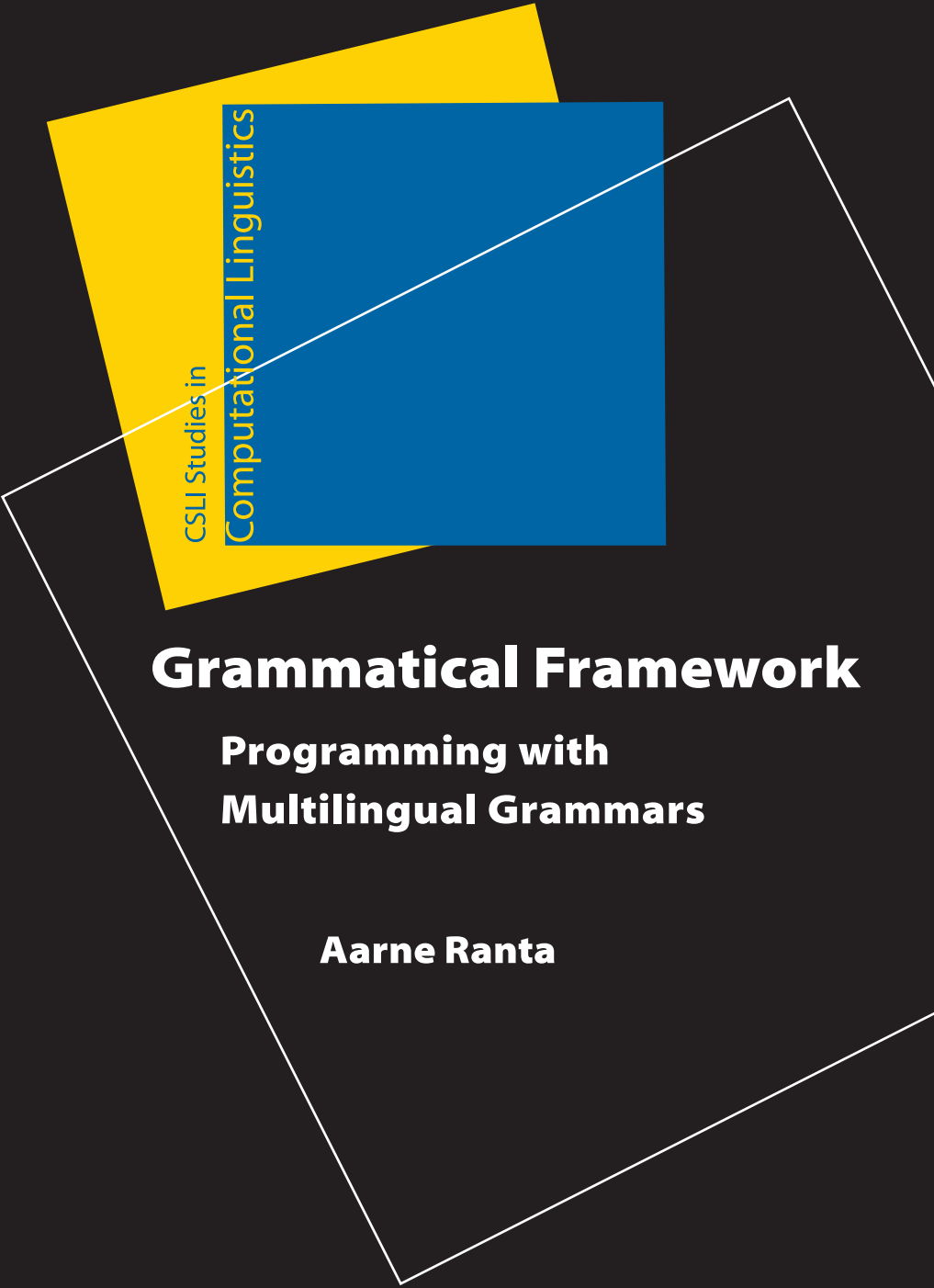




CSLI Studies in
Computational Linguistics

Grammatical Framework

**Programming with
Multilingual Grammars**

Aarne Ranta

Grammatical Framework

CSLI Studies in Computational Linguistics

This series covers all areas of computational linguistics and language technology, with a special emphasis on work which has direct relevance to practical applications, makes formal advances, and incorporates insights into natural language processing from other fields, especially linguistics and psychology. Books in this series describe groundbreaking research or provide an accessible and up-to-date overview of an area for nonspecialists. Also included are works documenting freely available resources for language processing, such as software, grammars, dictionaries, and corpora.

Series edited by Ann Copestake

CSLI Studies in
Computational Linguistics

Grammatical Framework

Programming with Multilingual Grammars

Aarne Ranta

Copyright © 2011
CSLI Publications
Center for the Study of Language and Information
Leland Stanford Junior University
Printed in the United States
15 14 13 12 11 1 2 3 4 5

Library of Congress Cataloging-in-Publication Data

Ranta, Aarne.
Grammatical Framework : programming with multilingual grammars /
Aarne Ranta.

p. cm. — (CSLI studies in computational linguistics)
Includes bibliographical references and index.

ISBN: 978-1-57586-627-7 (cloth : alk. paper)

ISBN: 978-1-57586-626-0 (pbk. : alk. paper)

1. Grammar, Comparative and general. 2. Multilingual computing.
3. Parsing (Computer grammar) 4. Computational linguistics.
- I. Title. II. Series.

P151.R623

2011

006.3'5—dc22

2011008006

CIP

eISBN: 978-1-57586-731-1 (electronic)

*CSLI Publications gratefully acknowledges a generous gift from Jill and
Donald Knuth in support of scholarly publishing that has made the
production of this book possible.*

∞ The acid-free paper used in this book meets the minimum requirements
of the American National Standard for Information Sciences—Permanence
of Paper for Printed Library Materials, ANSI Z39.48-1984.

CSLI was founded in 1983 by researchers from Stanford University, SRI
International, and Xerox PARC to further the research and development of
integrated theories of language, information, and computation. CSLI
headquarters and CSLI Publications are located on the campus of Stanford
University.

CSLI Publications reports new developments in the study of language,
information, and computation. Please visit our web site at
<http://cslipublications.stanford.edu/>
for comments on this and other titles, as well as for changes
and corrections by the author and publisher.

Contents

Preface **vii**

Acknowledgements **ix**

Web Resources and the GF System **xi**

1 Introduction **1**

I A GF Tutorial

2 Basic concepts of multilingual grammars **29**

3 Parameters, tables, and records **57**

4 Modular and scalable grammar writing **75**

5 Using the Resource Grammar Library **97**

6 Semantic actions and conditions in abstract syntax **127**

II Larger Grammars and Applications

7 Embedded grammars and code generation **155**

8 Interfacing formal and natural languages **177**

9 Getting started with resource grammar programming **199**

10	Extending the Resource Grammar Library	219

Appendix A:	A miniature resource grammar	237

Appendix B:	A glossary of linguistic terms	247

III GF Reference Manual

Appendix C:	The GF Programming Language	253

C.1	Overview of GF	253
C.2	The module system	254
C.3	Judgements	264
C.4	Types and expressions	272
C.5	Flags and pragmas	287
C.6	The grammar of GF	289

Appendix D:	The GF Resource Grammar Library	295

D.1	The category system	295
D.2	Syntax rules	298
D.3	Lexical Paradigms	303
D.4	Other library modules	304

Appendix E:	The GF Software System	307

E.1	The GF shell	307
E.2	The GF batch compiler	310

Appendix F:	Bibliography	311

F.1	Publications on GF	311
F.2	Background and related work	316

Index	323

Preface

GF, Grammatical Framework, is a special-purpose programming language for writing grammars. It supports the complexities found in different natural languages and provides engineering tools for large projects involving many programmers. It also supports abstractions and code sharing, both within single languages and across multiple languages. During its twelve years of existence, GF has found applications in multilingual translation systems, in language-based human-computer interaction, and in the creation of computational linguistic resources.

GF has a modern design based on typed functional programming languages, in particular Haskell and ML. Its ambition is to be accessible for users from different backgrounds, in particular those without a prior linguistic training. However, with its support for powerful generalizations and multilinguality, GF has also created interest among theoretically inclined linguists. Thus the users of GF include computer scientists and linguists, engineers and mathematicians, students and seniors.

This book aims to collect in one place all that one needs to know to use GF efficiently. It contains a tutorial starting from scratch and covering all features of GF, a set of chapters on practical applications and advanced grammar writing, and a reference manual giving the full picture of GF. Some of the material has previously appeared scattered in the 50-odd publications on GF and in web documents; yet it has been a serious problem for GF users that there has been no comprehensive reference in one place. Thus we hope that the book will be useful for both beginners and seasoned GF programmers.

Acknowledgements

An early version of this book was distributed to the 15 participants of a graduate course on GF arranged by GSLT (Swedish Graduate School in Language Technology) in 2007. A later version was distributed to the 30 participants of GF Summer School in 2009, who started GF projects with 16 new languages. The summer school version has also been used on courses taught by Hans Leiß at the University in Munich (LMU) and by Sebastià Xambó at Barcelona Tech (UPC). I am grateful for all feedback I've received from the teachers and participants of these events, as well as from other colleagues. In particular, I want to thank Krasimir Angelov and Björn Bringert, who have actually created parts of the material presented in the book. Håkan Burden, Olga Caprotti, Ramona Enache, Julia Hammar, Arto Mustajoki, Bengt Nordström, Jordi Saludes, and Annie Zaenen gave valuable comments on various versions of the manuscript. Special thanks go to Ann Copestake for her insightful help, both as the CSLI series editor and as a distinguished colleague in the field of grammar formalisms. Emma Pease gave the book a professional finish.

The GF software has been a collaborative open-source project, with important contributions of code from (in chronological order) Petri Mäenpää, Thomas Hallgren, Janna Khagai, Markus Forsberg, Kristofer Johansson, Peter Ljunglöf, Håkan Burden, Hans-Joachim Daniels, Björn Bringert, Krasimir Angelov, Moisés Salvador Meza Moreno, Kevin Kofler, Jordi Saludes, John Camilleri, Ramona Enache, and Grégoire Détrez.

The GF Resource Grammar Library, which is one of the main assets of GF, has received substantial contributions from Janna Khagai, Markus Forsberg, Inger Andersson, Thérèse Söderberg, Anni Laine, Jordi Saludes, Harald Hammarström, Ali El Dada, Jean-Philippe Bernardy, Andreas Priesnitz, Krasimir Angelov, Ilona Nowak, Adam

Slaski, Ramona Enache, Muhammad Humayoun, Shafqat Virk, Server Çimen, and Markos Kassa Gobena.

In addition to the direct contributions, the GF language and the theory around it have profited from several known ideas from type theory, functional programming, and linguistics. But they have also been inspired by colleagues who have contributed crucial, identifiable design ideas in personal communication: (still in chronological order) Per Martin-Löf, Thierry Coquand, Bengt Nordström, Lena Magnusson (later Pareto), Gilles Kahn (†2006), Rod Burstall, Marc Dymetman, Sylvain Pogodalla, Veronika Lux, Lauri Karttunen, Patrik Jansson, Koen Claessen, Robin Cooper, Gérard Huet, Philippe de Groote, Sibylle Schupp, Lauri Carlson, Hans Leiß, and Elisabet Engdahl.

Finally, I want to thank my dearest ones—Pihla, Eemu, Uula, Aamos, and Luukas—for their wonderful companionship, which constantly makes me happier and wiser.

Gothenburg, February 2011

Aarne Ranta
`aarne@chalmers.se`

Web Resources and the GF System

Supporting material can be found on the book's website,

<http://www.grammaticalframework.org/gf-book>

This material includes

- most of the code examples contained in the book
- clickable links to the references given in the book
- slides for class-room use
- solutions to some of the exercises

The book website is hosted under the main GF website,

<http://www.grammaticalframework.org>

which includes, among other things

- an updated reference manual with hyperlinks
- the full Resource Grammar Library API
- the source code for the GF system and the Resource Grammar Library
- executable binaries for the GF system

Anyone who reads this book is encouraged to download and install the GF system and try out the code examples and exercises. The system is available for all major computer platforms, including Linux, Mac OS, and Windows. The easiest way to obtain it is via

<http://www.grammaticalframework.org/download>

From there, the latest versions of the GF program can be obtained as ready-made executables, which only need to be downloaded and decompressed. GF Version 3.2 is tested to be in perfect match with this book, but later versions are aimed to be backward compatible, just adding new functionalities without removing old ones.

Version 3.2 will always be available via the book's website, including the sources, on-line documentation, and binaries for Linux, Mac OS, and Windows. To use one of the binaries, all you need is to download it, uncompress it, and move it to a place that is on your path of executable programs.

Introduction

1.1 What this book is about

This book is meant for everyone who wants to write computer programs that process natural language. Its main focus is on multilingual systems: systems that can deal with several languages. Translation systems are obvious examples of this. But we will cover other kinds of applications as well: natural language interfaces, spoken dialogue systems, language learning aids, and software localization. Even though such systems only use one language at a time, they must often be made available in many languages that the users can choose between. This is a growing need in the globalizing world.

