Contents

1 Formal Study of Language: The Method of Fragments
1.1 Natural Language and Formal Languages ............................................ 1
1.2 Syntax, Semantics and Pragmatics ..................................................... 3
1.3 Grammars for Games ...................................................................... 6
1.4 A Tiny Fragment of English ............................................................... 8
1.5 The Syntax of Primary School Arithmetic ......................................... 9
1.6 The Syntax of Propositional Logic ..................................................... 12
1.7 The Syntax of SIMPL ....................................................................... 14
1.8 The Syntax of Predicate Logic ........................................................ 16
1.9 Further Reading ............................................................................. 19

2 Formal Semantics for Fragments
2.1 Semantics of Sea Battle .................................................................. 21
2.2 Semantics of Primary School Arithmetic ......................................... 23
2.3 Semantics of Propositional Logic ..................................................... 24
2.4 Semantics of Mastermind ................................................................. 26
2.5 Semantics of SIMPL ....................................................................... 27
2.6 Semantics of Predicate Logic ........................................................ 30
2.7 Semantics of Natural Language Fragments ....................................... 34
2.8 Further Reading ............................................................................. 35

3 Functional Programming with Haskell
3.1 Types in Grammar and Types in Programming ................................. 37
3.2 Using the Haskell Interpreter ............................................................ 39
3.3 Properties of Things, Characteristic Functions ................................. 42
3.4 List Types and List Comprehension .................................................. 44
3.5 The map and filter Functions ............................................................ 45
3.6 Application versus Abstraction; Function Composition ..................... 47
3.7 List Conjunction, Disjunction and Quantification .............................. 48
3.8 Identifiers in Haskell ..................................................................... 49
3.9 List Recursion with foldr and foldl ............................................... 49
3.10 Some Text Processing Examples ...................................................... 52
3.11 Further Reading ........................................................................... 59
# Contents

## 4 More on Type Theory

4.1 The Hierarchy of Type Domains ............................................. 61
4.2 Connection with Typing in Haskell ..................................... 65
4.3 Representing a Domain of Entities ..................................... 66
4.4 The Language of Typed Logic and Its Semantics .................... 69
4.5 Reducing Expressions of Typed Logic .................................. 74
4.6 Further Reading ............................................................... 78

## 5 Evaluating Predicate Logic in Haskell

5.1 Representing a Model for Predicate Logic ............................ 79
5.2 Databases as Models for Predicate Logic .............................. 82
5.3 Predicate Logic in Haskell ................................................ 84
5.4 Evaluating Formulas in Models ......................................... 87
5.5 Adding Functional Terms .................................................. 91
5.6 Further Reading ............................................................... 98

## 6 The Composition of Meaning in NL

6.1 Rules of the Game ............................................................. 99
6.2 Misleading Form and Logical Form .................................... 100
6.3 Meaning in Natural Language ............................................ 101
6.4 Datastructures for Syntax ................................................ 103
6.5 Semantic Interpretation .................................................... 104
6.6 Translation into Logical Form ............................................ 108
6.7 Ambiguity in Natural Language ........................................ 113
6.8 Further Reading ............................................................... 115

## 7 Theorem Proving for Predicate Logic

7.1 Substitution ................................................................. 117
7.2 Unification ................................................................. 120
7.3 Skolemization ............................................................... 125
7.4 Detecting Inconsistencies with Semantic Tableaux ............... 129
7.5 Automating Tableau Reasoning ....................................... 138
7.6 Further Reading ............................................................... 145

## 8 Parsing

8.1 Recognizing and Parsing Context Free Languages .................. 147
8.2 Parsers and Parser Combinators ....................................... 152
8.3 Features and Feature Structures ....................................... 157
8.4 Categories ................................................................. 163
8.5 Phrase Structure Grammar with Structured Symbols ............ 166
8.6 Lexical Database ........................................................... 166
8.7 Scanning and Lexical Lookup .......................................... 182
8.8 Parsing Categories ........................................................ 184
8.9 Unbounded Dependencies ............................................... 189
8.10 Boring Latex and System Stuff ....................................... 199
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.11</td>
<td>Further Reading</td>
<td>203</td>
</tr>
<tr>
<td>9</td>
<td>Handling Relations in Type Theory</td>
<td>205</td>
</tr>
<tr>
<td>9.1</td>
<td>Interpreting DP Lists</td>
<td>205</td>
</tr>
<tr>
<td>9.2</td>
<td>A Data Type for Relations</td>
<td>207</td>
</tr>
<tr>
<td>9.3</td>
<td>Boolean Algebras of Relations</td>
<td>210</td>
</tr>
<tr>
<td>9.4</td>
<td>The Need for Flexible Type Assignment</td>
<td>211</td>
</tr>
<tr>
<td>9.5</td>
<td>Lifting the Types of DP Interpretations</td>
<td>211</td>
</tr>
<tr>
<td>9.6</td>
<td>Scope Reversal of Quantifiers</td>
<td>213</td>
</tr>
<tr>
<td>9.7</td>
<td>Interpreting Verbs as Arbitrary Arity Relations</td>
<td>214</td>
</tr>
<tr>
<td>9.8</td>
<td>Interpretation of the Categories</td>
<td>216</td>
</tr>
<tr>
<td>9.9</td>
<td>Evaluation</td>
<td>221</td>
</tr>
<tr>
<td>9.10</td>
<td>Quantifier Scoping</td>
<td>222</td>
</tr>
<tr>
<td>9.11</td>
<td>Further Reading</td>
<td>224</td>
</tr>
<tr>
<td>10</td>
<td>A Closer Look at Lambda Calculus</td>
<td>225</td>
</tr>
<tr>
<td>10.1</td>
<td>Untyped Lambda Calculus — Datatype Declaration</td>
<td>225</td>
</tr>
<tr>
<td>10.2</td>
<td>Substitution</td>
<td>226</td>
</tr>
<tr>
<td>10.3</td>
<td>Notions of Reduction for Lambda Calculus</td>
<td>227</td>
</tr>
<tr>
<td>10.4</td>
<td>Normal Forms</td>
<td>228</td>
</tr>
<tr>
<td>10.5</td>
<td>Reductions to Normal Form</td>
<td>229</td>
</tr>
<tr>
<td>10.6</td>
<td>Some Well-known Combinators</td>
<td>231</td>
</tr>
<tr>
<td>10.7</td>
<td>Church Numerals</td>
<td>232</td>
</tr>
<tr>
<td>10.8</td>
<td>Pretty Printing of Lambda Terms</td>
<td>233</td>
</tr>
<tr>
<td>10.9</td>
<td>Further Reading</td>
<td>234</td>
</tr>
<tr>
<td>11</td>
<td>Types for Lambda Terms</td>
<td>235</td>
</tr>
<tr>
<td>11.1</td>
<td>Typing of Lambda Terms</td>
<td>235</td>
</tr>
<tr>
<td>11.2</td>
<td>Pretty Printing of Type Schemes</td>
<td>236</td>
</tr>
<tr>
<td>11.3</td>
<td>Occurs Check for Type Schemes</td>
<td>237</td>
</tr>
<tr>
<td>11.4</td>
<td>Typed Lambda terms</td>
<td>237</td>
</tr>
<tr>
<td>11.5</td>
<td>Pretty Printing of Typed Terms</td>
<td>237</td>
</tr>
<tr>
<td>11.6</td>
<td>Environments, Constraints, Substitutions and Judgements</td>
<td>239</td>
</tr>
<tr>
<td>11.7</td>
<td>Initialisation of Environments</td>
<td>239</td>
</tr>
<tr>
<td>11.8</td>
<td>Constraint Resolution by Means of Unification</td>
<td>239</td>
</tr>
<tr>
<td>11.9</td>
<td>Binding of Type Variables to Types</td>
<td>240</td>
</tr>
<tr>
<td>11.10</td>
<td>Type Inference Rules</td>
<td>241</td>
</tr>
<tr>
<td>11.11</td>
<td>Computation of Most General Type Schemes for Lambda Terms</td>
<td>242</td>
</tr>
<tr>
<td>11.12</td>
<td>Substitution for Typed Lambda Terms</td>
<td>244</td>
</tr>
<tr>
<td>11.13</td>
<td>From Untyped Terms to Typed Terms</td>
<td>245</td>
</tr>
<tr>
<td>11.14</td>
<td>Further Reading</td>
<td>247</td>
</tr>
</tbody>
</table>
CONTENTS

Index 309
Preface

This book on applications of logic in the semantic analysis of language pays the reader the compliment of not assuming anything about what s/he knows (in particular, no specific logical knowledge or experience with programming is presupposed), while making very flattering assumptions about his or her intelligence and interest in the subject matter.

The method used throughout in the book is the pursuit of logical questions and implementation issues occasioned by concrete examples of formally defined language fragments. At first, no distinction is made between formal and natural language: in the first chapter it is explained why. At the end of the text the reader should have acquired enough knowledge and skills for the development of (at least the semantic part of) fairly serious NL processing applications.

Throughout the text, abstract concepts are linked to concrete representations in the functional programming language Haskell, a language that is well suited for our purposes because it comes with a very easy to use interpreter, Hugs. Haskell compilers, interpreters and documentation are freely available from the Internet (at address http://www.haskell.org). Everything one has to know about programming in Haskell to understand the programs in the book is explained as we go along, but we do not cover every aspect of the language. For a further introduction to Haskell we refer the reader to [HFP96]. I do assume that the reader is able to retrieve the appropriate software from the internet, and that s/he is acquainted with the use of a text editor for the creation and modification of programming code.

The book is intended for linguists who want to know more about logic, including recent developments in modal and dynamic logic, and its applicability to their subject matter, and for logicians with a curiosity about applications of their subject to linguistics.

This text grew out of a series of lectures I gave at Uil-OTS in the fall of 2000, for a mixed audience of linguists and formal semanticists. This was extended with material from a tutorial at the LOLA7 Pecs Summer School of 2002, with the results of an implementation project for Theta Theory at Uil-OTS, Utrecht, in the fall of 2002, and with examples from a natural language technology course developed in collaboration with Michael Moortgat.

There is a web page devoted to this book, at address http://www.cwi.nl/~jve/cs, where the full code given in the various chapters can be found. Suggestions and comments on the text can be sent to me at email address jve@cwi.nl and will be very much appreciated.
Acknowledgements  Thanks to Herman Hendriks (for acting as an oracle on references), to Michael Moortgat (for pleasant cooperation in teaching various natural language technology courses), to Rick Nouwen (for joint work on implementations of pronominal reference algorithms), to Tanya Reinhart (for allowing me to co-teach a course with her on a subject I initially knew nothing about), to Albert Visser (for illuminating discussions on dynamic semantics, life and everything), to Willemijn Vermaat (for expert advice on \LaTeX{} matters and on automated tree drawing), to the course participants of my Uil-OTS courses, to the participants in the 2002 Pecs Summer School Tutorial, to Nicole Gregoire and Rian Schippers, members of the 2002 Theta Theory implementation group (for pleasant discussions), to Theo Janssen (for a number of useful suggestions concerning presentation) and last but not least to Shalom Lappin, for numerous update requests, inspiration, editorial suggestions and bug reports.

Amsterdam, ...
Chapter 1

Formal Study of Language: The Method of Fragments

Summary

This chapter starts with giving some good reasons for blurring the distinction between natural language and formal languages. Next, the formal study of language is defined and defended, and the distinction between syntax, semantics and pragmatics is explained informally. In the remainder of the chapter we present definitions of some example languages: languages for playing simple games, logical languages, programming languages, and fragments of natural language.

1.1 Natural Language and Formal Languages

English, Dutch, French and Russian are natural languages. The language of primary school arithmetic, the language of propositional logic and the programming language Prolog are formal languages. Esperanto is a case in between. The dividing line is not absolute, because one can always look at fragments of natural languages like Dutch or English and describe their grammar in a formal way. Because the line is drawn arbitrarily (by stipulative definition of the one who decides what is in the fragment and what is outside), such a fragment is in fact a formal language. Formal fragments of natural language are the object of study in computational linguistics. They can be studied with the techniques of formal language theory, but their analysis also uses insights from ‘ordinary’ linguistics. The distinction between formal language theory and the theory of natural language is much less absolute than some linguists take it to be.

Perhaps the main difference between traditional linguistics and computational linguistics is one of purpose. Computational linguistics aims at specifying ‘interesting’ fragments of natural language in a completely formal way, using the tools of logic and formal language theory. Such a formal specification may then lead to implementation of natural language front ends to databases and similar programs. The criteria for success here are: ‘does it work?’ and ‘does it do something useful?’ Traditional linguistics also uses formal rule systems, but for a quite different purpose.
The ultimate aim of Chomsky style Principles and Parameters/Minimalist Program linguistic research is to explain, in some suitable sense, the linguistic capacity of human beings. This is of course a very ambitious goal. It is also a very respectable goal. I take off my hat for anyone pursuing it.

All the same, one should not close one’s eyes to the fact that the aims of computational linguistics and of traditional linguistics are quite different. Although this book is first and foremost aimed at computational linguists, linguists from other persuasions may still find something of use in the following pages. In spite of the differences of purpose, the two brands of linguistics influence one another, and methods and tools are mutually applicable. But let it be noted that linguists from one tradition who brand the activity of the other tradition as not really linguistics at all (unfortunately, such linguists do exist) believe in a monopoly of linguistic purpose which the present author does not share at all, and which, in fact, he abhors. Quite frankly, this attitude is unfruitful and intolerant, and such opinions deserve to be ignored. No-one has the right to impose his views on the ultimate aim of linguistic (or indeed, any other kind of) research on anyone else.

In informal studies of language we always rely on implicit knowledge about language. Grammar books presuppose that the reader does have some informal grasp of what verbs, adverbs, nouns and pronouns are, of what it means to use a pronoun to refer to an antecedent, of what the distinction is between declarative sentences, imperatives and questions, and so on.

In formal studies of language we do this too, of course. I have to presuppose that you understand English, otherwise you wouldn’t be able to read these pages. Still, I will not rely on your insights into what it means to be able to read and understand these pages.

To study language in a formal way means to use conceptual tools from the formal sciences (mathematics, logic, theoretical computer science) in the analysis of natural language. Whether this is a legitimate enterprise depends on one’s aim. Views on the ultimate aims of linguistics happen to diverge widely. If the ultimate aim is to gain insight in how it is possible that human babies learn human languages with incredibly complex rule systems in such an incredibly short time, then it might be that formal methods are not the most appropriate tool. But then again, it is still very probable that logic has something to teach here.

Choice of aims is free, and when we decide to concern ourselves with the processing of language by means of computers, formal methods are indispensable. We cannot rely on informal insights of the language user, for computers do not learn languages the way humans do. In computational linguistics we cannot, therefore, make use of informal insights of the language user in the way in which most reference manuals on grammar presuppose that the reader knows something of the language already.

One of the things that make the study of language from a formal point of view so fascinating is that we can borrow insights from the formal sciences: mathematics, logic, theoretical computer science all make use of formal languages and there exists a wide body of knowledge concerning languages developed to formalize aspects of mathematical reasoning (classical first order logic, intuitionistic logic), concerning languages developed for various styles of programming (imperative languages like Pascal, functional languages like LISP, Scheme and Haskell, logic programming languages like Prolog), and concerning various languages for performing artificial
intelligence tasks (modal logic, epistemic logic).

In contrast with the situation with informal studies, we will forge the conceptual tools for our study ourselves, by means of formal definitions. These definitions are meant to be be taken quite literally. Time and again it will prove necessary to remind oneself of the definitions to understand the analysis. In this respect formal study is quite different from reading novels. The formal approach to language invites you to chew, and chew, and chew again, to arrive at proper digestion. If you are not yet used to read things written in formal style and think the explanations go too fast for you, the remedy is easy: just read more slowly.

1.2 Syntax, Semantics and Pragmatics

A basic trichotomy in the study of language is that between syntax, semantics and pragmatics. Roughly speaking, syntax concerns the form aspect of language, semantics its meaning aspect, and pragmatics its use. Developing insight about the semantics of a language presupposes knowledge of the syntax of that language; study of language use presupposes knowledge about both syntax and semantics. It is not surprising, then, that for lots of languages we know more about their syntax and semantics than about pragmatic aspects of their use. Here are approximate definitions of the three notions:

**Syntax** The study of strings and the structure imposed on them by grammars generating them.

**Semantics** The study of the relation between strings and their meanings, i.e., their relation with the extra-linguistic structure they are `about'.

**Pragmatics** The study of the use of meaningful strings to communicate about extra-linguistic structure in an interaction process between users of the language.

In a slogan: syntax studies Form, semantics studies Form + Content, and pragmatics studies Form + Content + Use. In this chapter we will examine what can be said about this trichotomy in concrete cases.

It is a matter of stipulation what is to be taken as ‘form’ in natural language. In the following we will concentrate on written language, more specifically on well-formed sentences. Thus, we will look at languages as sets of strings of symbols taken from some alphabet.

This choice makes it particularly easy to bridge the gap between formal and natural languages, for formal languages can also be viewed as sets of strings. The difference between natural and formal languages lies in the manner in which the sets of strings are given. In the case of formal languages, the language is given by stipulative definition: a string belongs to the language if it is produced by a grammar for that language, or recognized by a parsing algorithm for that language. A parsing algorithm for Pascal, say, can be wrong in the sense that it does not comply with the ISO standard for Pascal, but ultimately there is no right or wrong here: the way in which formal languages are given is a matter of definition.

In the case of natural languages, the situation is quite different. A proposed grammar formalism for a natural language (or for a fragment of a natural language) can be *wrong* in the sense that
it does not agree with the intuitions of the native speakers of the language. Whether a given string is a well-formed sentence of some language is a matter to be decided on the basis of the linguistic intuitions of the native speakers of that language.

Once a grammar for a language is given, we can associate parse trees with sentences. For natural languages, native speakers have intuitions about which constituents of a sentence belong together, and a natural language grammar will have to comply with these judgements. This might constitute an argument for considering constituent trees rather than strings as the forms that are given.

Still other choices are possible, too. Such choices depend on one’s aim. If one considers the problem of natural language recognition, one might consider strings of phonemes as the basic forms. If one wants to allow for uncertainty in language recognition, the basic forms become word lattices (with full proof and fool proof listed on a par as alternatives), or phoneme lattices (strings of phonemes involving the possibility of choice). Maybe we can attach probabilities to the choices. And so on. And so on.

A widespread prejudice against formal semantics for natural language is that it is just an exercise in typesetting. You explain the meaning of and in natural language by saying that the meaning of Johnny slept and Mary smiled equals A and B, where A is the meaning of Johnny slept and B is the meaning of Mary smiled. It looks like nothing is gained by explaining and as and.

The answer to this is, of course, that and refers to an operation one is assumed to have grasped already, namely the operation of taking a Boolean meet of two objects in a Boolean structure. Assuming that we know what a Boolean structure is, this is a real explanation, and not just a typesetting trick. On the other hand, if one is just learning about Boolean algebra by being exposed to an account of the semantics of propositional logic, it may well seem that nothing happens in the semantic explanation of propositional conjunction.

A well known story about the linguist Barbara Partee has it that she once ended a course on Montague semantics with the invitation to her students to ask questions. As this was the end of the course, they could ask any question, however vaguely related to the subject matter. A student: ‘Do you mean any question?’ Partee: ‘Yes, any question’. Student: ‘What is the meaning of life?’ Partee: ‘To answer that question we just have to see how Montague would treat the word life. He would translate it into his intensional logic as the constant life’, and he would use a cup operator to indicate that he was referring to the meaning, i.e., the extension in all possible worlds. So the answer to your question is: the meaning of life is ‘life’. Any other questions?’

The core of the problem is that it is hard to avoid reference to meaning by means of symbols anyway. Compare the process of explaining the meaning of the word bicycle to someone who doesn’t speak English. One way to explain the meaning of the word by drawing a picture of a bicycle. If your pupil is familiar with bicycles, the meaning will get across once he or she grasps that the drawing is just another way to refer to the actual meaning. The drawing itself is just another symbol. In just the same way, and is just another symbol for Boolean meet.

An alternative road to meaning is pointing, or direct demonstration, arguably the starting point of basic concept learning. One’s views of the principles of concept learning tend to be colored by philosophical bias, so let us not get carried away by speculation. Instead, here is an account of
first-hand experience. This is how Helen Keller, born deaf-mute and blind, learnt the meaning of the word *water* from her teacher, Miss Sullivan:

> We walked down the path to the well-house, attracted by the fragrance of the honeysuckle with which it was covered. Some one was drawing water and my teacher placed my hand under the spout. As the cool stream gushed over one hand she spelled into the other the word *water*; first slowly, then rapidly. I stood still, my whole attention fixed upon the motions of her fingers. Suddenly I felt a misty consciousness as of something forgotten—a thrill of returning thought; and somehow the mystery of language was revealed to me. I know then that “w-a-t-e-r” meant the wonderful cool something that was flowing over my hand. That living word awakened my soul, gave it light, hope, joy, set it free!

[Kel02, p. 34].

Explanations of meaning fall in two broad classes: meaning as *knowing how* and meaning as *knowing that*. If we say that Johnny doesn’t know the meaning of a good spanking, we refer to operational meaning. If Helen Keller writes that *water* means the wonderful cool something that was flowing over her hand, then she refers to meaning as reference, or denotational meaning.

Usually the two are intertwined. Suppose I ask directions to the Opera House. ‘Turn right at the next traffic light and you will see it in front of you’. Understanding the meaning of this involves being able to follow the directions (being able to work out which traffic light counts as the *next* one, which direction is *right*, and being able to recognize the building in front). Phrased in terms of ‘being able to work out ...’, ‘being able to recognize ...’, this is meaning as knowing how. Being able to understand the meaning of the directions also involves being able to distinguish correct directions from wrong directions. In other words, the directions classify situations, with me being positioned at some location in some town, facing in a particular direction: in some situations the directions provide a description which is true, in other situations a description which is false. This is denotational meaning.

Denotational meaning can be formalized as knowledge of the conditions for truth in situations. Operational meaning can be formalized as algorithms for performing (cognitive) actions. The operational semantics of *plus* in the expression *seven plus five* is the operation of adding two natural numbers; this operation can be given as a set of calculating instructions, or as a description of the workings of a calculating machine. The distinction between denotational and operational semantics is basic in computer science, and often a lot of work is involved in showing that the two kinds match.

Often operational meaning is more fine-grained than denotational meaning. For instance, the expressions *seven plus five*, *two plus ten* and *two times six* all refer to the natural number twelve, so their denotational meaning is the same. They have different operational meaning, for the recipe for adding seven and five is different from the recipe for multiplying two and six.

The purpose of communication can and do vary. The most lofty aim is to use language as a tool for collective truth finding. But language is also a tool for making your fellow citizens believe things, or for confusing your enemies. Even if two language users agree on non-deception, this does not exclude the use of irony.
A very important use of language, and one that we will be concerned with a lot in the following pages is as a tool for describing states of affairs and as a reasoning tool. As this is a use which formal languages and natural language have in common (formal languages often are designed as query tools or reasoning tools, or at least as tools for formal reconstructions of reasoning processes), it is a very natural focus point for the formal study of language.

In developing and studying formal languages, it has turned out useful to start with small and simple fragments, and then gradually extend those. This method of fragments, of taking things step by step, was first proposed for natural language by the logician and philosopher Richard Montague, who demonstrated that it is possible to give an account of puzzles of natural language meaning by studying toy approximations of natural language in a logical manner (see [Mon73]). In the remainder of this chapter we present definitions of some example languages.

### 1.3 Grammars for Games

Certain games can be played with a very limited linguistic repertoire. An example is Sea Battle, which is played by two people on two halves of a chess-board, with both halves made invisible to the opposite player. Here is a typical start situation for one half of the battlefield:

```
    a b c d e f g h
1: battleship     sub
2:                
3:                
4: destroyer      
```

A grammar for the language of the game is given below:

- **character** → a | b | c | d | e | f | g | h
- **digit** → 1 | 2 | 3 | 4
- **position** → character digit
- **attack** → position
- **reaction** → missed | hit | sunk | defeated
- **turn** → attack reaction

Legend: the rewrite symbols are given in **boldface**, the terminal symbols or strings in *italics*. The grammar rules consist of a rewrite symbol followed by → followed by a number of alternatives separated by |. What this means is that the rewrite symbol can produce any of the alternatives listed in the righthand side of the rule. Computer scientists often write ::= instead of →. The symbols ‘→’ for ‘produces’ and ‘|’ for ‘or’ are meta symbols of the grammar formalism. This way of specifying a grammar is called BNF (Backus Naur Form).
1.3. GRAMMARS FOR GAMES

Exercise 1.1 Show that the Sea Battle grammar can be simplified by getting rid of one rewrite symbol.

To be fully precise we have to say that the language defined by Sea Battle consists of all strings one can get by rewriting the start symbol turn, and repeating the rewrite process until only terminal symbols are left. An example should make this clear:

\[
\text{turn} \Rightarrow \text{attack reaction} \Rightarrow \text{attack missed} \Rightarrow \text{character digit missed} \Rightarrow \\
\Rightarrow b \text{ digit missed} \Rightarrow b2 \text{ missed}.
\]

Here \(\Rightarrow\) is short for `rewrites in one step to'.

The `rewrites in one step' relation is formally defined as the smallest relation satisfying the following. If \(A \rightarrow \alpha\) is a production of the grammar (i.e., \(\alpha\) is one of the alternatives listed at the righthand side of the rule with \(A\) at the lefthand side), and \(\beta \alpha \gamma\) is a list of symbols, with \(\beta, \gamma\) both consisting of zero or more terminals and rewrite symbols, then \(\beta A \gamma \Rightarrow \beta \alpha \gamma\).

The symbol for `rewrites in zero or more steps to' is `\(\Rightarrow^*\)'. We call this the reflexive transitive closure of \(\Rightarrow\). In the example we have: \(\text{turn} \Rightarrow^* b2 \text{ missed}\).

The linguistic expression \(b2 \text{ missed}\) does not fit the situation of the figure above, but this has nothing to do with the syntax of the language. It is a matter of semantics, and we will turn to that in the next chapter.

A possible extension of the grammar is a rule for a complete game between two players:

\[
\text{game} \rightarrow \text{turn} | \text{turn game}.
\]

Exercise 1.2 Revise the grammar in such a way that it is made explicit in the grammar rules that the game is over once one of the players is defeated.

The exercise illustrates the fact that semantic desiderata may influence the syntactic design. And this is not all. The rule `do not attack the same position twice', for instance, is not a part of syntax or semantics, but of pragmatics: whoever plays the game in order to win will comply with it.

Another very simple language is the language of Mastermind:

\[
\begin{align*}
\text{color} & \rightarrow \text{red} | \text{yellow} | \text{blue} | \text{green} \\
\text{answer} & \rightarrow \text{black} | \text{white} | \text{blank} \\
\text{guess} & \rightarrow \text{color color color color} \\
\text{reaction} & \rightarrow \text{answer answer answer answer} \\
\text{turn} & \rightarrow \text{guess reaction} \\
\text{game} & \rightarrow \text{turn} | \text{turn game}
\end{align*}
\]

Exercise 1.3 Revise the grammar in order to guarantee that a game has at most four turns.
Exercise 1.4 Revise the grammar so as to ensure that the game can be played without ‘blank’ as a possible answer. A reaction should now consist of at most four times ‘black’ or ‘white’.

Exercise 1.5 Write your own grammars for ‘telephone chess’ and ‘bingo’.

All these example grammars (except for the one from Exercise (refFSOL.FourLim) have the property that they generate infinite languages.

Exercise 1.6 Check this.

If one is lenient enough one might consider these example languages as fragments of English. Conversely, it is also very simple to write a grammar for a language which has English contained in it as a fragment:

\[
\begin{align*}
\text{character} & \rightarrow A | \cdots | Z | a | \cdots | z | - | . | ? | ! | ; | : \\
\text{string} & \rightarrow \text{character} \mid \text{character string}
\end{align*}
\]

The set of all strings, according to this definition, contains the set of sentences of written English. Note, by the way, that we have tacitly introduced \( \cdots \) as a new abbreviation. This is strictly for human consumption: a computer would not have a clue how to read this. But we all have learnt the alphabet at primary school, and the abbreviation relies on this common knowledge. It is extremely tedious to write prose meant for human consumption without tacitly relying on such background knowledge. In computer programming, one has to make all these shortcuts explicit, of course. (This may be why people with a talent for human relationships—and a skill for seeking common ground—find it often difficult to relate to computers, and vice versa.)

Exercise 1.7 Give a grammar for strings which also generates the empty string.

1.4 A Tiny Fragment of English

Suppose we wish to write grammar rules for sentences of English like the following:

1.1 Some woman smiled.

1.2 No woman respected every man that smiled.

1.3 Every woman that some man respected smiled.

1.4 Every man respected the woman that Bill loved.

What we need is a rule for the subject predicate structure of sentences, a rule for the internal structure of noun phrases, a rule for common nouns with or without relative clauses, and that is just about it. The following grammar copes with the examples:
1.5. THE SYNTAX OF PRIMARY SCHOOL ARITHMETIC

This is very basic and much too crude, of course, but it should give an idea of what a grammar for a fragment of English might look like.

Exercise 1.8 Extend this fragment with adjectives happy and friendly.

Exercise 1.9 Extend the fragment with preposition phrases, in such a way that the sentence Mary saw the boy with the binoculars is generated in two structurally different ways, while there is only one parse tree for Mary saw Bill with the binoculars.

Exercise 1.10 Extend the fragment with complex relative clauses, where the relativized clause can be a sentence conjunction. The fragment should generate, among other things, the sentence: A man that Mary loved and Bill respected smiled. What problems do you encounter?

1.5 The Syntax of Primary School Arithmetic

To move on to our next example, here is a grammar for primary school additions:

\[
\begin{align*}
S & \rightarrow \text{NP VP} \\
\text{NP} & \rightarrow \text{Mary} | \text{Bill} | \text{DET CN} | \text{DET RCN}.
\end{align*}
\]

\[
\begin{align*}
\text{DET} & \rightarrow \text{every} | \text{some} | \text{no} | \text{the} \\
\text{CN} & \rightarrow \text{man} | \text{woman} | \text{boy} \\
\text{RCN} & \rightarrow \text{CN that VP} | \text{CN that NP TV} \\
\text{VP} & \rightarrow \text{laughed} | \text{smiled} | \text{TV NP} \\
\text{TV} & \rightarrow \text{loved} | \text{respected}
\end{align*}
\]

Exercise 1.11 Explain the difference between znumeral and numeral.

The grammar for additions differs from earlier examples in the fact that it generates ambiguities: the same end result can be derived in different ways. An example is the number 50 + 4 + 93, for we have:

\[
\text{number} \Rightarrow \text{number} + \text{number} \Rightarrow^{*} 50 + \text{number} \Rightarrow
\]
\[
\Rightarrow 50 + \text{number} + \text{number} \Rightarrow^{*} 50 + 4 + \text{number} \Rightarrow^{*} 50 + 4 + 93.
\]
There is a natural correspondence between these two derivations and the following two structure trees or parse trees:

![Structure Trees](image)

In this particular case the ambiguity is harmless, because addition of positive numbers (which is the intended semantics for this language fragment) has the following property: \( m + (n + k) = (m + n) + k \). We can express this fact by saying that the operation of addition is *associative*.

In general structurally different derivations (i.e., derivations corresponding to different structure trees) will get linked to different meanings. Compare the difference between the following:

![Parse Trees](image)

This situation also occurs in primary school arithmetic, if we focus on different operations. The following replacement of the rule for *number* extends the language with multiplications:

\[
\text{number} \rightarrow \text{numeral} \mid (\text{number} + \text{number}) \mid (\text{number} \times \text{number})
\]

This generates for example \((2 \times (3 + 4))\) and \(((2 \times 3) + 4)\). The brackets fix the structure, for without parentheses \(2 + 3 \times 4\) would have been ambiguous.

Another way to get rid of unwanted structural ambiguities is by means of *operator precedence rules*. If we agree that multiplication has priority over addition, then this fixes the parse tree of \(2 + 3 \times 4\) as:
1.5. THE SYNTAX OF PRIMARY SCHOOL ARITHMETIC

In cases where this second parse is intended, it should be forced by inserting suitable parentheses. Allowing for this possibility complicates the grammar, for we now get:

\[
\begin{align*}
\text{pdigit} & \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\text{digit} & \rightarrow 0 \mid \text{pdigit} \\
\text{znumeral} & \rightarrow \text{digit} \mid \text{digit} \ \text{znumeral} \\
\text{numeral} & \rightarrow \text{digit} \mid \text{pdigit} \ \text{znumeral} \\
\text{product} & \rightarrow \text{numeral} \mid \text{numeral} \ \text{x product} \\
\text{sum} & \rightarrow \text{product} + \text{product} \mid \text{product} + \text{sum} \\
\text{number} & \rightarrow \text{product} \mid \text{sum} \mid (\text{number} \ \text{x number}) \mid (\text{number} + \text{number}) \\
\text{expression} & \rightarrow \text{number} = \text{number}
\end{align*}
\]

Note how the priority rule about the order in which products and sums are to be constructed gets reflected in the choice of rewriting symbols.

Note also that in order to understand this grammar specification one has to know that ‘(’ and ‘)’ are object language symbols, while ‘|’ and ‘→’ belong to the metalanguage used for the specification. Nothing deep is going on here, it is all just a matter of being explicit about notational conventions.

**Exercise 1.12** Give structure trees for the following expressions generated by this grammar:

\[
\begin{align*}
234 + 34 \times 13 + 5 + 7 \\
(234 + 34) \times 13 + 5 + 7 \\
(234 + 34 \times 13) + 5 + 7
\end{align*}
\]

A grammar is *ambiguous* if there is at least one expression in the language generated by it which has more than one structure tree.

**Exercise 1.13** Show that the example grammar is not ambiguous. (Hint: If you don’t know how to approach this problem, look at Proposition 1.16 below for inspiration.)

**Exercise 1.14** Write a grammar for primary school arithmetic with positive numbers, + and ×, where × has priority over + and where no expression has superfluous brackets. For instance, \((2 \times 3 + 4)\) and \((2 \times 3) + 4\) are not part of the language.
1.6 The Syntax of Propositional Logic

Our next example is a grammar for propositional logic, where the atomic propositions are indicated with \( p, q, r, p', q', r', p'', q'', r'' \) ....

\[
\begin{align*}
\text{atom} & \rightarrow p \mid q \mid r \mid \text{atom}' \\
\text{formula} & \rightarrow \text{atom} \mid \neg \text{formula} \mid (\text{formula} \land \text{formula}) \mid (\text{formula} \lor \text{formula}).
\end{align*}
\]

Here \( \neg \) is the symbol for not, \( \land \) the symbol for and, and \( \lor \) the symbol for or.

Some formulas generated by this grammar are \( \neg \neg p'' \), \( ((p \lor p') \land p'') \), \( (p \land (p' \land p'')) \). Again, the brackets force unique readability. Without those brackets, one wouldn’t know whether to read \( p \land p' \lor p'' \) as

\[
\begin{align*}
\text{formula} & \rightarrow \text{formula} \land \text{formula} \land \text{formula} \\
& \rightarrow \text{formula} \lor \text{formula} \lor \text{formula} \\
& \rightarrow \text{formula} \lor \text{formula} \land \text{formula} \\
& \rightarrow p \lor p' \lor p''.
\end{align*}
\]

This structural ambiguity does affect the meaning, just as for the English sentence ‘She was young and beautiful or depraved’.

The formal definition of the language of propositional logic makes it possible to reason about the properties of the language in a formal way. For this we use the following principle:

**Theorem 1.15 (Principle of Structural Induction)** Every formula of propositional logic has property \( P \) provided

- **basic step** every atom has property \( P \),
- **induction step** if \( F \) has property \( P \), then so does \( \neg F \), if \( F_1 \) and \( F_2 \) have property \( P \), then so do \( (F_1 \land F_2) \) and \( (F_1 \lor F_2) \).

**Proof.** Let \( P' \) be the set of all propositional formulas with property \( P \). Then the basic step and the induction step together ensure that all propositional formulas are members of \( P' \). But this just means that every propositional formula has property \( P \).

This can be used as follows:

**Proposition 1.16** Every propositional formula has equal numbers of left and right parentheses.
**Proof.** Basic step: atoms do not have parentheses at all, so it holds vacuously they have the same numbers of left and right parentheses.

Induction step: If $F$ has the same numbers of left and right parentheses, then so does $\neg F$, for no parentheses are added. If $F_1$ and $F_2$ both have equal numbers of left and right parentheses, then so do $(F_1 \land F_2)$ and $(F_1 \lor F_2)$, for both numbers are increased by one.

Of course, as a proposition this is not very interesting. The interest is in the proof method. The same holds for the next result.

**Proposition 1.17** Propositional formulas are uniquely readable (have only one parse tree).

**Proof.** Basic step: If $F$ is an atom, then $F$ has only one parse tree.

Induction step: If $F$ has only one parse tree, then $\neg F$ has only one parse tree too. If $F_1$ and $F_2$ have only one parse tree each, then the same holds for $(F_1 \land F_2)$ and $(F_1 \lor F_2)$.

Note the use of so-called *meta-symbols* $F_1$ and $F_2$ in the reasoning above. Meta-symbols come in handy whenever one wants to talk about the general form of a formula. A formula of the form $(F_1 \land F_2)$ is called a *conjunction*, a formula of the form $(F_1 \lor F_2)$ a *disjunction*, and a formula of the form $\neg F$ a *negated formula*.

Sometimes the ‘official’ way of writing formulas is a bit clumsy. We will usually write $p_2$ for $p''$, $q_3$ for $q'''$, and so on.

**Exercise 1.18** Give a grammar rule which generates the following set of propositional atoms:

$$\{p, q, r, p_1, q_1, r_1, p_2, q_2, r_2, \ldots\}.$$ 

Also, we will often omit parentheses when this does not result in ambiguity. Propositional conjunction is associative, so $(p \land (q \land r))$ means the same as $((p \land q) \land r)$. Therefore, there is no harm in writing $(p \land q \land r)$ or even $p \land q \land r$ for both. Disjunction is also associative, so we will also write $(p \lor (q \lor r))$ and $((p \lor q) \lor r)$ as $p \lor q \lor r$. And so on, for longer conjunctions and disjunctions.

While we are at it, it is useful to introduce the following abbreviations. We will write $F_1 \rightarrow F_2$ for $\neg(F_1 \land \neg F_2)$. A formula of the form $F_1 \rightarrow F_2$ is called an *implication*. Also, we write $F_1 \leftrightarrow F_2$ for $(F_1 \rightarrow F_2) \land (F_2 \rightarrow F_1)$. A formula of the form $F_1 \leftrightarrow F_2$ is called an *equivalence*.

It is possible to write formulas for propositional logic which are uniquely readable, but without using parentheses. The solution is to use prefix (or: Polish) notation:

$$\text{pformula} \quad \rightarrow \quad \text{atom} \mid \neg \text{pformula} \mid \land \text{pformula} \mid \lor \text{pformula} \mid \text{pformula} \mid \lor \text{pformula} \mid \lor \text{pformula}.$$

This definition generates, for example, $\neg\neg\neg p$, $\land pqr$, $\land p \lor qr$.

**Exercise 1.19** Use the principle of structural induction to prove that the formulas of Polish propositional logic are uniquely readable.
It is also possible to put the operators after their operands; this gets us reverse Polish notation or postfix notation:

\[
\text{rpformula} \rightarrow \text{atom} \mid \text{rpformula} \Downarrow \\
\mid \text{rpformula} \text{ rpformula} \land \\
\mid \text{rpformula} \text{ rpformula} \lor .
\]

**Exercise 1.20** Write a grammar for primary school arithmetic with positive numbers, + and ×, in reverse Polish notation, that is to say with the operators after their operands. An example of a number written in this notation is: 5 7 + 3 ×.

Reverse Polish notation (also called Hewlett Packard notation) is very suitable for computer implementation, because the natural implementation uses a so-called data stack: a stack of memory cells, with two procedures push for putting new data on top of the stack, and pop for removing the top item from the stack. A stack structure is the simplest possible organization of computer memory.

Assume we have a procedure read(X) which reads the next numeral or next occurrence of + or × from the input, a function eof for indicating the end of the input, and operations + and * for calculating the sum or product of two arguments. Then the following procedure reads and evaluates a number in reverse Polish notation, and puts the result on the data stack. In fact, we have not yet given a formal definition of the imperative programming language for the next procedure definition, but we will do so below.

\[
\text{WHILE NOT eof DO} \\
\text{BEGIN} \\
\text{read(X);} \\
\text{IF X = '+' THEN BEGIN pop(X); pop(Y); push(X+Y) END} \\
\text{ELSE IF X = '*' THEN BEGIN pop(X); pop(Y); push(X*Y) END} \\
\text{ELSE push X} \\
\text{END}
\]

**Exercise 1.21** Give a step by step account of how this procedure deals with the input string 5 7 + 3 ×.

### 1.7 The Syntax of SIMPL

Before we move to our next example, it is useful to introduce an extension of BNF notation, the so-called EBNF (Extended Backus Naur Form) format. EBNF has extra meta-symbols \{ and \}, with the convention that \{A\} indicates that A may occur zero or more times. EBNF admits the following definition for strings:

\[
\text{string} \rightarrow \text{character} \{\text{character}\}.
\]

If the empty string is also admitted, this can be shortened to:

\[
\text{string} \rightarrow \{\text{character}\}.
\]
Of course, the new definition affects the format of the structure trees, e.g., for the string \textit{abcd}:

\begin{center}
\begin{tikzpicture}
  \node (root) {string};
  \node (left) [below left] {character string};
  \node (right) [below right] {character string};
  \node (left1) [below left] {character};
  \node (left2) [below right] {character string};
  \node (right1) [below left] {character};
  \node (right2) [below right] {character};
  \node (left21) [below left] {a};
  \node (left22) [below right] {b};
  \node (right21) [below left] {c};
  \node (right22) [below right] {d};
\end{tikzpicture}
\end{center}

versus

\begin{center}
\begin{tikzpicture}
  \node (root) {string};
  \node (left) [below left] {character character character character character};
  \node (right) [below right] {a b c d};
\end{tikzpicture}
\end{center}

It can be proved that every EBNF grammar can be transformed into a BNF grammar without the creation of new ambiguities.

Here is an EBNF grammar for SIMPL, a Simple Imperative Programming Language:

\begin{verbatim}
statement  →  variable := term |
           begin statement { ; statement } end |
           (if expression then statement) |
           (if expression then statement else statement) |
           while expression do statement
alfachar  →  a | ⋯ | z
string    →  alfachar {alfachar}
digit     →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
numeral   →  digit {digit}
variable  →  string
term      →  numeral | variable | ¬numeral | ¬variable |
           (term × term) | (term + term) | (term − term)
atom      →  string
expression →  atom | term = term | term < term |
            term > term | term ≤ term | term ≥ term |
            ¬expression | (expression ∧ expression) | (expression ∨ expression)
\end{verbatim}

As you can see, this mini programming language contains elements from the language of primary school arithmetic (the structure of terms) as well as from the language of propositional logic (the structure of expressions). The basic expressions are now not just atomic propositions (strings in this language), but also equalities and inequalities between terms.

The intuitive meaning of \textit{x := \ldots} is: store \ldots at memory location \textit{x}. If \ldots is a number, then that number itself gets stored, if it is a variable, then the value of that variable gets copied and
stored, if it is a complex expression, then the value of that expression gets computed and stored. The intuitive meanings of \( \text{if} \ldots \text{then} \ldots, \text{if} \ldots \text{then} \ldots \text{else} \ldots, \) and \( \text{while} \ldots \text{do} \ldots \) are as you would expect. The meaning of the semicolon is: do the stated things in the given order. \( < \) means is less than, \( > \) means is greater than, \( \leq \) means is less than or equal to, \( \geq \) means is greater than or equal to. This should be enough for an intuitive understanding of what programs in this language mean (further details are given in the next chapter).

Here are some programs (or procedures, or statements) of the mini programming language we have just defined:

\[
x := x + 1
\]

This program increases the value of the number stored at location \( x \) by one.

\[
(\text{if } x < 0 \text{ then } x := -x)
\]

This program does nothing if the value turns out to be positive, and reverses the sign of \( x \) otherwise.

\[
\text{begin } z := x; x := y; y := z \text{ end}
\]

This program uses \( z \) as an auxiliary ‘container’ to swap the contents of \( x \) and \( y \).

\[
\text{begin } i := 0; f := 1; \text{ while } i < x \text{ do begin } i := i + 1; f := (i \times f) \text{ end end.}
\]

**Exercise 1.22** Assume that this last program gets executed, and at the start state of the execution \( x \) has some positive value. Try to figure out what the value of \( f \) will be after the program has terminated.

More about the semantics of this fragment of imperative programming in the next chapter.

**Exercise 1.23** Check the procedure given above for evaluating reverse Polish arithmetical expressions. Which ‘violations’ of the formal grammar that we have just given do you see?

**Exercise 1.24** Give a structure tree for the following program:

\[
(\text{if } x < 0 \text{ then (if } y < 0 \text{ then } z := -y \text{ else } z := y)).
\]

**Exercise 1.25** Explain why the program from the previous exercise would be ambiguous without brackets.

### 1.8 The Syntax of Predicate Logic

The language of predicate logic can also be defined quite easily in BNF format. To simplify matters we assume that the predicates have arity less or equal to 3; higher arities are hardly ever needed, anyway. What this means is that we can refer to one-placed relations or properties (such as the property of being happy, or being an even number), to two-placed relations (such as the relation ‘greater than’ between numbers, or the relation ‘older than’ between people), and
to three-placed relations (such as the relation of giving, which has a subject, an object and a recipient or indirect object), but not to relations with more than three arguments.

Here is the definition of the language of predicate logic with infinitely many one-placed predicates, infinitely many two-placed predicates, and infinitely many three-placed predicates:

constant → a | b | c | constant'
variable → x | y | z | variable'
term → constant | variable
onepl-pred → P | onepl-pred'
twopl-pred → R | twopl-pred'
threepl-pred → S | threepl-pred'
atom → onepl-pred term | twopl-pred term term | threepl-pred term term term
formula → atom | term = term | ¬formula | (formula ∧ formula) | (formula ∨ formula) | ∀ variable formula | ∃ variable formula.

The following strings are examples of formulas of this predicate logical language:

\[ P' x'' \]
\[ ¬P' c \]
\[ (P x ∧ P' x'') \]
\[ ((P x ∧ P' x'') ∨ R x x') \]
\[ ∀ x R x x \]
\[ ∃ x (R x x' ∧ S x y x) \]
\[ ∀ x ∀ x' ¬(x = x' ∧ ¬ x' = x) \]

**Exercise 1.26** Prove that the formulas of this language have the property of unique readability.

The symbol \( \forall \) is the symbol for universal quantification; it means ‘for all’. The symbol \( ∃ \) expresses existential quantification; it means ‘there exists’. Thus, \( ∀ x R x x \) means that everything bears the \( R \) relation to itself, and \( ∀ x ∃ x' R x x' \) means that for everything there is something which is R-ed by that first thing. Incidentally, this paraphrase shows that the variable notation is quite handy. More information about the semantics of first order predicate logic will be given in the next chapter.

**Exercise 1.27** Give a formal definition of the predicate logical language with one one-placed predicate \( P \), and two two-placed predicates \( R \) and \( S \).

**Exercise 1.28** Give a BNF grammar for a predicate logical language with infinitely many predicate letters for every finite arity. (Hint: use \( "P", "P', "P'', \ldots \) for the set of three-placed predicate letters, and so on.)
Again, it is convenient to be lenient about notation. We will use
\[ a, b, c, a_1, b_1, c_1, a_2, b_2, c_2, \ldots \]
for individual constants,
\[ x, y, z, x_1, y_1, z_1, x_2, y_2, z_2, \ldots \]
for individual variables, and
\[ P, R, S, P_1, R_1, S_1, P_2, R_2, S_2, \ldots \]
for predicate symbols, plus the conventions for omitting parentheses when these are not essential for disambiguation. We will also continue to use the abbreviations \( F_1 \rightarrow F_2 \) for formulas of the form \( \neg(F_1 \land \neg F_2) \), and \( F_1 \leftrightarrow F_2 \) for formulas of the form \( (F_1 \rightarrow F_2) \land (F_2 \rightarrow F_1) \).

In talking about predicate logical formulas we want to be able to distinguish between the variable occurrences that are bound by a quantifier in that formula and the occurrences that are not. Binding is a syntactic notion, and is defined in terms of properties of syntax trees.

Node \( a \) dominates node \( b \) in a syntax tree if there is a downward path through the tree leading from \( a \) to \( b \) (with \( a = b \) in the limit case). Node \( a \) in a syntax tree c-commands (i.e., ‘constituent’-commands) node \( b \) in that tree if the first branching node above \( a \) dominates \( b \).

An occurrence of \( \forall x \) (or \( \exists x \)) in a formula \( F \) binds an occurrence of \( x \) in \( F \) if in the syntax tree for \( F \) the occurrence \( \forall x \) (or \( \exists x \)) c-commands \( x \). In such a case we also say that that occurrence of \( x \) is in the scope of the quantifier occurrence \( \forall x \) or \( \exists x \). An occurrence of \( x \) is bound in \( F \) is there is some quantifier occurrence that binds it, it is free otherwise.

Exercise 1.29 Give the bound occurrences of \( x \) in the formula \( (P x \land \exists x (R xy \lor S x y z)) \).

A predicate logical formula is called open if it contains at least one variable occurrence which is free, it is called closed otherwise. A closed predicate logical formula is also called a predicate logical sentence. Thus, \( (P x \land \exists x R x x) \) is an open formula, but \( \exists x (P x \land \exists x R x x) \) is a sentence.

One would hope that predicate logic is powerful enough to express the equations of primary school arithmetic. The above definition does not quite give us that expressive power, however, for we have only allowed for simple terms (terms that are either constants or variables), while an arithmetical term like \( ((5 + 3) \times 4) \) is a complex expression. Just adding the term forming operators \(+\) and \(\times\) is not the best solution, for we want our predicate logical language to be general enough to also apply to other domains than the domain of natural numbers. Instead of introducing a name for the ‘smaller than’ relation, we have introduced names for arbitrary two-placed relations. Just so: instead of introducing a name for the sum and product operations, we introduce name for arbitrary operations. Such a name is called a function constant. For convenience, we limit ourselves to one-placed, two-placed and three-placed function constants.
The definition of terms now becomes:

\[
\begin{align*}
\text{onepl-fct} & \rightarrow f \mid \text{onepl-fct}' \\
\text{twopl-fct} & \rightarrow g \mid \text{twopl-fct}' \\
\text{threepl-fct} & \rightarrow h \mid \text{threepl-fct}' \\
\text{term} & \rightarrow \text{constant} \mid \text{variable} \mid \text{onepl-fct}(\text{term}) \mid \\
& \quad \text{twopl-fct}(\text{term}, \text{term}) \mid \text{threepl-fct}(\text{term}, \text{term}, \text{term})
\end{align*}
\]

This allows for complex terms like the following:

\[
\begin{align*}
f''(x) \\
f'(g(c, x), f(x)) \\
g''(f'''(x), h(x, x, f(x)))
\end{align*}
\]

Again, we will write \(f_1, f_2, \ldots, g_1, g_2, \ldots, h_1, h_2, \ldots\) for \(f', f'', \ldots, g', g'', \ldots, h', h'', \ldots\).

**Exercise 1.30** Give a parse tree for the term \(g_2(f_3(x), h(x, x, f(x)))\).

A term \(t\) is *free for variable \(v\) in formula \(F\)* if every free occurrence of \(v\) in \(F\) can be replaced by \(t\) without any of the variables in \(t\) getting bound. For example, \(x_1\) is free for \(x\) in \((Px \rightarrow \forall x Px)\), but the same term is not free for \(x\) in \((\forall x_1 R xx_1 \rightarrow \forall x R xx)\). Similarly, \(g(x, x_1)\) is not free for \(x\) in \((\forall x_1 R xx_1 \rightarrow \forall x R xx)\).

Being free for a variable \(v\) in a formula is an important notion, for a term with this property can be substituted for free occurrences of \(v\) without an unintended change of meaning. If we replace the free occurrence of \(x\) in the open formula \((\forall x_1 R xx_1 \rightarrow \forall x R xx)\) by \(x_1\) we get a closed formula: \((\forall x_1 R xx_1 x_1 \rightarrow \forall x R xx)\). The result of the substitution is a sentence: a variable which originally was free has got captured. This kind of accident we would like to avoid.

If a term \(t\) is free for a variable \(v\) in a formula \(F\) we can substitute \(t\) for all free occurrences of \(v\) in \(F\) without worrying about variables in \(t\) getting bound by one of the quantifiers in \(F\). If, on the other hand, \(t\) is not free for \(v\) in \(F\), we can always rename the bound variables in \(F\) to ensure that substitution of \(t\) for \(v\) in \(F\) has the right meaning. Although \(g(x', c)\) is not free for \(x\) in \((\forall x_1 R xx' \rightarrow \forall x R xx)\), the term is free for \(x\) in \((\forall x_2 R xx_2 \rightarrow \forall x R xx)\), which is a so-called *alphabetic variant* of the original formula. An alphabetic variant of a formula is a formula which only differs from the original in the fact that it uses different bound variables.

### 1.9 Further Reading

On the relation between Chomsky-style linguistics and computational linguistics, see the concluding chapter of Morrill [Mor94]. A classic textbook on the application of logical methods to the study of language is Reichenbach [Rei47]. A beautiful book on historical and philological aspects of the evolution of word meaning is C.S. Lewis [Lew60]. The viewpoint that the real mystery in the understanding of natural language lies in the way human beings grasp meanings
of single words can be found in Plato’s dialogue *Cratylus*. See also Percy [Per83]. Helen Keller [Kel02] gives a moving account of the process of awakening by grasping the concept of language.

Formal language theory is the theory behind syntactic fragments. See, e.g., [R91]. Formal language theory started with [Cho59], where the so-called Chomsky hierarchy of formal languages is defined. Chomsky himself drew the conclusion that the formalism of context free grammars (the key formalism underlying the fragments above) is inadequate for a full description of natural language, as natural languages exhibit phenomena that are provably beyond the generative capacity of context free grammar rules. This motivated Chomsky’s adoption of transformation rules. While formal theory of grammar blossomed into a main field of theoretical computer science, sadly, Chomskyan linguistics parted company. A formal study of generative grammar formalisms is taken up in [Rog98].
Chapter 2

Formal Semantics for Fragments

Summary

The semantics of all the example languages from the previous chapter, except for the case of the natural language fragment, is given in formal detail, with a word or two about pragmatics thrown in here and there for good measure.

2.1 Semantics of Sea Battle

To specify the semantics of Sea Battle, we should first know something about the extra-linguistic reality that the game is about, and in the second place we should find out how to describe the way in which the linguistic utterances which are part of the game relate to that reality.

The reality of Sea Battle is made up of a set of states of the game-board. Such a state is a board with positions of ships indicated on it, and crosses indicating the fields on the board that were under attack so far in the game. For example:

```
    a b c d e f g h
  +---+---+---+---+---+---+---+---+
  4  |  |  |  |  |  |  |  |  |
  +---+---+---+---+---+---+---+---+
  3  X |  |   |   |  |  |  | X |
  +---+---+---+---+---+---+---+---+
  2  |  |  |  |  |  |  |  | X |
  +---+---+---+---+---+---+---+---+
  1  |  |  |  |  |  |  |  |  |
  +---+---+---+---+---+---+---+---+
```

The meaning of an attack is given by a state change. For example, the meaning of the attack b4 for the state given above can be specified as follows:
The complete meaning of the attack $b4$ is given by the function from ‘input’ states to ‘output’ states that for every input state of the game board returns that state adorned with an extra $X$ at position $b4$ as output state.

The Sea Battle reactions missed, hit, sunk, and defeated are interpreted as follows. Given a state and the position of the last attack, any of these reactions gives the value ‘true’ (= 1) or false (= 0). We can express this as a function $F$ from $S \times P \times R$ to $\{0, 1\}$, where $S$ is the set of states of the game, $P$ the set of positions, and $R$ the set of reactions (so $S \times P \times R$ is the set of triples consisting of a state, a position and a reaction).

\[
F(s, p, \text{missed}) = \begin{cases} 
1 & \text{if there is no ship at position } p \text{ in state } s, \\
0 & \text{otherwise}.
\end{cases}
\]

\[
F(s, p, \text{hit}) = \begin{cases} 
1 & \text{if there is a ship at position } p \text{ in state } s, \\
0 & \text{otherwise}.
\end{cases}
\]

\[
F(s, p, \text{sunk}) = \begin{cases} 
1 & \text{if there is a ship at position } p \text{ in state } s \\
& \text{which is covered by } X \text{'s everywhere but at } p, \\
0 & \text{otherwise}.
\end{cases}
\]

\[
F(s, p, \text{defeated}) = \begin{cases} 
1 & \text{if there is a ship at position } p \text{ in state } s \\
& \text{which is covered by } X \text{'s everywhere but at } p, \\
& \text{while all other ships are also covered everywhere by } X \text{'s,} \\
0 & \text{otherwise}.
\end{cases}
\]

Note that it follows from these semantic clauses that $F(s, p, \text{defeated}) = 1$ implies $F(s, p, \text{sunk}) = 1$, which in turn implies $F(s, p, \text{hit}) = 1$. It follows from this that the game rule ‘always react to an attack with a truthful reaction’ does not in all cases determine the appropriate reaction.

Game rules like this are within the province of pragmatics. The rudiments of a systematic treatment of this field were given by Grice [Gri75], with his observation that language users should try to fit their on and their partners’ contributions to spoken communication into a coherent whole by means of the following global principles:

**Cooperation** Try to adjust every contribution to the spoken communication to what is perceived at that point in the interaction as the common goal of the communication.

**Quality** Aspire to truthfulness. Do not say anything that you do not believe to be true. Do not say anything for which you have inadequate support.

**Quantity** Be as explicit as the situation requires, no more, no less.

**Mode of Expression** Don’t be obscure, don’t be ambiguous, aspire to conciseness and clarity.
As you can see, this is prescriptive in the vague way that all exhortations to behave well tend to be. Also, there is some overlap between the principles. Finally, the principles employ notions like truth and informativeness which are key notions of semantics. In this sense, pragmatics does indeed presuppose semantics. But however vague they are, Grice’s maxims are applicable to the Sea Battle example. The semantics of Sea Battle gives a very precise measure for informativeness of reactions. For why is the reaction sunk (when it is true) more informative than the reaction hit (also true in that same situation)? Because the reaction sunk applies truthfully to a strictly smaller set of game states, in other words, because this reaction excludes a larger number of states. The same holds for the relation between defeated and sunk. It follows from the principle of cooperation, based on insight in the goal of the utterance of letting the other player know where s/he stands, that in a situation where sunk applies the reaction hit is inappropriate. This same conclusion can also be argued for on the basis of the maxim of quantity, which states that the game demands maximally informative reactions at every stage.

Exercise 2.1 What more can be said about the pragmatics of Sea Battle in terms of Grice’s maxims?

Exercise 2.2 Describe the semantics and pragmatics of Mastermind and Bingo.

2.2 Semantics of Primary School Arithmetic

The non-linguistic structure that the language of primary school arithmetic is about is given by the set $\mathbb{N}^+$ of positive whole numbers. Intuitively, this set is given, in some appropriate abstract way, in the realm of thought. What kind of entities numbers are is a question which we will not try to answer here. Too deep and too philosophical! It is important, however, to see that there is a difference between the number 5 and the numeral ‘5’ which we use to denote this number. The easiest way to see the difference is by noticing that ‘V’ (in Roman notation) or ‘101’ (in binary Arabic notation) denote that same number. The number 5 is an object, and the numeral ‘5’ is a name for that object.

If you prefer, you can also have an explicit construction of the natural numbers. Here is the recipe of John von Neumann:

$$0 = \emptyset, 1 = \{\emptyset\}, 2 = \{\emptyset, \{\emptyset\}\}, \ldots, n + 1 = n \cup \{n\}.$$ 

Of course, this presupposes that you believe in the existence of the empty set $\emptyset$ and of the operations $\cup$ and $\{\cdot\}$. We will assume that we know what is meant by $\mathbb{N}^+ = \{1, 2, 3, \ldots\}$.

Assume that $\oplus$ is the addition operation on the set $\mathbb{N}^+$. In other words, if $\hat{n}$ and $\hat{m}$ are elements of $\mathbb{N}^+$, then $\hat{n} \oplus \hat{m}$ is the result of the addition of $\hat{n}$ and $\hat{m}$. Again: $\oplus$ is an operation on $\mathbb{N}^+$, and ‘+’ is a name of that operation. Similarly, $\otimes$ is the multiplication operation on $\mathbb{N}^+$, and ‘×’ is a name of that operation. The difference between $\oplus$, $\otimes$ on one hand and ‘+’, ‘×’ on the other, is completely analogous to that between the object 5 and the name ‘5’.

We will now link the language of primary school arithmetic to the structure $\langle \mathbb{N}^+, \oplus, \otimes \rangle$, using a
value function $V$ from arithmetical terms to the numbers they denote:

\[
V(n) = n \quad \text{if } n \text{ is a numeral},
\]
\[
V(n + m) = V(n) \oplus V(m)
\]
\[
V(n \times m) = V(n) \otimes V(m).
\]

Should you think that nothing happens here, then this is a sure sign that you have not yet fully grasped the distinction between language and structure, between names and the things they denote.

Finally, here is the semantics of arithmetical expressions:

\[
\llbracket n = m \rrbracket = \begin{cases} 1 & \text{if } V(n) \text{ is identical to } V(m) \\ 0 & \text{otherwise.} \end{cases}
\]

This expresses the difference between correct and incorrect solutions to arithmetical expressions:

\[
\llbracket 1 + 2 = 4 \rrbracket = 0,
\]

for 4 is an incorrect solution to the problem $1 + 2 = \ldots$.

**Exercise 2.3** Try to apply the Gricean maxims in the pragmatics of ‘writing down solutions to primary school arithmetic problems’.

### 2.3 Semantics of Propositional Logic

The first issue we must address for the semantics of propositional logic is: what are the extralinguistic structures that propositional logical formulas are about? Our answer to this: pieces of information about the truth or falsity of atomic propositions. This answer is encoded in so-called valuations, functions from the set $P$ of proposition letters to the set $\{0,1\}$ of truth values. If $V$ is such a valuation, then $V$ can be extended to a function $V^+$ from the set of all propositional formulas to the set $\{0,1\}$. In the following definition of $V^+$, and further on in the book, we use the abbreviation *iff* for *if and only if*.

\[
V^+(p) = V(p) \quad \text{for all } p \in P,
\]
\[
V^+(-F) = 1 \quad \text{iff } V^+(F) = 0,
\]
\[
V^+(F_1 \land F_2) = 1 \quad \text{iff } V^+(F_1) = V^+(F_2) = 1
\]
\[
V^+(F_1 \lor F_2) = 1 \quad \text{iff } V^+(F_1) = 1 \text{ or } V^+(F_2) = 1
\]

**Exercise 2.4** Give the semantic clause for $V^+(F_1 \rightarrow F_2)$ which follows from the definition above plus the convention for the use of the $\rightarrow$ sign.

**Exercise 2.5** Give the semantic clause for $V^+(F_1 \iff F_2)$ which follows from the definition above plus the convention for the use of the $\iff$ sign.

**Exercise 2.6** Let $V$ be given by $p \mapsto 0, q \mapsto 1, r \mapsto 1$. Give the $V^+$ values of the following formulas:

\[
(\neg p \lor p), (p \land \neg p), \neg(p \lor \neg r), \neg(p \land \neg r), (p \lor (q \land r)).
\]
Another way of presenting the semantics of the propositional connectives is by means of *truth tables* which specify how the truth value of a complex formula is calculated from the truth values of its components.

<table>
<thead>
<tr>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$\neg F_1$</th>
<th>$F_1 \land F_2$</th>
<th>$F_1 \lor F_2$</th>
<th>$F_1 \rightarrow F_2$</th>
<th>$F_1 \leftrightarrow F_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

It is not difficult to see that there are formulas $F$ for which the $V^+$ value does not depend on the $V$ value of $F$. Formulas $F$ with the property that $V^+(F) = 1$ for *any* $V$ are called *tautologies*. Notation for ‘$F$ is a tautology’: $\models F$. Formulas $F$ with the property that $V^+(F) = 0$ for *any* $V$ are called *contradictions*.

**Exercise 2.7** Which of the following are true:

- $\models p \lor \neg q$.
- $\models p \land \neg p$.
- $\models p \lor \neg p$.
- $\models p \rightarrow (p \lor q)$.
- $\models (p \lor q) \rightarrow p$.

**Exercise 2.8** Explain why the negation of a tautology is always a contradiction, and vice versa.

A formula $F$ is called *satisfiable* if there is at least one valuation $V$ with $V^+(F) = 1$. A formula $F$ is called *contingent* if it is satisfiable but not a tautology.

**Exercise 2.9** Which of the following are satisfiable (if a formula is satisfiable, you should also give a valuation which satisfies it):

- $p \land \neg q$.
- $p \rightarrow \neg p$.
- $p \land \neg p$.

Clearly, every tautology is satisfiable, but not every satisfiable formula is a tautology.

Two formulas $F_1$ and $F_2$ are called *logically equivalent* if $V^+(F_1) = V^+(F_2)$ for any $V$. Notation for ‘$F_1$ and $F_2$ are logically equivalent’: $F_1 \equiv F_2$.

Further on, we will often ignore the distinction between a valuation $V$ and its extension $V^+$, and we will refer to both as $V$. 

