.

- Zusatzaufgabe: A solved form is a set of equations `var_1=term_1 ... var_n=term_n` where all variables `var_1 ... var_n` are pairwise distinct. Test whether there exists a finite solution `sol:var->term` that satisfies all equations. Hint: there exists a finite solution if and only if the solved form is acyclic (when considered as a graph whose nodes are the variables).

- Zusatzaufgabe: implement unification using the algorithm of the lecture.

### Unification Algorithm

Here are the rules for computing the solved form of a conjunction of basic constraints. In the following, `S` and `B` are conjunctions of basic constraints. The algorithm is presented as a rewriting algorithm: it starts with `true:B` and terminates with `S:true` or `S:false`. When it terminates with `S:true`, `S` is the solved form of the input basic constraint `B`. When it terminates with `S:false`, the original `B` is unsatisfiable.

We say that `X` is bound in `S` if `S` contains a basic constraint with `X` on the left-side of the equation. The algorithm maintains the following invariant for `S:B`: if `S` contains an equation `X=Y`, then `X` does not occur in `B`, nor in `S` except in equation `X=Y`. We write `B[X/Y]` for the resulting of replacing every occurrence of `X` in `B` by `Y`.

\[
\begin{align*}
S : true & \land B & \rightarrow & S : B \\
S : X=X & \land B & \rightarrow & S : B \\
S : X=Y & \land B & \quad \text{where } X \text{ and } Y \text{ are distinct identifiers} \\
\quad & \text{if } X \text{ not bound in } S \\
& \rightarrow S[X/Y] & X=Y & : B[X/Y] \\
\quad & \text{else if } Y \text{ not bound in } S \\
& \rightarrow S[Y/X] & Y=X & : B[Y/X] \\
\end{align*}
\]
else $X = f(X_1...X_n)$ in $S$ and $Y = g(Y_1...Y_m)$ in $S$
  if $f = g$ or $n = m$
    $\rightarrow S : false$
  else
    $\rightarrow S[X/Y] & X = Y : (X_1 = Y_1 & ... & X_n = Y_n & B)[X/Y]$

$S : X = f(X_1...X_n) & B$
if $X = g(Z_1...Z_m)$ in $S$
  if $f = g$ or $n = m$
    $\rightarrow S : false$
  else
    $\rightarrow S : X_1 = Z_1 & ... X_n = Z_n & B$
else
  $\rightarrow S & X = f(X_1...X_n) : B$
Encapsulated Search

11.1 Unification as Constraint Solving

Unification is the process of solving constraints. The Oz constraint store contains only constraints in solved form. Whenever a constraint is told to the constraint store, unification becomes active and computes a new normal form. This new normal form is logically equivalent to the conjunction of the previous normal form and the newly added constraint.

Of course it can happen that a constraint told is inconsistent with the actual constraint store. In this case, unification fails and no constraint is added to the constraint store. Failure can be considered as logical information or as programming error. This will be explained later.

11.2 Failure versus Programming Error

Of course, it is possible to write down a set of equations without solutions. In this case unification fails. There are several interpretations for failure: Either it may be considered as logical information about unsolvability of equations or else as a programming error. In Oz, unsolvable equations should only occur using constructs from constraint programming. Otherwise, failure is considered as programming error. If you feed the following equations for instance, then failure is already recognized statically.

\[
\begin{align*}
X &= f(\_ \ a) \\
Y &= f(\_ \ g(b)) \\
X &= Y
\end{align*}
\]

The same kind of failure may also raise a programming error at run time:

```
declare
proc{Constrain X Y}
    X = f(\_ a)
    Y = f(\_ g(b))
end
{Constrain X X}
```

Only when used in search, failure is considered as logical information and not as programming error.
11.3 Encapsulated Search

FirstSol = {Search.one Pred}

to be written.

What is a search predicate?

11.3.1 Stability

to be written.

11.3.2 Global Variables

to be written.

11.3.3 Stateful Data Structures
Unification and Encapsulation

The encapsulation aspect of encapsulated search is also useful on its own right. Encapsulation can be used for encapsulating failure which may or may not occur by unification. This is important, since failure on top level is considered as a programming error in Oz.

12.1 Encapsulating Failure

For returning the result of the unification process, independently of failure or not, you can use the Oz search facilities as follows:

\[
\text{let X:} \{\text{Search.base.all proc} \{X\} \{\text{Constrain X _}\} \text{end}\}\]

This works fine for returning failure in terms of an empty list rather than a programming error. But there is still a problem with returning the result of unification when coreferences occur. How can you return the result in the following example?

\[
\text{let X:} \text{proc}\{\text{CoRef X}\} Y \text{in} X=f(Y Y) \text{end}
\]

\[
\text{let X:} \{\text{Search.base.all proc} \{X\} \{\text{CoRef X}\} \text{end}\}\]

Here you have lost the coreference between the first and second subtree of X. This problem can be solved in Oz as described in the next section.

12.2 How to treat a Constraint as a Value?

In order work with constraints, it would be nice if a constraint store could be a value accessible by a variable. In this way, one could, for instance report unification failure by binding a variable to a value, say the empty list for instance. One could also define all kinds of operations on constraints as discussed in the following examples.

For instance, one might consider the following existential constraints

\[
\text{local Y in } X=f(Y Y) \text{ end}
\]

which requires the value of X to be a tree with two identical subtrees at first and second position. Once this constraints is told to the constraint store, only the operations of the constraint store are applicable to it. For instance, one can ask the constraint store for the equality between both subtrees of X:
This conditional can behave in three possible manners depending on the state of the constraint store:

- Whenever the asked equality is logically implied by the constraint store, \textit{true} is browsed.
- If the equality is inconsistent with the constraint store, \textit{true} is browsed.
- If equation is not yet available but could be added later on the conditional suspends until more information is available.

The third case may be problematic if one would like to know about the actual state of the constraint store at the time point being. An alternative conditional which never blocks can be written in Oz. In general, however, it is not clear how to write operations for the constraint store which are not directly expressible by the constructs of the Oz-language.

A trivial way of how to represent constraints - which works in all programming languages - is to use ground terms (trees) for representing constraints and then to define all operations for them by hand. For representing the above constraints, we might write for instance:

\[
\text{f(1 1)}
\]

Here, the integer \(1\) is not intended to be a value itself but simply expresses the coreference between the two positions of the described term. When using ground terms for descriptions, however, it is impossible to use the operations of the constraint store like unification. For instance, the terms \(f(1 1)\) and \(f(3 \ g(2))\) should be unifiable since the information they represent can be combined. In order to do so, the user has to write its own unification procedure \texttt{Unif} which is able to deal with applications like:

\[
\text{(Unify f(1 1) f(3 g(2)))}
\]

Writing unification by hand is odd, in particular when using a programming language which provides unification.

### 12.3 Constraints as Predicates

Oz provides “constraints as values” in a way which can be paraphrased as “constraints as predicates”. In this section, we first give the general idea and then apply it: We show how to testing the unifiability of a set of terms with variables such that failure cannot raise a programming error.

For instance, the following predicate imposes a coreference constraint on its argument.

\[
\text{proc($X$) Y in X=f(Y Y) end}
\]
This predicate expresses the set of all trees $x$ with the property of having two equal subtrees at first and second position. When applying this procedure to an argument $z$ then the constraint $z = f(Y Y)$ is told to the Oz constraint store.

Predicates are values in Oz and can thus be bound to variables. Encapsulated search in Oz can resolve a predicate and return a list of its solutions. Each solution is a constraint which can be returned as a predicate again.

In the following, we use constraints as predicates in order to check whether a set of terms is unifiable. A term may contain variables but we assume that variables in distinct terms are distinct. A term $t$ with free variables $X_1, \ldots, X_n$ is encoded by a predicate

$$\text{proc}(\$ X) X_1 \ldots X_n \text{ in } X=t \text{ end}.$$ 

The procedure \texttt{Search.allP} inputs a unary predicate and returns a list of unary predicates each of which represents a solution to the input predicate. If a predicate encapsulates a constraint then a solution of it is a predicate which encapsulates a solved form of the constraint.

We can now run the tester for unifiability. Note that our test does not raise a programming error even if unification fails.

```
12.4 Unifying Attribute Value Matrices

We now show how to unify attribute value matrices (AVM’s) in Oz. The idea is to treat AVM’s exactly as we treated terms in the previous section. Here are three examples of AVM’s encoded as predicates.

142a (AVM’s as Predicates)

proc{AVM1 X} 
  Y in X=a(f:a(f:X g:Y h:Y))
end

proc{AVM2 X} 
  Y in X=a(f:a(f:X g:a(f:Y) h:Y))
end

proc{AVM3 X} 
  X=a(f:a(f:X g:nono h:_))
end

In fact, we can use the same procedure Unify for unification as before.

142b (Test AVM Unification)

declare
  (AVM's as Predicates)
  (Unify Terms)
proc{BrowseAVM AVMs} 
  {ForAll AVMs Browse}
  {Browse done}
end

/*
  {BrowseAVM {Unify AVM1 AVM2}}
  {BrowseAVM {Unify AVM2 AVM3}}
  {BrowseAVM {Unify AVM1 AVM3}}
*/

12.5 Exercises

- **Rule:** only stateful datastructures created inside the search space can be modified during search. It is possible to access global datastructures from within the search space, as this example demonstrates:

  declare
  C={NewCell 1}
  proc {Pred1 X} X={Access C} end
  (ExploreAll Pred1)

  but it is not possible to modify a global data-structure. The following example:

  declare
  C={NewCell 1}
  proc {Pred2 X} {Assign C 2} X={Access C} end
  (ExploreAll Pred2)
raises an exception because the search predicate $\text{Pred2}$ attempts to modify the global cell $C$. If you need to modify stateful datastructures inside your search predicate, this is only possible if these datastructures are also created inside the search predicate, i.e. inside the search space:

```plaintext
declare
proc (Pred3 X)
    C={NewCell 1}
in
    {Assign C 2}
    X={Access C}
end
(ExploreAll Pred3)
```

What is wrong about the following example? How should it be written instead?

```plaintext
declare
D={NewDictionary}
D.left := left(foo)
proc (Pred4 X)
    D.left = left(X)
    D.right := right(X)
end
(ExploreAll Pred4)
```

- **Rule**: a search predicate should not refer to global free variables. All identifiers within the search predicate must either be locally introduced, or, if they are not, refer to variables that were bound before the time the search is performed.

This is really just a variation on the previous rule: a unbound variable is a form of stateful datastructure: it can be assigned (bound) once. Therefore it too must be created by the search predicate. For example, the following code blocks:

```plaintext
declare
Y
proc (Pred6 X) Y=f(X) end
(ExploreAll Pred6)
```

In the Explorer, this is displayed by a yellow star. How should his code be written so that it doesn’t block?

- In previous exercises (see Section 10.7), we considered a ground representation of terms:

  \[
  \text{term ::= var | f(term ... term)}
  \]

Write a function $\text{MakeEncapsulatedTerm}$ which takes a ground representation of a term and returns an encapsulated term. Make sure that it works properly with respect to search. In particular, it should work with the following example:

```plaintext
declare
P1=(ToEncapsulatedTerm f(1 g(1 2)))
P2=(ToEncapsulatedTerm f(h(2 1) g(2 1)))
(ExploreAll proc {$ T} {P1 T} (P2 T) end)
```
Write a functional procedure `UnifyAllSubSequences` which inputs a list of terms \([T_1 \ldots T_n]\) encoded as unary predicates and outputs a chart with \(n+1\) positions. This chart contains an edge from position \(i\) to position \(j\) if and only if the conjunction of \(X=T_i, \ldots, X=T_j\) is satisfiable for some fresh variable \(X\). An edge between \(i\) and \(j\) should carry a solved form of this conjunction in terms of a predicate for \(X\).

For instance, your procedure `UnifyAllSubSequences` should be able to deal with the following input list \(Ts\):

```plaintext
declare proc{T1 X} Y in X=f(Y _ Y) end
proc{T2 X} X=f(a _ _) end
proc{T3 X} X=f(_ b _) end
proc{T4 X} Y in X=f(_ Y Y) end
proc{T5 X} X=f(_ _ c) end
Ts = [T1 T2 T3 T4 T5]
```

### 12.6 Program Collection

Here are the program generated from the chunks Test TimeUnif, Checking Unifiability, and Test AVM Unification:

```plaintext
declare proc(UnifBinTree X N)
  if N==0 then X=a
  else
    Y in
    X = f(Y Y)
    (UnifBinTree Y N-1)
  end
end
/*
declare X = (UnifBinTree $ 1000000){Browse done}
*/
/*
declare {Browse {Property.get time}}
*/
fun{Time P}
  T1 = {Property.get time}.user
  {P}
  T2 = {Property.get time}.user
  in
  T2-T1
end
fun{TimeUnif N}
  {Time proc($) {UnifBinTree _ N} end}
end
```
declare
proc{CoRef X} Y in X=f(Y Y) end
proc{ALeft X} X=f(a _) end
proc{BRight X} X=f(_,b) end
fun{Unify Ts}
  {Search.allP
    proc{$ X}
      {ForAll Ts proc{$ T} {T X} end}
    end
    1 _}
end
fun{IsUnifiable Ts}
  case {Unify Ts}
    of nil then false
    else true
  end
end
declare
proc{AVM1 X}
  Y in X=a(f:a{f:X g:Y h:Y})
end
proc{AVM2 X}
  Y in X=a(f:a{f:X g:a(f:Y) h:Y})
end
proc{AVM3 X}
  X=a(f:a{f:X g:nono h:_})
end
fun{Unify Ts}
  {Search.allP
    proc{$ X}
      {ForAll Ts proc{$ T} {T X} end}
    end
    1 _}
end
fun{IsUnifiable Ts}
/*
{Browse {TimeUnif 100000}}
{Browse {TimeUnif 1000000}}
{Browse {TimeUnif 2000000}}
{Browse {TimeUnif 4000000}}
*/
case {Unify Ts}
of nil then false
else true
end
end
proc {BrowseAVM AVMs}
  {ForAll AVMs Browse}
  {Browse done}
end
/*
{BrowseAVM {Unify AVM1 AVM2}}
{BrowseAVM {Unify AVM2 AVM3}}
{BrowseAVM {Unify AVM1 AVM3}}
*/
Chart Parsing for Unification Grammars

So far, we’ve looked at pure context-free grammars in which categories are atoms. In contemporary computational linguistics however, most grammars are feature-based, that is, they have categories that are feature terms (rather than atoms). In this section, we discuss first, how to implement feature-based grammars (also called, unification-based grammars as unification is the main operation used for comparing and combining categories) and second, how to implement a chart parser for such grammars.

13.1 Unification-based Grammars

As feature terms are the basic building blocks of UB-grammars, we start by discussing how they can be implemented in Oz. We then turn to the lexicon. Because feature-terms contain a lot more information than atoms for categories, the emergence of unification-based theories of grammar went hand-in-hand with a stark lexicalisation trend: information that earlier was part of the grammar proper (i.e. the syntactic rules and filters) got integrated in the lexicon. Practically this means that the lexicon contains a lot of information. To avoid redundancies and support maintenance, it therefore becomes important to structure the lexicon in some way. This is the object of the second section.

13.1.1 Feature Terms

Suppose we want to associate with the word "she" the information that it is a noun phrase in the third person whose number is singular and whose case is nominative. Using a ground term, we can represent this information as follows:

\[
\text{feat}(\text{np}, \text{third}, \text{sg}, \text{nom})
\]

The tuple has exactly four arguments whose positions indicate which information is filled in (i.e. category, person, number and case). It can only unify with a term of identical arity and compatible information. For instance, it unifies with the following term where _ represents an undefined value:
A more explicit way to represent the same information would be to use a record (feature tree). In that way, the various features are explicitly given and the order of feature-value pairs is irrelevant. For our example, the record can be completely described by the following ground feature term:

\[ \text{feat}(\text{cat:np \ pers:third \ nb:sg} \ '\text{case'}:\text{nom}) \]

which can unify with the following feature terms:

\[ \text{feat}(\text{cat:np \ pers:_ \ nb:sg} \ '\text{case'}:\text{nom}) \]
\[ \text{feat}(\text{pers:third \ cat:np \ nb:sg} \ '\text{case'}:\text{nom}) \]

but not with:

\[ \text{feat}(\text{pers:third \ cat:np \ nb:pl} \ '\text{case'}:\text{nom}) \]

In short, feature terms support a more explicit and thereby easier access to features for the grammar writer: s/he does not have to know which feature is in which position of the record nor in which order they come from, but can simply use their name to specify them in which order s/he pleases.

However note that in both cases, that a feature term fixes the arity of the feature tree it describes. This can make grammar writing very cumbersome as even if a feature value is undefined, it must be represent. For instance suppose that the feature tree for a verb has four features say Category, Form, Number and Pers, then a feature term representing this feature tree must always have four arguments even though perhaps some of the features are unspecified. For instance, given the verb "saw" the feature Number is undefined and feature tree can be described by the following feature term: \[ \text{feat}(\text{cat:verb \ form:fin \ nb:_ \ pers:_}) \] where \text{pers:} and \text{nb:} are in some sense superfluous.

When writing big grammars, such issues become crucial. There are basically two solutions to the problem: use open feature terms or let the grammar writer define the grammar as is intuitive (i.e. omitting features with undefined values) and write a compiler which translates the grammar so written into a record-based grammar.

The first solution involves working with open feature terms i.e. feature term which do not restrict the arity of the described tree. It yields just what’s needed for the grammar
writer but at the cost of efficiency: the grammar thus written will process much more slowly than one written with closed structures (structures with fixed arity). In Oz such open feature terms can be used though again much less efficiently than closed ones. Thus instead of using a closed terms as indicated above, we could use the open terms (the three dots indicate openness):

\[
\text{feat} (\text{cat}:\text{np} \ \text{pers}:\text{third} \ \text{nb}:\text{pl} \ \text{\textquote{case}}:\text{nom} \ ...)
\]

which will unify with any other term with identical label and compatible information e.g.

\[
\begin{align*}
\text{feat} (\text{cat}:\text{np}) \\
\text{feat} (\text{pers}:\text{third} \ \text{cat}:\text{np})
\end{align*}
\]

Neither the respective order of feature value pairs nor their number plays a role. In practice however, efficiency considerations strongly suggests the use of a grammar compiler.

### 13.1.2 Path equations

In the text above, we’ve associated a whole feature term with a lexical word. In practice however, it is more common to use path equations. A path equation is an equation of the form:

\[
X.\text{feat1}. ... .\text{featN} = \text{Value}
\]

where \(X\) names a feature term, \(\text{feat1}, ... ,\text{featN}\) are features and \(\text{Value}\) can either be a feature term, a path equation or a macro i.e. an abbreviation for some feature terms.

Using path equations, we can then rewrite the information associated with "she" as follows:

\[
\begin{align*}
X.\text{cat}=\text{np} \\
X.\text{agr}.\text{pers}=\text{third} \\
X.\text{agr}.\text{nb}=\text{sg} \\
X.\text{agr}.\text{\textquote{case}}=\text{nom}
\end{align*}
\]

Of course in this case it is not so clear why path equations are better than feature terms. We’ll see some more convincing examples in the next section where path equations are combined with macros to structure the lexicon.

Path equations can be written down immediately in Oz by using record selection. Note that the record selection on \(X\) suspends if the arity of \(X\) is unknown. In the above example, thus, \(X\) must be constrained a priori to be of the form
Chapter 13. Chart Parsing for Unification Grammars

Label (cat:_
  agr:Label1 (pers:_
    nb:_
    'case':_
    ...
  )
...

This restriction can be avoided by using feature constraints in Oz where the selecting operator is written by a hat ^ instead of the . . But using the hat will probably slow down your program. In practice it is much better to determine all arities by some types to be specified and compiled away before parsing starts.

13.1.3 Organising the lexicon

As already mentioned, unification-based grammars tend to concentrate information in the lexicon rather than in the syntactic rules. It is therefore particularly important to structure the lexicon and minimise redundancy. To do this a common practice consists in (i) introducing abbreviations for feature terms i.e. "macros" and (ii) structuring these macros into an inheritance hierarchy.

Here is a simple example. We have first to specify the set of features of different types of feature terms. The symbol @ prefixes type or macro names.

@type.verb: unit (cat:_
  phon:_
  subcat:_
  form:_
  agr:@type.agr)
@type.phrase: unit (cat:_
  phon:_
  agr:@type.agr)
@type.agr: unit (nb:_
  gender:_
  'case':_
  pers:_)

Next, define macros for feature terms of a particular type.

@verb (X): @type.verb (X),
  X.cat=v
@finite (X): @verb (X),
  X.form=finite
@intrans (X): @verb (X),
  X.subcat = [Y],
  type.phrase (Y),
  Y.cat=np,
  Y.agr.'case'=nom
@3pers (X): X.agr.pers=3
13.1. Unification-based Grammars

@sg(X): X.agr.nb=sg

runs(X): @finite(X), @intrans(X), @3pers(X), @sg(X)

In this lexicon, the feature term associated with "runs" is defined by a set of macros
and a path equation. The path equation says that the semantics of "runs" is the predi-
cate "run"; the macros @intrans says that intransitive verbs subcategorise for a single
element namely an np in the nominative case. Further it inherits from @verb which
requires that the category be a verb. The rest should be self-explanatory.

Macros and inheritance can be implemented in Oz using procedural abstraction: macros
are procedure which when called (recursively) expand the current feature term. Here
is an example:

```
Lexicon 151
```

```
declare
  Type = type(verb:fun($)
      unit(cat:_,
          phon:_,
          agr:{Type.agr},
          subcat:_,
          form:_)
  end

  phrase:fun($)
      unit(cat:_,
          phon:_,
          agr:{Type.agr})
  end

  agr: fun($)
      unit(nb:_,
          gender:_,
          'case':_,
          pers:_)
  end
)

proc{Verb X}
  X ={Type.verb}
  X.cat = v
end

proc{Finite X}
  {Verb X}
  X.form = finite
end

proc{Intrans X}
  {Verb X}
  Y
```
in
   X.subcat = [Y]
   Y=(Type.phrase)
   Y.cat = np
end

proc{ThirdSg X}
   X.agr.pers=3
   X.agr.nb=sg
end

fun{PhonToCat Phon}
   X
   in
      case Phon
      of runs then
         {Finite X} {Intrans X} {ThirdSg X}
      end
   end
   X.phon=Phon
   X
end

/*

declare X in {Browse {PhonToCat runs}}

*/

13.2 Unification-based Parsing

How does parsing with unification-based grammars differ from parsing with pure context-
free grammars? The difference of course lies in the categories which need more struc-
ture than simply being atoms.

13.2.1 Structured.categories

Let start with an example. Suppose that we want to express information on the number
feature of sentence phrase: If a sentences phrase consists of a noun phrase and a verb
phrase then all of them should have the same number feature, either singular or plural.
This can be expressed by the following two rules:

\[
s(nb:pl) \rightarrow np(nb:pl) \text{ vp}(nb:pl) \\
s(nb:sg) \rightarrow np(nb:sg) \text{ vp}(nb:sg)
\]

Note that the categories are generalized to feature trees. When considering rules of this
format, a parse tree for a sentence like the men run might look as follows.
Each inner node carries a feature tree for the category and has access to the child nodes. Each leave is labeled by a phoneme.

### 13.2.2 Is Context Free Grammar Enough?

Of course, we could express the same linguistic information by using standard trees. In order to do so we need some convention. We can assume for instance that the number feature does always reside at the first term position:

```
  s(pl
    np(pl
      det(pl the)
      n(pl men))
    vp(pl
      v(pl run))
```

Now, we could even encode the above rules within a context free grammar. As terminals we chose $sg, pl$ and for the non-terminals we need $s_sg, s_pl, np_sg, np_pl, vp_sg, vp_pl$.

```
  s -> s_pl
  s -> s_sg
  s_pl -> pl np_pl vp_pl
  s_sg -> pl np_sg vp_sg
```

This kind of an encoding does not look very promising. Now, categories carry additional information which is structured in itself. Therefore, it is inconvenient to encode categories as atoms; it is much nicer to encode all information of a category by a feature tree.

### 13.2.3 Rule Formats

Let us reconsider the two rules we stated from. Even though they are much better than the context free counterparts, they are still not very convincing: Why the hell should we need two distinct rules in order to express that the number of an s-phrase has to coincide with the numbers of its np-phrase and its vp-phrase? Clearly, this kind of doubling grammar rules would explode the size of the grammar when doing so consistently. A much better idea would be to consider the following rule which can be written in a unification grammar:

```
n(cat:s(nb:pl)
n(cat:np(nb:pl)
  n(cat:det(nb:pl)
    leave(the))
n(cat:n(nb:pl)
  leave(men)))
n(cat:vp(nb:pl)
  n(cat:v(nb(pl))
    leave(run)))))
```
Chapter 13. Chart Parsing for Unification Grammars

In fact, we do not even state here that $X$ should denote either $sg$ or $pl$. This information could be easily processed in Oz by using finite domain constraints but it is not strictly needed in the example grammar we present here.

Each rule of an unification based grammar is of one of the following two form:

\[ c \rightarrow c_1 \ldots c_n \]

where $c, c_1, \ldots, c_n$ are feature terms representing structured categories and $phon$ is an atom. Note that a feature term is like an attribute value matrix but with variables for expressing coreference, rather than integers.

When considering the grammar rules from above as Horn clauses with feature constraints, it is easy to define what a parse tree is. We only need one predicate symbol $p$ for parse tree and two function symbols $n$ for node and $leave$.

13.2.4 Chart Parsing

What we now have to do is to develop a parser which can deal with rules of a unification-based grammars according to the format given above. This is actually quite easy but there is one point that should be treated carefully namely, coreference checking.

Essentially, the idea for chart parsing with unification-based grammars is the same as for context free parsing. The only change consists in that the edges of a chart need carry structured categories. As we have already seen before, such categories can be represented by feature terms. Whereas previously, we could check for identity of categories, we cannot do so for feature terms, since they contain variables in order to express coreferences. Instead we have to unify such feature terms whereby coreferences are treated in the right way.

The only subtle point is the following: When unifying categories of two edges, then we are not allowed to change these edges since they might be needed again for other combinations. We also have to take care that unification does not raise a programming error due to failure. Actually, we know already how to do this: We simply encode the feature terms of the categories as predicates.

To illustrate the problem more concretely, we consider the following example:

\[ s(nb:X) \rightarrow s(nb:X) \, vp(nb:X) \]

Any employees with two supervisors can apply
Initially, "can apply" is a verb phase whose number is unspecified (it can be plural or singular). It can therefore combine with "two supervisors" and assuming that the sentence rule \( s - np \ vp \) enforces number agreement between subject np and vp, yields a plural sentence. The point is that as the category for "can apply" is matched against the vp category of the rule through unification, it should NOT be the case that this category suddenly be changed from being an np unspecified for number to being a plural np. Else it would no longer be able to combine with its real (singular) subject "Any employee with two supervisors" and a parse would fail to be found for a perfectly grammatical sentence.

To avoid that problem, the general strategy consists in creating copies of categories and of unifying these copies rather than the originals. In Oz, we do this by encapsulating constraints into predicates.

### 13.2.5 A Simplistic Example

For instance the lexicon of a simplistic unification-based grammar dealing with number agreement would look like this:

```oz
local
    fun{W Phon AVM} Phon#AVM end
Lexicon =
\[\{\begin{array}{c}
W \ john \ fun \ \{\} \ pn(nb:sg) \ end \\
W \ runs \ fun \ \{\} \ v(nb:sg) \ end \\
W \ run \ fun \ \{\} \ v(nb:_) \ end \\
W \ likes \ fun \ \{\} \ v(nb:sg) \ end \\
W \ sees \ fun \ \{\} \ v(nb:sg) \ end \\
W \ saw \ fun \ \{\} \ v(nb:_) \ end \\
W \ the \ fun \ \{\} \ det(nb:_) \ end \\
W \ man \ fun \ \{\} \ n(nb:sg) \ end \\
W \ men \ fun \ \{\} \ n(nb:pl) \ end \\
W \ 'with' \ fun \ \{\} \ prep \ end \\
W \ telescope \ fun \ \{\} \ n(nb:sg) \ end \} \]
LexRec = \{\begin{array}{c}
\text{List.toRecord lex Lexicon} \\
\text{in} \\
\text{fun\{PhonToCat Phon\} \\
LexRec.Phon} \\
\text{end} \\
\text{end} \}
```

Similarly, the rules are encapsulated into predicates too.