We have tried to make this book accessible for readers from many backgrounds. All we assume is interest for languages and some experience from computers. Technically, this book belongs to two fields: **computational linguistics** and **functional programming**. However, prior knowledge from both of these fields in combination is very uncommon. Hence we cannot presuppose any of them. This implies a risk of writing things that are elementary for readers who already know one of the fields. However, we do believe that the combination of ideas we present is unique enough to keep most readers awake.

The main subject of the book is a programming language called GF, Grammatical Framework. Our purpose is to cover all of GF, starting from the elements and proceeding to state-of-the-art applications. The book is organized as a programming language text rather than a linguistics text. The first part of the book is a tutorial, which gradually introduces the entire GF language in five chapters. The second part describes the use of GF in applications, as well as some issues of linguistic grammar writing. The third part contains a complete reference manual of the GF language, followed by a more summary manual of its standard library (the GF Resource Grammar Library) and

the GF software system (the interactive GF shell and the batch compiler).

GF is a system in constant development since 1998, with new functionalities added every year. The material in this book aims to be stable, so that the most experimental and novel features of GF are omitted. The current syntax of the GF programming language dates back to year 2000, and all GF grammars written since then are still usable with the current tools, except for grammars containing errors that early compiler versions failed to catch. The size of the standard library of GF, the Resource Grammar Library, with all its extensions, exceeds 500,000 lines of GF code in Summer 2010. In addition, millions of lines are included in GF applications, either hand-written or automatically generated from other linguistic resources. All this code makes it imperative to maintain a stability in the language.

Things that will *not* be covered in this book include linguistic theory and the computational metatheory of grammars. These issues are touched upon when they are relevant for the practical use of GF; but any in-depth discussion is deferred to the commented GF Bibliography at the end of the book.

1.2 How to use this book

The book is meant to be usable for teaching in a class, but also for self-studies without a teacher. Doing the exercises and testing them on a computer is in both cases an essential part of learning. The book's web page will give additional material and support for practical work.

The exercises are divided to the following classes:

- Basic, unmarked. Recommended to everyone, should be doable in a few minutes or less than an hour. May be essential for practical understanding of later material.
- Advanced, marked with * . May require extra knowledge or skills, and can take anything from a few minutes to days. Not presupposed later in the book (except perhaps in other advanced exercises).
- Laborious, marked with + . Will probably take more than an hour.

In our experience from teaching GF, the tutorial part (chapters 2 to 6) is possible to cover during a one-week intensive course at graduate student level. The application part (chapters 7 to 10) need not be read in sequence: it is more likely that readers interested in particular projects read the closest corresponding chapter to get ideas for their own design and implementation. The reference manual part need not be read in sequence either: like any reference manual, its purpose is to give quick access to information on different details. Nonetheless, we

have tried to make Appendix C, “The GF Programming Language”, into a text readable from the beginning to the end. Thus, besides all the necessary details, it also gives motivations for the design choices made in the GF language and explains how everything works together. Hence its reading is recommended for anyone who wants to get a deeper understanding of GF.

To assess the level of GF skills, for instance when giving credits to students, we have used projects of different sizes as criteria. To give some examples, we have defined the *one-week level* to be reached when the student has written an implementation of the “Foods” grammar (Chapter 3) for some language. The *three-week level* involves an implementation of the “miniature resource” (Chapter 9), or alternatively a query system with an embedded grammar with at least 50 abstract syntax rules (Chapter 7). The *ten-week level* involves the implementation of a resource grammar morphology and lexicon (Chapter 10) for a new language. The *twenty-week level*, equivalent to a Master’s thesis, involves the complete implementation of a resource grammar (Chapter 10) and a written report.

Many resource grammar projects have actually led to refereed publications and served as introduction to research. Of course, this project gets more and more seldom eligible, as the GF Resource Grammar Library grows with new languages. Ambitious students whose language is already “reserved” can then be given alternative projects, such as extending the coverage of the library (Section 10.8) or building comprehensive applications (Chapters 7 and 8). Whenever possible, students are encouraged to work on their own native languages, or other languages that they find interesting. In application projects (as opposed to resource grammars), it is always interesting to aim for multilingual systems.

The reader can now either dive straight into practical GF programming in Chapter 2, or reflect a moment on natural language processing and GF’s role in it. The discussion is intended to give perspective for the technical chapters and, as we hope, increase the motivation for learning GF. But it contains no technical definitions that will be presupposed later.

1.3 The role of grammars in language processing

How can we make computers process human language? Technology has two kinds of answers to this question. One answer, the traditional one, is that the computer should be given a **grammar** of the language—that is, a set of rules for analysing and producing text and speech.

Another answer, prominent in the last 20 years, is that the computer should be given a lot of **data**, raw linguistic material such as text, from which it can extract processing rules by building **statistical models**, or applying some of the techniques known as **machine learning**. These two answers divide the community of natural language processing into two camps, known as the **symbolic approach** and the **statistical approach**, respectively.

The statistical approach is currently dominating in both research and technology. In areas such as information retrieval, machine translation, and speech recognition, it has delivered useful applications that are able to deal with unrestricted written and spoken language. Grammars have a more theoretical flavour, and their scope is often very limited. However, one of the central goals of this book is to show how grammars can be useful in practice.

Interestingly, the symbolic and statistical approaches to language processing are analogous to two views on language teaching for humans. The traditional school stresses grammars: you learn a language by training yourself to apply a set of rules. The more recent school replaces grammars by exposure to concrete language material: you learn a language by hearing, reading, and using it. The “grammar” remains implicit, and you may be unable to spell out what “rules” you are following, or *why* you say in a certain way and not in another way. Your knowledge of language is implicit, pretty much like walking.

When humans start to learn their first language, they obviously cannot do it by grammars but only by exposure. When humans learn new languages at a later age, some learners prefer the explicit rules found in grammars whereas some prefer direct exposure and practice. For a computer, the first language is the machine language of its processor. It is not learnt: it just is there, as a precondition for anything the computer can be programmed to do. When a new programming language is “taught” to a computer, this happens via a **compiler**, which is a program based on a grammar of the language—the compiler is definitely not extracted from data.

The mathematics of both compilers and natural language grammars evolved jointly in the 1950’s and 1960’s, within a common theory of formal grammars (see e.g. Chomsky 1956). Despite the shared mathematical models, however, there is a fundamental difference between programming languages and natural languages: for the former, but not the latter, complete rule systems exist. In fact, a programming language is *defined* by its grammar: the language would simply not exist without it having been defined by grammatical rules, either explicitly by a formal grammar, or implicitly by the code for processing it. The

grammar is thus an essential component of programming languages and their compilers. It is also an integral part of any reference manual aimed for human programmers. This book provides a typical example: Section C.6 of the GF reference manual contains a grammar of the GF language.

For natural languages, grammars are artefacts—they are theories formed by observing an already existing system, which may not be entirely coherent. As the early 20th century linguist Sapir put it:

All grammars leak.

(Sapir 1921). Sapir claimed that languages are too complex and irregular to be completely captured by any formal rule systems. Grammars in grammar books, which are written for human readers, are either incomplete or sufficiently vague to hide their incompleteness. But grammars written for computers cannot be vague. Thus they must be either **incomplete** (not covering all of the language) or **overgenerating** (covering expressions that in reality are “ungrammatical”).

Since grammars leak, they are never *sufficient* for a complete understanding of language. Human learners repair the holes by intuitive understanding; computers use approximations, which in the data-driven approaches are regularities learnt from language data by statistics and machine learning. In extreme data-driven approaches, no hand-written grammars are used at all, but everything is learnt from data.

Even though grammars leak, all we need to justify an approach like GF is that grammars are *useful*, both for humans and for computers. For a human, a grammar provides a *shortcut*: since it gives general rules, it can replace a vast amount of concrete training material. Moreover, knowing grammar usually improves the *quality* of the language produced by a human: she makes less mistakes when writing or speaking, compared to a person who has learnt the language by pure exposure. The same applies to mechanical language processing, in particular when dealing with languages that have a higher structural complexity than English, implying that a large part of their valid expressions cannot be found in existing data.

For instance, the French verb has more than 50 different forms, when the English verb has only 5. To find all forms of any given French verb in a set of text data (if we exclude grammar books!) is practically impossible. But it is straightforward, although complex, to write grammatical rules that compute all forms of all verbs. It would be foolish not to use such rules in language processing, even in systems that have statistical components for other things. The strength of statistical language processing is that it can be boot-strapped from data—but this leads,

correspondingly, to the problems due to *sparse data*.

Partly as a consequence of the sparse data problem, statistical processing has difficulties with **long-distance dependencies**. A translation example illustrates this: the English sentence

my father immediately became very worried

is by Google’s translation (<http://translate.google.com/>, 15 August 2010) rendered in French as

mon père est immédiatement devenu très inquiet

which is correct. Changing *father* to *mother* results in the translation

ma mère est immédiatement devenu très inquiet

where *mon* (“my” for masculine nouns) is correctly changed into *ma* (“my” for feminine nouns). However, the completely correct translation would be

ma mère est immédiatement devenue très inquiète

where also “become” and “worried” are rendered in feminine forms, to agree with *mère* (“mother”). Google translation, like many other statistical translation systems, is based on *n-grams*, which are sequences of *n* words for some small *n*. As *ma* and *mère* are next to each other, they fit into the same *n*-gram. But *mère* and *devenu* are separated by two words. The chosen *n* may then be too small (i.e. smaller than 4), or sparse data can mean that this very 4-gram has never been encountered in data. But grammar-based systems have no particular problems with long-distance dependencies. There could be *any* number of words between *mère* and *devenue*, and the gender could still be assigned correctly, as long as the intervening words can be recognized as an adverbial phrase.

The agreement example may sound unimportant, since grammaticality is not crucial for understanding the sentence. So let us take another example. It has to do with **discontinuous constituents**, which are expressions consisting of separate words that can have other words between. The German verb *umbringen* (“kill”) is a typical example. It consists of two parts, *um* (“around”) and *bringen* (“bring”), which are in many verb forms separated by other words, in particular the object. An experiment with Google translate correctly renders

Ich bringe ihn um.

where *ich* is “I” and *ihn* is “him” , as

I’ll kill him.

But changing the object *ihn* to the longer *meinen besten Freund* (“my

best friend”),

Ich bringe meinen besten Freund um.

(“I kill my best friend”) results in

I bring to my best friend.

Again, grammars can easily solve the problem of linking together the parts of discontinuous constituents, whatever the number of words between them.

So, what is the proper division of labour between statistics and grammars? An obvious guiding principle is:

Don’t guess if you know.

(from the title of Tapanainen and Voutilainen 1994). If there is a grammar available, use it. Basic facts such as word inflection and long-distance agreement are well understood in grammars and easy to implement on a computer. This doesn’t mean that grammars manage to do everything. Since grammars leak, there is a problem of **robustness**: what to do with language input that is not covered by the grammar? Just return “syntax error” as compilers do, or try to guess what the input might have meant? Statistical language models usually manage all input by means of **smoothing** techniques.

In addition to robustness, the **cost of grammars** is a common issue, which we will discuss in the next section: this is indeed one of the main problems GF aims to solve. It is sometimes thought that statistical models come free of cost, since they are derived automatically from data. However, this is true only if enough data is available; if not, it can be very costly to collect and properly structure the needed data. Moreover, the automatic derivation of a system such as a translator requires a lot of human knowledge and judgement in the construction of the derivation algorithms and tuning their parameters. Much of this knowledge is grammatical in nature. For instance, the previous problem with French gender could be solved by a processing phase that clusters and classifies many-word phrases in an input so that shorter and more frequent n-grams are created. Designing such processing phases requires knowledge of grammar, even if the grammar is not explicit in the program code but just in the programmer’s mind.

The choice between data-driven and grammar-driven methods depends partly on whether we strive after **coverage** or **precision**. A system whose aim is to produce browse-quality translations of web pages needs coverage, whereas a system aimed for translating instruction manuals with publishing quality needs precision. Coverage and precision cannot be maximized at the same time; at least, no-one has

ever succeeded in this. There is thus a trade-off between them: adding coverage tends to decrease precision, and vice versa.

Coverage is sometimes associated with data-driven methods and precision with grammar-driven methods. But this need not be so. For instance, the OpenCCG English parser¹ combines a wide-coverage low-precision grammar with statistical filtering that adds precision.

Much of the current research in computational linguistics addresses **hybrid systems**, which combine statistical and grammar-based methods. GF is no exception: using GF in hybrid systems is an active topic. In this book, however, we will not enter this topic, but stick to the purely grammar-based methods available in GF itself. We just ask the reader to keep in mind that, because all grammars leak, these methods cannot solve all problems of natural language processing.