Exercise 2.10 Which of the following are true:

- \( \neg \neg p \equiv p \).
- \( \neg (p \land q) \equiv (\neg p \lor \neg q) \).
- \( \neg (p \lor q) \equiv (\neg p \land \neg q) \).
- \( p \rightarrow q \equiv \neg p \lor q \).
- \( \neg (p \rightarrow q) \equiv p \land \neg q \).
- \( \neg (p \leftrightarrow q) \equiv \neg p \land q \).

Note that it follows from the definition of logical equivalence that all tautologies are logically equivalent to one another, and so are all contradictions.

Formulas \( P_1, \ldots, P_n \) logically imply formula \( C \) (\( P \) for premise, \( C \) for conclusion) if every valuation which makes every member of \( P_1, \ldots, P_n \) true also makes \( C \) true. Notation for \( \vdash \): \( P_1, \ldots, P_n \vdash C \).

Exercise 2.11 Which of the following are true:

- \( p \vdash p \land q \).
- \( p \vdash p \lor q \).
- \( p \vdash \neg \neg p \).
- \( p \rightarrow q \vdash \neg p \rightarrow \neg q \).
- \( \neg p \vdash p \rightarrow q \).
- \( \neg q \vdash p \rightarrow q \).
- \( p, p \rightarrow q \vdash q \).
- \( \neg p, q \rightarrow p \vdash \neg q \).

To find out whether \( P_1, \ldots, P_n \models C \) we have to perform reasoning. Pragmatic issues do come into play here.

2.4 Semantics of Mastermind

As an example application of propositional logic, we briefly discuss the semantics of Mastermind. In the grammar for Mastermind of Section 1.3 it is assumed that there are four colours (red, yellow, blue, green) and four positions. According to the language, colours may be repeated. The following version of the game is slightly easier to analyse: there are four colours (red, yellow, blue, green), three positions, and no colours may be repeated.
Due to the ban on colour repetition there are $4 \times 3 \times 2 = 24$ possible settings. Use propositional formulas for the encoding, with proposition letters $r_1$ for ‘red occurs at position one’, and so on. The rules of the game are:

- no colour may be repeated: $r_1 \rightarrow (\neg r_2 \land \neg r_3), r_2 \rightarrow (\neg r_1 \land \neg r_3), r_3 \rightarrow (\neg r_1 \land \neg r_2)$, and similarly for the other colours.
- any position must have a colour from the list of four: $r_1 \lor y_1 \lor b_1 \lor g_1$, and similarly for the other positions.

An answer ‘black’ means: one coloured peg at the correct position. An answer ‘white’ means: one peg present of the right colour, but at an incorrect position.

Assume the initial setting is red, yellow, blue. The problem is solved as soon as the correct formula $r_1 \land y_2 \land b_3$ is logically implied by the formulas that encode the information about the rules of the game and the information provided by the answers to the guesses. Here is a possible game:

**Guess:** blue, yellow, green.

**Answer:** black, white.

We see immediately from this answer that the colour red has to occur. This leaves six possibilities: $(b_1 \land r_2 \land y_3) \lor (b_1 \land g_2 \land r_3) \lor (g_1 \land y_2 \land r_3) \lor (r_1 \land y_2 \land g) \lor (y_1 \land r_2 \land g_3) \lor (r_1 \land y_2 \land g_3)$.

**Guess:** blue, red, yellow.

**Answer:** white, white, white.

The correct position must be the other permutation of blue, red and yellow that was still open, which leaves only one possibility:

**Guess:** red, yellow, blue.

**Answer:** black, black, black.

**Exercise 2.12** Analyse the version where colours may be repeated. Assume three positions.

Note that again there is a role for pragmatics in this game. The pragmatic rule ‘ask for relevant information’ would forbid to make a guess that is already ruled out by answers to earlier guesses.

### 2.5 Semantics of SIMPL

The next example language from our list is the fragment of simple imperative programming SIMPL, the programming language for calculations with whole numbers. In this case the set

\[ Z = \{ \cdots, -3, -2, -1, 0, 1, 2, 3, \cdots \} \]
figures as our extralinguistic structure. We assume we know what we mean what \( \mathbb{Z} \) is. We also assume that we know what we mean by the operations \( \oplus \) (addition), \( \otimes \) (multiplication) and \( \ominus \) (sign inversion: this is a one-placed operation), plus the relation \( \leq \) (less than or equal to) on \( \mathbb{Z} \).

With these ingredients we define a function \( T \) which maps terms from the imperative programming fragment to elements of \( \mathbb{Z} \), provided we know what to do with the variables of the language. We want to think about the statements of the language as programs that we want to execute on a suitable machine. In this connection, the variables are nothing but names of machine registers, i.e., stores containing elements of \( \mathbb{Z} \), represented in some suitable way. The manner of representation will not concern us here. Also, we will disregard the fact that the machine memory is finite. Assume that \( X = \{x_1, x_2, x_3, \ldots \} \) is the set of all variables of the fragment. Then a machine which can run statements of the fragment has a memory organization which looks like this.

\[
\begin{array}{cccccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & \ldots \\
\end{array}
\]

Every square (or store, or register) has space for the representation of an element of \( \mathbb{Z} \). The variables are just the names of the registers. A machine state is a situation where every register does indeed contain some whole number. For example:

\[
\begin{array}{cccccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & \ldots \\
  -3 & 0 & 327 & 2 & 22 & -17 & 0 & -2 & \ldots \\
\end{array}
\]

Looking at this slightly more abstractly still, we see that a machine state \( s \) is essentially a function from \( X \) to \( \mathbb{Z} \). If \( s \) is a machine state, then we use \( s(x|z) \) for the state which differs from \( s \) at most in the fact that \( z \) is the value of \( x \) (where \( s \) might have assigned some different value). For example, if \( s \) is the memory state pictured above, then \( s(x_3|0) \) is the following state:

\[
\begin{array}{cccccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & \ldots \\
  -3 & 0 & 0 & 2 & 22 & -17 & 0 & -2 & \ldots \\
\end{array}
\]

**Exercise 2.13** Let machine state \( s \) be given. Define a function \( T_s \) which maps the terms of the fragment to members of \( \mathbb{Z} \), based on the values which \( s \) assigns to the program variables.

**Exercise 2.14** Let state \( s \) and the value function \( T_s \) based on \( s \) (as defined in the previous exercise) be given. Define a value function \( V_s \) which maps the expressions of the programming fragment to values in \( \{0, 1\} \), using the term values assigned by \( T_s \).

The only thing which remains to be done is to give the semantics of the programming statements. We may assume, of course, that we know in which state \( s \) we are, so we have \( T_s \) and \( V_s \) available from the previous exercises.

The meaning of a program statement is given in the same manner as the meaning of a Sea Battle attack: as a transition from one state to another. The state change effected by \( x := t \) is obvious:
the new state is equal to the old state, except for the fact that \( x \) has received a new value:

\[
s[x := t]' \text{ iff } s' = s(x|V_s(t)).
\]

For the other cases we define the state changes recursively:

\[
s[\text{statement}_1; \cdots; \text{statement}_n]' \text{ s' }\]
\[
\text{iff } s[\text{statement}_1]s_1 \cdots s_{n-1}[\text{statement}_n]s_n \text{ and } s' = s_n.
\]

\[
s[\text{if expression then statement}'] s'
\]
\[
\text{iff } \begin{cases} 
  s[\text{statement}]s' & \text{if } V_s(\text{expression}) = 1, \\
  s' = s & \text{otherwise.}
\end{cases}
\]

\[
s[\text{if expression then statement}_1 \text{ else statement}_2]' s'
\]
\[
\text{iff } \begin{cases} 
  s[\text{statement}_1]s' & \text{if } V_s(\text{expression}) = 1, \\
  s[\text{statement}_2]s' & \text{otherwise.}
\end{cases}
\]

\[
s[\text{while expression do statement}]' s'
\]
\[
\text{iff } \begin{cases} 
  s[\text{statement}_1]s_1 \cdots s_{n-1}[\text{statement}]s_n, \\
  V_s(\text{expression}) = V_s_1(\text{expression}) = \cdots = V_s_{n-1}(\text{expression}) = 1, \\
  V_{s_n}(\text{expression}) = 0, \text{ and } s' = s_n.
\end{cases}
\]

Note that in the clause for the \text{while} statement \( n \) is allowed to be equal to 0. In that case we have \( s = s' \); this is the case where the embedded \text{statement} does not get executed at all. Observe also that while statements which get into an infinite loop are not assigned a meaning by the above clause.

**Exercise 2.15** Give an example of a statement in the programming fragment which gets into an infinite loop.

**Exercise 2.16** Give a semantic clause for ‘repeat statement until expression’. Show that this statement is definable in our fragment as it stands.

**Exercise 2.17** Consider the following SIMPL program.

\[
\text{begin } i := 0; y := 0; \text{ while } i < x \text{ do begin } i := i + 1; y := i + y \text{ end end}
\]

What is the value of \( y \) after execution of this program in a state \( s \) with \( s(x) = 5 \)?

On the basis of the examples encountered so far we can already make a distinction between two flavours of semantics of language fragments: \textit{static}, i.e., in terms of truth and falsity, i.e., in terms of functions with values 0 and 1, or \textit{dynamic}, i.e., in terms of state changes. As we will see further on, this distinction is also quite important for the semantics of natural language fragments.

**Exercise 2.18** Check whether the semantics of Mastermind, Bingo, and Telephone Chess is \textit{static} or \textit{dynamic}. 
2.6 Semantics of Predicate Logic

The semantics of predicate logic is static again. For convenience we limit ourselves to a language fragment with just three predicate letters \( P, R, S \), where \( P \) has one place, \( R \) has two places, and \( S \) has three. In fact, we have already seen examples of two-placed predicate constants in the imperative language fragment: the constants \(<, \leq, \geq, >\). The structure \((Z, \leq)\) is suited for the interpretation of these constants.

For present purposes we make the story slightly more general. How should an extralinguistic structure for the constants \( P, R, S \) look? Such a structure should at least contain a domain of discourse \( D \) consisting of individual entities, with an interpretation for \( P \), for \( R \) and for \( S \). These interpretations are given by a function \( I \). We need an \( I \) with \( I(P) \subseteq D \), \( I(R) \subseteq D \times D \), and \( I(S) \subseteq D \times D \times D \). Here \( D \times D \) (which is sometimes written as \( D^2 \)) the set of all pairs of elements from \( D \), and \( D \times D \times D \) (also written as \( D^3 \) sometimes) the set of all ordered triples of elements from \( D \). The picture below gives an example of a domain with a possible \( I(P) \) and \( I(R) \).

\[
\begin{align*}
\text{Legend:} & \quad \circ \text{ marks objects with property } P; \quad \rightarrow \text{ indicates that two objects are } R\text{-related.}
\end{align*}
\]

A structure \( M = \langle D, I \rangle \) consisting of a non-empty domain \( D \) with an interpretation function for the constants of language \( L \) is called a model for \( L \). We will always assume that the domain \( D \) of a model is non-empty.

Given a structure with interpretation function \( M = \langle D, I \rangle \), we can define a valuation for the predicate logical formulas, provided we know how to deal with the values of individual variables. Here the story is similar to that for imperative programming. We assume we have a state or assignment function \( s : X \rightarrow D \). As before, \( s(x|d) \) means the assignment which is like \( s \) except for the fact that \( x \) gets value \( d \) (where \( s \) might have assigned a different value).

There is still one missing ingredient for a value function \( T \) which maps terms of predicate logic to values in the domain of discourse \( D \), namely an interpretation for the constants of the language. Let \( C \) be the set of constants. The interpretation of those constants must be given, so we assume that the interpretation function \( I \) maps the members of \( C \) to individuals in \( D \). Assume \( X \) is the set of variables of the language. To keep things simple, we will first assume that there are no function constants. (At the end of the section we will discuss how these are to be dealt with.)
The value function $V$ uses interpretation $I$ and state $s$, as follows:

$$V_{I,s}(t) = \begin{cases} 
I(t) & \text{if } t \in C, \\
\pi(t) & \text{if } t \in X.
\end{cases}$$

The final part of the semantic definition consists of stating in which case model $M = \langle D, I \rangle$ and state $s$ satisfy a formula $F$. We use the notation $M \models_s F$ for this notion. Here is the definition:

- $M \models_s Pt$ iff $V_{I,s}(t) \in I(P)$
- $M \models_s Rt_{1}t_{2}$ iff $\langle V_{I,s}(t_{1}), V_{I,s}(t_{2}) \rangle \in I(R)$
- $M \models_s St_{1}t_{2}t_{3}$ iff $\langle V_{I,s}(t_{1}), V_{I,s}(t_{2}), V_{I,s}(t_{3}) \rangle \in I(S)$
- $M \models_s t_{1} = t_{2}$ iff $V_{I,s}(t_{1}) = V_{I,s}(t_{2})$
- $M \models_s \neg F$ iff it is not the case that $M \models_s F$.
- $M \models_s (F_{1} \land F_{2})$ iff $M \models_s F_{1}$ and $M \models_s F_{2}$
- $M \models_s (F_{1} \lor F_{2})$ iff $M \models_s F_{1}$ or $M \models_s F_{2}$
- $M \models_s \forall x F$ iff for all $d \in D$ it holds that $M \models_s(x|d) F$
- $M \models_s \exists x F$ iff for at least one $d \in D$ it holds that $M \models_s(x|d) F$

What we have presented just now is a recursive definition of truth for predicate logical formulas.

If we evaluate closed formulas (formulas without free variables), the assignment $s$ becomes irrelevant, so for a closed formula $F$ we can simply put $M \models F$ iff there is some assignment $s$ with $M \models_s F$.

**Exercise 2.19** Let $M$ be the model pictured above. Which of the following statements are true:

- $M \models \exists x (P_{x} \land R_{xx})$.
- $M \models \forall x (P_{x} \rightarrow R_{xx})$.
- $M \models \forall x (P_{x} \rightarrow \exists y R_{xy})$.
- $M \models \exists x (P_{x} \land \neg R_{xx})$.
- $M \models \exists x (\exists y R_{yx} \land R_{xx})$.
- $M \models \forall x (\exists y R_{yx} \rightarrow R_{xx})$.
- $M \models \forall x (R_{xx} \rightarrow \exists y R_{xy})$.

A predicate logical sentence $F$ is called logically valid if $F$ is true in every model. Notation for ‘$F$ is logically valid’: $\models F$. From the convention that the domains of our models are always non-empty it follows that $\models \forall x F \rightarrow \exists x F$, for all $F$ with at most the variable $x$ free.
Exercise 2.20 Which of the following statements are true:

- $\models \exists x P x \lor \forall x \neg P x$
- $\models \exists x P x \lor \forall x \neg P x$
- $\models \forall x P x \lor \forall x \neg P x$
- $\models \forall x P x \lor \exists x \neg P x$
- $\models \exists x \exists y R x y \rightarrow \exists x R x x$
- $\models \forall x \forall y R x y \rightarrow \forall x R x x$
- $\models \exists x \exists y R x y \rightarrow \exists x \exists y R y x$
- $\models \forall x R x x \lor \exists x \exists y \neg R y x$
- $\models \forall x R x x \rightarrow \forall x \exists y R x y$
- $\models \exists x R x x \rightarrow \forall x \exists y R x y$
- $\models \forall x \forall y \forall z ((R x y \land R y x) \rightarrow R x y)$

The truth definition makes essential use of assignments, and still, in the exercises so far, where we have looked only at sentences (closed formulas), truth or falsity does not depend on which assignment we use. One might think, therefore, that we can do without assignments altogether if we confine ourselves to defining the truth values of the sentences (closed formulas) of predicate logic.

The trouble is that when we apply the truth definition above to a sentence, e.g., to the sentence $\forall x (P x \rightarrow \exists y R x y)$, then the clause for dealing with the universal quantifier makes reference to the notion of truth for the formula $(P x \rightarrow \exists y R x y)$, which is an open formula. In order to determine whether this is true we have to be told which object $x$ denotes. The situation is entirely analogous to the interpretation of natural language sentences:

2.1 Every student owns a bicycle.

2.2 He owns a bicycle.

To determine the truth of (2.2) we have to be told who is the referent of the pronoun he.

Exercise 2.21 Assume that we can extend the language with a set $A$ of proper names for objects, in such a way that every object in $D$ is named by an element from $A$, by means of an extension $I'$ of the interpretation function $I$ of the model. In other words, we have for all $d \in D$ that there is some $a \in A$ with $I'(a) = d$. Give an alternative truth definition for predicate logic that does not use assignments, but these extra names, plus the notion of substituting names from $A$ for variables.
2.6. SEMANTICS OF PREDICATE LOGIC

In order to extend the truth definition to structured terms (terms containing function constants), we have to assume that the interpretation function $I$ knows how to deal with a function constant. If $f$ is a one-placed function constant, then $I$ should map $f$ to a unary operation on the domain $D$, i.e., to a function $I(f) : D \rightarrow D$. In general: if $g$ is an $n$-placed function constant, then $I$ should map $g$ to an $n$-ary operation on the domain $D$, i.e., to a function $I(g) : D^n \rightarrow D$.

The value function $V$ is now extended to functional terms, as follows (note the recursion in the definition):

$$V_{I,s}(t) = \begin{cases} 
I(t) & \text{if } t \in C, \\
\ell(t) & \text{if } t \in X, \\
I(g)(V_{I,s}(t_1), \ldots, V_{I,s}(t_n)) & \text{if } t \text{ has the form } g(t_1, \ldots, t_n).
\end{cases}$$

This value function is used in the truth definition, but we leave the details to you:

**Exercise 2.22** Write out the truth definition for predicate logical languages with function constants.

An important aim of any logic language is to study the process of valid reasoning in that language. Every logical language comes with an appropriate notion of ‘valid consequence’ for that language. Intuitively, when can we draw a conclusion $C$ from premises $P_1, \ldots, P_n$? When it is inconceivable that the premises are true and the conclusion false, or in other words when the truth of the premises brings the truth of the conclusion in its wake.

Now that we have a fully precise notion of truth for predicate logic we can uses this to make the intuitive notion of valid consequence fully precise too. We say that predicate logical sentence $C$ logically follows from predicate logical sentence $P$ ($P$ for premise and $C$ for conclusion; we also say that $P$ logically implies $C$) if every model which makes $P$ true also makes $C$ true. Notation for ‘$P$ logically implies $C$’: $P \models C$.

How do we judge statements of the form $P \models C$ (where $P, C$ are closed formulas of predicate logic)? It is clear how we can refute the statement $P \models C$, namely, by finding a counterexample. A counterexample to $P \models C$ is a model $M$ with $M \models P$ but NOT: $M \models C$, or in abbreviated notation: $M \not\models C$. Here are some example questions about valid consequence in predicate logic.

**Exercise 2.23** Which of the following statements hold? If a statement holds, then you should explain why. If it does not, then you should give a counterexample.

- $\forall xPx \models \exists xPx$.
- $\exists xPx \models \forall xPx$.
- $\forall xRxx \models \forall x\exists yRxy$.
- $\forall x\forall yRxy \models \forall xRxx$.
- $\exists x\exists yRxy \models \exists xRxx$.
- $\forall x\exists yRxy \models \forall xRxx$.
• $\exists y \forall x Rxy \models \forall x \exists y Rxy$.
• $\forall x \exists y Rxy \models \exists y \forall x Rxy$.
• $\exists x \forall y Rxy \models \exists x \exists y Ryx$.
• $\forall x \exists xy \models \forall x \exists y Rxy$.
• $\exists x \exists Rxx \models \exists x \exists y Rxy$.

We can make this slightly more general by allowing sets of more than one premise. Assume that $P_1, \ldots, P_n, C$ are sentences (closed formulas) of predicate logic. We say that $C$ logically follows from $P_1, \ldots, P_n$, or, formally, that $P_1, \ldots, P_n \models C$, if for every model $M$ for the language with the property that $M \models P_1$ and ... and $M \models P_n$ it is the case that $M \models C$.

**Exercise 2.24** Which of the following hold:

• $\forall x \forall y (Rxy \rightarrow Ryx), Rab \models Rba$.
• $\forall x \forall y (Rxy \rightarrow Ryx), Rab \models Raa$.

This was not too difficult, but still, models for predicate logical languages can be extremely complex, so the fact that in some particular case we cannot find a counterexample does not always tell us that the conclusion follows logically from the premises. The jump from premises to conclusion may be too dangerous. It is for this reason that logicians have proposed systems of reasoning where the jumps from premise to conclusion are decomposed into smaller jumps, until the jumps become so small that we can easily check their correctness. Here opens a vast area of pragmatics for predicate logic ... 

### 2.7 Semantics of Natural Language Fragments

Now what about the meanings of the sentences in our simple fragment of English? Using what we know now about the meanings of predicate logical formulas, and assuming we have predicate letters available for the nouns and verbs of the fragment, we can easily translate the sentences generated by the example grammar into predicate logic. Assume the following translation key:

<table>
<thead>
<tr>
<th>lexical item</th>
<th>translation</th>
<th>type of logical constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>man</td>
<td>$M$</td>
<td>one-placed predicate</td>
</tr>
<tr>
<td>woman</td>
<td>$W$</td>
<td>one-placed predicate</td>
</tr>
<tr>
<td>boy</td>
<td>$B$</td>
<td>one-placed predicate</td>
</tr>
<tr>
<td>laughed</td>
<td>$L$</td>
<td>one-placed predicate</td>
</tr>
<tr>
<td>smiled</td>
<td>$S$</td>
<td>one-placed predicate</td>
</tr>
<tr>
<td>loved</td>
<td>$L$</td>
<td>two-placed predicate</td>
</tr>
<tr>
<td>respected</td>
<td>$R$</td>
<td>two-placed predicate</td>
</tr>
<tr>
<td>Mary</td>
<td>$m$</td>
<td>individual constant</td>
</tr>
<tr>
<td>Bill</td>
<td>$b$</td>
<td>individual constant</td>
</tr>
</tbody>
</table>
Then the translation of *Every man loved a woman* could become: \( \forall x (Mx \rightarrow \exists y (Wy \land Lxy)) \).

**Exercise 2.25** Check with the truth definition for predicate logic that this translation expresses the correct meaning (if we disregard the past tense, that is).

**Exercise 2.26** Give predicate logical translations for the following sentences from the fragment:

- Every woman that smiled loved a man.
- No boy that loved a woman respected every man.
- Every woman respected every man that loved a woman.
- Every woman respected a man that no boy loved.

Still, there is a sense in which these predicate logical translations do not ‘count’ as a semantic account of the sentences in the example grammar. The trouble is that one has to understand English in order to check that the translations are correct. In other words, one has to know the meanings of the sentences already. At a later stage an account of the semantics of this and similar fragments will be presented that ‘counts’.

### 2.8 Further Reading

Grice [Gri75] gives the classical account of pragmatics. An excellent textbook on pragmatics is Levinson [Lev83]. A good introduction to logic for linguists is Gamut [Gam91a]. More mathematically oriented introductions are Van Dalen [Dal83] and Enderton [End72]. Logic textbooks with good explanations of the many connections to applications in computer science are [Bur98] and [HR00]. An introduction to the semantics of imperative programming languages can be found in Gordon [Gor88] or Gunter [Gun92].
Chapter 3

Functional Programming with Haskell

3.1 Types in Grammar and Types in Programming

Adjectives are words that combine with nouns to form complex nouns: *nice* combines with the noun *guy* to form the noun *nice guy*. In the grammar formalism of categorial grammar, one calls $A/B$ the type of a word that needs a type $B$ word to its right in order to produce a type $A$ word, according to the rule:

$$A/B + B = A.$$ 

Applying this to the case of adjectives and nouns: if nouns have type $N$, then adjectives have type $N/N$, and the complex noun *nice guy* is produced by the rule:

$$\text{nice}_{N/N} + \text{guy}_N = (\text{nice guy})_N.$$ 

Here, $N/N$ is the grammatical function, $N$ the grammatical argument. Adverbials are words that map adjectives into complex adjectives:

$$\text{very}_{(N/N)/(N/N)} + \text{nice}_{N/N} + \text{guy}_N = (\text{very nice})_{N/N} + \text{guy}_N = (\text{very nice guy})_N.$$ 

Similarly, $B\backslash A$ is the type of a word that needs a $B$ type word on its lefthand side to produce an $A$ type word. In English, an adjective like *emeritus* behaves like this:

$$\text{professor}_N + \text{emeritus}_{\backslash N} = (\text{professor emeritus})_N.$$ 

Here $N \backslash N$ is the function, $N$ the argument.

We can also write this without the directional information:
very\(_{(N\rightarrow N)}\)!\((N\rightarrow N)\) nice\(_{N\rightarrow N}\) guy\(_{N}\) = (very nice)\(_{N\rightarrow N}\) guy\(_{N}\) = (very nice guy)\(_{N}\) 

(emeritus)\(_{N\rightarrow N}\) professor\(_{N}\) = (professor emeritus)\(_{N}\).

This perspective on the ways in which expressions combine is not only fruitful in linguistics, but also in programming. If integer numbers have type \texttt{Int}, then we get:

- \textit{addition of integer numbers} has type \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}.
- \textit{incrementing integers by 1} has type \texttt{Int} \rightarrow \texttt{Int}.
- \textit{squaring integers} has type \texttt{Int} \rightarrow \texttt{Int}.

This information is very useful to check whether a program is \textit{well typed}. Checking types is a handy way to spot common mistakes in programming.

In this book you will get acquainted with how this works in functional programming, more particularly in the lazy functional programming language Haskell. Haskell was named after the logician Haskell B. Curry. Curry, together with Alonzo Church, laid the foundations of functional computation around 1940, by developing the \textit{lambda calculus}. As a functional programming language, Haskell is a member of the Lisp family. Others family members are Scheme, ML, Occam, Clean. Haskell98 is intended as a standard for lazy functional programming. Lazy functional programming is a programming style where arguments are evaluated only when the value is actually needed.

The remainder of this book is written in so-called ‘literate programming’ style [Knu92]. Literate programming is a programming style where the program and its documentation are generated from the same source. The text of every chapter in this book can be viewed as the documentation of the program code in that chapter. Literate programming makes it impossible for program and documentation to get out of sync. Program documentation is an integrated part of literate programming, in fact the bulk of a literate program is the program documentation. When writing programs in literate style there is less temptation to write program code first while leaving the documentation for later. Programming in literate style proceeds from the assumption that the main challenge when programming is to make your program digestible for humans. For a program to be useful, it should be easy for others to understand the code. It should also be easy for you to understand your own code when you reread your stuff the next day or next week or next month and try to figure out what you were up to when you wrote your program.

To save you the trouble of retyping, the code discussed in this book can be retrieved from the book website. The program code is the text in \texttt{typewriter} font that you find in rectangular boxes throughout the chapters. Boxes may also contain code that is not included in the chapter modules, usually because it defines functions that are already predefined by the Haskell system, or because it redefines a function that is already defined elsewhere in the chapter.

Typewriter font is also used for pieces of interaction with the Haskell interpreter, but these illustrations of how the interpreter behaves when particular files are loaded and commands are given are not boxed.

Like the chapters that follow, this chapter is a a so-called Haskell module. The following two lines declare the Haskell module for the Haskell code of the chapter. This module is called FPH.
3.2 Using the Haskell Interpreter

We assume that you succeeded in retrieving the Haskell interpreter hugs from the Haskell homepage www.haskell.org and that you managed to install it on your computer. You can start the interpreter by typing hugs at the system prompt. When you start hugs you should see something like Figure (3.1). The string Prelude> on the last line is the Haskell prompt when no user-defined files are loaded. The Haskell prelude contains a number of useful predefined functions. The definitions are in the file Prelude.hs. When the interpreter starts up, it indicates the directory from which Prelude.hs is loaded. If you want to quickly learn a lot about how to program in Haskell, you should get into the habit of consulting this file regularly. The definitions of all the standard operations are open source code, and are there for you to learn from. The Haskell Prelude may be a bit difficult to read at first, but you will soon get used to the syntax.

You can use hugs as a calculator as follows:

Prelude> 2^16
65536
Prelude>

The string Prelude> is the system prompt. 2^16 is what you type. After you hit the return key (the key that is often labeled with Enter or ←), the system answers 65536 and the prompt Prelude> reappears.
Exercise 3.1 Try out a few calculations using \( \times \) for multiplication, \(+\) for addition, \(-\) for subtraction, \(^\) for exponentiation, \(/\) for division. By playing with the system, find out what the precedence order is among these operators.

Parentheses can be used to override the built-in operator precedences:

Prelude> \((2 + 3)^4\)
625

This uses the built-in definitions of the operators \(\ast, +, -, ^, \) and \(/\). We can also define our own functions. Here is an example:

Prelude> square 2 where square x = x \* x
4

We use a function \texttt{square}, and on the same line we tell the interpreter what we mean by it, using the reserved keyword \texttt{where}. Here is another way of achieving the same, this time using the reserved keywords \texttt{let} and \texttt{in}:

Prelude> let square x = x \* x in square 3
9

To quit the Hugs interpreter, type :quit or :q at the system prompt.

Exercise 3.2 Read Chapter 3 (‘Hugs for Beginners’) from the online Hugs 98 User Manual and try out some more commands. The manual can be found at address http://cvs.haskell.org/Hugs/pages/hugsman/index.html.

Exercise 3.2 has taught you how to load your own code. The code for the rest of the chapter can be retrieved from the book website, at address http://www.cwi.nl/~jve/cs.

Here is a definition of the squaring function, together with an appropriate type declaration. The type declaration expresses that \texttt{square} combines with an \texttt{Int} to produce an \texttt{Int}.

```
square :: Int \rightarrow Int
square x = x \* x
```

To load and use the chapter code, proceed as indicated below. Note that in the following lines, :l refers to the letter \texttt{l}, not the numeral 1. Next to :l, a very useful command after you have edited a file of Haskell code is :r, for reloading the file.

Prelude> :l FPH
Reading file "FPH.hs":

Hugs session for:
/usr/lib/hugs/lib/Prelude.hs
FPH.hs
FPH> square 7
49
FPH> square (-3)
9
FPH> square (square 7)
2401
FPH> square (square (square 7))
5764801

The string FPH> is the Haskell prompt, the rest of the first line is what you type. When you press Enter the system answers with the second line, followed by the Haskell prompt. And so on.

**Remark** In Haskell it is not strictly necessary to always give explicit type declarations. For instance, the definition of `square` would also work without the type declaration, since the system can infer the type from the definition. However, it is good programming practice to give explicit type declarations even when this is not strictly necessary. These type declarations are an aid to understanding, and they greatly improve the digestibility of functional programs for human readers.

Back to a natural language example. Take the statement *Diana loves Charles*. By means of abstraction, we can get all kinds of properties and relations from this statement:

- ‘loving Charles’
- ‘being loved by Diana’
- ‘loving’
- ‘being loved’

This works as follows. We replace the element that we abstract over by a variable, and we bind that variable by means of a lambda operator. Like this:

- `\x. x loves Charles` expresses ‘loving Charles’.
- `\x. Diana loves x` expresses ‘being loved by Diana’.
- `\x\y. x loves y` expresses ‘loving’.
- `\y\x. x loves y` expresses ‘being loved’.
Another way of writing the `square` function is by means of lambda abstraction. In Haskell, \( \lambda x \) expresses lambda abstraction over variable \( x \).

\[
\text{sqr} :: \text{Int} \to \text{Int} \\
\text{sqr} = \lambda x \to x \times x
\]

The intention is that variable \( x \) stands proxy for a number of type \text{Int}. The result, the squared number, also has type \text{Int}. The function \text{sqr} is a function that, when combined with an argument of type \text{Int}, yields a value of type \text{Int}. This is precisely what the type-indication \text{Int} \to \text{Int} expresses.

### 3.3 Properties of Things, Characteristic Functions

The property ‘being divisible by three’ can be represented as a function from numbers to truth values. The numbers \( \ldots, -9, -6, -3, 0, 3, 6, 9, \ldots \) get mapped to \text{True} by that function, all other numbers get mapped to \text{False}.

Programmers call a truth value a \text{Boolean}, in honour of the British logician George Boole (1815–1864). As the type of threefold we can therefore take \text{Int} \to \text{Bool}. Here is a definition of the property of being a threefold with lambda abstraction. This uses the predefined function \text{rem}. \text{rem} \( x \) \( y \) gives the remainder when \( x \) gets divided by \( y \).

\[
\text{threefold} :: \text{Int} \to \text{Bool} \\
\text{threefold} = \lambda x \to \text{rem} x 3 == 0
\]

```
FPH> threefold 5
False
FPH> threefold 12
True
```

\text{Less than} and \text{less than or equal} are examples of relations on the integers, and on various other number domains, in fact.

Viewed as relations on \text{Int}, these functions have type \text{Int} \to \text{Int} \to \text{Bool}. This is read as \text{Int} \to (\text{Int} \to \text{Bool}), for \( \to \) associates to the right, so there is no need to write the parentheses. \text{Less than} is predefined in Haskell as \( (\text{<}) \), \text{less than or equal} as \( (\leq) \). Other examples of relations are equality and inequality, predefined as \( (\text{=}) \) and \( (\text{/=}) \).

Let us turn to some examples with characters and strings, The Haskell type of characters is \text{Char}. Strings of characters have type \text{[Char]}. Similarly, lists of integers have type \text{[Int]}. The
empty string (or the empty list) is \[\]. The type [\texttt{Char}] is abbreviated as \texttt{String}. Examples of characters are 'a', 'b' (note the single quotes) examples of strings are "Montague" and "Chomsky" (note the double quotes). In fact, "Chomsky" can be seen as an abbreviation of the following character list:

\['C', 'h', 'o', 'm', 's', 'k', 'y'\].

If strings have type [\texttt{Char}] (or \texttt{String}), properties of strings have type [\texttt{Char}] \rightarrow \texttt{Bool}. Here is a simple property:

```haskell
aword :: [Char] \rightarrow \texttt{Bool}
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

This definition uses \textit{pattern} matching: (x:xs) is the prototypical non-empty list. The head of (x:xs) is x, the tail is xs. The head and tail are glued together by means of the operation :\, of type \texttt{a} \rightarrow [\texttt{a}] \rightarrow [\texttt{a}]. The operation combines an object of type \texttt{a} with a list of objects of the same type to a new list of objects, again of the same type.

It is common Haskell practice to refer to non-empty lists as \texttt{x:xs}, \texttt{y:ys}, and so on, as a useful reminder of the facts that \texttt{x} is an element of a list of \texttt{x}'s and that \texttt{xs} is a list.

Note that the function \texttt{aword} is called again from the body of its own definition. We will encounter such \texttt{recursive} function definitions again and again in the course of this book.

What the definition of \texttt{aword} says is that the empty string is not an \texttt{aword}, and a non-empty string is an \texttt{aword} if either the head of the string is the character \texttt{a}, or the tail of the string is an \texttt{aword}. As you can see, characters are indicated in Haskell with single quotes. The following calls to the definition show that strings are indicated with double quotes:

FPH> aword "Diana"
True
FPH> aword "loves"
False

The Haskell prelude contains the following definition of the identity function:

```haskell
id :: a \rightarrow a
id x = x
```

This function can be applied to objects of \textit{any} type. Applied to an argument, it just returns that argument. Since the type of the argument does not matter, we say that the \texttt{id} function is
polymorphic. In the type declaration of \texttt{id}, \(a\) is a type variable. The use of \(a\) indicates that any type will fit.

\begin{verbatim}
FPH> id True
True
FPH> id 'a'
'a'
FPH> id "Diana"
"Diana"
FPH> (id aword) "Diana"
True
\end{verbatim}

Type polymorphism, and the use of \(a, b\) as type variables, allow us to define polymorphic types for properties and relations. The polymorphic type of a property is \(a \rightarrow \text{Bool}\). The polymorphic type of a relation is \(a \rightarrow b \rightarrow \text{Bool}\). The polymorphic type of a relation over a single type is \(a \rightarrow a \rightarrow \text{Bool}\).

### 3.4 List Types and List Comprehension

Examples of list types are \([\text{Int}]\), the type of lists of integers, \([\text{Char}]\) the type of lists of characters, or strings, and \([\text{Bool}]\), the type of lists of Booleans.

But we can get more general than this. If \(a\) is a type, \([a]\) is the type of lists over \(a\). Again, \([a]\) is a polymorphic type.

As we have seen, \((x:xs)\) is the prototypical non-empty list. The \texttt{head} of \((x:xs)\) is \texttt{x}, the \texttt{tail} is \texttt{xs}. These functions are defined in the Haskell prelude as follows:

\begin{verbatim}
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
\end{verbatim}

The underscores \(_\) indicate anonymous variables that can be matched by anything of the right type. Thus, in \((x:_)\), \(_\) matches any list, and in \((_:xs)\), \(_\) matches any object.

\textit{List comprehension} is the list counterpart of set comprehension:

\[\{x \mid x \in A, P(x)\}\]

Here are some examples:
3.5. **THE MAP AND FILTER FUNCTIONS**

The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

If \( f \) is a function of type \( a \to b \) and \( xs \) is a list of type \([a]\), then \( \text{map } f \; xs \) will return a list of type \([b]\).

```haskell
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : map f xs
```

`map \text{sqr} [1..9]` will produce the list of squares, for `sqr` is the squaring function:

```
FPH> map sqr [1..9]
[1,4,9,16,25,36,49,64,81]
```

`map \text{threefold}` and `map \text{aword}` will produce lists of truth values.

```
FPH> map threefold [1..9]
[False,False,True,False,False,True,False,False,True]
FPH> map aword ["Diana", "loves", "Charles"]
[True,False,True]
```

The `filter` function takes a property and a list, and returns the sublist of all list elements satisfying the property. The `filter` function has the following type:

```haskell
filter :: (a -> Bool) -> [a] -> [a]
```
Here a denotes an arbitrary type. Indeed, one can filtrate strings, or lists of integers, or lists of whatever, given a property of the right type: [Char] -> Bool for strings, Int -> Bool for integers, and so on.

The combination of filter together with an argument has a type of its own, as can be checked with the :t command (the command for checking the types of expressions).

FPH> :t filter threefold
filter threefold :: [Int] -> [Int]
FPH> :t filter aword
filter aword :: [[Char]] -> [[Char]]

Here is the definition of the filter function:

```haskell
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
               | otherwise = filter p xs
```

A new programming element is the use of equation guarding by means of the Haskell condition operator |. A Haskell equation of the form

```haskell
foo t | condition = ...
```

is called a guarded equation. The expression condition, of type Bool (i.e., Boolean or truth value), is called the guard of the equation.

A list of guarded equations such as

```haskell
foo t | condition_1 = body_1
foo t | condition_2 = body_2
foo t | condition_3 = body_3
foo t | otherwise = body_4
```

can be abbreviated as

```haskell
foo t | condition_1 = body_1
     | condition_2 = body_2
     | condition_3 = body_3
     | otherwise = body_4
```

Such a Haskell definition is read as follows:

- in case condition_1 holds, foo t is by definition equal to body_1,
• in case condition_1 does not hold but condition_2 holds, foo t is by definition equal to body_2,

• in case condition_1 and condition_2 do not hold but condition_3 holds, foo t is by definition equal to body_3,

• and in case none of condition_1, condition_2 and condition_3 hold, foo t is by definition equal to body_4.

When we are at the end of the list we know that none of the cases above in the list apply. This is indicated by means of the Haskell reserved keyword otherwise.

The first line of the filter definition handles the case where the head of the list (x:xs) satisfies property p. The second line of the code applies otherwise, i.e., it covers the case where the head of the list (x:xs) does not satisfy property p.

The filter function can be applied as follows:

FPH> filter even [1..10]
[2,4,6,8,10]
FPH> filter threelfold [23,4,5,7,18,123]
[18,123]
FPH> filter aword ["Diana", "loves", "Charles"]
["Diana","Charles"]

3.6 Application versus Abstraction; Function Composition

If we combine an expression of type \( a \rightarrow b \) with an expression of type \( a \), we get a result of type \( b \), where \( a \) en \( b \) are arbitrary types. This is called application.

Application corresponds with the operation of modus ponens in logic: conclude from \( a \rightarrow b \) and \( a \) that \( b \). The types are now taken as logical formulas. This deep connection between types and formulas is known as Curry-Howard correspondence.

When we use lambda abstraction to abstract from an expression of type \( b \) a variable of type \( a \), the result is in expression of type \( a \rightarrow b \). Again \( a \) en \( b \) are arbitrary types. This procedure corresponds to the proof theoretical procedure in logic called implication introduction.

For the composition \( f \cdot g \) to make sense, the result type of \( g \) should equal the argument type of \( f \), i.e., if \( f :: a \rightarrow b \) then \( g :: c \rightarrow a \).

Under this typing, the type of \( f \cdot g \) is \( c \rightarrow b \).

Thus, the operator \((.)\) that composes two functions has the following type and definition:

\[
(\cdot) :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b
\]

\[
(f \cdot g) x = f (g x)
\]
This is used in the Haskell prelude to define \texttt{odd} as \texttt{not \ . \ even}. More examples of function composition will be given below.

### 3.7 List Conjunction, Disjunction and Quantification

The conjunction operator \texttt{&&} is generalized to lists of booleans by the (predefined) function \texttt{and}.

\begin{verbatim}
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && (and xs)
\end{verbatim}

The disjunction operator \texttt{||} is generalized to lists of booleans by the (predefined) function \texttt{or}.

\begin{verbatim}
or :: [Bool] -> Bool
or [] = False
or (x:xs) = x || (or xs)
\end{verbatim}

These generalized boolean operations are used for stating the definitions of the list quantification operators \texttt{any} and \texttt{all} (these are also predefined in the Haskell prelude):

\begin{verbatim}
any, all :: (a -> Bool) -> [a] -> Bool
any p = or . map p
all p = and . map p
\end{verbatim}

These definitions use function composition.

```
FPH> all aword ["Diana", "loves", "Charles"]
False
FPH> any aword ["Diana", "loves", "Charles"]
True
```

Another example of function composition can be found in the definition of the \texttt{elem} and \texttt{notElem} functions, also from the Haskell prelude:
3.8 IDENTIFIERS IN HASKELL

In the type specification of `elem` and `notElem`, `Eq a =>` specifies `a` as a type in the `Eq` class, i.e., as a type for which equality (`==`) and inequality (`/=`) are defined.

```
FPH> elem 5 [1..9]
True
```

3.8 Identifiers in Haskell

In Haskell, there are two kinds of identifiers:

- Variable identifiers are used to name functions. They have to start with a lower-case letter. Examples are `map`, `max`, `fct2list`, `fctToList`, `fct_to_list`.
- Constructor identifiers are used to name types. They have to start with an upper-case letter. Examples are `True`, `False`.

Functions are operations on data-structures, constructors are the building blocks of the data structures themselves (trees, lists, Booleans, and so on).

Names of functions always start with lower-case letters, and may contain both upper- and lower-case letters, but also digits, underscores and the prime symbol `'`. The following reserved keywords have special meanings and cannot be used to name functions.

```
case class data default deriving do else
   if import in infix infixl infixr instance
   let module newtype of then type where
   _
```

There is one more reserved keyword that is particular to Hugs: `forall`, for the definition of functions that take polymorphic arguments. See the Hugs documentation for further particulars.

The use of these keywords will be explained as we encounter them. `_` at the beginning of a word is treated as a lower-case character; `_` all by itself is a reserved word for the wild card pattern that matches anything.

3.9 List Recursion with `foldr` and `foldl`

Structural recursion over finite lists always distinguishes two cases: (i) the treatment of the empty list, and (ii) the treatment of non-empty lists, with given heads and tails. The general
scheme for definition of a function \( h \) by recursion on list structure is as follows:

\[
\begin{align*}
    h \; [] & := z \\
    h(x : xs) & := f \; x \; (h \; xs)
\end{align*}
\]

For example, the function \( s \) that computes the sum of the elements in a list of numbers is defined by:

\[
\begin{align*}
    s \; [] & := 0 \\
    s(n : xs) & := n + s \; xs
\end{align*}
\]

Here 0 is taken for \( z \), and + for \( f \). Thus, the only thing you need to know is what to plug in for the \( z \) and what for the \( f \). The definition of \( h \) then proceeds by using this \( z \) and \( f \) in the manner specified by the general scheme. The general scheme is what the predefined function \texttt{foldr} provides. What \texttt{foldr} offers is a general way to handle definitions by list recursion. Here is the definition:

\[
\begin{align*}
    \text{foldr} & :: (a \to b \to b) \to b \to [a] \to b \\
    \text{foldr} \; f \; z \; [] & = z \\
    \text{foldr} \; f \; z \; (x : xs) & = f \; x \; (\text{foldr} \; f \; z \; xs)
\end{align*}
\]

Summing a list of integers can now be done by \texttt{foldr} \((+)\) 0. In general, \( z \) is the identity element of the operation \( h \), i.e., the value you would start out with in the base case of a recursive definition of the operation. The identity element for addition is 0, for multiplication it is 1. The product of a list of numbers is given by \texttt{foldr} \((\times)\) 1. Here \((\times)\) is the binary product operation.

The following informal version of the definition of \texttt{foldr} may further clarify its meaning:

\[
\text{foldr} \; (\oplus) \; z \; [x_1, x_2, \ldots, x_n] := x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus z) \cdots)).
\]

Consider the definitions of generalized conjunction and disjunction again. Haskell has predefined these operations in terms of \texttt{foldr}. To see how we can use \texttt{foldr} to implement generalized conjunction and disjunction, we only need to know what the appropriate identity elements of the operations are. Should the conjunction of all elements of \([]\) count as true or false? As true, for it is indeed (trivially) the case that all elements of \([]\) are true. So the identity element for conjunction is \texttt{True}. Should the disjunction of all elements of \([]\) count as true or false? As false, for it is false that \([]\) contains an element which is true. Therefore, the identity element for disjunction is \texttt{False}. This explains the following Haskell definition in \texttt{Prelude.hs}:

\[
\begin{align*}
    \text{and, or} & :: [\text{Bool}] \to \text{Bool} \\
    \text{and} & = \text{foldr} \; (\&\&) \; \text{True} \\
    \text{or} & = \text{foldr} \; (||) \; \text{False}
\end{align*}
\]
The operation \texttt{foldr} folds ‘from the right’. Folding ‘from the left’ can be done with its cousin \texttt{foldl}, predefined in \texttt{Prelude.hs} as follows:

\begin{verbatim}
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []  = z
foldl f z (x:xs) = foldl f (f z x) xs
\end{verbatim}

An informal version may further clarify this:

\[
\text{foldl} (\oplus) z [x_1, x_2, \ldots, x_n] := (\cdots ((z \oplus x_1) \oplus x_2) \oplus \cdots) \oplus x_n.
\]

This can be used to flesh out the following recursion scheme:

\[
h z [] := z
h z (x:xs) := h (f z x) xs
\]

This boils down to recursion over lists with an extra parameter, and in fact \texttt{foldl} can be used to speed up list processing. The case of reversing a list is an example:

\[
r z [] := zs
r z (x:xs) := r (\lambda ys.\lambda y:ys zs x) xs.
\]

Here is a definition in terms of \texttt{foldl}:

\[
\text{rev} :: [a] -> [a]
\text{rev} = \text{foldl} (\\lambda xs x -> x:xs) []
\]

\textbf{Exercise 3.3} Consider the following definition of list reversal in terms of \texttt{foldr}.

\[
\text{rev'} :: [a] -> [a]
\text{rev'} = \text{foldr} (\\lambda x xs -> xs ++ [x]) []
\]

Make drawings of what happens when each of the functions \texttt{rev} and \texttt{rev'} gets applied to the string "banana". Can you see why \texttt{rev} is more efficient than \texttt{rev'}?
3.10 Some Text Processing Examples

We conclude this chapter with some examples of shallow text analysis, using two of Shakespeare’s sonnets as text material.

```haskell
sonnet18 = "Shall I compare thee to a summer’s day? \n" ++ "Thou art more lovely and more temperate: \n" ++ "Rough winds do shake the darling buds of May, \n" ++ "And summer’s lease hath all too short a date: \n" ++ "Sometime too hot the eye of heaven shines, \n" ++ "And often is his gold complexion dimm’d; \n" ++ "And every fair from fair sometime declines, \n" ++ "By chance or nature’s changing course untrimm’d; \n" ++ "But thy eternal summer shall not fade \n" ++ "Nor lose possession of that fair thou owest; \n" ++ "Nor shall Death brag thou wander’st in his shade, \n" ++ "When in eternal lines to time thou growest: \n" ++ " So long as men can breathe or eyes can see, \n" ++ " So long lives this and this gives life to thee."
```

As you can see, line ends are encoded as \n. This is the line feed control character. To display the sonnet on the screen you can use an output function that executes the linefeeds: putStrLn sonnet18.

```haskell
sonnet73 = "That time of year thou mayst in me behold \n" ++ "When yellow leaves, or none, or few, do hang \n" ++ "Upon those boughs which shake against the cold, \n" ++ "Bare ruin’d choirs, where late the sweet birds sang. \n" ++ "In me thou seest the twilight of such day \n" ++ "As after sunset fadeth in the west, \n" ++ "Which by and by black night doth take away, \n" ++ "Death’s second self, that seals up all in rest. \n" ++ "In me thou see’st the glowing of such fire \n" ++ "That on the ashes of his youth doth lie, \n" ++ "As the death-bed whereon it must expire \n" ++ "Consumed with that which it was nourish’d by. \n" ++ " This thou perceivest, which makes thy love more strong, \n" ++ " To love that well which thou must leave ere long."
```
To split a string consisting of several lines separated by linefeed characters, use `lines sonnet73`.

We will now develop a simple counting tool, for counting characters and words in a piece of text. Instead of defining two different counting functions, one for words and one for text, it is more convenient to define a single polymorphic function that can count any data type for which equality tests are possible. The data-types for which equality tests are possible are the types in class `Eq`. We use `Int` for the type of fixed precision integers. The counting function we are after has type `Eq a => a -> [a] -> Int`. What this means is that for any type `a` that is an instance of the `Eq` class, the function has type `a -> [a] -> Int`.

Objects of type `Int` are fixed precision integers. Their range can be found with:

```
Prelude> primMinInt
-2147483648
Prelude> primMaxInt
2147483647
```

Since $2147483647 = 2^{31} - 1$, we can conclude that the `hugs` implementation uses four bytes (32 bits) to represent objects of this type. `Integer` is for arbitrary precision integers: the storage space that gets allocated for `Integer` objects depends on the size of the object.

count 'e' "temperate" should give the number of e’s in `temperate`, count "thou" `sonnet18` should give the number of occurrences of "thou" in the sonnet, and so on.

We define `count` as the function that is like another function `cnt` (still to be defined) except for the fact that it takes its arguments in the inverse order. Inverting argument order is done with the predefined `flip` function. Why this argument inversion is convenient will become apparent below.

```
count :: Eq a => a -> [a] -> Int
count = flip cnt
```

We still have to define `cnt`, which should be of type `Eq a => [a] -> a -> Int`. As the type indicates, we can think of `cnt` as a map from lists over `a` to functions of type `a -> Int`. Every list should get mapped to the function yielding the correct count for all list elements and yielding 0 for every object not occurring in the list. What this means is that the empty list should yield the function that assigns 0 to any object of type `a`, and that the count function for a non-empty list should get computed from the count function for its tail. Here is the modifier on count functions that we need for that:

```
inr :: Eq a => a -> (a -> Int) -> (a -> Int)
inr c f = \ y -> if c == y then succ (f y) else f y
```
This uses the predefined function `succ` for the successor of an integer. What `incr` does for a given object `c` and count function `f` is increment the value that `f` gives for `c`, while leaving all other values unchanged.

The function that assigns 0 to anything is denoted by \_ \_ -> 0. This leads to the following definition for `cnt`:

```haskell
cnt :: Eq a => [a] -> a -> Int
cnt [] = \_ \_ -> 0
cnt (x:xs) = incr x (cnt xs)
```

The definition of `cnt` can be written in a still more compact way by using `foldr`, with \_ \_ -> 0 for the null function and `incr` for the operation on count functions.

```haskell
cnt :: Eq a => [a] -> a -> Int
cnt = foldr incr (\_ \_ -> 0)
```

Here are examples of using the `count` function, employing the predefined functions `toLower` for converting characters to lowercase, and `words` for splitting lines into words. You should check out their definitions in the Haskell prelude.

FPH> count 'e' sonnet18
63
FPH> [ count x (map toLower sonnet18) | x <- ['a'..'z'] ]
[40,5,9,20,63,10,10,31,27,0,1,23,23,33,44,4,0,29,42,40,13,5,5,1,8,0]
FPH> count "thou" (words sonnet18)
3
FPH> count "thou" (words (map toLower sonnet18))
4

Next, let us look at a function for the minimum of a list of objects. One could define such a function for integers, another one for words, and so on. Again, it is more convenient to define a polymorphic function, defined for all data-types that have an ordering relation on them. The ordered types are the types in class `Ord`. The data-types in this class are the types on which functions <= (less than or equal), >= (greater than or equal), < (less than) and > (greater than) are defined.

The function `min` is predefined for this class, as follows:
3.10. SOME TEXT PROCESSING EXAMPLES

\[
\begin{align*}
\text{min} :: & \text{Ord}\ a \Rightarrow a \to a \to a \\
\text{min} x y & | \ x \leq y \quad = x \\
& | \ \text{otherwise} \quad = y
\end{align*}
\]

To get the minimum of a non-empty list, we can use \texttt{foldr}:

\[
\begin{align*}
\text{minList} :: & \text{Ord}\ a \Rightarrow [a] \to a \\
\text{minList} (x:xs) & = \text{foldr}\ \text{min}\ x\ xs
\end{align*}
\]

Note that \texttt{minList} only is defined for non-empty lists. \texttt{minList (words sonnet18)} yields "And", \texttt{minList (words sonnet73)} yields "As".

**Exercise 3.4** Define a function \texttt{maxList} that gives the maximum of a list of objects in class \texttt{Ord}. Use the predefined function \texttt{max}.

**Conversion from Prefix to Infix in Haskell** A function can be converted to an infix operator by putting its name in back quotes, like this:

\[
\begin{align*}
Prelude> & \text{max} 4 5 \\
& 5 \\
Prelude> & 4 \text{`max`} 5 \\
& 5
\end{align*}
\]

**Conversion from Infix to Prefix, Construction of Sections** Conversely, an infix operator is converted to prefix by putting the operator in round brackets: \(\leq\) is an infix operator, and \((\leq)\) is the prefix version of the same operator.

If \texttt{op} is an infix operator, \(\texttt{(op)}\) is the prefix version of the operator. Thus, \(2\text{^10}\) can also be written as \(\texttt{(}^\text{2}\text{) 10}\). This is a special case of the use of \texttt{sections} in Haskell.

In general, if \texttt{op} is an infix operator, \(\texttt{(op x)}\) is the operation resulting from applying \texttt{op} to its right hand side argument, \(\texttt{(x op)}\) is the operation resulting from applying \texttt{op} to its left hand side argument, and \(\texttt{(op)}\) is the prefix version of the operator (this is like the abstraction of the operator from both arguments).

Thus \(\texttt{(}^\text{2}\text{)}\) is the squaring operation, \(\texttt{(2^)}\) is the operation that computes powers of 2, and \(\texttt{(}^\text{)}\) is exponentiation. Similarly, \(\texttt{(>3)}\) denotes the property of being greater than 3, \(\texttt{(3>)}\) the property of being smaller than 3, and \(\texttt{(>)}\) is the prefix version of the ‘greater than’ relation.

As a next example, let us define insertion sort by defining a function \texttt{srt} for sorting a list of items in class \texttt{Ord}.

First, define a function for inserting a new item at the correct position in an ordered list:

\[
\text{insrt} :: \text{Ord } a \Rightarrow a \to [a] \to [a] \\
\text{insrt} x [] = [x] \\
\text{insrt} x (y:ys) | x <= y = x:y:ys \\
| \text{otherwise} = y: \text{insrt} x ys
\]

Next, define insertion sort in terms of \text{insrt}, by means of \text{foldr}, as follows:

\[
\text{srt} :: \text{Ord } a \Rightarrow [a] \to [a] \\
\text{srt} = \text{foldr} \text{~insrt~} []
\]

\textbf{Exercise 3.5} Define a function \text{delete} that removes an occurrence of an object \text{x} from a list of objects in class \text{Ord}. If \text{x} does not occur in the list, the list remains unchanged. If \text{x} occurs more than once, only the first occurrence is deleted.

We define a function \text{srt2} that sorts a list of objects (in class \text{Ord}) in order of increasing size, by means of the following algorithm:

- an empty list is already sorted.
- if a list is non-empty, we put its minimum in front of the result of sorting the list that results from removing its minimum.

Note that this sorting procedure can be viewed as a kind of dual to insertion sort. It is implemented as follows:

\[
\text{srt2} :: \text{Ord } a \Rightarrow [a] \to [a] \\
\text{srt2} [] = [] \\
\text{srt2} xs = m : \text{srt2} (\text{delete} m \text{~xs}) \text{ where } m = \text{minList} \text{~xs}
\]

Here \text{delete} is the function you defined in Exercise 3.5. Note that the second clause is invoked when the first one does not apply, i.e., when the argument of \text{srt2} is not empty. This ensures that \text{delete} \text{~xs} and \text{minList} \text{~xs} never get called with an empty list argument.

Note the use of a \text{where} construction for the local definition of an auxiliary function.
Remark. Haskell has two ways to locally define auxiliary functions, the \texttt{where} and \texttt{let} constructions. The \texttt{where} construction is illustrated above, in the definition of \texttt{srt2}. This can also expressed with \texttt{let}, as follows:

\begin{verbatim}
srt2 :: [Int] -> [Int]
srt2 [] = []
srt2 xs = let
    m = minList xs
    in m : srt2 (delete m xs)
\end{verbatim}

The \texttt{let} construction uses the reserved keywords \texttt{let} and \texttt{in}.

Here is a function that calculates the average of a list of integers. The average of \( m \) and \( n \) is given by \( \frac{m+n}{2} \), the average of a list of \( k \) integers \( n_1, \ldots, n_k \) is given by \( \frac{n_1 + \cdots + n_k}{k} \). In general, averages are fractions, so the result type of \texttt{average} should not be \texttt{Int} but the Haskell data-type for floating point numbers, which is \texttt{Float}. There are predefined functions \texttt{sum} for the sum of a list of integers, and \texttt{length} for the length of a list. The Haskell operation for division \( / \) expects arguments of type \texttt{Float}, so we need a conversion function for converting \texttt{Ints} into \texttt{Floats}. This is done by \texttt{fromInt}. The function \texttt{average} can now be written as:

\begin{verbatim}
average :: [Int] -> Float
average [] = error "empty list"
average xs = fromInt (sum xs) / fromInt (length xs)
\end{verbatim}

Haskell allows a call to the \texttt{error} operation in any definition. This is used to break off operation and issue an appropriate message when \texttt{mnmInt} is applied to an empty list. Note that \texttt{error} has a parameter of type \texttt{String} (indicated by the double quotes).

\textbf{Exercise 3.6} Write a function to compute the average word length in Shakespeare’s sonnets 18 and 73. You can use \texttt{filter (‘notElem’ ‘?;:,.’)} to get rid of the punctuation signs.

Above we have seen how \texttt{sum} can be defined with \texttt{foldr}. Here is a definition of \texttt{length} in terms of \texttt{foldl}:

\begin{verbatim}
length :: [a] -> Int
length = foldl (\n    _ -> n + 1) 0
\end{verbatim}
Suppose we want to check whether a list \(xs1\) is a prefix of a list \(xs2\). Then the answer to the question \(\text{prefix} \, xs1 \, xs2\) should be either yes (true) or no (false), i.e., the type declaration for \(\text{prefix}\) should run: \(\text{prefix} :: \text{Ord} \, a => \, [a] \rightarrow [a] \rightarrow \text{Bool}\).

Prefixes of a list \(ys\) are defined as follows:

1. \([]\) is a prefix of \(ys\),
2. if \(xs\) is a prefix of \(ys\), then \(x:xs\) is a prefix of \(x:ys\),
3. nothing else is a prefix of \(ys\).

Here is the code for \(\text{prefix}\) that implements this definition:

```haskell
prefix :: Ord a => [a] -> [a] -> Bool
prefix [] ys = True
prefix (x:xs) [] = False
prefix (x:xs) (y:ys) = (x==y) && prefix xs ys
```

The definition of \(\text{prefix}\) uses the Haskell operator \&\& for conjunction.

**Exercise 3.7** Write a function \(\text{sublist}\) that checks whether \(xs1\) is a sublist of \(xs2\). The type declaration should run: \(\text{sublist} :: \text{Ord} \, a => \, [a] \rightarrow [a] \rightarrow \text{Bool}\).

The sublists of an arbitrary list \(ys\) are given by:

1. if \(xs\) is a prefix of \(ys\), \(xs\) is a sublist of \(ys\),
2. if \(ys\) equals \(y:ys'\) and \(xs\) is a sublist of \(ys'\), \(xs\) is a sublist of \(ys\),
3. nothing else is a sublist of \(ys\).

In order to remove duplicates from a list, the items in the list have to belong to a type in the \(\text{Eq}\) class, the class of data-types for which \(==\) is defined. It is standard to call the function for duplicate removal \(\text{nub}\):

```haskell
nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub (filter (\ y -> (y /= x)) xs)
```

We end with some applications of the above on Shakespeare’s sonnets.
3.11 Further Reading

The Haskell homepage \url{http://www.haskell.org} provides links to everything about Haskell that you might want to know, including information on how to download the latest version and the on-line tutorial. Among the tutorials on Haskell and Hugs that can be found on the internet, [HFP96] and [JR+] are recommended. The definitive reference for the language is [Jon03]. Recommended textbooks on functional programming in Haskell are [Bir98] and [Tho99]. A Haskell textbook focusing on multimedia applications is [Hud00].
Chapter 4

More on Type Theory

Summary

This chapter gives a quick summary of what one needs to know about type theory and its use in functional programming, for the purposes of the applications in later chapters.

4.1 The Hierarchy of Type Domains

According to Frege, the meaning of *John arrived* can be represented by a function argument expression $Aj$ where $A$ denotes a function and $j$ an argument to that function. Strictly speaking the expression $A$ does not reveal that it is supposed to combine with an individual term to form a formula (an expression denoting a truth value). One way to make this explicit is by means of lambda notation. The function expression of this example is then written as $(\lambda x.Ax)$. It is also possible to be even more explicit, and write

$$\lambda x.(Ax) :: e \rightarrow t$$

to indicate the type of the expression, or even:

$$(\lambda x.e.Ae \rightarrow t)x_{e \rightarrow t}.$$ 

The subscripts denote the *types* of the components of the expression, and the type of the whole. Much of this information is redundant, and in general we will omit type subscripts in the interest of readability.

The set of *types* over $e, t$ is given by the following BNF rule:

$$T ::= e \mid t \mid (T \rightarrow T).$$

The intended meaning is as follows. The basic type $e$ is the type of expressions denoting individual objects (or *entities*). The basic type $t$ is the type of formulas (of expressions which denote *truth values*). Complex types are the types of functions. For example, $(e \rightarrow t)$ or $e \rightarrow t$
(we assume that $\rightarrow$ is right-associative, and leave out parentheses when this does not create ambiguity) is the type of functions from entities to truth values. In general: $T_1 \rightarrow T_2$ is the type of expressions denoting functions from denotations of $T_1$ expressions to denotations of $T_2$ expressions.

It is easy to draw a concrete picture of such a type hierarchy. The types $e$ and $t$ are given. Individual objects or entities are objects taken from some domain of discussion $D$, so $e$ type expressions denote objects in $D$. The truth values are $\{0, 1\}$, so type $t$ expression denotes values in $\{0, 1\}$. For complex types we use recursion. This gives:

$$D_e = D, D_t = \{0, 1\}, D_A \rightarrow B = D_B^{D_A}.$$ 

Here $D_B^{D_A}$ denotes the set of all functions from $D_A$ to $D_B$.

**Exercise 4.1** Let a set $D_e = \{a, b, c\}$ be given. Draw a picture of an element of $D_e \rightarrow t$.

A function with range $\{0, 1\}$ is called a characteristic function, because it characterizes a set (namely, the set of those things which get mapped to 1). If $T$ is some arbitrary type, then any member of $D_T \rightarrow t$ is a characteristic function. The members of $D_e \rightarrow t$, for instance, characterize subsets of the domain of individuals $D_e$. As another example, consider $D_{(e \rightarrow t) \rightarrow t}$. According to the type definition this is the domain of functions $D_t^{D_{e \rightarrow t}}$, i.e., the functions in $\{0, 1\}^{D_{e \rightarrow t}}$. These functions characterize sets of subsets of the domain of individuals $D_e$.

**Exercise 4.2** Let a set $D_e = \{a, b, c\}$ be given. Draw a picture of an element of $D_{(e \rightarrow t) \rightarrow t}$.

As a next example, consider the domain $D_{e \rightarrow e \rightarrow t}$. This is shorthand for $D_{e \rightarrow (e \rightarrow t)}$; since we assume that $\rightarrow$ is right-associative, we can leave out the parentheses. Assume for simplicity that $D_e$ is the set $\{a, b, c\}$. Then we have:

$$D_{e \rightarrow e \rightarrow t} = D_{e \rightarrow t}^{D_e} = (D_t^{D_e})^{D_e} = (\{0, 1\}^{\{a, b, c\}})^{\{a, b, c\}}.$$ 

Let us picture a single element of $D_{e \rightarrow e \rightarrow t}$.

$$ \begin{array}{ccc}
  a & \mapsto & 1 \\
  b & \mapsto & 0 \\
  c & \mapsto & 0 \\
  a & \mapsto & 0 \\
  b & \mapsto & 1 \\
  c & \mapsto & 0 \\
  a & \mapsto & 0 \\
  b & \mapsto & 1 \\
  c & \mapsto & 1 \\
\end{array} $$

The elements of $D_{e \rightarrow e \rightarrow t}$ can in fact be regarded as functional encodings of two-placed relations $R$ on $D_e$, for a function in $D_{e \rightarrow e \rightarrow t}$ maps every element $d$ of $D_e$ to (the characteristic function of) the set of those elements of $D_e$ to which $d$ has the $R$-relation, i.e., to the set $\{x \in D_e \mid (d, x) \in R\}$. In the case of the example function we get the two-placed relation of Figure 4.1.
Exercise 4.3 Draw a picture of the function in $D_{e \rightarrow e \rightarrow t}$ corresponding with the two-placed relation of Figure 4.2.

As another example, note that $D_{t \rightarrow t}$ has precisely four members, namely:

<table>
<thead>
<tr>
<th>identity</th>
<th>negation</th>
<th>constant 1</th>
<th>constant 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $\mapsto$ 1</td>
<td>1 $\mapsto$ 0</td>
<td>1 $\mapsto$ 1</td>
<td>0 $\mapsto$ 0</td>
</tr>
<tr>
<td>0 $\mapsto$ 0</td>
<td>0 $\mapsto$ 1</td>
<td>0 $\mapsto$ 1</td>
<td>0 $\mapsto$ 0</td>
</tr>
</tbody>
</table>

The elements of $D_{t \rightarrow t}$ are functions from the set of truth values to the functions in $D_{t \rightarrow t}$, i.e., to the set of four functions pictured above. Here is an example, the function which maps 1 to the constant 1 function, and 0 to the identity:

$$
1 \mapsto \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}
$$

$$
0 \mapsto \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}
$$

Note that we can view this as a ‘two step’ version of the semantic operation of taking a disjunction: if the truth value of its first argument is 1, then the disjunction becomes true, and the truth value of the second argument does not matter (hence the constant 1 function); if the truth value of the first argument is 0, then the truth value of the disjunction as a whole is determined by the truth value of the second argument (hence the identity function).
Exercise 4.4 Specify the conjunction function in $D_{t \rightarrow t \rightarrow t}$.

Exercise 4.5 Specify the implication function in $D_{t \rightarrow t \rightarrow t}$. Assume that the first argument of the function is the antecedent of the implication.

Now that we know in principle what the type domains $D_T$ look like, for every type $T$ in the type hierarchy, it should be clear that the process of abstraction (creating new functions) brings one higher up in the hierarchy, while the operation of application (applying a function to an argument) brings one down in the hierarchy. If $Px$ is an expression of type $t$ and $x$ is of type $e$, then $(\lambda x.Px)$ is an expression of type $e \rightarrow t$. In fact, $(\lambda x.Px)$ is nothing but a more explicit notation for the so-called set abstraction notation $\{ x \mid Px \}$.

Similarly, if $P$ is an expression of type $e \rightarrow t$ and $x$ is of type $e$, then $(Px)$ denotes the application of $P$ to $x$; it is an expression of type $t$: application brings one down in the hierarchy. The compositional functional viewpoint demands that we write everything as function application.

4.1 Jan admires the Dalai Lama.

Thus, the natural language sentence 4.1 will get represented as $((Aj)d)$, where $j$ is an argument of the functional expression $A$, and $d$ in turn is an argument of the functional expression $(Aj)$. More precisely, $A$ is an expression of type $e \rightarrow e \rightarrow t$, and $j$ and $d$ are both of type $e$. The property of admiring the Dalai Lama is expressed by $(\lambda x.((Ax)d))$, the property of being admired by Jan is expressed by $(\lambda x.((Aj)x))$. The property of admiring oneself is expressed by $(\lambda x.((Ax)x))$.

Finally, the property of being a property of Jan is given by $\lambda X.((Xj)d)$, and the property of being a relation between Jan and the Dalai Lama is given by $\lambda Y.((Yj)d)$. Note that, if $X$ in $\lambda X.((Xj)d)$ is of type $e \rightarrow t$, as it should be to match the type $e$ of $j$, then the expression $\lambda X.((Xj))$ has type $(e \rightarrow t) \rightarrow t$. Similarly, $Y$ in $\lambda Y.((Yj)d)$ should have type $e \rightarrow e \rightarrow t$, so expression $\lambda Y.((Yj)d)$ itself has type $(e \rightarrow e \rightarrow t) \rightarrow t$.

The lavish use of parentheses is the trade mark of the so-called lambda calculus, a logical framework that is the basis of functional programming languages like LISP, ML, and Haskell. Haskell economizes on parentheses by stipulating that function application associates to the left.

We finish this section with some more examples of the definition of abstracts. In natural language it is possible to refer to properties as subjects, e.g., when we say things like (4.2) and (4.3)

4.2 Smoking is dangerous.

4.3 Staring at someone is impolite.

Choosing constants and variables of appropriate types, we can translate the first of these as $(D(\lambda x.(Sx)))$, with $D$ a constant of type $(e \rightarrow t) \rightarrow t$ and $S$ a constant of type $e \rightarrow t$, and the second as $(I(\lambda x.(\exists y((Sy)x))))$, with $I$ of type $(e \rightarrow t) \rightarrow t$ and $S$ of type $e \rightarrow e \rightarrow t$. 
4.2 Connection with Typing in Haskell

In this section we connect the above to how types appear in Haskell. The text inside the boxes is literal Haskell code. For convenience we collect the code of every chapter in a module. The present module is called MOTT, after the chapter title. It imports a module Domain containing the specification for a domain of entities (see below) and a module Model with a definition of a first order model in Haskell (also below).