```oz
BinaryRules =
\[\begin{array}{c}
\text{[fun \ \{\} \ X \ in \ rule(left:s(nb:X) \ right:\{np(nb:X) \ vp(nb:X)\}) \ end} \\
\text{fun \ \{\} \ X \ in \ rule(left:np(nb:X) \ right:\{det(nb:X) \ n(nb:X)\}) \ end} \\
\text{fun \ \{\} \ X \ in \ rule(left:np(nb:X) \ right:\{np(nb:X) \ pp(nb:_)) \ end} \\
\text{fun \ \{\} \ X \ in \ rule(left:vp(nb:X) \ right:\{v(nb:X) \ np(nb:_)) \ end} \\
\text{fun \ \{\} \ X \ in \ rule(left:vp(nb:X) \ right:\{vp(nb:X) \ pp(nb:_)) \ end} \\
\text{fun \ \{\} \ X \ in \ rule(left:pp(nb:X) \ right:\{prep np(nb:X)\}) \ end} \}
```

\[\text{155a} \] (PhonToCat 155a) ≡
\[\text{155b} \] (UnifBinaryRules 155b) ≡
UnifUnaryRules =

UnaryRules =
[fun ($) X in rule(left:np(nb:pl) right:[n(nb:pl)]) end
fun ($) X in rule(left:np(nb:sg) right:[pn(nb:sg)]) end
fun ($) X in rule(left:vp(nb:X) right:[v(nb:X)]) end]

We can then re-use the chart parser developed for context-free grammars and modify it to deal with feature terms. All that needs to be done is to modify rule applications so that they produce the correct copies of categories and the appropriate results.

ProcessBinaryUnif =
proc(ProcessBinary Edge)
  {ForAll BinaryRules
    proc{§ Rule}
      {ForAll {Chart.get Edge.'end'}
        proc{§ RE}
          proc{BR Result}
            X = {Rule}
            C1 = {Edge.cat}
            C2 = {RE.cat}
            [C1 C2]= X.right
            in
            Result=X.left
          end
        Sol = {Search.allP BR 1 _}
        in
        case Sol
        of nil then skip
        [] [Pred] then
          {Agenda.push
            {NewEdge Pred Edge.begin RE.'end'}}
        end
      end
    end
  end}

ProcessUnaryUnif =
proc(ProcessUnary Edge)
  {ForAll UnaryRules
    proc{§ Rule}
      proc{UR Result}
        X = {Rule}
        C = {Edge.cat}
        [C]= X.right
        in
        Result=X.left
      end
    Sol = {Search.allP UR 1 _}
    in
case Sol
of nil then skip
[] [Pred] then
  (Agenda.push
   (NewEdge Pred Edge.begin Edge.'end'))
end
end

157a Unification-based Chart Parser

declare
local
% functors have to be importet
% (edge.oz)
% (chart.oz)
% Stack = ‘to be imported
% (lexicon.oz)
(UnifBinaryRules)
(UnifUnaryRules)
in
fun(Parse Phons)
  Chart = {NewChart Phons)
  Agenda = (NewStack)
  (procedure Process)
local
  (ProcessUnaryUnif)
  (ProcessBinaryUnif)
in
  proc(Process Edge)
    {ProcessUnary Edge}
    {ProcessBinary Edge}
end
end
(procedure Process)
(Words to Agenda)
(Process Edges)

in
  Chart
end
end
fun(Result Chart)
  {Map
    (Filter (Chart.get Chart.min)
      fun($ E) E.'end' == Chart.max end)
    fun($ E) {E.cat} end)
end

PhonList = phonlist([john sees the man ‘with’ the telescope]
13.3 The Complete Package

The following package file contains the entire code for a passive chart parser for unification grammars.

```
unif-parser.pkg
```

You can extract the sources from the package file as follows:

```
ozmake --extract --package=unif-parser.pkg
```

You can also install this package:

```
ozmake --install --package=unif-parser.pkg
```

After installation, all functors of the package are available at URI

```
x-ozlib://oz-kurs/unif-parser
```

13.4 Exercises

- Modify the grammar so that it contains subcategorisation information. The resulting parser should accept:

  Jonathan runs
  Jonathan likes Mary
  Jonathan gives a book to Mary

  and reject

  Jonathan likes John
  Jonathan gives John a book

  `unif-parser.pkg`
* Jon runs mary
* Jon likes
* Jon gives a book on Mary

- Structure the lexicon used for the first exercise using macros and an inheritance hierarchy as described in the lecture notes.

- Modify the grammar and the parser so that context-free information is separated from the other constraints (e.g. agreement and subcategorisation). Given the appropriate modifications in the grammar, the parser should first check whether the context-free part of the rule is verified before applying other constraints.
14.1 Non concurrent active chart parsing

So far we’ve seen chart parsers whose chart contains constituents already found by the parser. One can also use the chart to store "parsing-hypotheses" that is hypotheses about what to look for next. These parsing hypotheses are represented using so-called "dotted rules" that is, rule of the following form:

\[ X \rightarrow Y_s . Z_s \]

where X is a category and Ys, Zs are (possibly empty) sequences of categories. The intuitive interpretation of this rule is:

"I’ve found the sequence Ys and if the sequence Zs can also be found then an X will have been found."

Whereas in a passive chart parser, the chart records constituents that have been parsed, in an active chart parser, the chart records parsing hypotheses. Practically, this means that edges in the chart instead of being labelled with categories, are labelled with dotted rules. There is a special kind of dotted rules however: dotted rules whose dot is right at the end of the rhs e.g.

\[ S \rightarrow NP VP. \]

In effect such a rule indicate a "complete" constituent. Nothing more is looked for, all the constituents on the rhs of the rule have been found and therefore an S has been parsed. Edges which are labelled with such dotted rules are called "passive edges"; other edges are called "active edges" (they still "look for something").

Given these preliminaries, we can now describe the various components of an active chart parser as follows:

- An initialisation of the chart
- A "fundamental rule" that combines an active edge with a passive edge.
• A control strategy (either top-down or bottom-up).

• A search strategy (either breadth-first or depth-first)

As this description indicates, active chart parsers can be of different types e.g. bottom-up and depth-first, top-down and breadth first etc. Here we concentrate on bottom-up, depth-first chart parsers. For such parser, the initialisation consists in adding to the chart the passive edges licensed by the words in the input string. For instance, given the sentence "the cat runs", the chart will be initialised to:

```
edge(1,2,Det --the.)
edge(2,3,N --cat.)
edge(3,4,V --runs.)
```

The fundamental rule combines an active edge with a passive edge as follows:

| Active edge | edge(i,j,X -- Ys.C Zs) |
| Passive edge | edge(j,k, C -- Ws.) |
| Resulting edge | edge(i,k,X -- Ys.CZs) |

The control strategy specifies how rules are invoked (in this case bottom-up) and more specifically how to create active edges:

```
Passive edge: edge(j,k, C -- Ys.)
Rule: X -- C Xs
Resulting active edge: edge(j,k,X -- C Xs)
```

If the chart contains a constituent of category C from position j to position k and if the grammar contains a rule with left-corner is C and general format X -- C Xs, then an active edge is produced which stretches from position j to position k and looks for Xs to form an X.

Finally the search strategy is defined by type of agenda we use: if we use a stack, the parser proceeds depth-first through the search space; if we use a queue, it proceeds breadth-first.

Given these basic components an active chart parsing algorithm can be defined as follows:

• The top loop is as for any chart parser with agenda: initialise the agenda, process the agenda until it is empty and look the chart up for edges stretching from beginning to end and labelled with the S category.

• To process the agenda:
  
  – Retrieve an edge from the agenda,
  – Add this edge to the chart, and active edges it implies to the agenda.
14.2 Exercise

Modify the unification chart parser given in section 12.3 to an active chart parser.
Part IV

Concurrent Constraint Programming and Semantic Underspecification
Concurrent Constraint Programming

In the remainder of the lecture, we introduce concurrent constraints programming in depth and apply it to natural language processing.

In this chapter, we recall the general ideas of concurrent constraint programming and introduce the corresponding programming concepts provided by Oz. We illustrate these concepts at some small examples relevant to computational linguistics.

We consider agreement checking based on finite domain constraints and present a solver for proposition logics which is based on disjunctive propagators. Furthermore, we introduce finite set constraints that will be needed all over the place in the follow up chapters.

15.1 General Ideas (Recalled)

15.1.1 Combinatoric Explosion

Concurrent constraint programming allows to solve combinatoric problem efficiently. The problem with combinatoric problems is that the number of combinations to be considered may grow exponentially. Therefore, it is not possible to generate and test all combinations. Generate and test quickly leads to a combinatoric explosion even for small problems.

Concurrent constraint programming helps to avoid the combinatoric explosion. The idea is to perform simple inferences first in order to avoid superfluous case distinctions. This method is called ‘propagate and distribute’ where propagation performs simple inferences and distribution case distinctions.

15.1.2 Constraint Store and Propagators

Most combinatoric problems can be modelled naturally by using some kind of logic formulas. We call logic formulas constraints. Two forms of constraints can be distinguished: basic constraints are so simple that they can be solved deterministically, whereas complex constraints cannot. Typical examples for basic constraints are symbolic equations as used in first-order unification. A good examples for complex constraints are arithmetic equations.

In what concerns concurrent constraint programming, basic constraints are accumulated incrementally in a constraint store whereas complex constraints are turned into
propagators. Propagators are agents that observe the constraint store. A propagator infers consequences of the constraints in the store and the formula by which it is defined. Such consequence are new constraints which can be either basic or complex again.

15.1.3 Distribution

In general, constraint propagation is incomplete. This means that a combinatoric problem cannot be solve due to simple inferences only. Thus, a propagation process may become stable, without having determined a solution of a combinatoric problem.

What is missing so far is the possibility to perform case distinctions. This can be done by distribution steps of the following form where A,B,C are propositional formulas:

\[( A \lor B ) \land C \Rightarrow ( A \land C ) \lor ( B \land C )\]

15.1.4 Ambiguities versus Disjunction

Much of natural language processing can be understood as solving combinatoric problems. The reason is that natural language is full of ambiguities which one may express by disjunctive logic formulas.

As an example, we consider agreement information for the the German adjective schön (nice). Its inflection schönen can have be used with a set of possible gender-case-number information. The possibilities can be described by the following propositional formula:

\[
\begin{align*}
\text{masc} & \land \text{dat} & \land \text{sg} & \Rightarrow \text{dem schönen Mann} \\
\lor \text{masc} & \land \text{acc} & \land \text{sg} & \Rightarrow \text{den schönen Mann} \\
\lor \text{masc} & \land \text{nom} & \land \text{pl} & \Rightarrow \text{die schönen Männer} \\
\lor \text{masc} & \land \text{gen} & \land \text{pl} & \Rightarrow \text{der schönen Männer} \\
\lor \text{masc} & \land \text{dat} & \land \text{pl} & \Rightarrow \text{den schönen Männern} \\
\lor \text{masc} & \land \text{acc} & \land \text{pl} & \Rightarrow \text{die schönen Männer} \\
\lor \text{fem} & \land \text{gen} & \land \text{sg} & \Rightarrow \text{der schönen Frau} \\
\lor \text{fem} & \land \text{dat} & \land \text{sg} & \Rightarrow \text{der schönen Frau} \\
\lor \text{fem} & \land \text{nom} & \land \text{pl} & \Rightarrow \text{die schönen Frauen} \\
\lor \text{fem} & \land \text{gen} & \land \text{pl} & \Rightarrow \text{der schönen Frauen} \\
\lor \text{fem} & \land \text{dat} & \land \text{pl} & \Rightarrow \text{den schönen Frauen} \\
\lor \text{fem} & \land \text{acc} & \land \text{pl} & \Rightarrow \text{die schönen Frauen}
\end{align*}
\]

The next question is how to express disjunctive information in a concurrent constraint language such as Oz.

15.2 Finite Domain Constraints

We consider the perhaps most popular class of constraints that are called finite domain (FD) constraints.

15.2.1 FD-Membership

Finite domain variables are variables that can denote one member of a finite set of integers. They can be used to express a simple form of disjunction. This form of disjunction is important when it comes to distribution.
A finite domain variable is a variable whose value is a natural number. Furthermore, the value of a finite domain variable can be constrained by some finite domain of natural numbers. For instance, the FD-membership constraint

\[ X :: 1 \# 5 \]

is equivalent to \( X \in \{1, 2, 3, 4, 5\} \) which in turn is equivalent to the disjunction:

\[ X = 1 \lor X = 2 \lor X = 3 \lor X = 4 \lor X = 5 \]

An FD-membership constraint such as \( X :: 1 \# 5 \) can be represented directly in the Oz constraint store. It is neither a propagator nor does it raise any case distinction.

### 15.2.2 FD-Propagators

Oz features several propagators for finite domain variables. We only present examples here and refer to the finite domain programming tutorial otherwise. The most important propagators are those for arithmetics. Propagators can be distinguished from pure evaluators by the colons like in \( =: \) or \( =<:\).

\[
3 \times X - Y =: 4 \times Z \] % linear arithmetics
\[
3 \times X - Y =<: 4 \times Z \] % inequations

For each FD-variable, a finite domain of possible values is maintained in the constraint store. What these propagators are doing is to restrict the upper and lower bounds of the domains of its variables; values from the interior of a finite domain are not excluded even if they contradict the logical semantics of the propagator.

Another useful propagator is the all-distinct propagator.

\[
\{ \text{FD.distinct \ [U V W X Y Z]} \}\]

Whenever the value of one of the variables in the list \( \{U V W X Y Z\} \) gets determined, this value is excluded from the domain of the others. The all-distinct propagator requires linear space in the number of variables it administrates, in contrast to a naive implementation which require quadratic space:

\[
\begin{align*}
U &= V & U &= W & U &= X & U &= Y & U &= Z \\
V &= W & V &= X & V &= Y & V &= Z \\
W &= X & W &= Y & W &= Z \\
X &= Y & X &= Z \\
Y &= Z
\end{align*}
\]

More on FD-propagators can be found in the tutorial on finite domain constraint programming in Oz.

### 15.2.3 FD-Reflect

to be written
15.2.4 FD-Distribution

Oz supports distribution for finite domain variables but only within encapsulated search. This is only operation which creates a choice node in a search tree.

Distributors can be created by applying the procedure `FD.distribute` to the name of a distribution strategy and a list of variables. For instance, the a distributor for the strategy first-fail (ff) picks a variable X of minimal current domain, splits this domain into two disjoint parts, each of which it considers in an independent part.

\[ X \in D_1 \cup D_2 \implies X \in D_1 \lor X \in D_2 \]

Given that the domain \( D_1 \cup D_2 \) is split, encapsulated search process both possibilities \( X \in D_1 \) and \( X \in D_2 \) independently.

As said before, the split operation is evoked by the procedure `FD.distribute`. For instance, the domains of \( X \) and \( Y \) are split when in the following example:

```
170a (Distribution [170b]=
   declare
   proc{Problem Sol}
   X Y
   in
   Sol = solution(x:X y:Y)
   X :: 1#5
   Y :: 2#3
   {FD.distribute ff [X Y]}
   end
   {Explorer.all Problem}
```

Distribution in Oz is support during encapsulated search only (but NOT on top-level). This means that a problem has to encapsulated into a unary procedure which is then and then passed to the Oz-Explorer. Applying this procedure directly does not lead to distribution on top-level.

Note also that a distributor such as `{FD.distribute ff [X Y]}` blocks its thread (all subsequent statements) until distribution has happened (for ever on top-level). Therefore, a distributor should always be the last statement of its thread. This can be achieved either by writing it into the last line of the problem definition or by using a new thread anyway.

```
thread {FD.distribute ff [X Y]} end
```

15.3 Finite Domains and Agreement Testing

A nice application of finite domain constraints in computational linguistics concern agreement checking. This can be done by describing Cartesian products of finite sets by finite domain constraints.
15.3. What’s the Problem

Suppose for instance, you want to parse a sentence like

Die schöne nette Frau sieht der kleine Mann

The first 5 words of the sentence do not determine whether the noun phrase
die schöne nette Frau

is nom_sg_fem or else acc_sg_fem. Only because the noun phrase der kleine Mann
is nom_sg_masc it follows that this noun phrase is the subject and hence the other one
the object. Thus die schöne nette Frau has to be nom_sg_fem.

The question is now how to process all the disjunctive information about agreements of
number, case, and gender. Of course it would be much better to propagate agreement
information that enumerate all possible case. For instance, we would like to conjunction
of the following agreement informations:

\{\text{ComputeAgree}\
  [[[sg 2 masc] [pl 3 masc] [pl 3 fem]]
  [[[sg 2 masc] [pl 2 masc] [pl 3 fem] [pl 2 fem]]}\}

The result should be [[[sg 2 masc] [pl 3 fem]]].

What we now need is a bijection between a product of finite sets $D_1 \times \ldots \times D_n$ and a set
of integers of the form $\{0, \ldots, m\}$. This bijection has to be turned into a bidirectional
converter.

15.3.2 Finite Domains as Abstract Data Structures

A concrete finite domain consists is a finite set of elements of some type `$a$. A domain
is an abstract data structure for a concrete finite domain. It has the following type:

\begin{verbatim}
(type domain $ \equiv $
  %% type element = 'a
  %% type domain = unit(toIndex:element->int
  %% toElement:int->element
  %% size:Size

Every domain $\text{Dom}$ has a function $\text{Dom.toIndex}$ that maps elements \{d_1, \ldots, d_n\} of
the concrete domain to integer indexes \{0, \ldots, n – 1\}. Furthermore, it provides an
inverse function $\text{Dom.toElement}$.

15.3.3 Finite Domain Functor

We next define a functor that accounts for domains of atoms or integers and for products
of domains.
functor
import
I(sum:Sum
prod:Prod) at 'x-ozlib://oz-kurs/domains/int.ozf'

export
Make % list(feature) -> domain
Product % list(domain) -> domain

(type domain [171a])
define
(Make Domain [172a])
(Product of Domains [173a])
end

15.3.4 Domains of Atoms and Integers

We start with domains whose elements are atoms or integers. These can be created as follows:

172a (Make Domain [172a] ≡
  fun { Make Es } % list(feature) -> domain
    IndTable = { List.toTuple unit Es } % record(int:feature)
    fun { ToElement Ind } 
      IndTable.(Ind+1)
    end
    Size = { Length Es }
    EleTable = { MakeRecord unit Es } % record(feature:int)
    % the fields of EleTable are still free variables
    for I in 0..Size-1 do
      EleTable.{ ToElement I } = I
    end
    fun { ToIndex Ele } 
      EleTable.Ele
    end
  in
    unit (toIndex : ToIndex
      toElement : ToElement
      size : Size)
  end

15.3.5 Auxiliary Arithmetic Functions

We need to auxiliary functions to compute weighted sums and products for a given range of integers.

\[ \text{sum}(m, n, f) = \sum_{i=m}^{n} f(i) \quad \text{and} \quad \text{prod}(m, n, f) = \prod_{i=m}^{n} f(i) \]

These functions can be easily defined by recursive procedures (or alternatively by looping relative to a global cells containing an integer):
15.3. Finite Domains and Agreement Testing

We can now define products of domains. The elements of product domains are vectors (n-tuples) of elements of other domains. From the operational view, we have to provide converter functions that map vectors in \( D_0 \times \ldots \times D_n \) to integer indexes and vice versa.

For simplicity, let us assume first that the elements of the given domains \( D_0, \ldots, D_n \) are already integer indexes:

\[
D_0 = \{0, \ldots, m_0\} \\
\ldots \\
D_n = \{0, \ldots, m_n\}
\]
We can then describe the index of a vector \((d_0, \ldots, d_n) \in D_0 \times \ldots \times D_n\) by the following formula.

\[
\text{index}((d_0, \ldots, d_n)) = \sum_{i=0}^{n} d_i \prod_{j=0}^{i-1} m_j + 1
\]

This formula is well known for the case where all domains contain the digits \(\{0, \ldots, 9\}\), i.e. if \(m_1 = \ldots = m_n = 9\). In this case, the formula tells us how to read a sequence of digits as a decimal number (up to inversion).

\[
\text{index}((d_n, \ldots, d_0)) = \sum_{i=0}^{n} d_i \cdot 10^i
\]

For implementing the conversion from vectors to numbers, we use an Oz table \texttt{DomTable} to index the domains \(D_0, \ldots, D_n\) by integers \(1, \ldots, n + 1\). Note that we exploit the conversion functions of the factors of the product to define the conversion functions of the product itself.

174a \(\text{ToIndex}\) \(\text{[174a]}\) =

\[
\text{fun}(\text{ToIndex} \ Vector) \ \% \ \text{list(feature)} \rightarrow \text{int} \\
\text{if} \ (\text{Length} \ Vector) \neq N \\
\text{then} \ \text{raise unit(msg:'length of vector not valid') } \\
\text{function:'ToIndex' } \\
\text{file:'domain.oz'} \\
\text{vector:Vector} \\
\text{length:N)} \\
\text{end} \\
\text{end} \\
\text{VectorTable} = \{\text{List.toTuple unit Vector}\} \\
in \\
\{\text{Sum} \ 1 \ N \ \text{fun}(\$ \ I) \\
\text{DomTable.I.toIndex VectorTable.I} \ * \\
\text{Prod} \ 1 \ \text{N=1 fun}(\$ \ J) \\
\text{DomTable.J.size} \\
\text{end} \}
\]

The back translation from indexes to vectors of elements requires some division modulo operations, which require some care.

174b \(\text{ToVector}\) \(\text{[174b]}\) =

\[
\text{fun}(\text{ToVectorHelp} \ \text{Index} \ N \ \text{InVector}) \\
\text{if} \ N=0 \\
\text{then} \ \text{InVector} \\
\text{else} \\
\text{Size=Prod} \ 1 \ N-1 \ \text{fun}(\$ \ I) \ \text{DomTable.I.size end} \\
\text{NextVector} = \{\text{DomTable.N.toElement (Index div Size)}\} \ \| \ \text{InVector} \\
in \\
\{\text{ToVectorHelp} \ \text{(Index mod Size)} \ N-1 \ \text{NextVector}\}
fun(ToVector Index) % int -> list(feature)
  if Index < 0 orelse Index > Size-1
  then
    raise error(function:’ToVector’
    file:’domain.oz’
    msg:’index out of range’
    index:Index
    range:0#(Size-1))
  end
end
{ToVectorHelp Index N nil)
end

15.3.7 Application to Agreement Checking

(Test products of finite domains)

%% import the domain functor
declare [Dom] = {Module.link ['domain.ozf']}
declare ’export’(make:MakeDomain
  product:Product ...) = Dom

%% define a single domain

declare DomainSpec = [sg pl]
declare Domain = {MakeDomain DomainSpec}
declare I = {Domain.toIndex sg} {Inspect I}
declare E = {Domain.toElement 0} {Inspect E}

%% define a product of domains

declare DomainSpecs = [[sg pl] [1 2 3] [masc fem]]
declare Domains = {Map DomainSpecs MakeDomain}
declare Domain = {Product Domains}

%% convert vectors to indexes

for V in [[sg 1 masc] [pl 1 masc] [sg 2 masc] [pl 2 masc] [sg 3 masc] [pl 3 masc] [sg 1 fem] [pl 1 fem] [sg 2 fem] [pl 2 fem] [sg 3 fem]...}
[pl 3 fem]]
do
   {Inspect (Domain.toIndex V)}
end

{Inspect Domains}

%% convert indexes to vectors %%%%%%%%%%%%

for I in 0..11 do
   {Inspect {Domain.toElement I}}
end

%% test agreement %%%%%%%%%%%%%%%%%%%%%%%%
declare proc {WaitUntilStable}  %% works only for encapsulated search
   {FD.distribute ff [1]}
end

declare fun {Agree A1 A2}
   proc {Constraints As}
      FD1 FD2
      FD1 :: (Map A1 Domain.toIndex)
      FD2 :: (Map A2 Domain.toIndex)
      FD1 = FD2
      {WaitUntilStable}
      Inds = {FD.reflect.dom FD1}
      in
         As = {Map Inds Domain.toElement}
      end
      in
         {Search.all Constraints 1 _}.1
      end
   {Inspect 'agreement intersection'}
   {Inspect {Agree
               [[sg 2 masc] [pl 3 masc] [pl 3 fem]]
               [[sg 2 masc] [pl 3 fem] [pl 2 fem]]})

15.3.8 The Complete Package: domains.pkg

The following package file contains the entire code for a passive chart parser for unification grammars.
domains.pkg

You can extract the sources from the package file as follows:

1 domains.pkg
2 domains.pkg
15.4 Finite Set Constraints

Finite set constraints are also known from constraint programming but much less popular than finite domain constraints. Nevertheless, it turns out that finite set constraints are extremely useful for natural language processing.

A finite set (FS) variable denotes a finite set of integers. A finite set constraint describes the values of finite set variables based on the usual set operations. The reader should carefully note the difference between finite domain (FD) variables and finite set variables. An FD-variable denotes a single integer which can be described by a finite set of possibilities. A FS-variable denotes a finite set of integers which may be empty or contain more than one element.

There is two forms of basic finite set constraint which can be entered directly into the Oz-constraint-store. The upper:

\[
X = \{\text{FS}. \text{var}. \text{upperBound 1} \# 6\} \\
X = \{\text{FS}. \text{var}. \text{lowerBound 2} \# 4\}
\]

The former constraint states an upper bound \(X \subseteq \{1, 2, 3, 4, 5, 6\}\) whereas the latter requires a lower bound \(\{2, 3, 4\} \subseteq X\). Beside of basic set constraints there are the following set propagators:

\[
\{\text{FS}.\text{subset} X Y\} \\
X = \{\text{FS}.\text{union} Y Z\} \\
X = \{\text{FS}.\text{partition} [U V W]\} \\
\{\text{FS}.\text{include} I X\}
\]

The declarative semantics of these constraints are rather obvious:

\[
X \subseteq Y \\
X = Y \cup Z \\
X = U \cup V \cup W \\
I \in X
\]

Operationally, set propagators increase upper bounds and decrease lower bounds of set variables in the constraint store. The propagation behaviour can be tested at the following example:
\begin{verbatim}
declare
X={FS.var.upperBound 1#6}
Y={FS.var.upperBound 1#2}
Z={FS.var.upperBound 2#3}
{FS.subset X {FS.union Y Z}}
{FS.subset Y Z}
{FS.include 2 Y}
{Browse [X Y Z]}
\end{verbatim}

There are more important set constraints in Oz that we will not present in this reader. Note also that we do not need distributors for set constraints.

### 15.5 Disjunctive Propagators

There are several ways in Oz to express disjunctive information. Disjunctive information can be written into the constraint store by using finite domain membership constraints such as \( X :: 1 \# 5 \). Another convenient way is to use disjunctive propagators which can be written in Oz by or-statements.

As we will see, both forms of disjunctive information can be interlocked in an useful manner.

#### 15.5.1 or-Statements

An \textit{or-statement} states disjunctive information in terms of a propagator. For instance, the congruence information of the German adjective ‘schönen’ can be described as follows:

\begin{verbatim}
178a (Or Statement 178a) ≡
  or [Gen Cas Num]=[masc dat sg] then skip % dem schönen Mann
  [] [Gen Cas Num]=[masc acc sg] then skip % den schönen Mann
  [] [Gen Cas Num]=[masc nom pl] then skip % die schönen Männer
  [] [Gen Cas Num]=[masc gen pl] then skip % der schönen Männer
  [] [Gen Cas Num]=[masc dat pl] then skip % den schönen Männern
  [] [Gen Cas Num]=[masc acc pl] then skip % die schönen Männer
  [] [Gen Cas Num]=[fem gen sg] then skip % der schönen Frau
  [] [Gen Cas Num]=[fem dat sg] then skip % der schönen Frau
  [] [Gen Cas Num]=[fem nom pl] then skip % die schönen Frauen
  [] [Gen Cas Num]=[fem gen pl] then skip % der schönen Frauen
  [] [Gen Cas Num]=[fem dat pl] then skip % den schönen Frauen
  [] [Gen Cas Num]=[fem acc pl] then skip % die schönen Frauen
end
\end{verbatim}

An or-statement consists of a set of \textit{clauses} each of which has a \textit{guard} and a \textit{body}. For instance, the guard of the second clause above is the constraint \([\text{Gen Cas Num}]=\text{[masc acc sg]}\). The body of all above clauses is the expression \textit{skip}. The distinct behaviour of guards and bodies is explained in the next section.
15.5.2 Operational Semantics

An or-statement behaves as a propagator which concurrently investigates all its alternatives. Each clause is continually monitored.

\[ \text{or GUARD then BODY end} \implies \text{Guard Body} \]

An or-statement reduces all its guards in parallel such that the constraints of the guard remain properly separated from those in the global constraint store. We say that every guard is executed in its own computation space.

As soon as the guard of a clause becomes inconsistent with the global constraint store, the clause is removed. An or-propagator block its thread. It goes away when one a single clause remains. If an or-statement goes away then the body of the remaining clause is executed.

We can observe the semantics of or-statements by feeding the following pieces of code:

```
(Or Statement)
{Browse 'An or-propagator blocks its thread'}
{Browse 'Inconsistent clauses are removed'}
{Browse 'An or-propagator with a single clause goes away'}
{Browse ['gender:' Gen 'case:' Cas 'number:' Num]}

/*
Cas=nom Gen=fem
*/
```

When having feeded these lines, nothing should happen since the or-propagator blocks its thread. But when feeding the additional constraint \( \text{Cas=nom Gen=fem} \) then the thread following the or-statement becomes active. Note in particular that the variable \( \text{Num} \) is determined to the value \( \text{pl} \).

15.5.3 Propositional Logic

We illustrate the usage of disjunctive propagators at an example for propositional logics. We would like to solve the following propositional logics formula:

\[ A_1 \lor \neg A_2 \land \neg A_1 \lor A_2 \]

The variables denote Boolean values that we represent by 0 and 1. We would like to enumerate all solutions of the above formula. We would like the following propagation rules to apply (unit propagation):

If \( A_1=0 \) then \( A_2=0 \). If \( A_1=1 \) then \( A_2=1 \).

We can define the following propagators to achieve this behaviour.