1.4 The cost of grammars

One well-known problem with grammars is their cost. A large-scale grammar implementation for just one language may easily require years of work, worth of a PhD thesis or more. It also requires training in linguistics and theoretical knowledge of the target language, in addition to programming skills. This kind of knowledge is not common. Statistical methods, in contrast, are within the reach of almost any programmer with some training in mathematics. What is more, it is a part of the art to let the computer do the job: one can run tests on different values of parameters, so that the optimal values can be chosen without *a priori* knowledge.

Grammars can also be costly in another sense. Once a grammar is there, run-time language processing with it can be slow and computationally expensive. A widely held belief is that an accurate description of natural languages needs at least **mildly context-sensitive grammars**. This is a language-theoretical concept, denoting a class of formal languages between the more well-known classes of **context-free** and **context-sensitive** languages. (Knowing about these concepts is not necessary for reading further; the main thing now is the **complexity** associated to classes of languages.) The worst-case parsing complexity of mildly context-sensitive grammars is $O(n^6)$, which means that the analysis of a sentence with n words may take a time proportionate to n^6 . This should be compared with the parsing complexity of most programming languages—the real success story for grammar-based computing methods—which is just linear, i.e. $O(n)$, due to the way in which programming languages are designed. For many programmers, language

¹<http://openccg.sourceforge.net/>, based on Steedman 2001

processing whose performance is worse than linear feels like a very bad thing to do at all.

One of the main claims we want to make is that GF lowers the cost of using grammars, both in development and at run-time. The development costs are lowered by language features that have proven useful in software engineering in general:

- a **static type system**, which detects many programming errors automatically
- a **module system**, which supports the division of labour
- the use of **functional programming**, which enables powerful abstractions
- the availability of **libraries**, which enables new grammars to build on earlier ones
- **compilers**, which convert GF grammars to other formats
- tools for **information extraction**, which convert linguistic resources from other formats to GF

The first four features are an integral part of the modern design of GF, which follows the model of **typed functional programming languages**, Haskell and ML. GF also borrows features from Java and C++, especially ones that have been proven in large-scale software engineering.

Writing GF grammars can be orders of magnitude more efficient than writing, for instance, context-free grammars (also known as BNF grammars). This is seen by the sheer fact that, when a GF grammar is expanded to a context-free grammar (an often forthcoming task; cf. Section 7.13), its size can grow by a factor of hundreds. This is due to the abstraction mechanism provided by functional programming, which enable the elimination of almost any form of repetition from the source code. Moreover, grammars once written can be reused in the same way as **software libraries**, which is one of the keys to productivity in software engineering in general.

The last two features of GF listed above—compilers and information extraction—are not unique to GF. It is, for instance, well known how various brands of unification grammars (see Section 1.9) are converted to lower-level and more repetitive formats, such as context-free grammars, which can be used for tasks such as speech recognition. As a more unique facility, GF grammars can also be compiled into JavaScript and thereby run on almost any web browser, including mobile telephones.

The run-time performance of GF grammars is constantly improved by optimizing compilers and algorithm development. Theoretically, GF

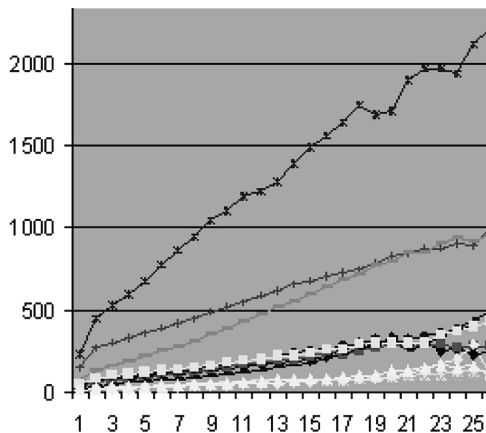


FIGURE 1 GF resource grammar parsing time msec/sentence as a function of sentence length for different languages (by Krasimir Angelov).

is equivalent to **PMCFG (Parallel Multiple Context-Free Grammars, Seki & al. 1991)**, which lies between mildly context-sensitive and fully context-sensitive grammars. Parsing in GF and PMCFG is polynomial ($O(n^k)$) but the exponent k depends on the grammar. Practical grammars, however, are often linear, partly by the grammarian’s own design and partly due to compilation techniques. Figure 1 shows the parsing speed for 12 languages in the GF Resource Grammar Library (Chapter 10). It shows the average processing time per sentence in milliseconds as a function of the number of tokens, for a test corpus covered by the resource grammar. The most demanding language is Finnish, followed by German and Italian; the least demanding ones are the Scandinavian languages and English. The shape of the curve is linear for all languages.

Generation of language is even faster than parsing in GF. This means that GF is particularly well suited for multilingual translation tasks with several simultaneous target languages. Generating text in one language can be done, even with a complex grammar, at a pace of 100,000 sentences per minute using a mid-range laptop computer from 2007.

1.5 Multilinguality

While we believe that GF is an excellent tool for grammar implementation in “traditional” tasks involving one language at a time, the real strength of GF is its **multilinguality**.

A GF grammar can deal with several languages at the same time.

The key to this is the division between two components of grammars: **abstract syntax** and **concrete syntax**. This division is familiar from compilers of programming languages. The abstract syntax is a tree-like representation that captures the semantically relevant structure of language. The concrete syntax relates the tree structure with linear string representations. In compilers, this division is useful because the language has two kinds of “speakers”, who have different needs: the human programmer prefers to read and write strings, whereas the compiler prefers trees. **Parsing** is the process that converts source language strings into internal tree representations. In modern compilers, parsing is just a small fraction of the compilation process. The major part is taken by operations such as name resolution, type checking, optimizations, and code generation, all of which operate on abstract syntax trees.

When a division is made between abstract and concrete syntax, it is possible to equip one abstract syntax with several concrete syntaxes. To take a programming-language-like example, consider a simple arithmetic expression,

2 + 3

This expression has an abstract syntax tree, which is displayed as the middle part of Figure 2. The concrete syntax can be varied *ad libitum*: we have the prefix form used in Lisp,

(+ 2 3)

the postfix form used in JVM (Java Virtual Machine) assembler,

iconst_2
iconst_3
iadd

the corresponding bytecode in binary form

0000 0101 0000 0110 0110 0000

as well as thousands of forms found in natural languages, such as English, French, and Finnish:

the sum of 2 and 3
la somme de 2 et de 3
2:n ja 3:n summa

The core idea of a compiler is that it parses a string by using the concrete syntax of the source language, and converts the resulting tree to a string by using the concrete syntax of the target language. These two steps in a Java-to-JVM compiler are shown in Figure 2. The procedure of converting trees to strings is called **linearization**.

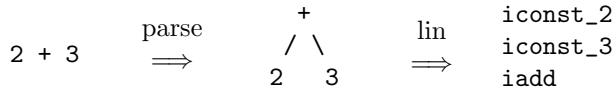


FIGURE 2 Compilation from Java to JVM via abstract syntax.

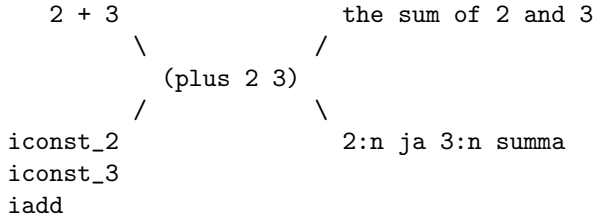


FIGURE 3 A multilingual grammar with Java, JVM, English, and Finnish; abstract syntax in the middle.

The compiler model is precisely what GF uses for translation between languages. The main difference from traditional compilers is that GF is more powerful than the grammars used for programming languages, so that it can also express the rules of natural languages. Another difference from most compilers is that GF grammars are **reversible**: any concrete syntax works for both linearization and parsing. In compilers, one usually doesn't care about linearizing to the source language or parsing the target language, but in GF, two-way translation is the norm. And this generalizes to any number of languages: in a multilingual GF syntax, as shown in Figure 3, one can translate from any language to any other one.

The “magic” of GF lies in the long distance that can be maintained between concrete syntaxes associated with one abstract syntax. This is to some extent seen in the “2+3” examples above, which can all be easily produced in GF. The abstract syntax of “+” expressions is given by a function that forms abstract syntax trees and is defined as follows:

```
fun plus : Exp -> Exp -> Exp
```

The concrete syntaxes are given by linearization rules, one for each language. Java, JVM, and English can be made very simple:

```
lin plus x y = x ++ "+" ++ y
lin plus x y = x ++ ";" ++ y ++ ";" ++ "iadd"
lin plus x y = "the sum of" ++ x ++ "and" ++ y
```

(The Java rule can be refined by adding parameters to deal with operator precedence; see Section 8.1). Finnish requires some more complexity, because expressions are inflected for case. The operands of addition are

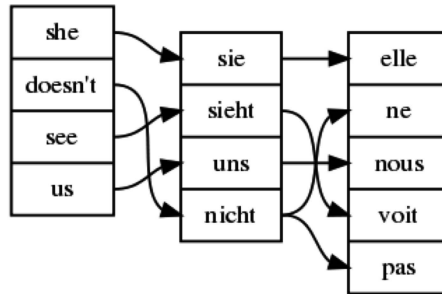


FIGURE 4 Word alignment between English, German, and French.

both in the genitive, and the noun *summa* (in the singular) receives the main principal case *c* of the construction. Without going to details until Chapter 3, we can express inflection by the `table` construct of GF:

```
lin plus x y = table {c =>
  x ! Gen ++ "ja" ++ y ! Gen ++ mkN "summa" ! Sg ! c
}
```

The important thing illustrated by these examples is that the abstract syntax is independent of features like word order and inflection, and can thus be made to fit widely different languages.

For a more traditional linguistic example, consider a very simple sentence: *she doesn't see us*. Figure 4 shows a **word alignment** between this sentence and its French and German equivalents. Equivalence here means that the sentences are linearizations of one and the same abstract syntax tree. The diagram has been produced by a word alignment visualization tool available in the GF grammar development system (Section 2.14).

Without going to details yet, let us point out two features of word alignment. The diagram shows “jumps”, where for instance the word expressing negation is the second word in the English sentence (*doesn't*) but the last word in the German equivalent (*nicht*). Anyone who knows English and German can easily verify that these jumps can be arbitrarily long. An even more interesting thing happens in French, where negation is **discontinuous**: it consists of two words, *ne* and *pas*, which can again appear very far from each other. Dealing with word order differences and discontinuities is one of the strength areas of GF.

Another area of difference between languages is **morphology**, which defines the **inflection** of words. Different languages can have vastly

different sets of forms for corresponding words. For instance, nouns in French have two forms (the singular and the plural) but nouns in Finnish have 30 to 1,000 forms, depending on how one counts. Morphology is also reflected in the syntax of languages, in **agreement**, where the choice of a word form depends on other words occurring in the sentence. Figure 5 shows the sets of strings generated from the same set of abstract syntax trees in English and German. The expressions are built from nouns (*man*, *woman*) with adjectives (*important*), numerals (*one*, *five*), and optional prepositions (*for*, *with*). In English, the only morphological variation is seen in nouns, which vary in number, in agreement to the numeral (*one man* vs. *five men*). In German, adjectives agree with the noun in number as well, but also in gender; moreover, both adjectives and nouns have a case that depends on the preposition (or the absence of it). The diagrams in Figure 5 are produced by yet another tool coming with GF: the compilation of grammars to finite-state automata, and its associated visualization tool; see Section 7.13.

One of the key ideas in multilingual grammars is that an abstract syntax need not care about features like morphology, agreement, word order, and discontinuity. The abstract syntax is, to use linguistic terminology, purely about **constituency**: it tells what the constituents—that is, significant parts—of an expression are, without telling what they exactly look like or what their linear order is.

The compiler equivalent of multilingual grammar applications are **multi-source** and **multi-target** compilers. For instance, **GCC (GNU Compiler Collection)** was originally written to compile the C programming language to the Motorola 68020 machine language. However, since most of its operation is on the level of abstract syntax, GCC has been completed with different parsers and code generators so that it can compile several source languages into dozens of target languages.

One should notice that the multilingual approach of GF is on a more general level than GCC-like compilers: not only the source and target languages, but also the abstract syntax can be varied. Thus we will see many different abstract syntaxes in this book, ranging from linguistic structures (Chapter 9) to fractals (Section 8.9).