```haskell
module MOTT where

import List
import Domain
import Model
```

In Haskell, the type for truthvalues is called Bool. Thus, the type for the constant function that maps all truth values to 1 is \( \text{Bool} \rightarrow \text{Bool} \). The truth value 1 appears in Haskell as True, the truth value 0 as False. The definitions of the constant True and the constant False function in Haskell run as follows:

```haskell
c1 :: Bool -> Bool
  c1 _ = True

c0 :: Bool -> Bool
  c0 _ = False
```

The definitions consist of a type declaration together with a value specification. What the value specification for \( c_1 \) says is that whatever argument you feed \( c_1 \), the result will be True. Similarly, the specification for \( c_0 \) says that whatever argument you give \( c_0 \), the result will be False. The underscore _ is used for any value whatsoever. An equivalent specification of these functions runs as follows:

```haskell
c1 :: Bool -> Bool
  c1 = \ p -> True

c0 :: Bool -> Bool
  c0 = \ p -> False
```

The function in \( \text{Bool} \rightarrow \text{Bool} \) that swaps the two truth values is predefined in Haskell as not. The identity function is predefined in Haskell as id. This function has the peculiarity that it
has *polymorphic type*: it can be used as the identity for *any* type for which individuation makes sense.

In Haskell, the disjunction function will have type

\[ \text{Bool} \to \text{Bool} \to \text{Bool} \]

This type is read as \( \text{Bool} \to (\text{Bool} \to \text{Bool}) \), because of the convention that the type forming operation \( \to \) associates to the right. Here is the implementation of \text{disj}, in terms of \text{c1} and \text{id}, the Haskell identity function.

```haskell
disj :: Bool -> Bool -> Bool
disj True = c1
disj False = id
```

Alternatively, we might have specified this as follows:

```haskell
disj :: Bool -> Bool -> Bool
disj = \ p -> if p then c1 else id
```

This operator behaves in fact as a home-made version of \( (\|) \).

### 4.3 Representing a Domain of Entities

To illustrate the type \( e \), we construct a small example domain of entities consisting of individuals \( A, \ldots, Z \), by declaring a datatype \text{Entity}.

```haskell
module Domain where

import List

data Entity = A | B | C | D | E | F | G
            | H | I | J | K | L | M | N
            | O | P | Q | R | S | T | U
            | V | W | X | Y | Z | Unspec
 deriving (Eq,Bounded,Enum)
```
4.3. REPRESENTING A DOMAIN OF ENTITIES

The special entity \texttt{Unspec} will play an important part in our account of underspecified relations. It will allow us to define relations with some argument places unspecified.

The stuff about \texttt{deriving (Eq,Bounded,Enum)} is there to enable us to do equality tests on entities (\texttt{Eq}), to refer to \texttt{A} as the minimum element and \texttt{M} as the maximum element (\texttt{Bounded}), and to enumerate the elements (\texttt{Enum}).

To display the entities on the screen, we put the entities in the class \texttt{Show}, and define a \texttt{show} function for it. This displays the special entity \texttt{Unspec} as \texttt{*}.

\begin{verbatim}
instance Show Entity where
  show (A) = "A"; show (B) = "B"; show (C) = "C";
  show (D) = "D"; show (E) = "E"; show (F) = "F";
  show (G) = "G"; show (H) = "H"; show (I) = "I";
  show (J) = "J"; show (K) = "K"; show (L) = "L";
  show (M) = "M"; show (N) = "N"; show (O) = "O";
  show (P) = "P"; show (Q) = "Q"; show (R) = "R";
  show (S) = "S"; show (T) = "T"; show (U) = "U";
  show (V) = "V"; show (W) = "W"; show (X) = "X";
  show (Y) = "Y"; show (Z) = "Z"; show (Unspec) = "*"

Because \texttt{Entity} is a bounded and enumerable type, we can put all of its elements in a finite list:

\begin{verbatim}
entities :: [Entity]
entities = [minBound..maxBound]
\end{verbatim}

Here is an example relation:

\begin{verbatim}
rel1 :: Entity -> Entity -> Bool
rel1 A A = True
rel1 B A = True
rel1 D A = True
rel1 C B = True
rel1 C C = True
rel1 C D = True
rel1 _ _ = False
\end{verbatim}

Arity reduction on binary functions can be done with:
The following definition picks the reflexive part out of \texttt{rel1}:

\begin{center}
\begin{verbatim}
rel2 = self rel1
\end{verbatim}
\end{center}

The newly defined operations can be executed once the module \texttt{Domain.hs} is loaded, as follows:

\[
\begin{array}{l}
\text{Domain}\textgreater \text{entities} \\
\text{Domain}\textgreater \text{rel1 A B} \\
\text{False} \\
\text{Domain}\textgreater \text{rel1 A A} \\
\text{True} \\
\text{Domain}\textgreater \text{rel2 A} \\
\text{True}
\end{array}
\]

Let us do some small experiments with the stuff we implemented so far. First, because \texttt{Entity} is a bounded and enumerable type, we can enumerate its elements in a list:

\[
\begin{array}{l}
\text{Domain}\textgreater \ [A .. Unspec] \\
\end{array}
\]

Using lambda abstraction, we can select sublists of this list that satisfy some property we are interested in. Suppose we are interested in the property of being related by \texttt{r} to the element \texttt{B}. This property can be expressed using \texttt{\lambda} abstraction as \texttt{\lambda x.R x b}. In Haskell, this same property is expressed as \texttt{(\ x \rightarrow \texttt{rel1 C x})}. Similarly, \texttt{(\ x \rightarrow \texttt{rel1 C x})} expresses the property of being an \texttt{r}-successor of \texttt{C}. Using a property to create a sublist is done by means of the \texttt{filter} operation. Here is how we use \texttt{filter} to find the list of entities satisfying the property:

\[
\begin{array}{l}
\text{Domain}\textgreater \text{filter (\ x \rightarrow \texttt{rel1 C x}) [A .. Unspec]} \\
[B, C, D]
\end{array}
\]

Instead of \texttt{[A .. Unspec]} we can use \texttt{entities}:

\[
\begin{array}{l}
\text{Domain}\textgreater \text{filter (\ x \rightarrow \texttt{rel1 C x}) entities} \\
[B, C, D]
\end{array}
\]

With negation, we can express the complement of this property. This gives:
Finally, here is how \( \lor \) (disjunction) is used to express the property of being either an \( r \)-successor of \( A \) or of \( C \):

\[
\text{filter} \left( \lambda x \rightarrow \text{rel1} A \, x \lor \text{rel1} C \, x \right) \text{ entities}
\]

\([A,B,C,D]"

With the predefined Haskell infix operator \&\& we can specify conjunctive properties.

### 4.4 The Language of Typed Logic and Its Semantics

In this section we will give formal definitions of the language of typed logic (or typed lambda calculus) and its semantics, using Haskell counterparts and implementations to illustrate the theory.

Assume that we have constants and variables available for all types in the type hierarchy. Then the language of typed logic over these is defined as follows.

\[
\begin{align*}
\text{type} &::= e \mid t \mid (\text{type} \rightarrow \text{type}) \\
\text{expression} &::= \text{constant}_{\text{type}} \mid \text{variable}_{\text{type}} \mid \\
& \quad (\lambda \text{variable}_{\text{type}_1}.\text{expression}_{\text{type}_2})(\text{type}_1 \rightarrow \text{type}_2) \mid \\
& \quad (\text{expression}_{\text{type}_1 \rightarrow \text{type}_2} \text{expression}_{\text{type}_1})_{\text{type}_2}
\end{align*}
\]

Note that for an expression of the form \((E_1 E_2)\) to be well-typed the types have to match, and that the type of the resulting expression is fully determined by the types of the components. Similarly, the type of a lambda expression \((\lambda v.E)\) is fully determined by the types of \( v \) and \( E \).

**Exercise 4.6** Assume constant \( A \) has type \( e \rightarrow t \) and constant \( B \) has type \( (e \rightarrow t) \rightarrow t \). Variable \( x \) has type \( e \), variable \( Y \) has type \( e \rightarrow t \). Which of the following expressions are well-typed?

1. \((\lambda x.(Ax)))\).
2. \((B(\lambda x.(Ax))))\).
3. \((\lambda Y.(Y(\lambda x.(Ax))))))\).
4. \((\lambda Y.(BY)))\).

Often we leave out the type information. The definition of the language then looks like this:

\[
\text{expression} ::= \text{constant} \mid \text{variable} \mid (\lambda \text{variable}.\text{expression}) \mid (\text{expression} \text{expression})
\]

A model \( M \) for typed logic consists of a domain \( \text{dom}(M) = D_e \) together with an interpretation function \( \text{int}(M) = I \) which maps every constant of the language to a function of the appropriate
type in the domain hierarchy based on $D_c$. A variable assignment $s$ for typed logic maps every variable of the language to a function of the appropriate type in the domain hierarchy. The semantics for the language is given by defining a function $[[\cdot]]_s^M$ which maps every expression of the language to a function of the appropriate type.

$$[[\text{constant}}]]_s^M = I(\text{constant}).$$

$$[[\text{variable}}]]_s^M = s(\text{variable}).$$

$$[[\lambda v_{T_1}, E_{T_2}]_s^M] = h,$$

where $h$ is the function given by $h : d \in D_{T_1} \mapsto [[E]]_s^[M(v[d)] \in D_{T_2}.$

$$[[E_1 \cdot E_2]]_s^M = [[E_1]]_s^M [[E_2]]_s^M.$$ 

**Exercise 4.7** Assume that $G(a, b, c)$ expresses that $a$ gives $b$ to $c$. What do the following expressions say:

1. $\lambda x.G(x, b, c)$.
2. $\lambda x.G(a, x, c)$.
3. $\lambda x.G(a, b, x)$.
4. $\lambda x.G(x, b, x)$.

**Exercise 4.8** Assume that $G(a, b, c)$ expresses that $a$ gives $b$ to $c$. What do the following expressions say:

1. $\lambda x.(\lambda y.G(y, b, x))$.
2. $\lambda x.(\lambda y.G(x, b, y))$.

In fact, the logical constants of predicate logic can be viewed as constants of typed logic, as follows. $\neg$ is a constant of type $t \rightarrow t$ with the following interpretation.

- $[[\neg]] = h,$ where $h$ is the function in $\{0, 1\}^{\{0, 1\}}$ which maps $0$ to $1$ and vice versa.

As we have seen already, $\land$ and $\lor$ are constants of type $t \rightarrow t \rightarrow t$ with the following interpretations.

- $[[\land]] = h,$ where $h$ is the function in $\{0, 1\}^{\{0, 1\}\{0, 1\}}$ which maps $1$ to $\{(1, 1), (0, 0)\}$ and $0$ to $\{(1, 0), (0, 0)\}$.
- $[[\lor]] = h,$ where $h$ is the function in $\{0, 1\}^{\{0, 1\}\{0, 1\}}$ which maps $1$ to $\{(1, 1), (0, 1)\}$ and $0$ to $\{(1, 1), (0, 0)\}$.

**Exercise 4.9** Give the interpretation of the material implication constant $\rightarrow$ in typed logic.

**Exercise 4.10** Give the interpretation of the material equivalence constant $\leftrightarrow$ in typed logic.
4.4. THE LANGUAGE OF TYPED LOGIC AND ITS SEMANTICS

The quantifiers $\exists$ and $\forall$ are constants of type $(e \rightarrow t) \rightarrow t$, with the following interpretations.

- $[\forall] = h$, where $h$ is the function in $\{0, 1\}^{D_{e\rightarrow t}}$ which maps the function that characterizes $D_e$ to 1 and every other characteristic function to 0.
- $[\exists] = h$, where $h$ is the function in $\{0, 1\}^{D_{e\rightarrow t}}$ which maps the function that characterizes $\emptyset$ to 0 and every other characteristic function to 1.

It is possible to add constants for quantification over different types. E.g., to express second order quantification (i.e., quantification over properties of things), one would need quantifier constants of type $((e \rightarrow t) \rightarrow t) \rightarrow t$.

**Exercise 4.11** What is the type of binary generalized quantifiers such as $Q_\forall(\lambda x.Px)(\lambda x.Qx)$

('all P are Q'),

$Q_\exists(\lambda x.Px)(\lambda x.Qx)$

('some P are Q'), and

$Q_M(\lambda x.Px)(\lambda x.Qx)$

('most P are Q')?

We will assume that for every type $T$ built from $e$ and $t$ we have a constant $i_{T\rightarrow T\rightarrow t}$ available to express the identity of two objects of type $T$.

Then $i_{e\rightarrow e\rightarrow t}$ denotes identity of individual objects, and it is convenient to abbreviate $(ib)a$, for $a, b$ of type $e$, as $a = b$.

**Exercise 4.12** Write out $i_{t\rightarrow t\rightarrow t}$. Which two-placed connective does this constant express?

With these constants added, the language of typed logic defined above contains ordinary predicate logic as a proper fragment. Here are some example expressions of typed logic, with their predicate logical counterparts:

$$(\neg (Px)) \quad \neg Px$$

$$( (\land (Px))(Qy)) \quad Px \land Qy$$

$$(\forall (\lambda x.Px)) \quad \forall x Px$$

$$(\forall (\lambda x.(\exists (\lambda y.(Rxy))))) \quad \forall x \exists y Rx$$

It is possible to reduce the difference in syntactic appearance between ordinary predicate logic and typed logic by means of some abbreviation conventions. Occasionally we will write

$$(\lambda x.(\lambda y.E))$$

as

$$\lambda xy.E,$$
and similarly for three or more successive lambda abstractions. Also, an expression of the form

\(((Ea)b)c\)

can be written as

\(E\ a\ b\ c\)

to make it resemble predicate logical notation more closely (similarly for different numbers of arguments; note the different order of the arguments). Finally, it is convenient to write

\(((\land E_1)E_2)\)

as

\(E_1 \land E_2\)

and similarly for \(\lor, \rightarrow\) and \(\leftrightarrow\), and to omit outermost parentheses. These conventions are similar to the Haskell operator binding conventions.

Note that although \(((Ea)b)c\) may be written as \(E\ a\ b\ c\), this is not the same as \(E(a,b,c)\). The difference is that in \(E\ a\ b\ c\), the arguments of the function \(E\) are consumed one by one, while in \(E(a,b,c)\), the function \(E\) takes a single argument which is a triple.

Technically, the process of converting \(((Ea)b)c\) to \(E(a,b,c)\) is done by means of a function that puts the arguments in an ordered triple.

The conversion of a function of type \(a \to b \to c\) to one of type \((a,b) \to c\) is called uncurrying, and the conversion in the other direction is called currying. Just like the programming language Haskell, these operations are named after one of the inventors of lambda calculus, the logician Haskell Curry. The operations are predefined in Haskell. Thus, we have:

```
Domain> rel1 C B
True
Domain> (uncurry rel1) (C,B)
True
```

curry and uncurry are predefined in the Haskell prelude, as follows:

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y

curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p = f (fst p) (snd p)
```
4.4. THE LANGUAGE OF TYPED LOGIC AND ITS SEMANTICS

For the case of three arguments, we need our own definition.

\[
\text{curry3} :: ((a,b,c) \rightarrow d) \rightarrow a \rightarrow b \rightarrow c \rightarrow d
\]
\[
\text{curry3} f x y z = f (x,y,z)
\]

\[
\text{uncurry3} :: (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow ((a,b,c) \rightarrow d)
\]
\[
\text{uncurry3} f (x,y,z) = f x y z
\]

To demonstrate the use of this, here is a definition of a two-placed relation in Haskell:

\[
\text{rel3} :: \text{Entity} \rightarrow \text{Entity} \rightarrow \text{Entity} \rightarrow \text{Bool}
\]
\[
\text{rel3} \ D \ x \ y = \text{rel1} \ x \ y
\]
\[
\text{rel3} \ E \ x \ y = \text{not} (\text{rel1} \ x \ y)
\]
\[
\text{rel3} \ _ \ _ \ _ = \text{False}
\]

If you want to execute this, you will need the code for the whole chapter, in the modules \texttt{MOTT.hs}, \texttt{Domain.hs} and \texttt{Model.hs}. Download these files, put them in the same directory, start up the Haskell interpreter, and load the module \texttt{MOTT.hs}. The prompt \texttt{MOTT>} will appear, and you will be able to check the definition of \texttt{rel3}:

\texttt{MOTT> rel3 A B D}
\texttt{False}
\texttt{MOTT> rel3 D B A}
\texttt{True}
\texttt{MOTT> uncurry3 rel3 (D,B,A)}
\texttt{True}

Assume \( R \) is a typed logical constant of type \( e \rightarrow e \rightarrow t \) which is interpreted as the two-placed relation of respect between individuals. Then \((Ra)b\) expresses that \( a \) respects \( b \). In abbreviated notation this becomes \( R \ a \ b \), in uncurried notation \( R(a, b) \). Now \( \lambda x.\exists yR \ x \ y \) is abbreviated notation for ‘respecting someone’, and \( \lambda x.\exists yR \ x \ y \) for ‘being respected by someone’. If \( G(a, b, c) \) is uncurried notation for ‘\( a \) gives \( b \) to \( c \)’, then \( \lambda x.\exists y\exists zG(x, y, z) \) expresses ‘giving something to someone’, and \( \lambda x.\exists yG(x, b, y) \) expresses ‘giving \( b \) to someone’. Finally, \( \lambda x.\exists yG(y, b, x) \) expresses ‘receiving \( b \) from someone’.

**Exercise 4.13** Assume \( G(x, y, z) \) means that \( x \) gives \( y \) to \( z \). Use this to find a typed logic expression for ‘receiving something from someone’.

Suppose we want to translate ‘Anne gave Claire the book’, which has syntactic structure

\[ [S[_{NP}\text{Anne }][_{VP}\text{[DTV gave]}][_{NP}\text{Claire }][_{NP}\text{the book }]]] \]
in a compositional way, using \( \lambda zyx.G(x, y, z) \) as translation for ‘give’. Translating all the combinations of phrases as function argument combination, we arrive at the following translation of the sentence.

\[ (((\lambda zyx.G(x, y, z)c)b)a) \]

This does indeed express what we want, but in a rather roundabout way. We would like to reduce this expression to \( G(a, b, c) \). The subject of reducing expressions of typed logic is taken up in the next section.

### 4.5 Reducing Expressions of Typed Logic

To reduce expression (4.4) from the previous section to its simplest form, three steps of so-called \( \beta \) conversion are needed. During \( \beta \) conversion of an expression consisting of a functor expression \( \lambda v.E \) followed by an argument expression \( A \), basically the following happens (we will state a proviso shortly). The prefix \( \lambda v. \) is removed, the argument expression \( A \) is removed, and finally the argument expression \( A \) is substituted in \( E \) for all free occurrences of \( v \). The free occurrences of \( v \) in \( E \) are precisely the occurrences which were bound by \( \lambda v. \) in \( \lambda v.E \).

Here is the proviso. In some cases, the substitution process described above cannot be applied without further ado, because it will result in unintended capturing of variables within the argument expression \( A \). Consider expression (4.5) (we use unabbreviated notation again, because the syntactic details do matter now).

\[ (((x:\!(y:\!(Ry))\!))\!y) \]

In this expression, \( y \) is bound in the functional part \( (\lambda x.(\lambda y.((Ry)x))) \) but free in the argument part \( y \). Reducing (4.5) by \( \beta \) conversion according to the recipe given above would result in \( (\lambda y.((Ry)y)) \), with capture of the argument \( y \) at the place where it is substituted for \( x \). This problem can be avoided by performing \( \beta \) conversion on an alphabetic variant of the original expression, say on (4.6).

\[ (((\lambda x.(\lambda z.((Rz)x)))\!y) \]

Another example where \( \alpha \) conversion (i.e., switching to an alphabetic variant) is necessary before \( \beta \) conversion to prevent unintended capture of free variables is the expression (4.7).

\[ (((\lambda p.\!\forall x((Ax)\!\leftrightarrow\!p))(Bx)) \]

In (4.7), \( p \) is a variable of type \( t \), and \( x \) one of type \( e \). Variable \( x \) is bound inside the functional part \( (\lambda p.\!\forall x((Ax)\!\leftrightarrow\!p)) \) but free in the argument part \( (Bx) \). Substituting \( (Bx) \) for \( p \) in the function expression would cause \( x \) to be captured, with failure to preserve the original meaning. Again, the problem is avoided if \( \beta \) conversion is performed on an alphabetic variant of the original expression, say on (4.8).
4.8 \((\lambda p.\forall z((Az) \leftrightarrow p))(Bx))\).

Performing \(\beta\) reduction on (4.8) yields \(\forall z.((Az) \leftrightarrow (Bx))\), with the argument of \(B\) still free, as it should be.

To state all this in a formally precise way, we start with a definition of ‘variable \(v\) is free in expression \(E\)’. \(v\) is free in \(E\) if the following holds (we use \(\approx\) for the relation of being syntactically identical, i.e. for being the same expression):

- \(v \approx E\),
- \(E \approx (E_1E_2)\), and \(v\) is free in \(E_1\) or \(v\) is free in \(E_2\),
- \(E \approx (\lambda x.E_1)\), and \(v \not\approx x\), and \(v\) is free in \(E_1\).

Exercise 4.14 Which occurrences of \(x\) are free?

1. \((\lambda x.(Px))\),
2. \(((\lambda x.(Px))x)\),
3. \((\lambda x.((Rx)x))\),
4. \(((\lambda x.((Rx)x))x)\),
5. \(((\lambda y.((Rx)y))x)\).

Exercise 4.15 Same question for \(((\lambda y.\exists x((Rx)y))x)\), where one should bear in mind that \(\exists x((Rx)y)\) is shorthand for \((\exists(\lambda x.((Rx)y)))\).

Next, the definition of substitution of \(s\) for free occurrences of \(v\) in \(E\), with notation \([v \mapsto s]E\).

- If \(E \approx v\) then \([v \mapsto s]E \approx s\),
  - if \(E \approx x \not\approx v\) (i.e., \(E\) is a variable different from \(v\)), then \([v \mapsto s]E \approx x\),
  - if \(E \approx c\) (i.e., \(E\) is a constant, and therefore different from \(v\)), then \([v \mapsto s]E \approx c\),
- If \(E \approx (E_1E_2)\) then \([v \mapsto s]E \approx ([v \mapsto s]E_1 [v \mapsto s]E_2)\),
- If \(E \approx (\lambda x.E_1)\), then
  - If \(v \approx x\) then \([v \mapsto s]E \approx E\),
  - if \(v \not\approx x\) then there are two cases:
    1. if \(x\) is not free in \(s\) or \(v\) is not free in \(E\) then \([v \mapsto s]E \approx (\lambda x.[v \mapsto s]E_1)\),
    2. if \(x\) is free in \(s\) and \(v\) is free in \(E\) then \([v \mapsto s]E \approx (\lambda y.[v \mapsto s][x \mapsto y]E_1)\), for some \(y\) which is not free in \(s\) and not free in \(E_1\).

This definition is rather involved. Especially the distinction within the final case in two subcases (1) and (2) may look like an overcomplication. To see that it is not, consider the following simple example.
4.9 \([x \mapsto y](\lambda y.(Px)).\)

The function \((\lambda y.(Px))\) is the function yielding \((Px)\) for any argument, i.e., the constant \((Px)\) function. If we apply the above definition without taking subcase (2) into account, we get:

\[
[x \mapsto y](\lambda y.(Px)) \approx (\lambda y.[x \mapsto y](Px)) \approx (\lambda y.(Py)).
\]

This is a completely different function, namely the function that assigns to argument \(a\) the result of applying \(P\) to \(a\). By using subcase (2) we do get the correct result:

\[
[x \mapsto y](\lambda y.(Px)) \approx (\lambda z.[x \mapsto y][y \mapsto z](Px)) \approx (\lambda z.[x \mapsto y](Px)) \approx (\lambda z.(Py)).
\]

Finally, here is the definition of reduction, which comes in three flavours: \(\beta\) reduction, \(\alpha\) reduction and \(\eta\) reduction, for which we use arrows \(\beta\), \(\alpha\) and \(\eta\). Here are the definitions:

**Beta reduction:** \((\lambda v.E)s \xrightarrow{\beta} [v \mapsto s]E.\)

Condition: \(v\) and \(s\) are of the same type (otherwise the expression to be reduced is not welltyped).

**Alpha reduction:** \((\lambda v.E) \xrightarrow{\alpha} (\lambda x.[v \mapsto x]E).\)

Conditions: \(v\) and \(x\) are of the same type, and \(x\) is not free in \(E\).

**Eta reduction:** \(((\lambda v.E)v) \xrightarrow{\eta} E.\)

The ‘real work’ takes place during \(\beta\) reduction. The \(\alpha\) reduction rule serves only to state in an explicit fashion that \(\lambda\) calculations are insensitive to switches to alphabetic variants. Whether one uses \(\lambda x\) to bind occurrences of \(x\) or \(\lambda y\) to bind occurrences of \(y\) is immaterial, just like it is immaterial in the case of predicate logic whether one writes \(\forall xPx\) or \(\forall yPy\). The \(\eta\) reduction rule makes a principle explicit that we have used implicitly all the time: if \((Pj)\) expresses that John is present, then both \(P\) and \((\lambda x.(Px))\) express the property of being present. This is so because \(((\lambda x.(Px))x) \xrightarrow{\eta} (Px)\), so \(P\) and \((\lambda x.(Px))\) give the same result when applied to argument \(x\), i.e., they express the same function. Applying \(\beta\) reduction to

\[
(((\lambda zyx.G(x,y,z)c)b)a),
\]

or in unabbreviated notation

\[
(((\lambda z.(\lambda y.(\lambda x.G(x,y,z)))c)b)a)
\]

gives:

\[
(((\lambda z.(\lambda y.(\lambda x.G(x,y,z)))c)b)a) \xrightarrow{\beta} (((\lambda y.(\lambda x.G(x,y,c)))b)a) \xrightarrow{\beta}
\]

\[
\xrightarrow{\beta} (\lambda x.G(x,b,c))a \xrightarrow{\beta} G(a,b,c).
\]

This is precisely what we want. To be fully precise we have to state explicitly that expressions can be reduced ‘in context’. The following principles express this:
Here $F$ is assumed to have the appropriate type, of course. These principles allow $\beta$ reductions at arbitrary depth within expressions.

**Exercise 4.16** Reduce the following expressions to their simplest forms:

1. $((\lambda x.(Yx))P)$.
2. $(((\lambda x.(Yx)))P)y$.
3. $((\lambda P,(\lambda Q,\exists x (Px \land Qx)))A)$.
4. $(((\lambda P,(\lambda Q,\exists x (Px \land Qx)))A)B)$.
5. $((\lambda P,(\lambda Q,\forall x (Px \rightarrow Qx)))(\lambda y,(\lambda x.R(x,y)))j))$.

Our type system with matching lambda language may seem complicated at first sight, but in fact it is a very elegant system, with some special properties that make it easy to handle. In the statement of the following property, we write $E \twoheadrightarrow E'$ for $E$ reduces in a number of $\alpha, \beta, \eta$ steps to $E'$.

**Confluence property (or: Church-Rosser property):** For all expressions $E, E_1, E_2$ of typed logic: if $E \twoheadrightarrow E_1$ and $E \twoheadrightarrow E_2$ then there is an expression $F$ with $E_1 \twoheadrightarrow F$ and $E_2 \twoheadrightarrow F$.

An expression of the form $((\lambda v.E)s)$ is called a $\beta$-redex (for: $\beta$ reducible expression). $[v \mapsto s]E$ is called the contractum of $((\lambda v.E)s)$. An expression that does not contain any redexes is called a normal form.

**Normal form property:** Every expression of typed logic can be reduced to a normal form.

Combining the confluence property and the normal form property we get that the normal forms of an expression $E$ are identical modulo $\alpha$ conversion. That is to say, all normal forms of $E$ are alphabetic variants of one another.

The normal form property holds thanks to the restrictions imposed by the typing discipline. Untyped lambda calculus lacks this property. In untyped lambda calculus it is allowed to apply expressions to themselves. In typed lambda calculus this is forbidden, because $(XX)$ cannot be consistently typed.

**Exercise 4.17** In untyped lambda calculus, expressions like $(\lambda x.(xx))$ are wellformed. Show that

$((\lambda x.(xx))(\lambda x.(xx)))$

does not have a normal form.
The process of reducing lambda expressions has drastic consequences for their syntactic appearance. Historically, the syntactic form of logical expressions translating natural language sentences was taken to reflect the *logical form* of these sentences. In Chapter 6 it is pointed out that the metamorphosis of $\beta$ conversion bears on certain historical problems of logical form.

### 4.6 Further Reading

A good introduction to type theory is given in [Hin97]. Type polymorphism was proposed in [Mil78]. An account of the typed logic behind Montague grammar is given in [Gal75]. Applications of type theory in natural language analysis are the broad topic of [Ben91].
Chapter 5

Evaluating Predicate Logic in Haskell

Summary

In this chapter, we demonstrate how models of predicate logic are defined as Haskell datatypes. Next, we turn to the problem of interpreting predicate logical languages in appropriate models.

5.1 Representing a Model for Predicate Logic

All we need to specify a first order model in Haskell is a domain of entities and suitable interpretations of proper names and predicates. The domain of entities is specified in the module Domain as Entity, so we import that module.

```haskell
module Model where
import Domain
```

Interpretations for proper names and definite descriptions:

```haskell
ann, bill, lucy, mary, johnny :: Entity
ann = A; bill = B; lucy = L
mary = M; johnny = J
```

Here is a conversion function for an easy specification of predicates:
list2pred :: Eq a => [a] -> a -> Bool
list2pred = flip elem

This uses the predefined functions `flip` and `elem`. `flip` is given in the Haskell Prelude as:

\[
flip :: (a -> b -> c) -> b -> a -> c \\
flip f x y = f y x
\]

Interpretations for predicates:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>man</td>
<td>list2pred [B,J]</td>
</tr>
<tr>
<td>boy</td>
<td>list2pred [J]</td>
</tr>
<tr>
<td>woman</td>
<td>list2pred [A,C,M,L]</td>
</tr>
<tr>
<td>tree</td>
<td>list2pred [T,U,V]</td>
</tr>
<tr>
<td>house</td>
<td>list2pred [H,K]</td>
</tr>
<tr>
<td>leaf</td>
<td>list2pred [X,Y,Z]</td>
</tr>
<tr>
<td>stone</td>
<td>list2pred [S]</td>
</tr>
<tr>
<td>gun</td>
<td>list2pred [G]</td>
</tr>
</tbody>
</table>

What a person or a thing is can be defined in terms of the predicates defined so far: a person is a man or a woman, and a thing is everything which is neither a person nor the special object `Unspec`. This gives `Unspec` the special status that we need for a proper definition of semantic operation.

\[
\text{person} = \lambda x \rightarrow (\text{man } x \lor \text{woman } x) \\
\text{thing} = \lambda x \rightarrow \neg(\text{person } x \lor x == \text{Unspec})
\]

Meanings for intransitive verbs, represented as `Entity -> Bool`.

<table>
<thead>
<tr>
<th>Verb</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>laugh</td>
<td>list2pred [M]</td>
</tr>
<tr>
<td>smile</td>
<td>list2pred [A,B,J,M]</td>
</tr>
</tbody>
</table>
5.1. REPRESENTING A MODEL FOR PREDICATE LOGIC

For an easy specification of binary relations (two-placed predicates), we model them as type 
\((a,a) -> \text{Bool}\), more specifically as \((\text{Entity}, \text{Entity}) -> \text{Bool}\).

Note the use of Unspec for defining an underspecified relation for *drop*.

\[
\begin{align*}
\text{love, respect, hate, own, wash, shave, drop0} : & (\text{Entity, Entity}) \rightarrow \text{Bool} \\
\text{love} & = \text{list2pred } [(B,M),(J,M),(J,J),(M,J),(A,J),(B,J)] \\
\text{respect} & = \text{list2pred } [(x,x) \mid x \leftarrow \text{entities}, \text{person } x] \\
\text{hate} & = \text{list2pred } [(x,B) \mid x \leftarrow \text{entities}, \text{woman } x] \\
\text{own} & = \text{list2pred } [(M,H)] \\
\text{wash} & = \text{list2pred } [(A,A),(A,J),(L,L),(B,B),(M,M)] \\
\text{shave} & = \text{list2pred } [(A,J),(B,B)] \\
\text{drop0} & = \text{list2pred } [(T,X),(U,Y),(U,Z),(\text{Unspec},V)]
\end{align*}
\]

For an easy specification of ternary relations (or three-placed predicates) we use the type 
\((a,a,a) -> \text{Bool}\), or in this particular case \((\text{Entity, Entity, Entity}) \rightarrow \text{Bool}\). We will interpret the transitive verbs *kill* and *break* as three-placed relations, with the first argument giving the agent, the second argument the patient, and the third argument the instrument. In cases where there is no instrument, we leave the third argument unspecified, in cases where there is no agent the first.

\[
\begin{align*}
\text{break0, kill} : & (\text{Entity, Entity, Entity}) \rightarrow \text{Bool} \\
\text{break0} & = \text{list2pred } [(M,V,S),(J,W,G)] \\
\text{kill} & = \text{list2pred } [(M,L,G),(\text{Unspec},A,D),(\text{Unspec},J,\text{Unspec})]
\end{align*}
\]

What the first entry says is that Mary broke \(V\) with \(S\), and Johnny \(W\) with \(G\). The second entry conveys that Mary killed Lucy with \(G\), Ann died of \(D\), and Johnny just died.

The verbs *give* and *sell* are also interpreted as ternary relations.

\[
\begin{align*}
\text{give, sell} : & (\text{Entity, Entity, Entity}) \rightarrow \text{Bool} \\
\text{give} & = \text{list2pred } [(M,V,L),(L,G,M)] \\
\text{sell} & = \text{list2pred } [(J,J,M),(J,T,M),(A,U,M)]
\end{align*}
\]

Finally, here are the (un)curry conversion functions for ternary relations:
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
curry3 f x y z = f (x,y,z)

uncurry3 :: (a -> b -> c -> d) -> ((a,b,c) -> d)
uncurry3 f (x,y,z) = f x y z

Exercise 5.1 Consider the verbs hates and washes, and the noun phrases Ann, Lucy, a man and every woman. Check for each sentence of the form NP (V NP) using these verbs and noun phrases whether that sentence is true or false in this model.

5.2 Databases as Models for Predicate Logic

In this section we will show how a database can be used as a model for predicate logic. We start out with the database given in Figure 5.1 (a full version of this is available on the book webpage).

module CINEMA
where

import List
import DB

The database in the DB module is called db, with type DB, where DB is a synonym for the type [WordList], where Wordlist is again a synonym for the type [String]. The reserved keyword type is used to declare these type synonyms. Notice the difference between defining a type synonym with type and declaring a new data type with data.

The database can be used to define the following lists of database objects, with list comprehension. Here db :: DB is the database list.

characters = nub [ x | ["play",_,_,x] <- db ]
movies = [ x | ["release",x,_] <- db ]
actors = nub [ x | ["play",x,_,_] <- db ]
directors = nub [ x | ["direct",x,_] <- db ]
dates = nub [ x | ["release",_,x] <- db ]
universe = nub (characters++actors++directors++movies++dates)
5.2. DATABASES AS MODELS FOR PREDICATE LOGIC

module DB
where

type WordList = [String]
type DB = [WordList]

db :: DB
db = [
  ["release", "Blade Runner", "1982"],
  ["release", "Alien", "1979"],
  ["release", "Titanic", "1997"],
  ["release", "Good Will Hunting", "1997"],
  ["release", "Pulp Fiction", "1994"],
  ["release", "Reservoir Dogs", "1992"],
  ["release", "Romeo and Juliet", "1996"],
  {- ... -}
  ["direct", "Brian De Palma", "The Untouchables"],
  ["direct", "James Cameron", "Titanic"],
  ["direct", "James Cameron", "Aliens"],
  ["direct", "Ridley Scott", "Alien"],
  ["direct", "Ridley Scott", "Blade Runner"],
  ["direct", "Ridley Scott", "Thelma and Louise"],
  ["direct", "Gus Van Sant", "Good Will Hunting"],
  ["direct", "Quentin Tarantino", "Pulp Fiction"],
  {- ... -}
  ["play", "Leonardo DiCaprio", "Romeo and Juliet", "Romeo"],
  ["play", "Leonardo DiCaprio", "Titanic", "Jack Dawson"],
  ["play", "Robin Williams", "Good Will Hunting", "Sean McGuire"],
  ["play", "John Travolta", "Pulp Fiction", "Vincent Vega"],
  ["play", "Harvey Keitel", "Reservoir Dogs", "Mr White"],
  ["play", "Harvey Keitel", "Pulp Fiction", "Winston Wolf"],
  ["play", "Uma Thurman", "Pulp Fiction", "Mia"],
  ["play", "Quentin Tarantino", "Pulp Fiction", "Jimmie"],
  ["play", "Quentin Tarantino", "Reservoir Dogs", "Mr Brown"],
  ["play", "Sigourney Weaver", "Alien", "Ellen Ripley"],
  {- ... -}
]

Figure 5.1: A Database Module.
Next, define lists of tuples, again by list comprehension:

\[
\text{direct} = [ (x,y) | ["direct",x,y] \leftarrow \text{db} ] \\
\text{act} = [ (x,y) | ["play",x,y,\_] \leftarrow \text{db} ] \\
\text{play} = [ (x,y,z) | ["play",x,y,z] \leftarrow \text{db} ] \\
\text{release} = [ (x,y) | ["release",x,y] \leftarrow \text{db} ]
\]

Finally, define one-placed, two-placed and three-placed predicates by means of lambda abstraction.

\[
\begin{align*}
\text{charP} &= \lambda x \rightarrow \text{elem } x \text{ characters} \\
\text{actorP} &= \lambda x \rightarrow \text{elem } x \text{ actors} \\
\text{movieP} &= \lambda x \rightarrow \text{elem } x \text{ movies} \\
\text{directorP} &= \lambda x \rightarrow \text{elem } x \text{ directors} \\
\text{dateP} &= \lambda x \rightarrow \text{elem } x \text{ dates} \\
\text{actP} &= \lambda (x,y) \rightarrow \text{elem } (x,y) \text{ act} \\
\text{releaseP} &= \lambda (x,y) \rightarrow \text{elem } (x,y) \text{ release} \\
\text{directP} &= \lambda (x,y) \rightarrow \text{elem } (x,y) \text{ direct} \\
\text{playP} &= \lambda (x,y,z) \rightarrow \text{elem } (x,y,z) \text{ play}
\end{align*}
\]

Note that the definition we have given of relations is equivalent to the definition with \texttt{flip elem}, for

\[
\lambda (x,y) \rightarrow \text{elem } (x,y) \text{ list}
\]

is equivalent to

\[
\lambda (x,y) \rightarrow \text{flip elem list } (x,y)
\]

which is in turn equivalent to

\texttt{flip elem list}

provided that the list is of type \([(a,a)]\), i.e., that it is a list consisting of pairs (use eta reduction).

### 5.3 Predicate Logic in Haskell

In this section we will give a formal definition of the language of predicate logic in Haskell. Once we have data types for terms and formulas, we can study the problem of specifying an interpretation function for a predicate logical language with respect to a model.
 module EPLIH where

 import List
 import Domain
 import Model

 We start out with a datatype for variables. It is convenient to give a variable both a name and an index. The indices will be needed for variable renaming when we turn to theorem proving (Chapter 7).

 type Name = String
 type Index = [Int]

 data Var = Var Name Index deriving (Eq,Ord)

 Variables are the type of object that can be shown on the screen. We therefore put them in the Show class, and define a function for showing them:

 instance Show Var where
    show (Var name []) = name
    show (Var name [i]) = name ++ show i
    show (Var name is) = name ++ showInts is
       where showInts [] = ""
           showInts [i] = show i
           showInts (i:is) = show i ++ "_" ++ showInts is

 Some example variables:

    x, y, z :: Var
    x = Var "x" []
    y = Var "y" []
    z = Var "z" []

 In the definition of formulas it is convenient to represent conjunction and disjunction as operations on lists of formulas. We use prefix notation for implications and equivalences. For starters, we will assume that all terms are variables. This will be modified below.
data Frm = Atom0 String [Var]  
  | Eq0 Var Var  
  | Neg0 Frm  
  | Impl0 Frm Frm  
  | Equi0 Frm Frm  
  | Conj0 [Frm]  
  | Disj0 [Frm]  
  | Forall0 Var Frm  
  | Exists0 Var Frm  

 deriving (Eq,Ord)

A function for showing formulas:

instance Show Frm where  
  show (Atom0 str []) = str  
  show (Atom0 str vs) = str ++ concat [ show vs ]  
  show (Eq0 t1 t2) = show t1 ++ "==" ++ show t2  
  show (Neg0 form) = '~': (show form)  
  show (Impl0 f1 f2) = "(" ++ show f1 ++ "==>" ++ show f2 ++ ")"  
  show (Equi0 f1 f2) = "(" ++ show f1 ++ "<=>" ++ show f2 ++ ")"  
  show (Conj0 []) = "true"  
  show (Conj0 fs) = "conj" ++ concat [ show fs ]  
  show (Disj0 []) = "false"  
  show (Disj0 fs) = "disj" ++ concat [ show fs ]  
  show (Forall0 v f) = "A" ++ show v ++ (" " : show f)  
  show (Exists0 v f) = "E" ++ show v ++ (" " : show f)

Some example formulas:

form0 = Atom0 "R" [x,y]  
form1 = Forall0 x (Atom0 "R" [x,x])  
form2 = Forall0 x  
  (Forall0 y  
    (Impl0 (Atom0 "R" [x,y]) (Atom0 "R" [y,x])))

These get displayed as follows:

EPLIH> form0
The first formula expresses that the $R$ relation is reflexive, the second that it is symmetric.

## 5.4 Evaluating Formulas in Models

We now turn to the problem of the evaluation of formulas of predicate logic with respect to a model. Two ingredients are interpretations and variable maps. An interpretation is a function from relation names to appropriate relations in the model. One possibility would be to distinguish according to the arity of the relation. Instead, we will let an interpretation be a function of the type $\text{String} \rightarrow [\text{a}] \rightarrow \text{Bool}$, or more specifically $\text{String} \rightarrow [\text{Entity}] \rightarrow \text{Bool}$.

To be able to use this type we need to be able to convert predicates of various arities into a single type. Here are the appropriate conversion functions.

```haskell
convert0 :: Bool -> [a] -> Bool
convert0 b [] = b

convert1 :: (a -> Bool) -> [a] -> Bool
convert1 p [x] = p x

convert2 :: ((a, a) -> Bool) -> [a] -> Bool
convert2 p [x,y] = p (x,y)

convert3 :: ((a,a,a) -> Bool) -> [a] -> Bool
convert3 p [x,y,z] = p (x,y,z)
```

This allows the following:

Model> convert2 hate [A,B]
True
Model> convert2 hate [B,B]
False

Here is an example interpretation function for some one-placed and two-placed relation symbols:
int0 :: String -> [Entity] -> Bool
int0 "P" = convert1 laugh
int0 "Q" = convert1 smile
int0 "R" = convert2 love
int0 "S" = convert2 hate

Now, the evaluation of \form in the model specified in Section 5.1, with respect to interpretation function \int, should yield False, for in the example model it is not the case that the love relation is reflexive.

To bring this about, we have to implement the evaluation function given in Section 2.6. This uses variable assignments \s. In the implementation, these have type \Var -> \Entity, or more generally \Var -> a. The crucial notion is \s(x\mid d), for the assignment that is like \s, but with \(x\) interpreted as \(d\).

change :: (Var -> a) -> Var -> a -> Var -> a
change s x d = \ v -> if x == v then d else s v

Now \change s x d is the implementation of \(s(x\mid d)\).

As an example, here is a definition of the assignment that maps every variable to object A.

ass0 :: Var -> Entity
ass0 = \ v -> A

The function that is like \ass0, except for the fact that \(y\) gets mapped to B, is given by:

ass1 :: Var -> Entity
ass1 = change ass0 y B

The evaluation function takes as its first argument an interpretation function, as its second argument a variable assignment, as its third argument a formula, and it yields a truth value. It is defined by recursion on the structure of the formula.

Before we give the implementation, let us note that there is a crucial difference between the definition of an evaluation function for predicate logic and an implementation of such a definition. Implementations allow us to check the value of a formula in a model. But it is unreasonable
to expect to be able to check the evaluation of arbitrary formulas in arbitrary models. Take
the famous Goldbach conjecture as an example. This conjecture — from a letter that Christian
Goldbach wrote to Leonhard Euler in 1742 — says that every even natural number can be
written as the sum of two primes. It can easily be stated in predicate logic as follows:

$$\forall x \exists y \exists z (2 \times x = y + z \wedge Py \wedge Pz).$$

This uses function symbols $\ast$ and $+$ for multiplication and addition, and a predicate symbol $P$
for being a prime number. The predicate $P$ can be defined in predicate logic using a binary
relation symbol $<$ for smaller than and a binary relation symbol $|$ for divisibility:

$$\forall x \exists y \exists z (2 \times x = y + z \wedge \neg \exists d (1 < d \wedge d < y \wedge d | y) \wedge \neg \exists d (1 < d \wedge d < z \wedge d | z)).$$

**Exercise 5.2** Give an equivalent formula that just uses $0, s, \ast, +, <$, i.e., show how $d | y$
can be defined in predicate logic with $0, s, \ast, +, <$ ($s$ for the successor function).

Although the interpretations of $<$ and $|$ are easy to check for given natural numbers, it is
unreasonable to expect an implementation of an evaluation function for predicate logic to check
whether this formula is true. The Goldbach conjecture is one of the many open problems of
the theory of natural numbers. At the moment of writing, nobody knows whether it is true
or not, because nobody has been able to provide a proof, and nobody has been able to find a
counterexample (the conjecture has been verified for numbers up to $4 \times 10^{14}$).

We will make the following assumptions about the domain of evaluation:

- In the first place, we will assume that tests for equality are possible on the domain. In
Haskell terminology: the type $a$ of our domain should in the class $Eq$.

- We will assume that the domain can be enumerated, and we will use the enumerated
domain as an argument to the evaluation function.

Note that our example domain of entities from module `Domain` satisfies these requirements.
However, the domain of non-negative integers satisfies the requirements as well. Indeed, the
requirements are not enough to guarantee termination of the evaluation function for all possible
arguments.

The above assumptions are reflected in the type of the evaluation function `eval`:

```haskell
eval :: Eq a =>
    [a] -> (String -> [a] -> Bool) -> (Var -> a) -> Frm -> Bool
```

The evaluation function is defined for all types $a$ that belong to the class $Eq$. It takes an
enumerated universe (type `[a]`) as its first argument, an interpretation function for relation
symbols (type `String -> [a] -> Bool`) as its second argument, a variable assignment (type
Var -> a) as its third argument, and a formula as its fourth argument. The definition is by structural recursion on the formula:

```haskell
  eval univ i s (Atom0 str vs) = i str (map s vs)
  eval univ i s (Eq0 v1 v2) = (s v1) == (s v2)
  eval univ i s (Neg0 f) = not (eval univ i s f)
  eval univ i s (Impl0 f1 f2) =
      not ((eval univ i s f1) && not (eval univ i s f2))
  eval univ i s (Equi0 f1 f2) =
      (eval univ i s f1) == (eval univ i s f2)
  eval univ i s (Conj0 fs) = and (map (eval univ i s) fs)
  eval univ i s (Disj0 fs) = or (map (eval univ i s) fs)
  eval univ i s (Forall0 v f) =
      and [ eval univ i (change s v d) f | d <- univ ]
  eval univ i s (Exists0 v f) =
      or [ eval univ i (change s v d) f | d <- univ ]
```

The assumption that the type a of the domain of evaluation is in Eq is needed in the evaluation clause for equalities, the assumption that the domain of the model is enumerable by a list is needed in the evaluation clauses for the quantifiers, where objects are taken one by one from the domain of the model.

We can check the claim that form1 makes about the model from Section 5.1, when R gets interpreted as the love relation, as follows:

```haskell
EPLIH> eval entities int0 ass0 form1
False
```

For checking that form0 holds for the interpretation of "R" as < on the non-negative integers, for an assignment that maps x to 1 and y to 2, we need an interpretation in the type Int:

```haskell
  rconvert :: (a -> a -> Bool) -> [a] -> Bool
  rconvert r [x,y] = r x y

  int1 :: String -> [Int] -> Bool
  int1 "R" = rconvert (<)
```

Here is an appropriate assignment:
In Haskell, if \( n \) is an element of an enumerable domain, then \([n..]\) will give the list of elements of the domain, from \( n \) onward. This may be an infinite list.

Suppose we want to evaluate formula \( \text{form4} \) in the infinite domain of non-negative integers. This domain can be listed in Haskell as \([0..]\). If you key in \([0..]\) at the Haskell prompt and you hit the return key, you will find that an infinite list of integers covers the screen. Try this out, and use Control-C to interrupt the process.

Because Haskell is a lazy functional programming language, we can use infinite lists as arguments to functions. The laziness consists in the fact that arguments are only evaluated when needed, and as far as needed. The elements of \([0..]\) will be accessed one at a time, and only at a point where they are actually needed.

```
EPLIH> eval [0..] int1 ass2 form0
True
```

Checking the claim that \( \text{form1} \) makes about the non-negative integers, with \( R \) interpreted as the less than relation, is not a feasible matter, for the evaluation procedure will keep on trying different numbers ... 

**Exercise 5.3** Consider the following formula:

```
form3 = Exists0 x
     (Exists0 y
      (Conj0 [Atom0 "R" [x,y], Atom0 "S" [x,y]]))
```

*Define an interpretation for this in the Cinema model ensuring that the formula expresses that some person is both actor and director in the same movie picture. Is the formula true in the Cinema model under this interpretation?*

### 5.5 Adding Functional Terms

In the above datatype for formulas it is assumed that all terms are variables. The following datatype for terms caters for variables plus functional terms. Functional terms with an empty variable list are the individual constants of the language.
data Term = Vari Var | Struct String [Term] deriving (Eq,Ord)

Some examples of variable terms:

\[
\begin{align*}
\text{tx, ty, tz :: Term} \\
\text{tx} &= \text{Vari x} \\
\text{ty} &= \text{Vari y} \\
\text{tz} &= \text{Vari z}
\end{align*}
\]

Some examples of constant terms:

\[
\begin{align*}
\text{a} &= \text{Struct "a" []} \\
\text{b} &= \text{Struct "b" []} \\
\text{c} &= \text{Struct "c" []} \\
\text{zero} &= \text{Struct "zero" []}
\end{align*}
\]

Some examples of useful functions:

\[
\begin{align*}
\text{s} &= \text{Struct "s"} \\
\text{t} &= \text{Struct "t"} \\
\text{u} &= \text{Struct "u"} \\
\text{one} &= \text{s[zero]} \\
\text{two} &= \text{s[one]} \\
\text{three} &= \text{s[two]} \\
\text{four} &= \text{s[three]} \\
\text{five} &= \text{s[four]}
\end{align*}
\]

Here is the declaration of Term as an instance of the Show class:

\[
\begin{align*}
\text{instance Show Term where} \\
\text{show (Vari v)} &= \text{show v} \\
\text{show (Struct str [])} &= \text{str} \\
\text{show (Struct str ts)} &= \text{str ++ concat [ show ts ]}
\end{align*}
\]
The function `isVar` checks whether a term is a variable:

```haskell
isVar :: Term -> Bool
isVar (Vari _) = True
isVar _ = False
```

The functions `varsInTerm` and `varsInTerms` give the variables that occur in a term or a term list. Variable lists should not contain duplicates; the function `nub` cleans up the variable lists.

```haskell
varsInTerm :: Term -> [Var]
varsInTerm (Vari v) = [v]
varsInTerm (Struct str ts) = varsInTerms ts

varsInTerms :: [Term] -> [Var]
varsInTerms = nub . concat . map varsInTerm
```

Once we have variables and terms, it is straightforward to declare a data type for the full language of predicate logic.

```haskell
data Form = Atom String [Term]
          | Eq Term Term
          | Neg Form
          | Impl Form Form
          | Equi Form Form
          | Conj [Form]
          | Disj [Form]
          | Forall Var Form
          | Exists Var Form
          deriving (Eq,Ord)
```

Note that an ordering on formulas is derived from the ordering on terms.

A `show` function for this new kind of formulas:

```haskell
```
instance Show Form where
    show (Atom id []) = id
    show (Atom id ts) = id ++ concat [ show ts ]
    show (Eq t1 t2) = show t1 ++ "==" ++ show t2
    show (Neg form) = '~': (show form)
    show (Impl f1 f2) = "(" ++ show f1 ++ "==>" ++ show f2 ++ ")"
    show (Equi f1 f2) = "(" ++ show f1 ++ "<=>" ++ show f2 ++ ")"
    show (Conj []) = "true"
    show (Conj fs) = "conj" ++ concat [ show fs ]
    show (Disj []) = "false"
    show (Disj fs) = "disj" ++ concat [ show fs ]
    show (Forall id f) = "A" ++ show id ++ (' ' : show f)
    show (Exists id f) = "E" ++ show id ++ (' ' : show f)

Exercise 5.4 Give a implementation of a function \( \text{varsInForm} :: \text{Form} \to [\text{Var}] \) that gives the list of variables occurring in a formula.

Exercise 5.5 Give a implementation of a function \( \text{freeVarsInForm} :: \text{Form} \to [\text{Var}] \) that gives the list of variables with free occurrences in a formula.

Exercise 5.6 Give a implementation of a function \( \text{openForm} :: \text{Form} \to \text{Bool} \) that checks whether a formula is open (see Section 2.6).

As in the case of the interpretation of relations, it is useful to have a general type for the interpretation of functions. We will use \( \text{String} \to [\text{a}] \to \text{a} \). For the specific case where the domain consists of objects of type \text{Entity}, we get:

\[ \text{String} \to [\text{Entity}] \to \text{Entity} \]

Again, we need appropriate conversion functions:

\[
\begin{align*}
\text{fconvert0} & :: \text{a} \to [\text{a}] \to \text{a} \\
\text{fconvert0} \ d \ [] & = d \\
\text{fconvert1} & :: (\text{a} \to \text{a}) \to [\text{a}] \to \text{a} \\
\text{fconvert1} \ f \ [x] & = f \ x \\
\text{fconvert2} & :: (\text{a} \to \text{a} \to \text{a}) \to [\text{a}] \to \text{a} \\
\text{fconvert2} \ f \ [x,y] & = f \ x \ y \\
\text{fconvert3} & :: (\text{a} \to \text{a} \to \text{a} \to \text{a}) \to [\text{a}] \to \text{a} \\
\text{fconvert3} \ f \ [x,y,z] & = f \ x \ y \ z
\end{align*}
\]
An example interpretation for functions, for the case where our domain of entities consists of integers.

\[
\begin{align*}
\text{fint1} & : \text{String} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
fint1 \text{ "zero" } [\text{]} & = 0 \\
fint1 \text{ "s" } [i] & = \text{succ } i \\
fint1 \text{ "plus" } [i, j] & = i + j \\
fint1 \text{ "times" } [i, j] & = i \times j
\end{align*}
\]

We have provided interpretations of the constant \texttt{zero}, of the one-placed function \texttt{s}, and of the two-placed functions \texttt{plus} and \texttt{times}. In Haskell, the type \texttt{Int} is bounded by \texttt{primMinInt} (for the smallest integer that gets represented) and \texttt{primMaxInt} (for the greatest integer that gets represented).

Before we can turn to evaluation of formulas, we have to lift valuation functions from type \texttt{Var} \rightarrow \texttt{a} to type \texttt{Term} \rightarrow \texttt{a}, given appropriate interpretations for function symbols. This is done with recursion on the structure of terms, as follows:

\[
\begin{align*}
liftVal & : (\text{Var} \rightarrow \texttt{a}) \rightarrow (\text{String} \rightarrow [\texttt{a}] \rightarrow \texttt{a}) \rightarrow (\texttt{Term} \rightarrow \texttt{a}) \\
liftVal s f i (\text{Vari } v) & = s v \\
liftVal s f i (\text{Struct } \text{str } ts) & = f i \text{ str } (\text{map (liftVal } s f i) \text{ ts})
\end{align*}
\]

Finally, we are ready to extend the semantic evaluation function from the previous section to the present case. The evaluation function takes one extra argument, for the function symbol interpretation.
To interpret formulas in the domain \( \text{Int} \), here is an example formula:

\[
\text{form4} = \exists x (\text{Atom } "R" [\text{zero}, \text{tx}])
\]

Evaluation with respect to relation symbol interpretation \( \text{int1} \), function symbol interpretation \( \text{fint1} \), and variable assignment \( \lambda (\text{Var } _\_ ) \to 0 \), in domain \([0..]\):

EPLIH> evl [0..] int1 fint1 (\ (Var _ _) \to 0) form4
True

The following formula should yield \text{False} under interpretation \( \text{int1} \), but in fact the evaluation procedure will not terminate.

\[
\text{form5} = \exists x (\text{Atom } "R" [\text{tx}, \text{zero}])
\]

To check \( \text{form5} \), the evaluation procedure will keep on trying candidates from the infinite list \([0..]\).

Even in cases where there are solutions, it may take a long time before the right candidate or the right tuple of candidates has been founds. This suggests that it makes sense to perform model checking in an interactive fashion, so that the user can state that there are no solutions
for certain queries, and influence the choice of suitable referents for free variables for certain
other queries.

Here is an example of a formula with nested quantifiers:

\[
\text{form6} = \forall x (\forall y (\text{impl} (\text{atom } "R" [tx,ty])
(\exists z (\text{conj} [\text{atom } "R" [tx,tz],
\text{atom } "R" [tz,ty]])))))
\]

This gets displayed as follows:

EPLIH> form6
 Ax Ay (R[x,y]==>Ez conj[R[x,z],R[z,y]])

**Exercise 5.7** What does \text{form6} express, assuming that \text{R} is interpreted as \text{<}? Is this formula true on the natural numbers? Is it true on the rational numbers (the numbers that can be written as \( \frac{n}{m} \) with \( n \) and \( m \) integers and \( m \neq 0 \))?

Again, we cannot expect to get an answer to the query

\text{evl [0..] int1 fint1 (\ (Var _ _) -> 1) form6}

in feasible time. And we cannot even check \text{form7} for the assignment (\ (Var _ [i]) -> i) to get a refutation:

\[
\text{form7} = \text{impl} (\text{atom } "R" [tx,ty])
(\exists z (\text{conj} [\text{atom } "R" [tx,tz],
\text{atom } "R" [tz,ty]]))
\]

The evaluation procedure for

\text{evl [0..] int1 fint1 (\ (Var _ [i]) -> i) form7}

will keep trying different values for \text{z}, and never find one that yields true.

Now let us try this on the model from Section 5.1, with \text{R} interpreted as the \text{hate} relation. Here is an appropriate interpretation function.
int3 :: String -> [Entity] -> Bool
int3 "R" = convert2 hate

Since \texttt{form7} does not contain function symbols, it does not matter which function symbol interpretation we use.

fint2 :: String -> [Entity] -> Entity
fint2 "zero" [] = A

EPLIH> evl entities int3 fint2 (\ (Var _ _) -> A) form7
True

5.6 Further Reading

In computer science, evaluation of formulas of suitable logical languages (predicate logic, but also temporal logics) in appropriate models is called model checking. This has developed into a discipline in its own right. See [HR00] for an introduction.
Chapter 6

The Composition of Meaning in NL

Summary

In this chapter, the meaning of natural language is analysed along the lines proposed by Gottlob Frege. In building meaning representations, we assume that the meaning of a complex expression derives from the meanings of its components. Typed logic is a convenient tool to make this process of composition explicit. Typed logic allows for the building of semantic representations for formal languages and fragments of natural language in a compositional way. The note ends with the discussion of an example fragment, implemented in the functional programming language Haskell.