```
(Disjunctive Propagators (Unit Propagation))
```
If \( A_1=0 \) is imposed then the first clause of the first propagator gets removed since its guard becomes inconsistent with the global store. Only the second clause remains. Thus the first propagator commits to its second clause and thereby adds the constraint \( A_2=0 \) to the global store.

Impose \( A_1=1 \) has a symmetric effect. The first clause of the second propagator gets removed since its guard becomes inconsistent. Only its second clause remains such that the second propagator commits to its second clause. Thereby \( A_2=1 \) is imposed globally.

Note that each of the or-statement propagators has to run in its own thread. Otherwise the first stated or-statement would block the execution of the second or-statement.

We obtain the following solver for the above propositional formula.

180a (Solve the Propositional Formula 180a)

\[
\begin{align*}
\text{declare} & \quad \text{proc\{Predicate Sol\)} \\
\text{A1, A2} & \\
\text{Vars = [A1 A2]} & \\
\text{Vars ::: 0} & \\
\text{in} & \\
\text{Sol = vars(a1:A1 a2:A2)} & \\
\text{thread or A1=1 [] A2=0 end end} & \\
\text{thread or A1=0 [] A2=1 end end} & \\
\{\text{FD.distribute naive [A1 A2]}\} & \\
\end{align*}
\]

(Explorer.all Predicate)

15.5.4 Choice Points versus Choice Variables

Unlike in Prolog, an Oz disjunction does not create a choice point, i.e., a case distinction. The only way to commit to one alternative is to cause all the others to become inconsistent.

180b (Disjunctive Propagator 180b)

\[
\begin{align*}
\text{or \{Equal N M\} then skip} & \\
[\] \{\text{DomPlus N M}\} then skip & \\
[\] \{\text{DomPlus M N}\} then skip & \\
[\] \{\text{Side N M}\} & \\
\end{align*}
\]

Yet, in order to perform search, we often need to force commitment to one or the other alternative. The standard trick in constraint programming is to introduce a choice variable, also known as a control variable.

We control the alternatives by a choice variable \( C \). \( C \) is a finite domain variable with the domain \( 1 \# 4 \); simply by equating it with 1, 2, 3 or 4, we can commit to the corresponding alternative.
15.6. Selection Constraint

A selection constraint describes a value that is obtained by selecting elements of a list of values.

Selection constraint select elements of a list by their index, where the index of the i-th element of a list is simply i itself. The power of the selection constraint depends on the fact that the indexes of selected elements need not to be known at before hand. They can be constraint themselves by finite domain of finite set constraints.

Oz supports two basic forms of selection constraints one for integers and one for sets of integers. Both of them are not available in the base environment but can be downloaded as a package form the MOGUL archive.

15.6.1 Element Constraint

We start with selection constraints over integers which is called element constraint in the literature.

15.6.1.1 Syntax and Semantics

An element constraint has has the following form:

\[ Z = \langle X_1, \ldots, X_n \rangle[Y] \]

All a variables \( Z, X_1, \ldots, X_n, Y \) denote integers. These integers may be restricted by additional finite domain constraints. The above element constraint is equivalent to

\[ Z = X_Y \land Y \in \{1, \ldots, n\} \]

i.e. it says that \( Z \) is the \( Y \)-th element of the list \( X_1, \ldots, X_n \).
15.6.1.2 Example (Finite Functions)

Element constraints can be used to describe applications of finite functions on integers partially. For instance,
\[ Y = 3^X \land X \in \{1, 2, 3\} \]
is equivalently expressed through the following element constraint:
\[ Y = (3, 9, 27)[X] \]
This constraint says that \( Y = f(X) \) where \( f \) is the finite function defined through:
\[ f : \{1, 2, 3\} \rightarrow \text{Nat}, f(Z) = 3^Z \text{ for all } Z \]

15.6.1.3 Propagation Rules

The operational semantics of element constraints can be specified through the following two propagation rules where \( i, j \) are natural numbers and \( S \) a finite set of natural numbers.
\[
\begin{align*}
  i \neq Z \land i = X_j \land Z = \langle X_1, \ldots, X_n \rangle[Y] & \rightarrow j \neq Y \\
  Y \in S \land \bigwedge_{j \in S} i \neq X_j \land Z = \langle X_1, \ldots, X_n \rangle[Y] & \rightarrow i \neq Z
\end{align*}
\]

15.6.2 Finite Set Selection Constraints

We discuss three versions of set selection constraints.

15.6.2.1 Syntax and Semantics

The first version is similar to the element constraint except that we now select an element from a list of finite sets (instead of a list of integers).
\[ S = \langle S_1, \ldots, S_n \rangle[Y] \]
The variables \( S, S_1, \ldots, S_n \) denote finite sets of integers whereas \( Y \) has type integer. The constraint is equivalent to
\[ S = S_Y \land Y \in \{1, \ldots, n\} \]
There are more powerful variants of set selection constraints. One can select a subset of elements of a list and apply a set operator to them. The variant for the union operator has the following form:
\[ S = \cup\langle S_1, \ldots, S_n \rangle[S_0] \]
All variables \( S, S_1, \ldots, S_n, S_0 \) denote finite sets of integers and may be restricted further by finite set constraints. This selection constraint is equivalent to
\[ S = \cup\{S_X \mid X \in S_0\} \land S_0 \subseteq \{1, \ldots, n\} \]
i.e. it says that \( S \) is the union of all those elements \( S_X \) in the \( S_1, \ldots, S_n \) whose index \( X \) belongs to \( S_0 \).
There exists an analogous selection constraint which employs the intersection instead of the union operator.
\[ S = \cap\langle S_1, \ldots, S_n \rangle[S_0] \]
15.6.2.2 Example

This example illustrates the propagation behaviour of the first of the above set selection constraints.

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

declare S1 = {FS.value.make [5]}
declare S2 = {FS.value.make [1 21]}
declare S3 = {FS.value.make [1 77]}
declare I = {FD.decl}
declare S = {Select.fs [S1 S2 S3] I}

{Browse S}
{Browse I}
```

When feeding this code to the Oz Emulator the browser displays:

```ozone
%% S{{}..{1 5 21 77}}#{1#2}
%% I{1#3}
```

This means that $\emptyset \subseteq S \subseteq \{1, 5, 21, 77\}$, card($S$) $\in \{1, 2\}$ and $I \in \{1, 2, 3\}$. Next, lets feed the following constraint in addition:

```ozone
{FS.exclude 5 S}
```

The Browse now updates its output to:

```ozone
%% S{{1}..{1 21 77}}#{2}
%% I{2#3}
```

Since the first set contributes 5 which does not belong to does $S$ the first set cannot be selected so that $I \neq 1$. Thus, either the second or third set must be selected. Since both of them have cardinality 2 this must also hold for $S$. Furthermore, $S$ must be a subset of both sets and thus subsumed by $\{1, 21, 77\}$.

15.7 Reified Constraints

A reified constraint consists of a constraint and a variable which denotes its truth value. For instance, \text{n} \text{truth}(X = 2) = N is equivalent to $(X = 2 \land N = 1) \lor (X \neq 2 \land N = 0)$. Reifed constraints are useful for for expressing propositional formulas over constraints. Most typically, they are used for expressing implications or equivalences between arbitrary constraints $C$ and $D$.

$$
C \leftrightarrow D \quad \text{iff} \quad \text{truth}(C) = \text{truth}(D)
$$

$$
C \Rightarrow D \quad \text{iff} \quad \text{truth}(C) \leq \text{truth}(D)
$$

Oz provides reified versions of most finite domain constraints and some set constraints directly. Reified versions \text{n} \text{truth}(C) = N of arbitrary constraints $C$ can be user defined in Oz by or-propagators. The only restriction is that the negation Not$C$ has to be available.
Propagation of reified constraints is bidirectional. For instance, $N = \text{truth}(X = 1) \land N = 1$ propagates $X = 1$ into the global constraint store, whereas $N = \text{truth}(X = 1) \land X = 1$ propagates $N = 1$. Syntactically, reification is written in Oz by paranthesis only. It looks like as if the word truth were ommited. A typical example it the implication $B = 1 \Rightarrow A \in \{2, 4\}$ which can be written as follows.

```
declare B A
[B A]::: 1#27
(B =: 1) ==<: (A :: 2#4)

{Browse [B A]}
/*
 B=1
 */
```

Alternatively, you may also use the special procedure \texttt{FD.impl} which expresses implication for Boolean variables.

```
declare B A
[B A]::: 1#27
(FD.impl (B =: 1) (A :: 2#4))

{Browse [B A]}
/*
 B=1
 */
```

Via reification and arithmetics, one can express disjunctions between arbitrary constraints for which reification is provided. The logical content of the disjunctive propagator or$_i=1^m (C_{ij})$ and the reified constraint $1 \leq \sum_{i=1}^n \sum_{j=1}^{m(i)} \text{truth}(C_{ij})$ are equal. Operationally however, both propagators differ significantly. Reification tests all constraints $C_{ij}$ in isolation for satisfiability with respect to the global constraint store, whereas the corresponding disjunctive propagator tests the whole conjunctions $\land_{j=1}^{m(i)} C_{ij}$.

### 15.8 Summary

- **Finite domain constraints** are a very important class of constraints which is supported by the Oz standard library (see the Oz-reference manual on System Modules). They specify relations between variables denoting members of a finite set of integers. Possible values can be narrowed down by propagation, and there are standard distribution strategies for distinguishing cases if necessary.
• **Finite set constraints** are an important class of constraints which is also supported by the Oz standard library (see the Oz-reference manual on System Modules). Finite set constraints provide propagators for the usual set operations.

• A *disjunction* can be used as a propagator in Oz if it is expressed by an `or` statement. An or-statement can be turned into a distributor by using a finite domain control variables and a finite domain distributor.

## 15.9 Exercises

### 15.9.1 Finite Domains

• **Type hierarchies.** Suppose you have a type hierarchy with types `top, a, b, bot` with an ordering relation satisfying

  \[ \text{top} > \text{a}, \text{top} > \text{b}, \text{a} > \text{bot}, \text{b} > \text{bot} \]

  Encode types in Oz such that the representations of each two types are unifiable. The idea is that a type \( t \) can be represented by a finite domain constraint. For each type \( t \) we fix a fresh variable \( X_t \) and represent \( t \) by the constraint \( X_t \in D(t) \) where \( D(t) \) is the finite set \( D(t) = \{ t' | t' \geq t \} \).

• **Grocery store.** Solve the following problem such that the search tree remains as small as possible: A kid goes into a grocery store and buys four items. The cashier charges $7.11, the kid pays and is about to leave when the cashier calls the kid back, and says “Hold on, I multiplied the four items instead of adding them; I’ll try again; Hah, with adding them the price still comes to $7.11”. What were the prices of the four items?

• **Morphs, his kids, and the television** This is a "Logelei" from the German newspaper Zeit from September 23, 1999 (thanks to Sebastian Pado for sending it to me.) Morphs has a problem. His kids are watching television the whole day over instead of doing their homework for school. But now, he has constructed a new TV and integrated into the network of the television. At this TV, there are 7 switches A, B, C, D, E, F, G each of which an be brought into 4 positions. But only in a single of the 16384 possible combinations, the television can be switched on. This combination can be derived from the respective negations of the following 10 conditions.

  – A must be on 3 but D cannot.
  – A must not be on 4 but F has to be on 1.
  – B must not be on 1. If C is on 4 then F must be on 2.
  – A must be on 1 but not B on 3.
  – G mustn’t be on 1, A has to be on 2 or 4.
  – Neither D on 1 nor B on 2.
  – Neither D on 2 nor G on 3.
  – B must be on 3 but G cannot be on 2.
  – G mustn’t be on 2, and if E is on 2 then F must be on 3.
– G must be 1 but A cannot be 1.

Morphs puts this list of condition on top of TV. A kid that is intelligent enough to find the right combination will also be able to choose good TV shows only

- **Choice.** Write a binary procedure `Choice` by using an or-statement and `FD.distribute`. A call of `{Choice P1 P2}` should behave such as `choice (P1) [] {P2} end`.

- **Placement.** Suppose you are given a rectangle of size `(sx, sy)` and a list of squares with dimensions `(d_1,d_1),…,(d_n,d_n)`. Find a placement of all squares within the rectangle such that no two squares overlap.

Hints:

– You can express the no-overlap constraint for each two small squares by a disjunction of arithmetic formulas. Let `X_i, Y_i` be the coordinates of square `1 ≤ i ≤ n`. The following constraint express that square `j` and `k` do not overlap:

```
X_j + d_j ≤ X_k ∨ X_k + d_k ≤ X_j ∨ Y_j + d_j ≤ Y_k ∨ Y_k + d_k ≤ Y_j
```

This disjunction for `j, k` can either be expressed by a reified FD-constraints or else by an or-propagator with choice variables. Alternatively, you can use a global constraint `FD.distinct2` for expressing all no-overlap constraints for all `1 ≤ j < k ≤ n` at once.
– For each column and row you need a capacity constraint which ensure that not too many squares have to be placed on it.
– It is important in which order you enumerate the finite domain variables: First place larger square and organize placement from left to right.

Here comes a list of example instances of the problem:

```
problems(
  unit(squares: [18 15 14 10 9 8 7 4 1] sx:32 sy:33)
  unit(squares: [3 2 2 1 1] sx:5 sy:4)
  unit(squares: [50 42 37 35 33 29 27 25 24 19 18 17 16 15 11 9 8 7 6 4 2] sx:112 sy:112)
  unit(squares: [9 8 8 7 5 4 4 4 4 3 3 3 2 2 1 1] sx:20 sy:20)
  unit(squares: [6 4 4 4 2 2 2 2] sx:10 sy:10)
)
```

Finally, here is a procedure that allows you to output the placement of squares graphically.

```
declare
proc {ShowTiles SX SY Squares Zoom}
  Color = red
  Off=1
  W = {New Tk.toplevel tkInit}
  {Tk.send wm(resizable W 0 0)}
  {Tk.send wm(title(W "Tiling Problems"))}
  Canvas = {New Tk.canvas tkInit(parent:W
        width:SX*Zoom
        height:SY*Zoom)}

  in
    {Tk.send pack(Canvas)}
    {ForAll Squares
      proc{$ S}
        {Canvas tk(crea(rectangle S.x*Zoom+Off
             S.y*Zoom+1
             (S.x+S.size)*Zoom+Off
             (S.y+S.size)*Zoom-Off
            o(fill:Color))})
      end}
    end
  end
/*

declare
Squares =
  [square(size:3 x:0 y:0) square(size:2 x:3 y:2)
    square(size:2 x:3 y:0) square(size:1 x:2 y:3)
    square(size:1 x:1 y:3) square(size:1 x:0 y:3)]
{ShowTiles 5 4 Squares 40}
*/
Non-deterministic parsing

Complete the following context–free parser by defining `Reduce` using either pre-defined `choice` or your own version of it. `Reduce` takes a list of categories and non-deterministically reduces it to a new list where reduction can result from one of 3 non-exclusive possibilities:

- The first two categories in the list can be reduced to one through application of a binary rule,
- The first category in the list can be reduced to some other category through application of a unary rule,
- The tail of the list can be reduced using one of these 3 possibilities (i.e. reduce the first two elements; reduce the first element; or reduce the tail).

```
dec {fun(MakeCategory P C)
   c(phon:P cat:C)
end

fun {BRules D1 D2}
   {Show [brules D1 D2]}
   {Browse [brules D1 D2]}
   C = case [D1.cat D2.cat]
      of [np vp] then s
       [] [det n] then np
       [] [np pp] then np
       [] [v np] then vp
       [] [vp pp] then vp
       [] [prep np] then pp
       else
         fail unit
   end
 in
   {MakeCategory (Append D1^phon D2^phon) C}
end

fun {URules D}
   {Show [urules D.cat]}
   {Browse [urules D.cat]}
C= case D.cat
   of pn then np
    [] v then vp
    else
      fail unit
   end
 in
   {MakeCategory D.phon C}
```
fun {PhonToCat Phon}
  {Show [Phon phon]}
  {Browse [Phon phon]}
C = case Phon
  of john then pn
  [] runs then v
  [] likes then v
  [] the then det
  [] man then n
  [] 'with' then prep
  [] telescope then n
end

in
  {MakeCategory [Phon] C}
end

% for each word W in the input list, select corresponding lexical entries
% Feed resulting list of lexical entries to Parse1

fun {Parse Phons}
  {Parse1 (Map Phons PhonToCat)}
end

fun {Parse1 Fs}
%  {Browse parse1(Fs)}
  case Fs of [F] then F
  else {Parse1 (Reduce Fs)} end
end

proc {SParse Phon}
  {Explorer.one fun ($) {Parse Phon} end}
end
16.1 Semantic Underspecification

The grammars we’ve used so far contains no semantic information – they do not say how the semantic representation of a phrase can be constructed nor what it is. We will now show how to represent the semantics of a sentence by dominance constraints. Dominance constraints permit to model quantifier scope in an underspecified manner.

16.1.1 Compositionality

Based on the principle of compositionality, it is easy to write a grammar which builds the syntax and the semantics of a phrase in parallel: it suffices to pair with each syntactic a semantic rule specifying how to build up the meaning of the constituent from the meaning of its parts. Computationally, Unification-Based grammars provide an easy way to integrate syntax and semantics. The feature structures representing the categories of linguistic constituent simply contain both a syntax and a semantics attribute the value of which reflect the syntactic and semantic information associated by the grammar with that constituent.

Traditionally, the semantic representation of a phrase is a lambda-term whose model theoretic interpretation models the meaning of that phrase. This is the view that was developed by Montague. However more recently, there has been a worry that such a framework is computationally intractable (and cognitively implausible) in that it enumerates all the possible readings of a constituent. Since ambiguity is a pervasive feature of natural language, this directly leads to a combinatorial explosion: a sentence with n scope bearing elements will have n! readings. The idea to delay the enumeration of reading has lead to the area of semantic underspecification.

16.1.2 Scope Underspecification

For instance, the following sentence contains a lot of scope ambiguities even though its its syntactically unambiguous.

A politician can fool most voters on most issues most of the time, but no politician can fool all voters on every single issue all of the time.
There are $5! \times 5! = 14400$ possible permutations of the scope bearing elements (though not all permutations lead to semantically different readings).

One way to remedy this problem is to under-specify the semantic representation of constituents. This technique has been put to use in particular, for scope ambiguities i.e. sentences such as:

Every yogi has a guru

where either there is one guru for all yogis, or one guru per yogi. These two readings are captured by different scopes for the quantifiers every yogi and a guru so that the two meanings of the above sentence are given by the following FOL formulae:

Reading 1: $\forall x(yogi(x) \rightarrow \exists y(guru(y) \land has(x, y)))$
Reading 2: $\exists y(guru(y) \land \forall x(yogi(x) \rightarrow has(x, y)))$

Note that in both cases, we have the same components namely, the yogi-quantifier, the guru-quantifier and the verb semantics. What varies is the order in which the two quantifiers occur.

16.1.3 Formulas as Trees

Thinking of the FOL formula as a tree, we can use dominance constraints to underspecify the structural relations holding between these three components and thereby get one compact representation for the two readings. To do this, we use a meta-language which is underspecified with respect to dominance and therefore can describe several lambda terms. The intuition is this. Suppose we represent the meaning of the above sentence by the following two trees:

The indices (1) and (2) on the lambda and var nodes represent binding links between quantifier variables (lambda nodes) and predicate variables (var nodes). For instance, the (2) co-indexing indicates that the quantifier variable $x$ in $\lambda P(\exists x(guru(x) \land P(x))$ binds the lambda variable $z$ in the predicate $\lambda z.\lambda u.has(z, u)$. 

16.1.4 Graphs Describing Trees

Assuming the above trees represent the two possible meanings of *Every yogi has a guru*, we can use the following dominance constraint to describe both trees:

```
apply
   ,
   ,
apply lambda(2) apply lambda(1)
   ,
   ,!
   ,!
a guru . every yogi .
```

where the dotted lines indicate dominance and the other lines immediate dominance. If the `lambda(2)` node dominates the `every yogi`-tree, we get the second reading and if the `every yogi`-tree dominates the `a guru`-tree, we get the first reading.

This is the intuition. We now see more precisely what the constraint language is and how a constraint solver can be implemented which given a dominance logic description, yields the set of trees satisfying that description.

16.1.5 Dominance Constraints

The syntax of our dominance logic DL is as follows:

\[
\varphi ::= \varphi \wedge \varphi' \mid X \rhd Y \\
| X: f(X_1, \ldots, X_n) \\
| \lambda(x) = y
\]

where \(X, Y\) are variables and \(\varepsilon\) is an atomic symbol from a given signature.

As for the semantics, a solution to a formula \(\Phi\) consists of a finite tree \(T\), a dominates relation, a lambda relation and an interpretation function \(I\) that maps each variable in \(\Phi\) to a node in \(T\). Further:

- \(x \rhd^* y\) means that in the solution tree \(T\), \(I(x)\) dominates \(I(y)\).
Chapter 16. Constraints in Semantic Underspecification

- \( x: f(x_1...x_n) \) means that \( I(x) \) is labeled by \( f \) and has the nodes \( I(x_1), \ldots, I(x_n) \) as immediate daughters (in that order).
- \( \lambda(x) = y \) means that \( I(x) \) is variable which is bound by the lambda node \( I(y) \).

16.1.6 Syntax-Semantics Interface

Using DL, we can now write a feature-structure grammar which associates a DL description with every grammatical constituent. First the lexicon:

\[
\begin{align*}
\text{Every} & \Rightarrow \text{(syntax: det,} \\
& \text{semantics: (desc: } [X0: \text{apply}(X1,X2) \& X1: \text{apply}(X3,X4) \& \\
& \hspace{1cm} X3: \text{every} \& X4: \text{R} \& X2: \text{lambda} \& X \text{Dom} X0], \\
& \hspace{1cm} \text{root_node: } X0, \\
& \hspace{1cm} \text{binder_node: } X2, \\
& \hspace{1cm} \text{top: } X \\
& \hspace{1cm} \text{restrictor: } R)) \\
\text{yogi} & \Rightarrow \text{(syntax: noun,} \\
& \text{semantics: man)}
\end{align*}
\]

\[
\begin{align*}
\text{loves} & \Rightarrow \text{(syntax: verb} \\
& \text{semantics: (desc: } [X0: \text{apply}(X1,X2) \& X1: \text{apply}(X3,X4) \& \\
& \hspace{1cm} X3: \text{has} \& X4: \text{var} \& X2: \text{var}] \\
& \hspace{1cm} \text{root_node: } X0 \\
& \hspace{1cm} \text{object_node: } X4 \\
& \hspace{1cm} \text{subject_node: } X2))
\end{align*}
\]

The feature path \text{semantics:desc} is assigned the DL formula describing the semantic contribution of an item. The features \text{root_node}, \text{binder_node}, \text{top}, \text{restrictor}, \text{object_node}, \text{subject_node} are used for co-indexing by the grammar rules as follows:

\[
\begin{align*}
\% \text{ NP } & \rightarrow \text{ Det Noun} \\
(\text{syntax:np} \\
\text{semantics: (desc :D} \\
\hspace{1cm} \text{root_node :R} \\
\hspace{1cm} \text{binder_node:B} \\
\hspace{1cm} \text{top :T}})) \Rightarrow \\
\rightarrow \\
(\text{syntax:det} \\
\text{semantics: (desc: D root_node: R binder_node: B top: T restrictor: R)}) \\
(\text{syntax:noun semantics: R})
\end{align*}
\]

\[
\begin{align*}
\% \text{ VP } & \rightarrow \text{ Verb NP} \\
(\text{syntax:vp} \\
\text{semantics: (desc: {Append D1 {Append D2 [B dom R \ B lambda O]}})}
\end{align*}
\]
16.2 Complexity of Dominance Constraints

In this chapter, we show how to solve dominance constraints by constraint programming with sets. While we won’t say anything about the details, the techniques used here can be used as a basis to build more underspecified processing mechanisms for dominance constraints. For instance, the encoding of nodes presented below lends itself very well to capturing the interaction of scope and anaphora as in *Every man loves a woman. Her name is Mary.* In the sentence, the anaphoric reference excludes one reading of the first sentence; we can make this inference purely with propagation.
16.2.1 Dominance Constraints

We will consider the following language of tree descriptions based on dominance constraints:

\[ \varphi ::= \varphi \land \varphi' \]

\[ X = Y \]

\[ X \neq Y \]

\[ X \triangleleft^* Y \]

\[ X \negtriangleleft^* Y \]

\[ X: (Y_1, \ldots, Y_n) \]

This language is a variant of the dominance constraints defined in the second lecture. The differences are as follows:

- \( X = Y \) expresses that \( X \) and \( Y \) must denote the same node. It’s an abbreviation of \( X \triangleleft^* Y \land Y \triangleleft^* X \).

- \( X \negtriangleleft^* Y \) expresses that \( X \) must not dominate \( Y \). This couldn’t be expressed in the original language.

- The new language doesn’t contain lambda binding constraints. This is for simplicity of presentation; it’s not difficult to add binding constraints to the implementation. Note that we can now speak just about trees, instead of lambda structures, as the models of dominance constraints.

- Labeling constraints have been replaced by ‘daughterhood’ constraints \( X: (Y_1, \ldots, Y_n) \); the difference is that daughterhood constraints don’t specify the label of \( X \). This, too, is for simplicity, and labels could be (and have been) added easily to the implementation.

16.2.2 Constraints as Graphs

We will consider a dominance constraint as a graph, its constraint graph. Each node of the constraint graph corresponds to a variable in the constraint. The graph can be seen as a variable centered representation of the constraint. All information distributed over several occurrences of a variable in a dominance constraint is centered around a single node of the constraint graph.

Note also the we have already seen graphs in the motivating examples above. All these graphs can be seen as the constraint graph of some dominance constraint.

16.2.3 Partitioning Trees

When regarded from a specific node, a tree is divided into 5 regions: (1) the node itself, (2) the nodes above, (3) the nodes below, (4) the nodes to the left, and (5) the nodes to the right.
We will aggregate the set of nodes to the left and to the right, and call the result the *side set*. A similar treatment can trivially be developed that retains the distinction; such a treatment would support precedence constraints.

### 16.2.4 A Generate and Test Algorithm

In the graph metaphor, solving a dominance constraint means to configure its the nodes of the constraint graph into a tree, such that all required dominance relations hold.

There exists a simple but naive ‘generate and test’ algorithm doing this. The idea is that for any two nodes $N_1$ and $N_2$ in the graph there exists only a 4 relative positions into which they can be configured in the final tree.

1. $N_1 = N_2$, they become equal
2. $N_1 <^+ N_2$, $N_1$ strictly dominates $N_2$
3. $N_2 <^+ N_1$, $N_2$ strictly dominates $N_1$
4. $N_1 \perp N_2$, $N_1$ is to the side of $N_2$ (i.e. none of the above).

We can thus decide the satisfiability of a dominance constraint as follows: First guess one of the above relationships for each two nodes in a graph. Then test, whether the graph satisfies all relationships required by the constraint and whether the graph augmented with the guessed relationships becomes tree-like. If one guess is consistent then the constraint graph is satisfiable, otherwise not.

### 16.2.5 Complexity of Dominance Constraints

As we have seen above, we can solve a dominance constraint by a ‘generate and test’ algorithm. There is an exponential number of combinations to generated each of which can be tested in polynomial time.

If we would have an oracle guessing the right combination then we could solve the problem in polynomial time (but of course there is no such oracle). This view show that the satisfiability of a dominance constraint can be tested in non-deterministic polynomial time algorithm. In terms of complexity theory, one says that the problem is in NP.
But the situation is even worse than one might hope at first sight. In fact satisfiability of dominance constraints is NP-complete. Thus, we cannot expect any deterministic polynomial algorithm to exist without winning the Turing award.

So do we have to give up at this point?

No, we just have to give up ‘generate and test’. If we cannot solve dominance constraints efficiently in all cases then we can still hope for an algorithm that is efficient for the applications to semantic underspecification.

16.3 Solving Dominance Constraints Efficiently

In this section, we provide an implementation of the dominance constraint solver based on finite set constraints in Oz. Our solver behave efficiently on the examples from scope underspecification we have seen so far.

16.3.1 Sets of Nodes

Suppose, we are given a dominance constraint and one of its solutions. With each node of the constraint graph \( N \) (i.e. each of its variables), we associate 4 mutually exclusive 4 finite sets of nodes:

1. \( \text{N.eq} \), the set of graph nodes of whose interpretation is equal \( N \),
2. \( \text{N.up} \), the set of graph nodes whose interpretations are strictly above \( N \),
3. \( \text{N.down} \), the set of graph nodes whose interpretations are strictly below \( N \),
4. \( \text{N.side} \), the set of graph nodes whose interpretations are to the side of \( N \).

The whole idea of our approach resides here: we encode a dominance constraints into set constraints such that each node in the constraint graph is modelled by the 4 set variables that we constrain to the sets above (in each solution).

The first constraint we require is the following: For all nodes \( N \), the 4 sets for \( N \) are disjoint and form a partition of the set \( V \) of all nodes of the constraint graph:

\[
V = \text{N.eq} \cup \text{N.up} \cup \text{N.down} \cup \text{N.side}
\]

16.3.2 Encoding Nodes as Records

In our implementation, we associate with each node a unique index which is an integer number \( I \) and a record with features

\[
\text{eq down up side eqdown equip daughters}
\]

The entries of the record are set variables \( \text{N.eq}, \text{N.down}, \text{N.up} \) and \( \text{N.side} \) whose meaning was explained above. Furthermore, there are auxiliary sets variables \( \text{N.eqDown} \) and \( \text{N.eqUp} \) which satisfy

\[
\text{N.eqDown} = \text{N.eq} \cup \text{N.down} \\
\text{N.eqUp} = \text{N.eq} \cup \text{N.up}
\]
Finally, the value of $N_{\text{daughters}}$ is the list of records for the daughter nodes. We require the following membership constraint in order to relate the index $i$ with the record $N$ of a node:

$$I \in N_{eq}$$

The functional procedure MakeNode below inputs the index $I$ of a node and a tuple $VDom=[1\#N]$ where $N$ is the number of all nodes in the constraint graph. It returns the record for the node and imposes the above given constraints on its set variables.

199a (DC: make node 199a)

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

If two nodes are forced to denote the same node then the corresponding records are unified (but not the corresponding indices). Thus all features of the unified records are unified as well. Hence, the corresponding records for the daughters of both nodes are also unified. This shows how record unification is used in the dominance constraint solver.

16.3.3 Translation to Set Constraints

If $N_1$ dominates $N_2$, then everything that is (weakly) below $N_2$ must be (weakly) below $N_1$, everything that is (weakly) above $N_1$ must be (weakly) above $N_2$, and everything that is beside $N_1$ is also beside $N_2$. Note however that there can be nodes beside $N_2$ that are below $N_1$.

199b (DC: dominates 199b)

```proc (Dominates N1 N2)
    {FS.subset N2.eqdown N1.eqdown}
    {FS.subset N1.equp N2.equp }
    {FS.subset N1.side N2.side }
end```
The equality constraint is simply implemented by unification:

\[ \text{(DC: equal)} \]
\[
\text{proc } \{ \text{Equal } N1 \ N2 \} \ N1=N2 \text{ end}
\]

The disequality constraint states that the Eq sets of N1 and N2 must be disjoint:

\[ \text{(DC: not equal)} \]
\[
\text{proc } \{ \text{NotEqual } N1 \ N2 \}
\{ \text{FS.disjoint } N1.\text{eq } N2.\text{eq} \}
\text{ end}
\]

N1 strictly dominates N2 iff it dominates N2 and is not equal to N2:

\[ \text{(DC: strictly dominates)} \]
\[
\text{proc } \{ \text{StrictlyDominates } N1 \ N2 \}
\{ \text{Dominates } N1 \ N2 \}
\{ \text{NotEqual } N1 \ N2 \}
\text{ end}
\]

If N1 is to the side of N2 (and reciprocally), then N1 and everything below it is to the side of N2 (and resp.):

\[ \text{(DC: side)} \]
\[
\text{proc } \{ \text{Side } N1 \ N2 \}
\{ \text{FS.subset } N1.\text{eqdown } N2.\text{side} \}
\{ \text{FS.subset } N2.\text{eqdown } N1.\text{side} \}
\text{ end}
\]

Finally, here is the constraint that deals with immediate dominance by specifying explicitly the daughters of a node \( N \) as a list \( \text{Nodes} \) of nodes. The set of nodes that are weakly below each of the daughters form a partition of the set of nodes that are strictly below the mother. Furthermore, the set of nodes that are strictly above each daughter is precisely the set of nodes that are weakly above the mother.

\[ \text{(DC: daughters)} \]
\[
\text{proc } \{ \text{Daughters } N \ L \}
N.\text{daughters } = L
\{ \text{FS.partition } \{ \text{Map } L \text{ fun } \{ \$ D \} \ D.\text{eqdown end} \} \ N.\text{down} \}
\text{ for } D \text{ in } L \text{ do } D.\text{up}=N.\text{equp end}
\text{ end}
\]

### 16.3.4 Treeness Condition

We say that any 2 nodes \( N_i \) and \( N_j \) must satisfy the *treeness condition* expressed as the following disjunction:

\[
N_i = N_j \lor N_i \triangleleft N_j \lor N_i \triangleleft N_j \lor N_i \triangle N_j
\]

The treeness condition that must hold between \( N_i \) and \( N_j \) is realized by four concurrent disjunctions and is controlled by choice variable \( c_{ij} \). The latter is a finite domain variable taking a value in \([1..4]\).
16.3. Solving Dominance Constraints Efficiently

The \texttt{thread ... end} statements in the code fragment cause the computation to create four new concurrent threads, one for each choice variable. This is necessary because the \texttt{or} statements within the new threads block until only one of their guards can be satisfiable, and we don’t want this to block our entire computation.

### 16.3.5 Encoding Dominance Constraints

We encode a dominance constraint as a functional procedure which inputs a list \([N_1 \ N_2 \ ... \ N_k]\) of nodes, one for each variable in the input constraint (the graph). Recall that each node comes as a record of set variables. The functional procedure encoding a dominance constraint imposes the right set constraints on these set variables, i.e. those which encode the dominance constraint.

Consider the dominance constraint which is typical for a scope ambiguity with two quantifiers.

\[
X_1 : (X_2) \land X_2 \lt^* X_5 \land X_3 : (X_4) \land X_4 \lt^* X_5
\]

We are interested in all solutions of this constraint where no variables are identified. This reflects that quantifiers should not be identified. It is slightly stronger than saying that nodes with distinct labels should not be identified.

\[
X_1 \neq X_2 \land X_1 \neq X_3 \land X_1 \neq X_4 \land X_1 \neq X_5 \land \\
X_2 \neq X_3 \land X_2 \neq X_4 \land X_2 \neq X_5 \land \\
X_3 \neq X_4 \land X_3 \neq X_5 \land \\
X_4 \neq X_5
\]

Using the \texttt{DC} module, it would be expressed as a record which contains the number of variables and a procedure which inputs a list of nodes and creates set constraints for these nodes and the dominance constraint.

```
201b \texttt{(DomConExample \[\texttt{201b}\])} \equiv

\begin{verbatim}
local proc {DomCon [N1 N2 N3 N4 N5]}
  {DC.daughters N1 [N2]}
  {DC.dominates N2 N5}
  {DC.daughters N3 [N4]}
  {DC.dominates N4 N5}
  {ForAll [N1#N2 N1#N3 N1#N4 N1#N5 N2#N3 N2#N4 N2#N5 N3#N4 N3#N5 N4#N5]}
\end{verbatim}
```
proc($ N\#M)$
   {DC.notEqual N M}
   end
end

in
DomConExample = ‘unit’(domCon:DomCon
   vars:5)
end

16.3.6 The Solver as a Module

The dominance constraint solver is provided by functor DC.ozf which is made available by package dominance.pkg.

202a (DC.ozf 202a)≡
   functor
   import
   FS FD Space
   export
   MakePredicate
   Daughters
   Dominates
   NotEqual
   define
   (DC: solver procedures 202b)
   end

202b (DC: solver procedures 202b)≡
   (DC: daughters 200e)
   (DC: dominates 199b)
   (DC: not equal 200b)
   local
   (DC: equal 200a)
   (DC: strictly dominates 200c)
   (DC: side 200d)
   (DC: make node 199a)
   in
   (DC: make predicate 203)
   end

202c (DC: dominance constraint solver as record 202c)≡
   local
   (DC: solver procedures 202)
   in
   DC=dom(makePredicate:MakePredicate
daughters:Daughters
dominates:Dominates
notEqual:NotEqual)
   end

1dominance.pkg
In particular, the record DC exports the procedure MakePredicate which turns a dominance constraint into a predicate appropriate as input to encapsulated search as provided by e.g. Explorer.all or Search.all. For example, we could now use the Explorer\[\text{all}\] to search for all possible (constructive) models of DomConExample:

203a \[
\begin{align*}
\text{declare} & \\
& \{\text{DC} = \{\text{Link } [\text{'x-ozlib://oz-kurs/DC.ozf'] }\} \\
& \{\text{DomConExample}\} \}
\end{align*}
\]

in

\[
\{\text{Explorer.all } \{\text{DC.makePredicate DomConExample}\}\}
\]

### 16.3.7 Solution Predicate

The functional procedure MakePredicate inputs a dominance constraint and computes a solution predicate for it which in turn can be explored by encapsulated search, e.g. Search._all or Explorer._all.

The input dominance constraint - the graph - comes as a record which contains the number of nodes N, i.e. the number of variables in the input constraint, and a procedure P which represents the dominance constraint itself. Recall that such a procedure takes a list of node representations of length N and imposes the set constraints encoding the dominance constraint.

A search predicate always has the same form: it is a unary predicate whose argument denotes a solution. First it posts all constraints on the solution, then it specifies a search/distribution strategy:

203b \[
\begin{align*}
\text{fun } & \{\text{MakePredicate } \text{'unit'} \{\text{domCon:DomCon vars:N}\}\} \\
\text{proc } & \{\$ \text{ Nodes}\} \\
& \{\text{DC:create nodes}\} \\
& \{\text{DC:translation to set constraints}\} \\
& \{\text{DC:impose tree}\} \\
\end{align*}
\]

The solution Nodes must be a list of N nodes. Each variable is represented by a distinct integer between 1 and N. Thus sets of variables can be represented by sets of integers. (We store the specification of the finite domain from 1 to N in the variable VDom.) For each variable, MakeNode creates a term representing the node that is the interpretation of this variable.

203c \[
\begin{align*}
\text{VDom} & = [1\#N] \\
\{\text{List.make } N \text{ Nodes}\} & \% \text{ constrains Nodes to a list} \\
& \% \{_, \ldots, _\} \text{ of length N} \\
\text{for } & \text{ Node in Nodes } I \text{ in } 1..N \text{ do } \{\text{MakeNode } I \text{ VDom Node}\} \text{ end}
\end{align*}
\]
Then we constrain these nodes using the procedure \texttt{DomCon} that implements the dominance constraint. After this we execute \texttt{choice skip end} whose only effect is to wait for stability; i.e. until constraint propagation has inferred as much as it could. Typically the dominance constraint \texttt{DomCon} provides very strong constraints and it is a good idea to impose them first and wait until they have achieved full effect before going on with the quadratic number of expensive treeness constraints.

\small

\begin{verbatim}
204a  (DC: translation to set constraints) ≡
      {DomCon Nodes}
      % waits for stability
      {Space.waitStable}
\end{verbatim}

Now we impose the treeness constraint between every pair of nodes \(\texttt{Ni}\) and \(\texttt{Nj}\). For every such pair we impose a choice which is controlled by its own choice variables with domain \([1..4]\). We collect the quadratic number of choice variables within the list \texttt{ChoiceVariables}.

\small

\begin{verbatim}
204b  (DC: impose treeness) ≡
      ChoiceVariables =
      {List.foldRTail Nodes
        fun {$ Ni|Ns Cs}
        {List.foldR Ns
          fun {$ Nj Cs}
            (DC: treeness condition between Ni and Nj)
            C|Cs
          end Cs}
        end nil}
\end{verbatim}

Finally, we specify the distribution strategy: here we use \texttt{First Fail} on the choice variables. Each choice variable is a finite domain variable in \([1..4]\). First fail is a strategy which attempts to minimize the branching factor in the search tree: it picks a (non-determined) variable with the minimum number of remaining possible assignments.

\small

\begin{verbatim}
204c  (DC: distribute) ≡
      {FD.distribute ff ChoiceVariables}
\end{verbatim}

\section*{16.3.8 Graphical Output with DaVinci}

It is often useful to output a picture of a graph rather than the list of its nodes and edges. For instance, one might want to draw constraint graphs or its solution trees. A quite simple way to do so is to use a professional tree drawing tool such as DaVinci which was developed at the University of Bremen in Germany.

Of course, you have to install DaVinci\footnote{http://www.tzi.de/~davinci} on your local machine if you want to run an applet based on DaVinci. The only problem might be that DaVinci does not run on your platform. For instance, DaVinci is not provided for Windows.

There also exists a simple interface from Oz to DaVinci. This interface is provided by the authors in terms of two Mozart-functors which are made available at the following URL:
The version number concerns the Pickle version of functors used by your Oz system. An example for the usage of the Oz-DaVinci-Interface can be found at:

http://www.ps.uni-sb.de/~niehren/DaVinci/example.oz

If you want to use the interface then you have to define a graph and the layout parameters for the nodes and edges of the graph. As a typical example we consider the following graph - the graph of the dominance constraints for 'Every man loves a woman'.

Example Graph

```
Graph = [node(id:1
   edges:[edge(to:2)]
   label:label('OBJECT':"X1: forall # men")

node(id:2
   edges:[edge(kind:dom to:5)]
   label:label('OBJECT':"X2")

node(id:3
   edges:[edge(to:4)]
   label:label('OBJECT':"X3: exists # woman")

node(id:4
   edges:[edge(kind:dom to:5)]
   label:label('OBJECT':"X4")

node(id:5
   edges:nil
   label:label('OBJECT':"X5: # loves")
]
```

The above graph definition already uses layout parameters to be defined. These parameters are used by the DaVinci interface in order to determine the attributes of nodes and edges to be drawn by DaVinci. We define the layout parameters as follows:

Layout Parameter

```
LayoutParameter =
unit(edge:unit (default:edge(to:default
   'EDGECOLOR':black
   'EDGEPATTERN':solid
   '_DIR':none
   'HEAD':arrow)
   dom:edge('EDGECOLOR':blue
       'EDGEPATTERN':dotted)
   void:edge('EDGEPATTERN':none)
)
label: unit (default:label('COLOR':white
   '_GO':text
   'OBJECT':default)
```
We are now in the position to draw the graph given above. First, you have to specify the path at which your executable DaVinci is situated.

\[\text{(DaVinci Path)}\]
\[
\text{DaVinciPath} = "/\text{export/global/ps/soft/daVinci\_V2.1/linux-ia64/daVinci}\"
\]

Next, you have to specify the pickle version that your Mozart system requires.

\[\text{(Pickle Version)}\]
\[
\text{Version} = \text{’Version.3.2’}
\]

Next, you have to combine the above graph and layout parameter with the DaVinci Interface.

\[\text{(Draw the Example Graph)}\]
\[
\text{URL} = \text{’http://www.ps.uni-sb.de/~niehren/DaVinci’}
\]
\[
\text{GraphOzf} = \text{’/\#Version\#/graph.ozf’}
\]
\[
\text{PipeOzf} = \text{’/\#Version\#/pipe.ozf’}
\]
\[
\text{[Mod1 Mod2]} = \{\text{Module.link [URL#GraphOzf URL#PipeOzf]}\}
\]
\[
\text{MakeDaVinciGraph} = \{\text{Mod1.davincigraph LayoutParameter}\}
\]
\[
\text{MakeDaVinciEdges} = \{\text{Mod1.davinciEdges LayoutParameter}\}
\]
\[
\text{NewDaVinci} = \{\text{Mod2.makeNewDaVinci DaVinciPath}\}
\]
16.3. Solving Dominance Constraints Efficiently

DVG = {MakeDaVinciGraph Graph} {Browse DVG}
DaVinci = {NewDaVinci}
{DaVinci.graph DVG}

16.3.9 Exercises

1. **Better Propagation:** Modify the dominance constraint solver such that more information is propagated. The idea is to use another treeness condition based on propagators for the following disjunctions:

   \[ N_i = N_j \lor N_i \neq N_j \]
   \[ N_i \triangleleft^+ N_j \lor N_i \triangleleft^- N_j \]
   \[ N_j \triangleleft^+ N_i \lor N_j \triangleleft^- N_i \]
   \[ N_j \perp N_i \lor N_i \perp N_j \]

2. **Symbolic Input:** the solver’s interface is not very user friendly: it takes some effort to convert a dominance constraint problem into a procedure that posts these constraints on the node representations used in the solver. A better idea would be to accept a symbolic description of a dominance constraint and automatically convert it into solver constraints. We would like a dominance constraint to be specified as a list (representing a conjunction) of ground terms; where each ground term \( C \) follows the abstract syntax below:

   \[ C ::= \text{label}(x \ f(y_1 \ldots \ y_n)) \mid \text{dom}(x \ y) \mid \text{ndom}(x \ y) \mid \text{neq}(x \ y) \]

   where \( x, y \) and \( y_i \) are atoms representing variables of the dominance constraint. \( \text{dom}(x \ y) \) indicates that \( x \) dominates \( y \), \( \text{ndom}(x \ y) \) is its negation, and \( \text{neq}(x \ y) \) represents non-equality.

   You should modify the solver, to accept such a list as argument and return a search predicate to solve the problem. A solution should be a record whose features are the atoms representing variables of the dominance constraint problem, and whose values are the corresponding node representations in the solver.

3. **Graphical Output:** Write a dominance constraint solver which output the solutions it computes graphically. The idea is to combine the Oz-DaVinci-Interface with the DC.ozf functor.

4. Configure the Explorer such that when you click at a solution node then it displays the solution tree with DaVinci.

5. Evaluate the dominance solver from the script and your own solver with better propagation and graphical output (previous exercises). A list of test constraints is given below. How much time is required, how many solutions are there, and how many failure nodes occur?

   - Here come the typical genitive construction where one of the alternations of all quantifiers is not possible.
   
   Two researcher of every company work on a program
   
   An underspecified description of the semantics of this sentence could be given by the following dominance constraint:
Chapter 16. Constraints in Semantic Underspecification

The graph of this constraint looks as follows:

```
[dom(x u) label(u 'of') label(y two_resercher) label(z a_program) label(x1 every_company(x))
  label(y two_resercher) label(z a_program) label(x1 every_company(x))
  label(y two_resercher) label(z a_program) label(x1 every_company(x))
  label(y two_resercher) label(z a_program) label(x1 every_company(x))]
```

A similar but larger constraint to the above can be produced when consid-
ering a sentence with 5 Quanitifies and two genitive constructions:

Some researcher of every departement of most companies see
most samples of every product.

An underspecified description of the semantics of this sentence could be
given by the following dominance constraint:

```
[dom(y5 u4) label(u4 'of') dom(y5 u4)
  label(x4 a_department) dom(y4 u3) dom(z4 u4) label(u3 'of')
  dom(y3 u2) label(x3 some_researcher) dom(y3 u2)
  label(x2 most_samples) dom(y2 u1) dom(z2 u2) label(x1 see)
  dom(x1 every_product) dom(z1 u1)]
```

The graph of this constraint looks as follows:

```
[dom(y1 a_professor) dom(x1 x3) label(x3 'of')
  dom(y2 every_mother) dom(x2 x13) label(x13 say)
  dom(x4 x5) label(x5 'not') dom(x9) label(x12 see)
  dom(x4 x6) label(x6 every_sample(x10 x11)) dom(x10 x3) dom(x11 x12)
  dom(x4 x8) label(x8 most_researcher(x20 x15)) dom(x20 x14) dom(x15 x16)
  label(x14 'of') dom(y3 a_company) dom(x16 x14)]
```

The following is a variant of a famous sentence of Hobbs and Shieber.

Every mother says most researchers of a company do not see
every sample of a professor.