1.6 Semantic actions and interoperability

Parsing and linearization are seldom the only components needed in a language processing system, be it a compiler or a natural language processing system. When a system receives an abstract syntax tree from the parser, it is expected to use it for something. In a limiting case it

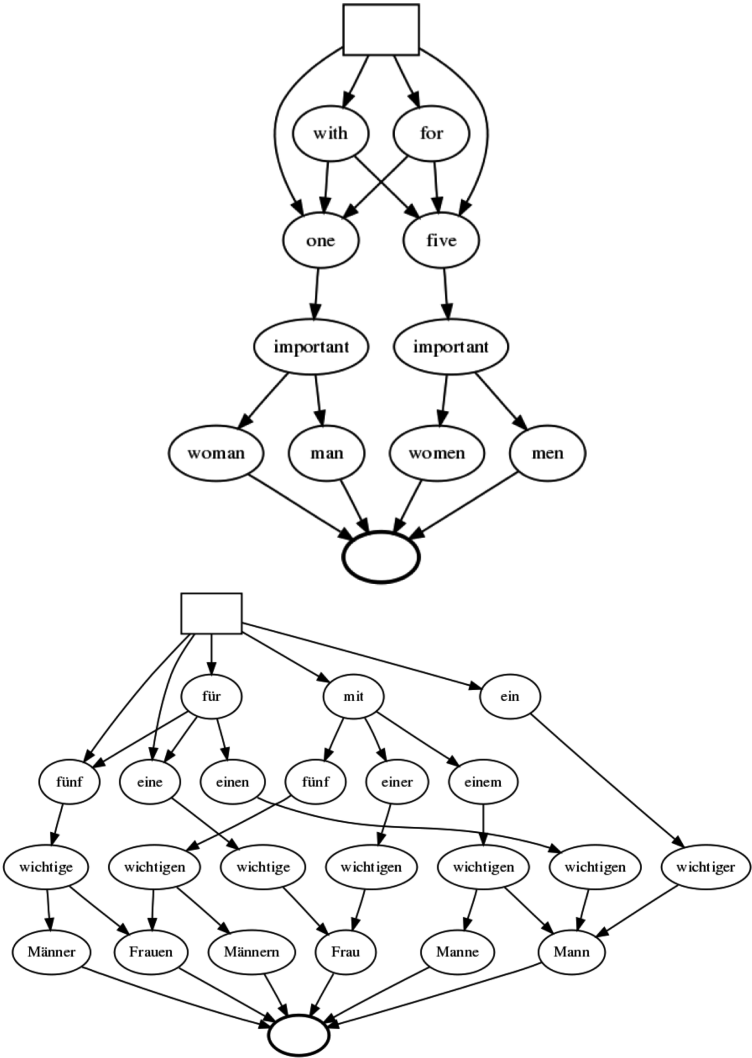


FIGURE 5 Noun phrases with prepositions in English and German.

just linearizes the tree back—which in GF is typically a translation to some other language than the source. In a more general setting, the tree is transformed to some other data object, such as a database query, or maybe interpreted as a command to do something, such as to play a song. Following compiler terminology, we call such tree operations **semantic actions**.

In compiler tools such as YACC, semantic actions are a part of the grammar specification language. More precisely, the grammar specification language is not purely a grammar formalism, but a combination of the **BNF** notation (**Backus-Naur Format**), to define the grammar, and code from a **host language**, such as C in the case of YACC, to define the semantic actions. A text-book example of this set-up is a calculator, which parses expressions built from numbers by arithmetic operations and whose semantic actions compute numeric values. An example YACC-style rule is

```
Exp ::= Exp "+" Exp {return $1 + $3}
```

This rule constructs an expression (**Exp**) from two expressions separated by a + sign. It returns the sum of the values returned for the two operand expressions, denoted by the variables **\$1** and **\$3** (**\$2** refers to the + sign). The semantic action, in curly brackets, can be just any piece of program code of the host language.

A natural language example of semantic actions is **transfer-based translation**, in which a tree received from parsing the source language is transformed to some other tree before it is linearized into the target language. For instance, a passive construction of English may be converted to an active construction when translated to a language like Finnish, where passive constructions with agents are rare and clumsy. The compiler equivalent of transfer is **optimization**. For instance, an optimizing compiler might not generate machine code directly from **2+3**: it might use an optimization called constant folding to first convert the expression tree to a tree representation of **5** and then check whether further constant folding is possible with **5** as an operand; if not, it would generate machine code corresponding to **5**.

Another example of semantic actions is **question answering systems**, which convert questions made in a natural language into answers in the same language. Internally in the system, there is a semantic action converting question trees into answer trees. A calculator can be seen as a simple example of question answering. More complex examples often use a database to search answers.

Yet more complex systems accept questions given in pieces, in several utterances. For instance, a system answering to questions about flight

prices may first take in the city of departure, then the destination, then the date, and so on, until a flight is fully specified. After each utterance by the user, the system may prompt her for more information. Such systems are called **dialogue systems**. They play an important role in the research of computational linguistics, since their construction involves “everything” that is needed in language processing, ranging from speech recognition via parsing and semantics to natural-language generation and speech synthesis.

In all systems involving both grammars and semantic actions, **interoperability** between the grammar formalism and the host language is needed. The host language has to be able to receive trees from the parser, manipulate trees, and send trees back to linearization. In YACC-like grammar formalisms, interoperability is guaranteed by the integration of a grammar formalism and a host language in one and the same specification. This happens, however, at the cost of **portability**: a YACC description works only for C, a JavaCup description only for Java, and so on. Moreover, the grammar is not cleanly separated from the rest of the processing program. This means, among other things, that the complexity guarantees for the parser no longer hold, as the semantic actions can have any complexity. It also means that the grammar may be unusable for any other purpose than the one originally intended. For instance, a grammar that directly computes questions to answers is not usable for the translation of these questions.

In natural language processing, **Definite Clause Grammars** written in Prolog have much the same virtues and problems as YACC-like language descriptions. A parser does not merely parse, but performs a semantic action, which can be an arbitrary Prolog program. The result is perfect interoperability with Prolog, but poor portability and poor separation between the grammar and semantic actions.

GF aims simultaneously for interoperability, portability, and a separation of grammars from semantic actions. The way to achieve this is **embedded grammars**. We keep GF itself pure, so that the only “semantic actions” grammars can do is construct abstract syntax trees. However, we provide libraries that make it possible to access GF grammars from code written in other programming languages: at the time of writing, C, Haskell, Java, JavaScript, and Python. In addition to making it easy to build GF applications, this technique guarantees that GF is not a blind alley: writing a grammar in GF leaves it open to build the other parts of the system in any programming language, and also to change from one host language to another without changing the GF component.

1.7 Application grammars and resource grammars

GF was originally designed for multilingual applications, in which the abstract syntax is designed to function as a semantic model. This abstract syntax has many concrete syntaxes, and linearization from abstract syntax trees maintains translation equivalence. Such grammars are called **application grammars**, since they are typically tailored for specific applications, such as mathematical exercises, tourist phrase-books, or dialogue systems. Each application has a specific domain, and the domain-specificity makes it easier to guarantee translation equivalence. One aspect of this is **word sense disambiguation**. For instance, the English adjective *odd* has several dictionary equivalents in French: *bizarre*, *étrange*, *dépareillé*, *impair*. If we know that the domain of translation is arithmetic, we know that *impair* is the correct choice. Another aspect is the domain's idiomatic syntax. For instance, when mathematical exercises are written in English, the imperative is used: *show that the sum of two odd numbers is even*. In French usage, the idiom is the infinitive: *démontrer que la somme de deux nombres impairs est paire*. General-purpose translation systems, aiming at large coverage, have difficulties with both of these aspects.

Application grammars can also be called **semantic grammars**, because their abstract syntax is modelled after some semantic structure. Such structures may be first defined in mathematical logic or in an **ontology** (in the modern sense of knowledge representations on the Web), but can usually be easily ported to GF. They can be contrasted with **syntactic grammars**, whose abstract syntax has more modest ambitions. What is captured by syntactic grammars is just **syntactic structures**, or **grammatical operations**, such as the formation of sentences from verbs and noun phrases. Syntactic grammars deal with what is usually meant by “syntax” in linguistics. They tend to cover larger parts of language than semantic grammars, since all semantic models known up to the date are very limited. The relation between semantic and syntactic grammars is illustrated by Figure 6, which shows two abstract syntax trees obtained by parsing the English sentence *the sum of 2 and 3 is prime* in GF.

The first tree in Figure 6 is syntactic and much more detailed: it shows that the sentence is a present-tense positive sentence built by predication from a noun phrase and an adjective. The noun phrase is formed with a definite article from the noun *sum*, and further modified by a prepositional phrase formed with *of* and the conjunction of two number expressions, conceived as proper names. These concepts belong to syntax, as it is traditionally understood.

```

UseCl
  (TTAnt TPres ASimul) PPos
  (PredVP
    (AdvNP
      (DetCN (DetQuant DefArt NumSg) (UseN sum_N))
      (PrepNP of_Prep
        (ConjNP and_Conj
          (BaseNP (UsePN n2_PN) (UsePN n3_PN))))))
    (UseComp (CompAP (PositA prime_A))))

Prime (Sum (Num 2) (Num 3))

```

FIGURE 6 A syntactic and a semantic tree for *the sum of 2 and 3 is prime*.

The second tree in Figure 6 is semantic and less detailed: it shows the application of a predicate **Prime** to a term formed by the two-place function **Sum** from two numeric terms. These concepts belong to logic, rather than syntax. The structure is much simpler than in the first tree. But it captures precisely what is essential for using the sentence in mathematical exercises or query systems.

At the same time as simpler, the semantic tree is more abstract than the syntactic tree. It abstracts away from syntactic details such as tenses and articles, or the fact that the primality predicate is expressed by using an adjective. This means that it also permits more variation in concrete syntax. In some languages, such as Finnish, the predicate **Prime** is usually not expressed by an adjective but by a noun, *x on alkuluku*, equivalently to *x is a prime-number*. The syntactic tree in Figure 6 could not, in a properly written grammar, have a linearization that uses a noun instead of an adjective. For the semantic tree, however, this is no problem.

Semantic grammars aim at language descriptions that are semantically accurate at the same time as syntactically correct. To see what this means, we need some GF notation. We want to formulate the concrete syntax rule of the predicate **Prime**. For English, we can write a linearization rule as follows:

```
lin Prime x = x ++ "is" ++ "prime"
```

which means that we concatenate the argument **x** with the words "is" and "prime". In Finnish, we can write

```
lin Prime x = x ++ "on" ++ "alkuluku"
```

We don't need to be concerned that *prime* is an adjective and *alkuluku* is a noun.

(a) A hand-written rule.

```
lin Prime x = \\ord,mod =>
  let
    ist = case <mod,x.n> of {
      <Ind, Sg> => "ist" ;
      <Ind, Pl> => "sind" ;
      <Conj,Sg> => "sei" ;
      <Conj,Pl> => "seien"
    }
  in case ord of {
    Main => x.s ! Nom ++ ist ++ "unteilbar" ;
    Sub => x.s ! Nom ++ "unteilbar" ++ ist ;
    Inv => ist ++ x.s ! Nom ++ "unteilbar"
  }
```

(b) The same rule using the resource grammar in full linguistic detail.

```
lin Prime x = UseCl
  (TTAnt TPres ASimul) PPos
  (PredVP x (UseComp (CompAP (PositA unteilbar_A))))
```

(c) The same rule using the high-level resource grammar API.

```
lin Prime x = mkS (mkCl x unteilbar_A)
```

FIGURE 7 The rule for the predicate *prime* in German in three ways.

However, variations in concrete syntax can become very complex. Sentence building typically involves agreement, mood, and word order issues. For instance, the sentence *2 is prime* has three variants in German, depending on whether it is a main clause (*2 ist unteilbar*) a subordinate clause (*2 unteilbar ist*) or an inverted clause (*ist 2 unteilbar*); see Figure 7 (a). All this soon becomes difficult to keep track of. If one believes that small-scale application grammars are faster to write than large-scale linguistic grammars, one can get disappointed, since so many linguistic problems arise already in very small grammars! As a modest example of this, Figure 7 (a) shows a GF rule for the predicate *prime* in German, taking into account mood variation (indicative or conjunctive), number agreement (where the expression *x.n* gives the grammatical number of the argument *x*), word order (the three-value parameter *ord*), and the nominative case of the argument. This rule is still far from general, as it doesn't cover negation, other tenses than the present, and agreement in person (other than the third).

Now, the same linguistic problems that have to be solved in mathe-

matical grammars appear again in dialogue system grammars, in tourist phrasebook grammars, and so on. This seems to indicate that it is really a waste of time to write application grammars: one should rather solve general linguistic problems such as agreement and word order once and for all in one application-independent grammar. However, such application-independent grammars cannot be made as accurate semantically and idiomatically as application grammars. A considerable amount of transfer programming is needed if one wants high-quality translation with general-purpose grammars.

Luckily, GF has a way out of this dilemma. Both semantic and syntactic grammars can obviously be written in GF; what is more, syntactic grammars can be used as **libraries** for writing semantic grammars. These libraries are called **resource grammars**. The first tree in Figure 6 is a tree in the **GF Resource Grammar Library**. Resource grammar trees can be used for defining concrete syntax in application grammars. For instance, the predicate **Prime** can be implemented as in Figure 7 (b), using the variable **x** of the predicate as the noun phrase in the resource grammar tree. The adjective **prime_A** is replaced by **unteilbar_A**, which is defined in the German resource morphology. The English and German implementations look exactly the same, except for the adjective **prime_A** vs. **unteilbar_A**.