6.1 Rules of the Game

An invitation to translate English sentences from an example fragment of natural language given by some set of grammar rules into predicate logic (or predicate logic with generalized quantifiers) presupposes two things: (i) that you grasp the meanings of the formulas of the representation language, and (ii) that you understand the meanings of the English sentences. Knowledge of the first kind can be made fully explicit; stating it in a fully explicit fashion is the job of the semantic truth definitions for the representation languages. In fact, translation exercises of this kind in logic textbooks are meant to expand the reader’s awareness of the expressive power of the logical representation language by inviting him/her to express an (intuitively) well-understood message in the new medium. Because of this presupposed understanding of the original message, such translations cannot count as explications of the concept of meaning for natural language.

Is it also possible to make knowledge of the meaning of a fragment of natural language fully explicit? More precisely: under what conditions does a translation procedure from natural language into some kind of logical representation language count as an explication of the concept of meaning for natural language? Obviously, the procedure should not presuppose knowledge of the meaning of complete natural language sentences, but rather should specify how sentence meanings are derived from the meanings of smaller building blocks. Thus, the meanings of
complex expressions should be derivable in a systematic fashion from the meanings of the smallest building blocks occurring in those expressions. The meaning of these smallest building blocks is taken as given. (It has been argued that the real mystery of semantics lies in the way human beings grasp the meanings of single words. Fortunately, this mystery is not explained away by logical analysis, for logic is a manifestation of the Power of Pure Intelligence rather than an explication of it.)

To count as an explication of the concept of meaning, a translation procedure from natural language should derive sentence meanings from the meanings of smaller building blocks. The meanings of complex expressions should be derivable in a systematic fashion from the meanings of their components. The meanings of the smallest building blocks can then be taken as given. Formal semantics has little or nothing to say about the interpretation of semantic atoms. It has rather a lot to say, however, about the process of composing complex meanings in a systematic way out of the meanings of components. The intuition that this is always possible can be stated somewhat more precisely; it is called the Principle of Compositionality:

The meaning of an expression is a function of the meanings of its immediate syntactic components plus their syntactic mode of composition.

The principle of compositionality is implicit in Gottlob Frege’s writings on philosophy of language; it has been made fully explicit in Richard Montague’s approach to natural language semantics.

6.2 Misleading Form and Logical Form

From John walked it follows that someone walked, but from No-one walked it does not follow that someone walked. Therefore, logicians such as Frege, Russell, Tarski and Quine have maintained that the structure of these two sentences must differ, and that it is not enough to say that they are both compositions of a subject and a predicate.

The logicians who used first order predicate logic to analyse the logical structure of natural language were struck by the fact that the logical translations of natural language sentences with quantified expressions did not seem to follow the linguistic structure. In the logical translations, the quantified expressions seemed to have disappeared. The logical translation of (6.1) does not reveal a constituent corresponding to the quantified subject noun phrase.

6.1 Every unmarried man courted Mary.

6.2 \( \forall x((\text{man } x \land \neg \text{married } x) \rightarrow x \text{ courted mary}). \)

In the translation (6.2) the constituent every unmarried man has disappeared: it is contextually eliminated. Frege remarks that a quantified expression like every unmarried man does not give rise to a concept by itself (eine selbständige Vorstellung), but can only be interpreted in the context of the translation of the whole sentence. Applied to this particular example: the literal paraphrase of (6.2) is:
All objects in the domain of discourse have the property of either not being unmarried men or being objects who courted Mary.

In this restatement of sentence (6.1) the phrase *every unmarried man* does not occur any more.

The logical properties of sentences involving quantified expressions (and descriptions, analyzed in terms of quantifiers) suggested indeed that the way a simple noun phrase such as a proper name combines with a predicate is logically different from the way in which a quantified noun phrase or a definite description combines with a predicate. This led to the belief that the linguistic form of natural language expressions was misleading.

The application of the logical tools of abstraction and reduction allow us to see that this conclusion was unwarranted. Using translation of natural language in expressions of typed logic we see that natural language constituents correspond to typed expressions that combine with one another as functions and arguments. After full reduction of the results, quantified expressions and other constituents may have been contextually eliminated, but this elimination is a result of the reduction process, not of the supposed misleading form of the original natural language sentence. Thus we see that, while fully reduced logical translations of natural language sentences may be misleading in some sense, the fully unreduced original expressions are not.

As an example of the way in which the λ tools smoothe logical appearances, consider the logic of the combination of subjects and predicates. In the simplest cases (*John walked*) one could say that the predicate takes the subject as an argument, but this does not work for quantified subjects (*no-one walked*). All is well, however, when we say that the subject always takes the predicate as its argument, and make this work for simple subjects by logically raising their status from argument to function. Using λ, this is easy enough: we translate *John* not as the constant $j$, but as the expression \((\lambda P. (Pj))\). This expression denotes a function from properties to truth values, so it can take a predicate translation as its argument. The translation of *no-one* is of the same type: \((\lambda P. \neg \exists x. ((\text{person } x) \land (Px)))\). Before reduction, the translations of *John walked* and *no-one walked* look very similar. These similarities disappear only after both translations have been reduced to their simplest forms.

## 6.3 Meaning in Natural Language

At long last we are ready for an exercise in composition of meaning for natural language. We will illustrate the theory with a simple natural language grammar fragment. Every syntax rule has a semantic counterpart to specify how the meaning representation of the whole is built from the meaning representations of the components. $X$ is always used for the meaning of the whole,
and \( X_n \) refers to the meaning representation of the \( n \)-th daughter.

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow Mary \\
NP & \rightarrow Bill \\
NP & \rightarrow DET \ CN \\
NP & \rightarrow DET \ RCN \\
DET & \rightarrow every \\
DET & \rightarrow some \\
DET & \rightarrow no \\
CN & \rightarrow man \\
CN & \rightarrow woman \\
RCN & \rightarrow CN \ that \ VP \\
RCN & \rightarrow CN \ that \ NP \ TV \\
VP & \rightarrow laughed \\
VP & \rightarrow smiled \\
VP & \rightarrow TV \ NP \\
TV & \rightarrow loved \\
TV & \rightarrow respected \\
\end{align*}
\]

As an example, we consider the sentence \textit{Bill loved Mary}, which has the following syntactic structure in our fragment:

\[
[s[NP \ Bill][VP \ loved][NP \ Mary]]
\]

According to the rules above, this gets assigned the following meaning:

\[
((\lambda P.(Pb))(\lambda x.(\lambda P.(Pm))(\lambda y.(((\lambda x.(\lambda y((Lx)y)y))y)x)x)))
\]

Reducing this gives:

\[
\begin{align*}
& \xrightarrow{\beta} ((\lambda P.(Pb))(\lambda x.(\lambda P.(Pm))(\lambda y.(((\lambda z.(Lz)y))y)x)x)) \\
& \xrightarrow{\beta} ((\lambda P.(Pb))(\lambda x.(\lambda P.(Pm))((\lambda y.(L_y)x)))) \\
& \xrightarrow{\beta} ((\lambda P.(Pb))(\lambda x.(\lambda y.(L_y)x))) \\
& \xrightarrow{\beta} ((\lambda P.(Pb))(\lambda x.(\lambda y.(L_y)x))) \\
& \xrightarrow{\beta} ((\lambda x.(\lambda y.(L_y)x))b) \\
& \xrightarrow{\beta} ((Lm)b).
\end{align*}
\]

**Exercise 6.1** Give the compositional translation for ‘Bill loved some woman’, and reduce it to normal form.
6.4 Datastructures for Syntax

We will now demonstrate that it is straightforward to give an implementation of compositional semantics of natural language if we use a programming language that is itself based on type theory. First we define the data structures for the predicates, the variables, and the formulas, with data declarations for the various syntactic categories. The text in the square boxes below is the actual program code.

```haskell
module CM where

import MOTT
import EPLIH
import Domain
import Model
import List

data Sent = Sent NP VP
    deriving (Eq,Show)

data NP = Ann | Mary | Bill | Johnny | NP1 DET CN | NP2 DET RCN
    deriving (Eq,Show)

data DET = Every | Some | No | The | Most | Atleast Int
    deriving (Eq,Show)

data CN = Man | Woman | Boy | Person | Thing | House
    deriving (Eq,Show)

data RCN = CN1 CN VP | CN2 CN NP TV
    deriving (Eq,Show)

data VP = Laughed | Smiled | VP1 TV NP
    deriving (Eq,Show)

data TV = Loved | Respected | Hated | Owned
    deriving (Eq,Show)
```

The suffix `deriving (Eq,Show)` in the data type declarations is the Haskell way to ensure that equality is defined for these data types, and that they can be displayed on the screen.
6.5 Semantic Interpretation

Next, we define for every syntactic category an interpretation function of the appropriate type, using \( \text{Entity} \) for \( e \) and \( \text{Bool} \) for \( t \). The interpretation of sentences has type \( \text{Bool} \), so the interpretation function \( \text{intS} \) gets type \( \text{Sent} \rightarrow \text{Bool} \). Since there is only one rewrite rule for \( S \), the interpretation function \( \text{intS} \) consists of only one equation:

\[
\text{intSent} :: \text{Sent} \rightarrow \text{Bool} \\
\text{intSent} \ (\text{Sent} \ np \ vp) = (\text{intNP} \ np) \ (\text{intVP} \ vp)
\]

The interpretation function \( \text{intNP} \) consists of four equations, one for every rewrite rule for \( \text{NP} \) in the grammar fragment. The function has type \( \text{NP} \rightarrow (\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool} \), which should be read as \( \text{NP} \rightarrow ((\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool}) \). Here \( \text{Entity} \rightarrow \text{Bool} \) is the Haskell counterpart to \( e \rightarrow t \), which is the type of the VP interpretation that the NP combines with to form a sentence.

\[
\text{intNP} :: \text{NP} \rightarrow (\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool} \\
\text{intNP} \ Ann = \ p \rightarrow p \ ann \\
\text{intNP} \ Mary = \ p \rightarrow p \ mary \\
\text{intNP} \ Bill = \ p \rightarrow p \ bill \\
\text{intNP} \ Johnny = \ p \rightarrow p \ johnny \\
\text{intNP} \ (\text{NP1} \ det \ cn) = (\text{intDET} \ det) \ (\text{intCN} \ cn) \\
\text{intNP} \ (\text{NP2} \ det \ rcn) = (\text{intDET} \ det) \ (\text{intRCN} \ rcn)
\]

Note the close connection between \( \ p \rightarrow p \ mary \) and \( \lambda P. (Pm) \) that we get by employing the Haskell counterpart to \( \lambda \).

For the interpretation of verb phrases we invoke the information encoded in our first order model.

\[
\text{intVP} :: \text{VP} \rightarrow \text{Entity} \rightarrow \text{Bool} \\
\text{intVP} \ Laughed = \text{laugh} \\
\text{intVP} \ Smiled = \text{smile}
\]

The interpretation of complex VPs is a bit more involved. We have to find a way to make reference to the property of ‘standing into the TV relation to the subject of the sentence’. We do this in the same way as in the type logic specification of the semantic clause for \( /TV \ NP/_{VP} \).
6.5. SEMANTIC INTERPRETATION

\[
\text{intVP} (\text{VP1 tv np}) = \\text{subj} \rightarrow \text{intNP} \text{np} (\\text{obj} \rightarrow \text{intTV} \text{tv (subj, obj)})
\]

Note that \text{subj} refers to the sentence subject and \text{obj} to the sentence direct object.

The interpretation of transitive verbs discloses another bit of information about the world. Again, we invoke the information about the world encoded in our first order model.

\[
\text{intTV} : \text{TV} \rightarrow (\text{Entity, Entity}) \rightarrow \text{Bool}
\]

\[
\begin{align*}
\text{intTV Loved} &= \text{love} \\
\text{intTV Respected} &= \text{respect} \\
\text{intTV Hated} &= \text{hate} \\
\text{intTV Owned} &= \text{own}
\end{align*}
\]

The interpretation of CNs is similar to that of VPs.

\[
\text{intCN} : \text{CN} \rightarrow \text{Entity} \rightarrow \text{Bool}
\]

\[
\begin{align*}
\text{intCN Man} &= \text{man} \\
\text{intCN Boy} &= \text{boy} \\
\text{intCN Woman} &= \text{woman} \\
\text{intCN Person} &= \text{person} \\
\text{intCN Thing} &= \text{thing} \\
\text{intCN House} &= \text{house}
\end{align*}
\]

Next, we get the most involved part of the implementation: the definition of the determiner interpretations. First the type. The interpretation of a DET needs two properties (type \text{Entity} \rightarrow \text{Bool}): one for the CN and one for the VP, to yield the type of an S interpretation, i.e., \text{Bool}.

\[
\text{intDET} : \text{DET} \rightarrow (\text{Entity} \rightarrow \text{Bool}) \rightarrow (\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool}
\]

The interpretation of \text{Some} just checks whether the two properties corresponding to CN and VP have anything in common. This is what this check looks like in Haskell:

\[
\text{intDET Some p q} = \text{any q (filter p entities)}
\]
Here \( \text{filter } p \ \text{entities} \) gives the list of all members of \( \text{entities} \) that satisfy property \( p \); \( \text{minBound} \) and \( \text{maxBound} \) are ways to refer to the first and the last element of the domain of \( \text{entities} \), and therefore \( \text{entities} \) gives the domain of entities in the form of a list.

\( \text{any} \) is a function taking a property and a list that returns \( \text{True} \) if the sublist of elements satisfying the property is non-empty, \( \text{False} \) otherwise. Thus, \( \text{any } q \ \text{list} \) checks whether any element of the list satisfies property \( q \).

The interpretation of \( \text{Every} \) checks whether the CN property is included in the VP property:

\[
\text{intDET Every } p \ q = \text{all } q \ (\text{filter } p \ \text{entities})
\]

Here \( \text{filter } p \ \text{entities} \) gives the list of all members of \( \text{entities} \) that satisfy property \( p \), and \( \text{all } q \ \text{list} \) checks whether every member of \( \text{list} \) satisfies property \( q \).

The interpretation of \( \text{The} \) consists of two parts:

1. a check that the CN property is unique, i.e., that it is true of precisely one entity in the domain,
2. a check that the CN and the VP property have an element in common, in other words, the \( \text{Some} \) check on the two properties.

\[
\text{intDET The } p \ q = \text{singleton } \text{plist} \ && q \ (\text{head } \text{plist}) \ \text{where}
\text{plist} = \text{filter } p \ \text{entities}
\text{singleton } [x] = \text{True}
\text{singleton } _ = \text{False}
\]

The interpretation of \( \text{No} \) is just the negation of the interpretation of \( \text{Some} \):

\[
\text{intDET No } p \ q = \text{not } (\text{intDET Some } p \ q)
\]

The interpretation of \( \text{Most} \) compares the length of the list of entities satisfying the first argument (the restrictor argument) with the length of the list of entities satisfying the second argument (the body argument).
6.5. SEMANTIC INTERPRETATION

\[
\text{intDET Most } p \ q = \text{length } p\text{qlist } > \text{length } (\text{plist } \setminus \text{ qlist})
\]

where
\[
\begin{align*}
\text{plist} &= \text{filter } p \text{ entities} \\
\text{qlist} &= \text{filter } q \text{ entities} \\
\text{pqlist} &= \text{filter } q \text{ plist}
\end{align*}
\]

Exercise 6.2 Implement the interpretation function for (Atleast n).

The interpretation of relativised common nouns of the form That CN VP checks whether an entity has both the CN and the VP property:

\[
\text{intRCN} :: \text{RCN} \to \text{Entity} \to \text{Bool}
\]

\[\text{intRCN (CN1 cn vp) = \ e \to ((\text{intCN cn e) } \&\& (\text{intVP vp e)))}\]

The interpretation of relativised common nouns of the form That CN NP TV checks whether an entity has both the CN property as the property of being the object of NP TV.

\[\text{intRCN (CN2 cn np tv) = \ e \to ((\text{intCN cn e) } \&\& (\text{intNP np (\ subj \to (\text{intTV tv (subj,e))))))))}\]

Assume you have collected all the code from the book website, and you have loaded the module CM.hs. Here are some examples of queries for this code:

```
CM> \text{intSent (Sent} (\text{NP1 The Boy}) \text{Smiled})
True
CM> \text{intSent (Sent} (\text{NP1 The Boy}) \text{Laughed})
False
CM> \text{intSent (Sent} (\text{NP1 Some Man}) \text{Laughed})
False
CM> \text{intSent (Sent} (\text{NP1 No Man}) \text{Laughed})
True
CM> \text{intSent (Sent} (\text{NP1 Some Man}) (\text{VP1 Loved}) (\text{NP1 Some Woman}))
True
CM> \text{intSent (Sent} (\text{NP1 Some Man}) (\text{VP1 Loved Mary})) \text{Laughed})
True
```
It is a bit awkward that we have to provide the datastructures of syntax ourselves. The process of constructing syntax datastructures from strings of words is called parsing; this topic will be taken up in Chapter 8.

6.6 Translation into Logical Form

One of the insights of Montague grammar is that the presence of a level of logical form is superfluous for model theoretic interpretation. In [Mon73] and [Mon74b] a typed language of logical formulas is used as a stepping stone on the way to semantic specification, but in [Mon74a] the meaning of of a fragment of English is specified without a detour through logical form.

The set-up above is along the lines of [Mon74a]. We will now demonstrate how translation into logical form can be implemented. We model our logical form language after the language of predicate logic from Chapter 5. In particular, we use the data types for Var and Term that were employed there. Here is a data type for generalized quantifiers.

```haskell
data GQ = ALL | SOME | THE | NO | MOST | ATLEAST Int deriving (Show,Eq,Ord)
```

In fact, the only difference between logical forms and formulas of predicate logic is the presence of generalized quantifiers.

```haskell
data LF = Atom1 Name [Term]
         | Eq1 Term Term
         | Neg1 LF
         | Impl1 LF LF
         | Equi1 LF LF
         | Conj1 [LF]
         | Disj1 [LF]
         | Quant GQ Var LF LF deriving (Eq,Ord)
```

A show function for logical forms:
The process of translating to logical form is a very easy variation on the interpretation process for syntactic structures. Instead of the type \texttt{Bool}, for interpretation in a model, we take \texttt{LF}, for a logical form of type \texttt{t}, and instead of the type \texttt{Entity} for entities in the model, take \texttt{Var}, for variables that are supposed to get mapped to entities.

In the basic cases, we translate proper names into constant terms and lexical CNs, VPs, TVs into atomic formulas.

```haskell
instance Show LF where
    show (Atom1 id []) = id
    show (Atom1 id ts) = id ++ concat [ show ts ]
    show (Eq1 t1 t2) = show t1 ++ "==" ++ show t2
    show (Neg1 form) = '~': (show form)
    show (Impl1 f1 f2) = "(" ++ show f1 ++ "==>" ++ show f2 ++ ")"
    show (Equi1 f1 f2) = "(" ++ show f1 ++ "<=>" ++ show f2 ++ ")"
    show (Conj1 []) = "true"
    show (Conj1 fs) = "conj" ++ concat [ show fs ]
    show (Disj1 []) = "false"
    show (Disj1 fs) = "disj" ++ concat [ show fs ]
    show (Quant gq var f1 f2) =
        show gq ++ " " ++ show var ++
        " (" ++ show f1 ++ "," ++ show f2 ++ ")"

lfSent :: Sent -> LF
lfSent (Sent np vp) = (lfNP np) (lfVP vp)
```

```haskell
lfNP :: NP -> (Term -> LF) -> LF
lfNP Ann = \ p -> p (Struct "Ann" [])
lfNP Mary = \ p -> p (Struct "Mary" [])
lfNP Bill = \ p -> p (Struct "Bill" [])
lfNP Johnny = \ p -> p (Struct "Johnny" [])
lfNP (NP1 det cn) = (lfDET det) (lfCN cn)
lfNP (NP2 det rcn) = (lfDET det) (lfRCN rcn)
```
\[ l_{fVP} :: \text{VP} \rightarrow \text{Term} \rightarrow \text{LF} \]
\[ l_{fVP} \text{ Laughed} = \lambda t \rightarrow \text{Atom}_1 \text{ "laugh" } [t] \]
\[ l_{fVP} \text{ Smiled} = \lambda t \rightarrow \text{Atom}_1 \text{ "smile" } [t] \]

\[ l_{fVP} (\text{VP} \text{ tv} \text{ np}) = \lambda \text{ subj} \rightarrow l_{fNP} \text{ np} (\lambda \text{ obj} \rightarrow l_{fTV} \text{ tv} (\text{subj}, \text{obj})) \]

\[ l_{fTV} :: \text{TV} \rightarrow (\text{Term}, \text{Term}) \rightarrow \text{LF} \]
\[ l_{fTV} \text{ Loved} = \lambda (t1, t2) \rightarrow \text{Atom}_1 \text{ "love" } [t1, t2] \]
\[ l_{fTV} \text{ Respected} = \lambda (t1, t2) \rightarrow \text{Atom}_1 \text{ "respect" } [t1, t2] \]
\[ l_{fTV} \text{ Hated} = \lambda (t1, t2) \rightarrow \text{Atom}_1 \text{ "hate" } [t1, t2] \]
\[ l_{fTV} \text{ Owned} = \lambda (t1, t2) \rightarrow \text{Atom}_1 \text{ "own" } [t1, t2] \]

\[ l_{fCN} :: \text{CN} \rightarrow \text{Term} \rightarrow \text{LF} \]
\[ l_{fCN} \text{ Man} = \lambda t \rightarrow \text{Atom}_1 \text{ "man" } [t] \]
\[ l_{fCN} \text{ Boy} = \lambda t \rightarrow \text{Atom}_1 \text{ "boy" } [t] \]
\[ l_{fCN} \text{ Woman} = \lambda t \rightarrow \text{Atom}_1 \text{ "woman" } [t] \]
\[ l_{fCN} \text{ Person} = \lambda t \rightarrow \text{Atom}_1 \text{ "person" } [t] \]
\[ l_{fCN} \text{ Thing} = \lambda t \rightarrow \text{Atom}_1 \text{ "thing" } [t] \]
\[ l_{fCN} \text{ House} = \lambda t \rightarrow \text{Atom}_1 \text{ "house" } [t] \]

The type of the logical form translation of determiners indicates that the translation takes a
determiner phrase and two arguments for objects of type \( \text{Term} \rightarrow \text{LF} \) (logical forms with term
holes in them), and produces a logical form.

\[ l_{fDET} :: \text{DET} \rightarrow (\text{Term} \rightarrow \text{LF}) \rightarrow (\text{Term} \rightarrow \text{LF}) \rightarrow \text{LF} \]

The translation of determiners takes some care, for it involves the construction of a logical form
where a variable gets bound. To ensure proper binding, we have to make sure that the newly
introduced variable will not get accidentally bound by a quantifier already present in the logical
form. For this, we first list the variables in logical forms.
varsInLf :: LF -> [Var]
varsInLf (Atom1 _ ts) = varsInTerms ts
varsInLf (Eq1 t1 t2) = varsInTerms [t1, t2]
varsInLf (Neg1 form) = varsInLf form
varsInLf (Impl1 f1 f2) = varsInLfs [f1, f2]
varsInLf (Equi1 f1 f2) = varsInLfs [f1, f2]
varsInLf (Conj1 fs) = varsInLfs fs
varsInLf (Disj1 fs) = varsInLfs fs
varsInLf (Quant gq var f1 f2) = var : varsInLfs [f1, f2]

varsInLfs :: [LF] -> [Var]
varsInLfs = nub . concat . map varsInLf

We need the list of variable indices of a list of logical forms in order to compute a fresh variable
index. All variables will get introduced by means of this mechanism, so if we start out with
variables of the form \texttt{Var "x" [0]}, and only introduce new variables of the form \texttt{Var "x" [i]},
we can assume that all variables occurring in our logical form will have the shape \texttt{Var "x" [i]}
for some integer \texttt{i}.

freshvar :: [LF] -> Var
freshvar lfs = (Var "x" [i])
where
  i = (foldr max 0 (xindices (varsInLfs lfs))) + 1
  xindices vs = map (\ (Var "x" [j]) -> j) vs

The definition uses \texttt{foldr} for defining the maximum of a list of integers (for the \texttt{foldr} function,
see page 50).

Define the term based on the variable with name \texttt{"x"} and index 0:

tx0 :: Term
tx0 = Vari (Var"x" [0])

Use this term as a dummy to provisionally fill in the term slot in formulas with term holes in them.
lfDET Some \( p \ q \) = Quant SOME \( v \) (\( p \ (\text{Vari} \ v) \)) (\( q \ (\text{Vari} \ v) \)) where 
\( v = \text{freshvar} \ [p \ tx0,q \ tx0] \)

lfDET Every \( p \ q \) = Quant ALL \( v \) (\( p \ (\text{Vari} \ v) \)) (\( q \ (\text{Vari} \ v) \)) where 
\( v = \text{freshvar} \ [p \ tx0,q \ tx0] \)

lfDET No \( p \ q \) = Quant NO \( v \) (\( p \ (\text{Vari} \ v) \)) (\( q \ (\text{Vari} \ v) \)) where 
\( v = \text{freshvar} \ [p \ tx0,q \ tx0] \)

lfDET The \( p \ q \) = Quant THE \( v \) (\( p \ (\text{Vari} \ v) \)) (\( q \ (\text{Vari} \ v) \)) where 
\( v = \text{freshvar} \ [p \ tx0,q \ tx0] \)

lfDET Most \( p \ q \) = Quant MOST \( v \) (\( p \ (\text{Vari} \ v) \)) (\( q \ (\text{Vari} \ v) \)) where 
\( v = \text{freshvar} \ [p \ tx0,q \ tx0] \)

**Exercise 6.3** Implement the translation function for \((\text{Atleast} \ n)\).

Use \text{Conj1} to conjoin the logical form for a common noun and the logical form for a relative clause into a logical form for a complex common noun.

lfRCN :: RCN -> Term -> LF
lfRCN (CN1 cn vp) = \ t -> Conj1 [lfCN cn t, lfVP vp t]
lfRCN (CN2 cn np tv) =
\ t -> Conj1 [lfCN cn t, lfNP np (\ subj -> lfTV tv (subj,t))]

This gives us:

CM> lfSent (Sent (NP1 Some Man) (VP1 Loved (NP1 Some Woman)))
SOME x2 (man[x2],SOME x3 (woman[x3],love[x2,x3]))
CM> lfSent (Sent (NP2 No (CN1 Man (VP1 Loved Mary))) Laughed)
NO x1 (conj[man[x1],love[x1, Mary]],laugh[x1])

**Exercise 6.4** Implement an evaluation function for logical forms in appropriate models.

**Exercise 6.5** The definition of logical forms uses \text{Quant GQ Var LF LF} for generalized quantifiers. Here \text{Quant} is just a tag, \text{GQ} is the generalized quantifier, \text{Var} is a variable ranging over the domain of discourse, and \text{LF} and \text{LF} are logical forms that we may take to range over truth values. In more standard notation, a generalized quantifier expression looks like this: \(Qv\phi_1\phi_2\). We may take this as an abbreviation of \(Q(\lambda v.(\phi_1,\phi_2))\). This shows that the generalized quantifier operator \(Q\) has type \((e \rightarrow (t,t)) \rightarrow t\). In Chapter 4 generalized quantifiers were introduced as operators of type \((e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t\): a generalized quantifier takes two one-placed predicates and produces a truth value. Give conversion functions for mapping operators of type \((e \rightarrow (t,t)) \rightarrow t\) to operators of type \((e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t\) and vice versa. Next, give an implementation in Haskell.
(using a type variable \(a\) for \(e\) and the type \(\text{Bool}\) for \(t\)). Hint: you will need pairing for converting from \((a \to (\text{Bool, Bool})) \to \text{Bool}\) to \((a \to \text{Bool}) \to (a \to \text{Bool}) \to \text{Bool}\), and projection (by means of \text{fst} and \text{snd}) for converting from \((a \to \text{Bool}) \to (a \to \text{Bool}) \to \text{Bool}\) to \((a \to (\text{Bool, Bool})) \to \text{Bool}\).

6.7 Ambiguity in Natural Language

It is illuminating to examine the notion of ambiguity for fragments of natural language with a compositional semantics. If a natural language expression \(E\) is ambiguous, i.e. if \(E\) has several distinct meanings, then, under the assumption that these meanings are arrived at in a compositional way, there are three possible sources for the ambiguity (combinations are possible, of course):

1. The ambiguity is lexical: \(E\) contains a word with several distinct meanings. Example: a splendid ball.

2. The ambiguity is structural: \(E\) can be assigned several distinct syntactic structures. Example: old [men and women] versus [old men] and women, or: the boy saw the [girl with the binoculars] versus the boy saw [the girl] [with the binoculars].

3. The ambiguity is derivational: the syntactic structure that \(E\) exhibits can be derived in more than one way. Example: Every prince sang some ballad. is not structurally ambiguous, but in order to account for the \(\exists\) reading one might want to assume that one of the ways in which the structure can be derived is by combining some ballad with the incomplete expression every prince sang —.

Lexical ambiguities can be handled in a fragment like ours by means of multiplying lexical entries. The two meanings of ball give rise to different translations:

\[
\begin{align*}
\text{CN} & \rightarrow \text{ball} & X & \rightarrow (\lambda x.(B'x)) \\
\text{CN} & \rightarrow \text{ball} & X & \rightarrow (\lambda x.(B''x))
\end{align*}
\]

If we use these translations, then lexical ambiguity shows up in the fact that The ball was nice will receive two parses, each with its own translation. An alternative approach would be to allow ambiguities in the logical representation language, by allowing ambiguous translations such as the following:

\[
\begin{align*}
\text{CN} & \rightarrow \text{ball} & X & \rightarrow (\lambda x.(\{B'x, B''x\}))
\end{align*}
\]

To work this out, such ambiguous translations should of course be given a clear meaning, for otherwise the translation would fail to count as a specification of the semantics. See Van Eijck and Jaspars [EJ96] for a proposal.

A correct handling of structural ambiguities should fall out of the matching between syntax and semantics.
Exercise 6.6 Extend the fragment with preposition phrases, in such a way that the sentence Mary saw the boy with the binoculars is generated in two structurally different ways, while there is only one parse tree for Mary saw Bill with the binoculars.

The next exercise will get you acquainted with the semantic handling of structural ambiguities.

Exercise 6.7 Extend the semantics to match the extended fragment. Hint: the two readings for the target sentence that you should aim at are the following (or equivalent reformulations):

\[ \exists x ((Bx) \land \exists y ((Bi y) \land ((Hy)x)) \land ((Sx)m)). \]

\[ \exists x ((Bx) \land \exists y ((Bi y) \land ((Hy)m)) \land ((Sx)m)). \]

Here \( Bi \) is an \( e \rightarrow t \) type constant for the property of being a pair of binoculars, and \( H \) is an \((e,e \rightarrow t)\) type constant for the relation of having. You should use \( H \) in the translation of with.

Exercise 6.8 Derive the two readings of Mary saw some boy with some binoculars in the extended fragment, and reduce them both to normal form.

Derivational ambiguities, finally, are very much a logician’s ploy. In an essay on philosophical logic by Peter Thomas Geach they are introduced as follows:

[... ] when we pass from “Kate / is loved by / Tom” to “Some girl / is loved by / every boy”, it does make a big difference whether we first replace “Kate” by “some girl” (so as to get the predicable “Some girl is loved by —” into the proposition) and then replace “Tom” by “every boy”, or rather first replace “Tom” by “every boy” (so as to get the predicable “— is loved by every boy” into the proposition) and then replace “Kate” by “some girl”. Two propositions that are reached from the same starting point by the same set of logical procedures (e.g. substitutions) may nevertheless differ in import because these procedures are taken to occur in a different order. [Gea80, section 64]

This is exactly the mechanism that has been proposed by Richard Montague to account for operator scope ambiguities in natural language. Montague introduces a rule for quantifying in of noun phrases in incomplete syntactic structures. The wide scope \( \exists \) reading for the example *Every prince sang some ballad* is derived by using a rule \( Q_i \) to quantify in *some ballad* for syntactic index \( i \) in the structure *Every prince sang \( PRO_i \)*. In more complex cases, where more than one occurrence of \( PRO_i \) is present, the appropriate occurrence is replaced by the noun phrase, and the other occurrences are replaced by pronouns or reflexives with the right syntactic agreement features. See [Mon73] for details.

A rudimentary version of quantifying-in, for the present fragment, would look like this, First we extend the syntax with dummy pronouns \( PRO_i \) (\( i \) a natural number which serves as an index), with a matching semantics, where a correspondingly indexed variable of type \( e \) is used. Next we add an indexed rule \( Q_i \) which combines an NP with a sentence \( S \) containing precisely one occurrence of \( PRO_i \) (notation for this: \( S[PRO_i] \)), and replaces that occurrence by an occurrence of the NP (notation: \( S[PRO_i/NP] \)). The translation instruction uses the fact that the sentence
translation contains a free variable $x_i$ resulting from the presence of the dummy pronoun PRO$_i$. This variable is bound, and the NP translation is applied to the result.

\[
\begin{align*}
\text{NP}_i & \rightarrow \text{PRO}_i \\
\text{NP} + S[\text{PRO}_i] & \Rightarrow Q_i \Rightarrow S[\text{PRO}_i/\text{NP}] \\
X & \rightarrow (\lambda P.(P.x_i)) \\
X + Y & \Rightarrow (X.(\lambda x_i.Y))
\end{align*}
\]

**Exercise 6.9** Use the quantifying-in rule to get a reading of Every man loved some woman where some woman gets widest scope. Reduce the translation to normal form.

In Chapter 9 we will develop a more principled account of operator scope ambiguities.

### 6.8 Further Reading

Lewis [Lew70] gives a good introduction to the process of building meanings in a compositional way. Excellent introductions to Montague grammar are Dowty, Wall and Peters [DWP81] and Gamut [Gam91b]. A nifty implementation of a natural language interpreter with a Montague style semantics as a lazy functional program can be found in [FL89].

The Montagovian approach to scope ambiguities does not account for restrictions on possible scope readings (at least not without further ado). Such restrictions can be imposed by means of constraints regulated by lexical features of determiner words. Imposing the right constraints is not easy, as the scope behaviour of natural language expressions tends to be influenced by the wider syntactic context. In Alshawi [Als92] details are given on how scoping mechanisms can be incorporated in large scale natural language systems.
Chapter 7

Theorem Proving for Predicate Logic

Summary

In this chapter, we demonstrate theorem proving for predicate logic by means of the tableau method. We first introduce substitutions and operations on substitutions, and present an algorithm for finding most general substitutions that unify pairs of terms or pairs of formulas (the unification algorithm). Unification is the key to the method of detecting inconsistencies with semantic tableaux.

7.1 Substitution

We start with declaring a module, and loading the formula code from the previous chapter, plus two standard utility modules, List and IOExts. The module IOExts contains the trace function that is useful for debugging. Its use will be demonstrated in Section 7.5.

```haskell
module TPFPL where

import List
import IOExts
import EPLIH
```

A variable binding is a pair consisting of a variable and a term. A binding binds the variable to the term. A binding \((v, t)\) is often represented as \(v \mapsto t\). A binding is proper if it does not bind variable \(v\) to term \(v\) (the same variable, viewed as a term). A variable substitution is a finite list of proper bindings, satisfying the requirement that no variable \(v\) occurs as a lefthand member in more than one binding \(v \mapsto t\).

Here is a data type declaration for substitutions as lists of bindings.
type Subst = [(Var, Term)]

The identity substitution $\epsilon$ (the substitution that changes nothing) gets defined as:

epsilon :: Subst
epsilon = []

Since the bindings in a substitution are supposed to be proper, bindings of the form $v \mapsto v$ have to be suppressed from a substitution representation. The function cleanUp takes care of this.

cleanUp :: Subst -> Subst
cleanUp = filter (\ (x, x') -> x' /= (Vari x))

The domain of a substitution is the list of all lefthand sides of its bindings. The range of a substitution is the list of all righthand sides of its bindings.

dom :: Subst -> [Var]
dom = map fst
rng :: Subst -> [Term]
rng = map snd

Application of a substitution to a variable is, in fact, a conversion from the representation of a substitution as a list of binders to its representation as a map from variables to terms:

appVar :: Subst -> Var -> Term
appVar [] y = (Vari y)
appVar ((x, x'):xs) y | x == y = x'
| otherwise = appVar xs y

Substitutions give rise to mappings from terms to terms via the following recursion.
The application of a substitution to a formula is given by the following recursion (note the way in which the binding power of the quantifiers affects the substitution):

\[
\text{appF} :: \text{Subst} \rightarrow \text{Form} \rightarrow \text{Form} \\
\text{appF} \ b \ (\text{Atom} \ a \ ts) = \text{Atom} \ a \ (\text{appTs} \ b \ ts) \\
\text{appF} \ b \ (\text{Neg} \ \text{form}) = \text{Neg} \ (\text{appF} \ b \ \text{form}) \\
\text{appF} \ b \ (\text{Impl} \ \text{form1} \ \text{form2}) = \text{Impl} \ (\text{appF} \ b \ \text{form1}) \ (\text{appF} \ b \ \text{form2}) \\
\text{appF} \ b \ (\text{Equi} \ \text{form1} \ \text{form2}) = \text{Equi} \ (\text{appF} \ b \ \text{form1}) \ (\text{appF} \ b \ \text{form2}) \\
\text{appF} \ b \ (\text{Conj} \ \text{forms}) = \text{Conj} \ (\text{appFs} \ b \ \text{forms}) \\
\text{appF} \ b \ (\text{Disj} \ \text{forms}) = \text{Disj} \ (\text{appFs} \ b \ \text{forms}) \\
\text{appF} \ b \ (\text{Forall} \ v \ \text{form}) = \text{Forall} \ v \ (\text{appF} \ b' \ \text{form}) \\
\text{where} \ b' = \text{filter} \ (\lambda (x,x') \rightarrow x /= v) \ b \\
\text{appF} \ b \ (\text{Exists} \ v \ \text{form}) = \text{Exists} \ v \ (\text{appF} \ b' \ \text{form}) \\
\text{where} \ b' = \text{filter} \ (\lambda (x,x') \rightarrow x /= v) \ b
\]

\[
\text{appFs} :: \text{Subst} \rightarrow \text{[Form]} \rightarrow \text{[Form]} \\
\text{appFs} \ b = \text{map} \ (\text{appF} \ b)
\]

The composition of substitution \(\sigma\) with substitution \(\tau\) should result in the substitution that one gets by applying \(\sigma\) after \(\tau\). The following definition has the desired effect.

**Definition 7.1 (Composition of substitution representations)** Let

\[\theta = [v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]\]

and \(\sigma = [w_1 \mapsto r_1, \ldots, w_m \mapsto r_m]\)

be substitution representations. Then \(\theta \circ \sigma\) is the result of removing from the sequence

\[w_1 \mapsto \theta(r_1), \ldots, w_m \mapsto \theta(r_m), v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\]

the bindings \(w_i \mapsto \theta(r_i)\) for which \(\theta(r_i) = w_i\), and the bindings \(v_j \mapsto t_j\) for which \(v_j \in\{w_1, \ldots, w_m\}\).

**Exercise 7.2** Prove that this definition gives the correct result.
Applying the recipe for composition to \( \{x \mapsto y\} \cdot \{y \mapsto z\} \) gives \( \{y \mapsto z, x \mapsto y\} \), applying it to \( \{y \mapsto z\} \cdot \{x \mapsto y\} \) gives \( \{x \mapsto z, y \mapsto y\} \). This example illustrates the fact that order of application of substitution matters. Substitutions do not commute.

\[\text{compose } \mathbf{x} \mathbf{s} \text{ ys implements application of substitution } \mathbf{x} \mathbf{s} \text{ after substitution ys.}\]

\[
\text{compose :: Subst } \rightarrow \text{ Subst } \rightarrow \text{ Subst}
\]

\[
\text{compose } \mathbf{x} \mathbf{s} \text{ ys } =
\]

\[
(\text{cleanUp } [ (y, (\text{appT } \mathbf{x} y')) | (y, y') <- \text{ys } ])
\]

\[
++
\]

\[
(\text{filter } (\backslash (x, x') \rightarrow x \text{ `notElem` (dom ys)} ) \mathbf{x}s)
\]

We use the notion of composition to define a relation \( \sqsubseteq \) on the set \( M \) of all substitutions (for given sets \( V \) and \( T \)), as follows. \( \theta \sqsubseteq \sigma \) if there is a substitution \( \rho \) with \( \theta = \rho \cdot \sigma \). (\( \theta \sqsubseteq \sigma \) is sometimes pronounced as: ‘\( \theta \) is less general than \( \sigma \).’)

The relation \( \sqsubseteq \) is a so-called pre-order: reflexive and transitive. \( \sqsubseteq \) is reflexive because for all \( \theta \) we have that \( \theta = \epsilon \cdot \theta \). \( \sqsubseteq \) is transitive because if \( \theta = \rho \cdot \sigma \) and \( \sigma = \tau \cdot \gamma \) then \( \theta = \rho \cdot (\tau \cdot \gamma) = (\rho \cdot \tau) \cdot \gamma \), i.e., \( \theta \sqsubseteq \gamma \).

### 7.2 Unification

If we have two expressions \( A \) and \( B \), each containing variables, then we are interested in the following questions:

- Is there a substitution \( \theta \) that makes \( A \) and \( B \) equal?
- How do we find such a substitution in an efficient way?

We introduce some terminology for this. The substitution \( \theta \) unifies expressions \( A \) and \( B \) if \( \theta A = \theta B \). The substitution \( \theta \) unifies two sequences of expressions \( (A_1, \ldots, A_n) \) and \( (B_1, \ldots, B_n) \) if, for \( 1 \leq i \leq n \), \( \theta \) unifies \( A_i \) and \( B_i \). Note that unification of pairs of atomic formulas reduces to unification of sequences of terms, for two atoms that start with a different predicate symbol do not unify, and two atoms \( P[t_1, \ldots, t_n] \) and \( P[s_1, \ldots, s_n] \) unify if the sequences \( [t_1, \ldots, t_n] \) and \( [s_1, \ldots, s_n] \) unify.

What we are going to need to apply tableau reasoning (Section 7.4) to predicate logic is unification of pairs of atomic formulas.

For example, we want to find a substitution that unifies the pair

\[P[x, g[a, z]], P[g[y, z], x].\]

In this example case, such unifying substitutions exist. A possible solution is

\[\{x \mapsto g[a, z], y \mapsto a\}.\]
for applying this substitution gives \( P[g[a, z], g[a, z]] \). Another solution is
\[
\{x \mapsto g[a, b], y \mapsto a, z \mapsto b\}.
\]
In this case, the second solution is an instance of the first, for
\[
\{x \mapsto g[a, b], y \mapsto a, z \mapsto b\} \subseteq \{x \mapsto g[a, z], y \mapsto a\},
\]
because
\[
\{x \mapsto g[a, b], y \mapsto a, z \mapsto b\} = \{z \mapsto b\} \cdot \{x \mapsto g[a, z], y \mapsto a\}.
\]
So we see that solution \( \{x \mapsto g[a, z], y \mapsto a\} \) is more general than solution \( \{x \mapsto g[a, b], y \mapsto a, z \mapsto b\} \).

If a pair of atoms is unifiable, it is useful to try and identify a solution that is as general as possible, for the more general a solution is, the less unnecessary bindings it contains. These considerations motivate the following definition.

**Definition 7.3** If \( \theta \) is a unifier for a pair of expressions (a pair of sequences of expressions), then \( \theta \) is called an mgu (a most general unifier) if \( \sigma \subseteq \theta \) for every unifier \( \sigma \) for the pair of expressions (the pair of sequences of expressions).

In the above example, \( \{x \mapsto g[a, z], y \mapsto a\} \) is an mgu for the pair \( P[x, g[a, z]], P[g[y, z], x] \).

The **Unification Theorem** says that if a unifier for a pair of sequences of terms exists, then an mgu for that pair exists as well. Moreover, there is an algorithm that produces an mgu for any pair of sequences of terms in case these sequences are unifiable, and otherwise ends with failure.

We will describe the unification algorithm and prove that it does what it is supposed to do. This constitutes the proof of the theorem.

We give the algorithm in the form of a Haskell program.

**Unification of terms.** Three cases:

- Unification of two variables \( x \) and \( y \) gives the empty substitution if the variables are identical, and otherwise a substitution that binds one variable to the other.
- Unification of \( x \) to a non-variable term \( t \) fails if \( x \) occurs in \( t \), otherwise it yields the binding \( \{x \mapsto t\} \).
- Unification of \( f \, \overline{t} \) and \( g \, \overline{r} \) fails if the two variable names are different, otherwise it yields the return of the attempt to do term list unification on \( \overline{t} \) and \( \overline{r} \).

If unification succeeds, a unit list containing a representation of a most general unifying substitution is returned. Return of the empty list indicates unification failure.
unifyTs :: Term -> Term -> [Subst]
unifyTs (Vari x) (Vari y) = if x==y then [epsilon] else [[(x,Vari y)]]
unifyTs (Vari x) t2 = [(x,t2)] | x `notElem` varsInTerm t2 ]
unifyTs t1 (Vari y) = [(y,t1)] | y `notElem` varsInTerm t1 ]
unifyTs (Struct a ts) (Struct b rs) = [ u | a==b, u <- unifyTlists ts rs ]

Unification of term lists:

- Unification of two empty term lists gives the identity substitution.
- Unification of two term lists of different length fails.
- Unification of two term lists $t_1, \ldots, t_n$ and $r_1, \ldots, r_n$ is the result of trying to compute a substitution $\sigma = \sigma_n \circ \cdots \circ \sigma_1$, where
  - $\sigma_1$ is a most general unifier of $t_1$ and $r_1$,
  - $\sigma_2$ is a most general unifier of $\sigma_1(t_2)$ and $\sigma_1(r_2)$,
  - $\sigma_3$ is a most general unifier of $\sigma_2\sigma_1(t_3)$ and $\sigma_2\sigma_1(r_3)$,
  - and so on.

unifyTlists :: [Term] -> [Term] -> [Subst]
unifyTlists [] [] = [epsilon]
unifyTlists [] (r:rs) = []
unifyTlists (t:ts) [] = []
unifyTlists (t:ts) (r:rs) = [(compose sigma2 sigma1) | sigma1 <- unifyTs t r,
                                  sigma2 <- unifyTlists (appTs sigma1 ts)
                                  (appTs sigma1 rs) ]

Our task is to show that these functions do what they are supposed to do: produce a unit list containing an mgu if such an mgu exists, produce the empty list in case unification fails.

The proof consists of a Lemma and two Theorems. The Lemma is needed in Theorem 7.5. The Lemma establishes a simple property of mgu’s. Theorem 7.6 establishes the result.

Lemma 7.4 If $\sigma_1$ is an mgu of $t_1$ and $s_1$, and $\sigma_2$ is an mgu of
then compose \( \sigma_2 \sigma_1 \) is an mgu of \( [t_1..t_n] \) and \( [s_1..s_n] \).

**Proof.** Let \( \theta \) be a unifier of \( [t_1..t_n] \) and \( [s_1..s_n] \). Given this assumption, we have to show that \( \text{compose} \ \sigma_2 \ \sigma_1 \) is more general than \( \theta \).

By assumption about \( \theta \) we have that \( \text{appT} \ \theta \ t_1 = \text{appT} \ \theta \ s_1 \). Since \( \sigma_1 \) is an mgu of \( t_1 \) and \( s_1 \), there is a substitution \( \rho \) with \( \theta = \text{compose} \ \rho \ \sigma_1 \).

Again by assumption about \( \theta \), it holds for all \( i \) with \( 1 < i \leq n \) that \( \text{compose} \ \theta \ t_i = \text{compose} \ \theta \ s_i \). Since \( \theta = \rho \ . \ \sigma_1 \), it follows that

\[
(\text{compose} \ \rho \ \sigma_1) \ t_i = (\text{compose} \ \rho \ \sigma_1) \ s_i,
\]

and therefore,

\[
\rho \ (\text{appT} \ \sigma_1 \ t_i) = \rho \ (\text{appT} \ \sigma_1 \ s_i).
\]

Since \( \sigma_2 \) is an mgu of

\[
\text{appT}s \ \sigma_1 \ [t_2,..,t_n] \ \text{and} \ \text{appT}s \ \sigma_1 \ [s_2,..,s_n],
\]

there is a substitution \( \nu \) with \( \rho = \text{compose} \ \nu \ \sigma_2 \). Therefore, \( \theta = \text{compose} \ \rho \ \sigma_1 = \text{compose} \ (\text{compose} \ \nu \ \sigma_2) \ \sigma_1 = \text{compose} \ \nu \ (\text{compose} \ \sigma_2 \ \sigma_1) \). This shows that \( \text{compose} \ \sigma_2 \ \sigma_1 \) is more general than \( \theta \), which establishes the Lemma.

\( \blacksquare \)

Theorem 7.5 shows, by induction on the length of term lists, that if \( \text{unifyTs} \ t \ s \) does what it is supposed to do, then \( \text{unifyTlists} \) also does what it is supposed to do.

**Theorem 7.5** Suppose \( \text{unifyTs} \ t \ s \) yields a unit list containing an mgu of \( t \) and \( s \) if the terms are unifiable, and otherwise yields the empty list. Then \( \text{unifyTlists} \ ts \ ss \) yields a unit list containing an mgu of \( ts \) and \( ss \) if the lists of terms \( ts \) and \( ss \) are unifiable, and otherwise produces the empty list.

**Proof.** If the two lists have different lengths then unification fails. The implementation reflects this, in the cases for \( \text{unifyTlists} \ [\] \ (r:rs) \) and \( \text{unifyTlists} \ (t:ts) \ [\] \).

Assume, therefore, that \( ts \) and \( ss \) have the same length \( n \). We proceed by induction on \( n \).

**Basis** \( n = 0 \), i.e., both \( ts \) and \( ss \) are equal to the empty list. In this case the epsilon substitution unifies \( ts \) and \( ss \), and this is certainly an mgu.

**Induction step** \( n > 0 \). Assume \( ts = [t_1..t_n] \) and \( ss = [s_1..s_n] \), with \( n > 0 \). Then \( ts = t_1:[t_2..t_n] \) and \( ss = s_1:[s_2..s_n] \).

What the algorithm does is:
1. It checks if \( t_1 \) and \( s_1 \) are unifiable by calling \( \text{unifyTs}\ t_1\ s_1 \). By the assumption of the theorem, \( \text{unifyTs}\ t_1\ s_1 \) yields a unit list \([\sigma_1]\), with \( \sigma_1 \) an mgu of \( t_1 \) and \( s_1 \) if \( t_1 \) and \( s_1 \) are unifiable, and yields the empty list otherwise. In the second case, we know that the lists \( ts \) and \( ss \) are not unifiable, and indeed, in this case \( \text{unifyTlists}\ ts\ ss \) will produce the empty list.

2. If \( t_1 \) and \( s_1 \) have an mgu \( \sigma_1 \), then the algorithm tries to unify the lists
   \[
   ([\text{app}\ T\ \sigma_1\ t_2]..[\text{app}\ T\ \sigma_1\ t_n]),
   ([\text{app}\ T\ \sigma_1\ s_2]..[\text{app}\ T\ \sigma_1\ s_n]),
   \]
   i.e., the lists of terms resulting from applying \( \sigma_1 \) to \( t_2..t_n \) and \( s_2..s_n \). By induction hypothesis we may assume that applying \( \text{unifyTlists}\) to these two lists produces a unit list \([\sigma_2]\), with \( \sigma_2 \) an mgu of the lists, if the two lists are unifiable, and the empty list otherwise.

3. If \( \sigma_2 \) is an mgu of the two lists, then the algorithm returns a unit list containing \( \text{compose}\ \sigma_2\ \sigma_1 \). By Lemma 7.4, \( \text{compose}\ \sigma_2\ \sigma_1 \) is an mgu of \( ts \) and \( ss \).

Theorem 7.6 clinches the argument. It proceeds by structural induction on terms. The induction hypothesis will allow us to use Theorem 7.5.

**Theorem 7.6** The function \( \text{unifyTs}\ t\ s \) either yields a unit list \([u]\) or the empty list. In the former case, \( u \) is an mgu of \( t \) and \( s \). In the latter case, \( t \) and \( s \) are not unifiable.

**Proof.** Structural induction on the complexity of \((t,s)\). There are 4 cases.

1. \( t = \text{Vari}\ x\), \( s = \text{Vari}\ y \). In this case, if \( x = y \), then the epsilon substitution is surely an mgu of \( t \) and \( s \). This is what the algorithm yields. If \( x \neq y \), then the substitution \([ (x, \text{Vari}\ y) ] \) is an mgu of \( x \) and \( y \). For suppose \( \sigma_1\ x = \sigma_1\ y \). Then \( \sigma_1\ x = (\text{compose}\ \sigma_1\ [(x, \text{Vari}\ y)])\ x \), and for all \( z \neq x \), \( \sigma_1\ z = (\text{compose}\ \sigma_1\ [(x, \text{Vari}\ y)])\ z \). So \( \sigma_1 = \text{compose}\ \sigma_1\ [(x, \text{Vari}\ y)] \).

2. \( t = \text{Vari}\ x\), \( s \) not a variable. If \( x \notin \text{varsIn}\ s \), then \([ (x, s) ]\) is an mgu of \( t \) and \( s \). For if \( \sigma_1\ x = \sigma_1\ s \), then \( \sigma_1\ x = (\text{compose}\ \sigma_1\ [(x, s)])\ x \), and for all variables \( z \neq x \), \( \sigma_1\ z = (\text{compose}\ \sigma_1\ [(x, s)])\ z \). So \( \sigma_1 = \text{compose}\ \sigma_1\ [(x, s)] \).

3. \( s = \text{Vari}\ x\), \( t \) not a variable. Similar to case 2.

4. \( t = \text{Struct}\ a\ ts\) and \( s = \text{Struct}\ b\ ss\). Then \( t \) and \( s \) are unifiable iff \( a = b \), and \( ts \) and \( ss \) are unifiable. Moreover, \( u \) is an mgu of \( t \) and \( s \) iff \( a = b \) and \( u \) is an mgu of \( ts \) and \( ss \).

By the induction hypothesis, we may assume for all subterms \( t' \) and \( s' \) of \( t \) and \( s \) that \( \text{unifyTs}\ t'\ s' \) yields the empty list if \( t' \) and \( s' \) do not unify, and a unit list \([u]\), with \( u \) an mgu of \( t' \) and \( s' \) otherwise. This means the condition of Theorem 7.5 is fulfilled, and it follows that \( \text{unifyTlists}\ ts\ ss \) yields \([u]\), with \( u \) an mgu of \( ts \) and \( ss \), if the term lists \( ts \) and \( ss \) unify, and \( \text{unifyTlists}\ ts\ ss \) yields the empty list if the term lists do not unify.

This establishes the Theorem.
Some examples of unification attempts (note the type distinction between variables and terms; unifyTs applies to terms, not variables):

TPFPL> unifyTs tx (s[tx])
[]
TPFPL> unifyTs tx (s[ty])
[[[x,s[y]]]]
TPFPL> unifyTs (t[tx,a]) (t[ty,tx])
[[[x,a),(y,a)]]

### 7.3 Skolemization

For an efficient use of the unification technique from the previous section in the context of predicate logic, it is common practice to transform the formulas into a more convenient format. In particular, we will transform the formulas of predicate logic into equivalent quantifier free formulas. For this, we will replace each existential quantification by a so-called skolem term. This process is called skolemization.

**Exercise 7.7** Below, we will assume that the formulas we start out with do not contain equivalences and implications. Implement a translation function `transl :: Form -> Form` that translates every formula into an equivalent formula without occurrences of the constructors Impl and Equi.

An occurrence of $\exists v F$ in a formula expresses existential quantification if it is within the scope of an even number of negations. An occurrence of $\forall v F$ in a formula expresses existential quantification if it is within the scope of an odd number of negations. Skolemization replaces the quantifiers that express existential quantification. The skolem term for an $\exists v F$ in a positive context (within the scope of an even number of negations) will depend on all universal quantifiers that have scope over $\exists v F$, and similarly for the skolem term for a $\forall v F$ in a negative context (within the scope of an odd number of negations).

**Exercise 7.8** Replace as many of the quantifiers as you can by appropriate skolem terms.

1. $\forall x \exists y Rxy$.
2. $\neg \forall x \exists y Rxy$.
3. $\forall x \forall y (\exists z Rzx \land Rzy \Rightarrow Rxz)$.
4. $\forall x \forall y (Rxz \Rightarrow \exists z \exists w Szw)$.

In the implementation, we will maintain a list of identifiers for the variables of the outscoping universal quantifiers. A skolem term is constructed from an identifier list and an `Int` as follows. The identifiers represent all the universal parameters that the skolem term depends on.
skolem :: Int -> [Var] -> Term
skolem k vs = Struct ("sk" ++ (show k)) [ (Vari x) | x <- vs ]

An example application: “make a skolem function with index 5 depending on variables \(x, x_0, x_1\)”: 

TPFPL> skolem 5 [x,y,z]
sk5[x,y,z]

To see whether a variable occurrence \(v\) in a formula is universally quantified, we not only have to know whether \(v\) is bound by \(\forall v\) or \(\exists v\), but also whether that quantifier is in the scope of an even or an odd number of negations. An occurrence \(v\) is universally quantified if

- \(v\) is bound by \(\forall v\), and \(\forall v\) is in the scope of an even number of negations (the subformula \(\forall vF\) has positive polarity in the whole formula), or
- \(v\) is bound by \(\exists v\), and \(\exists v\) is in the scope of an odd number of negations (the subformula \(\exists vF\) has negative polarity in the whole formula).

The skolemize function \(sk\) computes by a call to \(skf\), an auxiliary function that has a list argument for the current list of wide scope quantifier indices and a Boolean argument to indicate the current polarity, and that passes a parameter for skolem indices.

sk :: Form -> Form
sk f = fst (skf f [] True 0)

Arguments of \(skf\):

1. the first argument is the current formula to be put in skolemized form,
2. the second argument is the list of identifiers for universal quantifiers that have scope over the current formula (these are needed as parameters for the next skolem term),
3. the third argument is the polarity of the current context (\(True\) for positive, \(False\) for negative),
4. the fourth argument is the next available \(Int\) for a skolem identifier (this is needed to ensure that skolem terms for different existentially quantified variables are different).

Note that the code of \(skf\) uses application of a substitution to a formula.
7.3. **SKOLEMIZATION**

```haskell
skf :: Form -> [Var] -> Bool -> Int -> (Form,Int)
skf (Atom n ts) vs pol k = ((Atom n ts),k)
skf (Conj fs) vs pol k = ((Conj fs'),j)
    where (fs',j) = skfs fs vs pol k
skf (Disj fs) vs pol k = ((Disj fs'),j)
    where (fs',j) = skfs fs vs pol k
skf (Forall x f) vs True k = ((Forall x f'),j)
    where (f',j) = skf f vs' True k
        vs' = insert x vs
skf (Forall x f) vs False k = skf (appF b f) vs False (k+1)
    where b = [(x,(skolem k vs))]
skf (Exists x f) vs True k = skf (appF b f) vs True (k+1)
    where b = [(x,(skolem k vs))]
skf (Exists x f) vs False k = ((Exists x f'),j)
    where (f',j) = skf f vs' False k
        vs' = insert x vs
skf (Neg f) vs pol k = ((Neg f'),j)
    where (f',j) = skf f vs (not pol) k
```

**skfs** puts lists of formulas in skolemized form. Same arguments as **skf**.

```haskell
skfs :: [Form] -> [Var] -> Bool -> Int -> ([Form],Int)
skfs [] _ _ _ k = ([],k)
skfs (f:fs) vs pol k = ((f':fs'),j)
    where
    (f', j1) = skf f vs pol k
    (fs', j) = skfs fs vs pol j1
```

Some example formulas and their skolemized forms. First some relation symbols:

```plaintext
p = Atom "p"
q = Atom "q"
r = Atom "r"
```

Then some examples of relational properties expressed in predicate logic.
refl  = Forall x (r [tx,tx])
irrefl = Forall x (Neg (r [tx,tx]))
trans = Forall x (Forall y (Forall z 
(Disj [Neg (r [tx,ty]),Neg (r [ty,tz]),r [tx,tz]])))
ctrans = Forall x (Forall y (Forall z 
(Disj [r [tx,ty], r [ty,tz],Neg (r [tx,tz]]))))
symm  = Forall x (Forall y 
(Disj [Neg (r [tx,ty]), r [ty,tx]]))
asymm = Forall x (Forall y 
(Disj [Neg (r [tx,ty]), Neg (r [ty,tx]]))
serial  = Forall x (Exists y (r [tx,ty]))
serial1 = Forall x (Forall y (Exists z (r [tx,ty,tz])))
serial2 = Forall x (Exists y (Exists z (r [tx,ty,tz])))
relprop1 = Disj [(Neg asymm),irrefl]
relprop2 = Disj [(Neg trans),(Neg irrefl),asymm]
relprop3 = Disj [(Neg trans),(Neg symm),(Neg serial),refl]

And some example skolemizations:

TPFPL> serial
Ax Ey r[x,y]
TPFPL> serial1
Ax Ay Ez r[x,y,z]
TPFPL> serial2
Ax Ey Ez r[x,y,z]
TPFPL> sk serial
Ax r[x,sk0[x]]
TPFPL> sk serial1
Ax Ay r[x,y,sk0[x,y]]
TPFPL> sk serial2
Ax r[x,sk0[x],sk1[x]]
TPFPL> sk (Neg serial)
¬Ey r[sk0,y]
TPFPL> sk (Neg serial1)
¬Ez r[sk0,sk1,z]
TPFPL> sk (Neg serial2)
¬Ey Ez r[sk0,y,z]
TPFPL> sk (Disj [serial,serial1,serial2])
disj[Ax r[x,sk0[x]],Ax Ay r[x,y,sk1[x,y]],Ax r[x,sk2[x],sk3[x]]]
TPFPL> sk (Neg (Disj [serial,serial1,serial2]))
¬disj[ Ey r[sk0,y], Ez r[sk1,sk2,z], Ey Ez r[sk3,y,z]]
7.4 Detecting Inconsistencies with Semantic Tableaux

The method of semantic tableaux was invented in the 1950s by Beth [Bet59] and Hintikka [Hin55] to provide a systematic procedure for detecting logical inconsistencies. If there is an inconsistency in a (finite) set of formulas, we are bound to find it, provided we spell out all the possible states of affairs, while keeping track of all the basic inconsistencies that we encounter, where a basic inconsistency is the presence of a pair of alleged facts \( a, \neg a \).

The semantic tableau method proceeds in a step by step fashion, in each step replacing a check of an inconsistency by a simpler check, by decomposition of sentences and distinction of cases. Complex sentences come in two flavours: conjunctive compositions and disjunctive compositions. The disjunctive compositions are the ones that give rise to a case distinction. Following Smullyan [Smu68], we call conjunctive compositions \( \alpha \)-sentences and disjunctive compositions \( \beta \)-sentences. Here is what their components look like:

<table>
<thead>
<tr>
<th>conjunctive</th>
<th>disjunctive</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha )</td>
<td>( \alpha_1 )</td>
</tr>
<tr>
<td>( P \land Q )</td>
<td>( P )</td>
</tr>
<tr>
<td>( \neg(P \lor Q) )</td>
<td>( \neg P )</td>
</tr>
<tr>
<td>( \neg(P \Rightarrow Q) )</td>
<td>( P )</td>
</tr>
<tr>
<td>( \neg(P \Leftarrow Q) )</td>
<td>( \neg P )</td>
</tr>
</tbody>
</table>

This extends to longer disjunctions and conjunctions in an obvious way. For instance, the sentence \( \neg(y_1 \lor b_1 \lor g_1 \lor o_1) \) is an \( \alpha \)-sentence, with decomposition \( \neg y_1, \neg b_1, \neg g_1, \neg o_1 \). Note that \( \neg y_1 \land \neg b_1 \land \neg g_1 \land \neg o_1 \) is also an \( \alpha \) sentence, with the same decomposition.

A tableau for a set of sentences is constructed by applying to sentences from the set the following decomposition rules:

\[
\begin{array}{c|c|c}
\neg P & \alpha_1 & \beta_2 \\
\hline
\alpha & \alpha_2 & \\
\end{array}
\]

In tableau matters, an example picture to get across the idea often conveys more than a fully spelt out procedure in words. Here is a tableau tree for checking the consistency of \( \neg((P \Rightarrow Q \land Q \Rightarrow R) \Rightarrow (P \Rightarrow R)) \).
The picture shows that first \( \neg((P \Rightarrow Q \land Q \Rightarrow R) \Rightarrow (P \Rightarrow R)) \) was decomposed in its \( \alpha_1 \) and \( \alpha_2 \) parts \( P \Rightarrow Q \land Q \Rightarrow R \) and \( \neg(P \Rightarrow R) \). In the next step, the sentence \( P \Rightarrow Q \land Q \Rightarrow R \) was decomposed into its \( \alpha_1 \) and \( \alpha_2 \) parts \( P \Rightarrow Q \quad Q \Rightarrow R \). Next, \( \neg(P \Rightarrow R) \) was decomposed into its \( \alpha_1 \) and \( \alpha_2 \) parts \( P \) and \( \neg R \). At this point the \( \beta \)-sentence \( P \Rightarrow Q \) got tackled, causing a split into its \( \beta_1 \) and \( \beta_2 \) components \( \neg P \) and \( Q \). Finally, in both branches the \( \beta \)-sentence \( Q \Rightarrow R \) got decomposed, causing a further split on both sides.

The tableau shows that the sentence we started out with is inconsistent, for a check of the four branches reveals the pair \( P, \neg P \) along the first branch, the pairs \( P, \neg P \) and \( \neg R, R \) along the second branch, the pair \( Q, \neg Q \) along the third branch, and the pair \( \neg R, R \) along the fourth branch. This shows that all these avenues are closed. In fact, there was no need for the final decomposition step of \( Q \Rightarrow R \) on the left hand side, for once a tableau branch is closed, the search for a consistent state of affairs along that avenue is over.