An underspecified description of the semantics of this sentence could be
given by the following dominance constraint (given in Oz notation):

```
[dom(y1 a_professor) dom(x1 x3) label(x3 'of')
  dom(y2 every_mother) dom(x2 x13) label(x13 say)
  dom(x4 x5) label(x5 'not') dom(x9) label(x12 see)
  dom(x4 x6) label(x6 every_sample(x10 x11)) dom(x10 x3) dom(x11 x12)
  dom(x4 x8) label(x8 most_researcher(x20 x15)) dom(x20 x14) dom(x15 x16)
  label(x14 'of') dom(y3 a_company) dom(x16 x14)]
```

The graph of this constraint looks as follows:
16.3. Solving Dominance Constraints Efficiently

- In which solver and why can the following constraint be solved by pure propagation?

\[
\text{label}(y \ f(y_1\ y_2)\ \text{dom}(y_1\ u)\ \text{dom}(y_2\ v)\ \text{dom}(x\ u)\ \text{dom}(x\ v))
\]

The graph of this constraint looks as follows:

![Graph of the constraint]

16.3.10 Complete Solver

Test the dominance constraint solver:

```oz
declare
[DC]={Link ['x-ozlib://oz-kurs/DC.ozf']}
local
proc (DomCon [N1 N2 N3 N4 N5])
{DC.daughters N1 [N2]}
{DC.dominates N2 N5}
{DC.daughters N3 [N4]}
{DC.dominates N4 N5}
{ForAll [N1#N2 N1#N3 N1#N4 N1#N5]
```

```
The code of the dominance constraint solver:

functor
import
    FS FD Space
export
    MakePredicate
    Daughters
    Dominates
    NotEqual
define
    proc {Daughters N L}
    N.daughters = L
    (FS.partition {Map L fun {$ D} D.eqdown end} N.down)
    for D in L do D.up=N.up end
end
    proc {Dominates N1 N2}
    (FS.subset N2.eqdown N1.eqdown)
    (FS.subset N1.equp N2.equp )
    (FS.subset N1.side N2.side )
end
    proc {NotEqual N1 N2}
    (FS.disjoint N1.eq N2.eq)
end
local
    proc {Equal N1 N2} N1=N2 end
    proc {StrictlyDominates N1 N2}
    {Dominates N1 N2}
    {NotEqual N1 N2}
end
    proc {Side N1 N2}
    {FS.subset N1.eqdown N2.side}
    {FS.subset N2.eqdown N1.side}
end
fun {MakeNode I VDom}
Eq   = {FS.var.upperBound VDom}
Down = {FS.var.upperBound VDom}
Up   = {FS.var.upperBound VDom}
Side = {FS.var.upperBound VDom}
EqDown = {FS.union Eq Down}
EqUp  = {FS.union Eq Up}

in
{FS.partition [Eq Down Up Side] {FS.value.make VDom}}
{FS.include I Eq}
node(
eq    : Eq
down  : Down
up    : Up
side  : Side
eqdown : EqDown
equp   : EqUp
daughters : _)
end

in
fun {MakePredicate 'unit'(domCon:DomCon vars:N)}
  proc {$ Nodes}
    VDom = [1#N]
    {List.make N Nodes} % constrains Nodes to a list
    % [__ ... __] of length N
    for Node in Nodes I in 1..N do {MakeNode I VDom Node} end
    {DomCon Nodes}
    % waits for stability
    {Space.waitStable}
    ChoiceVariables =
    {List.foldRTail Nodes
      fun {$ Ni|Ns Cs}
        {List.foldR Ns
          fun {$ Nj Cs}
            C in C::1#4
              thread
                or C = 1 {Equal Ni Nj}
                [] C = 2 {StrictlyDominates Ni Nj}
                [] C = 3 {StrictlyDominates Nj Ni}
                [] C = 4 {Side Nj Ni}
              end
            end
          end Cs}
        end nil}
      in
        {FD.distribute ff ChoiceVariables}
      end
    end
end
Draw the example graph:

%% this file contains an example for the Oz-DaVinci-Interface

%% the following constants have to be adapted.

\begin{verbatim}
\textbf{declare}

%% The value of DaVinciPath has to be adapted to be
%% the path where to find the executable DaVinci-binaries

DaVinciPath= "/export/global/ps/soft/daVinci_V2.1/linux-i486/daVinci"
%% The value of Version determines the pickle version
%% required by your Oz System. Mozart 1.0.1 requires
%% version 1.5 and Mozart 1.1.0 needs version 2.0.

Version = 'Version.3.2'

%%
\end{verbatim}

%% the following constants remain unchanged.

\begin{verbatim}
URL = 'http://www.ps.uni-sb.de/~niehren/DaVinci'
GraphOzf = '/*\#Version#/graph.ozf'
PipeOzf = '/*\#Version#/pipe.ozf'

[Mod1 Mod2] = {Module.link [URL#GraphOzf URL#PipeOzf]}

MakeDaVinciGraph = {Mod1.daVinciGraph LayoutParameter}
MakeDaVinciEdges = {Mod1.daVinciEdges LayoutParameter}
NewDaVinci = {Mod2.makeNewDaVinci DaVinciPath}

LayoutParameter =
\textbf{unit}\{edge:unit\}(default:edge\{to:default

\hspace{1cm} 'EDGECOLOR':black
\hspace{1cm} 'EDGEPATTERN':solid
\hspace{1cm} '_DIR':none
\hspace{1cm} 'HEAD':arrow)

\hspace{1cm} dom:edge\{('EDGECOLOR':blue
\hspace{1cm} 'EDGEPATTERN':dotted)
\hspace{1cm} void:edge\{('EDGEPATTERN':none)
\}

\hspace{1cm} label: \textbf{unit}\{default:label\{('COLOR':white
\hspace{1cm} '_GO':text
\hspace{1cm} 'OBJECT':default)
\}
\end{verbatim}
16.4 Summary

- Concurrent Constraint Programming allows a very intuitive implementation of a solver for dominance constraints.

- Every variable in the dominance constraints corresponds to a node in the constraints graph. Every node of the constraint graph is associated with four sets of nodes: the sets of graph nodes that denote tree nodes equal, strictly above, strictly below, and to the side of it.

- *Finite set constraints* can be used to axiomatize the problem; they can be taken over in Mozart with only syntactic variations.

- The dominance constraint solver based on finite set constraints has been integrated into the *CHORUS demo system* and runs efficiently on dominance constraints from underspecified semantics.
Part V

Dependency Parsing with Finite Set Constraints
General Ideas

17.1 What are we Doing?

We present a parser for German which deals with its free word order. The word order of German is not really free but at least freer than in English. Our approach to parsing should in principle also apply to other languages with free word order such as Dutch and ???.

Our parser is based on the framework of dependency grammar. This framework is attractive for specifying the word order in languages with “free” word order. The reason is that dependency grammar leaves word order unspecified a priori and supports constraints for its specification a posteriori. We argue that word order constraints provide natural specifications of word order in German (or other free word order languages). In particular, we believe that word order specifications by constraints are more natural than by transformations.

There is very little agreement about what might constitute a dependency grammar or the essence of dependency parsing. It might also be unclear whether dependency grammar/parsing differs much from other kinds of grammar/parsing? We will mainly answer the question what dependency parsing is about and speak less about the grammar itself. For parsing, it turns out that we can reuse many ideas and programming techniques which we have already seen for other parsers. Nevertheless, we will also introduce a bunch of new ideas, in particular concerning constraints for word order and constraints for constituent structure.

We will show how to express word order constraints for German by finite set constraints in Oz. This leads to an efficient implementation of a dependency parser within the paradigm of constraint programming. The parser we present has been developed by Denys Duchier [1998].

17.2 Standard Principles

The dependency parser that we are going to present preserves many characteristics of parsers for unification based grammars.

- Unification and strong lexicalization are adapted from unification based grammar.
- The constituent structure of a sentence is represented by a tree. The trees we are using are called dependency structures. The idea is that a dependency structure is like a phrase structure without word order restrictions.
- Constituent structures are reconstructed in a head driven manner.

17.3 Constraining the Word Order

There are a bunch of new ideas which are mostly concerned with free word order and its consequences.

- Word order is left completely unspecified a priori and restricted only when needed. This can be achieved by representing word order orthogonally to constituent structure.
- The constituent structure of a sentence is modeled by a dependency structures, and its word order by an additional ordering relation (on the nodes of its dependency structure).
- The word ordering of a sentence is specified by word order constraints. The dependency structure of a sentence is expressed by role dependency constraints. Since both structure are interlocked, both kinds of constraints are also interlocked.
- Word order constraints can be encoded by finite set constraints in Oz and then be resolved by propagation and distribution.

17.4 Is a Chart still Appropriate?

Free word order may render parsing more complex: Combining adjacent constituents according to some grammar rules is no longer sufficient. Instead, also non-adjacent constituents have to be considered for combination.

One way out of the problem might be to use a chart which is generalized in that edges between non-adjacent constituents are allowed. Parsing then amounts to generating all pairs of constituents dynamically and testing whether they can be combined according to some grammar rule.

It seems however, that this idea is infeasible since words could be freely combined in arbitrary order. Given a sentence with n words, there might be \( \text{fac}(n) \) orderings in which to combine its words. Hence, a generalized chart seems not fully appropriate for parsing a free word order language.

In fact, we will not use any chart. It will be replaced by new forms of constraints.

17.5 Constraining the Constituent Structure

As argued above, it might be a good idea to avoid “generate and test” as supported by a chart parser. Since we are concerned with constraint programming, the idea would be to use “propagate and distribute” instead.
Following this idea, we should restrict the form constituents (previously stored as edges of a chart) by constraints rather than generated them by grammar rules. In fact, the dependency parser to be presented does exactly this: Traditional grammar rules are replaced completely in favor of two kinds of constraints:

- role dependency constraints
- word order constraints.
18.1 Topological Fields

There is a long tradition in German linguistics to describe German word order in the theory of topological fields [Herling 1821, Erdmann 1886, Bierwisch 1963, Höhle 1995]. The word order constraints that we are going to present includes a formalization of parts of this theory.

Classically, three topological types of German sentences are distinguished depending on the position of the main verb: verb-two type, verb-one type, and verb-last type.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>verb-two:</td>
<td>(denn) damit hat keiner gerechnet</td>
</tr>
<tr>
<td></td>
<td>der Professor hatte dem Studenten vor einigen Monaten versprochen</td>
</tr>
<tr>
<td></td>
<td>das Gutachten bald zu schreiben</td>
</tr>
<tr>
<td>verb-one:</td>
<td>regnet es?</td>
</tr>
<tr>
<td></td>
<td>(aber) würde jemand den Hund füttern morgen abend</td>
</tr>
<tr>
<td>verb-last:</td>
<td>(daß) es regnet</td>
</tr>
<tr>
<td></td>
<td>das Gutachten bald zu schreiben</td>
</tr>
</tbody>
</table>

The theory of topological fields predicts that the words a sentences can be partitioned into several fields depending on the type of sentence (Vorfeld, Mittelfeld, etc) such that certain field constraints are satisfied.

In this script, we will only consider verb-two sentences, i.e. no relative sentences and no questions. This topological type is described by the following scheme:

<table>
<thead>
<tr>
<th>KOORD</th>
<th>Vorfeld</th>
<th>FINIT</th>
<th>Mittelfeld</th>
<th>Verbkomplex</th>
<th>Nachfeld</th>
</tr>
</thead>
<tbody>
<tr>
<td>denn</td>
<td>damit</td>
<td>hat</td>
<td>keiner</td>
<td>gerechnet</td>
<td></td>
</tr>
<tr>
<td></td>
<td>der Professor</td>
<td>hatte</td>
<td>dem Studenten vor einigen Monaten</td>
<td>versprochen</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>das Gutachten</td>
</tr>
</tbody>
</table>

The topological fields for German impose the following field constraints for verb-two type sentences, which in fact are quite typical and natural:

- the Vorfeld contains exactly one constituent.
- the FINIT-field contains a finite verb.
Chapter 18. Word-Order and Dependency Structure

- a verb-two sentence can be partitioned into the following disjoint fields: Vorfeld, FINIT, Mittelfeld, Verbkomplex, Nachfeld
- the fields come in the order in which they are given above.

18.2 Non-Projectivity

The word order in German is quite liberal. Of course, constituents can be freely put into fields as long as some quite liberal constraints are satisfied. Even worse: it may even be possible that constituents are not projective. This means that a constituent does not form a subsequence of the whole sentence, i.e. that words of other constituents may intervene at some positions. For instance, the following examples contain non-projective constituents:

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schmoren müssen wird er in der Hölle.</td>
</tr>
<tr>
<td>Cäsar hat Gallier getroffen, die ein ganzes Wildschwein auf einmal verzehren konnten.</td>
</tr>
<tr>
<td>Das Gutachten hat der Professor versprochen zu schreiben.</td>
</tr>
</tbody>
</table>

18.3 Dependency Structures

Dependency Structures allow to represent the constituent structure of sentences in whatever order its words are organized. For instance: A dependency structure is a finite feature tree whose features are called roles (subject,pp,infinitive). Each node of a dependency structure is labeled with a word of the sentence. An edge from a head to one of its complements always points to this complement’s head.

Figure 18.1: A Dependency Structure (to be fixed)
The word order is left unspecified in a dependency structure. It can be expressed by specifying an ordering on its nodes. In the example, we name nodes by natural numbers and order the nodes by the ordering on natural numbers.

The yield of a node is the set of nodes accessible from it. For instance, the yield of node 2 (müssen) in the above example is the set \( \{5,1,2\} \) (in der Hölle schmoren müssen). Note that this constituent is non-projective.

## 18.4 Word Order Constraints

We now give some examples for how to express word order constraints for a dependency structure.

- a determiner is the minimal element of the yield of its head (the noun).
- an adjective comes before its head (the noun).
- The position of a zu particle must be the one preceding its head (the verb).

## 18.5 Role Dependency Constraints

We call a node of a dependency structure a lexical node. Each lexical node of a dependency structure contains information on category, agreement and valency (i.e. what types of complements are required by the word).

- Each word of a sentence must satisfy one of its frames in the lexicon.
- Subjects must be nominative and agree with its head (the verb).
- Accusative objects must be accusative.
- The root of a sentence must be a finite verb.
- A lexical node must have precisely the complements required by its valency. Each required complement is satisfied by a unique edge labeled with the complement (role) type.

## 18.6 Putting it together in an example

Consider the sentence: die Frau liebt der Mann It is clear that ‘der Mann’ must be the subject and ‘die Frau’ the accusative object, but how does constraint propagation derive this conclusion.

1. ‘liebt’ must be the root of the sentence since it is the only finite verb.
2. ‘der’ cannot be the determiner for ‘Frau’ since it comes after it.
   
   (a) therefore ‘Mann’ is the only possible head for ‘der’.
   
   (b) as a consequence, ‘Mann’ cannot also be head of ‘die’.
(c) thus, only ‘Frau’ can be head of ‘die’.

3. due to the agreement constraint, ‘der Mann’ must be nominative:

   (a) therefore ‘Mann’ cannot be the accusative object of ‘liebt’.
   (b) the only role left for ‘Mann’ is to be subject of ‘liebt’.
   (c) therefore, the only role left for ‘Frau’ is to be object of ‘liebt’.
19

Constraint-Based Dependency Parsing

19.1 Introduction

There is very little agreement about what might constitute a dependency grammar. What does a dependency grammar look like? How can we process it efficiently? What is the essence of dependency parsing? In this chapter, we attempt to provide an answer to these questions. Our formulation should by no means be construed as the definitive word on the issue. Rather, we propose to demonstrate how the problem can be approached in a declarative, constraint-based fashion, that leads naturally to an elegant and efficient implementation.

The presentation will proceed in a manner similar to our treatment of dominance descriptions in Section 16.2: we will develop a formal characterization of what constitutes a valid dependency tree. This formalization, modulo details of syntax, can then be viewed as a constraint-based program.

Again, it will turn out that simple set theory is a wonderful modeling tool, and permits to succinctly capture the more complex properties of the formal models of interest. We are going to use it a lot; in fact, almost exclusively! Computationally speaking, this is also good news since Oz directly supports sets and constraints on sets.

Before we dive into the formal treatment, let us first briefly motivate the dependency grammar approach. One reason you may find dependency grammar attractive is that it doesn’t lose its marbles when confronted with a language with free word order (freer than english). Consider the sentence below: What is challenging in this sentence?

Figure 19.1: Challenging sentence with liberal word order
is the very liberal word order. Peter, the subject of the main auxiliary verb hat, appears between the past participle versprochen and its dative object mir. The infinitive clause zu lesen is extraposed to the right out of the main clause, while its own accusative object is moved, for purposes of emphasis, to the very front of the sentence. As illustrated in the figure below, such a sentence requires an analysis with crossing edges. Its dependency parse tree will traditionally be represented by a collection of attribute value matrices (AVMs) where the boxed integers represent coreference indices (a conventional notation in computational linguistics). These AVMs form a feature structure organized into a dependency tree where the root is node 3.

In the remainder of this chapter, we are going to turn our attention entirely to the study of such collections of AVMs. We are going to make precise the conditions required for

---

1Joachim Niehren proposes the following sentence, which exhibits the same structure, but sounds more convincing to the german ear: Genau diese Flasche Wein hat mir mein Kommissionär versprochen auf der Auktion zu ersteigern
19.2. The Big Picture

What are you supposed to learn in this chapter? First, you don’t have to be a fan of dependency parsing in order to benefit from the material here. You could take these ideas and techniques and apply them to HPSG for example, and that would be a wonderful project… or you might apply them to molecular biology, perhaps. To help you keep ‘the big picture’ in perspective, we now review briefly the more important points worth learning.

19.2.1 Model Generation vs Model Elimination

Many of you may have grown up, professionally speaking, on a diet of logic programming or on even more classical fare. As such, you have been encouraged to think about problems in a generative/constructive way. Constraint programming requires a completely different perspective. This is best illustrated with an analogy:

- The traditional way is much like constructing objects with Lego: you put pieces together until you have achieved the desired shape.
- The constraint programming way is rather like sculpting or carving: you take away what you don’t want until only the desired shape remains.

In constraint programming, you state an ideal, then remove undesirable models from consideration by incremental strengthening of your judgement. You might look at it as computational philosophy… or not, if you happen to be a philosopher! This chapter illustrates the point quite clearly with dependency structures:

- we do not assemble the dependency structure by constructively connecting nodes together
- instead we state the ideal truth that characterizes what it means for a collection of nodes to be arranged in a dependency structure
- then we remove from consideration all models that violate grammatical conditions such as agreement.

It is important not to fall in the generate and test trap. In order to take full advantage of constraint programming, you must model your problem in a way that takes maximal advantage of constraint propagation. In constraint programming, negative information plays an essential role: you first state a universal truth, and then indicate which parts of it are not true. In the end, as Sherlock Holmes might put it, ‘what ever remains must be the case.’
19.2.2 Finite Formulations

It is worth stressing the fact that constraint programming is exceptionally strong when applied to problems with a finite formulation. This was the case for dominance descriptions: each description contains a finite number of variables. It is the case again for dependency parsing: there are finitely many words in the sentence, and each one has finitely many lexical frames.

Whenever you tackle a new problem, it is a good idea to look for the parts of it that may be given a finite formulation.

19.2.3 Sets Are Wonderful

Simple set theory is a remarkably expressive tool. Section 16.2 already made this point, but our treatment of dependency parsing really brings the point home: we use sets for everything! Furthermore, since Oz supports sets and constraints on sets, using them as part of the formulation immediately lends it efficient computational content.

19.2.4 Modelization Techniques

This last point concerns a more specific issue: practically, how does one go about modeling a complex problem in a way that benefits from concurrent constraint programming? There is, of course, no simple or easy answer to this question. However, we illustrate various modelization techniques on the following two issues:

- the treatment of dependency structure
- the treatment of word order

19.2.4.1 Partial vs Total Functions

To take advantage of constraint propagation, it is desirable to state global truths. Partial functions have the disadvantage that you can only state properties at points where they are defined. For example, the features of a feature structure are often regarded as partial functions.

We demonstrate a nice trick to turn partial functions into total functions: Given a partial function \( f : A \rightarrow B \) replace it by the total function \( F : A \rightarrow 2^B \) where \( F(x) \) is of cardinality at most 1. \( f(x) \in F(x) \) if \( f \) is defined at \( x \), else \( F(x) = \emptyset \). Thus \( F \) will map to the empty set at points where \( f \) was not previously defined, and to a singleton set elsewhere.

19.3 Formal Framework

Our presentation of dependency grammar follows modern linguistic practice: it consists of a lexicon and a collection of principles. Further, it illustrates the expressiveness of set constraints and selection constraints and demonstrates how they can provide elegant encoding and efficient processing of various forms of ambiguity such as lexical and attachment ambiguity.
19.3.1 Dependency Grammar

A dependency grammar $G$ is a 7-tuple

$$(\text{Words}, \text{Cats}, \text{Agrs}, \text{Comps}, \text{Mods}, \text{Lexicon}, \text{Rules})$$

**Words** a finite set of strings notating the fully inflected forms of words

**Cats** a finite set of categories such as $\text{det}$ for determiner or $\text{vfin}$ for finite verb

**Agrs** a finite set of agreement tuples such as $<$masc sing 3 nom$>$

**Comps** a finite set of complement role types such as $\text{subject}$ or $\text{dative}$ (for dative NP complement)

**Mods** a finite set of modifier role types, such as $\text{adj}$ for adjectives, disjoint from $\text{Comps}$. We write $\text{Roles} = \text{Comps} \uplus \text{Mods}$ for the set of all role types. They will serve to label the edges of a dependency tree.

**Lexicon** a finite set of lexical entries (see later)

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

In practice, in addition to category and agreement, we will need other features to encode miscellaneous details (e.g. ‘zu’ particle, detachable verb prefixes, choice of auxiliary verb). For the time being, we will simply ignore them and postpone their introduction until the presentation of our implementation.

19.3.2 Lexicon and Lexical Entries

The lexicon is a finite collection of lexical entries. A lexical entry is an AVM with signature:

$${\begin{array}{lll} \text{string} & : & \text{Words} \\
\text{cat} & : & \text{Cats} \\
\text{agr} & : & \text{Agrs} \\
\text{roles} & : & \{\text{Comps}\}^2 \\
\end{array}}$$

We write attribute access in functional notation. If $e$ is a lexical entry: $\text{string}(e)$ is the full form of the corresponding word, $\text{cat}(e)$ is the category, $\text{agr}(e)$ the agreement, and $\text{roles}(e)$ the valency expressed as a set of complement roles.
Constraint-Based Dependency Parser: Implementation

In this chapter, we develop a complete implementation of a dependency parser that closely follows the theoretical framework presented earlier. We also provide a Demo Applet\(^1\) with which you can interactively experiment with our dependency parser.

20.1  **DG_Parser.oz**

This functor exports function \texttt{MakeScript} which takes as argument a list of atoms representing an input sentence and returns a predicate appropriate for use with e.g. \texttt{Search.all} or \texttt{Explorer.all}.

\begin{verbatim}
231a (DG_Parser.oz 231a)≡
  functor import
  Lex at ’DG_Lexicon.ozf’
  Sel(select : Select) at ’DG_Selection.ozf’
  Wgh(weightC : WeightC) at ’DG_Weight.ozf’
  FD FS
  export
  MakeScript CheckWords
  define
    (DG: MakeScript 232a)
    (DG: WordOrderLocal 234b)
    (DG: Word2Node 234c)
    (DG: Gamma 235d)
    (DG: Encoded Values 238e)
    (DG: CheckWords 242c)
  end
\end{verbatim}

20.1.1  **MakeScript**

\texttt{MakeScript} takes as argument a list of atoms representing the words of a sentence to be analyzed and returns a predicate appropriate for encapsulated search. This predicate characterizes a solution: a list \texttt{Nodes} of lexical nodes.

\(^1\)code/DG_Demo.oz
Chapter 20. Constraint-Based Dependency Parser: Implementation

232a  \( \text{(DG: MakeScript} \text{232a)} \equiv \)

\[
\text{fun \{MakeScript Words\}} \quad \text{proc \{$N \}$ Nodes) \quad N = \text{(Length Words)} \quad \text{(DG MakeScript: Defs 232c)} \\
\quad \text{in \{DG MakeScript: create lexical nodes 232b\}} \quad \text{DG MakeScript: sentence partitioning rule 232c) \quad \text{DG MakeScript: install strict yields 232d \quad DG MakeScript: word order constraints 232e \quad DG MakeScript: distribution strategy 232f) \quad end \quad end}
\]

First we create the lexical nodes and then we install some binary constraints between every pair of them. Note

232b  \( \text{(DG MakeScript: create lexical nodes} \text{232b)} \equiv \)

\[
\text{Nodes = \{List.mapInd Words \text{fun \{$I W\}$ (Word2Node I W N Topo) end\}} \quad \text{(ForAll Nodes \text{proc \{$W1\}$ \quad \text{(ForAll Nodes \text{proc \{$W2\} \quad \text{(W2 is daughter of W1 iff W1 is mother of W2 232a \quad \text{(Try all possible edges from W1 to W2 232b) \quad end \quad end})}})}}}
\]

A german sentence is partitioned into 4 fields: (1) the Vorfeld \(VF\), (2) the root field \(RF\), which is a singleton containing just the finite \(Root\) of the sentence, (3) the Mittelfeld \(MF\), and (4) the Nachfeld \(NF\) for extrapositions. There is an additional restriction that the Vorfeld must contain just one constituent: thus we denote by \(VFR\) the root of the Vorfeld. The fields are sequential and form a partition of the \(Sentence\). Variable \(Topo\) denotes a record that contains all useful information pertaining to fields.

232c  \( \text{(DG MakeScript: Defs} \text{232c)} \equiv \)

\[
\text{VF = \{FS.var.upperBound 1\#N\}} \quad \text{Root = \{FD.int 1\#N\}} \quad \text{RF = \{FS.var.upperBound 1\#N\}} \quad \text{\{FS.cardRange 1 1 RF\}} \quad \text{\{FS.include Root RF\}} \quad \text{MF = \{FS.var.upperBound 1\#N\}} \quad \text{NF = \{FS.var.upperBound 1\#N\}} \quad \text{VFR= \{FD.int 1\#N\}} \quad \text{\{FS.include VFR VF\}} \quad \text{Topo = o(vf:VF rf:RF mf:MF nf:NF list:[VF RF MF NF] vfr:VFR root:Root\}} \quad \text{Sentence = \{FS.value.make 1\#N\}} \quad \text{\{FS.int.seq \quad \text{Topo.list\}} \quad \text{\{FS.partition Topo.list Sentence\}}}
\]
233a  (W2 is daughter of W1 iff W1 is mother of W2) ≡
(FS.reified.include W2.index W1.daughters) =
(FS.reified.include W1.index W2.mother)

233b  (Try all possible edges from W1 to W2) ≡
{ForAll Roles
  proc ($ R)
    thread
      or (FS.include W2.index W1.role.R)
      (Gamma R W1 W2)
    [] (FS.exclude W2.index W1.role.R)
  end
end)

The *sentence partitioning rule* states that each word of the input sentence fills precisely one role: either it is root or it fills a role on some other node. Thus the root field RF together with the daughter sets of all nodes must form a partition of the input Sentence.

233c  (DG MakeScript: sentence partitioning rule) ≡
{FS.partition
  {FoldL Nodes
    fun ($ L N) {Append (Record.toList N.role) L} end [RF]}
Sentence}

The strict yield of W1 is the union of contributions from all nodes. W2 contributes nothing if it is not a daughter of W1, otherwise it contributes its full yield. This condition can be expressed using the *weighted set constraint* WeightC.

233d  (DG MakeScript: install strict yields) ≡
{ForAll Nodes
  proc ($ W1)
    W1.yields =
    {FS.unionN
      {Map Nodes
        fun ($ W2)
          {WeightC
            (FS.reified.include W2.index W1.daughters)
            W2.yield)
        end}}
    end}
end)

Local word order constraints are primarily enforced by procedure WordOrderLocal.

233e  (DG MakeScript: word order constraints) ≡
{ForAll Nodes WordOrderLocal}

So far, we just enforce the condition that the prenominal domain of an NP is convex (i.e. forms a contiguous interval with no holes, no insertions). Here, we take advantage of the fact that roles det and adj can only occur on NPs; thus, it is harmless to enforce the constraint also when the Node is not an NP.
Chapter 20. Constraint-Based Dependency Parser: Implementation

234a (DG: WordOrderLocal)

```tcl
proc {WordOrderLocal Node}
    SELF = (FS.value.single Node.index)
    L1 = [Node.role.det Node.role.adj SELF]
    in
        {FS.int.seq L1}
        {FS.int.convex {FS.unionN L1}}
    end
```

In order to solve the CSP, we need a labeling strategy. Here is an obvious one that doesn’t attempt to be particularly clever:

1. apply the default naive labeling strategy on the collection of mother sets:

   \{mother(w) | w ∈ V\}

2. apply first-fail on the collection of lexicon entry selectors:

   \{entryIndex(w) | w ∈ V\}

3. finally use again the default naive labeling strategy on the collection of daughter sets:

   \{ρ(w) | ρ ∈ Roles w ∈ V\}

234b (DG MakeScript: distribution strategy)

```tcl
{FS.distribute naive {Map Nodes fun {§ N} N.mother end}}
{FD.distribute ff {Map Nodes fun {§ N} N.entryIndex end}}
{FS.distribute naive
  {FoldL Nodes
    fun {§ L N} {Append {Record.toList N.role} L} end nil}}
```

20.1.2 Word2Node

procedure Word2Node creates a new lexical Node. It takes as arguments I, the position of the word in the input sentence, W, the atom representing the word, N, the number of words in the sentence, and Topo, a record containing information on topological fields.

234c (DG: Word2Node)

```tcl
proc {Word2Node I W N Topo Node}
    L = (Dictionary.get Lex.lexicon W)
    in
        Node =
            node(index : I
                  word : W
                  entries : L
                  field : Topo
            )
            (DG Word2Node: features)
    end
```

235c Word2Node
We equip the lexical node with feature `entryIndex` to indicate which entry in list \( L \) is selected.

Further we also give it features `cat` (category) and `agr` (agreement) which must be licensed by the selected entry:

Feature `marks` is a set that contains `zu` if the ‘zu’ particle is morphologically part of the word (e.g. einzukaufen), `vpref` if the separable prefix is not separated, `haben` if auxiliary ‘haben’ is desired, `sein` if auxiliary ‘sein’ is desired. Feature `aux` is also a set of marks and is only used on auxiliary verbs to indicate which auxiliary it is (i.e. either `haben` or `sein`).

They must also correspond to the selected entry:

Feature `vprefs` is a set that is either empty or contains the expected separated verb prefix (e.g. ‘ein’ as in ‘ich kaufe etwas ein’).
A lexical entry specifies required complements in feature `comps_req` and optional complements in feature `comps_opt`. The set of complement roles of the lexical node is represented by feature `comps` and is bounded by the required complements at the lower end and the union of the required complements and the optional complements at the upper end.