The use of abstract syntax trees of one grammar in the concrete syntax of another grammar is known as **grammar composition**. The end result is similar to writing the rules by hand. But the engineering advantage is that a **division of labour** is achieved:

- The resource grammar is written by a linguist, who knows the rules of agreement, word order, etc.
- The application grammar is written by a domain expert, who knows the terminology, domain idioms, etc.

The rule for **Prime** in Figure 7 (b) is still quite complicated, with lots of linguistic concepts and long function names. It has the level of granularity of the smallest grammatical functions that together make up adjectival predication. Normally, the user of the application grammar does not need this granularity. She will rather follow the **Resource Grammar API (Application Programmer's Interface)**, which allows her to write as in Figure 7 (c). The API ignores much of the internal structure (such as the tense, just providing the present tense as default), and uses uniform function names of form **mkC** to construct expressions of type *C*.

As for syntactic combinations, the API is the same for all languages. Thus for the English grammar, we can write

```
Prime x = mkS (mkCl x prime_A)
```

and for Finnish,

```
lin Prime x = mkS (mkCl x alkuluku_N)
```

Thus only the content word (*unteilbar*, *prime*, *alkuluku*) needs to be varied.

The current Resource Grammar Library has around 200 grammatical functions implemented for 16 languages. For each of these languages, the library also has a complete **inflectional morphology**, that is, a set of functions that are capable of producing all forms of all words. The content words for *prime* can be defined as follows, using the inflection functions for each language:

```
prime_A      = mkA "prime"
alkuluku_N   = mkN "alkuluku"
unteilbar_A  = mkA "unteilbar"
```

While the English `mkA` function needs to produce only one form, the German `mkA` produces more than 20 and the Finnish `mkN` more than 30.

In this book, we will cover both application grammar writing and resource grammar writing. We will start with application grammars, since they are simpler and lead sooner to interesting uses (Chapters 2 to 4). Using the Resource Grammar Library is covered in Chapter 5. Library-internal grammar writing is covered as a more advanced task in Chapters 9 and 10; just using the library doesn't require knowledge of how it is implemented.

1.8 History of GF and its applications

The first steps leading to GF can be found in (Ranta 1991, 1994), attempting to build an integrated formalization of natural language syntax and semantics within constructive type theory (Martin-Löf 1984). The result was a theory reminiscent of Montague grammar (Montague 1974), but with constructive type theory enabling a new analysis of pronouns and other anaphoric expressions, most notably in the so-called donkey sentences such as *every man who owns a donkey beats it*, which were usually studied separately from type theory by e.g. Kamp (1981).

Type-theoretical grammars were first implemented in the proof system ALF (Magnusson 1994) and later in Standard ML and Haskell. The first implementation of GF as a programming language of its own was presented in February 1998 within the project "Multilingual Document Authoring" at Xerox Research Centre Europe in Grenoble. The project had the slogan "create a document in a language you don't

know while seeing it evolve in a language you know” (Dymetman & al. 2000). The concept was familiar from the WYSIWYM system of Power & Scott (1998), but the aim of GF was to make authoring systems customizable with declarative grammar definitions, and also to complement generation with parsing.

Since 1999, the main development site of GF is at Chalmers University of Technology and University of Gothenburg. In the new environment, GF was developed as a functional programming language, with a type system and operational semantics defined in the reference article (Ranta 2004). The computational complexity was established by Ljunglöf (2004), leading to improved parsing algorithms (Burden and Ljunglöf 2005, Angelov 2009). The growing size and complexity of grammars, often written by groups of several authors, created the need of a module system (Ranta 2007). Finally, the portable “GF machine language”, PGF, was created for the needs of efficient run-time processing and embedded grammars (Angelov & al. 2009).

The resource grammar project started in 2001. In 2006, it was estimated that the GF system itself and the GF Resource Grammar Library were equal in size, in terms of both code (40k lines) and human effort (3 person years). After further developments in the library and a refactoring of the GF system, they are drifting apart in a healthy way (25k lines for the system and 500k lines for the library, in Summer 2010).

Applications built in parallel with the basic technology have partly determined its focus. Hallgren and Ranta (2000) built a natural-language interface to the proof editor Alfa. Hähnle & al. (2002) started the development of an authoring system for software specifications, released with the verification tool KeY (Beckert & al. 2006). This system comprised English and German and the formal language OCL (Object Constraint Language, Warmer and Kleppe 1999). It used the Resource Grammar Library and extended GF grammars with natural language generation techniques (Burke and Johannisson 2005). The authoring interface was based on the work of Khegai & al. (2003). The common idea in all these applications was multilingual translation via abstract syntax. This line of work was continued in the WebALT translation system that renders mathematical exercises in seven languages (Caprotti 2006) and integrates GF with a computer algebra editor (WebALT Consortium 2006).

The European research project TALK (2004–2006) on dialogue systems introduced GF in spoken language applications. The aspects that were covered were dialogue management via abstract syntax (Ranta and Cooper 2004), multimodality combining speech with pointing

(Bringert & al. 2005), multilingual resource grammars (Ljunglöf & al. 2006), language models for speech recognition (Jonson 2006, Bringert 2007b), and grammar engineering (Lemon and Liu 2006, Perera and Ranta 2007). All these ingredients were combined in the multimodal and multilingual dialogue system TramDemo (Ljunglöf & al. 2006).

The European project MOLTO (“Multilingual On-Line Translation”, March 2010–2013) will develop GF as a tool for publishing-quality multilingual translation on the web. While still focusing on specialized domains, MOLTO tries to enhance the productivity of GF programming and also combine GF grammars with statistical translation techniques to improve robustness and to learn grammars from data.

The bibliography of this book gives a comprehensive list of publications in the GF project, with short descriptions of each piece of work.

1.9 Related work

GF is rooted in at least four research traditions:

- logic: type theory and logical frameworks
- formal linguistic syntax
- compiler construction
- functional programming

Combining logic and linguistic syntax is not new; GF can actually be seen as an implementation and generalization of the grammar formalism implicit in **Montague grammar** (Montague 1974), which has for four decades been the standard approach of combining syntax and semantics. One of the original motivations was to enable the analysis of anaphora in grammars, and thus cover questions treated in **Discourse Representation Theory (DRT)**, (Kamp 1981).

The concepts of abstract and concrete syntax come from computer science (McCarthy 1962, Landin 1967). Abstract syntax is the central datastructure in compilers: most compiler phases (type checking, instruction selection, optimizations) operate on abstract syntax trees (Appel 1998). Thus we have pointed out above that the structure of a multilingual GF grammar is actually the same as the structure of a multi-source multi-target compiler, such as GCC (GNU Compiler Collection, Stallman 2004).

In linguistics, the distinction between abstract and concrete syntax was suggested by Curry (1961), under the headings of **tectogrammatical** and **phenogrammatical** structure. This architecture was implicit in Montague grammar (Montague 1974), but took a long time to get support in linguistics. The current millennium has seen a wave

of interest in Curry-style grammar architecture, exemplified by GF and also by ACG (Abstract Categorical Grammar, de Groote 2001), HOG (Higher-Order Grammar, Pollard 2004), and Lambda Grammar (Muskens 2001). Even though the possibility of multilingual grammars was already mentioned in Curry (1961), the other Curry-based formalisms have focused on other aspects than multilinguality.

The most well-known grammar formalisms used by computational linguists are DCG (Definite Clause Grammars, Pereira and Warren 1980), LFG (Lexical-Functional Grammars, Bresnan 1982), HPSG (Head-Driven Phrase Structure Grammars, Pollard and Sag 1994), TAG (Tree-Adjoining Grammars, Joshi 1985), and CCG (Combinatory Categorical Grammar, Steedman 1988, 2001). With the exception of TAG and CCG, all these formalisms are based on the operation of **unification**. None of them has an explicit notion of abstract syntax comparable to the Curry-based formalisms. But they do use abstract representations as the basis of semantics. QLF (Quasi-Logical Forms) is the representation used in the DCG-based Core Language Engine (Alshawhi 1992), whereas LFG has F-structures. MRS (Minimal Recursion Semantics, Copestake & al. 2005) is a formalism that has been used in HPSG, LFG, and TAG.

One important application of QLF and MRS is in translation systems—not directly as an interlingua, but as the level on which transfer is defined. Thus different languages have different QLF/MRS representations, and explicit transfer functions convert between them.

The programming language Prolog has built-in support for unification and has therefore been a popular implementation language for grammar formalisms. In particular, DCG's can be implemented as embedded grammars in Prolog. Since Prolog is a general-purpose language, it provides a platform for building applications where grammars are integrated. This has been shown in numerous textbooks, including Pereira and Shieber (1987) and Blackburn and Bos (2003). The Regulus platform (Rayner & al. 2006) applies these ideas for dialogue systems. Regulus moreover provides a compiler from definite clause grammars into language models for speech recognition in the Nuance format (<http://www.nuance.com/>). This was a direct inspiration for corresponding developments in GF (see Section 7.13).

The NLTK system (Natural Language Toolkit, Bird & al. 2009), is a set of tools for language processing in Python. It supports context-free grammar writing and parsing natively, but since 2010, NLTK also has GF bindings, which enable the embedding of GF grammars in Python programs.

The Zen toolkit (Huet 2005) is a library for morphology and lexicon implementations in the OCAML programming language. Because of the similarities of OCAML and GF, morphology descriptions can often be ported between Zen and GF. Zen moreover inspired the development of a similar library for Haskell, entitled Functional Morphology (Forsberg and Ranta 2004).

Special-purpose languages for natural language grammar writing include NL-YACC (Ishii & al. 1994), which comes with a YACC-like parser generator and supports context-free grammars, the LKB system for HPSG (Lexical Knowledge Builder, Copestake 2002), and the XLE system for LFG (Xerox Linguistic Environment). Of these systems, NL-YACC and LKB are open-source like GF, whereas XLE is proprietary.

The idea of large resource grammars as a basis of smaller application grammars is inherited to GF from CLE (Core Language Engine, Alshawi & al. 1992, Rayner & al. 2000). The GF Resource Grammar Library has used the CLE grammars as a benchmark for coverage. Regulus resource grammars are a more direct continuation of the work in CLE, using a similar formalism, but Regulus grammars are more narrowly focused on speech applications. As their grammar specialization method, CLE and Regulus use a technique called explanation-based learning (Rayner & al. 2000), which gives similar results as GF's grammar compilation but in a different way.

The multilingual resource grammar package of CLE contained four languages, whereas Regulus has currently seven. Other multilingual resource grammar packages are the LinGO Matrix (Bender and Flickinger 2004, later renamed to DELPH-IN) written in HPSG, and Pargram (Butt 2003) written in LFG. Both LinGO/DELPH-IN and Pargram aim at large-coverage grammars usable for text parsing, rather than specializable resources. Some of these grammars are much more comprehensive than the GF resource grammars. Extended with suitable relaxing techniques for dealing with out-of-grammar input, these grammars can be used for parsing arbitrary text, in particular in English.

Statistical language processing is covered by text books such as Jurafsky & Martin (2000) and Nugues (2006). The modern statistical approach to machine translation was pioneered by Brown & al. (1990); Lopez (2008) gives a survey of the contemporary issues, including some uses of grammatical information.

Part I

A GF Tutorial

Basic concepts of multilingual grammars

This chapter introduces the basics of multilingual grammars and GF. It covers the following topics:

- BNF grammars and their use in the GF system
- GF functionalities: parsing, generation, translation
- abstract vs. concrete syntax
- abstract syntax trees vs. parse trees
- limitations of the BNF format
- string-based GF grammars as a generalization of BNF
- the module structure of GF
- free variation
- limitations of string-based GF
- visualization of trees and word alignments
- lexing, unlexing, and character encoding

2.1 The BNF grammar format

The **Backus-Naur Format (BNF)**, also known as **context-free grammars**, is probably the most widely known grammar formalism. It is used both in computer science, to specify programming languages in compilers, and in linguistics, to provide an introduction to natural language syntax. Even large-scale natural language grammars are sometimes written in BNF—for instance, many speech recognition systems support language models defined in BNF. Also GF supports BNF grammars as an alternative, simplified notation for full GF grammars.

Full GF is in many ways more powerful than BNF. But due to its familiarity and simplicity, BNF provides a smooth introduction to the more general concepts. Therefore we will start the tutorial with a BNF

```

Pred.      Comment ::= Item "is" Quality
This.      Item     ::= "this" Kind
That.      Item     ::= "that" Kind
Mod.       Kind     ::= Quality Kind
Wine.      Kind     ::= "wine"
Cheese.    Kind     ::= "cheese"
Fish.      Kind     ::= "fish"
Very.      Quality  ::= "very" Quality
Fresh.     Quality  ::= "fresh"
Warm.      Quality  ::= "warm"
Italian.   Quality  ::= "Italian"
Expensive. Quality  ::= "expensive"
Delicious. Quality  ::= "delicious"
Boring.    Quality  ::= "boring"

```

FIGURE 8 A GF-readable BNF grammar for comments about food.

grammar. We will show how the grammar can be used in the GF system, and then point out why we want something more powerful, in particular, if we want to deal with many languages simultaneously.