Now if \( \neg((P \Rightarrow Q \land Q \Rightarrow R) \Rightarrow (P \Rightarrow R)) \) is inconsistent, \( (P \Rightarrow Q \land Q \Rightarrow R) \Rightarrow (P \Rightarrow R) \) has to be true no matter what. The tableau method can be viewed as a refutation method: we have tried to refute \( (P \Rightarrow Q \land Q \Rightarrow R) \Rightarrow (P \Rightarrow R) \), but in vain, so we have discovered a truth of logic.

Next consider \( P \Rightarrow Q \land Q \Rightarrow R \land \neg R \). This gives the following tableau; this time we indicate closure of a branch by means of \( \times \).
The example indicates that the sentence we started out with is consistent, because in the state of affairs $\neg P, \neg Q, \neg R$ the sentence is true.

Some reflection shows that the sentences that we can harvest from a fully developed open branch in a tableau satisfy a number of simple requirements:

- For no sentence $P$ are both $P$ and $\neg P$ present in the set.
- If a doubly negated sentence $\neg\neg P$ is present in the set, then $P$ is also present.
- If an $\alpha$-sentence is present in the set, then both its $\alpha_1$ and $\alpha_2$ components are present.
- If a $\beta$-sentence is present in the set, then either its $\beta_1$ or $\beta_2$ component is present.

A set of sentences satisfying these requirements is called a Hintikka set.

A propositional state of affairs is called a Boolean valuation: if we consider the proposition letters as variables, a Boolean valuation is a function from the set of proposition letters to the truth values $t$ and $f$. Every Hintikka set can be taken as an approximation of a valuation: map propositional variable $a$ to $t$ if $a$ is in the Hintikka set, to $f$ if $\neg a$ is in the Hintikka set. In general, a Hintikka set is not a full specification of a valuation, for there may be basic propositions $a$ that are not decided by the Hintikka set (neither $a$ nor $\neg a$ is in the set).

It is clear that propositional tableaux can be viewed as a systematic procedure for constructing Hintikka sets. If a Hintikka set for a propositional sentence exists, then the tableau method will find it in a finite number of steps. A propositional tableau is fair if for any of its branches holds that either the branch is closed, or any $\alpha$ sentence has both its $\alpha_1$ and its $\alpha_2$ component on the branch, any $\beta$ sentence has either its $\beta_1$ or its $\beta_2$ component on the branch, and any doubly negated sentence $\neg\neg P$ has $P$ on the branch. It takes a finite number of steps to develop a fair tableau for a propositional sentence. Tableaux are a decision method for propositional consistency or propositional satisfiability.

From Propositional Logic to Predicate Logic If we make the step from propositional logic to predicate logic by allowing predicates and quantification over individuals satisfying those predicates, things get more involved, and more interesting. We now allow sentences of the
form ‘for all $x$ $F$ holds’, or symbolically $\forall x F$, and ‘for some $x$ $F$ holds’, or symbolically $\exists x F$.

The intended meaning is clearly that with respect to a certain universe $U$ a universal sentence $\forall x F$ is true if and only if $F$ remains true, no matter which element $d$ in $U$ we let $x$ refer to. Similarly, an existential sentence $\exists x F$ is true if and only if $F$ remains true for at least one choice of an element $d$ in $U$ that $x$ can refer to. It is useful to have a notation for “$F$, with $x$ interpreted as $d$”. For this, we use $[x \mapsto d]F$.

The tableau treatment of quantification reflects the way quantifiers are dealt with in mathematical reasoning. If it has been established in the course of a mathematical argument that there exists an object having a certain property $P$, then it is customary to say: let $a$ be such an object. This boils down to giving one of the things satisfying $P$ the name $a$. In such cases one always has to make sure that $a$ is not used as a name for anything else: after all, we have not established that anything has $P$, but just that at least one thing has $P$. But as long as the baptism does not clash with other naming conventions the switch from $\exists x P x$ to $Pa$ is legitimate. To be on the safe side, one should take a fresh name.

The Smullyan classification of sentences gets extended with universal or $\gamma$ sentences and existential or $\delta$ sentences.

<table>
<thead>
<tr>
<th>$\gamma$</th>
<th>$\gamma_1$</th>
<th>$\delta$</th>
<th>$\delta_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall x F$</td>
<td>$[x \mapsto d]F$, $d$ any name</td>
<td>$\exists x F$</td>
<td>$[x \mapsto d]F$, $d$ a new name</td>
</tr>
<tr>
<td>$\neg \exists x F$</td>
<td>$[x \mapsto d](\neg F)$, $d$ any name</td>
<td>$\neg \forall x F$</td>
<td>$[x \mapsto d](\neg F)$, $d$ a new name</td>
</tr>
</tbody>
</table>

To facilitate talk about $\gamma$ and $\delta$ sentences and their components, we agree to call the $\gamma_1$ component of a certain $\gamma$ formula, with $d$ as the chosen name, $\gamma(d)$. Similarly, we agree to call the $\delta_1$ component of a certain $\delta$ sentence, with $d$ as the new name, $\delta(d)$. Thus, if the $\gamma$ sentence is $\forall x R x b$, and the chosen name is $b$, then $\gamma(b)$ indicates the sentence $Rbb$.

A tableau for a set of sentences of predicate logic is constructed by applying to sentences from the set the following decomposition rules:

$$
\begin{array}{c}
\neg \neg F \\
\hline
\alpha \\
\hline
\alpha_1 \quad \beta_1 \quad \beta_2 \quad \gamma \quad \delta_1 \\
\hline
\alpha_2 \quad d \text{ new}
\end{array}
$$

It is not difficult to see that the propositional and the $\gamma$ tableau rules preserve consistency. For the $\delta$ rule, we state and prove the consistency as follows:

**Proposition 7.9** If $S$ is satisfiable, $\delta \in S$, and $d$ is any name that occurs nowhere in $S$, then $S \cup \{\delta(d)\}$ is satisfiable.

Proof: satisfiability of $S$ means that there is a universe $U$ and an interpretation $I$ of the predicates from $S$ in $U$, together with an assignment $s$ that maps the constants in $S$ to elements of $U$, such that for every $F \in S$ it holds that $F_s$ is true in $(U, I)$. In particular, $\delta_s$ is true in $(U, I)$. From this it follows that for at least one $u \in U$, $\delta(u)_s$ is true in $(U, I)$. Now let $s'$ be the assignment that extends $s$ by sending $d$ to $u$. Then $s'$ is defined for all constants in $S \cup \{\delta(d)\}$, and, since $\delta(d)_{s'}$ equals $\delta(u)_s$, $\delta(d)_{s'}$ is true in $(U, I)$.
If a tableau branch is consistent, i.e., if there is a model that makes all formulas along the branch true, then extending the tableau by means of one of the tableau decomposition rules will result in a consistent tableau. It follows from this that the tableau method is sound:

**Theorem 7.10 (Soundness)** If $F$ is consistent, then any tableau for $F$ will have an open branch.

---

![Figure 7.1: Closed Tableau.](image)

\[ \neg(\forall x (P x \Rightarrow Q x) \Rightarrow (\forall x P x \Rightarrow \forall x Q x)) \]

\[ \forall x (P x \Rightarrow Q x) \]

\[ \neg(\forall x P x \Rightarrow \forall x Q x) \]

\[ \forall x P x \]

\[ \neg \forall x Q x \]

\[ \neg Q d_1 \]

\[ P d_1 \]

\[ P d_1 \Rightarrow Q d_1 \]

\[ \neg P d_1 \]

\[ Q d_1 \]

The example tableau in Figure 7.1 establishes that $\neg(\forall x (P x \Rightarrow Q x) \Rightarrow (\forall x P x \Rightarrow \forall x Q x))$ is inconsistent, or, in other words, that $\forall x (P x \Rightarrow Q x) \Rightarrow (\forall x P x \Rightarrow \forall x Q x)$ is a predicate logical validity.

Note that the object (named) $d_1$ gets introduced into the tableau by decomposition of the $\delta$ sentence $\neg \forall x Q x$. This yields $\delta(d_1) = \neg Q d_1$. Next, apply the $\gamma$ rules, for object $d_1$, to the sentences $\forall x P x$ and $\forall x (P x \Rightarrow Q x)$.

To start off a tableau for a $\gamma$ sentence, assume the universe is non-empty, so it contains an object $d_1$:
Since $\gamma$ formulas impose a standing obligation (for all names $d$ that turn up along a tableau branch, the appropriate $\gamma(d)$ sentence has to be added), predicate logical tableaux can run on indefinitely, as is demonstrated in the tableau for

$$\forall x \exists y Rxy \land \forall x \forall y \forall z ((Rxy \land Ryz) \Rightarrow Rxz) \land \forall x \lnot Rxx$$

in Figure 7.2. In this example case, this infinite tree generation process is due to the format of the $\delta$ rule; a slight relaxation would allow us to re-use $d_1$ as the object satisfying $\exists y Rd_2 y$. This would have resulted in an open branch corresponding to the model $\bullet \nrightarrow \bullet$.

But it is easy to see that clever emendations of the tableau rules cannot remedy the situation in general. The extra requirement of asymmetry rules out loops. Thus, the following formula only has infinite models $\bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \ldots$

$$\forall x \exists y Rxy \land \forall x \forall y \forall z ((Rxy \land Ryz) \Rightarrow Rxz) \land \forall x \lnot Rxx \land \forall x \forall y (Rxy \Rightarrow \lnot Ryx).$$

A tableau branch corresponding to a model for this sentence is bound to be infinite.

The requirements for a Hintikka set for predicate logic reflect the treatment of universal and existential sentences. A Hintikka set, for a universe $U$, now is a set of sentences satisfying the following:

- For no sentence $F$ are both $F$ and $\lnot F$ present in the set.
- If a doubly negated sentence $\lnot \lnot F$ is present in the set, then $F$ is also present.
- If an $\alpha$-sentence is present in the set, then both its $\alpha_1$ and $\alpha_2$ components are present.
- If a $\beta$-sentence is present in the set, then either its $\beta_1$ or $\beta_2$ component is present.
- If a $\gamma$-sentence is present in the set, then for all $d \in U$, $\gamma(d)$ is in the set.
- If a $\delta$-sentence is present in the set, then for at least one $d \in U$, $\delta(d)$ is present in the set.

Again, fair tableau development should yield Hintikka sets along the open branches. The requirement on $\gamma$ sentences leads to a crucial difference with the propositional case: there is no longer a guarantee that fair tableaux are finite. As a consequence, tableaux in predicate logic are not a decision engine for predicate logical satisfiability.

Crucial for establishing completeness of the tableau method for predicate logic (“if $F$ has a closed tableau then $F$ is not satisfiable”) is the following satisfiability lemma for Hintikka sets.
\[ \forall x \exists y Rxy \land \forall x \forall y \forall z ((Rxy \land Ryz) \Rightarrow Rxz) \land \forall x \neg Rxx \]

\begin{itemize}
  \item \[ \forall x \exists y Rxy \] \[ \forall x \forall y \forall z ((Rxy \land Ryz) \Rightarrow Rxz) \] \[ \forall x \neg Rxx \] \[ \exists y Rd_{1y} \] \[ \neg Rd_{1d_1} \] \[ Rd_{1d_2} \] \[ \neg Rd_{2d_2} \] \[ (Rd_{1d_2} \land Rd_{2d_1}) \Rightarrow Rd_{1d_1} \] \[ \neg (Rd_{1d_2} \land Rd_{2d_1}) \] \[ Rd_{1d_1} \times \] \[ \neg Rd_{1d_2} \times \] \[ \neg Rd_{2d_1} \] \[ \exists y Rd_{2y} \] \[ Rd_{2d_3} \] \[ \neg Rd_{3d_3} \] \[ : \]
\end{itemize}
Lemma 7.11 Every Hintikka set $H$ for a domain $U$ is satisfiable in $U$.

The proof of this is a matter of constructing an appropriate predicate logical model from a Hintikka set. To specify the model, simply give any sentence $Pt_1 \cdots t_n$ the value $t$ if $Pt_1 \cdots t_n$ is in $H$, the value $f$ if $Pt_1 \cdots t_n$ is not in $H$. The terms $t_i$ can range over individual variables of the language, together with names from $U$. If the language has individual constants, then these should be included in $U$. Next, show by induction on sentence structure for each sentence $F$ of $H$ that $F$ will get the value $t$ in this model.

To use this for a completeness proof, we have to establish that open branches in fully developed tableaux yield Hintikka sets. Unfortunately, this is not true without further ado. Since $\gamma$ rules have to be repeated we can get infinite tableaux. Since tableaux are finitely branching trees, by König’s lemma (“Every finitely branching infinite tree has an infinite branch”), an infinite tableau must have an infinite branch. Infinite branches are open, of course, but to make them fair we need to develop the tableau according to a strategy guaranteeing a fair treatment of every formula. In particular, not only do all $\alpha$ and $\beta$ sentences have to be decomposed into their $\alpha_1, \alpha_2$ or $\beta_1, \beta_2$ components, and all $\delta$ sentences have to spawn a $\delta(d)$ for a fresh $d$, but all $\gamma$ sentences have to generate $\gamma(d)$ on a branch for all $d$ occurring on the branch. One way to achieve this is to let an application to a $\gamma$ rule to a branch be followed by the act of putting a copy of the $\gamma$ formula at the end of the branch [Smu68].

A fair tableau is finished if it cannot be extended by further applications of the fair tableau development procedure. For finished fair tableaux we have that open branches correspond to Hintikka sets. This yields: if $F$ has a finished fair tableau with an open branch, then $F$ is consistent. In other words:

Theorem 7.12 (Completeness) If $F$ is a contradiction, then a fair tableau procedure will yield a closed tableau for $F$ after finitely many steps.

The formulation of the completeness theorem illustrates once more that tableau reasoning is a refutation method. To show that $F$ is a theorem, try to refute $\neg F$ by means of fair tableau development. If this procedure terminates with a closed tableau then $F$ is a theorem. Note again that tableau reasoning for predicate logic is not a decision algorithm: the fair tableau development for $\neg F$ may run forever.

Free Variable Tableaux A problem with the implementation of tableau reasoning for predicate logic is with the efficiency of the treatment of the $\gamma$ sentences. A $\gamma$ formula has to yield $\gamma(d)$ for all $d$ along the branch, but which $d$ should be tried first?

The idea of free variable tableaux [Fit96, Hae01] is to postpone this choice by letting a $\gamma$ formula yield $\gamma(x)$, for a free variable $x$. This allows the use of unification in checking for closure. But one has to be careful, as the following example illustrates:
7.4. DETECTING INCONSISTENCIES WITH SEMANTIC TABLEAUX

\[ \forall x (P_x \lor Q_x) \land \neg P_a \land \neg Q_b \]

\[ \begin{align*}
\forall x (P_x \lor Q_x) \\
\neg P_a \\
\neg Q_b \\
P_x \lor Q_x \\
P_x \\
Q_x \\
\times \\
\times \\
x \leftarrow a \\
x \leftarrow b
\end{align*} \]

This may look like a closed tableau, but in fact the sentence we started out with is consistent: take universe \{a, b\} and let \(P_b, Q_a\) give the interpretation of \(P\) and \(Q\). The problem is that the free variable \(x\) is a so-called *rigid* variable: it occurs on both sides of a tableau split. The trouble with rigid variables is that they cannot be interpreted universally on a single tableau branch, but have to be read universally on the whole tableau. Thus, closure cannot be checked on single branches, but should be checked globally, for the whole tableau.

Another complication is that introduction of new names in \(\delta\) rule applications may introduce hidden dependencies on free variables. Consider our earlier example:

\[ \forall x \exists y R_{xy} \]

\[ \exists y R_{xy} \]

\[ R_x d_1 \]

This tableau suggests, misleadingly, that \(R_x d_1\) will be true irrespective of the reference of \(x\). This is wrong, for it blurs the scope distinction between \(\forall x \exists y R_{xy}\) and \(\exists y \forall x R_{xy}\). In fact, the choice of \(d_1\) is a dependent choice. This has to be made explicit by the use of a skolem function:

\[ \forall x \exists y R_{xy} \]

\[ \exists y R_{xy} \]

\[ R_x f(x) \]
By far the easiest way to deal with $\delta$ sentences is by prefixing a translation step to skolemized formulas to the tableau procedure, so that the tableau rules do not have to deal with $\delta$ sentences at all.

### 7.5 Automating Tableau Reasoning

To automate tableau reasoning, we need a marriage of tableau development and unification for checking tableau closure. To perform unification on tableaux, let a tableau branch consist of two lists of terms (the terms corresponding to the positive literals, and the terms corresponding to the negative literals), plus a list of pending formulas. A node of a tableau consists of an index in the tableau tree, plus information about the initial segment of the tableau branch up to that node:

```haskell
data Node = Nd Index [Term] [Term] [Form] deriving Show
```

The tree indexing scheme that we will use is illustrated in the following example tree:

![Example Tree]

A tableau, expanded to a certain depth, is a list of nodes:

```haskell
type Tableau = [Node]
```

As we work with formulas in skolemized form, there are no $\delta$ rule applications. Here is code for distinguishing between $\alpha$, $\beta$ and $\gamma$ formulas:
7.5. AUTOMATING TABLEAU REASONING

\[
\begin{align*}
\text{alpha} & : \text{Form} \rightarrow \text{Bool} \\
\text{alpha} \ (\text{Conj} \ _) & = \text{True} \\
\text{alpha} \ (\text{Neg} \ (\text{Disj} \ _)) & = \text{True} \\
\text{alpha} \ _ & = \text{False} \\
\text{beta} & : \text{Form} \rightarrow \text{Bool} \\
\text{beta} \ (\text{Disj} \ _) & = \text{True} \\
\text{beta} \ (\text{Neg} \ (\text{Conj} \ _)) & = \text{True} \\
\text{beta} \ _ & = \text{False} \\
\text{gamma} & : \text{Form} \rightarrow \text{Bool} \\
\text{gamma} \ (\text{Forall} \ _ \ _) & = \text{True} \\
\text{gamma} \ (\text{Neg} \ (\text{Exists} \ _ \ _)) & = \text{True} \\
\text{gamma} \ _ & = \text{False}
\end{align*}
\]

Code for identifying the positive literals, the negative literals, and the double negations:

\[
\begin{align*}
\text{plit}, \ \text{nlit}, \ \text{dneg} & : \text{Form} \rightarrow \text{Bool} \\
\text{plit} \ (\text{Atom} \ n \ ts) & = \text{True} \\
\text{plit} \ _ & = \text{False} \\
\text{nlit} \ (\text{Neg} \ (\text{Atom} \ n \ ts)) & = \text{True} \\
\text{nlit} \ _ & = \text{False} \\
\text{dneg} \ (\text{Neg} \ (\text{Neg} \ f)) & = \text{True} \\
\text{dneg} \ _ & = \text{False}
\end{align*}
\]

Function for converting a literal (an atom or a negation of an atom) to a term:

\[
\begin{align*}
\text{f2t} & : \text{Form} \rightarrow \text{Term} \\
\text{f2t} \ (\text{Atom} \ n \ ts) & = \text{Struct} \ n \ ts \\
\text{f2t} \ (\text{Neg} \ (\text{Atom} \ n \ ts)) & = \text{Struct} \ n \ ts
\end{align*}
\]

The components of a (non-literal) formula are given by:
CHAPTER 7. THEOREM PROVING FOR PREDICATE LOGIC

\[
\begin{align*}
\text{components} :: \text{Form} & \to [\text{Form}] \\
\text{components} \ (\text{Conj} \ fs) & = fs \\
\text{components} \ (\text{Disj} \ fs) & = fs \\
\text{components} \ (\text{Neg} \ (\text{Conj} \ fs)) & = \text{map} \ (\ \backslash \ f \to \text{Neg} \ f) \ fs \\
\text{components} \ (\text{Neg} \ (\text{Disj} \ fs)) & = \text{map} \ (\ \backslash \ f \to \text{Neg} \ f) \ fs \\
\text{components} \ (\text{Neg} \ (\text{Neg} \ f)) & = [f] \\
\text{components} \ (\text{Forall} \ x \ f) & = [f] \\
\text{components} \ (\text{Neg} \ (\text{Exists} \ x \ f)) & = [\text{Neg} \ f]
\end{align*}
\]

For universal sentences, the following function returns the binder:

\[
\begin{align*}
\text{binder} :: \text{Form} & \to \text{Var} \\
\text{binder} \ (\text{Forall} \ x \ f) & = x \\
\text{binder} \ (\text{Neg} \ (\text{Exists} \ x \ f)) & = x
\end{align*}
\]

For universal sentences, the following function allows to strip the list of all universal quantifiers:

\[
\begin{align*}
\text{decompose} :: \text{Form} & \to ([\text{Var}],\text{Form}) \\
\text{decompose} \ \text{form} & = \text{decomp} \ [] \ \text{form} \ \text{where} \\
\text{decomp} \ \text{xs} \ \text{f} & = \text{if} \ \gamma \ \text{f} \ \text{then} \ \text{decomp} \ (\text{xs} \ ++ \ [x]) \ \text{f'} \ \text{else} \ (\text{xs},\text{f}) \\
\text{where} \ x & = \text{binder} \ \text{f} \\
\text{[f']} & = \text{components} \ \text{f}
\end{align*}
\]

It is convenient to prune branches as quickly as possible, by removing nodes that close under any substitution.

Note that \(\forall\) formulas are not removed from the list of pending formulas: if \(\forall x F\) is treated, \(\forall x F\) gets replaced at the head of the formula list by a renaming of \(F\), and \(\forall x F\) gets appended to the formula list. Similarly, if \(\neg \exists x F\) is treated, \(\neg \exists x F\) gets replaced at the head of the formula list by a renaming of \(\neg F\), and \(\neg \exists x F\) gets appended to the formula list. Because of the fact that \(\gamma\) formulas are not decomposed tableau expansion can go on forever. To ensure that renamed variables are fresh to the tableau, we use the node index as a new variable index.

An expansion step of a branch looks like this.
7.5. AUTOMATING TABLEAU REASONING

The treatment of γ formulas is a potential source of infinitary behaviour. We can set an arbitrary boundary by keeping track of the number of γ rule applications. For this, we need a version of \texttt{step} with a parameter for γ-depth [Fit96].

```haskell
step :: Node \to Tableau
step (Nd i pos neg []) = [Nd i pos neg []]
step (Nd i pos neg (f:fs))
  \mid plit f = if \text{elem} (f2t f) neg
    then [] else [Nd i ((f2t f):pos) neg fs]
  \mid nlit f = if \text{elem} (f2t f) pos
    then [] else [Nd i pos ((f2t f):neg) fs]
  \mid dneg f = [Nd i pos neg ((components f) ++ fs)]
  \mid alpha f = [Nd i pos neg ((components f) ++ fs)]
  \mid beta f = [(Nd (i++[n]) pos neg (f':fs)) |
    (f',n) \leftarrow \text{zip} (\text{components} f) [0..] ]
  \mid gamma f = [Nd i pos neg (f':(fs++[f]))]
where
  (xs,g) = \text{decompose} f
  b = [((\text{Var name j}), \text{Var} (\text{Var name i})) \mid (\text{Var name j}) \leftarrow xs ]
  f' = \text{appF} b g
```

A tableau node is fully expanded if its list of pending formulas is empty.
expanded :: Node -> Bool
expanded (Nd i pos neg []) = True
expanded _ = False

To expand a tableau up to a given positive $\gamma$ depth $n$, apply expansion steps to the first node that needs expansion, until the $\gamma$ depth gets decreased, next move on with the next node. This ensures that the nodes are treated fairly. Recursively carry out this procedure until the $\gamma$ depth becomes 0 or the tableau is fully expanded.

expand :: Int -> Tableau -> Tableau
expand 0 tableau = tableau
expand _ [] = []
expand n (node:nodes) = if expanded node
                          then (node:(expand n nodes))
                          else if k == n
                               then expand n (newnodes ++ nodes)
                               else expand (n-1) (nodes ++ newnodes)
                          where (k,newnodes) = stepD n node

For debugging, we also give a version that prints nodes as we go along. This uses the trace function that allows us to insert appropriate strings to be printed as expand gets computed.

expandD :: Int -> Tableau -> Tableau
expandD 0 tableau = tableau
expandD _ [] = []
expandD n (node:nodes) = if expanded node
                          then
                            trace (show node ++ "\n\n")
                            (node:(expandD n nodes))
                          else if k == n
                               then
                                 trace (show node ++ "\n\n")
                                 (expandD n (newnodes ++ nodes))
                               else
                                 trace (show node ++ "\n\n")
                                 (expandD (n-1) (nodes ++ newnodes))
                          where (k,newnodes) = stepD n node
To check a branch for closure at a node, we make use of the fact that the literals are represented as terms, so we can apply \texttt{unifyTs} to its lists of positive and negative literals. A node closes when it is possible to unify one of its positive literals against one of its negative literals. The unifying substitutions are needed to instantiate the rest of the tableau, so we collect them in a list.

\begin{verbatim}
    checkN :: Node -> [Subst]
    checkN (Nd _ pos neg _) = 
        concat [ unifyTs p n | p <- pos, n <- neg ]
\end{verbatim}

To check a tableau for closure, we try to close its first branch. For any closing substitution $\sigma$ that we get, we try to close the $\sigma$ image of the remaining branches. For this we need functions for applying substitutions to nodes and tableaux.

\begin{verbatim}
    appNd :: Subst -> Node -> Node
    appNd b (Nd i pos neg forms) = 
        Nd i (appTs b pos) (appTs b neg) (appFs b forms)

    appTab :: Subst -> Tableau -> Tableau
    appTab = map . appNd
\end{verbatim}

The function \texttt{checkT} is used in the closure check for a tableau. Note that a tableau consisting of an empty list of nodes counts a closed, because all of its nodes close. A tableau is closed if its list of closing substitutions is non-empty.

\begin{verbatim}
    checkT :: Tableau -> [Subst]
    checkT [] = [epsilon]
    checkT [node] = checkN node
    checkT (node:nodes) = 
        concat [ checkT (appTab s nodes) | s <- checkN node ]
\end{verbatim}

No-one can devise a program that decides FOL theoremhood, as Church and Turing discovered in the 1930s. Our refutation engine will expand a formula to a given $\gamma$ depth, and then check for closure.

The function \texttt{initTab} creates an initial tableau for a formula.
The function \texttt{refuteDepth} tries to refute a formula by expanding a tableau for it up to a given \( \gamma \) depth. Note: to turn on debugging, replace \texttt{expand} by \texttt{expandD} in this definition.

\begin{verbatim}
refuteDepth :: Int -> Form -> Bool
refuteDepth k form = checkT tableau /= []
    where tableau = expand k (initTab form)
      -- where tableau = expandD k (initTab form)
\end{verbatim}

To prove that a formula is a theorem, negate it, put it in skolemized form, and feed it to the refutation engine (for a given \( \gamma \) depth):

\begin{verbatim}
thm :: Int -> Form -> Bool
thm n = (refuteDepth n) . sk . Neg
\end{verbatim}

To check whether a formula is satisfiable, put it in skolemized form, feed it to the refutation engine (for a given \( \gamma \) depth), and negate the answer.

\begin{verbatim}
sat :: Int -> Form -> Bool
sat n = not . (refuteDepth n) . sk
\end{verbatim}

We use the tableau engine to prove that every transitive, symmetric and serial relation is reflexive.

\begin{verbatim}
formula = Disj [Neg trans, Neg symm, Neg serial, refl]
\end{verbatim}

We get:

TPFPL> thm 10 formula
False
TPFPL> thm 20 formula
True
This shows that the formula is a theorem of predicate logic.

**Exercise 7.13** Use the tableau engine to show that every strict partial order is asymmetric. (A strict partial order is a relation that is transitive and irreflexive.)

**Exercise 7.14** Use the tableau engine to show that every asymmetric relation is irreflexive.

### 7.6 Further Reading

The classic account of tableau reasoning for first order logic is [Smu68]. An illuminating textbook on theorem proving with first order logic is [Fit96].
Chapter 8

Parsing

Summary

In this chapter we will first look at the general problem of recognizing and parsing context
free languages. Next, we introduce the notion of parser combinators, and show how these can
be used to create parsers from sets of grammar rules. Parser combinators are functions that
transform parsers for a language into parsers for a different language. We apply these insights
in the construction of a recursive descent parser for a phrase structure grammar with a feature
agreement component. Finally, we use the example of relative clause formation to demonstrate
how parser combinators can be used to capture the effects of movement rules.

8.1 Recognizing and Parsing Context Free Languages

We declare a module for the chapter that imports the standard List and Char modules, and a
lexicon module LexDB that will be specified in Section 8.6

```haskell
module Parsing
where
    import List
    import Char
    import LexDB
```

A context free language or CF language is a language that is generated by a set of context free
rules. A rule is context free if it is of the form $V \rightarrow T_1 \cdots T_n$, where $V$ is a rewrite symbol, and
the $T_i$ are either terminal symbols or rewrite symbols. Most example language fragments from
Chapter 1 are context free.
Remark. Whether the language of predicate logic is CF depends on whether one insists that variable binding is non-vacuous. Variable binding is non-vacuous if each quantifier binds at least one variable occurrence. Under this assumption the language is not CF, as can be shown with a pumping argument. Something similar holds for programming languages like Pascal, C and Java: if variables can only be declared if they are actually used in the program text, then these languages are not CF. See, e.g., [AU72].

Recognizing strings of terminal symbols with a context free grammar is deciding whether a string \( w \) of terminal symbols of the language can be generated from \( V \), for any rewrite symbol of the grammar. If the answer is ‘yes’ then \( w \) is in the \( V \) part of the language, if the answer is ‘no’ then it isn’t.

We will demonstrate a recognizer for an example grammar. One of the ingredients is a function for splitting strings in substrings. The following function gives all possible ways of splitting a string into two substrings:

\[
\text{split2} :: [a] \to \{([a],[a])\} \\
\text{split2} [] = \{([],[])\} \\
\text{split2} (x:xs) = \\
\quad (\{\},(x:xs)):(\text{map} (\ \ (ys,zs) \to ((x:ys),zs)) \ (\text{split2} xs))
\]

This gives:

\[
\text{Parsing} > \text{split2} \ "Jan" \\
\{("","Jan"),("J","an"),("Ja","n"),("Jan","")\}
\]

The following function gives all possible ways of splitting a string into \( n \) substrings, for \( n \geq 2 \). The function uses \text{split2} for the case \( n = 2 \), and \text{split2} plus recursion for the case \( n > 2 \).

\[
\text{splitN} :: \text{Int} \to [a] \to \{[[a]]\} \\
\text{splitN} n xs | n <= 1 \quad = \text{error} \ "cannot split" \\
\quad | n == 2 \quad = \{ [ys,zs] | (ys,zs) \leftarrow \text{split2} xs \} \\
\quad | \text{otherwise} = \{ ys:yss | (ys,zs) \leftarrow \text{split2} xs, \\
\quad \quad \quad \quad yss \leftarrow \text{splitN} (n-1) zs \}
\]

This gives, e.g.:

\[
\text{Parsing} > \text{splitN} 3 \ "Jan" \\
\{["","","Jan"],[","","J","an"],[","","Ja","n"],[","","Jan",""],"J","","an"], \\
["J","a","n"],["J","an",""],/["Ja","","n"],["Ja","n",""],["Jan","","""]\}
\]
Now consider the following CF grammar:

\[ A \rightarrow \epsilon | a | b | c | aAa | bAb | cAc. \]

This grammar generates all palindromes over \{a, b, c\}. Here is a function for recognizing whether a string is generated by the grammar.

```haskell
recognizeA :: String -> Bool
recognizeA = \ xs ->
  null xs || xs == "a" || xs == "b" || xs == "c"
  || or [ recognizeA ys | ["a",ys,"a"] <- splitN 3 xs ]
  || or [ recognizeA ys | ["b",ys,"b"] <- splitN 3 xs ]
  || or [ recognizeA ys | ["c",ys,"c"] <- splitN 3 xs ]
```

This gives:

```
Parsing> recognizeA "aabcbbcbaa"
True
Parsing> recognizeA "aabcbbcba"
False
```

It should be clear that the above recipe for recognizer construction works for any set of CF rules. For a rewrite rule of the form \( V \rightarrow T_1 \cdots T_n \), the split function \( \text{splitN} \) should be used with argument \( n \), for splitting the input into \( n \) substrings.

From recognition to generation is a simple step. First, here are tools for generating all strings over a given alphabet:

```haskell
generate :: Int -> String -> [String]
generate 0 alphabet = ["""]
generate n alphabet = [ x:xs | x <- alphabet, xs <- generate (n-1) alphabet ]

gener :: Int -> String -> [String]
gener n alphabet = generate n alphabet ++ gener (n+1) alphabet

generAll :: String -> [String]
generAll alphabet = gener 0 alphabet
```

Combining this with the recognizer, we get a generation function for the language:
The function genA generates the infinite list of all palindromes over \{a, b, c\}. Here is a finite prefix of this list:

```
Parsing> take 15 genA
["","a","b","c","aa","bb","cc","aaa","aba","aca","bab","bbb","bcb","cac","cbc"]
```

This uses the predefined Haskell function \texttt{take}:

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) | n>0 = x : take (n-1) xs
take _ _ = error "Prelude.take: negative argument"
```

Parsing with a CF grammar is the process of constructing syntax trees. Let the types for the terminal symbols and the rewrite symbols of a grammar be given as \texttt{Tsym}, and \texttt{Rsym}, respectively:

```
type Tsym = String
    type Rsym = String
```

Then the data type of syntax trees can be defined as follows:

```
data SyntaxTree = Leaf Tsym | Tree Rsym [SyntaxTree] deriving Eq
```

An appropriate show function for this datatype:

```
instance Show SyntaxTree where
    show (Leaf t) = t
    show (Tree r ts) = "[" ++ r ++ " " ++ show ts ++ "]"
```
Here are the appropriate changes to `recognizeA` to create a parser for the example grammar:

```haskell
parseA :: String -> [SyntaxTree]
parseA = \ xs ->
  [Leaf "[]" | null xs ]
  ++
  [Leaf "a" | xs == "a" ]
  ++
  [Leaf "b" | xs == "b" ]
  ++
  [Leaf "c" | xs == "c" ]
  ++
  [Tree "A" [Leaf "a", tree, Leaf "a"] | ["a",ys,"a"] <- splitN 3 xs, 
    tree <- parseA ys ]
  ++
  [Tree "A" [Leaf "b", tree, Leaf "b"] | ["b",ys,"b"] <- splitN 3 xs, 
    tree <- parseA ys ]
  ++
  [Tree "A" [Leaf "c", tree, Leaf "c"] | ["c",ys,"c"] <- splitN 3 xs, 
    tree <- parseA ys ]
```

Again, it should be clear that this procedure works for any CF grammar.

Here is a demo:

```
Parsing> parseA "aabaa"
[[A [a,[A [a,b,a]],a]]
Parsing> parseA "aabbaa"
[[A [a,[A [a,[A [b,[]],b]],a]],a]]
Parsing> parseA "abbcbbcaaa"
Parsing> parseA "abbcbbcaaa"
[]
```

**Exercise 8.1** For CF grammars without $\epsilon$ productions, i.e., without rules of the form $V \longrightarrow \epsilon$, the functions for recognition and parsing can be made more efficient by suitable modifications of the functions `split2` and `splitN`. How?

**Exercise 8.2** Explain why the approach to parsing CF grammars with `split2` and `splitN` has no trouble with so-called left-recursive rules, i.e., rules of the form $A \longrightarrow AT_2 \cdots T_n$. 
8.2 Parsers and Parser Combinators

Let us try to take a more general perspective on the above. A parser for producing objects of type \( b \) from lists of objects of type \( a \) has the following general type:

\[
\text{type Parser} \ a \ b = [a] \rightarrow [(b, [a])] 
\]

The idea behind this is that the parser works in a recursive descent fashion, as follows. The parser scans a list of tokens of type \( a \) and tries to construct a parse object of type \( b \) from a prefix of the input list, leaving the remainder of the input list. The outcome of the parse process is the list of all pairs \((b, [a])\) that the parser can construct. Parse failure is indicated by return of \([]\).

The simplest possible parser is the parser for the empty input string. This parser always succeeds, and it does not consume any tokens.

\[
\text{epsilon :: Parser} \ a \ () \\
\text{epsilon} \ xs = [((), x)] 
\]

In the code for \( \text{epsilon} \) the empty string is represented as the so-called trivial type \((\)\). This type has only one member, also called \((\)\).

Another elementary parser that is a variant on the \( \text{epsilon} \) parser is the parser that also does not consume input but that succeeds with a given value:

\[
\text{succeed :: } b \rightarrow 
\text{Parser} \ a \ b \\
\text{succeed} \ r \ xs = [(r, x)] 
\]

Note that \( \text{epsilon} \) is equivalent to \( \text{succeed} () \).

Here is a parser for recognizing individual symbols:

\[
\text{symbol :: Eq} \ a \Rightarrow a \rightarrow 
\text{Parser} \ a \ a \\
\text{symbol} \ c \ [] = [] \\
\text{symbol} \ c \ (x:xs) \mid c == x = [(x, xs)] \\
\mid \text{otherwise} = [] 
\]
A combinator to check whether the first element of a list of elements satisfies a given property is `satisfy`:

```
satisfy :: (a -> Bool) -> Parser a a
satisfy p [] = []
satisfy p (x:xs) | p x = [(x,xs)]
                | otherwise = []
```

As an example, here is a parser that checks whether the first symbol of a string is a lowercase character. This uses the predefined function `isLower`.

```
lowerC :: Parser Char Char
lowerC = satisfy isLower
```

**Exercise 8.3** Show how the function `symbol` can be defined in terms of the function `satisfy`.

Recognition of a list of symbols (a token) is done by `token`. This uses the predefined functions `take` and `drop` for taking and dropping prefixes of a given length `n` from a list.

```
token :: Eq a => [a] -> Parser a [a]
token cs xs | cs == take n xs = [(cs,drop n xs)]
            | otherwise = []
            where n = length cs
```

With this we can define parsers like `token "begin"`, for parsing the keyword `begin`, and so on. We now turn to ways of combining or transforming parsers. For that, we use so-called parser combinators. Here is a parser combinator that takes a parser and returns a parser that does the same as the input parser, except for the fact that it also insists that the remainder string is empty:

```
just :: Parser a b -> Parser a b
just p = filter (null.snd) . p
```

Here is a declaration of an operator for sequential composition of parsers:
This operator composes a parser of type \texttt{Parser a b} with a parser of type \texttt{Parser a [b]} to a parser of type \texttt{Parser a [b]}, as follows:

\[
(\langle\ast\rangle) :: \texttt{Parser a b} \to \texttt{Parser a [b]} \to \texttt{Parser a [b]}
\]

\[
(p \langle\ast\rangle q) \; \texttt{xs} = [ (r:rs,zs) \mid (r,ys) \leftarrow p \; \texttt{xs},
\quad (rs,zs) \leftarrow q \; \texttt{ys} ]
\]

This can be used for collecting the results of a list of parsers operating one after the other:

\[
\texttt{collect} :: \texttt{[Parser a b]} \to \texttt{Parser a [b]}
\]

\[
\texttt{collect} \; [] = \texttt{succeed} \; []
\]

\[
\texttt{collect} \; (p:ps) = p \langle\ast\rangle \texttt{collect} \; ps
\]

The next operation we want to tackle is choice. Here is a so-called fixity declaration for a binary parser combinator that handles choice:

\[
\texttt{infixr 4 \langle|\rangle}
\]

The definition of choice uses list concatenation:

\[
(\langle|\rangle) :: \texttt{Parser a b} \to \texttt{Parser a b} \to \texttt{Parser a b}
\]

\[
(p1 \langle|\rangle p2) \; \texttt{xs} = p1 \; \texttt{xs} ++ p2 \; \texttt{xs}
\]

The operator \(\langle|\rangle\) should not be confused with the operator \(\mid\mid\) for boolean disjunction. The operators have quite different types.

The following parser combinator handles the ‘parse arbitrary many copies’ operator from EBNF. Recall that EBNF has extra meta-symbols \{ and \}, with the convention that \{A\} indicates that A may occur zero or more times. The parser combinator produces the longest match first.
many :: Parser a b -> Parser a [b]
many p =
p <*>> many p
<>
succeed []

It is often convenient to do postprocessing on the results of a parse. For that we introduce a new parser combinator <$>.

infixl 7 <$>  

This combinator takes a function and a parser as arguments, and applies the function to the result of the parse.

(<$>) :: (a -> b) -> Parser s a -> Parser s b
(f <$> prs) xs = [ (f x,ys) | (x,ys) <- prs xs ]

This can be used, e.g., to map the result of a parse to a suitable value. Here is a simple example:

digit :: Parser Char Int
digit = f <$> satisfy isDigit
    where f c = ord c - ord '0'

For further applications of postprocessing of parse results, we generalize the type of syntax trees, as follows:

data ParseTree a b = Eps | Lf a | Tr b [ParseTree a b] deriving Eq

Here Eps denotes the empty tree. Given show functions for the types a and b, we define a show function for ParseTree a b as follows:
We are interested in parsers that map symbols into parse trees. Here is the appropriate type:

\[
\text{type PARSER } a \ b = \text{Parser } a \ (\text{ParseTree } a \ b)
\]

The following parser maps the null input string to the parse tree \text{Eps}:

\[
\text{epsilonT :: PARSER } a \ b \\
\text{epsilonT = succeed Eps}
\]

The following parser recognizes a symbol and outputs an appropriate leaf parse tree.

\[
\text{symbolT :: Eq } a =\Rightarrow a \rightarrow \text{PARSER } a \ b \\
\text{symbolT } c = (\ x \rightarrow \text{Lf } x) \ <$> \text{symbol } c
\]

In order to use parser combinators for parsing CF rules, we need a parser combinator for handling the sequencing of symbols in the righthand side of a rule (under a given lefthand symbol), and a parser combinator for handling multiple rules with the same lefthand sides.

Here is a parser combinator for handling sequencing. The parser combinator combines a list of parsers into a single new parser, using a rewrite symbol (the lefthand side symbol of the rule) given by \text{b}.

\[
\text{parseAs :: } b \rightarrow [\text{PARSER } a \ b] \rightarrow \text{PARSER } a \ b \\
\text{parseAs } s \ ps = (\ x s \rightarrow \text{Tr } s \ xs) \ <$> \text{collect } ps
\]

CF grammars are built with choice and concatenation, so we have all the ingredients for specifying parsers for CF grammars. With choice and concatenation we can specify the following parser for the example grammar from the previous section:
palindr :: PARSER Char Char
palindr = epsilonT <|> symbolT 'a' <|> symbolT 'b' <|> symbolT 'c'
        <|> parseAs 'A' [symbolT 'a', palindr, symbolT 'a']
        <|> parseAs 'A' [symbolT 'b', palindr, symbolT 'b']
        <|> parseAs 'A' [symbolT 'c', palindr, symbolT 'c']

This gives:

Parsing> palindr "abba"
[([],"abba"),('a','bba'),(['A',['a',['A',['b',[],'b']],'a']),'""')]
Parsing> just palindr "abba"
[(['A',['a',['A',['b',[],'b']],'a']),'""')]

As a further example, here is a parser for the language of Mastermind from Chapter 1.

color, answer, guess, reaction, turn, game :: PARSER String String
color    = symbolT "red" <|> symbolT "yellow"
          <|> symbolT "blue" <|> symbolT "green"
answer   = symbolT "black" <|> symbolT "white" <|> symbolT "blank"
guess    = parseAs "GUESS" [color,color,color,color]
reaction = parseAs "REACTION" [answer,answer,answer,answer]
turn     = parseAs "TURN" [guess,reaction]
game     = turn <|> parseAs "GAME" [turn,game]

Exercise 8.4 Does the approach to parsing CF grammars with parser combinators, along the lines sketched in this section, work for all CF grammars? Hint: look at left-recursive rules, i.e., rules of the form \( A \rightarrow AT_2 \cdots T_n \).

8.3 Features and Feature Structures

Syntactic features are important in most syntactic formalisms for natural language analysis. We will develop a simple feature system for a small fragment of English.

There are two opposing views of feature agreement. According to the first view, the features of a syntactic component are computed in some way from the features of a ‘controlling’ component. Call this the derivational view on feature agreement. On this view there is a strict distinction between source and target of feature agreement. According to the second view, features of syntactic structures are partially specified by the constituents of the structure by means of a constraint handling mechanism. On this view, feature agreement has neither source nor target,
but is the result of certain compatibility checks between syntactic constituents. The second view is prevalent in so-called unification-based approaches to syntax. See, e.g., [PS87, PS94].

To implement the first view is easy in a functional approach: just define the feature set of a component in terms of a function from the feature set of the source of the agreement. To implement the second view, we need a tool that we already developed for theorem proving in predicate logic: unification. Instead of term unification we now need graph unification.

We will start with defining a number of useful features. A principled approach would be to distinguish various types of feature values, e.g., to put \texttt{Masc} and \texttt{Neutr} in type \texttt{Gender}, to put \texttt{Sg} and \texttt{Pl} in type \texttt{Number}, and so on. For present purposes it is more convenient, however, to put all feature values in the same data type. This will make the system easier to use and modify, at the cost of making feature coding errors slightly more difficult to detect (at least they will not be caught by the type checking).

```haskell
data Fvalue = Masc | Fem | Neutr
  | Sg | Pl
  | Refl | Pers | Wh
  | Fst | Snd | Thrd
  | Nom | Acc
  | Pres | Past
  | Fin | Inf | Perf | Pass
  | On | With | By | To | From
  deriving (Eq,Ord)
```

Make sure that feature values are shown in a reasonable format.

```haskell
instance Show Fvalue where
  show Masc = "M";
  show Fem = "F";
  show Neutr = "N"
  show Sg = "Sg";
  show Pl = "Pl"
  show Refl = "Refl";
  show Pers = "Pers";
  show Wh = "Wh"
  show Fst = "1";
  show Snd = "2";
  show Thrd = "3"
  show Nom = "nom";
  show Acc = "acc"
  show Pres = "pres";
  show Past = "past"
  show Fin = "fin";
  show Inf = "inf";
  show Perf = "perf"
  show Pass = "pass"
  show On = "on";
  show By = "by";
  show With = "with"
  show To = "to";
  show From = "from"
```

To ensure that features can be read in the same format, we will put them in the \texttt{Read} class (the class of Haskell types for which a \texttt{read} function is defined). The implementation uses a predefined function \texttt{lex} for lexical scanning of a string.
instance Read Fvalue where
  readsPrec p = \ str ->
    [(Masc,rest) | ("M", rest)  <- lex str ]
++ [(Fem,rest) | ("F", rest)  <- lex str ]
++ [(Pers,rest) | ("Pers", rest)  <- lex str ]
++ [(Refl,rest) | ("Refl", rest)  <- lex str ]
++ [(Wh,rest) | ("Wh", rest)  <- lex str ]
++ [(Neutr,rest) | ("N", rest)  <- lex str ]
++ [(Sg,rest) | ("Sg", rest)  <- lex str ]
++ [(Pl,rest) | ("Pl", rest)  <- lex str ]
++ [(Fst,rest) | ("i", rest)  <- lex str ]
++ [(Snd,rest) | ("2", rest)  <- lex str ]
++ [(Thrd,rest) | ("3", rest)  <- lex str ]
++ [(Nom,rest) | ("nom", rest)  <- lex str ]
++ [(Acc,rest) | ("acc", rest)  <- lex str ]
++ [(Pres,rest) | ("pres",rest)  <- lex str ]
++ [(Past,rest) | ("past",rest)  <- lex str ]
++ [(Fin,rest) | ("fin",rest)  <- lex str ]
++ [(Inf,rest) | ("inf",rest)  <- lex str ]
++ [(Perf,rest) | ("perf",rest)  <- lex str ]
++ [(Pass,rest) | ("pass",rest)  <- lex str ]
++ [(On,rest) | ("on", rest)  <- lex str ]
++ [(By,rest) | ("by", rest)  <- lex str ]
++ [(With,rest) | ("with",rest)  <- lex str ]
++ [(To,rest) | ("to", rest)  <- lex str ]
++ [(From,rest) | ("from", rest)  <- lex str ]

Here is a data type for feature labels.

    data Flabel = Gender | Number | Person | Tense
                  | Case | Prep | Ptype | Vform
    deriving (Eq,Ord,Show,Read)

From feature labels and feature values we can now build feature equations. These equations are
meant to be read disjunctively. E.g.,

    data FeatEq = Ft Flabel [Fvalue] deriving (Eq,Ord)
Put feature equations in the Show and Read classes:

```haskell
instance Show FeatEq where
    show (Ft label values) = show label ++ show values

instance Read FeatEq where
    readsPrec p = \ str ->
        [(Ft label values, rest) |
            (label,str0) <- reads str :: [(Flabel,String)],
            (values,rest) <- reads str0 :: [[Fvalue,String]] ]
```

Some useful conversion functions:

```haskell
labelF :: FeatEq -> Flabel
labelF (Ft label _) = label

valuesF :: FeatEq -> [Fvalue]
valuesF (Ft _ values) = values
```

The following `makeF` function creates a feature equation from a feature label and a list of feature values on condition that the list of values is non-empty, with failure indicated by `[]`.

```haskell
makeF :: Flabel -> [Fvalue] -> [FeatEq]
makeF label vs = [Ft label vs | not (null vs) ]
```

Labels for feature value structures:

```haskell
data FVlabel = Subject | Object
    deriving (Eq,Ord,Show,Read)
```
A single feature value structure is either a list of feature equations or a feature value label pointing to a list of feature value structures. We put feature value structures in the class `Ord` because we need to be able to sort them, i.e., present them in a canonical order.

```haskell
data FV = Has FeatEq
         | At FVlabel [FV] deriving (Eq,Ord)
```

Put feature structures in the `Show` class:

```haskell
instance Show FV
  where
    show (Has fe) = show fe
    show (At label fvs) = show label ++ show fvs
```

Put feature structures in the `Read` class:

```haskell
instance Read FV where
  readsPrec p = \ str ->
    [(Has fe, rest) |
      (fe, rest) <- reads str :: [(FeatEq,String)]]
    ++
    [(At label fss, rest) |
      (label, str0) <- (reads str) :: [(FVlabel,String)],
      (fss, rest) <- (reads str0) :: [(FV,String)]]
```

Sorting of feature structures and lists of feature structures:

```haskell
srt :: FV -> FV
srt (Has (Ft label values)) = Has (Ft label (sort values))
srt (At label fvs) = At label (srts fvs)
srts :: [FV] -> [FV]
srts = sort . (map srt)
```

Examples of feature structures:
fs1 = [At Subject [Has (Ft Case [Nom]), Has (Ft Person [Thrd]),
          Has (Ft Number [Pl])],
       Has (Ft Case [Acc])]
f2 = [At Subject [Has (Ft Tense [Past])]]
f3 = [At Subject [Has (Ft Tense [Pres])]]

We can also read them in a more convenient format:

fs4 = (srt . read) "[Subject [Case [nom], Person [3], Number [Pl]], Case [acc]]"
fs5 = (srt . read) "[Case [acc]]"
fs6 = (srt . read) "Gender []"
fs7 = (srt . read) "Gender [M,F]"
fs8 = (srt . read) "Subject [Case [nom], Person [3], Number [Pl]]"
fs9 = (srts . read) "[Gender [M]]"

Unification of feature structure lists is different from unification of term lists: missing information from one feature structure list is copied from the other feature structure list. Only in case of a feature clash at a node does unification fail. The unification algorithm makes essential use of the fact that the feature structure lists are in canonical order. If unification succeeds it produces a unit list containing the resulting feature structure list, again in canonical order. Failure of unification is indicated by [].
8.4 CATEGORIES

We will now generalize context free rules by replacing the rewrite symbols by structured symbols (categories), employing the category labels defined in the lexical database.

Categories are data structures containing a category label, a string (for the phonological content of the category), a feature structure list, a subcategorization list (a list of categories) and a constituent list (also a list of categories). The subcategorization list gives the list of categories that it needs to combine with, the constituent list the list of categories that it has combined with. The phonological content string will be used only for the lexical categories; it will be set to the empty string for all complex categories.

```
data Cat = Ct CatLabel String [FV] [Cat] [Cat] deriving Eq
```

We will use "_" as a variable and null default over phonological strings.
instance Show Cat where
  show (Ct label phon fss subc const) =
    show label ++ " " ++ phon ++ " " ++ show fss
    ++ show subc ++ show const ++ "\n"

The following will enable us to read in categories from strings. We assume the constituent lists are empty in the lexicon.

instance Read Cat where
  readsPrec p = \ str ->
    [(Ct label phon (srts fss) subcat [], rest) |
      (label,str0) <- reads str :: [(CatLabel,String)],
      (phon, str1) <- lex str0,
      (fss, str2) <- reads str1 :: [[FV],String],
      (subcat,rest) <- reads str2 :: [[[Cat],String]]
    ]

Getting the category label, the phonological string, the feature structures list, the subcatlist and the constituent list from a category:

labelC :: Cat -> CatLabel
labelC (Ct label _ _ _ _) = label

phonC :: Cat -> String
phonC (Ct _ phon _ _ _) = phon

fvC :: Cat -> [FV]
fvC (Ct _ _ fvs _ _) = fvs

subcatC :: Cat -> [Cat]
subcatC (Ct _ _ cats _) = cats

constituentsC :: Cat -> [Cat]
constituentsC (Ct _ _ _ cats) = cats

Assigning a feature structure to a category, with success indicated by a unit list and failure by [].
The function `unifCatsFVs` unifies the feature value lists of two categories.

\[
\text{unifCatsFVs} :: \text{Cat} \to \text{Cat} \to \[
\text{[FV]}\]
\text{unifCatsFVs} \text{ cat1 cat2} = \text{unifFV} (\text{fvC cat1}) (\text{fvC cat2})
\]

The function `agreeC` checks whether there is a feature clash or not when two categories are combined.

\[
\text{agreeC} :: \text{Cat} \to \text{Cat} \to \text{Bool}
\text{agreeC} \text{ cat1 cat2} = \text{unifCatsFVs} \text{ cat1 cat2} /= []
\]

Unification of phonological strings, taking "_" as a variable:

\[
\text{unifPhon} :: \text{String} \to \text{String} \to \[
\text{String}\]
\text{unifPhon} "_" \text{ phon} = \text{[phon]}
\text{unifPhon} \text{ phon} "_" = \text{[phon]}
\text{unifPhon} \text{ ph1 ph2} \mid \text{ph1 == ph2} = \text{[ph1]}
\mid \text{otherwise} = []
\]

Unification of categories and lists of categories.

\[
\text{unifyCats} :: \text{Cat} \to \text{Cat} \to \[
\text{Cat}\]
\text{unifyCats} (\text{Ct label1 phon1 fvs1 subcat1 const1})
(\text{Ct label2 phon2 fvs2 subcat2 const2})
= \text{[ Ct label1 phon fvs subcat const |}
\quad \text{label1 == label2},
\quad \text{phon} \leftarrow \text{unifPhon phon1 phon2},
\quad \text{fvs} \leftarrow \text{unifFV fvs1 fvs2},
\quad \text{subcat} \leftarrow \text{unifyCatLists subcat1 subcat2},
\quad \text{const} \leftarrow \text{unifyCatLists const1 const2 } ]
\]

Unification of lists of categories: unification of a non-empty list with an empty list succeeds.

```
unifyCatLists :: [Cat] -> [Cat] -> [[Cat]]
unifyCatLists [] [] = [[]]
unifyCatLists cats [] = [cats]
unifyCatLists [] cats = [cats]
unifyCatLists (cat1:cats1) (cat2:cats2) = [ cat:rest | cat <- unifyCats cat1 cat2,
                                      rest <- unifyCatLists cats1 cats2 ]
```

Unification of category lists will be used below in subcategorization checks.

### 8.5 Phrase Structure Grammar with Structured Symbols

We will implement a parser based on the following phrase structure grammar.

```
S    ->  DP VP
DP    ->  Name | Pers | Refl | DET CN
PP    ->  Prep DP
DPorPP ->  DP | PP
DET    ->  every | some | a | no | the | ...
\(\vee\)P ->  Aux VP-ts \{DPorPP\}
\(\wedge\)P ->  VP+ts \{DPorPP\}
Aux    ->  did | didn’t | was | ...
```

The categories \textit{Pers} and \textit{Refl} are for personal and reflexive pronouns. In the rules for \(\wedge\)P, \{DPorPP\} denotes an empty or non-empty list of DPs or PPs. We will use feature agreement to get basic facts about case right, such as \textit{I broke the vase} and \textit{Max shaved me} versus *\textit{Me broke the vase} and *\textit{Max shaved I}. We will also employ the feature system for subcategorization checks on the DP or PP complement lists of verbs. In Section 8.9 we will add relative clauses, yes/no questions and Wh-questions to this fragment.

### 8.6 Lexical Database

This section lists the lexical database.
module LexDB

where

The lexical database is structured by word category. We employ the following category labels.

data CatLabel = Sent | YN | WH
  | DP | DET | VERB | VP | VPBAR | CN
  | PREP | PP | AUX | REL | COMP
  deriving (Eq,Show,Read)

A lexical database is a mapping from category labels to lists of strings:

type DB = CatLabel -> [[String]]

Declare \texttt{db} as a database. Then \texttt{db DP} lists the DPs, \texttt{DET} the determiners, and so on.

db :: DB

DPs in the lexicon: personal pronouns.
DPs in the lexicon: reflexive pronouns.

["myself", "DP myself "
  ++ " [Ptype[Ref1],Number[Sg],Person[1],Case[acc]] []"]
,""ourselves", "DP ourselves "
  ++ " [Ptype[Ref1],Number[Pl],Person[1],Case[acc]] []"]
,""yourself", "DP yourself "
  ++ " [Ptype[Ref1],Number[Sg],Person[2],Case[acc]] []"]
,""yourselves", "DP yourselves "
  ++ " [Ptype[Ref1],Number[Pl],Person[2],Case[acc]] []"]
,""himself", "DP himself "
  ++ " [Ptype[Ref1],Gender[M],Number[Sg],Person[3],Case[acc]] []"]
,""herself", "DP herself "
  ++ " [Ptype[Ref1],Gender[F],Number[Sg],Person[3],Case[acc]] []"]
,""itself", "DP itself "
  ++ " [Ptype[Ref1],Gender[N],Number[Sg],Person[3],Case[acc]] []"]
,""themselves", "DP themselves "
  ++ " [Ptype[Ref1],Number[Pl],Person[3],Case[acc]] []"]
],

DPs in the lexicon: WH phrases.

["myself", "DP myself "
  ++ " [Ptype[Ref1],Number[Sg],Person[1],Case[acc]] []"]
,""ourselves", "DP ourselves "
  ++ " [Ptype[Ref1],Number[Pl],Person[1],Case[acc]] []"]
,""yourself", "DP yourself "
  ++ " [Ptype[Ref1],Number[Sg],Person[2],Case[acc]] []"]
,""yourselves", "DP yourselves "
  ++ " [Ptype[Ref1],Number[Pl],Person[2],Case[acc]] []"]
,""himself", "DP himself "
  ++ " [Ptype[Ref1],Gender[M],Number[Sg],Person[3],Case[acc]] []"]
,""herself", "DP herself "
  ++ " [Ptype[Ref1],Gender[F],Number[Sg],Person[3],Case[acc]] []"]
,""itself", "DP itself "
  ++ " [Ptype[Ref1],Gender[N],Number[Sg],Person[3],Case[acc]] []"]
,""themselves", "DP themselves "
  ++ " [Ptype[Ref1],Number[Pl],Person[3],Case[acc]] []"]
],
8.6. LEXICAL DATABASE

DPs in the lexicon: proper names.

```
[
    "who",
    "DP who [Ptype[Wh],Gender[M,F],Person[3]] []",
    "whom",
    "DP whom [Ptype[Wh],Case[acc],Gender[M,F],Number[Sg],Person[3]] []",
    "what",
    "DP what [Ptype[Wh],Gender[N],Number[Sg],Person[3]] []",
]
```

Verb phrases in the lexicon.

The subcat list gives the DP and PP complements, and their expected order. We use _ as dummy for the phonological components of the subcategorized categories.

The lexical entries reveal a lot of redundancy, thus suggesting the need for a morphological component that can generate the appropriate verb entries from succinct descriptions of the verb morphology and subcategorization patterns.

The verb buy.
The verb *break*. 

```plaintext
db VERB = [
  "buys",
  "VERB buys [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [] ] ",
  "VERB buys [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[from]] [] ]",
  "buy",
  "VERB buy [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [] ] ",
  "VERB buy [Number[Pl],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [] ] ",
  "VERB buy [Vform[inf]] "
  ++ " [DP _ [Case[acc]] [] ] ",
  "VERB buy [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[from]] [] ] ",
  "VERB buy [Number[Pl],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[from]] [] ] ",
  "VERB buy [Vform[inf]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[from]] [] ] ",
  "bought",
  "VERB bought [Tense[past],Vform[fin]] [DP _ [Case[acc]] [] ] ",
  "VERB bought [Vform[perf]] [DP _ [Case[acc]] [] ] ",
  "VERB bought [Tense[past],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[from]] [] ] ",
  "VERB bought [Vform[perf]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[from]] [] ] ",
  "VERB bought [Vform[pass]] "
  ++ " [PP _ [Prep[by]] [] ]",
  "VERB bought [Vform[pass]] "
  ++ " [PP _ [Prep[by]] [],PP _ [Prep[from]] [] ]",
  "VERB bought [Vform[pass]] "
  ++ " [PP _ [Prep[from]] [] ] "
],
```

The verb *break*. 

```plaintext
The verb *die*.

The verb *drop*. 

---

"breaks",
"VERB breaks [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [] ]",
"VERB breaks [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[with]] [] ]",
"break",
"VERB break [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [] ]",
"VERB break [Number[Pl],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [] ]",
"VERB break [Vform[inf]] "
  ++ " [DP _ [Case[acc]] [] ]",
"VERB break [Tense[past],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[with]] [] ]",
"broke",
"VERB broke [Tense[past],Vform[fin]] [DP _ [Case[acc]] [] ]",
"VERB broke [Tense[past],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[with]] [] ]",
"broken",
"VERB broken [Vform[perf]] [DP _ [Case[acc]] [] ]",
"VERB broken [Vform[perf]] [DP _ [Case[acc]] [],PP _ [Prep[with]] [] ]",
"VERB broken [Vform[pass]] []",
"VERB broken [Vform[pass]] [PP _ [Prep[by]] [] ]",
"VERB broken [Vform[pass]] [PP _ [Prep[with]] [] ]",
"VERB broken [Vform[pass]] [PP _ [Prep[by]] [],PP _ [Prep[with]] [] ]",
"dies",
"VERB dies [Person[3],Number[Sg],Tense[pres],Vform[fin]] []",
"die",
"VERB die [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] []",
"VERB die [Number[Pl],Tense[pres],Vform[fin]] []",
"VERB die [Vform[inf]] []",
"died",
"VERB died [Tense[past],Vform[fin]] []",
"VERB died [Vform[perf]] []

---

The verb *die*.

The verb *drop*. 

---

"breaks",
"VERB breaks [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [] ]",
"VERB breaks [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[with]] [] ]",
"break",
"VERB break [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [] ]",
"VERB break [Number[Pl],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [] ]",
"VERB break [Vform[inf]] "
  ++ " [DP _ [Case[acc]] [] ]",
"VERB break [Tense[past],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[with]] [] ]",
"broke",
"VERB broke [Tense[past],Vform[fin]] [DP _ [Case[acc]] [] ]",
"VERB broke [Tense[past],Vform[fin]] "
  ++ " [DP _ [Case[acc]] [],PP _ [Prep[with]] [] ]",
"broken",
"VERB broken [Vform[perf]] [DP _ [Case[acc]] [] ]",
"VERB broken [Vform[perf]] [DP _ [Case[acc]] [],PP _ [Prep[with]] [] ]",
"VERB broken [Vform[pass]] []",
"VERB broken [Vform[pass]] [PP _ [Prep[by]] [] ]",
"VERB broken [Vform[pass]] [PP _ [Prep[with]] [] ]",
"VERB broken [Vform[pass]] [PP _ [Prep[by]] [],PP _ [Prep[with]] [] ]",
"dies",
"VERB dies [Person[3],Number[Sg],Tense[pres],Vform[fin]] []",
"die",
"VERB die [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] []",
"VERB die [Number[Pl],Tense[pres],Vform[fin]] []",
"VERB die [Vform[inf]] []",
"died",
"VERB died [Tense[past],Vform[fin]] []",
"VERB died [Vform[perf]] []

---

The verb *die*.

The verb *drop*. 