```
local Lo = {Select {Map L fun {E} E.comps_req end} Node.entryIndex}
Hi = {Select {Map L fun {E} {FS.union E.comps_req E.comps_opt} end} Node.entryIndex}
in {FS.subset Lo Node.comps} {FS.subset Node.comps Hi}
end
```

On feature `role` is a record that maps each possible role \( \rho \) to a set of (indices) of lexical nodes denoting the immediate daughters of type \( \rho \). For example, `Node.role.adj` denotes the set of adjectives of this node.

```
role : {MakeRoleRecord N}
```

A complement role has cardinality at most 1. It is 1, precisely when it is licensed by the node's valency `Node.comps`.

```
end
```
(ForAll Comps
  proc ($ C)
    {FS.card Node.role.C) =
    {FS.reified.include Lex.role.val2int.C Node.comps}
  end)

The set of daughters of the node is formed by the union of all its role sets.

The only node without a mother is the root

We also introduce fieldIndex to indicate in which field the word occurs. (not used so far!)

Now we enforce the condition that the Vorfeld contains a unique constituent. More precisely, we distinguish a root Topo.vfr of the Vorfeld: (1) either the word is not in the vorfeld, (2) or it is the root of the vorfeld, (3) or it is in the vorfeld, but is not the root, and its mother is also in the vorfeld.
thread
  or {FS.exclude Node.index Topo.vf}
  [] Node.index = Topo.vf
  [] Node.index\=Topo.vf
    {FS.include Node.index  Topo.vf}
    {FSsubset Node.mother Topo.vf}
  end
end

If the word occurs in the nachfeld, then its mother cannot be root (is that right?)

A word cannot at the same time take a ‘zu’ particle complement and also have a ‘zu’
particle in its morphological form. We equip the node with a haszu feature to indicate
whether it has either.

We equip a lexical node with a roleLabel feature. This serves no other purpose than
convenience for display purposes. It indicates the label on the incoming dependency
edge if any.
A subject must be a nominative NP and must agree with its mother.

For a nominative complement (e.g. with sein) the conditions are the same except that agreement is not required.

An object complement must be an accusative NP.

A dative complement must be a dative NP.

A det (determiner) complement must have category det and agree with its mother. Also, it must occur left-most in the yield of the mother (i.e. it must the minimum element).
A ‘zu’ particle must have category `part` (particle), must be the word `zu` (this should really be encoded), and must immediately precede its mother.

A separable verb prefix must have category `vpref` and must be the prefix expected by the mother. Furthermore, it can only occur separated at the end of the mittelfeld, and its mother must be root.

A `vp_zu` complement is an infinitive with zu: `W2` must be an infinitive verb and it must have a ‘zu’ as indicated by its `zu` feature.

A `vp_inf` complement is an infinitive without zu: `W2` must be an infinitive verb and it must have no ‘zu’ as indicated by its `zu` feature.
A \textit{vp\_past} complement is a past participle. The auxiliary it expects must correspond to the mother. Further, since we are not treating relative clauses here, we can impose the restriction that if the mother is root, then then complement must occur at the end of the mittelfeld.

\begin{verbatim}
[ ] vp\_inf then
   W2.cat = CAT_VINF
   W2.haszu = 0
\end{verbatim}

\begin{verbatim}
[ ] vp\_past then
   W2.cat = CAT_VPAST
   \{FS.card (FS.intersect W1.aux W2.marks)} >: 0
   (W2.index=:FS.int.max W2.field.mf)\=(W1.index=:W1.field.root)
\end{verbatim}

For an \textit{adj} (adjective) modifier, its category must be \textit{adj} and its mother must be a noun. The adjective must agree with the noun.

\begin{verbatim}
[ ] adj then
   W1.cat = CAT_N
   W2.cat = CAT_ADJ
   W1.agr = W2.agr
\end{verbatim}

For an \textit{adv} (adverb) modifier, the mother must be a verb. Furthermore, if \textit{W1} is not root, then both must be in the same field.

\begin{verbatim}
[ ] adv then
   \{FS.include W1.cat CATS_V\}
   W2.cat = CAT_ADV
   (W1.index\=:W1.field.root)\=<:(W1.fieldIndex\=:W2.fieldIndex)
\end{verbatim}

For a \textit{pp\_np} (PP) modifier, it must have category \textit{prep} (yes, we consider the preposition to be head of the PP) and the mother must be an NP or a verb. Further, the preposition must be left-most in its own yield.

\begin{verbatim}
[ ] \textit{pp\_np} then
\end{verbatim}
Chapter 20. Constraint-Based Dependency Parser: Implementation

Now some additional constraints that it pays off to inject locally. If \( W_2 \) is in the Nachfeld, then either its mother \( W_1 \) is also in the Nachfeld, or \( W_1 \) is an infinitive verb (no relative clauses yet).

20.1.4 CheckWord

procedure CheckWords is used by our demo to check that all words of an input sentence are actually known to the lexicon prior to invoking MakeScript.

20.2 DG_Lexicon.oz

We now turn to the lexicon.

functor import Lat at 'DG_Lattice.ozf'
export
  Cat Cats
  Gender Number Person Case Agr Agrs
  Role Roles Vpref Vprefs Mark Marks Entry
  Comps Mods
  Lexicon
define
  Cat = (New Lat.domain init([n pro vinf vfin vpast det part vpref adj adv prep]))
  Cats = (New Lat.set init(Cat))
  Gender = (New Lat.domain init([masc fem neut]))
Number = (New Lat.domain init([sing plur]))
Person = (New Lat.domain init([1 2 3]))
Case = (New Lat.domain init([nom acc dat gen]))
Agr = (New Lat.cartesian init([Gender Number Person Case]))
Agrs = (New Lat.set init(Agr))
Comps = [det subject nominative object dative zu vpref vp_zu vp_past vp_inf]
Mods = [adj adv pp_np]
Role = (New Lat.domain init([Append Comps Mods]))
Roles = (New Lat.accumulatingSet init(Role))
Vpref = (New Lat.domain init([ein]))
Vprefs = (New Lat.accumulatingSet init(Vpref))
Mark = (New Lat.domain init([zu vpref haben sein]))
Marks = (New Lat.accumulatingSet init(Mark))
Entry = (New Lat.avm init(o(cats:Cats agrs:Agrs comps_req:Roles comps_opt:Roles vpدخل vp_conf:Vprefs marks:Marks aux:Marks)))

Lexicon = {Dictionary.new}
proc {ENTER Word Spec}
    {Dictionary.put Lexicon Word}
    {Entry.encode Spec}|{Dictionary.condGet Lexicon Word nil}
end

{ENTER peter
lex(cats : [n]
    agrs : [[masc sing 3 [nom acc dat]]]
    comps_opt : [det])

{ENTER peters
lex(cats : [n]
    agrs : [[masc sing 3 gen]]
    comps_opt : [det])

{ENTER maria
lex(cats : [n]
    agrs : [[fem sing 3 [nom acc dat]]]
    comps_opt : [det])

{ENTER marias
lex(cats : [n]
    agrs : [[fem sing 3 gen]]
    comps_opt : [det])

{ENTER ich
lex(cats : [pro]
    agrs : [[sing 1 nom]])

{ENTER mich
lex(cats : [pro]
    agrs : [[sing 1 acc]])

{ENTER mir
lex(cats : [pro]
    agrs : [[sing 1 dat]])
\{ENTER du
lex(cats : [pro]
agrs : [[sing 2 nom]])
\}
\{ENTER dich
lex(cats : [pro]
agrs : [[sing 2 acc]])
\}
\{ENTER dir
lex(cats : [pro]
agrs : [[sing 2 dat]])
\}
\{ENTER er
lex(cats : [pro]
agrs : [[masc sing 3 nom]])
\}
\{ENTER ihn
lex(cats : [pro]
agrs : [[masc sing 3 acc]])
\}
\{ENTER ihm
lex(cats : [pro]
agrs : [[masc neut 3 dat]])
\}
\{ENTER es
lex(cats : [pro]
agrs : [[neut sing 3 [nom acc]]])
\}
\{ENTER sie
lex(cats : [pro]
agrs : [[fem sing 3 [nom acc]]])
\}
\{ENTER ihr
lex(cats : [pro]
agrs : [[fem sing 3 [nom acc]] [plur 3 [nom acc]]])
\}
\{ENTER ihnen
lex(cats : [pro]
agrs : [[fem sing 3 dat] [plur 2 nom]])
\}
\{ENTER wir
lex(cats : [pro]
agrs : [[plur 3 dat]])
\}
\{ENTER uns
lex(cats : [pro]
agrs : [[plur 1 nom]])
\}
\{ENTER euch
lex(cats : [pro]
agrs : [[plur 2 [acc dat]]])
\}
\{ENTER jemand
lex(cats : [pro]
agrs : [[sing 3 nom]])
\}
\{ENTER jemanden
lex(cats : [pro]
agrs : [[sing 3 acc]])
\}
\{ENTER jemandem
lex(cats : [pro]
agrs : [[sing 3 dat]])
\}
{ENTER frau
  lex(cats : [n]
  agrs : [[fem sing 3]]
  comps_opt : [det]})

{ENTER frauen
  lex(cats : [n]
  agrs : [[fem plur 3]]
  comps_opt : [det]})

{ENTER mann
  lex(cats : [n]
  agrs : [[masc 3 sing [nom acc dat]]]
  comps_opt : [det]})

{ENTER mannes
  lex(cats : [n]
  agrs : [[masc 3 sing gen]]
  comps_opt : [det]})

{ENTER männer
  lex(cats : [n]
  agrs : [[masc 3 plur [nom acc gen]]]
  comps_opt : [det]})

{ENTER männern
  lex(cats : [n]
  agrs : [[masc 3 plur dat]]
  comps_opt : [det]})

{ENTER buch
  lex(cats : [n]
  agrs : [[neut 3 sing [nom acc dat]]]
  comps_opt : [det]})

{ENTER buches
  lex(cats : [n]
  agrs : [[neut 3 sing gen]]
  comps_opt : [det]})

{ENTER Bücher
  lex(cats : [n]
  agrs : [[neut 3 plur [nom acc gen]]]
  comps_opt : [det]})

{ENTER büchern
  lex(cats : [n]
  agrs : [[neut 3 plur dat]]
  comps_opt : [det]})

% {ENTER jeder
%  lex(cats : [det]
%  agrs : [[masc sing 3 nom]
%          [fem sing 3 [dat gen]]
%  ]})
Chapter 20. Constraint-Based Dependency Parser: Implementation

{ENTER der
  lex(cats : [det]
     agrs : [[masc sing 3 nom]
            [fem sing 3 [dat gen]]
            [plur 3 gen]])}

{ENTER das
  lex(cats : [det]
     agrs : [[neut sing 3 [nom acc]]])}

{ENTER den
  lex(cats : [det]
     agrs : [[masc sing 3 acc]
            [plur 3 dat]])}

{ENTER dem
  lex(cats : [det]
     agrs : [[[masc neut] sing 3 dat]]})

{ENTER des
  lex(cats : [det]
     agrs : [[[masc neut] sing 3 gen]])}

{ENTER die
  lex(cats : [det]

% {ENTER eine
%  lex(cats : [det]
%     agrs : [[[nom acc] [[fem sing]] 3]])

{ENTER lieben
  lex(cats : [vinf]
     comps_opt : [zu object]])

{ENTER liebe
  lex(cats : [vfin]
     agrs : [[1 sing nom]]
     comps_req : [subject]
     comps_opt : [object]])

{ENTER liebst
  lex(cats : [vfin]
     agrs : [[2 sing nom]]
     comps_req : [subject]
     comps_opt : [object]])

{ENTER liebt
  lex(cats : [vfin]
     agrs : [[3 sing nom]]
     comps_req : [subject]
     comps_opt : [object]])

{ENTER lieben
  lex(cats : [vfin]
     agrs : [[[1 3] plur nom]]
     comps_req : [subject]
     comps_opt : [object]])
{ENTER liebet
lex(cats : [vfin]
agrs : [[2 plur nom]]
comps_req : [subject]
comps_opt : [object]])

{ENTER geliebt
lex(cats : [vpast]
comps_opt : [object]
marks : [haben]])

{ENTER laufen
lex(cats : [vinf]
comps_opt : [zu]])

{ENTER laufe
lex(cats : [vfin]
agrs : [[1 sing nom]])

{ENTER laufst
lex(cats : [vfin]
agrs : [[2 sing nom]])

{ENTER lauft
lex(cats : [vfin]
agrs : [[3 sing nom] [2 plur nom]])

{ENTER laufen
lex(cats : [vfin]
agrs : [[1 3] plur nom]])

{ENTER gelaufen
lex(cats : [vpast]
marks : [sein]])

{ENTER lesen
lex(cats : [vinf]
comps_opt : [zu object]])

{ENTER lese
lex(cats : [vfin]
agrs : [[1 sing nom]]
comps_req : [subject]
comps_opt : [object]])

{ENTER liest
lex(cats : [vfin]
agrs : [[2 3] sing nom]]
comps_req : [subject]
comps_opt : [object]])

{ENTER lesen
lex(cats : [vfin]
agrs : [[[1 3] plur nom]]
comps_req : [subject]
comps_opt : [object]])

{ENTER leset
lex(cats : [vfin]
comps_opt : [zu object]])
agrs : [[2 plur nom]]
comps_req : [subject]
comps_opt : [object])

{ENTER gelesen
lex(cats : [vpast]
    comps_opt : [object]
    marks : [haben]])

{ENTER versprechen
lex(cats : [vinf]
    comps_opt : [zu object dative])

{ENTER versprechen
lex(cats : [vinf]
    comps_req : [vp_zu]
    comps_opt : [zu dative])

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

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

{ENTER versprichst
lex(cats : [vfin]
    agrs : [[2 sing nom]]
    comps_req : [subject]
    comps_opt : [object dative])

{ENTER versprichst
lex(cats : [vfin]
    agrs : [[2 sing nom]]
    comps_req : [subject vp_zu]
    comps_opt : [dative])

{ENTER verspricht
lex(cats : [vfin]
    agrs : [[3 sing nom]]
    comps_req : [subject]
    comps_opt : [object dative])

{ENTER verspricht
lex(cats : [vfin]
    agrs : [[3 sing nom]]
    comps_req : [subject vp_zu]
    comps_opt : [dative])

{ENTER versprechen
lex(cats : [vfin]
    agrs : [[[1 3] plur nom]]
    comps_req : [subject]
{ENTER versprechen
lex(cats : [vfin]
agrs : [[[[1 3] plur nom]]
comps_req : [subject vp_zu]
comps_opt : [dative]])
{ENTER versprechet
lex(cats : [vfin]
agrs : [[2 plur nom]]
comps_req : [subject]
comps_opt : [object dative]])
{ENTER versprochen
lex(cats : [vpast]
comps_opt : [object dative]
marks : [haben]])
{ENTER ein
lex(cats : [vpref]])
{ENTER kaufen
lex(cats : [vinf]
comps_opt : [zu object dative]])
{ENTER kaufe
lex(cats : [vfin]
agrs : [[1 sing nom]]
comps_req : [subject]
comps_opt : [object dative]])
{ENTER kauft
lex(cats : [vfin]
agrs : [[2 sing nom]]
comps_req : [subject]
comps_opt : [object dative]])
{ENTER kauft
lex(cats : [vfin]
agrs : [[3 sing nom][2 plur nom]]
comps_req : [subject]
comps_opt : [object dative]])
{ENTER kaufen
lex(cats : [vfin]
agrs : [[[1 3] plur nom]]
comps_req : [subject]
comps_opt : [object dative])}
{ENTER gekauft
lex(cats : [vpast]
comps_opt : [object dative]
marks : [haben])}

{ENTER einkaufen
lex(cats : [vinf]
comps_opt : [zu object dative]
marks : [vpref])}
{ENTER einzukaufen
lex(cats : [vinf]
comps_opt : [object dative]
marks : [vpref zu])}
{ENTER kaufe
lex(cats : [vfin]
agrs : [[1 sing nom]]
vpref : [ein]
comps_req : [subject vpref]
comps_opt : [object dative])}
{ENTER einkaufe
lex(cats : [vfin]
agrs : [[1 sing nom]]
comps_req : [subject]
comps_opt : [object dative]
marks : [vpref])}
{ENTER kaufst
lex(cats : [vfin]
agrs : [[2 sing nom]]
vpref : [ein]
comps_req : [subject vpref]
comps_opt : [object dative])}
{ENTER einkaufst
lex(cats : [vfin]
agrs : [[2 sing nom]]
comps_req : [subject]
comps_opt : [object dative]
marks : [vpref])}
{ENTER kaufst
lex(cats : [vfin]
agrs : [[3 sing nom][2 plur nom]]
vpref : [ein]
comps_req : [subject vpref]
comps_opt : [object dative])}
{ENTER einkauf
lex(cats : [vfin]
agrs : [[3 sing nom][2 plur nom]]
comps_req : [subject]
comps_opt : [object dative]
marks : [vpref])

{ENTER kaufen
lex(cats : [vfin]
agrs : [[[[1 3] plur nom]]
vpref : [ein]
comps_req : [subject vpref]
comps_opt : [object dative]])

{ENTER einkaufen
lex(cats : [vfin]
agrs : [[[[1 3] plur nom]]
comps_req : [subject]
comps_opt : [object dative]
marks : [vpref])

{ENTER eingekauft
lex(cats : [vpast]
comps_opt : [object dative]
marks : [vpref])

{ENTER zu
lex(cats : [part])

{ENTER mit
lex(cats : [prep]
comps_req : [dative])

{ENTER haben
lex(cats : [vinf]
comps_opt : [zu object])

{ENTER habe
lex(cats : [vfin]
agrs : [[1 sing nom]]
comps_req : [subject]
comps_opt : [object]])

{ENTER hast
lex(cats : [vfin]
agrs : [[2 sing nom]]
comps_req : [subject]
comps_opt : [object]])

{ENTER hat
lex(cats : [vfin]
agrs : [[3 sing nom]]
comps_req : [subject]
comps_opt : [object]])

{ENTER haben
lex(cats : [vfin]
agrs : [[[1 3] plur nom]]
comps_req : [subject]
comps_opt : [object])
{ENTER haben
lex(cats : [vinf] aux : [haben] comps_opt : [zu] comps_req : [vp_past])
\{ENTER habe
lex(cats : [vfin] agrs : [[1 sing nom]] aux : [haben] comps_req : [subject vp_past])
\{ENTER hast
lex(cats : [vfin] agrs : [[2 sing nom]] aux : [haben] comps_req : [subject vp_past])
\{ENTER hat
lex(cats : [vfin] agrs : [[3 sing nom]] aux : [haben] comps_req : [subject vp_past])
\{ENTER haben
lex(cats : [vfin] agrs : [[1 3] plur nom]] aux : [haben] comps_req : [subject vp_past])
\{ENTER haben
lex(cats : [vfin] agrs : [[2 plur nom]] aux : [haben] comps_req : [subject vp_past])
\{ENTER gehabt
lex(cats : [vpast] aux : [haben] comps_req : [vp_past] marks : [haben])
\{ENTER sein
lex(cats : [vinf]
comps_opt : [zu nominative])}

{ENTER bin
lex(cats : [vfin]
agrs : [[1 sing nom]]
comps_req : [subject]
comps_opt : [nominative])}

{ENTER bist
lex(cats : [vfin]
agrs : [[2 sing nom]]
comps_req : [subject]
comps_opt : [nominative])}

{ENTER ist
lex(cats : [vfin]
agrs : [[3 sing nom]]
comps_req : [subject]
comps_opt : [nominative])}

{ENTER sind
lex(cats : [vfin]
agrs : [[[1 3] plur nom]]
comps_req : [subject]
comps_opt : [nominative])}

{ENTER seid
lex(cats : [vfin]
agrs : [[2 plur nom]]
comps_req : [subject]
comps_opt : [nominative])}

{ENTER gewesen
lex(cats : [vpast]
comps_opt : [nominative]
marks : [sein]])

{ENTER sein
lex(cats : [vinf]
aux : [sein]
comps_opt : [zu]
comps_req : [vp_past])}

{ENTER bin
lex(cats : [vfin]
agrs : [[1 sing nom]]
aux : [sein]
comps_req : [subject vp_past])}

{ENTER bist
lex(cats : [vfin]
agrs : [[2 sing nom]]
aux : [sein]
comps_req : [subject vp_past])}

{ENTER ist
lex(cats : [vfin]
agrs : [[3 sing nom]]
aux : [sein]
comps_req : [subject vp_past])}

{ENTER sind
lex(cats : [vfin]
agrs : [[0 3] plur nom])
aux : [sein]
comps_req : [subject vp_past])}

{ENTER seid
lex(cats : [vfin]
agrs : [[2 plur nom])
aux : [sein]
comps_req : [subject vp_past])}

{ENTER gewesen
lex(cats : [vpast]
aux : [sein]
comps_req : [vp_past]
marks : [sein])}

{ENTER heute
lex(cats : [adv])}

{ENTER schöne
lex(cats : [adj]
agrs : [nom [acc [fem neut]]])}

{ENTER schönen
lex(cats : [adj]
agrs : [[masc sing [acc dat gen]]
[ [fem neut] sing [dat gen]]
plur])}

{ENTER große
lex(cats : [adj]
agrs : [[sing [nom [acc [fem neut]]]]])}

{ENTER großen
lex(cats : [adj]
agrs : [[masc sing [acc dat gen]]
[ [fem neut] sing [dat gen]]
plur])}

{ENTER wollen
lex(cats : [vfin]
comps_opt : [zu object])}

{ENTER wollen
lex(cats : [vfin]
comps_opt : [zu]
comps_req : [vp_inf])}

{ENTER will
lex(cats : [vfin]
agrs : [[0 3] sing nom])
comps_req : [subject]
comps_opt : [object])

{ENTER will
lex(cats : [vfin]
agrs : [[[1 3] sing nom]]
comps_req : [subject vp_inf])

{ENTER willst
lex(cats : [vfin]
agrs : [[2 sing nom]]
comps_req : [subject]comps_opt : [object])

{ENTER willst
lex(cats : [vfin]
agrs : [[2 sing nom]]
comps_req : [subject vp_inf])

{ENTER wollen
lex(cats : [vfin]
agrs : [[[1 3] plur nom]]
comps_req : [subject]comps_opt : [object])

{ENTER wollen
lex(cats : [vfin]
agrs : [[[1 3] plur nom]]
comps_req : [subject vp_inf])

{ENTER wollt
lex(cats : [vfin]
agrs : [[2 plur nom]]
comps_req : [subject]comps_opt : [object])

{ENTER wollt
lex(cats : [vfin]
agrs : [[2 plur nom]]
comps_req : [subject vp_inf])

{ENTER gewollt
lex(cats : [vpast]
comps_opt : [object]marks : [haben])

{ENTER gewollt
lex(cats : [vpast]
comps_req : [vp_inf]marks : [haben])

{ENTER versuchen
lex(cats : [vinf]
comps_opt : [zu object])

{ENTER versuchen
lex(cats : [vinf]
comps_opt : [zu]comps_req : [vp_zu])}
{
  {ENTER versuche}
  lex(cats : [vfin]
      agrs : [[1 sing nom]]
      comps_req : [subject]
      comps_opt : [object])

  {ENTER versucht}
  lex(cats : [vfin]
      agrs : [[1 sing nom]]
      comps_req : [subject vp_zu])

  {ENTER versuchst}
  lex(cats : [vfin]
      agrs : [[2 sing nom]]
      comps_req : [subject]
      comps_opt : [object])

  {ENTER versuchst}
  lex(cats : [vfin]
      agrs : [[2 sing nom]]
      comps_req : [subject vp_zu])

  {ENTER versucht}
  lex(cats : [vfin]
      agrs : [[3 sing nom][2 plur nom]]
      comps_req : [subject]
      comps_opt : [object])

  {ENTER versucht}
  lex(cats : [vfin]
      agrs : [[3 sing nom][2 plur nom]]
      comps_req : [subject vp_zu])

  {ENTER versuchen}
  lex(cats : [vfin]
      agrs : [[[1 3] plur nom]]
      comps_req : [subject]
      comps_opt : [object])

  {ENTER versuchen}
  lex(cats : [vfin]
      agrs : [[[3] plur nom]]
      comps_req : [subject vp_zu])

  {ENTER versucht}
  lex(cats : [vpast]
      comps_opt : [object]
      marks : [haben])

  {ENTER versucht}
  lex(cats : [vpast]
      comps_req : [vp_zu]
      marks : [haben])

  {ENTER behaupten}
  lex(cats : [vinf]
      comps_opt : [zu vp_zu])

  {ENTER behaupte}
lex(cats : [vfin]
    agrs : [[1 sing nom]]
    comps_req : [subject]
    comps_opt : [vp_zu])

{ENTER behauptest
lex(cats : [vfin]
    agrs : [[2 sing nom]]
    comps_req : [subject]
    comps_opt : [vp_zu])

{ENTER behauptet
lex(cats : [vfin]
    agrs : [[3 sing nom][2 plur nom]]
    comps_req : [subject]
    comps_opt : [vp_zu])

{ENTER behaupten
lex(cats : [vfin]
    agrs : [[1 plur nom][3 plur nom]]
    comps_req : [subject]
    comps_opt : [vp_zu])

{ENTER behauptet
lex(cats : [vpast]
    comps_opt : [vp_zu]
    marks : [haben]])

{ENTER schenken
lex(cats : [vinf]
    comps_opt : [zu object dative]])

{ENTER schenke
lex(cats : [vfin]
    agrs : [[1 sing nom]]
    comps_req : [subject]
    comps_opt : [object dative]])

{ENTER schenkst
lex(cats : [vfin]
    agrs : [[2 sing nom]]
    comps_req : [subject]
    comps_opt : [object dative]])

{ENTER schenkt
lex(cats : [vfin]
    agrs : [[3 sing nom]]
    comps_req : [subject]
    comps_opt : [object dative]])

{ENTER schenken
lex(cats : [vfin]
    agrs : [[1 plur nom][3 plur nom]]
    comps_req : [subject]
    comps_opt : [object dative]])

{ENTER schenket
lex(cats : [vfin]

agrs : [[2 plur nom]]
comps_req : [subject]
comps_opt : [object dative])
{ENTER getschenkt
lex(cats : [vpast]
comps_opt : [object dative]
marks : [haben])}

{ENTER richter
lex(cats : [n]
agrs : [[masc 3 sing [nom acc dat]]
[masc 3 plur [nom acc gen]]]
comps_opt : [det])}
{ENTER richters
lex(cats : [n]
agrs : [[masc 3 sing gen]]
comps_opt : [det])}
{ENTER richtern
lex(cats : [n]
agrs : [[masc 3 plur dat]]
comps_opt : [det])}

{ENTER gestehen
lex(cats : [vinf]
comps_opt : [zu object dative])}
{ENTER gestehen
lex(cats : [vinf]
comps_req : [vp_zu]
comps_opt : [zu dative])}
{ENTER gestehe
lex(cats : [vfin]
agrs : [[1 sing nom]]
comps_req : [subject]
comps_opt : [object dative])
{ENTER gestehe
lex(cats : [vfin]
agrs : [[1 sing nom]]
comps_req : [subject vp_zu]
comps_opt : [dative])}
{ENTER gestehst
lex(cats : [vfin]
agrs : [[2 sing nom]]
comps_req : [subject]
comps_opt : [object dative])
{ENTER gestehst
lex(cats : [vfin]
agrs : [[2 sing nom]]
comps_req : [subject vp_zu]
comps_opt : [dative])}
{ENTER gesteht
lex(cats : [vfin]
agrs : [[3 sing nom]]
comps_req : [subject]
comps_opt : [object dative])

{ENTER gestehet
lex(cats : [vfin]
agrs : [[2 plur nom]]
comps_req : [subject]
comps_opt : [object dative])

{ENTER gestehen
lex(cats : [vfin]
agrs : [[1 plur nom][3 plur nom]]
comps_req : [subject vp_zu]
comps_opt : [dative])

{ENTER gestehen
lex(cats : [vfin]
agrs : [[3 sing nom]]
comps_req : [subject vp_zu]
comps_opt : [dative])

{ENTER gestanden
lex(cats : [vpast]
marks : [haben]
comps_opt : [object dative])

{ENTER gestanden
lex(cats : [vpast]
marks : [haben]
comps_req : [vp_zu]
comps_opt : [dative])

{ENTER tat
lex(cats : [n]
agrs : [[fem 3 sing]]
comps_opt : [det])

{ENTER taten
lex(cats : [n]
agrs : [[fem 3 plur]]
comps_opt : [det])}
\{ENTER begehen
  lex(cats : [vinf]
       comps_opt : [zu]
       comps_req : [object])\}
\{ENTER begehe
  lex(cats : [vfin]
       agrs : [[1 sing nom]]
       comps_req : [subject object])\}
\{ENTER begehst
  lex(cats : [vfin]
       agrs : [[2 sing nom]]
       comps_req : [subject object])\}
\{ENTER begeht
  lex(cats : [vfin]
       agrs : [[3 sing nom]]
       comps_req : [subject object])\}
\{ENTER begehen
  lex(cats : [vfin]
       agrs : [[[1 3] plur nom]]
       comps_req : [subject object])\}
\{ENTER begehet
  lex(cats : [vfin]
       agrs : [[2 plur nom]]
       comps_req : [subject object])\}
\{ENTER begangen
  lex(cats : [vpast]
       comps_req : [object]
       marks : [haben])\}
\{ENTER fahrrad
  lex(cats : [n]
       agrs : [[neut 3 sing [nom acc dat]]]
       comps_opt : [det])\}
\{ENTER fahrrads
  lex(cats : [n]
       agrs : [[neut 3 sing gen]]
       comps_opt : [det])\}
\{ENTER fahrräder
  lex(cats : [n]
       agrs : [[neut 3 plur [nom acc gen]]]
       comps_opt : [det])\}
\{ENTER fahrrädern
  lex(cats : [n]
       agrs : [[neut 3 plur dat]]
       comps_opt : [det])\}
\{ENTER reparieren
  lex(cats : [vinf]
       comps_opt : [zu object])\}
We now turn to the library support for composable lattices.