Our first example is the grammar `foodEng.cf` shown in Figure 8. It is a grammar for making comments about food, for example,

this Italian cheese is very expensive

This grammar can be used as a part of a larger project, for instance, an electronic phrasebook for tourists running on mobile phones; Section 7.11 shows such an application in action. In this book, we will return repeatedly to the “food grammar” and make extensions and variations of it.

Each line in a BNF grammar is a **labelled rule**, which defines a **production** for some **category**. The general form of a rule is

Label . Category ::= Production

where a production consists of categories (unquoted identifiers) and **tokens** (quoted strings).

A BNF grammar can be seen as a declarative program, which defines two operations: **generation** and **parsing**. To generate with a grammar is to follow the productions starting from some category, say **Comment**, and build a sequence of tokens by recursively generating a sequence of tokens for each of the categories in the production. In the inverse operation of **parsing**, one starts from a sequence of tokens and finds out whether and how it can be built by using the rules of the grammar.

The result of parsing is a set of **trees**, which can be constructed either from rule labels or from categories. For instance, the string

this Italian cheese is expensive

corresponds to the tree

Pred (This (Mod Italian Cheese)) Expensive

as built from the rule labels. A graphical visualization of this tree is shown in Figure 10 (a).

The labels given to rules are a special feature of GF and play a crucial role in constructing trees. But they are not found in all BNF notations outside GF. Even in GF, the labels can be omitted from the source code, and the system then automatically generates labels for each rule.

Labels and categories can be just any names, as long as they are valid **identifiers**. An identifier in GF is a letter followed by a sequence of letters, digits, and characters ' or -. Programmers should of course use logical and informative identifiers, indicating the role of each category and label.

Since `foodsEng.cf` is meant as a semantic grammar in the sense of Section 1.7, it uses semantic categories rather than syntactic ones. There is of course a correspondence between these semantic categories and familiar syntactic ones. Thus a **Comment** is syntactically a sentence, an **Item** is a noun phrase, a **Kind** is a common noun, and a **Quality** is an adjective. In this chapter, we stick to the semantic names. In Chapter 5, we will set them in precise correspondence with the syntactic categories of the resource grammar.

For most labels in `foodsEng.cf`, plain English words are used. The label **Pred** used for the first rule stands for **predication** (predicating a quality of an item, i.e. saying that the item has that quality). The label **Mod** stands for **modification** (modifying a kind by a quality).

2.2 Using the GF system

In order to see GF grammars in action, you must install the GF system, following the instructions in the beginning of this book. The main way in which we use GF in this book is the interactive **GF shell**. It is the program that is invoked by the operating system command `gf` and that executes **GF commands**, such as linearization and parsing. Figure 9 shows what happens when `gf` is launched. The GF logo and some initial information are shown, followed by the **prompt** `>`. In this picture, as elsewhere, we use the symbol `>` for the GF prompt and `$` for the operating system prompt; you are expected to type the text after the

```
shell$ gf
```

```

      * * *
    *       *
  *         *
*           *
*           *
*         * * * * *
*         *       *
*         * * * * *
    *       *       *
      * * *

```

This is GF version 3.2.

License: see `help -license`.

Bug reports:

<http://code.google.com/p/grammatical-framework/issues/list>

Languages:

```
>
```

FIGURE 9 Opening the GF shell.

prompt but not the prompt itself.

The first command you may want to give in the GF shell is `help`,

```
> help
```

It returns a list of commands with short descriptions. If you give it another command name as argument, it gives detailed help on that command. For instance,

```
> help parse
```

The system also shows the CPU time used for executing every command; this is useful information when working with large and complex grammars.

All GF commands have both short names (1 or 2 letters) and long names. The short name is the first letter of the long name, except that, if the long name contains an underscore, the short name has a second letter, which is the one after the underscore. For example, `parse` = `p` and `generate_random` = `gr`. Thus the command `help parse` could also be written

```
> h p
```

In general, short names are handy in actual use, whereas long names are more informative in documents like this book. They are also useful in scripts, i.e. files with sequences of GF commands, because they tell what the commands mean. A list of the most important commands and their usage is given in Appendix E of this book.

2.3 Testing a grammar in the GF system

The first step in testing a grammar in GF is to **import** it in the system. The command for this is **import** = **i** followed by the file name:

```
> import foodEng.cf
linking ... OK
```

The message **linking ... OK** indicates that a grammar datastructure is built in the memory and ready to be used.

After importing a grammar, we can parse strings of words with the command **parse** and the string in quotes:

```
> parse "this Italian cheese is very expensive"
Pred (This (Mod Italian Cheese)) (Very Expensive)
```

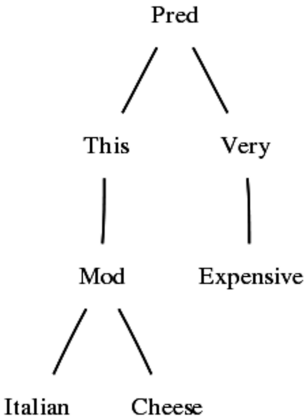
The response of the system is a **tree**, which is given in a parenthesis notation rather than graphically. But the tree can also be seen graphically, as shown in Figure 10 (a). The graphical tree was generated by the GF command **vt** for tree visualization explained in Section 2.14.

The tree returned by parsing is an **abstract syntax tree**, which encodes the partially ordered set of rule applications used for building the sentence. The nodes of the tree are rule labels. Abstract syntax trees are in GF the focal objects of language processing, just like in compilers.

A grammatical representation alternative to abstract syntax trees is **parse trees**. A parse tree doesn't use the rule labels, but has categories as its nodes and tokens as its leaves. Figure 10 shows both an abstract syntax tree (a) and a parse tree (b) for a sentence. This tree was generated by the GF command **vp** for parse tree visualization explained in Section 2.14. This tree has crossing branches, which could be avoided in this very case, but not in GF in general, as we will see later.

Parse trees are commonly used as a pedagogical device, but they are not so useful in language processing as abstract syntax trees are, since they show details that are irrelevant for later processing. For instance, the branch for the word *is* in Figure 10 has no relevance, since it doesn't add anything to the abstract syntax tree. Compilers usually don't build parse trees, but return abstract syntax trees directly and throw away irrelevant information.

(a) Abstract syntax tree.



(b) Parse tree.

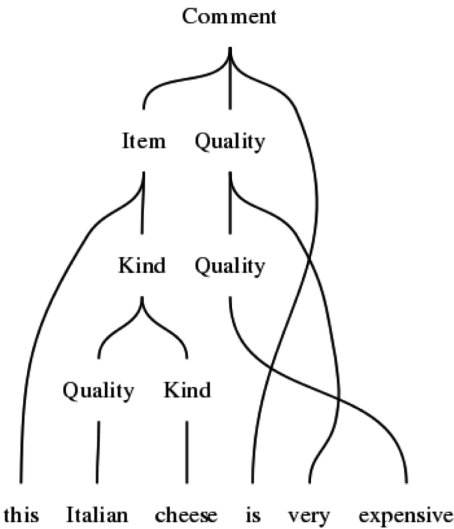


FIGURE 10 Trees for *this Italian cheese is very expensive*.

One very useful functionality of GF is **incremental parsing**. This means that the parser is able to list the possible continuations of a started string. Unlike prediction in systems like T9 for mobile phones, prediction in GF is sensitive to the grammar. It is invoked by pressing the tab key (<TAB>) in the shell. Thus, with the grammar `foodEng.cf`, you get in the beginning of a sentence

```
> p "<TAB>
that this
```

Later in the sentence you get

```
> p "this cheese is <TAB>
Italian boring delicious expensive fresh very warm
```

Giving just enough many letters the parser completes words automatically:

```
> p "this cheese is e<TAB>
> p "this cheese is expensive
```

An abstract syntax tree can be converted to a sequence of tokens by **linearization**. The GF command `l = linearize` takes a tree as its argument and returns a sequence of tokens as a plain unquoted sequence of words:

```
> linearize Pred (This Fish) Delicious
this fish is delicious
```

It is not common to actually write a tree as an argument of the `linearize` command. Trees are usually obtained as results of other commands. One such command is the **random generation** of trees:

```
> generate_random
Pred (That Cheese) Italian
```

GF supports Unix-style **pipes**, marked with the symbol `|`, for sending the output of one command as input to another command. A good way to test a grammar—to see what it actually generates—is random generation piped into linearization (`gr = generate_random`):

```
> generate_random | linearize
that expensive delicious boring wine is expensive
```

Intermediate stages of a pipe can be seen by using the **trace option** `-tr`:

```
> generate_random -tr | linearize
Pred (That Fish) Warm
that fish is warm
```


To generate many examples, you can press the up arrow key to get the previous command and thus avoid typing. This facility can be used for navigating to any place in the command history. For random generation, a specific **number** flag is also available:

```
> generate_random -number=100 | linearize
```

The **gt = generate_trees** for **exhaustive generation** returns all trees up to a certain depth:

```
> generate_trees | linearize
that cheese is boring
that cheese is delicious
that cheese is expensive
that cheese is fresh
...
```

The default depth is 4, but it can be changed by the flag **-depth=n**.

Exercise 2-1. Copy (by hand or from the book's web page) the grammar **foodEng** and test it with some parsing and generation.

Exercise 2-2. Add words and rules to **foodEng**, and test again in GF.

2.4 A BNF grammar for Italian

English is a language for which BNF grammars work better than for many other languages. However, the English grammar in Figure 8 was carefully designed to cover a piece of language for which we can also write an Italian BNF grammar, as shown in Figure 11.

The **Comment** corresponding to *this Italian cheese is very expensive* generated by this grammar is

questo formaggio italiano è molto caro

The main difference between the English grammar in Figure 8 and the Italian grammar in Figure 11 is the choice of tokens: one grammar can almost be obtained from the other by replacing words with their dictionary equivalents. The only rule for which this does not hold is the **Mod** rule: in English, the **Quality** appears before the **Kind** (*Italian cheese*), in Italian, after it (*formaggio italiano*).

Exercise 2-3. Try out the Italian grammar in GF with the same commands as the English grammar.

Exercise 2-4. Write a version of the **food** grammar for a language of your choice. Some rules will probably produce incorrect output, but don't worry: this will be solved later. Just make a list of the things that come out wrong, and use it later when you redesign the grammar.

```

Pred.      Comment ::= Item "è" Quality
This.      Item     ::= "questo" Kind
That.      Item     ::= "quel" Kind
Mod.       Kind     ::= Kind Quality
Wine.      Kind     ::= "vino"
Cheese.    Kind     ::= "formaggio"
Fish.      Kind     ::= "pesce"
Very.      Quality  ::= "molto" Quality
Fresh.     Quality  ::= "fresco"
Warm.      Quality  ::= "caldo"
Italian.   Quality  ::= "italiano"
Expensive. Quality  ::= "caro"
Delicious. Quality  ::= "delizioso"
Boring.    Quality  ::= "noioso"

```

FIGURE 11 A BNF grammar for food comments in Italian.

2.5 BNF grammars and translation

The main idea of GF is that translation goes via abstract syntax trees: the tree obtained by parsing in the source language is linearized into the target language. With `foodEng.cf`, we get

```

> parse "this cheese is expensive"
Pred (This Cheese) Expensive

```

With `foodIta.cf`, we get

```

> linearize Pred (This Cheese) Expensive
questo formaggio è caro

```

What we would like to do is to use a pipe:

```

> parse "this cheese is expensive" | linearize
questo formaggio è caro

```

But this is not really possible, since `foodEng.cf` and `foodIta.cf` are separate grammars, and they cannot be imported in the same GF session. They do use the same labels for similar rules, and therefore they have similar abstract syntax trees (although the parse trees are of course different). But this similarity breaks at one point: the modification rule (`Mod`), which means that intended translation equivalents have different trees:

```

Italian cheese      - Mod Italian Cheese
formaggio italiano  - Mod Cheese Italian

```

This is a problem, which can be solved by proceeding from BNF to the GF formalism.

The part of BNF rules that is relevant for building abstract syntax trees is its **category skeleton**, i.e. the part of the rule obtained by ignoring the terminals. For example, the rules

```
Comment ::= Item "is" Quality
Comment ::= Item "è" Quality
```

both have the same category skeleton,

```
Comment ::= Item Quality
```

and so do the rules

```
Quality ::= "expensive"
Quality ::= "caro"
```

whose skeleton is

```
Quality ::=
```

But the English and Italian modification rules

```
Kind ::= Quality Kind
Kind ::= Kind Quality
```

have different category skeletons—which are the same as the rules themselves, since there are no terminals.