---
The verb *give*. 

```
["drops",
 "VERB drops  [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
 ++    " [DP _ [Case[acc]] [] ] "]
,"drop",
 "VERB drop  [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] "
 ++    " [DP _ [Case[acc]] [] ] ",
 "VERB drop  [Number[Pl],Tense[pres],Vform[fin]] "
 ++    " [DP _ [Case[acc]] [] ] ",
 "VERB drop  [Vform[inf]] "
 ++    " [DP _ [Case[acc]] [] ] "],
["dropped",
 "VERB dropped [Tense[past],Vform[fin]] [DP _ [Case[acc]] [] ] ",
 "VERB dropped [Vform[perf]] [DP _ [Case[acc]] [] ] "]
```
The verb **hate**.
The verb *kill*. 

```plaintext
["hates",
 "VERB hates [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
   ++ " [DP _ [Case[acc]] [] ] ",
["hate",
 "VERB hate [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] "
   ++ " [DP _ [Case[acc]] [] ] ",
 "VERB hate [Number[Pl],Tense[pres],Vform[fin]] "
   ++ " [DP _ [Case[acc]] [] ] ",
 "VERB hate [Vform[inf]] "
   ++ " [DP _ [Case[acc]] [] ] "],
["hated",
 "VERB hated [Tense[past],Vform[fin]] [DP _ [Case[acc]] [] ] ",
 "VERB hated [Vform[perf]] [DP _ [Case[acc]] [] ] ",
 "VERB hated [Vform[pass]] [] ",
 "VERB hated [Vform[pass]] [PP _ [Prep[by]] [] ]"]
```
The verb *laugh*. 

```plaintext
"kills",
  "VERB kills [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
    ++ " [DP_ [Case[acc]] [] ] ",
  "VERB kills [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
    ++ " [DP_ [Case[acc]] [],PP_ [Prep[with]] [] ] ",
["kill",
  "VERB kill [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] "
    ++ " [DP_ [Case[acc]] [] ] ",
  "VERB kill [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] "
    ++ " [DP_ [Case[acc]] [],PP_ [Prep[with]] [] ] ",
  "VERB kill [Number[Pl],Tense[pres],Vform[fin]] "
    ++ " [DP_ [Case[acc]] [] ] ",
  "VERB kill [Number[Pl],Tense[pres],Vform[fin]] "
    ++ " [DP_ [Case[acc]] [],PP_ [Prep[with]] [] ] ",
  "VERB kill [Vform[inf]] "
    ++ " [DP_ [Case[acc]] [] ] ",
  "VERB kill [Vform[inf]] "
    ++ " [DP_ [Case[acc]] [],PP_ [Prep[with]] [] ] ",
["killed",
  "VERB killed [Tense[past],Vform[fin]] "
    ++ " [DP_ [Case[acc]] [] ] ",
  "VERB killed [Tense[past],Vform[fin]] "
    ++ " [DP_ [Case[acc]] [],PP_ [Prep[with]] [] ] ",
  "VERB killed [Vform[perf]] [DP_ [Case[acc]] [] ] ",
  "VERB killed [Vform[perf]] "
    ++ " [DP_ [Case[acc]] [],PP_ [Prep[with]] [] ] ",
  "VERB killed [Vform[pass]] "
    ++ " [PP_ [Prep[by]] [],PP_ [Prep[with]] [] ] ",
]"
```
The verb *love*.

Exercise 8.5 As you can see, the redundancy is enormous. Write a tool for generating verb entries for an ‘exemplar’ lexicon database. The main function should have the following type:

```
verbEntry :: (String,String,String) -> (String,String,String) -> db VERB -> db VERB
```

A call `verbEntry (sell,sold,sold) (give,gave,given)` entries should generate the appropriate verb entries for sell on the basis of the existing entries for give, in verb database entries. The result should be a new verb database, with just the entries for sell.
For the rest of the verb database we make no attempt to be exhaustive.

The verb *open*.

```
["opened", "VERB open [Tense[past]] "
  ++ " [DP _ [Case[acc]] []",
  "VERB open [Tense[past]] "
  ++ " [DP _ [Case[acc]] [], PP _ [Prep[with]] []"]",
  "VERB open [] []"],
```

The verb *receive*.

```
["received", "VERB receive [Tense[past]] "
  ++ " [DP _ [Case[acc]] []",
  "VERB receive [Tense[past]] "
  ++ " [DP _ [Case[acc]] [], PP _ [Prep[from]] []"]",
  "VERB receive [] []"]
```

The verb *respect*.

```
["respects",
  "VERB respects [Person[3],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] []"]",
["respect",
  "VERB respect [Person[1,2],Number[Sg],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] []"]",
  "VERB respect [Number[Pl],Tense[pres],Vform[fin]] "
  ++ " [DP _ [Case[acc]] []"]",
  "VERB respect [Vform[inf]] "
  ++ " [DP _ [Case[acc]] []"]",
["respected",
  "VERB respected [Tense[past],Vform[fin]] [DP _ [Case[acc]] []] "
  "VERB respected [Vform[perf]] [DP _ [Case[acc]] []]",
  "VERB respected [Vform[pass]] []",
  "VERB respected [Vform[pass]] [PP _ [Prep[by]] []]"]
```

The verb *shave*. 


The verb *smile*.

```
["smiled", "VERB smiled [Tense[past],Vform[fin]] []", "VERB smiled [Vform[perf]] [] "]
```

The verb *sell*.

```
["sold", "VERB sold [Tense[past]] "
++ " [DP _ [Case[acc]] []]",
  "VERB sold [Tense[past]] "
++ " [DP _ [Case[acc]] [],PP _ [Prep[to]] []]",
  "VERB sold [] [] " "]
```

The verb *wash*.

```
["washed", "VERB washed [Tense[past]] "
++ " [DP _ [Case[acc]] []]",
  "VERB washed [] [] "]
```

DETs in the lexicon:
8.6. LEXICAL DATABASE

```
db DET = [
    [<"every"],  "DET every [Number[Sg]] []",
    [<"all"],  "DET all [Number[Pl]] []",
    [<"some"],  "DET some [Number[Sg]] []",
    [<"several"],  "DET several [Number[Pl]] []",
    [<"a"],  "DET a [Number[Sg]] []",
    [<"no"],  "DET no [Number[Sg]] []",
    [<"the"],  "DET the [Number[Sg]] []",
    [<"most"],  "DET most [Number[Pl]] []",
    [<"many"],  "DET many [Number[Pl]] []",
    [<"this"],  "DET this [Number[Sg]] []",
    [<"that"],  "DET that [Number[Sg]] []",
    [<"these"],  "DET these [Number[Pl]] []",
    [<"those"],  "DET those [Number[Pl]] []",
    [<"which"],  "DET which [Ptype[Wh]] []"]
]
```

CNs in the lexicon:

```
db CN = [
    [<"boy"],  "CN boy [Gender[M],Number[Sg],Person[3]] []",
    [<"boys"],  "CN boys [Gender[M],Number[Pl],Person[3]] []",
    [<"book"],  "CN book [Gender[N],Number[Sg],Person[3]] []",
    [<"books"],  "CN books [Gender[N],Number[Pl],Person[3]] []",
    [<"gun"],  "CN gun [Gender[N],Number[Sg],Person[3]] []",
    [<"guns"],  "CN guns [Gender[N],Number[Pl],Person[3]] []",
    [<"house"],  "CN house [Gender[N],Number[Sg],Person[3]] []",
    [<"houses"],  "CN houses [Gender[N],Number[Pl],Person[3]] []",
    [<"leaf"],  "CN leaf [Gender[N],Number[Sg],Person[3]] []",
    [<"leaves"],  "CN leaves [Gender[N],Number[Pl],Person[3]] []",
    [<"man"],  "CN man [Gender[M],Number[Sg],Person[3]] []",
    [<"men"],  "CN men [Gender[M],Number[Pl],Person[3]] []"]
]```
<table>
<thead>
<tr>
<th>Word</th>
<th>Category</th>
<th>Gender</th>
<th>Number</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>mouse</td>
<td>CN</td>
<td>N</td>
<td>Sg</td>
<td>3</td>
</tr>
<tr>
<td>mice</td>
<td>CN</td>
<td>N</td>
<td>Pl</td>
<td>3</td>
</tr>
<tr>
<td>person</td>
<td>CN</td>
<td></td>
<td>Sg</td>
<td>3</td>
</tr>
<tr>
<td>persons</td>
<td>CN</td>
<td></td>
<td>Pl</td>
<td>3</td>
</tr>
<tr>
<td>stone</td>
<td>CN</td>
<td>N</td>
<td>Sg</td>
<td>3</td>
</tr>
<tr>
<td>stones</td>
<td>CN</td>
<td>N</td>
<td>Pl</td>
<td>3</td>
</tr>
<tr>
<td>thing</td>
<td>CN</td>
<td>N</td>
<td>Sg</td>
<td>3</td>
</tr>
<tr>
<td>things</td>
<td>CN</td>
<td>N</td>
<td>Pl</td>
<td>3</td>
</tr>
<tr>
<td>tree</td>
<td>CN</td>
<td>N</td>
<td>Sg</td>
<td>3</td>
</tr>
<tr>
<td>trees</td>
<td>CN</td>
<td>N</td>
<td>Pl</td>
<td>3</td>
</tr>
<tr>
<td>vase</td>
<td>CN</td>
<td>N</td>
<td>Sg</td>
<td>3</td>
</tr>
<tr>
<td>vases</td>
<td>CN</td>
<td>N</td>
<td>Pl</td>
<td>3</td>
</tr>
<tr>
<td>window</td>
<td>CN</td>
<td>N</td>
<td>Sg</td>
<td>3</td>
</tr>
<tr>
<td>windows</td>
<td>CN</td>
<td>N</td>
<td>Pl</td>
<td>3</td>
</tr>
<tr>
<td>woman</td>
<td>CN</td>
<td>F</td>
<td>Sg</td>
<td>3</td>
</tr>
<tr>
<td>women</td>
<td>CN</td>
<td>F</td>
<td>Pl</td>
<td>3</td>
</tr>
</tbody>
</table>

Auxiliaries in the lexicon:
8.6. LEXICAL DATABASE

```plaintext
[db AUX = [
    "do", "AUX do [Person[1,2],Tense[pres]] "
    ++ " [VP _ [Vform[inf]] []]",
    "AUX do [Person[3],Number[Pl],Tense[pres]] "
    ++ " [VP _ [Vform[inf]] []]",
    "don't", "AUX don't [Person[1,2],Number[Sg],Tense[pres]] "
    ++ " [VP _ [Vform[inf]] []]",
    "AUX don't [Number[Pl],Tense[pres]] "
    ++ " [VP _ [Vform[inf]] []]",
    "does", "AUX does [Person[3],Number[Sg],Tense[pres]] "
    ++ " [VP _ [Vform[inf]] []]",
    "doesn't", "AUX doesn't [Person[3],Number[Sg],Tense[pres]] "
    ++ " [VP _ [Vform[inf]] []]",
    "did", "AUX did [Tense[past]] "
    ++ " [VP _ [Vform[inf]] []]",
    "didn’t", "AUX didn’t [Tense[past]] "
    ++ " [VP _ [Vform[inf]] []]",
    "am", "AUX am [Person[1],Number[Sg],Tense[pres]] "
    ++ " [VP _ [Vform[pass]] []]",
    "is", "AUX is [Person[3],Number[Sg],Tense[pres]] "
    ++ " [VP _ [Vform[pass]] []]",
    "are", "AUX are [Person[2],Number[Sg],Tense[pres]] "
    ++ " [VP _ [Vform[pass]] []]",
    "were", "AUX were [Person[2],Number[Sg],Tense[past]] "
    ++ " [VP _ [Vform[pass]] []]",
    "was", "AUX was [Person[1,3],Number[Sg],Tense[past]] "
    ++ " [VP _ [Vform[pass]] []]",
    "has", "AUX has [Person[3],Number[Sg],Tense[pres]] "
    ++ " [VP _ [Vform[perf]] []]",
    "have", "AUX have [Person[1],Number[Sg],Tense[pres]] "
    ++ " [VP _ [Vform[perf]] []]",
    "had", "AUX had [Tense[past]] "
    ++ " [VP _ [Vform[perf]] []]
]
```

Prepositions in the lexicon. Note that the PREP entries assign accusative case to the DPs on their subcat lists.
8.7 Scanning and Lexical Lookup

The `scan` function scans an input string and puts whitespace in front of punctuation marks.

```haskell
scan :: String -> String
scan [] = []
scan (x:xs) | x == '.' || x == '?' = ' ':x:scan xs
            | otherwise = x:scan xs
```

The lexer will produce lists of words (strings). Here is the datatype for such lists:

```haskell
type Words = [String]
```

The lexer creates such lists of words from the input string. For robustness, we convert everything to lowercase. The `lexer` function uses the predefined function `words` to split the input into separate words. The function `preprocess` is defined below.
8.7. SCANNING AND LEXICAL LOOKUP

lexer :: String -> Words
lexer = preprocess . words . (map toLower) . scan

It is useful to do some preprocessing on the list of input words. We assume that "." or "?" marks the end of the input, otherwise an error is generated.

preprocess :: Words -> Words
preprocess [] = error "\. or \? missing at the end."
preprocess [".""] = []
preprocess ["?""] = []
preprocess ("do":"not":xs) = "don't": preprocess xs
preprocess ("does":"not":xs) = "doesn't": preprocess xs
preprocess ("did":"not":xs) = "didn't": preprocess xs
preprocess ("nothing":xs) = "no":"thing": preprocess xs
preprocess ("nothing":xs) = "no":"person": preprocess xs
preprocess ("something":xs) = "some":"thing": preprocess xs
preprocess ("somebody":xs) = "some":"person": preprocess xs
preprocess ("everything":xs) = "every":"thing": preprocess xs
preprocess ("everybody":xs) = "every":"person": preprocess xs
preprocess ("less":"than":xs) = "less_than": preprocess xs
preprocess ("more":"than":xs) = "more_than": preprocess xs
preprocess ("at":"least":xs) = "at_least": preprocess xs
preprocess ("at":"most":xs) = "at_most": preprocess xs
preprocess (x:xs) = x : preprocess xs

The lexicon consists of a lexical database of type DB. The following lookup function fetches (word,category) pairs from a lexical database. The category labels employed in the lexicon are DP, VERB, DET, CN, AUX, PREP and REL.

lexListing :: DB -> [(String, Cat)]
lexListing = \ db -> [(word, cat) | label <- [DP, VERB, DET, CN, AUX, PREP, REL], entry <- db label, word <- [head entry], ct <- tail entry, cat <- [read ct] ]

The lookupWord function maps words to lists of categories from the lexicon.
CHAPTER 8. PARSING

lookupWord :: DB -> String -> [Cat]
lookupWord dbase word =
    [ ct | (wrd,ct) <- lexListing dbase, wrd == word ]

An error is generated on encounter of a word that is not in the lexical database.

collectCats :: DB -> Words -> [[Cat]]
collectCats db words =
    let
        listing = map (\ x -> (x,lookupWord db x)) words
        unknown = map fst (filter (null.snd) listing)
    in
        if unknown /= [] then error ("unknown words: " ++ show unknown)
        else initCats listing

Initialisation collects all the different cat-lists that are compatible with the categories the lexicon assigns to the words from a given word list.

initCats :: [(String,[Cat])] -> [[Cat]]
initCats [] = [[]]
initCats ((wrd,cats):pairs) =
    [ cat:rest | cat <- cats, rest <- initCats pairs ]

8.8 Parsing Categories

A parser for producing categories from lists of categories has the type Parser Cat Cat.
The function foundCat is a map from category labels to parsers. The function just checks if the first category on the category list has the appropriate label.

foundCat :: CatLabel -> Parser Cat Cat
foundCat label [] = []
foundCat label (cat:cats) = [ (cat,cats) | labelC cat == label ]
8.8. PARSING CATEGORIES

DPs

DPs are either found in the lexicon, or are the result of combining a DET and a CN. Here is the rule for complex DPs:

```
dpRule :: Parser Cat Cat
dpRule = \xs -> [(Ct DP "_" fvs [] [det,cn], zs) |
                   (det,ys) <- parseDET xs,
                   (cn, zs) <- parseCN ys,
                   fvs <- unifCatsFVs det cn ]
```

Parsing a DP is done by either finding it on the input stream or constructing it with the DP rule:

```
parseDP :: Parser Cat Cat
parseDP = foundCat DP <|> dpRule
```

PPs

PPs are parsed by combining a PREP with an appropriate DP. Here is the rule:

```
ppRule :: Parser Cat Cat
ppRule = \xs -> [(Ct PP "_" fvs [] [prep,dp], zs) |
                   (prep,ys) <- parsePrep xs,
                   (dp,zs) <- parseDP ys,
                   fvs <- unifCatsFVs prep dp ]
```

Although there happen to be no PPs in the lexicon, in the parser for PPs we include a call to foundCat PP. This is to avoid making the syntax dependent on the particular lexicon we have loaded.

```
parsePP :: Parser Cat Cat
parsePP = foundCat PP <|> ppRule
```
Determiners

All determiners are lexical:

\[
\text{parseDET} = \text{foundCat DET}
\]

Exercise 8.6 Add a rule for complex determiners consisting of a predeterminer and a numeral. This should cater for less than N, more than N, at least N, at most N, and exactly N. Pre-processing makes all predeterminers available as single words. You will also need a parser for numerals, of course.

CNs

The rule for parsing CNs as a lexical CN plus a relative clause goes like this:

CNs are lexical, or consist of a lexical CN plus a relative clause. The rule for parsing relative clauses will be discussed in Section 8.9. For now, we treat all CNs as lexical:

\[
\text{parseCN} :: \text{Parser Cat Cat} \\
\text{parseCN} = \text{foundCat CN}
\]

DPs or PPs

\[
\text{parseDPorPP} :: \text{Parser Cat Cat} \\
\text{parseDPorPP} = \text{parseDP} \; \text{|} \; \text{parsePP}
\]

Lists of DPs and PPs

We look for lists of DPs and PPs that match a given subcategorization list of a verb.

\[
\text{parseDPsorPPs} :: \text{Parser Cat [Cat]} \\
\text{parseDPsorPPs} = \text{many parseDPorPP}
\]
8.8. PARSING CATEGORIES

Preps
All prepositions are lexical:

```haskell
parsePrep :: Parser Cat Cat
parsePrep = foundCat PREP
```

Auxiliaries
All AUX entries are lexical:

```haskell
parseAUX = foundCat AUX
```

VPbars
In a V\textsuperscript{P} without auxiliary, we assign finite form to the VP. Here is the appropriate FV:

```haskell
finiteForm :: FV
finiteForm = Has (Ft Vform [Fin])
```

Two rules for VPbars:

```haskell
finiteVpRule :: Parser Cat Cat
finiteVpRule = \xs \rightarrow [ (Ct VPBAR "_" (fvC vp') []) [vp], ys ) |
    (vp, ys) <- parseVP xs,
    vp' <- assignFV finiteForm vp ]

auxVpRule :: Parser Cat Cat
auxVpRule = \xs \rightarrow
    [ (Ct VPBAR "_" fva []) (aux : vp'), zs ) |
    (aux@(Ct AUX phon fva subcataux []),ys) <- parseAUX xs,
    (vp,zs) <- parseVP ys,
    vp' <- unifyCatLists [vp] subcataux,
    agreeC aux vp ]
```
Apply one of these rules for parsing a VPbar:

```haskell
parseVPbar :: Parser Cat Cat
parseVPbar = finiteVpRule <|> auxVpRule
```

### VPs

For a VP, pick the Verb from the lexicon, and use its subcatlist information to impose the appropriate constraints on the verb complement. Just a single rule for VPs:

```haskell
vpRule :: Parser Cat Cat
vpRule = \xs ->
  [ (Ct VP "_" (fvC verb) []) (verb : xps'), zs) |
    (verb,ys) <- foundCat VERB xs,
    subcatlist <- [subcatC verb],
    (xps,zs) <- parseDPsorPPs ys,
    length subcatlist == length xps,
    xps' <- unifyCatLists subcatlist xps ]
```

For parsing a VP, apply this rule:

```haskell
parseVP :: Parser Cat Cat
parseVP = vpRule
```

### Sentences

In sentences, nominative case should be assigned to the subject. Here is the appropriate feature value structure:

```haskell
nominative :: FV
nominative = Has (Ft Case [Nom])
```

The effect of assigning nominative case to the subject is to rule out sentences like *Me broke the vase*. Just a single rule for sentences:
For parsing sentences, apply this rule.

For parsing sentences, apply this rule.

Finally, here is a function that puts it all together:

You can try this out with `prs "I broke the vase."`, and so on.

8.9 Unbounded Dependencies

In this section we will demonstrate the use of a parse stack of extracted categories for the treatment of unbounded dependencies (also called extraction or dislocation phenomena). This treatment is related to the gap threading mechanism proposed for extraction in GPSG [GKPS85] and HPSG [PS94].

Left extraction in natural language occurs when a subconstituent of some constituent is missing, and some other constituent to the left of the incomplete constituent represents that missing constituent in some way. In the generative tradition, such dislocations are accounted for by means of transformations that move a constituent while leaving a trace. Computational and
logic-oriented approaches to NL processing and understanding replace the transformational account with an *in situ* analysis, through gap threading (lexical functional grammar, categorial grammar, GPSG, HPSG) and/or through extension of the context free rule format with wrapping operations (extraposition grammars, tuple-based and tree-based extensions of context free grammars).

As we will see now, treatment of extraction phenomena with extraction stacks remains close to the spirit of the original movement analysis. Relative clause formation in English is a simple example of left extraction. The structure of the relative clause in (8.1) is represented by the annotation that links a relative pronoun *that* to its trace $t_i$.

8.1 I hated the man that $t_i$, the woman sold the house to $t_j$.

We already have a parser for a fragment of natural language without relative clauses. In our extended fragment, the relative clause in (8.1) will receive the following parse.

To treat dependencies, we will enrich our parsers with a stack of ‘extracted categories’. Call this a stack parser:

$$\text{type StackParser } a \ b = [b] \to [a] \to [(b, [b], [a])]$$

The following function pushes a category to the stack of a a stack parser:

$$\text{push :: Cat }\to \text{ StackParser Cat Cat }\to \text{ StackParser Cat Cat}$$

$$\text{push } ct \ f \ cts = f (ct:cts)$$
8.9. UNBOUNDED DEPENDENCIES

A parser combinator for popping a category from the gaps stack, provided it has the right category label:

```haskell
pop :: CatLabel -> StackParser Cat Cat
pop _ [] xs = []
pop l (u:us) xs | labelC u == l = [(u,us,xs)]
| otherwise = []
```

Now we have to redo what we did in the previous section in terms of stack parsers:

```haskell
fCat :: CatLabel -> StackParser Cat Cat
fCat label stack [] = []
fCat label stack (x:xs) = [ (x,stack,xs) | labelC x == label ]
```

```haskell
infixr 4 <+>
```

```haskell
(<+>) :: StackParser a b -> StackParser a b -> StackParser a b
(f1 <+> f2) stack xs = f1 stack xs ++ f2 stack xs
```

```haskell
dpR :: StackParser Cat Cat
dpR = \ us xs -> [(Ct DP "\_" fvs [], [det,cn], ws, zs) |
                 (det,vs,ys) <- pDET us xs,
                 (cn, ws,zs) <- pCN vs ys,
                 fvs <- unifCatsFVs det cn ]
```

The stack parser that handles DPs should take the so-called island constraints on movement (first proposed in [Ros67]) into account. The island constraints on extraction rule out configurations of the form

```
...that_i...[DP...[REL that_j [s_i...t_i...t_j]]]]
```

The island constraints are imposed to explain the ungrammaticality of examples like (8.2).


8.2 *I hated the woman that, \( t_i \) you loved \( [DP \] the man that, \( t_j \) sold the house to \( t_j \)].

As you can check, example (8.2) does not receive a parse in the fragment, but (8.3) does.

8.3 I hated the woman that, \( t_i \) you loved \( [DP \] the man that, \( t_j \) you sold the house to \( t_j \)].

Because of the fact that DP is a DP gap barrier, our treatment of relative clauses can get by with just a single gap marker #. Occurrences of # are always linked to the closest relative pronoun having the occurrence in its scope. Thus, removing the trace indices from (8.3) is completely harmless:

8.4 I hated the woman that \[ # loved \[ DP the man that [you sold the house to #]].

Each occurrence of # gets bound by the closest that above it. This property follows directly from the island constraint. Incidentally, in order to formulate the island constraint, one has to describe what is forbidden and rule it out. For this one needs coindexing of traces. For describing proper parse trees with left dislocation traces in a non-ambiguous way, coindexing of traces turns out not to be needed.

If a gap of type DP is on top of the stack, then it is possible to parse the DP as this gap, and it is also possible to parse the DP using the lexicon or a DP rule, and carry the gap along, but it is impossible to resolve the gap somewhere inside the DP.

To achieve this, we define a function for propagating a category through a stack parser:

\[
\begin{align*}
propag \, : \, & \text{CatLabel} \to \text{StackParser} \text{Cat} \text{Cat} \to \text{StackParser} \text{Cat} \text{Cat} \\
propag \, & \, \text{l} \, f \, [] \, \text{xs} \, = \, [] \\
propag \, & \, \text{l} \, f \, (\text{u} : \text{us}) \, \text{xs} \\
| \, \text{labelC} \, \text{u} \, \text{==} \, \text{l} \, = \, [(\text{cat}, \text{u} : \text{vs}, \text{ys}) | (\text{cat}, \text{vs}, \text{ys}) \leftarrow f \, \text{us} \, \text{xs}] \\
| \, \text{otherwise} \, = \, []
\end{align*}
\]

The following code for handling DPs handles a DP on top of the stack in the appropriate way:

\[
\begin{align*}
pDP \, : \, & \text{StackParser} \text{Cat} \text{Cat} \\
pDP \, & \, = \, f\text{Cat} \, \text{DP} \\
& \; \leftrightarrow \, \text{dpR} \\
& \; \leftrightarrow \, \text{pop} \, \text{DP} \\
& \; \leftrightarrow \, \text{propag} \, \text{DP} \, \text{pDP}
\end{align*}
\]

The rule for PPs:
8.9. UNBOUNDED DEPENDENCIES

ppR :: StackParser Cat Cat
ppR = \ us xs -> [(Ct PP "_" fvs [] [prep,dp], ws, zs) |
                        (prep,vs,ys) <- pPrep us xs,
                        (dp,ws,zs)  <- pDP vs ys,
                        fvs        <- unifCatsFVs prep dp ]

Parsing a PP: again we make sure that a PP gap on top of the stack is handled in the appropriate way.

pPP :: StackParser Cat Cat
pPP = fCat PP
    <+> ppR
    <+> pop PP
    <+> propag PP pPP

Parsing a determiner:

pDET :: StackParser Cat Cat
pDET = fCat DET

The rule for parsing CNs as a lexical CN plus a relative clause goes like this:

cnrelR :: StackParser Cat Cat
cnrelR = \ us xs -> [(Ct CN "_" fvs [] [cn,rel], ws,zs) |
                        (cn,vs,ys)  <- fCat CN us xs,
                        (rel,ws, zs)  <- pRel vs ys,
                        fvs        <- unifCatsFVs cn rel ]

For parsing relative clauses we use the stack mechanism for pushing a DP gap down the stack while parsing a sentence:
A second rule caters for so-called ‘that’-less relative clauses, as in (8.5).

8.5 *Bill loved the woman he had sold the house to.*

For parsing a relative clause, apply one of these two rules.

```
thatlessR :: StackParser Cat Cat
thatlessR = \ us xs ->
    [ (Ct COMP "_" [] [] [s] ,vs, ys) |
      gap <- [Ct DP "\#" [Has (Ft Case [Acc])] [] []],
      (s,vs,ys) <- push gap pSent us xs ]
```

Parsing a CN:

```
pCN :: StackParser Cat Cat
pCN = fCat CN <+> cnrelR
```

Parsing a DP or PP:

```
pDPorPP :: StackParser Cat Cat
pDPorPP = pDP <+> pPP
```
Parsing a list of DPs and PPs (note that we need a local declaration of $\langle \rangle$ because of the type difference with the global $\langle \rangle$):

```haskell
pDPsorPPs :: [Cat] -> [Cat] -> [[(Cat),[Cat],[Cat]]]
pDPsorPPs = pDPorPP $\langle \rangle$ pDPsorPPs $\langle \rangle$ eps
where
  eps us xs = [[],[us,xs]]
  (f $\langle \rangle$ g) us xs =
    [(r:rs,ws,zs) | (r,vs,ys) <- f us xs, (rs,ws,zs) <- g vs ys ]
  (f $\langle \rangle$ g) us xs = f us xs ++ g us xs
```

```haskell
fPrep :: StackParser Cat Cat
fPrep = fCat PREP
```

```haskell
fAUX :: StackParser Cat Cat
fAUX = fCat AUX
  $\langle \rangle$ pop AUX
  $\langle \rangle$ propag AUX fAUX
```

```haskell
finiteVpR :: StackParser Cat Cat
finiteVpR = \ us xs ->
  [ (Ct VPBAR "_" (fvC vp') []) [vp], vs, ys ) |
    (vp, vs, ys) <- pVP us xs,
    vp' <- assignFV finiteForm vp ]
```

```haskell
auxVpR :: StackParser Cat Cat
auxVpR = \ us xs ->
  [ (Ct VPBAR "_" fva []) (aux : vp'), ws, zs ) |
    (aux@(Ct AUX phon fva subcataux []),vs,ys) <- pAUX us xs,
    (vp,ws,zs) <- pVP vs ys,
    vp' <- unifyCatLists [vp] subcataux,
    agreeC aux vp ]
```
A Yes/No question can be thought of as the result of extracting an auxiliary from a sentence.
8.9. UNBOUNDED DEPENDENCIES

This gives the YN-question *Do you love me?* the following analysis:

Finally, we turn to Wh-questions. A Wh-question can be thought of as the result of extracting a Wh-phrase (either a DP or a PP) from a YN-question. Here is a check whether a category is a Wh category:

Parsing a Wh-question is just a matter of first finding a Wh-phrase and next letting a parser for YN-questions look for a matching Wh-phrase gap:
For *What did they break it with?* this yields the following analysis:

For *With what did they break it?* we get the following:

All of the above is put together in the following parse function for sentences, yes/no questions and wh-questions.
8.10 BORING LATEX AND SYSTEM STUFF

Here is module for stand-alone use. This module can be used to compile a program that parses the argument that the program is called with, and writes the resulting parse tree in \LaTeX{} format to a temporary file.\footnote{For this you will need a Haskell compiler, e.g., \texttt{GHC} (see \url{http://www.haskell.org/ghc/}), and the \LaTeX{} typesetting system (see \url{http://www.tug.org/}).}

```haskell
module Main
where

import System
import LexDB
import Parsing

Define a class of types that can be displayed with \LaTeX{}. For this class a function \texttt{latex} is defined for displaying the members of the category out in \LaTeX{} format. In the case of the members of class \texttt{Cat}, we will employ the tree format of the \LaTeX{} package \texttt{qtree}. \texttt{Qtree} is a package written by Jeffrey Mark Sisskind, with a front end by Alexis Dimitriadis. The reader is referred to the documentation of this package for details of the code below. Bear in mind also that \texttt{\backslash} is a special character in Haskell, and that \texttt{\backslash\backslash} has to be used for quoting the backslash character.

First we say that the \texttt{Latex} class is a subclass of the \texttt{Show} class, and that members of the class should have a function \texttt{latex} defined on them.
```
class Show a => Latex a where
    latex :: a -> String

Declare the type of feature value structures to be in class Latex, and make sure that latexing only shows the values. This is to make sure that trees will fit on the screen or paper.

instance Latex FV where
    latex (Has fe) = show (valuesF fe)
    latex _ = ""

One of the category labels should be latexed in a special way:

instance Latex CatLabel where
    latex VPBAR = "\$\overline{\text{VP}}\$"
    latex label = show label

Next, declare the type of categories to be an instance of Latex, and define the appropriate latex function for categories:

instance Latex Cat where
    latex (Ct label _ [] _ []) =
        "\n[. " ++ latex label ++ " _ ]"
    latex (Ct label phon [] _ []) =
        "\n[. " ++ latex label ++ " { " ++ phon ++ " } ]"
    latex (Ct label _ [] _ constituents) =
        "\n[. " ++ latex label ++ " _ ]"
    latex (Ct label phon [] _ constituents) =
        "\n[. " ++ latex label ++ " { " ++ phon ++ " } "
            ++ latex constituents ++ " ]"
    latex (Ct label _ features _ []) =
        "\n[. " ++ latex label ++ latex features ++ " ]"
    latex (Ct label _ features _ constituents) =
        "\n[. " ++ latex label ++ latex features ++ " ]"
    latex (Ct label phon features _ constituents) =
        "\n[. " ++ latex label ++ latex features ++ " { " ++ phon ++ " } "
            ++ latex constituents ++ " ]"
Lists of things that are in class \texttt{Latex} are latexed by string concatenation:

\begin{verbatim}
instance Latex a => Latex [a] where
  latex [] = ""
  latex (x:xs) = latex x ++ latex xs
\end{verbatim}

Now that we have the \texttt{latex} formatting function defined for categories and lists of categories, we can define a \LaTeX tree drawing function:

\begin{verbatim}
trees :: String -> [String]
trees string =
  [ "\\Tree " ++ latex s ++ "\n\n" | s <- parse string ]
\end{verbatim}

The main function of the module is called \texttt{main}. This is the function that gets called when the module is compiled for stand-alone use.

\begin{verbatim}
main :: IO ()
main = do {
  args <- getArgs;
  let
    trs = trees (concat args)
    in
    if null trs then error "NO PARSE"
    else
      writeFile "/tmp/tree.tex" (concat trs)
  }\end{verbatim}

This parses a sentence and saves a \LaTeX-formatted tree of the parse to a temporary file (/tmp/tree.tex) for later display.

\LaTeX display can then yield trees like the following:
Suppose a \LaTeX file named \textit{ParseTree.tex} with the following contents is present in directory `/tmp:

```latex
\documentclass[12pt]{article}
\usepackage{amsmath}
\usepackage{qtree}
\pagestyle{empty}
\begin{document}
\input{/tmp/tree.tex}
\end{document}
```
If the program is compiled as `prs`, then the following is an example of a UNIX shell script for displaying parse trees:

```
#!/bin/bash
pushd /tmp
/home/jve/bin/prs "$1" \
  && latex ParseTree \
  && xdvi -mfmode ljfour:600 ParseTree &
popd
```

Call this script `parse`. Once installed, this script can be invoked from the command line, as follows:

```
$ parse "Max broke the vase."
```

**Exercise 8.7 (For Windows programmers:)** Write a module for display of parse trees on Windows machines.

### 8.11 Further Reading

Natural language parsing is assimilated to logical deduction in [PW83]. The bible of parsing in the context of formal language theory is [AU72]. Parsing natural language fragments with Prolog is the topic of [PS87]. Parser combinators in the context of functional programming are introduced in [Wad85] and receive further discussion in [Hut92]. Examples of parser combinators can also be found in [JS01].
Chapter 9

Handling Relations in Type Theory

Summary

The problem of encoding relations in type theory in such a way that the usual logical operations can be applied to them (relations should be ‘conjoinable’, as the jargon has it) has been studied extensively in the semantic literature. We will define a type for arbitrary arity relations in polymorphic type theory.

9.1 Interpreting DP Lists

Instead of analysing the sentence *Every cat chased a mouse* as a relation between the CN property of being a cat and the VP property of chasing mice (namely the relation of inclusion), it is also possible to look at the complex expression *Every cat — a mouse*, and interpret that as a function that takes a relation (a denotation of a transitive verb, such as *chased, killed*) and produces a truth value. Similarly, *Every firm received a letter from some lawyer* can be analysed as stating that the set of firms is included in the set of letters received from some lawyer, but it is also possible to look at the complex expression *Every firm — a letter from some lawyer*, and even at *Every firm — a letter — some lawyer*. The interpretation of *Every firm — a letter from some lawyer* is again a function from binary relations to truth values, the interpretation of *Every firm — a letter — some lawyer* is a function from ternary relations to truth values.

Call a function from properties (unary relations) to truth values a type \( (1) \) function, a function from binary relations to truth values a type \( (2) \) function, and, in general, a function from \( n \)-ary relations to truth values a type \( (n) \) function.

A type \( (1) \) function \( f \) can be lifted to a function \( f' \) from \( n + 1 \)-ary relations to \( n \)-ary relations by means of:

\[
f'(R) = \{ \langle d_1, \ldots, d_n \rangle \mid f(\{d | \langle d_1, \ldots, d_n, d \rangle \in R \}) = 1 \}.
\]

This is of the required type, for if \( R \) is an \( n + 1 \)-ary relation then \( f'(R) \) is an \( n \)-ary relation.
These lifted type \( h_1 \) functions can then be composed by means of:

\[
f' \circ g' = \lambda R. f'(g'(R)).
\]

Here it is assumed, of course, that \( R \) has arity \( \geq 2 \). The lifted type \( h_2 \) function \( f' \circ g' \) maps \( n + 2 \)-ary relations to \( n \)-ary relations.

Similarly, a type \( h_n \) function \( F \) can be lifted to a function \( F' \) from \( m + n \)-ary relations to \( m \)-ary relations, by means of:

\[
F'(R) = \{ \langle d_1, \ldots, d_m \rangle \mid F([\langle d_{m+1}, \ldots, d_{m+n} \rangle]|\langle d_1, \ldots, d_n, d_{m+1}, \ldots, d_{m+n} \rangle \in R) = 1 \}.
\]

Clearly, many type \( h_2 \) functions can be decomposed in this way into pairs of type \( h_1 \) functions. E.g., the type \( h_2 \) function \( F \) that interprets the complex expression \( \text{every cat } \_ \_ \text{ a mouse} \) can be decomposed into a type \( h_1 \) function \( g \) that interprets \text{a mouse} and a type \( h_1 \) function \( f \) that interprets \text{every cat}, for \( f' \circ g' \) equals \( F' \).

In [Kee92] it is shown that there are cases where compound quantifiers are \textit{not} decomposable. Keenan shows that the following sentence exhibits an example of non-decomposable type \( h_2 \) quantification:

\[9.1\] Different students answered different questions.

For sentence (9.1) to make sense, we have to assume that there are at least two students. The sentence is true if there is a one-to-one correspondence between students and sets of questions they answered. Thus, \textit{Different students }\_\_ \textit{different questions} is interpreted as the type \( h_2 \) function expressing that its argument relation \( R \) satisfies the property that all the \( aR \), with \( a \) ranging over students, are different (here \( aR \) is used as shorthand for \( \{ x \mid aRx \} \)). Keenan has an ingenious method to prove this fact. He states and proves a theorem to the effect that for any two type \( h_2 \) functions \( F, G \) that are reducible it holds that these functions are equal iff they act the same on cartesian products, i.e., if for all subsets \( P, Q \) of the domain of discourse \( U \) it holds that \( F(P \times Q) = G(P \times Q) \).

How can this be used to show that a type \( h_2 \) function \( F \) is non-reducible? Here is how, for the example of (9.1). Let \( F \) be the type \( h_2 \) function that interprets \textit{different students }\_\_ \textit{different answers}. Pick a universe containing at least three students \( s_1, s_2, s_3 \) and at least two questions \( q_1, q_2 \). Let \( P \times Q \) be a product relation, i.e., a relation that links every object in \( P \) to every object in \( Q \). If there are two students not in \( P \), then they bear \( P \times Q \) to the same questions, namely, no questions. If there are two students in \( P \), then the questions they bear \( P \times Q \) to are again the same, namely \( Q \cap \text{QUESTION} \). Again, \( F(P \times Q) = 0 \). Let \( 0 \) be the type \( h_1 \) function that is false for any argument. Then, by the above, \( F(R) = 0 \circ 0(R) \) for any product relation \( R \). But obviously, \( F \) is different from the composition \( 0 \circ 0 \), for \( F \) is true of \( \{ \langle s_1, q_1 \rangle, \langle s_2, q_2 \rangle \} \), and \( 0 \circ 0 \) is not. Thus, by Keenan’s theorem, \( F \) is not reducible.

Here are some further examples of quantifiers that Keenan shows to be not reducible.

\[9.2\] Three boys in my class dated the same girl.

\[9.3\] The women at the wedding all wore different hats.
9.4 The students gave different answers to different questions.

In this chapter we will interpret determiner phrase clusters as relation reducers, with a cluster consisting of \( n \) DPs interpreted as a type \((n)\) function. In the cases where such a DP cluster interpretation is reducible to \( n \) type \((1)\) functions, the reduction \( f_1 \circ \cdots \circ f_n \) imposes a scope ordering on the DPs. This means that quantifier scoping can be handled by considering different orderings of the reducing type \((1)\) functions, while taking care that the scope reorderings respect the relevant argument binding constraints. E.g., to reorder the scoping of \( f \circ g \), first define a swap operation \( S \) by means of:

\[
S := \lambda R \lambda x \lambda y. (Ry)x.
\]

This reorders the outermost two argument places of a relation. The inverse scope reading of \( f \) and \( g \) can now be expressed as \( g \circ f \circ S \).

9.2 A Data Type for Relations

We declare the chapter module and import some relevant modules that were defined before.

```haskell
module HRITT where

import Domain
import Model
import LexDB
import Parsing
```

A binary relation on a set \( A \) can be viewed as a function of type \( A \to \mathcal{P}(A) \). It maps elements \( a \) of \( A \) to subsets \( aR \) of \( A \), where \( aR \) is shorthand for \( \{ x \in A \mid aRx \} \). Equivalently, a binary relation can be viewed as a function of type \( A \to A \to t \). It maps elements \( a \) of \( A \) to characteristic functions \( \chi_a \) on \( A \), where \( \chi_a \) is given by \( \chi_a(b) = 1 \) iff \( aRb \).

Similarly, a unary relation on a set \( A \) is a function of type \( A \to t \), a ternary relation on a set \( A \) is a function of type \( A \to A \to A \to t \), and so on. A special case is a nullary relation: this is just a truth value. Note that an \( n + 1 \)-ary relation on \( A \) can be viewed as a function from \( A \) to the type of \( n \)-ary relations over \( A \).

Declare \( \text{Rel} \) as a datatype of arbitrary arity relations over an arbitrary type \( a \), as follows.

```haskell
data Rel a = R1 Bool | R2 (a -> Rel a)
```

What this says is that a relation either is nullary, in which case it is just a boolean preceded by the tag \( \text{R1} \), or it is \( n \)-ary, with \( n \geq 1 \). This is indicated by the tag \( \text{R2} \). If a relation over objects
of type $a$ is $n$-ary, with $n \geq 1$, then it can be viewed as a function of type $a \to R_{n-1}$, where $R_{n-1}$ indicates the type of $n-1$-ary relations.

Here is a function for applying a relation to its argument. In case the relation turns out to be nullary, an error is generated.

```haskell
apply :: Rel a -> a -> Rel a
apply (R1 _) _ = error "no argument position left"
apply (R2 f) x = f x
```

Next, we define a function for determining the arity of a relation. Here we need a constraint on $a$, for we should be able to locate an instance of the type. If the type is bounded, then \texttt{minBound} gives us the desired instance.

```haskell
arity :: Bounded a => Rel a -> Int
arity (R1 _) = 0
arity (R2 f) = arity (f minBound) + 1
```

A binary relation over objects of type $a$ can be viewed as a function of type $a \to a \to t$, or, uncurried, of type $(a,a) \to t$, a ternary relation over objects of type $a$ as a function of type $a \to a \to a \to t$, or, uncurried, of type $(a,a,a) \to t$, and so on. Here are functions for encoding relations of various fixed arities into \texttt{Re}ls (it is assumed that the relations are uncurried).

```haskell
encode0 :: Bool -> Rel a
encode0 b = R1 b

encode1 :: (a -> Bool) -> Rel a
encode1 f = R2 (\ x -> encode0 (f x))

encode2 :: ((a,a) -> Bool) -> Rel a
encode2 f = R2 (\ x -> encode1 (\ y -> f (x,y)))

encode3 :: ((a,a,a) -> Bool) -> Rel a
encode3 f = R2 (\ x -> encode2 (\ (y,z) -> f (x,y,z)))
```

We also need conversions from 0-ary relations to type \texttt{Bool} and from unary relations to type $a \to \texttt{Bool}$. Note that the decoding function \texttt{decode0} is only defined for nullary relations (relations of arity 0), the function \texttt{decode1} only for relations of arity 1:
9.2. A DATA TYPE FOR RELATIONS

Here is a flexible means of representing a relation as a list of lists.

```
rel2lists :: (Enum a, Bounded a) => Rel a -> [[a]]
rel2lists (R1 b) = [[] | b]
rel2lists (R2 f) =
    [ x: tuple | x <- [minBound..maxBound],
      tuple <- rel2lists (f x) ]
```

So why didn’t we use the type \[[a]\] as the type of arbitrary arity relations in the first place? The trouble is that the type \[[a]\] does not guarantee a fixed arity in the way \texttt{Rel} does. Translating from \texttt{Rel a} to \[[a]\] yields lists that are all of the same length, with the length determined by the arity of the relation. But the type \[[a]\] is more general than that. List members can have different lengths, which is undesirable for the representation of relations.

For good measure, here are conversions from unary, binary and ternary predicates to lists of lists:

```
upred2lists :: (Enum a, Bounded a) => (a -> Bool) -> [[a]]
upred2lists p = [ [x] | x <- filter p [minBound..maxBound] ]

bpred2lists :: (Enum a, Bounded a) => ((a,a) -> Bool) -> [[a]]
bpred2lists p = [ [x,y] | (x,y) <- filter p tuples ]
    where
tuples = [ (u,v) | u <- [minBound..maxBound],
                   v <- [minBound..maxBound] ]

tpred2lists :: (Enum a, Bounded a) => ((a,a,a) -> Bool) -> [[a]]
tpred2lists p = [ [x,y,z] | (x,y,z) <- filter p tuples ]
    where
tuples = [ (u,v,w) | u <- [minBound..maxBound],
                   v <- [minBound..maxBound],
                   w <- [minBound..maxBound] ]
```
This gives:

```
HRIIT> tpred2lists give
[[L,G,M],[M,V,L]]
```

### 9.3 Boolean Algebras of Relations

In this section we define generalisations of the boolean operations to relations.

Conjoining of relations of the same arity is a generalisation of conjunction for booleans. The result is conjoining relations \( r \) and \( s \) is called the boolean meet of \( r \) and \( s \), and is often denoted with \( r \cap s \). It is implemented in terms of a general function for folding relations (of the same arity) for a given boolean operation:

\[
\text{foldREL} :: (\text{Bool} \to \text{Bool} \to \text{Bool}) \to \text{Rel} a \to \text{Rel} a \to \text{Rel} a
\]

\[
\text{foldREL} \; \text{op} \; (R1 \; b) \; (R1 \; c) = R1 \; (\text{op} \; b \; c)
\]

\[
\text{foldREL} \; \text{op} \; (R2 \; f) \; (R2 \; g) = R2 \; (\lambda \; x \to \text{foldREL} \; \text{op} \; (f \; x) \; (g \; x))
\]

\[
\text{foldREL} \; \text{op} \; _ \; _ = \text{error} \; "\text{different arities}"
\]

In terms of this, relation conjunction is defined by doing a relational fold with conjunction:

\[
\text{conjR} :: \text{Rel} a \to \text{Rel} a \to \text{Rel} a
\]

\[
\text{conjR} = \text{foldREL} \; (\&\&)
\]

Disjoining relations works the same way. The result of disjoining \( r \) and \( s \) is called the boolean join of \( r \) and \( s \), and is denoted with \( r \sqcup s \).

\[
\text{disjR} :: \text{Rel} a \to \text{Rel} a \to \text{Rel} a
\]

\[
\text{disjR} = \text{foldREL} \; (||)
\]

Finally, here is an operation for taking the complement of a relation. The complement (negation) of a relation \( r \) is often written as \( \overline{r} \).

\[
\text{negR} :: \text{Rel} a \to \text{Rel} a
\]

\[
\text{negR} \; (R1 \; b) = R1 \; (\text{not} \; b)
\]

\[
\text{negR} \; (R2 \; f) = R2 \; (\lambda \; x \to \text{negR} \; (f \; x))
\]
9.4 THE NEED FOR FLEXIBLE TYPE ASSIGNMENT

These definitions illustrate that relations of the same arity form a boolean algebra [DP90, Hal63]. These operations are what is needed for performing logical relations on complex VPs, to get interpretations for *to hurt but not kill*, *to break or drop*, *to give or sell*, and so on. If $H$ is the binary relation that interprets *to hurt*, and $K$ the binary relation that interprets *to kill*, then $H \cap K$ interprets *to hurt but not kill*, and so on.

Here are some illustrations using the implementation:

HRITT> rel2lists (disjR (encode2 wash) (encode2 shave))
HRITT> rel2lists (disjR (encode3 give) (encode3 sell))

Note that the generalized operations for conjunction, disjunction and negation are in fact generalizations of conjunction, disjunction and negation for Booleans. Booleans can be viewed as 0-ary relations, and we have generalized the operations to $n$-ary relations, for arbitrary $n$.

**Exercise 9.1** Define and implement the relation of entailment between $n$-ary relations. For the implementation, assume that the underlying type is a bounded enumeration type.

9.4 The Need for Flexible Type Assignment

There is extensive semantic literature on the need for flexible type assignment to syntactic categories (see, e.g., [Hen93]). Many of the arguments for this have to do with the need to account for various scoping ambiguities resulting from the interaction of quantified DPs with other quantified DPs or with intensional verb contexts.

Now note that the type $\text{Rel}$ is flexible, for this type can do duty for the whole family of types $t, e \rightarrow t, e \rightarrow e \rightarrow t$, and so on.

We intend to use the flexible type $\text{Rel}$ for the interpretations of verbs, and we will perform a ‘flexible lift’ on DP interpretations, so that a DP can be viewed as an operation on verb meanings, i.e., as a function of type $\text{Rel} \rightarrow \text{Rel}$. For this perspective on the semantics of DPs, compare [Kee87, Kee92].

9.5 Lifting the Types of DP Interpretations

An important conversion function that we need is that from a characteristic function of properties to a function that maps relations to relations (type $\text{Rel} \ a \rightarrow \text{Rel} \ a$). We will use this to lift the type of DP interpretations.

Consider the problem of applying a function of type

$$(a \rightarrow \text{Bool}) \rightarrow \text{Bool}$$
not to a property (type \( a \to \text{Bool} \)), but to a relation with arity \( > 1 \).

Note that \( a \to \text{Bool} \) can be viewed as a the type of a relation of arity 1, and \( \text{Bool} \) as the type of a relation of arity 0. Thus,

\[
(a \to \text{Bool}) \to \text{Bool}
\]

is the type of reduction of 1-ary to 0-ary relations.

So what about the general case: the reductions of \( n + 1 \)-ary relations to \( n \)-ary relations?

The general shape of a relation of arity \( n + 1 \) is:

\[
\lambda x_1 \cdots \lambda x_n \lambda y.r
\]

Here, \( r \) is a Boolean expression (type \( \text{Bool} \)).

Converting a property function \( f \) into a relation transformer is done by reducing that relation by recursion.

If the argument relation has the form \( \lambda y.r \), with \( r \) of type \( \text{Bool} \), then property function \( f \) can be applied to it, yielding a boolean \( f(\lambda y.r) \).

If the argument relation has the form \( \lambda x_1 \cdots \lambda x_n \lambda y.r \), the new relation is given by (9.2).

\[
\lambda x_1 \cdots \lambda x_n.f(\lambda y.r)
\]

So this is what reduction of a relation \( \lambda x_1 \cdots \lambda x_n \lambda y.r \) should yield. Our obvious next question is: how is the operation defined that yields this result?

Let (9.3) be the result of reducing \( f \) with \( \lambda x_1 \cdots \lambda x_n \lambda y.r \).

\[
R f (\lambda x_1 \cdots \lambda x_n \lambda y.r).
\]

Then our question becomes: how is \( R \) defined? The recursion that defines the operation \( R \) can be expressed as follows:

\[
R f (\lambda y.r) := f(\lambda y.r)
\]

\[
R f (\lambda x_1 \cdots \lambda x_n \lambda y.r) := \lambda x.R f ((\lambda x_1 \cdots \lambda x_n \lambda y.r)(x)).
\]

Here is the implementation:

```haskell
reduce :: Bounded a => ((a -> Bool) -> Bool) -> Rel a -> Rel a
reduce f r
    | arity r == 1 = R1 (f (decode1 r))
    | otherwise     = R2 (\ x -> (reduce f (apply r x)))
```
9.6 Scope Reversal of Quantifiers

Using this, the conversion function that we need for lifting the types of DP interpretations is defined as follows (assume \( f \) is the DP interpretation and \( r \) is the interpretation of the verb):

\[
\text{lift} = \lambda f \lambda r. R f r.
\]

Here is the implementation:

```
lift :: Bounded a => ((a -> Bool) -> Bool) -> Rel a -> Rel a
lift = \ f r -> reduce f r
```

Note that if \( r \) has arity \( n + 1 \), then \( \text{lift} f r \) has arity \( n \). Thus, \( \text{lift} \) is indeed a reducer of \( n + 1 \)-ary relations to \( n \)-ary relations.

9.6 Scope Reversal of Quantifiers

We have defined generalized negation and generalized quantification as maps from relations to relations, operators of type \( \text{Rel} \ a \to \text{Rel} \ a \). A difference between these two kinds of operators is that generalized negation does not reduce the arity of its argument relation, but generalized quantification does. The relational operator interpretation of a DP reduces the arity of its argument relation by 1, for applying the generalized quantifier to the relation consumes one of the arguments of the relation.

Now suppose we want to swap the scope of two generalized quantifier operations. We can view the scope swap as an operation of the following type:

\[
\text{qscopeReversal :: (Rel a -> Rel a) -> (Rel a -> Rel a) -> (Rel a -> Rel a)}
\]

The swap takes two operators \( Q_1 \) and \( Q_2 \) and creates a new operator that should express the effect of giving \( Q_2 \) scope over \( Q_1 \).

How should this scope swapping operation be defined? Just defining it as \( Q_2 \cdot Q_1 \), for performing \( Q_2 \) after \( Q_1 \) and thereby giving \( Q_2 \) wide scope, is almost right, but not quite. The problem is that this switches the argument positions that the operators work on.

Look at it like this. In \( Q_1(Q_2r) \), \( Q_2 \) gets applied to \( r \) first, so it applies to the outermost argument of \( r \). Suppose \( r \) has arity \( n \), then \( Q_2 \) applies to the \( n \)-th argument, and produces a new relation \( Q_2r \) of arity \( n - 1 \). Next, \( Q_1 \) applies to the outermost argument of \( Q_2r \), and produces a new relation \( Q_1(Q_2r) \) of arity \( n - 2 \).

Now suppose we want to perform the applications in the opposite order. Then \( Q_1 \) has to apply to \( r \) first, but \( Q_1 \) still should apply to the \( n - 1 \)-th argument position of \( r \). Next, we should get a new relation \( Q_1r \), and \( Q_2 \) should apply to what used to be the \( n \)-th argument of \( r \). To bring this about, we have to define the reverse scoping as follows:

\[
Q_2 \cdot Q_1 \cdot S,
\]
where $S$ is a swap operation defined by:

$$S := \lambda r \lambda x \lambda y. (ry)x.$$

Here is the implementation:

```haskell
swap :: Rel a -> Rel a
swap = \ r ->
    R2 (\ x ->
        R2 (\ y ->
            (apply (apply r y) x)))
qscopeReversal :: (Rel a -> Rel a) -> (Rel a -> Rel a)
    -> (Rel a -> Rel a)
qscopeReversal = \ q1 q2 -> q2 . q1 . swap
```

Thus, for scope reversal, we first swap the two outermost argument places of the relation, and next apply the two quantifiers in reversed order.

For a scope reversal of a quantifier and a negation no argument swapping is needed, as the negation operation is not an arity reducer.

In Section 9.10 we will see that the presence of ordered lists of realized theta roles on verb concepts allows us to generate all the possible scope readings of a sentence in a particularly easy way, by swapping the quantifiers in parallel with the corresponding realized theta roles.

### 9.7 Interpreting Verbs as Arbitrary Arity Relations

We will interpret verbs as arbitrary arity relations, type $\text{Rel}$. In a full fledged theory of thematic role assignment, the arity of the relation will depend on the number of realized ‘theta roles’ of the verb, but we will gloss over the details.

Theta theory is the theory of the way in which thematic roles like $\text{Agent}$, $\text{Patient}$, $\text{Cause}$, $\text{Instrument}$, and so on, function in the thematic structure of predicates in the lexicon and, more generally, in syntax. In Government and Binding theory [Cho81] and in the later Minimalist Program [Cho92], lexical entries for verbs are provided with a \textit{theta grid} in which the thematic roles are given that the verb assigns. The verb $\text{kill}$, e.g., would assign three thematic roles, one for $\text{Agent}$, one for $\text{Patient}$ and one for $\text{Instrument}$.

Although thematic relations have been with us for a long time — early references for an analysis in terms of “abstract thematic roles” are [Gru65] and [Jac72] — it seems fair to say that most linguistic work in theta theory is informal and speculative. Also, the theory of thematic roles has received relatively little attention from formal semanticists (with [Dow89] as one of the exceptions). There is no general agreement on the number of theta roles that should be
distinguished. [Rei00] came up with the proposal to link theta roles to feature clusters defined in terms of ± c (‘c’ for ‘cause’) and ± m (‘m’ for ‘mental’). There are 9 consistent subsets of the feature set

\{ +c, -c, +m, -m \},

and they can get linked to the following nine theta roles:

- Agent \([+c+m]\)
- Experiencer \([-c+m]\)
- Cause \([+c]\)
- Source, Subject Matter \([-m]\)
- None \([\ ]\)
- Instrument \([+c-m]\)
- Patient, Theme \([-c-m]\)
- Animate \([+m]\)
- Goal \([-c]\)

The thematic role assignments of a verb are part of the link between language and reality (the realm of things outside language that language is ‘about’). To implement thematic roles by means of type constraints in semantics would be a mistake, for in \textit{John kicked the door} and \textit{John kicked Bill, the door} and \textit{Bill} are both in the thematic role of \textit{Patient}, but this cannot be taken to imply that \textit{the door} and \textit{Bill} have to be things of the same kind (except in the trivial sense of being the type of things that can receive kicks).

Verb interpretations have a number of roles, identifiable by indices 1, \ldots, \(n\). Here is an example of a predicate:

\[
\begin{array}{ccc}
\text{kill} & 1 & 2 & 3 \\
\text{Cause} & \text{Patient} & \text{Instrument}
\end{array}
\]

If Max kills Lucy with an axe, then Max is the cause, Lucy is the patient, and the axe is the instrument. Of course, this is just for illustration. There may also be a theta role for location, and perhaps there are still other roles. In general, to interpret a predicate with \(n\) thematic roles, the predicate will get as its denotation an \(n\)-ary relation.

For a rudimentary version of Theta Theory it is enough to be able to extract a new relation from a given relation by specifying a list of argument positions. For the implementation we need to be able to pick the entity at an index from a list. For this, we use a predefined Haskell function. The only snag is that \(!!\) computes from index 0, whereas we would like to start at index 1.

\[
\text{role :: Int } \rightarrow [\text{a}] \rightarrow \text{a} \\
\text{role } n \ es = \ es !! (n-1)
\]

Using this, we implement the extraction function as follows:
extract :: (Enum a, Bounded a, Eq a) => [Int] -> Rel a -> Rel a
extract is rel = extr is (rel2lists rel)
  where
  extr [] ess = R1 (not (null ess))
extr (i:js) ess =
  R2 (\ x ->
      extr js (filter (\ es -> role i es == x) ess))

Thus, if $r$ is a ternary relation, then \texttt{extract [2,1,3]} $r$ is the relation that is like $r$ except for the fact that its first two argument positions are swapped. We will use the extraction function below in the implementation of scope reversals.

Some examples to check what we have done:

\[
\begin{align*}
\text{ex1} &= (~\text{rel2lists} . \text{extract [1,2]}~) (~\text{encode2 shave}~) \\
\text{ex2} &= (~\text{rel2lists} . \text{extract [2,1]}~) (~\text{encode2 shave}~) \\
\text{ex3} &= (~\text{rel2lists} . \text{extract [2,3]}~) (~\text{encode3 give}~) \\
\text{ex4} &= (~\text{rel2lists} . \text{extract [1,2,3]}~) (~\text{encode3 give}~) \\
\text{ex5} &= (~\text{rel2lists} . \text{extract [2,1,3]}~) (~\text{encode3 give}~)
\end{align*}
\]

This gives:

HRITT> ex1
[[A,J],[B,B]]
HRITT> ex2
[[B,B],[J,A]]
HRITT> ex3
[[G,M],[V,L]]
HRITT> ex4
[[L,G,M],[M,V,L]]
HRITT> ex5
[[G,L,M],[V,M,L]]

9.8 Interpretation of the Categories

First specify interpretation functions to map names to entities in the model:
**9.8. INTERPRETATION OF THE CATEGORIES**

```
int :: String -> Entity
int "ann" = ann; int "bill" = bill; int "lucy" = lucy
int "mary" = mary; int "johnny" = johnny
```

An interpretation function that maps lexical common nouns to predicates:

```
intP :: String -> Entity -> Bool
intP "man" = man; intP "men" = man
intP "woman" = woman; intP "women" = woman
intP "boy" = boy; intP "boys" = boy
intP "tree" = tree; intP "trees" = tree
intP "house" = house; intP "houses" = house
intP "leaf" = leaf; intP "leaves" = leaf
intP "stone" = stone; intP "stones" = stone
intP "gun" = gun; intP "gun" = gun
intP "person" = person; intP "persons" = person
intP "thing" = person; intP "things" = person
```

An interpretation function that maps verbs to relations:

```
intR "laughed" = encode1 laugh
intR "smiled" = encode1 smile
intR "loved" = encode2 love
intR "respected" = encode2 respect
intR "hated" = encode2 hate
intR "owned" = encode2 own
intR "wash" = encode2 wash
intR "shaved" = encode2 shave
intR "dropped" = encode2 drop0
intR "broke" = encode3 break0
intR "killed" = encode3 kill
intR "gave" = encode3 give
intR "sold" = encode3 sell
```

**Interpretation of DPs**

For the interpretations of DPs we use the familiar PTQ procedure proposed by Richard Montague. The interpretations of proper names have type `Entity`. Interpretations of DPs are
property functions, so they have type \((\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool}\). In order to lift the type of proper names to that of DPs, we interpret the names as functions for checking whether a given property is a property of the bearer of the name.

\[
\text{intDP} :: \text{Cat} \rightarrow (\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool}
\]
\[
\text{intDP} (\text{Ct DP name } \_ \_ [\_]) = \ \lambda \ p \rightarrow p \ (\text{int name})
\]
\[
\text{intDP} (\text{Ct DP } \_ \_ \_ [\text{det, cn}]) = (\text{intDET det}) \ (\text{intCN cn})
\]

**Interpretation of PPs**

Treat PPs like the DPs they contain:

\[
\text{intPP} :: \text{Cat} \rightarrow (\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool}
\]
\[
\text{intPP} (\text{Ct PP } \_ \_ [\text{prep, dp}]) = \text{intDP} \ dp
\]

**Interpretation of DPs or PPs**

\[
\text{intDPorPP} :: \text{Cat} \rightarrow (\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool}
\]
\[
\text{intDPorPP} \ dp@(\text{Ct DP } \_ \_ \_ \_ ) = \text{intDP} \ dp
\]
\[
\text{intDPorPP} \ pp@(\text{Ct PP } \_ \_ \_ \_ ) = \text{intPP} \ pp
\]

**Interpretation of DETs**

First order quantifiers:

\[
every, some, several, no, the :: \ (\text{Entity} \rightarrow \text{Bool}) \rightarrow (\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool}
\]
\[
every = \ \lambda \ p \ q \rightarrow \text{all} \ q \ (\text{filter} \ p \ \text{entities})
\]
\[
some = \ \lambda \ p \ q \rightarrow \text{any} \ q \ (\text{filter} \ p \ \text{entities})
\]
\[
several = \ \lambda \ p \ q \rightarrow \text{length} \ (\text{filter} \ p \ (\text{filter} \ q \ \text{entities})) \geq 2
\]
\[
no = \ \lambda \ p \ q \rightarrow \text{not} \ (\text{some} \ p \ q)
\]
\[
\text{the} = \ \lambda \ p \ q \rightarrow \text{length} \ (\text{filter} \ p \ \text{entities}) \leq 1
\]
\[
&& \ \text{some} \ p \ q
\]
9.8. **Interpretation of the Categories**

Use this in the interpretation of determiners:

\[
\text{intDET} :: \text{Cat} \rightarrow \\
(\text{Entity} \rightarrow \text{Bool}) \rightarrow (\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool} \\
\text{intDET} (\text{Ct} \ \text{DET} \ "\text{every}" \ _ \ _ \ _) = \text{every} \\
\text{intDET} (\text{Ct} \ \text{DET} \ "\text{all}" \ _ \ _ \ _) = \text{every} \\
\text{intDET} (\text{Ct} \ \text{DET} \ "\text{some}" \ _ \ _ \ _) = \text{some} \\
\text{intDET} (\text{Ct} \ \text{DET} \ "\text{a}" \ _ \ _ \ _) = \text{some} \\
\text{intDET} (\text{Ct} \ \text{DET} \ "\text{several}" \ _ \ _ \ _) = \text{several} \\
\text{intDET} (\text{Ct} \ \text{DET} \ "\text{no}" \ _ \ _ \ _) = \text{no} \\
\text{intDET} (\text{Ct} \ \text{DET} \ "\text{the}" \ _ \ _ \ _) = \text{the} \\
\]

**Interpretation of CNs**

\[
\text{intCN} :: \text{Cat} \rightarrow \text{Entity} \rightarrow \text{Bool} \\
\text{intCN} (\text{Ct} \ \text{CN} \ \text{name} \ _ \ _ \ _) = \text{intP name} \\
\]

**Interpretation of Verbs**

Use conversion to a relation for verb interpretations:

\[
\text{intVerb} :: \text{Cat} \rightarrow \text{Rel Entity} \\
\text{intVerb} (\text{Ct} \ \text{VERB} \ \text{name} \ _ \ _ \ _) = \text{intR name} \\
\]

**Interpretation of DP and PP Lists** We intend to interpret the list of DPs and PPs on a Verb-complement list as a function of type \(\text{Rel} \rightarrow \text{Rel}\). The composition \(\text{lift} . \text{intDPorPP}\) lifts the interpretation of a single DP or PP from type \((\text{Entity} \rightarrow \text{Bool}) \rightarrow \text{Bool}\) to type \(\text{Rel Entity} \rightarrow \text{Rel Entity}\). To get our intended interpretation, we proceed as follows:

- The empty list of DPs is interpreted as the identity function on relations.
- If the DP list is nonempty, we want the first DP on the list to work on the first argument of the relation that interprets the VP. We also want the first DP on the list to have wider scope than the other DPs on the list.
The following implementation has these effects.