```oz
functor
import FS
export Base Flat Domain Cartesian Avm Set AccumulatingSet
define
  %% a lattice has top, bottom, and values. values can be represented
  %% either in decoded form (user oriented) or encoded form (e.g. an
  %% integer or a set value). encoded values can be combined using
  %% lub and glb operations.
  %%
  Top = {NewName}
  Bot = {NewName}
  class Base
    feat decode encode
      attr top bot
    meth init
      top <= Top
      bot <= Bot
```
self.decode = fun ($ I) {self decode(I $)} end
self.encode = fun ($ V) {self encode(V $)} end

meth top($) @top end
meth bot($) @bot end
meth lub(_, _, _) Base,notImplemented(lub ) end
meth glb(_, _, _) Base,notImplemented(glb ) end
meth encode(_, _) Base,notImplemented(encode) end
meth decode(_, _) Base,notImplemented(decode) end
meth notImplemented(Meth)
  {Exception.raiseError lattice(notImplemented(Meth))}
end

meth lubN(L $)
  {FoldL L fun ($ Accu X) {self lub(Accu X $)} end @bot}
end
meth glbN(L $)
  {FoldL L fun ($ Accu X) {self glb(Accu X $)} end @top}
end

%%% just 1 flat level with all ground values, possibly infinitely
%%% may of them
%%% class Flat from Base
meth lub(X Y $)
  if X==@bot then Y
  elseif Y==@bot then X
  elseif X==Y then X
  else @top end
end
meth glb(X Y $)
  if X==@top then Y
  elseif Y==@top then X
  elseif X==Y then X
  else @bot end
end

%%% a Domain is a Flat lattice with finitely many values
%%% identified by atoms and integers.
%%% class Domain from Flat
feat card values val2int int2val val2ints
meth init(L)
  Flat.init
  self.card = {Length L}
  self.values = L
  self.val2int = {List.toRecord o}
    {List.mapInd L fun ($ I) V#I-1 end}

```oz

self.int2val = {List.toRecord o (List.mapInd L fun {I V} I-1#V end)}
self.val2ints = {Record.map self.val2int fun {V} [V] end}
end
meth encode(V $) self.val2int.V end
meth decode(I $) self.int2val.I end
end

%%
fun (Map2 L1 L2 F)
case L1#L2 of nil#nil then nil
  [] (H1|T1)#(H2|T2) then {F H1 H2} |{Map2 T1 T2 F} end
end
%%
class Cartesian from Flat
  feat axes card val2ints
  meth init(L)
    Flat.init
    self.axes = L
    self.card = if L==nil then 0 else
      {FoldL L fun {N Axis} N*Axis.card end 1}
    end
    self.val2ints = {ByNeed fun ($) Cartesian,MkVal2Ints($) end}
  end
  meth encode(Vs $)
    {self encodeInts({Map2 Vs self.axes
      fun {V Axis} {Axis.encode V} end}) $}
  end
  meth encodeInts(Is $)
    Cartesian,EncodeInternal(Is self.axes 1 $)
  end
  meth EncodeInternal(Is Axes K $)
    case Is#Axes of nil#nil then 0
      [] (I|Is)#(Axis|Axes) then
        I*K + Cartesian,EncodeInternal(Is Axes K*Axis.card $)
      end
    end
  end
  meth decode(I $)
    Cartesian,DecodeInternal(I self.axes 1 _ $)
  end
  meth DecodeInternal(I Axes K R L)
    case Axes of nil then R=I L=nil
      [] Axis|Axes then R2 L2 in
        Cartesian,DecodeInternal(I Axes K*Axis.card R2 L2)
        R = R2 mod K
        L = {Axis.decode R2 div K}|L2
      end
    end
  end
  meth MkVal2Ints($)
Table = {Dictionary.new}
in{List.forAllInd self.axes proc {$ I D} {ForAll D.values proc {$ V} {Dictionary.put Table V Cartesian,Val2Ints(V I $)} end} end} {Dictionary.toRecord o Table}
end
meth Val2Ints(V I $)
VI = {{Nth self.axes I} encode(V $)}
L1 = {DomainsToListOfInts {List.drop self.axes I} [nil]}
L2 = {Map L1 fun {$ X} VI X end} L3 = {DomainsToListOfInts {List.take self.axes I-1} L2}
in{Map L3 fun {$ T} {self encodeInts(T $) end} end}
end
%%
fun {DomainsToListOfInts Ds Suffix}
case Ds of nil then Suffix
[] D|Ds then L={DomainsToListOfInts Ds Suffix}
in{FoldR {List.number 0 D.card-1 1} fun {$ I Accu} {FoldR L fun {$ X Accu} (I|X)|Accu end Accu} end nil}
end
end
%%
class Avm from Base
  feat domain
  meth init(R)
    Base.init
    self.domain = R
top <- (Record.map R fun {$ D} {D top($)} end)
bot <- (Record.map R fun {$ D} {D bot($)} end)
end
meth encode(S $)
  {Adjoin @top (Record.mapInd S fun {$ F V} {self.domain.F.encode V} end)}
end
meth decode(S $)
  {Record.mapInd S fun {$ F I} {self.domain.F.decode I} end}
end
meth glb(X Y $)  
   {Record.mapInd self.domain fun {F D} {D glb(X.F.Y.F $)}} end
end
meth lub(X Y $)  
   {Record.mapInd self.domain fun {F D} {D lub(X.F.Y.F $)}} end
end

%%
class Set from Base  
  feat domain full empty card StrictSpecs  
meth init(D strictSpecs:S<=false) Base,init  
      self.domain = D  
      self.card = {Pow 2 D.card}  
      self.empty = FS.value.empty  
      self.full = {FS.value.make 0#D.card-1}  
      top <- self.full  
      bot <- self.empty  
      self.StrictSpecs = S
end
meth glb(S1 S2 $) {FS.intersect S1 S2} end
meth lub(S1 S2 $) {FS.union S1 S2} end
meth encode(Spec $)  
   if self.StrictSpecs then {self strictEncode(Spec $)}  
   else {self encodeDisj(Spec $)} end
end
meth strictEncode(Spec $)  
   case Spec of compl(Spec) then  
      {FS.diff self.full {self strictEncode(Spec $)}}  
   else {FS.value.make {Map Spec self.domain.encode}} end
end
meth decode(S $) Set,decodeLowerBound(S $) end
meth decodeLowerBound(S $) Set,decodeList({FS.reflect.lowerBoundList S} $) end
meth decodeUpperBound(S $) Set,decodeList({FS.reflect.upperBoundList S} $) end
meth decodeList(L $)  
   {Map L self.domain.decode}
end
meth encodeDisj(Spec $)  
   if Spec=nil then self.empty  
   elseif (IsValue Spec) then  
      {FS.value.make self.domain.val2ints.Spec}  
   elseif (IsList Spec) then  
      {FoldL  
         {Map Spec fun {S} {self encodeConj(S $)}} end  
         fun {$ Accu S} {FS.union Accu S} end}
Chapter 20. Constraint-Based Dependency Parser: Implementation

We now turn to the implementation of the weighted set constraint

```
20.4 DG_Weight.oz

We now turn to the implementation of the weighted set constraint

```

```
20.5 DG_Selection.oz

We now turn to the implementation of the selection constraint

267a (DG_Selection.oz 267a)

functor
import FS W(weightC:WeightC) at 'DG_Weight.ozf'
export Select
define
  %% implements the selection constraint on sets
  %% L[I] = S
  proc {Select L I S1}
    L2 = {List.mapInd L
      fun ${ J S2}
        B=(I=#:J)
      in
        B#S2#{WeightC B S2}
      end}
    in
      %% first instantiate S1
      (FS.unionN {Map L2 fun ${ _#_=#S3} S3 end} S1)
      %% then reified.equal won’t block
      (ForAll L2
        proc ${ B#S2#=}
          B=<::(FS.reified.equal S1 S2}
        end)
      end
    end
end
end

20.6 DG_DTreeFrame.oz

We now turn to the implementation of the GUI responsible for displaying dependency parse trees (possibly partial).

267b (DG_DTreeFrame.oz 267b)

functor
import Tk
export 'class' : DtreeFrame
define
  DB = o{
    bg : ivory
    word : o(font : o(family:helvetica
      weight:bold
      size :12)
    color: black)
    label: o(font : o(family:fixed
      weight:normal
      size :12)
    color: black)
class Font from Tk.font
attr ascent descent
meth tkInit(...) = M
    Tk.font, M
    ascent <- {Tk.returnInt font(metrics self ascent:unit)}
    descent <- {Tk.returnInt font(metrics self descent:unit)}
end
meth measure(Text $)
    {Tk.returnInt font(measure self Text)}
end
meth getAscent($) @ascent end
meth getDescent($) @descent end
end

WordFont = {New Font tkInit(family:DB.word.font.family
    weight:DB.word.font.weight
    size :DB.word.font.size)}
LabelFont = {New Font tkInit(family:DB.label.font.family
    weight:DB.label.font.weight
    size :DB.label.font.size)}

class DtreeFrame from Tk.frame
attr canvas tag
meth tkInit(...) = M
    Tk.frame, M
    canvas <- {New Tk.canvas tkInit(parent:self bg:DB.bg)}
    tag <- {New Tk.canvasTag tkInit(parent:@canvas)}
    H = {New Tk.scrollbar tkInit(parent:self orient:horizontal)}
    V = {New Tk.scrollbar tkInit(parent:self orient:vertical)}
in
    (@canvas tk(configure scrollregion:q(0 0 0 0)))
    {Tk.addXScrollbar @canvas H}
    {Tk.addYScrollbar @canvas V}
    {Tk.batch [grid(rowconfigure self 0 weight:1)
        grid(columnconfigure self 0 weight:1)
        grid(@canvas row:0 column:0 sticky:nswe)
        grid(H row:1 column:0 sticky:we)
        grid(V row:0 column:1 sticky:ns)]}
end
meth clear (@canvas tk(delete @tag)) end
meth show(L)
    {self clear}
    % L is a list of elements of the form
    % o(string:WORD index:INDEX parent:INDEX2 label:LABEL)
    Nodes = {Map L}
fun ($ X)
  o(string:X.string label:X.label
    index:X.index parent:{CondSelect X parent unit}
    height:_ left:_
    width:{Max
      {WordFont measure(X.string $)}
      {LabelFont measure(X.label $)})})
end
NodesR = {List.toRecord o {Map Nodes fun {$ N} N.index#N end}}

% we compute how high to place each node
fun (Height N)
  if {IsDet N.height} then N.height
  elseif N.parent==unit then N.height=1
  else N.height=(1+(Height NodesR (N.parent))) end
end
MaxHeight = {FoldL Nodes fun {$ Accu N} {Max Accu (Height N)} end 0}
Top = MaxHeight*DB.vstep+DB.margin.top

% compute left coord of each node
ScrollWidth = {FoldL Nodes fun {$ Left N} N.left=Left Left+N.width+DB.hstep end
  DB.margin.left}=DB.hstep+DB.margin.left
ScrollHeight = Top+DB.margin.top+{WordFont getDescent($)}+{WordFont getAscent($)}
{@canvas tk(configure scrollregion:q(0 0 ScrollWidth ScrollHeight))}

% create slanted lines
{ForAll Nodes
  proc ($ N)
    if N.parent\=unit then
      P = NodesR. (N.parent)
      X1 = N.left+(N.width div 2)
      Y1 = N.height*DB.vstep
      X2 = P.left+(P.width div 2)
      Y2 = P.height*DB.vstep
      in
        {@canvas tk(create line X1 Y1 X2 Y2 fill:DB.sline.color
          width:DB.vline.width tags:@tag)}
    end
  end
}
% create text items, vertical lines, and labels
{ForAll Nodes
  proc ($ N)
    X = N.left+(N.width div 2)
    Y = N.height*DB.vstep
    in
      {@canvas tk(create line X Top X Y fill:DB.vline.color
        width:DB.vline.width tags:@tag)}
      {@canvas tk(create text X Top fill:DB.word.color
        width:DB.word.width tags:@tag)}

20.7 **DG_Demo.oz**

We now turn to the application that binds it all together

```oz
functor
  import
    Application Tk Explorer TkTools FS
    DtreeFrame at 'DG_DTreeFrame.ozf'
    Parser at 'DG_Parser.ozf'
  define
    fun {Normalize L}
      case L of nil then nil
            elseof & | T then {Normalize & | T}
            elseif X | T then Umlaut.X | (Normalize T)
            elseif X | T then X | (Normalize T) end
    end
    Umlaut = o("ä" ":ä" ":ö" "ü" "ß")
    ToLower = Char.toLower
    ToAtom = String.toAtom
    proc (Parse)
      S = (In tkReturnString(get $))
      L = case {Normalize {Map S ToLower}} of & | T then T
                       elseif T then T end
      Words = {Map {String.tokens L & } ToAtom}
      in
        try
          {Out clear}
          {Parser.checkWords Words}
          {Explorer.all
            (Parser.makeScript Words)}
          catch unknown(W) then
            {New TkTools.error
              tkInit(master:Top text:'unknown word: ' #W)_}
          end
        end
    end
    fun {MakeSpec N}
      o(index:N.index string:N.word
        parent:if {Not {IsDet N.mother}} then unit
              elsecase {FS.reflect.lowerBoundList N.mother}
                   of [I] then I else unit end
```

---

270a **(DG_Demo.oz)**

We now turn to the application that binds it all together

```oz
functor
  import
    Application Tk Explorer TkTools FS
    DtreeFrame at 'DG_DTreeFrame.ozf'
    Parser at 'DG_Parser.ozf'
  define
    fun {Normalize L}
      case L of nil then nil
            elseof & | T then {Normalize & | T}
            elseif X | T then Umlaut.X | (Normalize T)
            elseif X | T then X | (Normalize T) end
    end
    Umlaut = o("ä" ":ä" ":ö" "ü" "ß")
    ToLower = Char.toLower
    ToAtom = String.toAtom
    proc (Parse)
      S = (In tkReturnString(get $))
      L = case {Normalize {Map S ToLower}} of & | T then T
                       elseif T then T end
      Words = {Map {String.tokens L & } ToAtom}
      in
        try
          {Out clear}
          {Parser.checkWords Words}
          {Explorer.all
            (Parser.makeScript Words)}
          catch unknown(W) then
            {New TkTools.error
              tkInit(master:Top text:'unknown word: ' #W)_}
          end
        end
    end
    fun {MakeSpec N}
      o(index:N.index string:N.word
        parent:if {Not {IsDet N.mother}} then unit
              elsecase {FS.reflect.lowerBoundList N.mother}
                   of [I] then I else unit end
```
label: if {IsDet N.roleLabel} then N.roleLabel else '' end
end

proc {Show I R}
  (Out show({Map R MakeSpec}))
end

{Explorer.object add(information Show label:'DTree')}

Status
Top = {New Tk.toplevel tkInit}
Out = {New DtreeFrame.'class' tkInit(parent:Top)}
Bot = {New Tk.frame tkInit(parent:Top)}
In = {New Tk.entry tkInit(parent:Bot width:50)}
B1 = {New Tk.button tkInit(parent:Bot text:'Quit'
    action:proc($) Status=0 end)}
B2 = {New Tk.button tkInit(parent:Bot text:'Clear'
    action:proc($)
      {In tk(delete 0 'end')}
    end)}
B3 = {New Tk.button tkInit(parent:Bot text:'Parse'
    action:Parse)}
{In tkBind(event:'<Return>' action:Parse)}
{Tk.batch {grid(rowconfigure Top 0 weight:1)
  grid(columnconfigure Top 0 weight:1)
  grid(Out row:0 column:0 sticky:nswe)
  grid(Bot row:1 column:0 sticky:we)
  grid(columnconfigure Bot 1 weight:1)
  grid(B1 row:0 column:0 sticky:w)
  grid(In row:0 column:1 sticky:we)
  grid(B2 row:0 column:2 sticky:w)
  grid(B3 row:0 column:3 sticky:w)}}
{Application.exit Status}
end
Part VI

Concurrent Programming
Concurrent Programming

The goal of this chapter is twofold. One the one hand side, we would like to introduce some general ideas of concurrent programming and how to implement them in Oz. On the other hand side, we are giving the basics needed for the concurrent chart parser which will be presented in a later Chapter. For this purpose, we present an abstract model of a concurrent agenda.

21.1 What is Concurrency

Concurrency is an way to organise computation based on the notion of concurrent processes. Concurrency is well-known from operating systems like UNIX which support multi-tasking in order to administrate multiple windows each of which runs in its own process.

Concurrent processes can be executed in parallel whenever they are independent. Whether concurrent processes are executed in parallel matters for efficiency only. Concurrency is an interesting way in which to organise program and independent of parallel or interleaving execution. For instance, when you use a Sparc 10 with 2 processors then UNIX can run two processes in parallel even though you might not be aware of it.

Oz supports concurrent computation on a high level of abstraction. A process in Oz is called a thread. As in most other programming languages (except Oz1 or AKL), process creation is explicit in Oz2. In Oz processes communicate over logic variables residing in a shared constraint store. Logic variables play the same role as channels (for instance in CML or PICT).

21.2 Threads

A new concurrent thread in Oz is created by using the expression of the form thread ... end. For instance, during the execution of the following program two threads proceed concurrently:

```
declare
X {Browse X}
thread (Wait X) end
```

```
X = 1
```
Synchronization of threads is supported in Oz on the basis of logic variables. This is illustrated by the following program whose execution again needs two threads (compare chapter 8 of Seif Haridi’s tutorial on Oz which you can find at http://mozart.ps.unisb.de/documentation/tutorial/node8.html#chapter.concurrency).

```
declare X0 X1 X2 X3 in
{Browse [X0 X1 X2 X3]}

thread
    local Y0 Y1 Y2 Y3 in
    {Browse [Y0 Y1 Y2 Y3]}
    Y0 = X0+1
    Y1 = X1+Y0
    Y2 = X2+Y1
    Y3 = X3+Y2
    {Browse completed}
end
end
X0 = 0
X1 = 1
X2 = 2
X3 = 3
```

If you would like to observe the two threads in the execution you might be willing to use the Oz-debugger called Oscar which is available form the Mozart user interface (below the menue entry Oz).

### 21.3 Streams and Ports

A stream is an infinite list. Streams are important for concurrent programming. Concurrent processes may produce an infinite stream of items when running forever. Of course, only a finite part of a stream is known at every time point. Infiniteness accounts the unboundedness of its length only.

A port is a stateful data structure which gives access to a logic variable at the “end” of a partially known stream. A process can instantiate this variable by sending an item to the port.

```
declare S P
{Browse S}
P = {Port.new S}
{Browse S}
```

If you enter the following statements incrementally you will observe that S gets incrementally more defined.

```
{Port.send P 1}
{Port.send P 2}
```
A good example for illustrating ports and streams is to write a procedure which merges two streams in a fair manner. This means that each of the input streams is merged into the result at some time point even if one of them is infinite.

Fair merging of streams can be easily implemented in Oz which supports fair scheduling of threads. This means that at some time point, each runnable thread will be runned for a while. This also means that running threads have to be preempted in order to give a time slot to some other runnable thread.

```
declare
proc{Merge L1 L2 L}
  P={Port.new L}
  proc{Forward X}
    (Port.send P X)
  end
in
  thread {ForAll L1 proc{$ X} {Forward X} {Time.delay 100} end} end
  thread {ForAll L2 proc{$ X} {Forward X} {Time.delay 200} end} end
end
```

You can verify the fairness of this `Merge` procedure by entering the following lines of code:

```
declare
L1={MakeList 1000} {ForAll L1 proc{$ X} X=a end}
L2={MakeList 1000} {ForAll L2 proc{$ X} X=b end}

{Browse {Merge L1 L2})
```

Finally note that ports allows to programm client-server applications. A server simply posts items to the port of a stream whereas a clients read all items of a stream and accesses those that he is interested in. More about streams in concurrency can be found in chapter 8 of the Application Tutorial written by Denys Duchier, Leif Kornsteadt, and Christian Schulte, and in chapter 9 of Seif Haridi’s tutorial.

### 21.4 State and Indeterminism

We have said that a port is a stateful data structure. It is stateful since it gives access to the actual “end” of a stream which changes over time. It should be noted that state introduces indeterminism. This means that the result of a computation is not uniquely determined by its inputs. For instance, the final stream obtained by merging two other streams is indeterministic.

In general, state is important for concurrent programming. However, state renders concurrent programming also difficult, since the indeterminism it raises has to be controlled. For instance, if two processes compete for the same resource then only the process who comes first is served.
Chapter 21. Concurrent Programming

21.5 Exception Handling

Suppose, you want to kill a running process. But how can you do this? The answer is that you have to use exception handling. This is a control mechanism also known from SML and Java. A typical application of exception handling is to catch programming errors at runtime and to execute some code which keeps the system in a consistent state. But there is much more you can do with exception handling, for instance killing arbitrary processes.

The basic idea of exception handling is as follows: You can throw an exception in a thread by using the construct `raise` and catch this exception with the construct `try catch` end. For instance try the following:

```
declare
proc(Process) {Browse started} end
proc(Loop) {Loop} end

try
(Process) raise done end {Loop}
catch X then {Browse '*** ### ***'}
finally {Browse 'does not loop'}
end
```

For more about exception handling can be found in Chapter 5 of Seif Haridis Tutorial.

21.6 Mail Box Model

In the mail box model, each agent is equipped with his own mail box. In the simplest case, the agent is known to others only through its mailbox. A mailbox can be simply modelled by a port and an agent by a thread reading the stream of this port.