2.6 Abstract and concrete syntax

The step leading from BNF to GF is to separate the rules defining abstract syntax trees from the rules telling how trees are linearized. For example, the predication rule

```
Pred. Comment ::= Item "is" Quality
```

now becomes two rules:

```
fun Pred : Item -> Quality -> Comment ;
lin Pred item quality = item ++ "is" ++ quality ;
```

The first rule, marked **fun**, defines **Pred** as a **function** for building trees. This function has a **type**, which corresponds to the category skeleton of the BNF rule. The type of **Pred** has **Comment** as its **value type** and **Item** and **Quality** as its two **argument types**. As customary in functional programming languages, arguments types and the value type are separated from each other by an arrow (**->**). This is ultimately justified by the technique of **currying**, which means that an *n*-place function is really a 1-place function that returns an *n*–1-place function. This is reflected in the notation where the arrow is right associative,

that is,

$$A \rightarrow B \rightarrow C \equiv A \rightarrow (B \rightarrow C)$$

Currying has several advantages, to which we will return later, in Section 3.9.

The second rule, marked **lin**, defines the **linearization** of **Pred** in terms of the linearizations of its two arguments. These linearizations are represented by the two **variables** **item** and **quality**. These variable names are chosen purely in documentation purpose; **x** and **y** would work quite as well. The variables mark the occurrences of the categories in the original BNF rule. The token "**is**" is in quotes, as in BNF. The items in the production are combined by the **concatenation** operator **++**, which combines sequences of tokens. Leaving out **++** in GF would be an error, because juxtaposition—as customary in functional programming languages—is the notation used for function application.

Now that we have separated **fun** rules from **lin** rules, we have no problems in dealing with the word order difference between English and Italian. The function is

```
fun Mod : Quality -> Kind -> Kind
```

The English linearization is

```
lin Mod quality kind = quality ++ kind
```

and the Italian linearization is

```
lin Mod quality kind = kind ++ quality
```

All that matters in a linearization rule is that it defines a string as a function of the variables that it depends on. In full GF, this condition is generalized from strings to arbitrary **linearization types**, which will be introduced in Chapter 3.

To maintain an order among different kinds of rules, GF has a module system. The two main types of modules are **abstract syntax** and **concrete syntax**. The **fun** rules belong to abstract syntax modules, and the **lin** rules to concrete syntax modules. A **multilingual grammar** is a system with one abstract syntax and any number of concrete syntaxes.

Figure 12 shows the abstract syntax module resulting from the labelled BNF grammar of Figure 8. Figure 13 shows the English concrete syntax and Figure 14 the Italian concrete syntax. In addition to **fun** and **lin** rules, these modules contain some more notation, which will be explained in Section 2.8.

Files with real GF code are recognized from the suffix **.gf**. They can be imported in the GF shell in the same way as **.cf** files:

```

abstract Food = {
  flags startcat = Comment ;
  cat
    Comment ; Item ; Kind ; Quality ;
  fun
    Pred : Item -> Quality -> Comment ;
    This, That : Kind -> Item ;
    Mod : Quality -> Kind -> Kind ;
    Wine, Cheese, Fish : Kind ;
    Very : Quality -> Quality ;
    Fresh, Warm, Italian,
      Expensive, Delicious, Boring : Quality ;
}

```

FIGURE 12 The abstract syntax of the Food grammar.

```

concrete FoodEng of Food = {
  lincat
    Comment, Item, Kind, Quality = Str ;
  lin
    Pred item quality = item ++ "is" ++ quality ;
    This kind = "this" ++ kind ;
    That kind = "that" ++ kind ;
    Mod quality kind = quality ++ kind ;
    Wine = "wine" ;
    Cheese = "cheese" ;
    Fish = "fish" ;
    Very quality = "very" ++ quality ;
    Fresh = "fresh" ;
    Warm = "warm" ;
    Italian = "Italian" ;
    Expensive = "expensive" ;
    Delicious = "delicious" ;
    Boring = "boring" ;
}

```

FIGURE 13 An English concrete syntax for the Food grammar.

```

concrete FoodIta of Food = {
  lincat
    Comment, Item, Kind, Quality = Str ;
  lin
    Pred item quality = item ++ "è" ++ quality ;
    This kind = "questo" ++ kind ;
    That kind = "quel" ++ kind ;
    Mod quality kind = kind ++ quality ;
    Wine = "vino" ;
    Cheese = "formaggio" ;
    Fish = "pesce" ;
    Very quality = "molto" ++ quality ;
    Fresh = "fresco" ;
    Warm = "caldo" ;
    Italian = "italiano" ;
    Expensive = "caro" ;
    Delicious = "delizioso" ;
    Boring = "noioso" ;
}

```

FIGURE 14 An Italian concrete syntax for the Food grammar.

```

> import FoodEng.gf
- compiling Food.gf... wrote file Food.gfo
- compiling FoodEng.gf... wrote file FoodEng.gfo
linking ... OK

```

When working with BNF grammars and `.cf` files in Section 2.3, the entire grammar was contained in one file. With `.gf` files, grammars can be divided in any number of modules. Usually the minimum is two: one abstract and one concrete syntax. The `import` command indicates only the concrete syntax file: all other modules on which it depends are imported automatically. The modules enjoy **separate compilation**, which means that each of them is compiled into its own `.gfo` file (GF Object file). At later occasions, only those `.gf` source files need recompilation to `.gfo` that have changed or are affected by changes in other modules.

2.7 Translating in GF

Since we now have a multilingual grammar, we can use the GF shell for translation. We first import the two grammars:

```

> import FoodEng.gf

```

```
> import FoodIta.gf
```

Then we translate by piping:

```
> p -lang=Eng "this wine is Italian" | l -lang=Ita
questo vino è italiano
```

```
> p -lang=Ita "quel pesce è molto caro" | l -lang=Eng
that fish is very expensive
```

The `lang` flag given to parsing and linearization takes as its value the last three letters of the concrete syntax name, which by convention is the 3-letter ISO 639-3 language code. It is also possible to use the full module name, e.g. `-lang=FoodIta`.

Any number of concrete syntaxes can reside simultaneously in a GF session, as long as they have the same abstract syntax. If no language name is given in a command, all languages are used. So we have, for instance, multilingual random generation:

```
> gr | l
that delicious warm fish is fresh
quel pesce caldo delizioso è fresco
```

The parser likewise tries to find trees with all languages that are in scope.

In addition to translation, a simple “end-user application” that can be built with multilingual grammars is the **translation quiz**, the command `tq = translation.quiz`. The quiz generates random sentences in its `from` language and expects their translations in its `to` language.

```
> tq -from=Eng -to=Ita
Welcome to GF Translation Quiz. The quiz is over when
you have done at least 10 examples with at least 75 %
success.
```

```
* that wine is very boring
quel vino è molto noioso
Yes. Score 1/1
```

```
* that cheese is very warm
questo fromage è molto caldo
No, not questo fromage è molto caldo, but
quel formaggio è molto caldo
Score 1/2
```

When a new **import** command is given in the shell, the grammar imported is assumed to have the same abstract syntax as the already imported ones, if there are any. To import a grammar with a new abstract syntax (including one that just extends the previous one), the command **e = empty** must be used to empty the environment.

Exercise 2-5. Try out translation and multilingual generation with **FoodEng** and **FoodIta**.

2.8 The structure of grammar modules

The abstract syntax module in Figure 12, just as any modules in GF, has two main parts:

- a **module header** indicating the **module type** (**abstract**) and the module name (**Food**);
- a **module body** consisting of **judgements**.

BNF grammars have just one kind of rules, whereas full GF has many kinds, classified under different **forms of judgement**. (The term “judgement” is standard in logical frameworks; it is the common term for definitions and declarations, and could be approximated by the term “rule”.) The module **Food** in Figure 12 contains judgements of three such forms:

- **flags**, **flag definitions**, here of the flag **startcat**, which selects the start category, i.e. the one in which parsing and generation are performed by default;
- **cat**, **category declarations**, telling what categories (types of trees) there are;
- **fun**, **function declarations**, telling what tree building functions there are.

The concrete syntax module in Figure 13 has a module header indicating that the module type is a **concrete** syntax of the abstract syntax **Food**, and the module name is **FoodEng**. Two forms of judgement are employed:

- **lincat**, **linearization type definitions**, saying what type of objects linearization produces from trees of each category,
- **lin**, **linearization rules**, telling how trees are linearized.

A concrete syntax is **complete** with respect to an abstract syntax, if it contains

- a **lincat** for each **cat**,
- a **lin** for each **fun**.

It is **well-typed** if

- all types used in **lin****cat** judgements are valid linearization types,
- all linearization rules define well-typed functions.

Valid linearization types include the type **Str**; more types will be introduced in Chapter 3. Well-typed functions in **lin** rules are defined in terms of the linearization types: if

$$\mathbf{fun} \, f : A_1 \rightarrow \dots \rightarrow A_n \rightarrow A$$

then

$$\mathbf{lin} \, f x_1 \dots x_n = t$$

if t is an object of the linearization type of A under the assumption that each x_i is an object of the linearization type of A_i . Quoted string literals ("**foo**") are of type **Str**, and the concatenation (**++**) of two **Str**s is likewise of type **Str**.

Judgements are terminated by semicolons. A shorthand permits the sharing of the keyword in subsequent judgements, for example,

$$\mathbf{cat} \, C ; D \equiv \mathbf{cat} \, C ; \mathbf{cat} \, D$$

Another shorthand permits the sharing of the right-hand-side in subsequent judgements of the same form

$$\mathbf{fun} \, f, g : A \equiv \mathbf{fun} \, f : A ; g : A$$

We use the symbol $\equiv$ to indicate **syntactic sugar** when speaking about GF. Thus it is not a symbol of the GF language. That an expression form E is syntactic sugar for D means that E is translated to D before it is processed further.

Each judgement introduces a **name**, which is the first identifier in the judgement. Each name can only be introduced once in the same module (that is, as the first identifier in a judgement); local variables such as those following the name introduced in a **lin** judgement can be reused in other judgements.

Names are in **scope** in the rest of the module, i.e. usable in the other judgements of the module (subject to type restrictions, of course). The order of judgements in a module is free. In particular, a name need not be introduced before it is used, and judgements of the same form need not be grouped together.

In addition to judgements, GF files may contain **comments** of the forms

- *-- after two dashes, anything until a newline*
- *{- after left brace and dash, anything until dash and right brace -}*

2.9 On the limitations of BNF grammars

Even though we managed to write **FoodEng** in the labelled BNF format, we cannot do this for GF grammars in general. It is enough to try this with **FoodIta** at the same time as **FoodEng**: we lose an important aspect of multilinguality, namely that the abstract syntax does not presuppose any linear order of constituents.

In general terms, the separation of concrete and abstract syntax allows three deviations from context-free grammar:

- **permutation**: changing the order of constituents
- **suppression**: omitting constituents
- **reduplication**: repeating constituents

The reduplication property is the one that definitely shows that GF is stronger than BNF: GF can define the **copy language** $\{x \ x \mid x \prec (a|b)^*\}$, which is known not to be context-free. The grammar is shown in Figure 15. The term `[]` stands for the **empty token list**, which can be alternatively denoted by the empty string `""`.

```
abstract CopyAbs = {
  cat S ; AB ;
  fun s    : AB -> S ;
    end : AB ;
  a,b : AB -> AB ;
}

concrete Copy of CopyAbs = {
  lincat S, AB = Str ;
  lin s x = x ++ x ;
    end = [] ;
  a x = "a" ++ x ;
  b x = "b" ++ x ;
}
```

FIGURE 15 The copy language in GF.

The other properties have to do with the trees that the grammar can associate with strings: permutation is important in multilingual grammars, and suppression is exploited in grammars where trees carry some hidden information (see next section). In language-theoretical terms, the copy language shows that GF is stronger than context-free in **weak**

generative capacity, i.e. the capacity to define sets of strings. Suppression and reduplication have to do with the **strong generative capacity**, i.e. the capacity to define relations between trees and strings, and not just strings. Thus both pre-modification (as in English) and post-modification (as in Italian) are rules producing context-free sets of strings, but pairs of strings with the same trees for pre- and post-modification could not be produced by context-free grammars.

The BNF notation is severely limited also from the grammar engineering point of view. It gives no support to modules, functions, and parameters, which are so central for the productivity of GF, as we will see throughout this book. We will show that GF can easily produce grammars where, say, ten lines of GF code correspond to hundreds of lines of BNF code. Thus the apparent conciseness of BNF, compared with GF, will soon be outperformed by GF's abstraction mechanisms.

Exercise 2-6.* Define the **reverse** operation as a GF grammar by using one abstract syntax and two concrete syntaxes. Translation between the concrete syntaxes should read a sequence of symbols and return them in the opposite order. For instance, **a b c** is translated to **c b a**.