```plaintext
intDPsorPPs :: [Cat] -> Rel Entity -> Rel Entity
intDPsorPPs [] = id
intDPsorPPs (xp:xps) = ((lift . intDPorPP) xp) . (intDPsorPPs xps)
```

The relation transformer that is applied last will have widest scope, so in fact what we have done is implementing a fixed scoping mechanism where subject outscopes direct object, direct object outscopes indirect object, and so on. In so far as the order of the verb complement list reflects surface word order, our scoping mechanism gives surface scope order. For generating other possible scopings, this needs to be modified: see Section 9.10.

**Interpretation of \( \text{VP} \)**  A simple \( \text{VP} \) consists of a verb and a list of DPs and PPs. This can be interpreted by applying the function that interprets the complement list to the function that interprets the verb. However, we will not do so here, but relegate this to the interpretation of sentences.

**Interpretation of Sentences**  If all goes well, the arity of the relation that interprets the verb concept matches the number of DPs in the sentence (including the subject DP).

To check this, we use the function `arity` on relations.

The result of this check may be that certain parses are rejected at this stage. Therefore `intSent` is implemented as a function of type `Sent -> [Bool]`. If the arity check fails, the empty list is returned. Otherwise, the function returns a unit list with the appropriate truth value.

To get at the truth value, the 0-arity relation that results from applying the interpretation of the DP list to the interpretation of the verb concept gets converted with `decode0`.

```plaintext
intSent :: Cat -> [Bool]
```

For sentences with VPs without auxiliaries, we use the appropriate verb from the lexicon.
9.9 Evaluation

```haskell
code0 ((intDPsorPPs (dp:complement)) (intVerb verb))
| vp <- [head (constituentsC vpbar)],
  VP <- [labelC vp],
  verb <- [head (constituentsC vp)],
  complement <- [tail (constituentsC vp)],
  arity (intVerb verb) == length (dp:complement) ]
```

```haskell
eval :: String -> [[Bool]]
eval string = [ intSent s | s <- parse string ]
```

If the sentence has a parse, and the parse tree satisfies the arity constraints of the verb interpretation, we get a result of the form \([bs]\), where \(bs\) is a non-empty list of booleans.

HRITT> eval "Bill killed Lucy with a stone."
\([\text{False}]\)

This indicates that the sentence has a single parse, and under that parse it interprets as false in the model under consideration.

HRITT> eval "Mary killed Lucy with the gun."
\([\text{True}]\)

This indicates that there is a single parse, and that the sentence is true in the model under that parse.

If a sentence has no parse according to the phrase structure grammar that our parser is based on, this is indicated by \([],\).

HRITT> eval "Bill killed Lucy Ann."
\([]\)

If a sentence has a parse, but the parse tree violates the arity constraints imposed by the semantic evaluator, this is indicated by \([[]]\).
CHAPTER 9. HANDLING RELATIONS IN TYPE THEORY

This sentence has a successful parse, but the semantic evaluator cannot deal with it because the instrument is missing.

9.10 Quantifier Scoping

We show in this section that the ordered list of realized theta roles on a verb concept allows us to formalize and implement a particularly simple and elegant quantifier scoping algorithm. Our approach to scoping can be viewed as a ‘clean’ version of Richard Montague’s *Quantifying In Rule* [Mon73]. The cleanup is made possible by the fact that our relational approach avoids the use of free variables in syntax or semantics.

Our quantifier scoping algorithm can also be viewed as a ‘clean’ version of the quantifier storage approach proposed in [Coo75, Coo83] (cleaner, again, because no free variables are involved).

If we want to consider all different scope orders of \( n \) quantifiers then we have to look at the \( n! = 1 \times \cdots \times n \) different permutations of the quantifiers.

We start out from of a function that gives all the permutations of a finite list:

```haskell
perms :: [a] -> [[a]]
perms [] = [[]]
perms (x:xs) = concat (map (insrt x) (perms xs))
  where
    insrt :: a -> [a] -> [[a]]
    insrt x [] = [[x]]
    insrt x (y:ys) = (x:y:ys) : map (y:) (insrt x ys)
```

This gives:

HRITT> perms [1,2,3]
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]

In our case, where we have verb concepts with realized theta roles together with lists of DPs corresponding to the realized roles, all we have to do is permute the theta role list and the DP list in parallel.

```haskell
permsInPar :: [a] -> [b] -> [[(a),(b)]]
permsInPar xs ys = map unzip (perms (zip xs ys))
```
This gives:

```
HRITT> permsInPar [1,2,3] [5,6,7]
[([1,2,3],[5,6,7]),([2,1,3],[6,5,7]),([2,3,1],[6,7,5]),
  ([1,3,2],[5,7,6]),([3,1,2],[7,5,6]),([3,2,1],[7,6,5])]
```

Use `permsInPar` to generate lists of triples consisting of a list indicating the permutation of surface scope order, the relation under that permutation, and the list of DPs and PPs, again under the same permutation of surface order.

```haskell
permRelDPs :: (Eq a, Bounded a, Enum a) => Rel a -> [Cat] -> [(Int, Rel a, [Cat])]
permRelDPs rel xps =
    [ (perm, extract perm rel, pxps) |
      (perm, pxps) <- permsInPar [1..(length xps)] xps ]
```

The first element of the result triple is a list indicating the way surface scope order got permuted, the second element gives the relation with permuted ‘theta roles’, and the third element gives the permuted XP list.

The scope algorithm will produce a list of pairs consisting of an indication of the way in which surface order got permuted, and a boolean for the truth value of the interpretation of the sentence under this scope ordering.

```haskell
allScopings :: Cat -> [(Int,Bool)]
```

To generate all scopings of a simple sentence, permute the XP list consisting of the subject and the verb complement list in parallel with roles of the relation interpreting the verb, and evaluate the results.

```haskell
allScopings (Ct Sent _ _ _ [dp,vpbar]) =
    [ (perm, decode0 ((intDPsorPPs nxps) newrel)) |
      vp <- [head (constituentsC vpbar)],
      VP <- [labelC vp],
      verb <- [head (constituentsC vp)],
      complement <- [tail (constituentsC vp)],
      arity (intVerb verb) == length (dp:complement),
      (perm, newrel, nxps) <- permRelDPs (intVerb verb) (dp:complement) ]
```
This gives:

```
HRITT> map allScopings (parse "Somebody respected every woman."
[[[(1,2),False],(2,1),True]]
```

This indicates that under the surface scope order reading of the DPs the sentence is false in the example model, but under the reverse scope order reading of the DPs ("For every woman there is a someone who respects her") the sentence is true.

```
HRITT> map allScopings (parse "Nobody sold a thing to every woman."
[[[(1,2,3),True],(2,1,3),True],(2,3,1),True],
    [(1,3,2),True],(3,1,2),False],(3,2,1),True)]
```

**Exercise 9.2** (Open ended) We only have the rudiments of a semantic treatment. Extend the semantic coverage to a larger part of the syntactic fragment.

**Exercise 9.3** (Open ended) In our treatment of the semantics, we do direct interpretation in the model. As an alternative to this, implement a translation function from categories to logical forms.

### 9.11 Further Reading

In [Mus89a, Mus89b] it is shown that relations instead of functions can be taken as the basic ingredients of type theory. Our approach is slightly different, for our implementation of relations is based on recursive datatypes. Recursive datatypes are treated in any textbook on functional programming. See, e.g., [RL99] for an account of recursive datatypes in Haskell.
Chapter 10

A Closer Look at Lambda Calculus

Summary

In this chapter we take a closer look at substitution and reduction of lambda terms to normal form. An implementation in Haskell is provided to illustrate the principles involved.

Note: full documentation and explanation of the code below still to be written . . .

10.1 Untyped Lambda Calculus — Datatype Declaration

We declare a module LC, and a datatype Lam for untyped lambda terms, with variables represented as integers.

(Abs 1 (Abs 2 (Var 1))) represents the term \( \lambda x_1 \lambda x_2 . x_1 \), (Abs 1 (Var 1)) represents \( \lambda x_1 . x_1 \), (App (Abs 1 (Var 1)) (Var 5)) represents \( (\lambda x_1 . x_1) x_5 \), and so on.

```haskell
module LC

where

data Lam = Var Int | App Lam Lam | Abs Int Lam deriving (Eq, Show)
```

(isfree x e) is true iff x has a free occurrence in e.
isfree :: Int -> Lam -> Bool
isfree x (Var y) = x == y
isfree x (App e f) = isfree x e || isfree x f
isfree x (Abs v body) = x /= v && isfree x body

(firstfreshindex list) computes the smallest integer not in list.

firstfreshindex :: [Int] -> Int
firstfreshindex s = find 0 s
    where find x s | x `elem` s = find (x+1) s
                      | otherwise = x

(frees e) collects the free variables of e in a list

frees :: Lam -> [Int]
frees (Var v) = [v]
frees (App e f) = (frees e) ++ (frees f)
frees (Abs v body) = difference (frees body) [v]

list difference

difference [] t = []
difference (s:ss) t | elem s t = difference ss t
    | otherwise = s:(difference ss t)

10.2 Substitution

(subst x e f) computes f[x := e] changing bound variables as the need arises.
10.3. NOTIONS OF REDUCTION FOR LAMBDA CALCULUS

\[
\text{subst} :: \text{Int} \to \text{Lam} \to \text{Lam} \to \text{Lam}
\]
\[
\begin{align*}
\text{subst} \ x \ e \ f@(\text{Var} \ v) & = \begin{cases} 
\text{e} & \text{if } x = v \\
\text{f} & \text{otherwise}
\end{cases} \\
\text{subst} \ x \ e \ (\text{App} \ f \ g) & = \text{App} \ (\text{subst} \ x \ e \ f) \ (\text{subst} \ x \ e \ g) \\
\text{subst} \ x \ e \ f@(\text{Abs} \ \text{bndvar} \ \text{body}) & = \begin{cases} 
\text{f} & \text{if not } \text{isfree} \ x \ f \\
\text{Abs} \ \text{bndvar} \ (\text{subst} \ x \ e \ \text{body}) & \text{if not } \text{isfree} \ \text{bndvar} \ e \\
\text{Abs} \ z \ (\text{subst} \ x \ e \ (\text{subst} \ \text{bndvar} \ (\text{Var} \ z) \ \text{body})) & \text{otherwise}
\end{cases}
\end{align*}
\]

where \( z = \text{firstfreshindex} \ ((\text{frees} \ e) ++ (\text{frees} \ f)) \)

10.3 Notions of Reduction for Lambda Calculus

\( (\alpha\text{RedNotion} \ e \ f) \) is true iff \( e \) and \( f \) are both abstractions and \( e \) and \( f \) are alphabetic variants

\[
\alpha\text{RedNotion} :: \text{Lam} \to \text{Lam} \to \text{Bool}
\]
\[
\begin{align*}
\alpha\text{RedNotion} \ (\text{Abs} \ \text{bndvar} \ \text{body}) \ (\text{Abs} \ \text{bndvar'} \ \text{body'}) & = \\
\text{body'} & = \text{subst} \ \text{bndvar} \ (\text{Var} \ \text{bndvar'}) \ \text{body}
\end{align*}
\]
\[\alpha\text{RedNotion} \_ \_ = \text{False}\]

\( (\alpha\text{Variants} \ e \ f) \) is true iff \( e \) and \( f \) are alphabetic variants

\[
\alpha\text{Variants} :: \text{Lam} \to \text{Lam} \to \text{Bool}
\]
\[
\begin{align*}
\alpha\text{Variants} \ (\text{Var} \ i) \ (\text{Var} \ j) & = i == j \\
\alpha\text{Variants} \ (\text{Abs} \ \text{bndvar} \ \text{body}) \ (\text{Abs} \ \text{bndvar'} \ \text{body'}) & = \\
\alpha\text{Variants} \ \text{body} \ (\text{subst} \ \text{bndvar'} \ (\text{Var} \ \text{bndvar}) \ \text{body'}) \\
\alpha\text{Variants} \ (\text{App} \ \text{term1} \ \text{term2}) \ (\text{App} \ \text{term3} \ \text{term4}) & = \\
\alpha\text{Variants} \ \text{term1} \ \text{term3} \ \&\& \alpha\text{Variants} \ \text{term2} \ \text{term4} \\
\alpha\text{Variants} \_ \_ & = \text{False}
\end{align*}
\]

\( (\beta\text{RedNotion} \ e \ f) \) is true iff \( e = (\tilde{m}) \ n \) and \( f = m[v:=n] \).
\[
\text{betaRedNotion} :: \text{Lam} \rightarrow \text{Lam} \rightarrow \text{Bool} \\
\text{betaRedNotion} (\text{App} (\text{Abs} \ v \ \text{m}) \ n) \ f = f == \text{subst} \ v \ n \ \text{m} \\
\text{betaRedNotion} \ _\ _ = \text{False}
\]

\(\text{etaRedNotion} \ e \ f\) is true iff \(e == (\bar{f}) \ v\) and \(v\) not among the free variables of \(f\).

\[
\text{etaRedNotion} :: \text{Lam} \rightarrow \text{Lam} \rightarrow \text{Bool} \\
\text{etaRedNotion} (\text{Abs} \ \text{var} (\text{App} \ f (\text{Var} \ \text{var}'))) \ f' = \\
\quad \text{var} == \text{var}' \land \text{not} (\text{isfree} \ \text{var} \ f) \land f == f' \\
\text{etaRedNotion} \ _\ _ = \text{False}
\]

### 10.4 Normal Forms

\(\text{normalform} \ e\) is true iff \(e\) is in normal form i.e., \(e\) is an expression without redexes.

\[
\text{normalform} :: \text{Lam} \rightarrow \text{Bool} \\
\text{normalform} (\text{Var} \ v) = \text{True} \\
\text{normalform} (\text{Abs} \ v \ \text{body}@\text{App} \ e (\text{Var} \ \text{w})) \\
\quad | \ v == \text{w} \land \text{not} (\text{isfree} \ \text{w} \ \text{frees} \ e) = \text{False} \\
\quad | \ \text{otherwise} \quad = \text{normalform} \ \text{body} \\
\text{normalform} (\text{Abs} \ v \ \text{body}) = \text{normalform} \ \text{body} \\
\text{normalform} (\text{App} \ (\text{Var} \ v) \ e) = \text{normalform} \ e \\
\text{normalform} (\text{App} \ (\text{Abs} \ v \ \text{body}) \ e) = \text{False} \\
\text{normalform} (\text{App} \ (\text{App} \ e \ f) \ g) = \text{normalform} (\text{App} \ e \ f) \land \text{normalform} \ g
\]

\(\text{startswithvar} \ e\) checks whether \(e\) starts with a variable. This check is used in the checks for head normal form and weak head normal form.
10.5. Reductions to Normal Form

We use a representation of a term as a triple consisting of a core function, a binder list and an argument list. Binder list and argument list are used as accumulators in the normal form reduction procedure.

NB reduction is in applicative order: arguments are normalized before they are put on the argument stack.
nf :: Lam -> Lam
nf term = nf_ba term [] []

nf_ba :: Lam -> [Int] -> [Lam] -> Lam
nf_ba (Var v) vars args = compose (Var v) vars args
nf_ba (Abs v body) vars [] = nf_ba body (v:vars) []
nf_ba (Abs v body) vars (f:args) = nf_ba (subst v f body) vars args
nf_ba (App e f) vars args = nf_ba e vars ((nf f):args)

Compose a term from its representation as core function term + var stack + arg stack. Note
that the procedure ensures that the variable bindings have scope over the function argument
applications.

compose term [] [] = term
compose term (v:vars) [] = compose (Abs v term) vars []
compose term vars (a:args) = compose (App term a) vars args

Normal form reduction using accumulators for binder list and argument list, but now we reduce
in left-to-right order: Arguments are normalized after normalization of the core function.

nf_lr :: Lam -> Lam
nf_lr term = nf_lr_ba term [] []

nf_lr_ba :: Lam -> [Int] -> [Lam] -> Lam
nf_lr_ba (Var v) vars args = compose_nf (Var v) vars args
  where
    compose_nf term [] [] = term
    compose_nf term (v:vars) [] =
      compose_nf (Abs v term) vars []
    compose_nf term vars (a:args) =
      compose_nf (App term (nf_lr a)) vars args
nf_lr_ba (Abs v body) vars [] = nf_lr_ba body (v:vars) []
nf_lr_ba (Abs v body) vars (f:args) = nf_lr_ba (subst v f body) vars args
nf_lr_ba (App e f) vars args = nf_lr_ba e vars (f:args)
10.6. SOME WELL-KNOWN COMBINATORS

Head normal form reduction using accumulators for binder list and argument list.

\[
\text{hnf} :: \text{Lam} \rightarrow \text{Lam} \\
\text{hnf} \ \text{term} = \text{hnf}_\text{ba} \ \text{term} \ [] \ []
\]

\[
\text{hnf}_\text{ba} :: \text{Lam} \rightarrow [\text{Int}] \rightarrow [\text{Lam}] \rightarrow \text{Lam} \\
\text{hnf}_\text{ba} (\text{Var} \ v) \ \text{vars} \ \text{args} = \text{compose} (\text{Var} \ v) \ \text{vars} \ \text{args} \\
\text{hnf}_\text{ba} (\text{Abs} \ v \ \text{body}) \ \text{vars} \ [] = \text{hnf}_\text{ba} \ \text{body} \ (v:\text{vars}) \ [] \\
\text{hnf}_\text{ba} (\text{Abs} \ v \ \text{body}) \ \text{vars} \ (f:\text{args}) = \text{hnf}_\text{ba} \ (\text{subst} \ v \ f \ \text{body}) \ \text{vars} \ \text{args} \\
\text{hnf}_\text{ba} (\text{App} \ e \ f) \ \text{vars} \ \text{args} = \text{hnf}_\text{ba} \ e \ \text{vars} \ (f:\text{args})
\]

Weak head normal form reduction using accumulators for binder list and argument list

\[
\text{whnf} :: \text{Lam} \rightarrow \text{Lam} \\
\text{whnf} \ \text{term} = \text{whnf}_\text{ba} \ \text{term} \ [] \ []
\]

\[
\text{whnf}_\text{ba} :: \text{Lam} \rightarrow [\text{Int}] \rightarrow [\text{Lam}] \rightarrow \text{Lam} \\
\text{whnf}_\text{ba} (\text{Var} \ v) \ \text{vars} \ \text{args} = \text{compose} (\text{Var} \ v) \ \text{vars} \ \text{args} \\
\text{whnf}_\text{ba} (\text{Abs} \ v \ \text{body}) \ \text{vars} \ [] = \text{whnf}_\text{ba} \ (\text{Abs} \ v \ \text{body}) \ \text{vars} \ [] \\
\text{whnf}_\text{ba} (\text{Abs} \ v \ \text{body}) \ \text{vars} \ (f:\text{args}) = \text{whnf}_\text{ba} \ (\text{subst} \ v \ f \ \text{body}) \ \text{vars} \ \text{args} \\
\text{whnf}_\text{ba} (\text{App} \ e \ f) \ \text{vars} \ \text{args} = \text{whnf}_\text{ba} \ e \ \text{vars} \ (f:\text{args})
\]

10.6 Some Well-known Combinators

k, i, s and omega are well known combinators (closed lambda terms).

\[
k = (\text{Abs} \ 1 \ (\text{Abs} \ 2 \ (\text{Var} \ 1))) \\
i = (\text{Abs} \ 1 \ (\text{Var} \ 1)) \\
s = (\text{Abs} \ 1 \ (\text{Abs} \ 2 \ (\text{Abs} \ 3 \ (\text{App} \ (\text{App} \ (\text{Var} \ 1) \ (\text{Var} \ 3)) \ (\text{App} \ (\text{Var} \ 2) \ (\text{Var} \ 3))))))) \\
\text{omega} = (\text{App} \ (\text{Abs} \ 1 \ (\text{App} \ (\text{Var} \ 1) \ (\text{Var} \ 1))) \\
\ \ (\text{Abs} \ 1 \ (\text{App} \ (\text{Var} \ 1) \ (\text{Var} \ 1))))
\]
y is the fixed point combinator

\[
y = (\text{Abs } 1 (\text{App } (\text{Abs } 2 (\text{App } (\text{Var } 1) (\text{App } (\text{Var } 2) (\text{Var } 2)))) (\text{Abs } 2 (\text{App } (\text{Var } 1) (\text{App } (\text{Var } 2) (\text{Var } 2))))))
\]

Some complex combinators:

\[
\begin{align*}
\text{skk} &= (\text{App } (\text{App } s k) k) \\
\text{sk} &= (\text{App } s k) \\
\text{b} &= (\text{App } (\text{App } s (\text{App } k s)) k) \\
\text{ik} &= (\text{App } i k) \\
\text{b'} &= (\text{App } (\text{App } (\text{App } b (\text{Var } 1)) (\text{Var } 2)) (\text{Var } 3))
\end{align*}
\]

10.7 Church Numerals

\[
\begin{align*}
\text{cn } n &= (\text{Abs } 1 (\text{Abs } 0 (\text{cn\_body } n))) \\
\text{cn\_body } 0 &= (\text{Var } 0) \\
\text{cn\_body } (n+1) &= (\text{App } (\text{Var } 1) (\text{cn\_body } n)) \\
\text{one} &= \text{cn } 1 \\
\text{two} &= \text{cn } 2 \\
\text{three} &= \text{cn } 3 \\
\text{four} &= \text{cn } 4
\end{align*}
\]

Mapping Church Numerals to integers:

\[
\begin{align*}
\text{number } (\text{Abs } 1 (\text{Abs } 0 (\text{Var } 0))) &= 0 \\
\text{number } (\text{Abs } 1 (\text{Abs } 0 (\text{App } (\text{Var } 1) \text{ term}))) &= \\
&= (\text{number } (\text{Abs } 1 (\text{Abs } 0 \text{ term}))) + 1
\end{align*}
\]
10.8. PRETTY PRINTING OF LAMBDA TERMS

Operations on Church numerals:

\[
\begin{align*}
\text{plus} &= (\text{Abs} \ 1 \ (\text{Abs} \ 2 \ (\text{Abs} \ 3 \ (\text{Abs} \ 4 \\
&\quad \quad \quad (\text{App} \ (\text{App} \ \text{Var} \ 1 \ \text{Var} \ 3) \\
&\quad \quad \quad \quad \text{App} \ (\text{App} \ \text{Var} \ 2 \ \text{Var} \ 3) \ \text{Var} \ 4)))))) \\
\text{expo} &= (\text{Abs} \ 1 \ (\text{Abs} \ 2 \ (\text{App} \ \text{Var} \ 2 \ \text{Var} \ 1))) \\
\text{expo22} &= (\text{App} \ (\text{App} \ \text{expo} \ \text{two} \ \text{two}) \\
\text{expo33} &= (\text{App} \ (\text{App} \ \text{expo} \ \text{three} \ \text{three})
\end{align*}
\]

10.8 Pretty Printing of Lambda Terms

We will print lambda terms as \( v \) or \( M_1 \cdots M_m \) (with \( m > 0 \)), or \( \lambda x_1 \cdots x_n.v \) (with \( n \geq 1 \)), using \( ^\_ \) for \( \lambda \), or \( \lambda x_1 \cdots x_n.(M_1 \cdots M_m) \), with \( n \geq 1, m > 1 \). Application associates to the left, so \( MPQ \) is shorthand for \( ((MP)Q) \), but we need brackets in \( M(PQ) \). Thus, \( M_1 \cdots M_m \) is read as \( \cdots (M_1M_2)...M_m \). Here is a CF grammar that produces this format:

\[
\begin{align*}
\text{Lam} &::= v \mid ^\_v \text{Lam1} \mid \text{Lam Lam2} \\
\text{Lam1} &::= .v \mid v \text{Lam1} \mid .(\text{Lam Lam2}) \\
\text{Lam2} &::= v \mid ^\_v \text{Lam1} \mid (\text{Lam Lam2})
\end{align*}
\]

Lam produces single variables, abstractions with an explicit binder \( ^\_ \), and applications without parentheses.

Lam1 produces abstractions without explicit binder \( ^\_ \), single variables preceded by a dot, and applications with parentheses, preceded by a dot.

Lam2 produces single variables, abstractions with an explicit binder, and applications in parentheses.

The print procedures showsLam, showsLam1, and showsLam2 follow the rules above.
CHAPTER 10. A CLOSER LOOK AT LAMBDA CALCULUS

```haskell
showLam :: Lam -> String
showLam term = showsLam term ""

-- showsLam : print with explicit binder, without outer parentheses

showsLam :: Lam -> String -> String
showsLam (Abs v term) string = '^': shows v (showsLam1 term string)
showsLam (App e f) string = showsLam e (showsLam2 f string)
showsLam (Var v) string = shows v string

-- showsLam1 : abstractions without and applications with outer parentheses

showsLam1 :: Lam -> String -> String
showsLam1 (Abs v term) string = shows v (showsLam1 term string)
showsLam1 (App e f) string = '.' : '(' : showsLam e (showsLam2 f (')': string))
showsLam1 (Var v) string = '.': shows v string

-- showsLam2 : abstractions and applications with outer parentheses

showsLam2 :: Lam -> String -> String
showsLam2 (Abs v term) string =
    '(' :'^': shows v (showsLam1 term (')': string))
showsLam2 (App e f) string = '(' : showsLam2 e (showsLam2 f (')':string))
showsLam2 (Var v) string = shows v string
```

We can use showLam to show normal forms:

```haskell
snf :: Lam -> String
snf = showLam . nf
```

10.9 Further Reading

The bible of lambda calculus is Barendregt [Bar84]. A less encyclopedic account can be found in [Han94].
Chapter 11

Types for Lambda Terms

Summary

An algorithm for computing most general type schemes is presented and illustrated by means of Haskell code.

Note: full documentation and explanation of the code below still to be written …

11.1 Typing of Lambda Terms

The module Typing uses the module Lambda. We declare a datatype Type for type schemes. Because we will encounter lambda terms that cannot be typed, we include a constructor Notype to indicate this failure to find a type.

module Typing
where
import LC

data Type = Notype
  | Uvar Int
  | Fct Type Type deriving (Eq,Show)
11.2 Pretty Printing of Type Schemes

We use the following CF rules for printing of type schemes:

```
Type ::= Int | Type1 -> Type
Type1 ::= Int | (Type1 -> Type)
```

These rules express that \( \rightarrow \) associates to the right, i.e. \((a \rightarrow (b \rightarrow c))\) gets printed as \(a \rightarrow b \rightarrow c\), \(((a \rightarrow b) \rightarrow c)\) gets printed as \((a \rightarrow b) \rightarrow c\).

Indices 0, 1, 2 .. get printed as 'a', 'b', 'c' ..

```haskell
showType :: Type -> String
showType t = showsType t ""

showsType :: Type -> String -> String
showsType Notype string = "<< no type >>"
showsType (Uvar i) string = chr(i + 97):string
showsType (Fct t1 t2) string =
   showsType1 t1 (' ':'-':'>':' ':(showsType t2 string))

showsType1 :: Type -> String -> String
showsType1 Notype string = "<< no type >>"
showsType1 (Uvar i) string = chr(i + 97):string
showsType1 (Fct t1 t2) string =
   ('(':showsType1 t1 (' ':'-':>'':':':(showsType t2 ('')':string))))
```
11.3 Occurs Check for Type Schemes

\[
\text{occurs\_in\_type :: Int} \to \text{Type} \to \text{Bool} \\
\text{occurs\_in\_type i t = occurs\_in\_types i [t]} \\
\]

\[
\text{occurs\_in\_types :: Int} \to \text{[Type]} \to \text{Bool} \\
\text{occurs\_in\_types i [] = False} \\
\text{occurs\_in\_types i (Notype:ts) = occurs\_in\_types i ts} \\
\text{occurs\_in\_types i ((Uvar j):ts) = i == j || occurs\_in\_types i ts} \\
\text{occurs\_in\_types i ((Fct t1 t2):ts) = occurs\_in\_types i (t1:t2:ts)}
\]

11.4 Typed Lambda terms

We allow for the possibility of constants (Cst) and placeholders (Phr) in terms. Only the types of the variables, constants and placeholders are explicitly given.

\[
data \text{Term a = Tvar a} \\
\mid \text{Phr a} \\
\mid \text{Cst a} \\
\mid \text{Tapp (Term a) (Term a)} \\
\mid \text{Tabs a (Term a)} \\
deriving (\text{Eq, Show})
\]

\[
type \text{It = (Int, Type)} \\
type \text{Tlt = Term It}
\]

11.5 Pretty Printing of Typed Terms

We print variables as v0\_type, v1\_type, ..., placeholders as p0\_type, p1\_type, ..., constants as c0\_type, c1\_type ..., and suppress all other type information.

In printing typed indices, we just show the index.
showsIt :: It -> String -> String
showsIt (i,t) string = shows i string

The format for the printing of typed lambda terms is as for untyped lambda terms.

showTlt :: Tlt -> String
showTlt term = showsTlt term ""

-- showsTlt : print with explicit binder, without outer parentheses

showsTlt :: Tlt -> String -> String
showsTlt (Tabs it term) string =
    '^':'v':(showsIt it (showsTlt1 (term) string))
showsTlt (Tapp e f) string = showsTlt e (showsTlt2 f string)
showsTlt (Tvar it) string = 'v':(showsIt it string)
showsTlt (Phr it) string = 'p':(showsIt it string)
showsTlt (Cst it) string = 'c':(showsIt it string)

-- showsTlt1 : abstractions without and applications with outer parentheses

showsTlt1 :: Tlt -> String -> String
showsTlt1 (Tabs it term) string =
    '"':'^':'v':(showsIt it (showsTlt1 term string))
showsTlt1 (Tapp e f) string = ',' : '(' : showsTlt e
    (showsTlt2 f (')': string))
showsTlt1 (Tvar it) string = '.':'v':(showsIt it string)
showsTlt1 (Phr it) string = '.':'p':(showsIt it string)
showsTlt1 (Cst it) string = '.':'c':(showsIt it string)

-- showsTlt2 : abstractions and applications with outer parentheses

showsTlt2 :: Tlt -> String -> String
showsTlt2 (Tabs it term) string =
    '(':'^':'v':(showsIt it (showsTlt1 (term) (')': string)))
showsTlt2 (Tapp e f) string = '(' : showsTlt2 e (showsTlt2 f (')': string))
showsTlt2 (Tvar it) string = 'v':(showsIt it string)
showsTlt2 (Phr it) string = 'p':(showsIt it string)
showsTlt2 (Cst it) string = 'c':(showsIt it string)
11.6 Environments, Constraints, Substitutions and Judgements

An environment allows to judge the type of an unbound variable. The initial environment maps unbound variable i to type variable i.

A constraint on type schemes is an identity $t_1 \approx t_2$, where $t_1$ and $t_2$ are type schemes. Constraints are represented as $[(\text{Type}, \text{Type})]$.

A substitution for type schemes is a list of variable/type pairs. A substitution $\sigma$ is denoted as $u_i/t_1, \ldots, u_j/t_n$.

Type inferencing is the process of deriving type judgements.

A type judgement is a statement about the typing of a term $M$, given an environment, and a set of constraints on type scheme identities. Every type judgement carries as an extra parameter an integer for the first free type index.

```haskell
type Env = [It]
type Constraints = [(Type,Type)]
type Subst = [(Int,Type)]
type Judgement = (Lam, Type, Env, Constraints, Int)
```

11.7 Initialisation of Environments

```haskell
initEnv :: Lam -> Env
initEnv term = [(i,(Uvar i)) | i <- frees term]

lookupEnv :: Int -> Env -> Type
lookupEnv i [] = Notype
lookupEnv i ((j,t):env) | i == j = t
                        | otherwise = lookupEnv i env
```

11.8 Constraint Resolution by Means of Unification

Constraint resolution takes place according to the following rules (where $C$ is a set of constraints, and $\sigma$ a substitution).
\[
\frac{u_i \approx u_j \land C \parallel \sigma}{C \parallel \sigma} \quad i = j
\]

\[
\frac{u_i \approx u_j \land C \parallel \sigma}{C[u_i := u_j] \parallel \sigma[u_i := u_j] \land u_i/u_j} \quad i \neq j
\]

\[
\frac{u_i \approx t \land C \parallel \sigma}{C[u_i := t] \parallel \sigma[u_i := t] \land u_i/t} \quad u_i \text{ does not occur in } t
\]

\[
\frac{t \approx u_i \land C \parallel \sigma}{C[u_i := t] \parallel \sigma[u_i := t] \land t_1/t} \quad u_i \text{ does not occur in } t
\]

\[
\frac{t_1 \rightarrow t_2 \approx t_3 \rightarrow t_4 \land C \parallel \sigma}{C \land t_1 \approx t_3 \land t_2 \approx t_4 \parallel \sigma}
\]

The rules indicate that unification should be a map from constraint/substitution pairs to constraint/substitution pairs. Since unification can fail, we also need a Boolean output parameter to indicate success.

```haskell
resolveCS :: (Constraints,Subst) -> (Constraints,Subst,Bool)
resolveCS ([], sb) = ([], sb, True)
resolveCS (((Uvar i),(Uvar j)):cs , sb)
  | i == j = resolveCS (cs, sb)
  | otherwise = resolveCS ((bindCs i (Uvar j) cs), ((i, (Uvar j)):sb'))
    where sb' = bindSb i (Uvar j) sb
resolveCS (((Uvar i),t):cs , sb)
  | occurs_in_type i t = ([],[],False)
  | otherwise = resolveCS ((bindCs i t cs), ((i,t):sb'))
    where sb' = bindSb i t sb
resolveCS ((t,(Uvar i)):cs , sb)
  | occurs_in_type i t = ([],[],False)
  | otherwise = resolveCS ((bindCs i t cs), ((i,t):sb'))
    where sb' = bindSb i t sb
resolveCS (((Fct t1 t2),(Fct t3 t4)):cs , sb)
  = resolveCS (((t1,t3):(t2,t4):cs), sb)
```

11.9 Binding of Type Variables to Types

- in types
11.10 Type Inference Rules

The type inference rules look like this:

\[
\frac{E(x) = \alpha \land C}{E \vdash x : \alpha \parallel C}
\]

\[
\frac{E \vdash M : \beta \rightarrow \alpha \parallel C \quad E \vdash N : \beta \parallel C}{E \vdash (MN) : \alpha \parallel C}
\]

\[
\frac{E, x : \beta \vdash M : \gamma \parallel \alpha = \beta \rightarrow \gamma \land C}{E \vdash \lambda x.M : \alpha \parallel C}
\]
If these rules are read bottom-up, they can be used to collect a set of constraints that an initial typing for a term $M$ has to satisfy.

```haskell
infer :: Judgement -> Judgement
infer ((Var i), t, env, cs, k) = 
  ((Var i), t, env, ((lookupEnv i env), t):cs, k)
infer ((App e f), t, env, cs, k) = 
  infer (e,
    (Uvar (k'+1)),
    env,
    ((Uvar (k'+1)),(Fct (Uvar (k+1)) (Uvar k))):cs',
    k'+1)
  where (_, _, _, cs', k') = infer (f, (Uvar (k+1)), env, cs, k+1)
infer ((Abs i body), t, env, cs, k) = 
  infer (body,
    (Uvar (k+2)),
    ((i,(Uvar (k+1))):env),
    (((Fct (Uvar (k+1)) (Uvar (k+2))),(Uvar k)):cs),
    k+2)
```

### 11.11 Computation of Most General Type Schemes for Lambda Terms

To avoid a clash with the mapping provided by the initial environment we have to choose our indices outside the set of indices of free variables of our untyped lambda term.

$(\text{freshindex}\ \text{list})$ gives the first index above every element in list.

```haskell
freshindex :: [Int] -> Int
freshindex [] = 0
freshindex (i:is) = (maximum (i:is)) + 1
```
11.11. COMPUTATION OF MOST GENERAL TYPE SCHEMES FOR LAMBDA TERMS

```haskell
giveType :: Lam -> Subst -> Type
giveType term sb = lookupIndex k sb
  where
    k = freshindex (frees term)

lookupIndex :: Int -> Subst -> Type
lookupIndex i [] = Notype
lookupIndex i ((j,t):sb) | i == j = t
  | otherwise = lookupIndex i sb
```

(mgt term) gives the most general type of a term.

```haskell
mgt :: Lam -> Type
mgt term | result == False = Notype
  | otherwise = lookupIndex k sb
  where
    (_,sb,result) = resolveCS (cs, [])
    (_, _, _, cs, _) = infer
      (term, (Uvar (k+1)), (initEnv term),[([Uvar k),(Uvar (k+1))]), (k+1))
    k = freshindex (frees term)
```

```haskell
givesub :: Lam -> Constraints
givesub term = cs
  where
    (_, _, _, cs, _) = infer (term, (Uvar k), (initEnv term), [], k)
    k = freshindex (frees term)
```

```haskell
resolve :: Constraints -> Subst
resolve cs = sb where (_,sb,_) = resolveCS (cs,[])
```

```haskell
res :: Lam -> Subst
res term = sb where sb = resolve (givesub term)
```

Use `showType` to print most general types.
11.12 Substitution for Typed Lambda Terms

(newindexForType t list) computes the smallest integer with (i,t) not in list.

newindexForType :: Type -> [It] -> Int
newindexForType t its = firstfreshindex [ i | (i,t) <- its ]

(freeVarsTlt e) collects the free variables of e in a list

freeVarsTlt :: Tlt -> [It]
freeVarsTlt (Tvar it) = [it]
freeVarsTlt (Phr it) = []
freeVarsTlt (Cst it) = []
freeVarsTlt (Tapp e f) = (freeVarsTlt e) ++ (freeVarsTlt f)
freeVarsTlt (Tabs it body) = difference (freeVarsTlt body) [it]

(phrsTlt e) collects the placeholders of e in a list

phrsTlt :: Tlt -> [It]
phrsTlt (Tvar it) = []
phrsTlt (Phr it) = [it]
phrsTlt (Cst it) = []
phrsTlt (Tapp e f) = (phrsTlt e) ++ (phrsTlt f)
phrsTlt (Tabs it body) = difference (phrsTlt body) [it]

(isfreeTlt it e) is true iff it has a free occurrence in e.
11.13. FROM UNTYPED TERMS TO TYPED TERMS

(isfreeTlt :: It -> Tlt -> Bool
isfreeTlt it (Tvar it') = it == it'
isfreeTlt it (Phr it') = False
isfreeTlt it (Cst it') = False
isfreeTlt it (Tapp e f) = isfreeTlt it e || isfreeTlt it f
isfreeTlt it (Tabs it' body) = it /= it' && isfreeTlt it body

(substTlt x e f) computes f[x := e] for typed f, e changing bound variables as the need arises.

(substTlt :: It -> Tlt -> Tlt -> Tlt
substTlt it e f@(Tvar it')
  | it == it' = e
  | otherwise = f
substTlt it e f@(Phr it') = f
substTlt it e f@(Cst it') = f
substTlt it e (Tapp f g) = Tapp (substTlt it e f) (substTlt it e g)
substTlt it e f@(Tabs it' body)
  | not (isfreeTlt it f) = f
  | not (isfreeTlt it' e) = Tabs it' (substTlt it e body)
  | otherwise =
    Tabs new (substTlt it e (substTlt it' (Tvar new) body))
where k = newindexForType t ((freeVarsTlt e) ++ (freeVarsTlt f))
  t = snd it'
  new = (k,t)

11.13  From Untyped Terms to Typed Terms

(initTlt t k)
**CHAPTER 11. TYPES FOR LAMBDA TERMS**

```haskell
initTlt :: Lam -> Int -> (Tlt,Int)
initTlt (Var i) k = ((Tvar (i,(Uvar i))), k)
initTlt (App e f) k = ((Tapp term1 term2), k2)
  where
  (term1,k1) = initTlt f (k+1)
  (term2,k2) = initTlt e (k1+1)
initTlt (Abs i b) k = ((Tabs (i,(Uvar (k+1))) term), (k'+1))
  where
  (term,k') = initTlt b (k+2)

applySubst :: Subst -> Tlt -> Tlt
applySubst sb (Tvar (i,t)) | t' /= Notype = (Tvar (i,t'))
  | otherwise = (Tvar (i,t))
  where t' = lookupIndex i sb
applySubst sb (Cst it) = Cst it
applySubst sb (Phr it) = Phr it
applySubst sb (Tapp e f) = Tapp (applySubst sb e) (applySubst sb f)
applySubst sb (Tabs (i,t) b) | t' /= Notype =
  Tabs (i,t') (applySubst ((i,t):sb) b)
  | otherwise =
  Tabs (i,t) (applySubst ((i,t):sb) b)
  where t' = lookupIndex i sb

lamToTlt :: Lam -> Tlt
lamToTlt term = applySubst sb tlt
  where
  (tlty,_) = initTlt term (k+1)
  (_,sb,result) = resolveCS (cs, [])
  (_, _, _, cs, _) = infer
  (term, (Uvar (k+1)), (initEnv term),[((Uvar k),(Uvar (k+1))未来的], (k+1))
  k = freshindex (frees term)

st t = showsTlt tlt (''::(smgt t))
  where tlt = lamToTlt t
```
11.14 Further Reading

An algorithm for computing most general type schemes is given in [DM82]. The mechanism for computing the types of Haskell functions is explained in [Jon99].
Chapter 12

Dynamic Semantics for NL

Summary

This chapter gives a brief sketch of dynamic semantics for NL modelled after Dynamic Predicate Logic, with a fragment implemented in the functional programming language Haskell.

12.1 Point of Departure: DPL

Dynamic Predicate Logic or DPL [GS91] is our point of departure. Assuming a first order model $M = (D, I)$ and a set of variables $V$, states for DPL are functions in $D^V$.

Given a model $M = (D, I)$ and a state $s \in D^V$, we interpret terms of the language by means of $[c]_s^M = I(c)$ for constants $c$ and $[v]_s^M = s(v)$ for variables $v$. This in turn allows us to define the relation

$$M \models_s P t_1 \cdots t_n$$

by means of:

$$M \models_s P t_1 \cdots t_n : \Leftrightarrow \langle [t_1]_s^M, \ldots, [t_n]_s^M \rangle \in I(P),$$

and the relation

$$M \models_s t_1 = t_2$$

by means of:

$$M \models_s t_1 = t_2 : \Leftrightarrow [t_1]_s^M = [t_2]_s^M.$$

If $x \in V, d \in D$ and $s \in D^V$, we use $(x|d)s$ for the state $s'$ that differs from $s$ at most in the fact that $x$ gets mapped to $d$.

The DPL interpretation of formulas can now be given as a map in $D^V \to \mathcal{P}(D^V)$:
\[
\begin{align*}
\left[\exists x\right](s) & := \{(x|d)s \mid d \in D\} \\
\left[Pt_1 \cdots t_n\right](s) & := \begin{cases} 
\{s\} & \text{if } M \models_s Pt_1 \cdots t_n \\
\emptyset & \text{otherwise,}
\end{cases} \\
\left[t_1 \equiv t_2\right](s) & := \begin{cases} 
\{s\} & \text{if } M \models_s t_1 \equiv t_2 \\
\emptyset & \text{otherwise,}
\end{cases} \\
\left[\neg \phi\right](s) & := \begin{cases} 
\{s\} & \text{if } \left[\phi\right](s) = \emptyset, \\
\emptyset & \text{otherwise,}
\end{cases} \\
\left[\phi ; \psi\right](s) & := \bigcup \left\{\left[\psi\right](s') \mid s' \in \left[\phi\right](s)\right\}
\end{align*}
\]

Note that predicates, identities and negations are interpreted as tests: if \(s\) is the input state they either return \(\{s\}\), in case the test succeeds, or \(\emptyset\), in case the test fails.

Dynamic implication \(\phi \Rightarrow \psi\) can be defined in terms of \(\neg\) and \(\;\). Universal quantification \(\forall x \phi\) is defined in terms of \(\exists\), \(\neg\), and \(\;\); as \(\neg(\exists x ; \neg \phi)\).

The advantage of the propagation of variable states is that they carry anaphoric information that can be used for the interpretation of subsequent discourse.

12.1 Some\(^1\) man loved some\(^2\) woman.

The DPL rendering of (12.1) is \(\exists u_1 ; Mu_1 ; \exists u_2 ; Wu_2 ; Lu_1 u_2\). This gets interpreted as the set of all maps \(u_1 \mapsto e_1, u_2 \mapsto e_2\) that satisfy the relation ‘love’ in the model under consideration. The result of this is that the subsequent sentence (12.2) can now use this discourse information to pick up the references:

12.2 He\(_1\) kissed her\(_2\).

12.2 Extensions to Typed Logic

Attempts to incorporate DPL stype dynamic semantics in mainstream Montague style natural language semantics [Mon73] can be found in [GS90, Chi92, Jan98, Mus95, Mus96, Mus94, Eij97, EK97, KKP96, Kus00].

The type hierarchy employed has basic types for entities \((e)\), truth values \((t)\), and markers \((m)\). The states themselves can be viewed as maps from markers to suitable referents, i.e., a state has type \(m \rightarrow e\). We abbreviate this as \(s\). This can either be built into the type system from the start (see [Eij97]) or enforced by means of axioms (a kind of meaning postulates for proper state behaviour; see [Mus95, Mus94]).

Various set-ups of the encoding of DPL to type theory are possible. In the most straightforward approach, the meaning of a formula is no longer a truth value, but a state transition, i.e., the interpretations of formulas have type \(s \rightarrow s \rightarrow t\).
Following [Eij97], we will take \((u|x)\) as a primitive operation of type \(s \rightarrow s\) that resets the value of \(u\) to \(x\). Thus, \((u|x)a\) denotes the state \(a'\) that differs from state \(a\) at most in the fact that the value in \(a'\) for marker \(u\) is (the interpretation of) \(x\). Then the translation of an indefinite noun phrase \(a\ man\) becomes something like:

\[
\lambda P \lambda a \lambda a'. \exists x (\text{man } x \land Pu_i (u_i|x)a a').
\]

Here \(P\) is a variable of type \(m \rightarrow s \rightarrow s \rightarrow t\) and \(a, a'\) are variables of type \(s\), so the translation (12.3) has type \((m \rightarrow s \rightarrow s \rightarrow t) \rightarrow s \rightarrow s \rightarrow t\). Note that \(s \rightarrow s \rightarrow t\) is the type of a (characteristic function of) a binary relation on states, or, as we will call it, the type of a state transition. The \(P\) variable marks the slot for the VP interpretation. In (the present version of) dynamic semantics, VPs are interpreted as maps from markers to state transitions.

Translation (12.3) introduces an anaphoric index \(i\); as long as \(u_i\) does not get reset, any reference to \(u_i\) will pick up the link to the indefinite man that was introduced into the discourse.

### 12.3 A Toy Fragment

To give a dynamic version of the toy fragment from [Eij00c], it is useful to define some dynamic operations in typed logic. Assume \(\phi\) and \(\psi\) have the type of state transitions, i.e., type \(s \rightarrow s \rightarrow t\), and that \(a, a', a''\) have type \(s\).

\[
\begin{align*}
\exists u_i & := \lambda a a'. \exists x ((u_i|x)a = a') \\
\neg \phi & := \lambda a a'. (a = a' \land \neg \exists a'' \phi a a'') \\
\phi ; \psi & := \lambda a a'. \exists a'' (\phi a a'' \land \psi a'' a') \\
\phi \Rightarrow \psi & := \neg (\phi ; \neg \psi)
\end{align*}
\]

These operations encode the DPL semantics for dynamic quantification, dynamic negation, dynamic conjunction and dynamic implication in typed logic.

It is also useful to define an operation \(! : (s \rightarrow t) \rightarrow t\) to indicate that the state set \(s \rightarrow t\) is not empty. Thus, \(!\) serves as an indication of success. Assume \(p\) to be an expression of type \(s \rightarrow t\), the definition of \(!\) is:

\[
!p := \exists a. (pa).
\]

Note that \(\neg\) can now be defined in terms of \(!\), as

\[
\neg \phi := \lambda a a'. (a = a' \land \neg ! \phi a)
\]

We have to assume that the lexical meanings of CNs, VPs are given as one-placed predicates (type \(e \rightarrow t\)) and those of TVs as two-placed predicates (type \(e \rightarrow e \rightarrow t\)). It makes sense to define blow-up operations for lifting one-placed and two-placed predicates to the dynamic level. Assume \(A\) to be an expression of type \(e \rightarrow t\), and \(B\) an expression of type \(e \rightarrow e \rightarrow t\); we use
$r, r'$ as variables of type $m$, $a, a'$ as variables of type $s = m \to e$, and we employ postfix notation for the lifting operations:

\[
A^o := \lambda r \lambda a \lambda a' (a = a' \land A(ar)) \\
B^o := \lambda r \lambda r' \lambda a \lambda a' (a = a' \land B(ar)(ar'))
\]

The encodings of the DPL operations in typed logic and the blow-up operations for one- and two-placed predicates are employed in the semantic specification of the fragment. The semantic specifications employ variables $P, Q$ of type $m \to s \to s \to t$, variables $u, u'$ of type $m$, and variables $a, a'$ of type $s$.

\[
S ::= \text{NP VP} \\
S ::= \text{if } S S \\
NP ::= \text{Mary}^n \\
NP ::= \text{Bill}^n \\
NP ::= \text{PRO}^n \\
NP ::= \text{DET CN} \\
NP ::= \text{DET RCN} \\
DET ::= \text{every}^n \\
DET ::= \text{some}^n \\
DET ::= \text{no}^n \\
DET ::= \text{the}^n \\
CN ::= \text{man} \\
CN ::= \text{woman} \\
CN ::= \text{boy} \\
RCN ::= \text{CN that VP} \\
RCN ::= \text{CN that NP TV} \\
VP ::= \text{laughed} \\
VP ::= \text{smiled} \\
VP ::= \text{TV NP} \\
TV ::= \text{loved} \\
TV ::= \text{respected}
\]

As a quick comparison with the fragment in [Eij00c] should make clear, the types of the dynamic meanings are systematically related to the types of the earlier static meanings by a replacement of truth values (type $t$) by transitions (type $s \to s \to t$), and of entities (type $e$) by markers (type $m$).

The translation of the proper names assumes that every name is linked to an anchored marker (a marker that is never updated).
12.4 Implementation – Basic Types

module DS where
import Domain
import Model

Apart from Booleans, we need basic types for entities and (reference) markers. Entities are
defined in the module Domain. Reference markers are the dynamic variables that carry discourse
information. For convenience, we also declare a type for indices, and a map from indices to
markers.

data Marker = U0 | U1 | U2 | U3 | U4 | U5 | U6 | U7 | U8 | U9
    deriving (Eq,Bounded,Enum,Show)

type Idx = Int

i2m :: Idx -> Marker
i2m i | i < 0 || i > fromEnum (maxBound::Marker) = error "idx out of range"
    | otherwise = toEnum i

12.5 States, Propositions, Transitions

States are functions from markers to entities, so their type is Marker -> Entity. For purposes
of implementation, we will represent maps from markers to entities as lists of marker/entity
pairs.

type State = [(Marker,Entity)]

Propositions are collections of states, i.e., functions from states to truth values, i.e., they have
type State -> Bool, i.e., (Marker -> Entity) -> Bool. Again, for purposes of implementa-
tion, we will represent propositions as lists of states

type Prop = [State]
Transitions are mappings from states to propositions, i.e., their type is \texttt{State -> Prop}.

\begin{verbatim}
  type Trans = State -> Prop
\end{verbatim}

Next, we need a function for application of a state to a marker. An error message is generated if the marker is not in the domain.

\begin{verbatim}
apply :: State -> Marker -> Entity
apply [] m = error (show m ++ " not in domain")
apply ((m,e):xs) m' | m == m' = e
                   | otherwise = apply xs m'
\end{verbatim}

### 12.6 Updates

Updating a reference marker by mapping it to a new entity in a state (the implementation of the operation \(u|x\)):

\begin{verbatim}
update :: Marker -> Entity -> State -> State
update m e s = replace m e s where
    replace _ _ [] = error "state undefined for marker"
    replace m e ((m',e'):xs) | m == m' = (m,e):xs
                              | otherwise = (m',e'):replace m e xs
\end{verbatim}

### 12.7 Dynamic Negation, Conjunction, Quantification

Dynamic negation is a test on a state \(s\) that succeeds if the negated formula fails in that state, and succeeds otherwise.

\begin{verbatim}
neg :: Trans -> Trans
neg = \ phi s -> if (phi s == []) then [s] else []
\end{verbatim}

Dynamic conjunction applies the first conjunct to the initial state, next applies the second conjuncts to all intermediate results, and finally collects all end results.
12.8 Anchors for Proper Names

To get a reasonable treatment of proper names, we assume that some of the discourse markers are anchored to entities:

\[
\text{anchored :: Marker \to Entity \to Bool}
\]

anchored U6 A = True
anchored U7 M = True
anchored U8 B = True
anchored U9 J = True
anchored _ _ = False
12.9 Syntax of the Fragment

The datatype declarations for syntax are almost as in the implementation of the fragment of [Eij00c]. The main difference is that all noun phrases now carry index information. The index on proper names and pronouns is directly attached to the name or pronoun; the index information on a complex NP is attached to the determiner.

```haskell
data Sent = Sent NP VP | If Sent Sent
  deriving (Eq,Show)

data NP = Ann Idx | Mary Idx | Bill Idx | Johnny Idx | PRO Idx
  | NP1 DET CN | NP2 DET RCN
  deriving (Eq,Show)

data DET = Every Idx | Some Idx | No Idx | The Idx
  deriving (Eq,Show)

data CN = Man | Woman | Boy | Person | Thing
  deriving (Eq,Show)

data RCN = CN1 CN VP | CN2 CN NP TV
  deriving (Eq,Show)

data VP = Laughed | Smiled | VP TV NP
  deriving (Eq,Show)

data TV = Loved | Respected
  deriving (Eq,Show)
```

12.10 Lexical Meaning

For a transparent treatment of the semantics of VPs, we assume that lexical VPs have a lexical meaning of type Entity -> Bool, which is subsequently blown up to a suitable discourse type.

```haskell
lexVP :: VP -> Entity -> Bool
lexVP Laughed = laugh
lexVP Smiled = smile
```
Lexical meanings of transitive verbs are two-placed predicates.

\[
\text{lexTV} :: \text{TV} \to (\text{Entity,Entity}) \to \text{Bool}
\]
\[
\text{lexTV Loved} = \text{love}
\]
\[
\text{lexTV Respected} = \text{respect}
\]

Lexical meanings of CNs are one-placed predicates.

\[
\text{lexCN Man} = \text{man}
\]
\[
\text{lexCN Woman} = \text{woman}
\]
\[
\text{lexCN Boy} = \text{boy}
\]
\[
\text{lexCN Person} = \text{person}
\]
\[
\text{lexCN Thing} = \text{thing}
\]

These lexical meanings are blown up to the appropriate discourse types. Mapping one-place predicates to functions from markers to transitions (or: discourse predicates) is done by:

\[
\text{blowupPred} :: (\text{Entity} \to \text{Bool}) \to \text{Marker} \to \text{Trans}
\]
\[
\text{blowupPred pred m s} | \ pred \ (\text{apply s m}) = [s] \\
\ | \ otherwise = []
\]

Discourse blow-up for two-place predicates.

\[
\text{blowupPred2} :: ((\text{Entity,Entity}) \to \text{Bool}) \to \text{Marker} \to \text{Marker} \to \text{Trans}
\]
\[
\text{blowupPred2 pred m1 m2 s} | \ pred \ ((\text{apply s m1}), (\text{apply s m2})) = [s] \\
\ | \ otherwise = []
\]

### 12.11 Dynamic Interpretation

The interpretation of sentences now produces transitions (type \text{Trans}) rather than booleans:
In fact, we can get at the types of all the translation instructions by systematically replacing \texttt{Bool} by \texttt{Trans}, and \texttt{Entity} by \texttt{Marker}.

Here is the function for the interpretation of noun phrases. The code checks whether the proper names employ a suitably anchored marker, and generates an error message in case they are not.

\begin{verbatim}
intNP :: NP -> (Marker -> Trans) -> Trans
intNP (Ann i) p | anchored m A = p m
  | otherwise = error "wrong anchor"
  where m = i2m i
intNP (Mary i) p | anchored m M = p m
  | otherwise = error "wrong anchor"
  where m = i2m i
intNP (Bill i) p | anchored m B = p m
  | otherwise = error "wrong anchor"
  where m = i2m i
intNP (Johnny i) p | anchored m J = p m
  | otherwise = error "wrong anchor"
  where m = i2m i
intNP (PRO i) p = p m where m = i2m i
intNP (NP1 det cn) p = (intDET det) (intCN cn) p
intNP (NP2 det rcn) p = (intDET det) (intRCN rcn) p
\end{verbatim}

Interpretation of lexical VPs uses the dynamic blow-up from the lexical meanings. Interpretation of \((\text{VP TV NP})\) is a straightforward generalization of the treatment in [Eij00c].

\begin{verbatim}
intVP :: VP -> Marker -> Trans
intVP Laughed subject = blowupPred (lexVP Laughed) subject
intVP Smiled subject = blowupPred (lexVP Smiled) subject
intVP (VP tv np) subj = intNP np phi where phi obj = intTV tv obj subj
\end{verbatim}

Interpretation of TVs uses discourse blow-up of two-place predicates.
12.11. DYNAMIC INTERPRETATION

\[
\text{intTV} :: \text{TV} \to \text{Marker} \to \text{Marker} \to \text{Trans} \\
\text{intTV} tv = \text{blowupPred2} (\text{lexTV} tv)
\]

Interpretation of CNs uses discourse blow-up of one-place predicates.

\[
\text{intCN} :: \text{CN} \to \text{Marker} \to \text{Trans} \\
\text{intCN Man} = \text{blowupPred} (\text{lexCN Man}) \\
\text{intCN Boy} = \text{blowupPred} (\text{lexCN Boy}) \\
\text{intCN Woman} = \text{blowupPred} (\text{lexCN Woman})
\]

Code for checking that a discourse predicate is unique.

\[
\text{unique} :: \text{Marker} \to \text{Trans} \to \text{Trans} \\
\text{unique} m \phi s \mid \text{singleton} \, \text{xs} = [s] \\
\quad \mid \text{otherwise} \quad = [] \\
\text{where} \, \text{singleton} \, \, [x] = \text{True} \\
\quad \text{singleton} \, \_ = \text{False} \\
\quad \text{xs} = [x \mid x <- [\text{minBound}..\text{maxBound}], \text{success} \, (\text{update} \, m \times x \, s) \, \phi] \\
\quad \text{success} \, s \, \psi = \phi \, s \neq []
\]

Discourse type of determiners: combine two discourse predicates into a transition.

\[
\text{intDET} :: \text{DET} \to (\text{Marker} \to \text{Trans}) \to (\text{Marker} \to \text{Trans}) \to \text{Trans}
\]

Interpretation of determiners in terms of the dynamic operators defined above.

\[
\text{intDET} \, (\text{Some} \, i) \, \phi \, \psi = \exists m \, \text{‘conj’} \, (\phi \, m) \, \text{‘conj’} \, (\psi \, m) \\
\quad \text{where} \, m = \text{i2m} \, i \\
\text{intDET} \, (\text{Every} \, i) \, \phi \, \psi = (\exists m \, \text{‘conj’} \, (\phi \, m)) \, \text{‘impl’} \, (\psi \, m) \\
\quad \text{where} \, m = \text{i2m} \, i \\
\text{intDET} \, (\text{No} \, i) \, \phi \, \psi = \neg \exists m \, \text{‘conj’} \, (\phi \, m) \, \text{‘conj’} \, (\psi \, m) \\
\quad \text{where} \, m = \text{i2m} \, i \\
\text{intDET} \, (\text{The} \, i) \, \phi \, \psi = (\exists m \, (\phi \, m)) \, \text{‘conj’} \\
\quad \, (\exists m) \, \text{‘conj’} \, (\phi \, m) \, \text{‘conj’} \, (\psi \, m) \\
\quad \text{where} \, m = \text{i2m} \, i
\]
The interpretation of relativised common nouns is a straightforward generalisation of the treatment in [Eij00c].

```haskell
intRCN :: RCN -> Marker -> Trans
intRCN (CN1 cn vp) m = conj (intCN cn m) (intVP vp m)
intRCN (CN2 cn np tv) m = conj (intCN cn m) (intNP np (intTV tv m))
```

### 12.12 Testing It Out

We need a suitable start state. Note that this start states respects the anchoring information for the proper names.

```haskell
startstate :: State
startstate = [(U0,A),(U1,B),(U2,C),(U3,D),(U4,E),(U5,F),(U6,A),(U7,M),(U8,B),(U9,J)]
```

Evaluation starts out from the initial state and produces a list of states. An empty output state set indicates that the sentence is false, a non-empty output state set indicates that the sentence is true. The output states encode the anaphoric discourse information that is available for subsequent discourse.