```
declare
MailBox % will be bound to a port
proc (MakeAgent Process MailBox)
    thread {ForAll {Port.new $ MailBox} Process} end
end
proc(Process X)
    {Browse 'new mail arrived'}
end
{MakeAgent Process MailBox}
{Port.send MailBox 1}
{Port.send MailBox 2}
```

Do you know how should a newgroup model work?
21.7 Atomicity of State Change

Changing the state of a CONCURRENT system is a dangerous operation. It would be harmless if changing state could be done by an atomic operation, i.e. by an operation that cannot be interrupted. Of course, no operation can be interrupted in a sequential system, i.e. state change is harmless in sequential systems. However, atomicity cannot be ensured in a concurrent system with fair scheduling of threads.

Let us assume a complex state-change-operation composed of several atomic changes. For instance, the method \texttt{change} in the following example provides a state change composed of at least two atomic operations:

\begin{verbatim}
279a (change 279a)≡
  meth change
  x←@y+1
  y←@y+1
\end{verbatim}

If the state-change is not atomic then there exists a time point where a partially changed state can be fetched by a concurrent actor, which can in turn apply state-change-operation. This may lead to a complex interleaving of partial state changes.

Consider for instance the following program where the attributes \( x \) and \( y \) of the object \( O \) should always have the same value. But this is not always true as you can observe when running the following program:

\begin{verbatim}
279b (inconsistent state change 279b)≡
  declare
  class C
  attr
    x:0
    y:0
  meth change
    x←@y+1 {Time.delay 500}
    y←@y+1
  end
  meth test
    try
      if @x==@y then skip
      else {Show 'oh weia'}
        %%%% raise done end
    end
  thread {self test} end
  {self change}
  {self test}
  catch _ then {Show stop}
  end
  end
end

O={New C test}
\end{verbatim}
21.8 Locks

The problem illustrated in the previous section raises a principle question: How can we gain the effect of atomic state change? Oz provides locks in order to solve this problem. A lock allows you to block the access of the state of an object until some sequence of operations is executed. For instance, you can repair the program of the above example as follows:

```oz
{consistent state change}

declare
class C
prop locking
attr
  x:0
  y:0
meth change
  lock
    x<-@y+1 {Time.delay 500}
    y<-@y+1
  end
end
meth test
  lock
  try
    if @x==@y then {Show ok}
    else {Show ‘oh weia’}
    raise done end
  end
thread {self test} end
{sself change}
{sself test}
catch _ then {Show stop}
end
end

O={New C test}
```

It should be noted carefully that the lock in the method `test` is also necessary. Otherwise the test for `@x==@y` might still return `false`. The problem is that unlocked methods can still access the state of a locked object. Only locked methods are prohibited to access the state of a locked object.

It might also be worth noting that the method `test` creates recursively locks inside of locks. Note that nested locks have the same effect as simple locks, i.e. for all statements E it holds that:

```
lock lock E end end   <=>   lock E end
```

In other words, Oz provides for so called thread-entrant locks (similar to Java).
21.9 Exercises

- Describe the newsgroup model informally and then implement it in Oz.

- Read Part II of the application tutorial and write an explanation about what a client server model (in German).

- Figure out how to kill a thread without exception handling and write down your solution (in German).
22.1 Concurrent Agenda Model

22.1.1 Idea

We next approach the concurrent agenda model. This model is fairly more complex model than the mail box but it also captures the essence of the concurrent chart parser of the next section.

An instance of the concurrent agenda model consists of a set of agents (simply threads) and a board. The agents communicate over the board only (and not directly with each other). The board is realized as a port (mail box) which gives access to a stream of items on the board.

In our application to parsing, the board will play the role of a chart and carries edges as items. The collection of all agents will model an agenda. Each agent models a dotted rule which tries to combine with all items of the board. Combination produces a new item on the board (an edge) or new agent (a dotted edge).

The most complex duty to be solved will be to detect the time point when no agent can become active any more, i.e. when the concurrent processing of the agenda is terminated. In order to do so, we distinguish two kinds of items, information items and control items, both of which can be posted to the board. Furthermore, an instance of the concurrent agenda model is equipped with a stateful agent, the watcher, which reads all control items and detects termination.

22.1.2 Description

Here is a more precise description of the model:

- All agents are stateless. Processing may add new agents to the system and new information and control items to the board.

- Each agent processes each information item on the board exactly once.

- After having processed an information item, an agent has to post the control item `itemProcessed` to the board.

- Before a new agent is created the control item `newAgent` is posted to the board.
• There is an initialization phase which create some agents and information items. When this phased is finish, the control item \textit{initialized} is posted to the board.

• When the whatcher detects termination then he send the control item \textit{stop} to the board.

• When the whatcher detects termination then he send the control item \textit{stop} to the board.

• If \textit{stop} is on the board then all agents terminate. The result consists in the finite subset of information items on the board.

22.1.3 Implementation

The functional procedure \textit{RunAgenda} inputs a description of an agenda, and starts an instance of the concurrent agenda model parametrized by this description.

284a \textit{(RunAgenda)} \equiv
\begin{verbatim}
fun{RunAgenda agenda{initialize:Initialize

processInfo:ProcessInfo}}

(Board)
(StartWatcher)
(FilterInfos)

in

{StartWatcher Board}
{Initialize Board}
{Board.post initialized}
{FilterInfos Board.items}
end
\end{verbatim}

First, a record \textit{Board} is created which provides access to a stream of items, allows to post new items and to start new agents connected to the board. All agents created process information the items on the board concurrently, according to the procedure \textit{ProcessInfo}. Second, the board is initialized with agents and items according to the procedure \textit{Initialize}. Third, the main thread calls the procedure \textit{FilterInfos} which computes the list of all information items on the board incrementally, and terminates once the control item \textit{stop} is posted to the board.

284b \textit{(FilterInfos)} \equiv
\begin{verbatim}
fun{FilterInfos Items}

case Items

of info(Info) | Is then Info |{FilterInfos Is}

elseof stop | _ then nil

elseof _ | Is then {FilterInfos Is}
end
end
\end{verbatim}

22.1.4 Board

The board can be realized as an instance of the mail box model where all agents connected to the board read all mails send. The variable \textit{Board} is bound to a record with
three features \textit{items}, \textit{post}, \textit{startAgent}. The feature \textit{items} gives access to the stream of all items on the board. The features \textit{post} and code \texttt{prolang=oz/startAgent/}
provide the expected facilities.

\begin{verbatim}
285a (Board 285a) ≡
  local
  Items
  local
    MailBox={Port.new Items}
  in
    proc {Post Item}
      {Port.send MailBox Item}
    end
  end
(StartAgent 285b)
  in
    Board = board(items:Items
    post:Post
    startAgent:StartAgent)
  end
\end{verbatim}

\subsection{22.1.5 Agents}

Every agent is realized as its own thread which is created when the agent is started, and killed when the exception \texttt{done} is raised. Killing a thread is necessary only in order to clean the space in the store occupied by the thread.

\begin{verbatim}
285b (StartAgent 285b) ≡
  local
    (Process 286a)
  in
    proc{StartAgent Agent}
      {Board.post newAgent}
    thread
      try {ForAll Board.items
        proc {$ Item}
          {Process Agent Item Board}
        end}
      catch done then skip end
    end
  (Show done)
end
\end{verbatim}

Every agent processes every item on the board by calling the procedure \texttt{Process}. When the control item \texttt{stop} is found, the exception \texttt{done} is raised and the thread of the agent is killed. All other control items are ignored. An information item is processed by calling the procedure \texttt{ProcessInfo} which is an extern parameter of the concurrent agenda model.
286a (Process 286a)≡

```
proc(Process Agent Item Board)
    case Item
        of stop then raise done end
        elseif info(Info) then
            NewObjects = {ProcessInfo Agent Info}
            in
                {ForAll NewObjects
                    proc($ O)
                        case O of agent(A) then {StartAgent A}
                        elseif info(I) then {Board.post info(I)}
                        end
                        end
                }
                {Board.post infoProcessed}
            else skip % other control items
            end
        end
    end

22.1.6 Termination Detection: The Watcher

The most difficult part of the implementation of the concurrent agenda model is to
decide about when the concurrent computation is terminated. As mentioned before,
termination is decided by a stateful agent that we call Watcher.

The duty of the Watcher is to count the number of tasks that have still to be performed
some agent in the concurrent system. A task is given by an agent and an information
item and means that the agent has to process the item.

The Watcher's state is implemented by a stateful Oz object with attributes for counting
numbers of agents, infos, tasks in the system. Furthermore, there is an attribute
initialize for memoizing whether the board has been initialized.

Suppose for instance that a new agent has been started. In this case, the number of
agents has to be increased by one but also the number of tasks has to be increased by
the tasks that the new agent has to perform. Hence, the number of task increases by the
number of information items that are actually on the board.

286b (WatcherState 286b)≡

```
class WatcherState
    attr
        agents:0
        infos:0
        tasks:0
        initialized: false
    meth init skip end
    meth initialized
        initialized<-true
        end
    meth infoProcessed
        tasks<-@tasks-1
        end
```
meth newAgent
    agents<-@agents+1
    tasks<-@tasks+@infos
end
meth info(...)
    infos<-@infos+1
    tasks<-@tasks+@agents
end
meth done($) 
    (And @initialized @tasks==0)
end
end

287a (StartWatcher 287a)
local 
  (WatcherState 286b)
 in
  proc(StartWatcher Board)
    local
      State={New WatcherState init}
    in
      proc(Process Item)
        {State Item}
        if (State done($))
          then
            {Board.post stop}
            raise done end
        else
          skip
        end
      end
    end
  in
    thread
      try {ForAll Board.items Process}
      catch done 
      then skip
      end
    end
  end
end

22.1.7 Correctness of the Watcher (Why Locks are NOT Needed)

One might wonder whether locks are needed for the implementation of the watcher. In fact, this is not the case. The reason is that the watcher changes its state fully sequentially. No concurrent actor can access the state of the watcher and thus only the watcher has access to its partially changed states. The watcher is written such that it completes a state change before it takes the next one. Hence the watcher treats state sequentially, such that locking is not needed.
If the watcher would process all items concurrently then the state of the watcher might in fact become inconsistent. Consider for instance a situation where a new information item and a new agent are created concurrently. Suppose that we had the following numbers before of agents, information items and tasks before:

<table>
<thead>
<tr>
<th>agents</th>
<th>infos</th>
<th>tasks</th>
<th>% before</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

3 3 7 % newAgent:
% agents<-@agents+1
% tasks<-@tasks+@infos

3 4 10 % info
% infos<-@infos+1
% tasks<-@tasks+@agents

The first operation of adding a new agent adds 4 new tasks and the second operation of adding a new information item adds 3 further tasks.

But now suppose that the operations of adding new agents or new tasks can be pre-empted. Then we could obtain the following sequence of state changes:

<table>
<thead>
<tr>
<th>agents</th>
<th>infos</th>
<th>tasks</th>
<th>% before</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

3 3 4 % newAgent:
% agents<-@agents+1 ...

3 4 4 % info
% infos<-@infos+1 ...

3 4 8 % newagent:
% ... tasks<-@tasks+@infos

3 4 11 % info:
% ... tasks<-@tasks+@agents

In fact this result is wrong, since there is one tasks to much, i.e. one task has been counted twice.

Hence, we have to ensure that state change of the watcher is atomic! In the implementation given, we have made the watcher sequential for this reason. Alternativly, one can use a concurrent watcher with locks.

### 22.2 A Concurrent Chart Parser

(288a) \( \text{Test the Concurrent Parser} \)

-declare (RunAgenda)
22.2. A Concurrent Chart Parser

```plaintext
(MakeAgenda)
local
  Dir='www.ps.uni-sb.de/~niehren/vorlesung/Programs/Functors/'
  URL='http://'#Dir
  [GrammarMod] = (Module.link [URL#'Grammar.ozf'])

in
  Rules = GrammarMod.rules
  Lexicon = GrammarMod.lexicon
end

fun{Parse Words}
  Agenda = {MakeAgenda Rules Lexicon Words}
  Edges = {RunAgenda Agenda}
  {Browse Edges}
  FullEdges = {Filter Edges fun{E}
    (And E.left==1
     E.right=={Length Words}+1}
    end}

  {Map FullEdges fun{E}
    (E.pred).head
    end}
end

/*
declare Words = [john saw the man with the telescope]
{Browse {Parse Words}}
*/
```

289a (MakeAgenda)

```plaintext
fun{MoveDot Rule}
  case Rule of rule(head:Head done:Done todo:Next|Todo dot:Index)
    then rule(head:Head done:Next|Done todo:Todo dot:Index+1) end
end

fun{ProcessInfo Agent Edge}% computes new a info item or a new agent
% whenever possible by applying the
% dotted rule of the Agent to the Edge

fun{Create Pred Right}
  if Agent.rule.todo.2==nil then
    info(edge(left: Agent.left
      right : Right
      cat : Agent.rule.head
      pred : Pred))
  else
    agent(edge(left:Agent.left
```
right:Right
rule:({MoveDot Agent.rule}
pred:Pred))

end
end

in
if Edge.left==Agent.right andthen
  Edge.cat==Agent.rule.todo.1
then
  {Map
    {Search.allP
      proc ($ Tree)
        {Agent.pred Tree}
        {Edge.pred}.head = Tree.args.(Agent.rule.dot)
        end 1 _
      fun($ Pred) (Create Pred Edge.right) end
    }else
      nil
    end
  end
end

proc(Initialize Board)
(List.forAllInd Words
  proc ($ I Word)
    {ForAll {Lexicon Word}
      proc ($ Cat#Fun)
        fun {Pred} root(head:({Fun} args:args) end
        in
          {Board.post info(edge(left:I right:I+1
            cat:Cat pred:Pred))} end
      }end)
    {ForAll Rules
      proc ($ Rule)
        %-- for each rule
      case Rule of rule(Head Body Fun) then
        N = {Length Body}
        R = rule(head:Head done:nil todo:Body dot:1)
        fun {Pred}
          Args = {Tuple.make args N}
        in
          root(head:({Fun Args} args:Args)
        end
      in
        {List.forAllInd Words
          proc ($ I _)
            %-- for each position
            {Board.startAgent edge(left:I right:I rule:R pred:Pred)}
          end
        end
    end}
22.3 Sample Grammar

Here is a sample grammar, packaged as a functor that exports **Lexicon** and **Rules**.

```
291a (Grammar Functor 291a)≡
  functor
  export
    Lexicon Rules
  define
    (Lexicon 291b)
    (Rules 292b)
  end

The lexicon is extremely simple but has nonetheless a **number** feature in order to illustrate agreement constraints.

291b (Lexicon 291b)≡

Lex =
  lex(
    the:
      [det  #fun {} det(phon:the number:_)]
    a:
      [det  #fun {} det(phon:a number:singular)]
    john:
      [np  #fun {} np(phon:john number:singular)]
    man:
      [n    #fun {} n(phon:man number:singular)]
    men:
      [n    #fun {} n(phon:men number:plural)]
    woman:
      [n    #fun {} n(phon:woman number:singular)]
    women:
      [n    #fun {} n(phon:women number:plural)]
    'with':
      [prep #fun {} prep(phon:'with')]
Verb and subject, as well as determiner and noun, must agree in number. We also copy the number feature upward whenever necessary.
22.4 The Complete Program

The complete program of the concurrent parser below runs under Mozart 3.0.0 only.

```plaintext
declare
fun{RunAgenda agenda(initialize:Initialize
  processInfo:ProcessInfo)}

local
  Items
local
  MailBox={Port.new Items}
in
  proc {Post Item}
    {Port.send MailBox Item}
  end
end

local
proc{Process Agent Item Board}
  case Item
  of stop then raise done end
  elseof info(Info) then
    NewObjects = {ProcessInfo Agent Info}
in
    {ForAll NewObjects
      proc($ O)
        case O of agent(A) then {StartAgent A}
        elseif info(I) then {Board.post info(I)}
        end
    end}
    {Board.post infoProcessed}
  else skip % other control items end
end
in
proc{StartAgent Agent}
  {Board.post newAgent}
thread
  try {ForAll Board.items
      proc {$ Item}
        {Process Agent Item Board}
    end}
catch done then skip end
end
{Show done} end
```
end
in
Board = board(items:Items
  post:Post
  startAgent:StartAgent)
end
local
class WatcherState
  attr
    agents:0
    infos:0
    tasks:0
    initialized:false
  meth init skip end
  meth initialized
    initialized<-true
  end
  meth infoProcessed
    tasks<-@tasks-1
  end
  meth newAgent
    agents<-@agents+1
    tasks<-@tasks+@infos
  end
  meth info(...)
    infos<-@infos+1
    tasks<-@tasks+@agents
  end
  meth done($)
    {And @initialized @tasks==0}
  end
end
in
proc(StartWatcher Board)
  local
    State={New WatcherState init}
in
  proc(Process Item)
    {State Item}
    if {State done($)}
      then
        {Board.post stop}
        raise done end
      else
        skip
      end
    end
end
in
thread
   try {ForAll Board.items Process}
   catch done
   then skip
   end
   end
fun{FilterInfos Items}
   case Items
      of info(Info) | Is then Info | {FilterInfos Is}
      elseof stop | _ then nil
      elseof _ | Is then {FilterInfos Is}
   end
   end
{StartWatcher Board}
{Initialize Board}
{Board.post initialized}
{FilterInfos Board.items}
end
fun{MakeAgenda Rules Lexicon Words}
fun{MoveDot Rule}
   case Rule of rule(head:Head done:Done todo:Next | Todo dot:Index)
      then rule(head:Head done:Next | Done todo:Todo dot:Index+1) end
   end
fun{ProcessInfo Agent Edge}% computes new a info item or a new agent
   % whenever possible by applying the
   % dotted rule of the Agent to the Edge
fun{Create Pred Right}
   if Agent.rule.todo.2 == nil then
      info(edge(left: Agent.left
               right : Right
               cat : Agent.rule.head
               pred : Pred))
   else
      agent(edge(left:Agent.left
               right:Right
               rule:{MoveDot Agent.rule}
               pred:Pred))
   end
   end
in
   if Edge.left == Agent.right andthen
      Edge.cat == Agent.rule.todo.1
   then
      {Map
      (Search.allP
   end
proc ($ Tree)
  (Agent.pred Tree)
  (Edge.pred).head = Tree.args.(Agent.rule.dot)
  end 1 _)
fun($ Pred) (Create Pred Edge.right) end
else
  nil
end
end

proc(Initialize Board)
(List.forAllInd Words
  proc ($ I Word)
    (ForAll {Lexicon Word}
      proc ($ Cat#Fun)
        fun (Pred) root(head:{Fun} args:args) end
      in
        (Board.post info(edge(left:I right:I+1
cat:Cat pred:Pred))) end)
    end)
  (ForAll Rules
    proc ($ Rule)
      %% -- for each rule
      case Rule of rule(Head Body Fun) then
        N = {Length Body}
        R = rule(head:Head done:nit todo:Body dot:1)
        fun (Pred)
          Args = {Tuple.make args N}
        in
          root(head:{Fun Args} args:Args)
        end
      in
        (List.forAllInd Words
          proc ($ I _)
            %% -- for each position
            (Board.startAgent edge(left:I right:I rule:R pred:Pred})
        end)
      end
    end)
  end
in
agenda(initialize:Initialize
  processInfo:ProcessInfo)
end
local
Dir='www.ps.uni-sb.de/~niehren/vorlesung/Programs/Functors/
URL='http://'#Dir
[GrammarMod] = {Module.link [URL#Grammar.ozf]}

```
in
  Rules = GrammarMod.rules
  Lexicon = GrammarMod.lexicon
end

fun(Parse Words)
  Agenda = {MakeAgenda Rules Lexicon Words}
  Edges = {RunAgenda Agenda}
  {Browse Edges}
  FullEdges = {Filter Edges fun($ E)
    (And E.left==1
      E.right==(Length Words)+1)
    end}
  in
  {Map FullEdges fun($ E)
    (E.pred).head
    end}
end

/*

declare Words = [john saw the man with the telescope]
{Browse {Parse Words}}

*/

If you want another Mozart version instead then you have to recompile the grammar functor and to use your new version. Here is the code of the grammar functor:

functor
export
  Lexicon Rules
define
  Lex =
  lex(
    the:
      [det #fun ($) det(phon:the number:_)_ end]
    a:
      [det #fun ($) det(phon:a number:singular) end]
    john:
      [np #fun ($) np(phon:john number:singular) end]
    man:
      [n #fun ($) n(phon:man number:singular) end]
    men:
      [n #fun ($) n(phon:men number:plural) end]
    woman:
      [n #fun ($) n(phon:woman number:singular) end]
    women:
      [n #fun ($) n(phon:women number:plural) end]
'with':
    [prep #fun ($) prep(phon:'with') end]

  telescope:
    [n #fun ($) np(phon:telescope number:singular) end]

  saw:
    [v #fun ($) v(phon:saw number:_ ) end]

  sees:
    [v #fun ($) v(phon:see number:singular) end]

  likes:
    [v #fun ($) v(phon:likes number:singular) end]

    n #fun ($) n(phon:likes number:plural ) end]

  like:
    [v #fun ($) v(phon:like number:plural) end]

fun (Lexicon Word) Lex.Word end

Rules =
  [rule{s [np vp]}
    fun ($ Args)
    
    Args.1^number=Args.2^number
    s(Args.1 Args.2)
    
   end)

  rule(np [det n]]
    fun ($ Args)
    
    Args.1^number=Args.2^number
    np(number:Args.2^number Args.1 Args.2)
    
   end)

  rule(np [n]]
    fun ($ Args)
    
    Args.1^number=plural
    np(number:plural Args.1)
    
   end)

  rule(np [np pp]]
    fun ($ Args)
    
    np(number:Args.1^number Args.1 Args.2)
    
   end)

  rule(vp [v np]]
    fun ($ Args)
    
    vp(number:Args.1^number Args.1 Args.2)
    
   end)

  rule(vp [vp pp]]
    fun ($ Args)
    
    vp(number:Args.1^number Args.1 Args.2)
    
   end)

  rule(pp [prep np]]
    fun ($ Args)
    
    pp(Args.1 Args.2)
    
   end)

  rule(vp [v]]
    fun ($ Args)
22.5 Exercises

- How can you raise an exception in a running thread? The assumption is that you have access to the name of the thread, and that the thread alone would run without raising an exception. (Search the documentation).

- Define ports as provided by Oz but based on first principle, i.e. by using objects in Oz.

- Add a new word to the lexicon of the ChoLi system.

- Change the board of the concurrent chart parser such that it contains an array of ports one for each position of the input sentence and also a port for control items.
Window Programming

23.1 Write a Chart Window yourself
Part VII

Programming Environment and the Internet
24.1 Modules and Lazy Loading

In Mozart, modules of the base environment are loaded on need only. For observing this, start a fresh Mozart system and execute:

```
{Browse FD}
```

If your Mozart is really fresh than the Browser displays

```
FD:Future
```

meaning that the variable `FD` will be bound to the FD-module once it is needed in the future. This mechanism is called lazy loading.

The idea of lazy loading is as follows: When the Mozart system comes up then not all modules are automatically loaded in order to speed up the start-up time. However, there already exists a propagator in the system, who knows a web-address or a directory from where to import the value of `FD`. Once the value of `FD` is needed, this propagator becomes active and binds `FD` to the FD-module.

There are several ways of expressing the need of a value. For instance,

```
{Wait FD}
```

waits until the future value of the variable `FD` is known and loads its value. In a fresh programming environment, the future value of `FD` is known to be the finite domain module. Thus `{Wait FD}` simply leads to loading the finite domain module. Another way of expressing the need of `FD` is by selection a feature from the module, such as by executing `FD.decl`.

For programming with functors, a programmer has to know the names of the standard modules he uses. For instance, the procedure `Browse` is imported by the module `Browser` of the base environment.

```
{Browser.browse 'Browse belongs module Browser'}
```

The procedure `Show` belongs to the module `System` of the base environment.

```
{System.show 'Show belongs to the module System'}
```

In Mozart, you can also write your own modules that are loaded lazily by using the syntax for describing functors.
24.2 Functors

A functor is a piece of compiled Oz-code. A functor can be stored and accessed from a web address.

Oz provides extra values for describing web-addresses (URL’s), so called virtual strings. Virtual strings are tuples composed from hashes and atoms. Virtual stings allow the programmer to compute with URL’s.

```oz
PSLab = 'http://www.ps.uni-sb.de'
Dir='~niehren/oz-course.html/Functors'
PickleVersion = 'Version.3.2'
fun{Conc URLs}
    case URLs
      of [U] then U
      [] U|Us then U#'/#{Conc Us}
    end
end
URL= {Conc [PSLab Dir PickleVersion]}
```

A functor is a value in Mozart. This means that a variable can have a functor as its value, i.e. the value of a variable can be a piece of compiled Oz-code.

```oz
Functors as Values
FileName='NewChart.ozf'
File = {Conc [URL FileName]}
NewChart_Functor={Pickle.load File}
```

The procedure `Pickle.load` returns a functor when given an URL. A more interesting thing you can do with a functor is to turn it into a module, which is loaded on need.

```oz
Functor To Module
[Mod1 Mod2] = {Module.link [{Conc [URL 'NewChart.ozf']}]
                 {Conc [URL 'ChartToWindow.ozf']})}
```

The procedure `Module.link` inputs a list of URL’s and output a list of modules (which are loaded on need).

```oz
Browse Modules
% the modules Mod1, Mod2 are not yet loaded
% even though their future value is defined
% {Wait X} waits until a future value
% for X gets defined (as for instance
% by {Module.link [X]}) and then loads
% this value.
```

```oz
declare
    (Functor To Module)
    (Browse [Mod1 Mod2])
/*
{ForAll [Mod1 Mod2] Wait} % {Wait X} waits until a future value
% for X gets defined (as for instance
% by {Module.link [X]}) and then loads
% this value.
*/```
24.3 Creating your Own Functors

Of course, there is also new syntax in Mozart for describing functors. Consider for instance the following functor definition:

```oz
functor
import
    Browser
    System
export
    NewChart
define
    Browse = Browser.browse
    Show = System.show
    % something reasonable has to be put here
    fun {NewChart} unit end
    {Browse 'function NewChart computed'}
    {Show 'function NewChart computed'}
end
```

Suppose that the content of the chunk `NewChart.oz` is stored in a file that is also called `NewChart.oz`. Then you can compile the functors definition in this file calling the Oz compile within you shell. This can be done by executing the following shell command:

```
ozc -c NewChart.oz -o NewChart.ozf
```

The shortcut `ozc` stand for Oz compiler. The option `-c NewChart.oz` means that the compiler is asked to compile the functor definition in the file `NewChart.oz`. The output option `-o NewChart.ozf` requires that the functor obtained is stored in the file `NewChart.ozf`. The output option can be omitted. In this case, a default file name is used storing the functor. The default name is computed from the input file name by replacing the suffix `.oz` by `.ozf`.

24.4 Foreign Functors: Graphical Output for a Chart

You can also use functors written by others without knowing their precise definition. For instance, let us use a functor for displaying a chart as a window.

```oz
declare
    (Functor To Module)
NewChart = Mod1.newChart
ChartToWindow = Mod2.chartToWindow
Phons = [john sees the man 'with' the telescope]
Edges = (Edges)
Chart=(NewChart Phons)
(ForAll Edges Chart.add)
Window = {ChartToWindow Chart}
```
## 24.5 Applets on the Internet

To be written

## 24.6 Program Collection

The chunk ‘Browse Modules’:

```oz
declare
PSLab = 'http://www.ps.uni-sb.de'
Dir='~niehren/oz-course.html/Functors'
PickleVersion = 'Version.3.2'
fun(Conc URLs)
  case URLs
  of [U] then U
  [] U|Us then U#'/'#(Conc Us)
end
end
URL= {Conc [PSLab Dir PickleVersion]}
[Mod1 Mod2] = {Module.link [{Conc [URL 'NewChart.ozf']}
  {Conc [URL 'ChartToWindow.ozf']}]}
(Browse [Mod1 Mod2]) % the modules Mod1, Mod2 are not yet loaded
  % even though their future value is defined
/*
{ForAll [Mod1 Mod2] Wait} % {Wait X} waits until a future value
  % for X gets defined (as for instance
  % by {Module.link [X]}) and then loads
  % this value.
*/
```

The chunk ‘Graphical Output of a Chart’:

```oz
declare
PSLab = 'http://www.ps.uni-sb.de'
Dir='~niehren/oz-course.html/Functors'
PickleVersion = 'Version.3.2'
fun(Conc URLs)
  case URLs
  of [U] then U
  [] U|Us then U#'/'#(Conc Us)
end
end
URL= {Conc [PSLab Dir PickleVersion]}
[Mod1 Mod2] = {Module.link [{Conc [URL 'NewChart.ozf']}
  {Conc [URL 'ChartToWindow.ozf']}]}
NewChart = Mod1.newChart
ChartToWindow = Mod2.chartToWindow
```
24.7 Exercises

- Turn your chart parser with named edges into a functor (a compiled piece of code). Make this functor available at the web by putting it into the file

    ~/public_html/oz-course/chart_parser.ozf

A solution of this exercise consists in a URL where your functor can be obtained from.

- Write a chart parser with window output in Oz. Use the chart parser functor of the previous excercise and the functor at the URL:

    http://www.ps.uni-sb.de/~niehren/oz-course.html/Functors/Version.3.2/ChartToWindow.ozf

Phons = [john sees the man ‘with’ the telescope]
Edges = [edge(begin:1 cat:s ‘end’:3 id:17 ids:[16 12])
    edge(begin:1 cat:s ‘end’:8 id:18 ids:[16 13])
    edge(begin:1 cat:s ‘end’:8 id:19 ids:[16 15])
    edge(begin:1 cat:s ‘end’:5 id:20 ids:[16 14])
    edge(begin:1 cat:np ‘end’:2 id:16 ids:[1])
    edge(begin:1 cat:pn ‘end’:2 id:1 ids:nil)
    edge(begin:2 cat:vp ‘end’:3 id:12 ids:[2])
    edge(begin:2 cat:vp ‘end’:8 id:13 ids:[2 11])
    edge(begin:2 cat:vp ‘end’:8 id:15 ids:[14 9])
    edge(begin:2 cat:vp ‘end’:5 id:14 ids:[2 10])
    edge(begin:2 cat:v ‘end’:3 id:2 ids:nil)
    edge(begin:3 cat:np ‘end’:8 id:11 ids:[10 9])
    edge(begin:3 cat:np ‘end’:5 id:10 ids:[3 4])
    edge(begin:3 cat:det ‘end’:4 id:3 ids:nil)
    edge(begin:4 cat:n ‘end’:5 id:4 ids:nil)
    edge(begin:5 cat:pp ‘end’:8 id:9 ids:[5 8])
    edge(begin:5 cat:prep ‘end’:6 id:5 ids:nil)
    edge(begin:6 cat:np ‘end’:8 id:8 ids:[6 7])
    edge(begin:6 cat:det ‘end’:7 id:6 ids:nil)
    edge(begin:7 cat:n ‘end’:8 id:7 ids:nil)]
Chart=(NewChart Phons)
(ForAll Edges Chart.add)

Window = {ChartToWindow Chart}
Debugger

To be written