```haskell
eval :: Sent -> Prop
eval s = intSent s startstate
```

Here are some example sentences to test.
12.13 Chief Weakness

The weaknesses of DPL carry over to its extensions in typed logic. The chief weakness of DPL-based NL semantics is the need to supply indices for all noun phrases in advance. Moreover, re-use of an index destroys access to the previous value of the corresponding marker. This is a reflection of the fact that DPL has destructive assignment: quantification over $u$ replaces the previous contents of register $u$ by something new, and the old value gets lost forever.

12.14 Further Reading

When dynamic semantics for NL first was proposed in [Kam81] and [Hei82], the approach invoked strong opposition from the followers of Montague [Mon73]. Rational reconstructions to restore compositionality were announced in [GS91] and carried out in [GS90, Chi92, Jan98, Mus95, Mus96, Mus94, Eij97, EK97, KKP96, Kus00]. All of these reconstructions are based in some way or other on DPL [GS91], and they all inherit the main flaw of this approach: the destructive assignment problem. Interestingly, DRT itself did not suffer from this problem: the discourse representation construction algorithms of [Kam81] and [KR93] are stated in terms of functions with finite domains, and carefully talk about ‘taking a fresh discourse referent’ to extend the domain of a verifying function, for each new NP to be processed.
Chapter 13

Context Semantics for NL

Summary

This chapter sketches a dynamic incremental semantics for NL in polymorphic type theory, with a couple of fragments implemented in Haskell. We start out with a barebones framework, and next extend this to a set-up that can handle salience and pronoun resolution in context. The barebones framework can be viewed as the first reconstruction of Discourse Representation Theory in type theory that does justice to the incrementality and the finite domain semantics of the original.

13.1 Point of Departure: Incremental Dynamics

Destructive assignment is the main weakness of Dynamic Predicate Logic (DPL, [GS91], but see also [Bar87]) as a basis for a compositional semantics of natural language: in DPL, the semantic effect of a quantifier action $\exists x$ is that the previous value of $x$ gets lost forever. Below, we replace DPL by an incremental logic for NL semantics — call it ID for Incremental Dynamics [Eij01] — and build a type theoretic version of a compositional incremental semantics for NL that is without the destructive assignment flaw. ID can be viewed as the one-variable version of sequence semantics for dynamic predicate logic proposed in [Ver93].

Assume a first order model $M = (D, I)$. We will use contexts $c \in D^*$, and replace variables by indices into contexts. The set of terms of the language is $\mathbb{N}$. We use $|c|$ for the length of context $c$.

Given a model $M = (D, I)$ and a context $c = c[0] \cdots c[n - 1]$, where $n = |c|$ (the length of the context), we interpret terms of the language by means of $[i]_c = c[i]$. A snag is that $[i]_c$ is undefined for $i \geq |c|$; we will therefore have to ensure that indices are only evaluated in appropriate contexts. $\uparrow$ will be used for ‘undefined’. This allows us to define the relations

$$M \models_c P_{i_1} \cdots i_n, \quad M \models_i P_{i_1} \cdots i_n$$
by means of:

\[ M \models_c P_{i_1} \cdots i_n : \iff \forall j (1 \leq j \leq n \Rightarrow [i_j]_c \neq \uparrow) \quad \text{and} \quad ([i_1]_c, \ldots, [i_n]_c) \in I(P), \]

\[ M \models_c P_{i_1} \cdots i_n : \iff \forall j (1 \leq j \leq n \Rightarrow [i_j]_c \neq \uparrow) \quad \text{and} \quad ([i_1]_c, \ldots, [i_n]_c) \notin I(P), \]

and the relations

\[ M \models_c i_1 \equiv i_2, \quad M \models_c i_1 \not\equiv i_2 \]

by means of:

\[ M \models_c i_1 \equiv i_2 : \iff [i_1]_c \neq \uparrow \quad \text{and} \quad [i_2]_c \neq \uparrow \quad \text{and} \quad [i_1]_c = [i_2]_c. \]

\[ M \models_c i_1 \equiv i_2 : \iff [i_1]_c \neq \uparrow \quad \text{and} \quad [i_2]_c \neq \uparrow \quad \text{and} \quad [i_1]_c \neq [i_2]_c. \]

If \( c \in D^n \) and \( d \in D \) we use \( c \cdot d \) for the context \( c' \in D^{n+1} \) that is the result of appending \( d \) at the end of \( c \).

The ID interpretation of formulas can now be given as a map in \( D^* \hookrightarrow \mathcal{P}(D^*) \) (a partial function, because of the possibility of undefinedness):

\[
[\exists]_c(c) := \{c \cdot d \mid d \in D\}
\]

\[
[P_{i_1} \cdots i_n]_c(c) := \begin{cases} \uparrow & \text{if } \exists j (1 \leq j \leq n \text{ and } [i_j]_c = \uparrow) \\ \{c\} & \text{if } M \models_c P_{i_1} \cdots i_n \\ \emptyset & \text{if } M \models_c P_{i_1} \cdots i_n \end{cases}
\]

\[
[i_1 \equiv i_2]_c(c) := \begin{cases} \uparrow & \text{if } [i_1]_c = \uparrow \text{ or } [i_1]_c = \uparrow \\ \{c\} & \text{if } M \models_c i_1 \equiv i_2 \\ \emptyset & \text{if } M \models_c i_1 \equiv i_2 \end{cases}
\]

\[
[\neg \phi]_c(c) := \begin{cases} \uparrow & \text{if } [\phi]_c(c) = \uparrow \\ \{c\} & \text{if } [\phi]_c(c) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}
\]

\[
[\phi; \psi]_c(c) := \begin{cases} \uparrow & \text{if } [\phi]_c(c) = \uparrow \\ \bigcup \{[\psi](c') \mid c' \in [\phi](c) \} & \text{if } \exists c' \in [\phi](c) \text{ with } [\psi](c') = \uparrow \\ \emptyset & \text{otherwise} \end{cases}
\]

The definition of the semantic clause for \( \phi; \psi \) employs the fact that all contexts in \([\phi](c)\) have the same length. This property follows by an easy induction on formula structure from the definition of the relational semantics. Thus, if one element \( c' \in [\phi](c) \) is such that \([\psi](c') = \uparrow\), then all \( c' \in [\phi](c) \) have this property.

Dynamic implication \( \phi \Rightarrow \psi \) is defined in terms of \( \neg \) and \( \land \) by means of \( \neg(\phi; \neg \psi) \). Universal quantification \( \forall \phi \) is defined in terms of \( \exists, \neg \) and \( \land \) as \( \neg(\exists; \neg \phi) \), or alternatively as \( \exists \Rightarrow \phi \).

One advantage of the use of contexts is that indefinite NPs do not have to carry index information anymore.

13.1 Some man loved some woman.
The ID rendering of (13.1) is $\exists; Mi; \exists; Wi + 1; Li(i + 1)$, where \( i \) denotes the length of the input context. On the empty input context, this gets interpreted as the set of all contexts $[e_0, e_1]$ that satisfy the relation ‘love’ in the model under consideration. The result of this is that the subsequent sentence (13.2) can now use this contextual discourse information to pick up the references:

13.2 \( \text{He}_0 \) kissed her\(_1\).

Further on, we will specify a procedure for reference resolution of pronouns in a given context, but for now we will assume that pronouns carry index information.

### 13.2 Extension to Typed Logic

The Proper Treatment of Context (PTC) for NL developed in [Eij00a] in terms of polymorphic type theory (see, e.g., [Hin97, Mil78]) uses type specifications of contexts that carry information about the length of the context. E.g., the type of a context is given as $[e]_i$, where $i$ is a type variable. Here, we will cavalierly use $[e]$ for the type of any context, and $i$ for the type of any index, thus relying on meta-context to make clear what the current constraints on context and indexing into context are. In types such as $i \rightarrow [e]$, we will tacitly assume that the index fits the size of the context. Thus, $i \rightarrow [e]$ is really a type scheme rather than a type, although the type polymorphism remains hidden from view. Since $i \rightarrow [e]$ generalizes over the size of the context, it is shorthand for the types $0 \rightarrow [e]_0$, $1 \rightarrow [e]_1$, $2 \rightarrow [e]_2$, and so on.

The translation of an indefinite noun phrase a man becomes something like:

13.3 $\lambda P \lambda e. \lambda c'. \exists x (\text{man } x \land P[e](c'x)c')$.

Here $P$ is a variable of type $i \rightarrow [e] \rightarrow [e] \rightarrow t$, while $c, c'$ are variables of type $[e]$ (variables ranging over contexts). The translation (13.3) has type $(i \rightarrow [e] \rightarrow [e]) \rightarrow [e] \rightarrow [e] \rightarrow t$. The $P$ variable marks the slot for the VP interpretation. $|c|$ gives the length of the input context, i.e., the position of the next available slot. Note that $c'x|c| = x$.

Translation (13.3) does not introduce an anaphoric index, as in DPL based dynamic semantics for NL (see [Eij00b]). Instead, an anaphoric index $i$ is picked up from the input context. Also, the context is not reset but incremented: context update is not destructive like in DPL.

### 13.3 A First Toy Fragment

First we declare a module, and import the standard List module.
module CS where

import List

To give an incremental version of the toy fragment from [Eij00c] and [Eij00b], we define the appropriate dynamic operations in typed logic. Assume $\phi$ and $\psi$ have the type of context transitions, i.e., type $[e] \rightarrow [e] \rightarrow t$, and that $c, c', c''$ have type $[e]$. Note that ` is an operation of type $[e] \rightarrow e \rightarrow [e]$.

\begin{align*}
\exists & \quad \lambda cc'. \exists x (c' x = c') \\
\neg \phi & \quad \lambda cc'. (c = c' \land \neg \exists c'' \phi cc'') \\
\phi ; \psi & \quad \lambda cc'. \exists c'' (\phi cc'' \land \psi cc'' c')
\end{align*}

These operations encode the semantics for incremental quantification, dynamic incremental negation and dynamic incremental conjunction in typed logic.

Again, we also define an operation $\downarrow$: $([e] \rightarrow t) \rightarrow t$ to indicate that the context set $[e] \rightarrow t$ is not empty. Thus, $\downarrow$ serves as an indication of success. Assume $p$ to be an expression of type $[e] \rightarrow t$, the definition of $\downarrow$ is:

\[ \downarrow p \quad := \quad \exists c. (pc). \]

We have to assume that the lexical meanings of CNs, VPs are given as one place predicates (type $e \rightarrow t$) and those of TVs as two place predicates (type $e \rightarrow e \rightarrow t$). We therefore define blow-up operations for lifting one-placed and two-placed predicates to the dynamic level. Assume $A$ to be an expression of type $e \rightarrow t$, and $B$ an expression of type $e \rightarrow e \rightarrow t$; we use $c, c'$ as variables of type $[e]$, and $j, j'$ as variables of type $t$, and we employ postfix notation for the lifting operations:

\begin{align*}
A^o & \quad := \quad \lambda jcc'. (c = c' \land Ac[j]) \\
B^* & \quad := \quad \lambda j j' cc'. (c = c' \land Bc[j][c[j']])
\end{align*}

The encodings of the ID operations in typed logic and the blow-up operations for one- and two-placed predicates are employed in the semantic specification of the fragment. The semantic specifications employ variables $P, Q$ of type $\iota \rightarrow [e] \rightarrow [e] \rightarrow t$, variables $j, j'$ of type $\iota$, and variables $c, c'$ of type $[e]$.

We will assume that pronouns are the only NPs that carry indices. Appropriate indices for proper names are now extracted from the current context. The topic of pronoun reference resolution in context will be taken up below, in our second fragment.
The main difference with the fragments in [Eij00c] and [Eij00b] is that determiners do not carry indices anymore. The appropriate index is provided by the length of the input context. As in [Eij00b], it is assumed that all proper names are linked to anchored elements in context. In fact, the anchoring mechanism has been greatly improved by the switch from DPL-style to ID-style dynamics, for the incrementality of the context update mechanism ensures that no anchored elements can ever be overwritten.

In the implementation below we also throw in reflexive pronouns; the formulation of the abstract interpretation rule for those is left as an exercise for the reader.

## 13.4 Implementation — Basic Types

For our Haskell [JH+99] implementation, we start out from basic types for booleans and entities. Contexts get represented as lists of entities. Propositions are lists of contexts. Transitions are maps from contexts to propositions. Indices are integers:
data Entity = A | B | C | D | E | F | G | H | I | J | K | L | M
    deriving (Eq, Bounded, Enum, Show)

type Context = [Entity]
type Prop = [Context]
type Trans = Context -> Prop
type Idx = Int

13.5 Index Lookup and Context Extension

lookupIdx is the implementation of $c[i]$. extend is the implementation of $c \cdot x$. extend replaces the destructive update from [Eij00b].

lookupIdx :: Context -> Idx -> Entity
lookupIdx [] i = error "undefined context element"
lookupIdx (x:xs) 0 = x
lookupIdx (x:xs) i = lookupIdx xs (i-1)

extend :: Context -> Entity -> Context
extend = \ c e -> c ++ [e]
13.6 **Dynamic Negation, Conjunction, Implication, Quantification**

\[
\text{neg} :: \text{Trans} \rightarrow \text{Trans} \\
\text{neg} = \lambda \phi \; c \rightarrow \text{if } \phi \; c = [] \text{ then } [c] \text{ else } [] \\
\text{conj} :: \text{Trans} \rightarrow \text{Trans} \rightarrow \text{Trans} \\
\text{conj} = \lambda \phi \; \psi \; c \rightarrow \text{concat } [\psi \; c' | c' \leftarrow (\phi \; c)] \\
\text{impl} :: \text{Trans} \rightarrow \text{Trans} \rightarrow \text{Trans} \\
\text{impl} = \lambda \phi \; \psi \rightarrow \text{neg } (\phi \; \text{conj} \; (\text{neg } \psi)) \\
\text{exists} :: \text{Trans} \\
\text{exists} = \lambda c \rightarrow [\text{extend } c \; x | x \leftarrow \text{[minBound..maxBound]}] \\
\text{forall0} :: \text{Trans} \rightarrow \text{Trans} \\
\text{forall0} = \lambda \phi \rightarrow \text{neg } (\text{exists } \text{conj} \; (\text{neg } \phi))
\]

13.7 **Anchors for Proper Names**

The anchors for proper names are extracted from an initial context.

\[
\text{context} :: \text{Context} \\
\text{context} = [A,M,B,J] \\
\text{anchor} :: \text{Entity} \rightarrow \text{Context} \rightarrow \text{Idx} \\
\text{anchor} = \lambda e \; c \rightarrow \text{anchor}' \; e \; c \; 0 \text{ where} \\
\text{anchor}' \; e \; [] \; i = \text{error } (\text{show } e ++ " \text{ not anchored in context")} \\
\text{anchor}' \; e \; (x:xs) \; i \mid e = x = i \\
| \text{otherwise } = \text{anchor}' \; e \; xs \; (i+1)
\]

13.8 **Syntax of the First Fragment**

No index information on NPs, except for pronouns. Otherwise, virtually the same as the datatype declaration in [Eij00b]. The NPs *He*, *She*, *It* will be used in our second fragment.
data S = S NP VP | If S S | Txt S S
    deriving (Eq,Show)

data NP = Ann | Mary | Bill | Johnny |
    | PRO Idx | He | She | It |
    | NP1 DET CN | NP2 DET RCN
    deriving (Eq,Show)

data DET = Every | Some | No | The
    deriving (Eq,Show)

data CN = Man | Woman | Boy | Person | Thing | House | Cat | Mouse
    deriving (Eq,Show)

data RCN = CN1 CN VP | CN2 CN NP TV
    deriving (Eq,Show)

data VP = Laughed | Smiled | VP1 TV NP | VP2 TV REFL
    deriving (Eq,Show)

data REFL = Self deriving (Eq,Show)

data TV = Loved | Respected | Hated | Owned
    deriving (Eq,Show)

13.9 Model Information

Our model has named entities, one-placed predicates and two-placed predicates. Names:

    ann, mary, bill, johnny :: Entity
    ann = A
    mary = M
    bill = B
    johnny = J

One-placed predicates:
13.10 Lexical Meaning

The lexical meanings of VPs and CNs are one-placed predicates, those of TVs two-placed predicates. These lexical meanings are blown up to the appropriate discourse types. Mapping one-placed predicates to functions from indices to context transitions (or: context predicates) is done by:

\[
\text{blowupPred :: (Entity -> Bool) -> Idx -> Trans}
\]

\[
\text{blowupPred} = \backslash \text{pred} \ i \ c \ ->
\text{if pred (lookupIdx c i) then [c] else []}
\]

Discourse blow-up for two-placed predicates.
blowupPred2 :: (Entity -> Entity -> Bool) -> Idx -> Idx -> Trans
blowupPred2 = \ pred i1 i2 c ->
    if pred (lookupIdx c i1) (lookupIdx c i2) then [c] else []

Interpretation of VPs consisting of a TV with a reflexive pronoun uses the relation reducer self. Note the polymorphism of this definition. We will use the relation reducer on relations in type Idx -> Idx -> Trans rather than Entity -> Entity -> Bool.

self :: (a -> a -> b) -> a -> b
self = \ p x -> p x x

13.11 Dynamic Interpretation

The interpretation of sentences, in type S -> Trans:

intS :: S -> Trans
intS (S np vp) = (intNP np) (intVP vp)
intS (If s1 s2) = (intS s1) ‘impl’ (intS s2)
intS (Txt s1 s2) = (intS s1) ‘conj’ (intS s2)

Interpretations of proper names and pronouns.

intNP :: NP -> (Idx -> Trans) -> Trans
intNP Mary = \ p c -> p (anchor mary c) c
intNP Ann = \ p c -> p (anchor ann c) c
intNP Bill = \ p c -> p (anchor bill c) c
intNP Johnny = \ p c -> p (anchor johnny c) c
intNP (PRO i) = \ p -> p i

Interpretation of complex NPs as expected:

intNP (NP1 det cn) = (intDET det) (intCN cn)
intNP (NP2 det rcn) = (intDET det) (intRCN rcn)
Interpretation of \((VP_1 \ TV \ NP)\) as expected. Interpretation of \((VP_2 \ TV \ REFL)\) uses the relation reducer \texttt{self}. Interpretation of lexical VPs uses discourse blow-up from the lexical meanings.

\begin{verbatim}
intVP :: VP -> Idx -> Trans
intVP (VP1 tv np) = \ subj -> intNP np (\ obj -> intTV tv obj subj)
intVP (VP2 tv _) = self (intTV tv)
intVP Laughed = blowupPred laugh
intVP Smiled = blowupPred smile
\end{verbatim}

Interpretation of TVs uses discourse blow-up of two-placed predicates.

\begin{verbatim}
intTV :: TV -> Idx -> Idx -> Trans
intTV Loved = blowupPred2 love
intTV Respected = blowupPred2 respect
intTV Hated = blowupPred2 hate
intTV Owned = blowupPred2 own
\end{verbatim}

Interpretation of CNs uses discourse blow-up of one-placed predicates.

\begin{verbatim}
intCN :: CN -> Idx -> Trans
intCN Man = blowupPred man
intCN Boy = blowupPred boy
intCN Woman = blowupPred woman
intCN Person = blowupPred person
intCN Thing = blowupPred thing
intCN House = blowupPred house
intCN Cat = blowupPred cat
intCN Mouse = blowupPred mouse
\end{verbatim}

Code for checking that a discourse predicate is unique.
Discourse type of determiners: combine two context predicates into a transition.

\[
\begin{align*}
\text{intDET} & ::= \text{DET} \to (\text{Idx} \to \text{Trans}) \to (\text{Idx} \to \text{Trans}) \to \text{Trans} \\
\text{intDET Some} &= \lambda \phi \psi c \to \text{let } i = \text{length } c \text{ in} \\
& \quad \text{(exists ‘conj’ (phi i) ‘conj’ (psi i)) } c \\
\text{intDET Every} &= \lambda \phi \psi c \to \text{let } i = \text{length } c \text{ in} \\
& \quad \neg \text{(exists ‘conj’ (phi i) ‘conj’ (neg (psi i))) } c \\
\text{intDET No} &= \lambda \phi \psi c \to \text{let } i = \text{length } c \text{ in} \\
& \quad \neg \text{(exists ‘conj’ (phi i) ‘conj’ (psi i)) } c \\
\text{intDET The} &= \lambda \phi \psi c \to \text{let } i = \text{length } c \text{ in} \\
& \quad ((\text{unique } i \text{ (phi i)) ‘conj’} \\
& \quad \text{exists ‘conj’ (phi i) ‘conj’ (psi i)) } c
\end{align*}
\]

The interpretation of relativised common nouns is as expected:

\[
\begin{align*}
\text{intRCN} & ::= \text{RCN} \to \text{Idx} \to \text{Trans} \\
\text{intRCN} (\text{CN1 } cn \ vp) &= \lambda i \to \text{conj (intCN cn i) (intVP vp i)} \\
\text{intRCN} (\text{CN2 } cn \ np \ tv) &= \lambda i \to \text{conj (intCN cn i)} \\
& \quad \text{(intNP np (intTV tv i))}
\end{align*}
\]
13.12 Testing It Out

The initial context from which evaluation can start is given by context.

\[
\text{eval} :: S \rightarrow \text{Prop} \\
\text{eval} = \lambda s \rightarrow \text{intS} \ s \ \text{context}
\]

Here is a set of example sentences, for use as a test suite:

- ex1 = S Johnny Smiled
- ex2 = S Bill Laughed
- ex3 = If (S Bill Laughed) (S Johnny Smiled)
- ex4 = (S Bill Laughed) ‘Txt’ (S Johnny Smiled)
- ex5 = (S Bill Smiled) ‘Txt’ (S (PRO 1) (VP1 Loved (NP1 Some Woman)))
- ex6 = S (NP1 The Boy) (VP1 Loved (NP1 Some Woman))
- ex7 = S (NP1 Some Man) (VP1 Loved (NP1 Some Woman))
- ex8 = S (NP1 Some Man) (VP1 Respected (NP1 Some Woman))
- ex9 = S (NP1 The Man) (VP1 Loved (NP1 Some Woman))
- ex10 = S (NP1 Every Man) (VP1 Loved (NP1 Some Woman))
- ex11 = S (NP1 Every Man) (VP1 Loved Johnny)
- ex12 = S (NP1 Some Woman) (VP1 Loved Johnny)
- ex13 = S Johnny (VP1 Respected (NP1 Some Man (VP1 Loved Mary)))
- ex14 = S (NP1 No Woman) (VP1 Loved Bill)
- ex15 = S (NP2 No (CN1 Woman (VP1 Hated Johnny))) (VP1 Loved Bill)
- ex16 = S (NP2 Some(CN1 Woman (VP1 Respected Johnny))) (VP1 Loved Bill)
- ex17 = S (NP1 The Boy) (VP1 Loved Johnny)
- ex18 = S (PRO 2) (VP1 Loved (PRO 1))
- ex19 = S (PRO 2) (VP1 Respected (PRO 1))
- ex20 = If (S (NP1 Some Man) (VP1 Loved (NP1 Some Woman)))
- ex21 = Txt (S (NP1 Some Man) (VP1 Loved (NP1 Some Woman)))
- ex22 = S (NP1 Some Woman) (VP1 Owned (NP1 Some Thing))
- ex23 = S (NP1 Some Woman) (VP1 Owned (NP1 The House))
- ex24 = S (NP1 Some Man) (VP1 Owned (NP1 The House))
- ex25 = S (NP1 Some Woman) (VP1 Owned (NP1 The House))
- ex26 = S (NP1 Every Man) (VP1 Respected Self)
- ex27 = S (NP1 Some Man) (VP2 Respected Self)
13.13 Updating Salience Relations in a Context

Pronoun resolution should resolve pronouns to the most salient referent in context, modulo additional constraints such as gender agreement. To handle salience, we need contexts with slightly more structure, so that context elements can be permuted without danger of losing track of them. Contexts as lists of elements under a permutation are conveniently represented as lists of index/element pairs. To talk about the permutation \textbf{abcd} of the context \textbf{abcd} we represent as \([(0, a), (1, b), (2, c), (3, d)]\) and reshuffle this to \([(2, c), (1, d), (0, a), (3, d)]\). Then we can continue to use index 2 to pick up the reference to \(c\), no matter where \(c\) ends up in the reshuffled context, while the list ordering encodes salience of the items.

In an indexed context \(c\), the entity with index \(i\) is given by \(c[i]\). E.g.,

\[
[(2, c), (1, d), (0, a), (3, d)][*0] = a.
\]

Before, a test on a context left the context unaffected when it succeeded. We now have to modify this to allow for salience reshuffles. For this, we reformulate the semantics for ‘contexts under permutation’ as follows.

If \(c\) is a context under permutation, let \((i)c\) be the result of placing the item \((i, c[i])\) upfront. E.g., if \(c = [(2, c), (1, b), (0, a), (3, d)]\), then \((3)c = [(3, d), (2, c), (1, b), (0, a)], i.e., it is the result of moving \((3, d)\) to the head position in the list. It is not hard to see that successive applications of this operation can generate all permutations of a context.

If \(d \in D, d : c\) is the result of putting item \((|c|, d)\) at the head position of the context list. Thus, if \(c = [(2, c), (1, b), (0, a), (3, d)]\), \(e : c = [(4, e), (2, c), (1, b), (0, a), (3, d)]\). This operation is used for adding a new element to the context, in most salient position.

Finally, we need a way of cutting context down to size: \([\hat{i}] \ c\) specifies the result of removing all items with an index \(\geq i\) from the context \(c\). E.g.,

\[
[3] [(2, c), (1, b), (0, a), (3, d)] = [(2, c), (1, b), (0, a)].
\]

Note that \([\hat{i}] \ c\) produces a context of length \(i\). Lift this operation to sets of contexts \(C\) by means of \([\hat{i}] \ C := \{[\hat{i}] \ c \mid c \in C\}\).

Assume \(i, j\) to be indices with \(i, j < |c|\). Then \(M \models_c Pi, M \models_c Pij\) and \(M \models_c Pij\) are given by:

\[
M \models_c Pi \iff c[i] \in I(P), \
M \models_c Pij \iff c[i] \notin I(P).
\]

We specify the positive and negative relational meaning of the basic context logic (in a similar style to [Ber96]) as follows:
[3]⁺(c) := \{d : c | d ∈ D\}
[3]⁻(c) := ∅

[Pi]⁺(c) :=
\begin{cases}
\{(i)c\} & \text{if } M \models_c Pi \\
\emptyset & \text{if } M \not\models_c Pi
\end{cases}

[Pi]⁻(c) :=
\begin{cases}
\{(i)c\} & \text{if } M \models_c Pi \\
\emptyset & \text{if } M \not\models_c Pi
\end{cases}

[Pij]⁺(c) :=
\begin{cases}
\{(i)(j)c\} & \text{if } M \models_c Pij \\
\emptyset & \text{if } M \not\models_c Pij
\end{cases}

[Pij]⁻(c) :=
\begin{cases}
\{(i)(j)c\} & \text{if } M \models_c Pij \\
\emptyset & \text{if } M \not\models_c Pij
\end{cases}

[-φ]⁺(c) := [[ϕ]⁻(c)
[-φ]⁻(c) := \|c\| [[ϕ]⁺(c)

[[ϕ; ψ]⁺(c) := \bigcup\{[[ψ]⁺(c') \mid c' ∈ [[ϕ]⁺(c)\}

[[ϕ; ψ]⁻(c) :=
\begin{cases}
\emptyset & \text{if } \exists c' ∈ [[ϕ]⁺(c) \text{ with } [[ψ]⁺(c') \neq ∅} \\
[[ϕ]⁻(c) & \text{if } [[ϕ]⁺(c) = ∅} \\
\{\|c\| \bigcup \{[[ψ]⁻(c') \mid c' ∈ [[ϕ]⁺(c)\} \} & \text{otherwise}
\end{cases}

The notion of a test on context now gets refined, as follows. Let \( c \sim c' :⇔ c \text{ is a permutation of } c' \). Then a test-new-style on input context \( c \) is a formula \( φ \) with the property that for any model it holds that all \( c' ∈ [[ϕ]⁺(c) \) satisfy \( c \sim c' \), and all \( c' ∈ [[ϕ]⁻(c) \) satisfy \( c \sim c' \).

The following proposition asserts that negations and predications are tests-new-style. The proof is immediate from the definitions.

**Proposition 13.1** For all \( φ \) in the language, for all contexts \( c \):

- if \( c' ∈ [[Pi]⁺(c) \) then \( c \sim c' \), if \( c' ∈ [[Pi]⁻(c) \) then \( c \sim c' \),
- if \( c' ∈ [[Pij]⁺(c) \) then \( c \sim c' \), if \( c' ∈ [[Pij]⁻(c) \) then \( c \sim c' \),
- if \( c' ∈ [[-φ]⁺(c) \) then \( c \sim c' \), if \( c' ∈ [[-φ]⁻(c) \) then \( c \sim c' \).

Note that double negation does change the interpretation of \( φ \) into a test modulo permutation, for we have:

\([[−−φ]⁺(c) = [[−φ]⁻(c) = \|c\|[[φ]⁺(c)\].

The next proposition reassures us that the associativity property of sequential composition still holds.

**Proposition 13.2** For all \( φ, ψ, χ \) in the language, for all contexts \( c \):

\([φ; (ψ; χ)]⁺(c) = [[(φ; ψ); χ]⁺(c),

\]
and

$$[(\phi; \psi; \chi)^-(c) = [[(\phi; \psi); \chi]^-(c).$$

**Proof.** The case of $$[(\phi; \psi; \chi)^+(c) = [[(\phi; \psi); \chi]^+(c)$$ is straightforward. The reasoning for $$[(\phi; \psi; \chi)^-(c) = [[(\phi; \psi); \chi]^-(c)$$ is as follows.

First assume $$\exists c' \in [(\phi)^+(c) with $$[(\psi; \chi)^+(c') \neq \emptyset$$. This is equivalent to $$\exists c' \in [\phi; \psi]^+(c)$$ with $$[\chi]^+(c') \neq \emptyset$$. Therefore, in this case $$[(\phi; \psi; \chi)^-(c) = [(\phi; \psi); \chi]^-(c) = \emptyset$$.

Next assume $$[(\phi)^+(c) = \emptyset$$. Then by the definition of $$[\cdot]^-$ we have that $$[\phi; \psi; \chi]^-(c) = [\phi]^-(c)$$. From $$[\phi]^+(c) = \emptyset$$ we have that $$[\phi; \psi]^+(c) = \emptyset$$. Therefore $$[\psi; \chi]^-(c) = [\phi; \psi]^-(c)$$. Again by $$[\phi]^+(c) = \emptyset$$ we get that $$[\phi; \psi]^-(c) = [\phi]^-(c)$$. Therefore, in this case, $$[(\phi; \psi; \chi)^-(c) = [(\phi; \psi); \chi]^-(c) = [\phi]^-(c)$$.

Finally assume $$[(\phi)^+(c) \neq \emptyset$$, and for all $$c' \in [(\phi)^+(c)$$ it holds that $$[(\psi; \chi)^+(c') = \emptyset$$. Two cases: (i) $$[(\phi; \psi)^+(c) = \emptyset$$ or (ii) $$[(\phi; \psi)^+(c) \neq \emptyset$$.

In case (i),

$$[(\phi; \psi; \chi)^-(c) = \{c| \cup \{[(\psi; \chi)^-(c') | c' \in [(\phi)^+(c)$$

$$= \{c| \cup \{[(\psi)^-(c') | c' \in [(\phi)^+(c)$$

$$= [(\phi; \psi)^-(c)$$

$$= [(\phi; \psi); \chi)^-(c).$$

In case (ii), we get:

$$[(\phi; \psi; \chi)^-(c) = \{c| \cup \{[(\psi; \chi)^-(c') | c' \in [(\phi)^+(c)$$

$$= \{c| \cup \{[(\psi)^-(c') | c' \in [(\phi)^+(c)$$

$$= \{c| \cup \{[(\psi)^-(c') | c' \in [(\phi; \psi)^+(c)$$

$$= [(\phi; \psi); \chi)^-(c).$$

Here is an example to further illustrate the definitions of $$[\cdot]^+$$ and $$[\cdot]^-$$. Suppose we have a context with a reference to Mary, say, an item $$(3, \text{m})$$ in a context of size 5. Let us assume the context, with its salience ordering, looks like this:

$$[(4, \text{b}), (2, \text{a}), (1, \text{c}), (3, \text{m}), (0, \text{d})]$$

Then *No woman loves Mary* will have translation $$\neg(\exists; W5; L5 3)$$ in this context. Suppose we are interpreting this translation in a model where the sentence is true. Then $$\exists; W5; L5 3$$ should turn out false. For that, every update of the context with an item $$(5, w)$$, where $$w$$ is some woman in the model should render $$L5 3$$ false. The processing of this falsity check will have as a result that the context gets reshuffled to

$$[(5, w), (3, \text{m}), (4, \text{b}), (2, \text{a}), (1, \text{c}), (0, \text{d})]$$
Finally, when this is cut down to the original size 5 of the input context, we get:

\[
[5] \ [(5, w), (3, m), (4, b), (2, a), (1, c), (0, d)] = [(3, m), (4, b), (2, a), (1, c), (0, d)].
\]

The result is a salience update of the input context, with (3, m) moved to the salient position.

### 13.14 Modifications of Basic Type Declarations

We turn to the implementation. As we are going to do pronoun resolution, we need more informative contexts, for the context should reveal which pronoun got resolved to which context element. For this, we define a language of constraints, as follows:

```haskell
data Constraint = C1 VP Idx | C2 TV Idx Idx
   deriving Eq

instance Show Constraint where
   show (C1 vp i) = show vp ++ (' ':'show i)
   show (C2 tv i j) = show tv ++ (' ':'show i) ++ (' ':'show j)
```

Examples of constraints are \(C1 \text{Laughed} \ 3\) and \(C2 \text{Hated} \ 4 \ 5\). These are displayed on the screen as \(\text{Laughed} \ 3\) and \(\text{Hated} \ 4 \ 5\), respectively.

To keep track of the indices in a constraint we define a function that retrieves its largest index.

```haskell
maxIndex :: Constraint -> Idx
maxIndex (C1 vp i) = i
maxIndex (C2 tv i j) = max i j
```

To keep track of the modifications, we use \(\text{Context'}\) for the type of contexts-new-style, and similarly for \(\text{Prop'}\) and \(\text{Trans'}\). Contexts-new-style have constraint lists built into them.

```haskell
type Context' = [(Idx,Entity)],[Constraint]
type Prop' = [Context']
type Trans' = Context' -> Bool -> Prop'
```

The new datatype for transitions allows for the specification of positive and negative transitions: if \(\phi :: \text{Trans'}\) and \(c :: \text{Context'}\), then \(\phi \ c \ \text{True}\) specifies the positive transition for \(c\), and \(\phi \ c \ \text{False}\) specifies the negative transition for \(c\).
For the new kind of context we need a new function to retrieve its size:

\[
\text{size} :: \text{Context'} \rightarrow \text{Int} \\
\text{size} (c,co) = \text{length} c
\]

### 13.15 New Utility Functions

`lookupIdx'` looks up at an index to retrieve an entity; `lookupIdx' c i` is the implementation of `c[*i]`.

\[
\text{lookupIdx'} :: \text{Context'} \rightarrow \text{Idx} \rightarrow \text{Entity} \\
\text{lookupIdx'} ([],co) \ j = \text{error "undefined context element"} \\
\text{lookupIdx'} ((i,x):xs,co) \ j | i == j = x \\
| \text{otherwise} = \text{lookupIdx'} (xs,co) \ j
\]

`adjust` adjusts the context by putting a discourse item up front, thus permuting the salience ordering.

\[
\text{adjust} :: (\text{Idx},\text{Entity}) \rightarrow \text{Context'} \rightarrow \text{Context'} \\
\text{adjust} (i,x) (c,co) | \text{elem } (i,x) \ c = ((i,x):(\text{filter } (/=(i,x)) \ c)),co \\
| \text{otherwise} = \text{error "item not found in context"}
\]

New version of `extend`:

\[
\text{extend'} :: \text{Context'} \rightarrow \text{Entity} \rightarrow \text{Context'} \\
\text{extend'} = \\lambda (c,co) e \rightarrow \text{let } i = \text{length} c \text{ in } ((i,e):c),co
\]

`success` checks if transition is possible from context.

\[
\text{success} :: \text{Context'} \rightarrow \text{Trans'} \rightarrow \text{Bool} \\
\text{success} = \\lambda c \phi \rightarrow \phi c \text{ True } /= []
\]
13.16. NEW VERSIONS OF THE DYNAMIC OPERATIONS

\( \text{cutoff } cs \ i \) cuts off all context elements in the list of contexts \( cs \) with index \( \geq i \). This is the implementation of \([i] C\), the operation used to cut all contexts in \( C \) down to size \( i \). The constraints that employ indices \( \geq i \) are also removed, of course.

\[
\begin{align*}
cutoff :: [\text{Context'}] \rightarrow \text{Idx} \rightarrow [\text{Context'}] \\
cutoff \ [] \ i &= [] \\
cutoff \ ((c,co):cs) \ i &= (\text{cutoffc} \ c \ i, \text{cutoffco} \ co \ i):(\text{cutoff} \ cs \ i) \\
\text{where} \\
\text{cutoffc} \ [] \ i &= [] \\
\text{cutoffc} \ ((j,x):xs) \ i \mid j \geq i &= \text{cutoffc} \ xs \ i \\
\mid \text{otherwise} &= (j,x):(\text{cutoffc} \ xs \ i) \\
\text{cutoffco} \ [] \ i &= [] \\
\text{cutoffco} \ (co:cos) \ i \mid \text{maxIndex} \ co \geq i &= \text{cutoffco} \ cos \ i \\
\mid \text{otherwise} &= co:(\text{cutoffco} \ cos \ i)
\end{align*}
\]

The result of a cutoff may be that we end up with multiple copies of the same context. To remedy this, we need a function for removing superfluous copies: \text{nub}, from the standard \text{List} module.

13.16 New Versions of the Dynamic Operations

The \text{neg'} of a transition-new-style is got by swapping the truth value, and doing a cutoff for the negative transition relation.

\[
\begin{align*}
\text{neg'} :: \text{Trans'} \rightarrow \text{Trans'} \\
\text{neg'} &= \phi \ c \ b \rightarrow \text{if } b \ \text{then } \phi \ c \ False \\
&\quad \text{else } \text{cutoff} \ (\phi \ c \ True) \ (\text{size} \ c)
\end{align*}
\]

If \text{conj'} succeeds then the result is as before, if \text{conj'} fails the salience order of the input gets adjusted by cutting a ‘counterexample’ context back to its original size.
conj' :: Trans' -> Trans' -> Trans'
conj' = \ phi psi c b ->
  if b then concat [ psi c' True | c' <- phi c True ]
  else if any (\c' -> psi c' True /= []) (phi c True)
    then []
  else if (phi c True) == []
    then (phi c False)
  else
    nub (cutoff
     (concat [ psi c' False | c' <- phi c True ]) (size c))

impl' can now simply be defined in terms of neg' and conj'.

impl' :: Trans' -> Trans' -> Trans'
impl' = \ phi psi -> neg' (phi `conj'` (neg' psi))

exists' takes into account that the quantifier action always succeeds.

exists' :: Trans'
exists' = \ c b -> if b then [ (extend' c e) | e <- [minBound..maxBound]]
    else []

13.17 Syntax, Lexical Semantics

Nothing new, except for the fact that pronouns do not carry indices anymore.
The procedure for predicate blowup is modified, for this is the spot where salience gets reset.
Note that the salience relations of the context are adjusted even if the predicate does not hold.
blowupPred' :: (Entity -> Bool) -> Idx -> Trans'
blowupPred' = \ pred i c b ->
  let
    e = lookupIdx' c i
    c' = adjust (i,e) c
  in
    if b then if pred e then [c'] else []
    else if pred e then [] else [c']

For blowup of VPs we add an extra touch to this, by including the relevant constraint in the context if the predicate holds.

blowupVP :: VP -> (Entity -> Bool) -> Idx -> Trans'
blowupVP = \ vp pred i c b ->
  let
    e = lookupIdx' c i
    (c',cos) = adjust (i,e) c
    co = C1 vp i
  in
    if b then if pred e then [(c',co:cos)] else []
    else if pred e then [] else [(c',cos)]

The new definition of predicate blowup for two-placed predicates ensures that subject is more salient than object.

blowupPred2' :: (Entity -> Entity -> Bool) -> Idx -> Idx -> Trans'
blowupPred2' = \ pred object subject c b ->
  let
    e1 = lookupIdx' c object
    e2 = lookupIdx' c subject
    c' = adjust (subject,e2) (adjust (object,e1) c)
  in
    if b then if pred e1 e2 then [c'] else []
    else if pred e1 e2 then [] else [c']

To use this for blowup of TVs, we also need to add the relevant constraint to the context if the predicate holds.
**13.18 Reference Resolution**

The following code implements a simple but powerful reference resolution mechanism. It just picks the indices of the entities satisfying the gender constraint from the current context, in order of salience.

\begin{verbatim}
blowupTV :: TV -> (Entity -> Entity -> Bool) -> Idx -> Idx -> Trans'
blowupTV = \ tv pred object subject c b ->
  let
    e1 = lookupIdx' c object
    e2 = lookupIdx' c subject
    (c',cos) = adjust (subject,e2) (adjust (object,e1) c)
    co = C2 tv subject object
    in
    if b then if pred e1 e2 then [(c',co:cos)] else []
    else if pred e1 e2 then [] else [(c',cos)]
\end{verbatim}

\begin{verbatim}
resolveMASC :: Context' -> [Idx]
resolveMASC (c,co) = resolveMASC' c where
  resolveMASC' [] = []
  resolveMASC' ((i,x):xs) | man x = i : resolveMASC' xs
                        | otherwise = resolveMASC' xs

resolveFEM :: Context' -> [Idx]
resolveFEM (c,co) = resolveFEM' c where
  resolveFEM' [] = []
  resolveFEM' ((i,x):xs) | woman x = i : resolveFEM' xs
                        | otherwise = resolveFEM' xs

resolveNEUTR :: Context' -> [Idx]
resolveNEUTR (c,co) = resolveNEUTR' c where
  resolveNEUTR' [] = []
  resolveNEUTR' ((i,x):xs) | thing x = i : resolveNEUTR' xs
                         | otherwise = resolveNEUTR' xs
\end{verbatim}

Names are resolved by picking the most salient index to the named entity from the current context. If the name has no index in context, the context is extended with an index for the named object.
resolveNAME :: Entity -> Context' -> (Idx,Context')
resolveNAME x c | i /= -1 = (i,c)
    | otherwise = (j,extend' c x)
where i = index x c
    j = size c
index x ([],co) = -1
index x ((i,y):xs,co) | x == y = i
    | otherwise = index x (xs,co)

As matters stand now, we will still get wrong results for examples like he respected him, where
the well-known constraint should be imposed that he and him do not co-refer (see, e.g., Reinhart
[Rei83]). This constraint on coreference can be imposed with the following ‘irreferxivizer’:

nonCoref :: (Idx -> Idx -> Trans') -> Idx -> Idx -> Trans'
nonCoref = \ p i j c b -> if i /= j then (p i j c b) else []

When imposed on [VP TV NP] where NP is not a reflexive pronoun, this has the desired effect
of blocking coreference.¹

13.19 Dynamic Interpretation — New Version

Nothing really new, except for the fact that now we can do pronoun resolution in context. To
ensure that the old test suite still runs, we also provide a rule for PRO².

intS' :: S -> Trans'
intS' (S np vp) = (intNP' np) (intVP' vp)
intS' (If s1 s2) = (intS' s1) 'impl' (intS' s2)
intS' (Txt s1 s2) = (intS' s1) 'conj' (intS' s2)

¹In [Rei83], it is argued that the constraint on coreference is a pragmatic constraint, and that there are exceptions
to this rule. Our implementation takes this into account, for the non-coreference constraint is implemented
as a constraint on context, not on the underlying reality, where the relation that interprets the TV need not be
irreflexive.
\[
\begin{align*}
\text{intNP'} :: \text{NP} \to (\text{Idx} \to \text{Trans}') \to \text{Trans}' \\
\text{intNP'} \text{ Ann} = \lambda p \ c \rightarrow \text{let } (i,c') = \text{resolveNAME ann c in p i c'} \\
\text{intNP'} \text{ Mary} = \lambda p \ c \rightarrow \text{let } (i,c') = \text{resolveNAME mary c in p i c'} \\
\text{intNP'} \text{ Bill} = \lambda p \ c \rightarrow \text{let } (i,c') = \text{resolveNAME bill c in p i c'} \\
\text{intNP'} \text{ Johnny} = \lambda p \ c \rightarrow \text{let } (i,c') = \text{resolveNAME johnny c in p i c'} \\
\text{intNP'} \text{ He} = \lambda p \ c \ b \rightarrow \text{concat } \{p i c b \mid i \leftarrow \text{resolveMASC c}\} \\
\text{intNP'} \text{ She} = \lambda p \ c \ b \rightarrow \text{concat } \{p i c b \mid i \leftarrow \text{resolveFEM c}\} \\
\text{intNP'} \text{ It} = \lambda p \ c \ b \rightarrow \text{concat } \{p i c b \mid i \leftarrow \text{resolveNEUTR c}\} \\
\text{intNP'} \text{ (PRO i)} = \lambda p \ c \rightarrow p \ i \ c \\
\text{intNP'} \text{ (NP1 det cn)} = (\text{intDET'} \det) (\text{intCN'} \cn) \\
\text{intNP'} \text{ (NP2 det rcn)} = (\text{intDET'} \det) (\text{intRCN'} \rcn)
\end{align*}
\]

In the interpretation rule for \((\text{VP1 tv np})\) we now impose the non-coreference constraint. For the interpretation of \((\text{VP1 tv refl})\), we use the polymorphic \text{self}, this time as an operation of type \((\text{Idx} \to \text{Idx} \to \text{Trans}') \to \text{Idx} \to \text{Trans}'.\)

\[
\begin{align*}
\text{intVP'} :: \text{VP} \to \text{Idx} \to \text{Trans}' \\
\text{intVP'} \text{ (VP1 tv np)} = \lambda \text{subj} \rightarrow \\
\quad \text{intNP'} \text{ np } (\lambda \text{obj} \rightarrow \text{nonCoref } (\text{intTV'} \text{ tv}) \text{ obj subj}) \\
\text{intVP'} \text{ (VP2 tv refl)} = \text{self } (\text{intTV'} \text{ tv}) \\
\text{intVP'} \text{ Laughed} = \text{blowupVP Laughed laugh} \\
\text{intVP'} \text{ Smiled} = \text{blowupVP Smiled smile}
\end{align*}
\]

\[
\begin{align*}
\text{intTV'} :: \text{TV} \to \text{Idx} \to \text{Idx} \to \text{Trans}' \\
\text{intTV'} \text{ Loved} = \text{blowupTV Loved love} \\
\text{intTV'} \text{ Respected} = \text{blowupTV Respected respect} \\
\text{intTV'} \text{ Hated} = \text{blowupTV Hated hate} \\
\text{intTV'} \text{ Owned} = \text{blowupTV Owned own}
\end{align*}
\]
intCN' :: CN -> Idx -> Trans'
intCN' Man = blowupPred' man
intCN' Boy = blowupPred' boy
intCN' Woman = blowupPred' woman
intCN' Person = blowupPred' person
intCN' Thing = blowupPred' thing
intCN' House = blowupPred' house
intCN' Cat = blowupPred' cat
intCN' Mouse = blowupPred' mouse

Uniqueness check adjusted to new datastructures:

unique' :: Idx -> Trans' -> Trans'
unique' = \ i phi c b ->
  let
    xs = [ x | x <- [minBound..maxBound], success (extend' c x) phi ]
  in
    if b then if singleton xs then [c] else []
      else if singleton xs then [] else [c]

intDET' :: DET -> (Idx -> Trans') -> (Idx -> Trans') -> Trans'
intDET' Some = \ phi psi c -> let i = size c in
  (exists' 'conj' (phi i) 'conj' (psi i)) c
intDET' Every = \ phi psi c -> let i = size c in
  ((exists' 'conj' (phi i)) 'impl' (psi i)) c
intDET' No = \ phi psi c -> let i = size c in
  ((exists' 'conj' (phi i)) 'impl' (neg' (psi i))) c
intDET' The = \ phi psi c -> let i = size c in
  ((unique' i (phi i)) 'conj'
    exists' 'conj' (phi i) 'conj' (psi i)) c

intRCN' :: RCN -> Idx -> Trans'
intRCN' (CN1 cn vp) = \i -> (intCN' cn i) 'conj' (intVP' vp i)
intRCN' (CN2 cn np tv) = \i ->
  (intCN' cn i) 'conj' (intNP' np (intTV' tv i))
13.20 Initiatisation and Evaluation – New Style

Conversion from contexts-old-style to contexts-new-style.

\[
\text{convert} :: \text{Context} \rightarrow \text{Context'}
\]
\[
\text{convert} \ c = (\text{convert'} \ c \ (\text{length} \ c - 1),[]) \ where
\]
\[
\text{convert'} \ [] \ i = []
\]
\[
\text{convert'} \ (x:xs) \ i = (i,x):(\text{convert'} \ xs \ (i-1))
\]

Evaluation of sentences, for initial context, checking for truth:

\[
\text{eval'} :: S \rightarrow \text{Prop'}
\]
\[
\text{eval'} \ s = \text{intS'} \ s \ (\text{convert} \ \text{context}) \ True
\]

13.21 Examples

The position at the front of the context list is the most salient one. We give some variations on the earlier test examples that can be used to check pronoun resolution.
13.4 *He loved some woman.*

In a context where referents for the pronoun are available, *he* can be resolved to any referent that satisfies the property. And this is what we get. Let us first have a look at the initial context.

```
CS> convert context
([(3,A),(2,M),(1,B),(0,J)],[])
CS>
```

This context contains two women and two men. The referents (1,B) and (0,J) in the context are referents for men.

```
CS> eval' nex1
([(1,B),(4,A),(3,A),(2,M),(0,J)],[Loved 1 4]),
([(1,B),(4,M),(3,A),(2,M),(0,J)],[Loved 1 4])
CS>
```

As it turns out, in the model only B is a lover. Therefore *he* gets resolved to B, and the new contexts have B and B-s loved one as the two most salient items, with the subject more salient than the object.
13.5 *He hated some thing.*

Here we expect that we get all the new contexts where B or J with their objects of hatred are added, again with the subjects more salient than the objects. And this is what we get:

```
CS> eval' nex2
[[[(1,B),(4,E),(3,A),(2,M),(0,J)], [Hated 1 4]],
 [[(1,B),(4,F),(3,A),(2,M),(0,J)], [Hated 1 4]],
 [[(1,B),(4,G),(3,A),(2,M),(0,J)], [Hated 1 4]],
 [[(1,B),(4,H),(3,A),(2,M),(0,J)], [Hated 1 4]],
 [[(1,B),(4,I),(3,A),(2,M),(0,J)], [Hated 1 4]],
 [[(1,B),(4,K),(3,A),(2,M),(0,J)], [Hated 1 4]],
 [[(1,B),(4,L),(3,A),(2,M),(0,J)], [Hated 1 4]],
 [[(0,J),(4,E),(3,A),(2,M),(1,B)], [Hated 0 4]],
 [[(0,J),(4,F),(3,A),(2,M),(1,B)], [Hated 0 4]],
 [[(0,J),(4,G),(3,A),(2,M),(1,B)], [Hated 0 4]],
 [[(0,J),(4,H),(3,A),(2,M),(1,B)], [Hated 0 4]],
 [[(0,J),(4,I),(3,A),(2,M),(1,B)], [Hated 0 4]],
 [[(0,J),(4,K),(3,A),(2,M),(1,B)], [Hated 0 4]],
 [[(0,J),(4,L),(3,A),(2,M),(1,B)], [Hated 0 4]]
CS>
```

13.6 *Bill smiled. He loved some woman.*

In the same initial context as before we get for example (13.6):

```
CS> eval' nex3
[[[(1,B),(4,A),(3,A),(2,M),(0,J)], [Loved 1 4, Smiled 1]],
 [[(1,B),(4,M),(3,A),(2,M),(0,J)], [Loved 1 4, Smiled 1]]
CS>
```

*He* gets resolved to Bill, for as it happens, in the model Bill is the only man in love. Note that the example still works out with an empty initial context:

```
CS> intS' nex3 ([], []) True
[[[(0,B),(1,A)], [Loved 0 1, Smiled 0]],
 [[(0,B),(1,M)], [Loved 0 1, Smiled 0]]
CS>
```

The referent of *Bill* does not occur in the context, so it gets added.

13.7 *Bill smiled. He hated some thing.*

Evaluation in the empty context gives:
13.21. EXAMPLES

CS> intS' nex4 ([],[]) True
[((0,B),(1,E)], [Hated 0 1, Smiled 0]),
[((0,B),(1,F)], [Hated 0 1, Smiled 0]),
[((0,B),(1,G)], [Hated 0 1, Smiled 0]),
[((0,B),(1,H)], [Hated 0 1, Smiled 0]),
[((0,B),(1,I)], [Hated 0 1, Smiled 0]),
[((0,B),(1,K)], [Hated 0 1, Smiled 0]),
[((0,B),(1,L)], [Hated 0 1, Smiled 0])]
CS>

Since Bill is the only referent available for resolution of the pronoun, we get Bill with his objects
of hatred as new contexts. In a richer initial context, we may get more, of course:

CS> eval' nex4
[((1,B),(4,E),(3,A),(2,M),(0,J)], [Hated 1 4, Smiled 1]),
[((1,B),(4,F),(3,A),(2,M),(0,J)], [Hated 1 4, Smiled 1]),
[((1,B),(4,G),(3,A),(2,M),(0,J)], [Hated 1 4, Smiled 1]),
[((1,B),(4,H),(3,A),(2,M),(0,J)], [Hated 1 4, Smiled 1]),
[((1,B),(4,I),(3,A),(2,M),(0,J)], [Hated 1 4, Smiled 1]),
[((1,B),(4,K),(3,A),(2,M),(0,J)], [Hated 1 4, Smiled 1]),
[((1,B),(4,L),(3,A),(2,M),(0,J)], [Hated 1 4, Smiled 1]),
[((0,J),(4,E),(1,B),(3,A),(2,M)], [Hated 0 4, Smiled 1]),
[((0,J),(4,F),(1,B),(3,A),(2,M)], [Hated 0 4, Smiled 1]),
[((0,J),(4,G),(1,B),(3,A),(2,M)], [Hated 0 4, Smiled 1]),
[((0,J),(4,H),(1,B),(3,A),(2,M)], [Hated 0 4, Smiled 1]),
[((0,J),(4,I),(1,B),(3,A),(2,M)], [Hated 0 4, Smiled 1]),
[((0,J),(4,K),(1,B),(3,A),(2,M)], [Hated 0 4, Smiled 1]),
[((0,J),(4,L),(1,B),(3,A),(2,M)], [Hated 0 4, Smiled 1])]
CS>

13.8 The mouse hated the cat. It smiled.

We get that it can both be resolved to the mouse and to the cat, with a preference for the first
resolution, as subject position is more salient.

CS> eval' nex7
[((4,I),(5,F),(3,A),(2,M),(1,B),(0,J)], [Smiled 4, Hated 4 5]),
[((5,F),(4,I),(3,A),(2,M),(1,B),(0,J)], [Smiled 5, Hated 4 5])
CS>

For examples like (13.8), Kameyama [Kam98] has argued that the other order of resolution is
more plausible, for relying on world knowledge (cartoon knowledge?) we can tell that a cat-
hat ing mouse in sight of a cat is less likely to smile than a cat spotting a contemptuous mouse.
But note that even that mechanism is taken into account by our little reference resolution engine.
We simply take our model of the world as our yardstick for what is likely and what is not. In
the model under consideration the mouse *does* smile, and that is a piece of world knowledge that makes the reading with *it* resolved to the mouse plausible.\(^2\)

13.9 *The mouse respected the cat. It hated it.*

This time, we only get a reading where the mouse hates the cat. This is because it so happens that in the background model the cat does not hate the mouse.\(^3\)

```
CS> eval' nex8
([([(4,I),(5,K),(3,A),(2,M),(1,B),(0,J)], [Hated 4 5, Respected 4 5]])

CS>
```

13.10 *He respected him.*

Here the non-coreference constraint comes into play and forces the two pronouns to get resolved to different men in context.

```
CS> eval' nex11
([([(1,B),(0,J),(3,A),(2,M)], [Respected 1 0]),
([(0,J),(1,B),(3,A),(2,M)], [Respected 0 1])])

CS>
```

13.11 *The mouse respected itself. It hated the cat.*

We get the right outcome, with *it* resolved to the mouse, and the mouse ending up in most salient position in the output context:

```
CS> eval' nex13
([([(4,I),(5,K),(3,A),(2,M),(1,B),(0,J)], [Hated 4 5, Respected 4 4]])

CS>
```

13.12 *Some man respected himself. Some woman loved him.*

Here things go slightly wrong:

```
CS> eval' nex14
([([(5,A),(4,B),(3,A),(2,M),(1,B),(0,J)], [Loved 5 4, Respected 4 4]),
([(5,A),(1,B),(4,B),(3,A),(2,M),(0,J)], [Loved 5 1, Respected 4 4]),
([(5,A),(0,J),(4,B),(3,A),(2,M),(1,B)], [Loved 5 0, Respected 4 4]),
([(5,C),(4,B),(3,A),(2,M),(1,B),(0,J)], [Loved 5 4, Respected 4 4]),
([(5,C),(1,B),(4,B),(3,A),(2,M),(0,J)], [Loved 5 1, Respected 4 4]),
([(5,C),(0,J),(4,B),(3,A),(2,M),(1,B)], [Loved 5 0, Respected 4 4]),
([(5,M),(4,B),(3,A),(2,M),(1,B),(0,J)], [Loved 5 4, Respected 4 4])]
```

\(^2\)We may presume object *I* in the model to be Ignatz Mouse.

\(^3\)The reason for this is that *K* in the model happens to be Krazy Kat.
What we see is that for each man who respects himself we get for every woman that the pronoun resolves to its possible referents in the right order of plausibility, with the man who respects himself as the most salient referent for the pronoun. This, however, is not the right overall order of plausibility. The problem is that choice of reference for indefinites is independent of salience order.

13.13 If some man respected himself, some woman loved him.

CS> eval' nex15

[[[(3,A),(2,M),(1,B),(0,J)],[]],
[[[1,B),(3,A),(2,M),(0,J)],[]],
[[[0,J),(3,A),(2,M),(1,B)],[]]]

CS>

In this example, we cannot quite see what has happened. The reason is that the conditional acts as a test. It adds an individual to the context that gets removed again at the end of processing.


Again we get that the reference resolution is in the right order per choice of woman:

CS> eval' nex16

[[[(4,A),(0,J),(3,A),(2,M),(1,B)],[Loved 4 0]],
[[[4,A),(1,B),(0,J),(3,A),(2,M)],[Loved 4 1]],
[[[4,C),(0,J),(3,A),(2,M),(1,B)],[Loved 4 0]],
[[[4,C),(1,B),(0,J),(3,A),(2,M)],[Loved 4 1]]]
13.22 Related Work

New Rational Reconstruction of DRT  The present approach, based on ID rather than DPL, makes clear how the instruction to take fresh discourse referents when needed can be made fully precise by using the standard toolset of (polymorphic) type theory. To our knowledge this is the first reconstruction of DRT in type theory that does justice to the incrementality and the finite state semantics of the original.

Pronoun Resolution in Context Logic and in DRT  The proposal for the treatment of salience in the second fragment is an extension of the basic context logic. It should be compared with the treatment of pronoun resolution in DRT proposed in the second volume of [BB99], as well as with the earlier proposal for pronoun resolution in DRT in [WA86].

Visser-Style Contexts  Visser’s context theories [Vis97, Vis00b, Vis00a, VV96] also start out as rational reconstructions of DRT. Visser’s view of contexts is considerably more abstract than the simple-minded approach taken here.

The Centering Approach to Reference Resolution  Central claim of the centering theory of local coherence in discourse [GS86, GJW95] is that pronouns are used to signal to the hearer that the speaker continues to talk about the same thing. See [WJP98] for extensions and variations that take world knowledge into account, and [Bea00] for a reformulation in terms of optimality theory. The reference resolution mechanism proposed above is meant as a demonstration that reference resolution can be brought within the compass of dynamic semantics in a relatively straightforward way, and that very simple means are enough to implement something quite useful.

Referent Systems  Referent systems [Ver94, Ver93, Ver95] are a mechanism for indirect reference to objects, via variable names and indices (‘pegs’). Referent systems are employed in [GSV96] to reformulate the logic of DPL in an incremental fashion in order to solve certain puzzles of epistemic modality in dynamic semantics, and in [Bea99] in a sketch of a logic of anaphora resolution. We hope to have shown that reliance on context leads to a much simpler set-up of incremental dynamic logic, with context indices as ‘pegs’. In a sense, contexts or contexts under permutation are what is left of reference systems when one leaves out the variable names.
13.23 Future Work

Above, we have demonstrated pronoun reference resolution related to a fixed background model. More realistic is a set-up where the ‘compatible’ models grow incrementally with the discourse. This can be achieved by using a tableau unfolding mechanism for model generation (cf. [BvE82, Koh00, EHN00]). We hope to describe and implement tableau based model generation for context logic in a future paper.

13.24 Further Reading

Incremental dynamics is presented in [Eij01]. This framework can be viewed as the one-variable version of sequence semantics for dynamic predicate logic proposed in [Ver93]. Incremental dynamics is extended to polymorphic type theory in [Eij00a].
Appendix A

Solutions to the Exercises

Solutions to Exercises from Chapter 1

1.1 Replace the rule \texttt{attack} \rightarrow \texttt{position} by \texttt{attack} \rightarrow \texttt{character digit} and leave out the rule for \texttt{position}.

1.2

\begin{align*}
\text{character} & \rightarrow a | b | c | d | e | f | g | h \\
\text{digit} & \rightarrow 1 | 2 | 3 | 4 \\
\text{attack} & \rightarrow \texttt{character digit} \\
\text{reaction} & \rightarrow \texttt{missed} | \texttt{hit} | \texttt{sunk} \\
\text{turn} & \rightarrow \texttt{attack reaction} \\
\text{end} & \rightarrow \texttt{attack defeated} \\
\text{game} & \rightarrow \texttt{end} | \texttt{turn game}.
\end{align*}

1.3

\begin{align*}
\text{color} & \rightarrow \texttt{red} | \texttt{yellow} | \texttt{blue} | \texttt{green} \\
\text{answer} & \rightarrow \texttt{black} | \texttt{white} | \texttt{blank} \\
\text{guess} & \rightarrow \texttt{color color color color color} \\
\text{reaction} & \rightarrow \texttt{answer answer answer answer} \\
\text{turn} & \rightarrow \texttt{guess reaction} \\
\text{game} & \rightarrow \texttt{turn} | \texttt{turn turn} | \texttt{turn turn turn} | \texttt{turn turn turn turn}
\end{align*}
1.4

\[\text{color} \rightarrow \text{red} | \text{yellow} | \text{blue} | \text{green}\]
\[\text{answ} \rightarrow \text{black} | \text{white}\]
\[\text{guess} \rightarrow \text{color color color color}\]
\[\text{reaction} \rightarrow \text{answ} | \text{answ answ} | \text{answ answ answ} | \text{answ answ answ answ}\]
\[\text{turn} \rightarrow \text{guess reaction}\]
\[\text{game} \rightarrow \text{turn} | \text{turn turn} | \text{turn turn turn} | \text{turn turn turn turn}\]

1.7 Assume $\epsilon$ is the empty string. Then here is a grammar for strings that also generates the empty string:

\[\text{character} \rightarrow A | \cdots | Z | a | \cdots | z | \_ | . | \_ | ? | ! | ; | :\]
\[\text{string} \rightarrow \epsilon | \text{character string}\]

1.11 A numeral different from zero can not have leading zeros, a znumeral can.

**Solutions to Exercises from Chapter 5**

```haskell
module SolEPLIH

where

import EPLIH
import CINEMA
import DB
import List
```

5.1
Ann (hates/washes) Ann
Ann (hates/washes) Lucy
Ann (hates/washes) a man
Ann (hates/washes) every man
Lucy (hates/washes) Ann
Lucy (hates/washes) Lucy
Lucy (hates/washes) a man
Lucy (hates/washes) every man
A man (hates/washes) Ann
A man (hates/washes) Lucy
A man (hates/washes) a man
A man (hates/washes) every man
Every man (hates/washes) Ann
Every man (hates/washes) Lucy
Every man (hates/washes) a man
Every man (hates/washes) every man

5.2 Substitute \( d/y := \exists w (d \ast w = y) \) in the formula given in the book.

5.3

\[
\begin{align*}
\text{intCINEMA} & : \text{String} \to \text{[String]} \to \text{Bool} \\
\text{intCINEMA} \ "R" &= \text{convert2 actP} \\
\text{intCINEMA} \ "S" &= \text{convert2 directP} \\
\text{assCINEMA} & : \text{Var} \to \text{String} \\
\text{assCINEMA} &= \ \lambda \ x \to \ "\text{Marlon Brando}" \\
\text{solution} &= \text{eval universe intCINEMA assCINEMA form3}
\end{align*}
\]

5.4
APPENDIX A. SOLUTIONS TO THE EXERCISES

5.5

openForm :: Form -> Bool
openForm frm = freeVarsInForm frm /= []

5.7 This formula expresses that between any pair of numbers $x, y$ with $x$ smaller than $y$ it is possible to find a number $z$ that is in between. On the natural numbers, this is false, for there
is no natural number between \( n \) and \( n + 1 \). For the reals this is true, for if \( r_1 < r_2 \) then \( r_1 < \frac{r_1 + r_2}{2} < r_2 \).

**Solutions to Exercises from Chapter 6**

```haskell
module SolCM
where
import CM

6.5 The present type for GQs is \((e \rightarrow (t,t)) \rightarrow t\), the type previously given was \((e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t\). Conversion from \((e \rightarrow (t,t)) \rightarrow t\) to \((e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t\) is done by:

\[ C_1 = \lambda Q \lambda f \lambda g. Q((\lambda x.((fx),(gx)))). \]

For conversion in the other direction, use:

\[ C_2 = \lambda Q \lambda h. Q((\lambda x.\pi_1(hx))(\lambda x.\pi_2(hx))). \]

Here it is assumed that \( \pi_1 \) is the function that picks the first element from a pair, \( \pi_2 \) the function that picks the second element. The implementation is as follows:

```haskell
c1 :: ((a -> (Bool,Bool)) -> Bool)
    -> (a -> Bool) -> (a -> Bool) -> Bool
c1 q f g = q (\ x -> (f x, g x))

C2 :: ((a -> Bool) -> (a -> Bool) -> Bool)
    -> (a -> (Bool,Bool)) -> Bool
C2 q h = q (\ x -> fst (h x)) (\ x -> snd (h x))
```
APPENDIX A. SOLUTIONS TO THE EXERCISES

Solutions to Exercises from Chapter 9

module SolHRITT

where

import HRITT
import Model

9.1 Relation $R$ entails relation $S$ if every tuple that is in $R$ also is in $S$. Note that this definition only makes sense if $R$ and $S$ have the same arity. Here is the implementation:

```
entailREL :: (Enum a, Bounded a) => Rel a -> Rel a -> Bool
entailREL (R1 b) (R1 c) = not b || c
entailREL (R2 f) (R2 g) =
  all (\x -> entailREL (f x) (g x)) [minBound..maxBound]
entailREL _ _ = error "different arities"
```

This gives:

```
SolHRITT> entailREL (encode2 shave) (encode2 wash)
True
SolHRITT> entailREL (encode2 wash) (encode2 shave)
False
SolHRITT> entailREL (encode2 shave) (encode3 give)
Program error: different arities
```
Bibliography


[Vis00a] A. Visser. Freut sich die Lies — context modification in action. Lecture Notes, Department of Philosophy, Utrecht University, November 2000.


Index

[n..], 91
[n..m], 45
&&, 58
(op x), 55
(x op), 55
++, 45
/, 57
/=, 49
:, 43
:1, 39
:r, 40
<, 54
<=, 54
==, 49
>, 54
>=, 54
|, 46

all, 48
and, 48
any, 48
average, 57

Church, A., 38
constructor, 49
constructor identifier, 49
conversion, 57
count, 53
curry, 72
Curry, H.B., 38

data, 82
database query, 82
delete, 56

elem, 48
Eq, 49
equation guarding, 46
error, 57
filter, 45
flip, 53, 80
Float, 57
foldl, 51
foldr, 50
fromInt, 57
fst, 72
guard, 46
head, 44

identity function, 43
in, 40, 57
infix notation, 55
Int, 53
Integer, 53

lambda abstraction, 84
length, 57
let, 40, 57
list, 43
list comprehension, 44
load Haskell file, 39

map, 45
max, 55
method of fragments, 6
mgu, 121
min, 54
mod, 45
module, 38
most general unifier, 121

notElem, 48
nub, 58
or, 48
otherwise, 47
prefix, 58
prefix notation, 55
Prelude>, 39
recursive definition, 43
recursive descent parsing, 152
reload Haskell file, 40
rem, 42
reserved keywords, 49
rev, 51
sections, 55
skolem term, 125
snd, 72
srt2, 56
String, 43
String, 43
succ, 54
sum, 57
tail, 44
take, 150
Term, 92
toLower, 54
type, 82
type conversion, 57
type declaration, 40
uncurry, 72
unification algorithm, 121
VaDr, 85
variable identifier, 49
where, 38, 56
wild card, 49
words, 54