2.10 Suppression and metavariables

The easiest way to add **pronouns** to the food grammar is to treat them as **Item** expressions. Since only food stuff is addressed, no human beings, it is enough to add the rules

```
fun Pron : Item
lin Pron = "it"
```

But what do pronouns mean? A rough explanation is that they replace other noun phrases, which serve as their **referents**. The referent is typically a noun phrase used a short time before. Thus the pronoun *it* in the text *This wine is Italian. It is very expensive.* refers to *this wine*. The way to express this semantics in GF is to make noun phrases into **hidden arguments** of trees, i.e. arguments that are present in the abstract syntax but suppressed in the concrete syntax:

```
fun Pron : Item -> Item
lin Pron r = "it"
```

The argument **r** suppressed in linearization indicates the referent of the pronoun, which is needed for the full semantic analysis of the pronoun. In parsing, GF returns **metavariables** for suppressed arguments, indicated by question marks:

```
> parse "it is very expensive"
```

Pred (Pron ?) (Very Expensive)

These metavariables can be processed by further analysis of syntax trees, such as **anaphora resolution** in the case of pronouns. For instance, if the parsing occurs in a context where the previous sentence is *this wine is Italian*, the metavariable might be instantiated with **This Wine**. The instantiation itself can sometimes be performed within GF, as Section 6.10 will show. But it can also be done in a separate processing phase and in some other programming language, by using the techniques of Chapter 7.

Another use of metavariables is to control random and exhaustive generation. If a syntax tree with metavariables is given as an argument to these commands, they produce trees by just instantiating the metavariables. Thus

```
> generate_random Pred (This ?) Italian
```

generates only trees of the form *this X is Italian* where *X* is a random Kind. Likewise, translation quiz (Section 2.7) can be given such a term as an argument, which restricts the quiz to a certain format of trees.

2.11 Free variation

GF has a notion of **free variation**: one and the same abstract syntax tree can map to several concrete syntax objects. The operator `|` is used for separating alternatives in free variation. Thus the English linearization rule for **Delicious** in Figure 13 could be extended to

```
lin Delicious = "delicious" | "exquisite" | "tasty"
```

without extending the abstract syntax. All of these words would then also become translation equivalents for the Italian *delizioso* as defined in Figure 14.

In linguistics, a common wisdom is that “there is no free variation”. By this it is meant that whenever two expressions are different, there is some difference in their meaning or at least their stylistic value. In other words, even if the expressions are interchangeable in most contexts, they are not interchangeable in all contexts. For instance, the English negation forms *do not* and *don’t* are usually interchangeable, but do have a stylistic difference which makes one or the other inappropriate in some contexts. Thus they are not in free variation in *all* respects, but only in *some* respects.

Now, an application grammar in GF, as explained in Section 1.7, is a semantic grammar, in which language is described on a certain level of abstraction determined by the abstract syntax. If *delicious* and *exquisite* are equivalent on this abstraction level, they can be treated

```

lin Ticket X Y =
  (((("I" ++ ("would like" | "want") ++ "to get" |
    ("may" | "can") ++ "I get" |
    "can you give me" |
    []) ++
    "a ticket") |
  []) ++
  "from" ++ X ++ "to" ++ Y ++
  ("please" | []) ;

```

FIGURE 16 Some alternative ways to request a ticket from X to Y.

as variants even though they might be different in some other respects. This idea is used heavily in applications that involve user input, for instance, in dialogue systems (see Section 7.15). For instance, a request of a train ticket from X to Y can be expressed in numerous ways. The linearization rule in Figure 16 defines just some of them—14 to be precise. Providing alternatives makes the system easier to use, because the user does not need to learn one exact wording. Implementing them as free variants for one and the same abstract syntax construct makes it easier to implement the semantic actions of the system. It also makes it possible to provide different sets of alternative expressions in different languages.

To understand the example in Figure 16 correctly, one needs to know that `|` binds weaker than `++`.

Exercise 2-7. Write a grammar containing the rule in Figure 16 and, minimally, two names of stations. See all different alternatives by using linearization with the flag `-all` in GF. Add more ways of requesting a ticket, for instance, *I want to go from X to Y*.

Exercise 2-8. Add negative predication to the **Food** grammar, expressed by *is not* or *isn't* in English, and by *non è* in Italian.

2.12 Ambiguity

By using free variation, a tree can be given several linearizations. The opposite direction is even more common: a string can have several trees. This phenomenon is known as **ambiguity**. A typical example of ambiguity is the one known as **PP attachment**. A PP, prepositional phrase, can attach to a sentence or a noun phrase, or many other kinds of expressions. For instance, *with cheese* is a prepositional phrase, and adding the rules

```

fun With : Kind -> Kind -> Kind ;
lin With kind1 kind2 = kind1 ++ "with" ++ kind2 ;

```

to the Food grammar would generate useful expressions such as *fish with cheese*. However, the expression *fish with cheese with wine* could be generated in two ways, corresponding to the trees

```

With (With Fish Cheese) Wine
With Fish (With Cheese Wine)

```

Ambiguous grammars are allowed in GF, and the `parse` command returns all trees obtainable from a string.

Different trees come by nature with different semantics: in the first tree above, for instance, the fish is more closely associated with cheese than with wine. However, if the `With` construction is seen as mere listing, this might be considered as **spurious ambiguity**, which should be eliminated from the grammar. This can be made by dividing the category `Kind` into two, one for simple kind expressions (without *with*) and the other for complex ones (with *with*). Then the rule could be rewritten

```

fun With : Kind -> ComplexKind -> ComplexKind ;

```

and some other rules should be rewritten as well.

Ambiguity may or may not be relevant for translation. For instance, the `With` construction in Italian has the rule

```

lin With kind1 kind2 = kind1 ++ "con" ++ kind2 ;

```

and the ambiguity is hence exactly the same as in English. However, in many cases different trees are translated differently. Suppose, for instance, that the grammar also has questions of the form *do you want this wine*. This sentence has four Italian translations: *vuoi questo vino* (singular, familiar), *vuole questo vino* (singular, polite), *voLETE questo vino* (plural, familiar), and *vogliono questo vino* (plural, polite). In French, there are two translations (merging the last three), and in German, three (merging the second and the fourth). If the abstract syntax is to serve as a reliable interlingua for these languages, it has to provide distinct trees for these translations of *you*. In general, the ambiguity cannot be resolved on the level of a single sentence, but only in a larger context in which the sentence appears.

Exercise 2-9. Add the *with* construction to the Food grammar, in both the ambiguous and the unambiguous versions.

Exercise 2-10.* How many trees are there for an expression of form *Kind with . . . with Kind* for 2, 3, and 4 *with*'s? This series of numbers is

known as the **Catalan numbers**, and it is a common pattern of counting in combinatorics; see http://en.wikipedia.org/wiki/Catalan_number for other examples.

2.13 Remaining problems

The step from BNF to GF has given us two things. First, we are now able to share an abstract syntax between languages that have different word orders. Secondly, we have a higher expressive power even language-theoretically: we can define sets of strings that could not be defined by any context-free grammar. But we have not yet introduced full GF—and we have remaining problems, which will motivate its introduction.

The most obvious problem arises as soon as we try to extend the Food grammar with some new words. If we introduce the noun *pizza*, say, we get problems with **gender** in the Italian grammar. In Italian, nouns and adjectives are associated with two genders—the masculine and the feminine. The nouns introduced so far are luckily all masculine. But *pizza* (the same word in Italian as in English) is feminine. Feminine nouns require their associated adjectives also to be feminine; this is known as **agreement**. Thus we must say *questa pizza italiana è deliziosa* for *this Italian pizza is delicious*, but the current grammar would generate *questo pizza italiano è delizioso*. Since gender is a language-dependent feature, we want to introduce it in the Italian concrete syntax, without changing the abstract syntax and the English concrete syntax.

A similar problem as with the Italian gender arises even with the current Food grammar if we try to port it to new languages, for instance, German. Most languages have morphological distinctions that show up even in the tiniest language fragments, and require a more powerful grammar formalism than the fragment of GF introduced by now. We could call the current fragment **string-based grammars**. This means that all linearization types (`lincat`) are defined as `Str`. String-based GF grammars are more powerful than context-free grammars, but still not as expressive as we really want.

What we have to do is to generalize strings to more complex data structures. Readers impatient to see how this works can now skip to Chapter 3. Those interested in practical programming issues may first take a look at the three remaining sections of this chapter.

Exercise 2-11. (This is a variation of the exercise in Section 2.5, now permitting translation between languages.) Write a concrete syntax of Food for your favourite language. Use random generation to see how

correct it becomes. Don't care about ungrammatical sentences due to gender and related things yet.

2.14 Graph-based visualization and shell escapes

GF has several visualization commands based on graphs that can be rendered by the **Graphviz** tool (available from <http://www.graphviz.org/>). The abstract syntax tree was built by one of them, using the command `vt = visualize_tree`. If you simply do

```
> parse "this cheese is very expensive" | vt
```

GF displays a few lines of graphviz code, which you may or may not be able to understand. But you can save the code in a file by using the command `wf = write_file`, which takes a file name as a flag. This file should have the extension `.dot`:

```
> parse "this..." | vt | wf -file=tree.dot
```

The file `tree.dot` can be processed further by using the graphviz command `dot`. You can do this by first quitting GF; but this is one of the cases where it is more practical to execute shell commands from inside GF, by the prefix `!`, the so-called **shell escape**:

```
> ! dot -Tpng tree.dot >tree.png
```

Now you have produced a `.png` file, which you can open in a viewing program such as a web browser; you may also be able to open it with a shell command, such as `open` in Mac OS X or `eog` in Ubuntu Linux,

```
> ! open tree.png
```

A window should now pop up with a nice abstract syntax tree, similar to the one in Figure 10 (a). If you don't care about saving the `.dot` and the `.png` files (which we did when producing illustrations for this book), you can give the PNG viewer command as a flag to the `vt` command:

```
> parse "this cheese is boring" | vt -view=open
```

Now some temporary files are written invisibly to you, and a window displaying the tree pops up directly.

The command `vp = visualize_parse` is used in the same way as `vt`. The `vp` command shows parse trees, as illustrated above in Figure 10 (b). The command `vd = visualize_dependency` shows **dependency trees**, which consist of links between words partly determined by the abstract tree structure. The default dependencies generated by GF are not always optimal, but they can easily be changed by using configuration files as explained in Appendix E.

Yet another application of graph-based visualization is **word alignment**, invoked by the command `aw = align_words`:

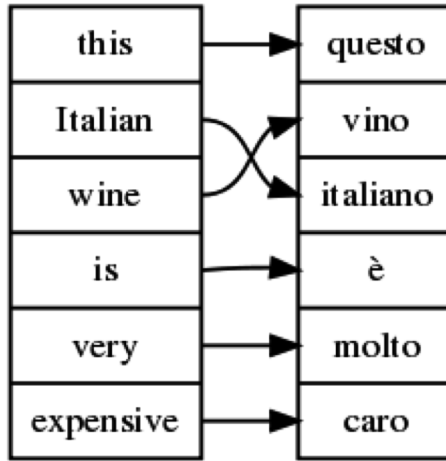


FIGURE 17 A word alignment rendered as a graph.

```
> p "this Italian wine is very expensive" | aw
```

This creates the graph shown in Figure 17. The underlying algorithm uses a syntax tree and its linearizations in two (or more languages). It creates a link between words whose smallest covering subtree is the same. This algorithm is very general and can deal with things like crossing and many-one links. Word alignment graphs can be shown immediately with the `-view` option in precisely the same way as syntax trees.

Yet another use of visualization in GF, dependency graphs, will be explained in Section 4.10.

In addition to the system command escape `!`, the GF shell also provides the **system pipe** escape `?`. The escaped command receives its input from a GF pipe. Thus

```
> gr | 1 | ? wc -c
```

counts the characters of a random-generated string, and

```
> gr | 1 | ? espeak -f
```

sends the string to the speech synthesizer **espeak** and plays it through the loudspeaker, if **espeak** is installed. Other speech synthesizers, such as **festival** and **flite**, can be invoked in the same way.

Exercise 2-12. Create a word alignment graph for a string of length 10 by using the reversing grammar of the exercise in Section 2.9.

Exercise 2-13. See what happens with word alignment if a word in one language is translated by a phrase of several words in another one. Write first a grammar that displays this behaviour.

2.15 Lexing and unlexing

The input of parsing in GF is not just a string, but a list of **tokens**. By default, a list of tokens is obtained from a string by analysing it into **words**, which means chunks separated by spaces. Thus for instance

```
"(12 + (3 * 4))"
```

is split into the tokens

```
"(12", "+", "(3", "*", "4))"
```

The parser then tries to find each of these tokens among the tokens occurring in the grammar, i.e. among the strings that can appear in linearizations. However, a normal grammar for arithmetic expressions, as defined e.g. in Section 8.1, will find only the tokens "+" and "*" among its tokens, and parsing will therefore fail.

The proper way to split the above string into tokens would be

```
("", "12", "+", "(", "3", "*", "4", ")", "")
```

Therefore, GF has a way to select a **lexer**, a function that splits strings into tokens in some desired way, such as following the conventions of a programming language.

A lexer can be given as an option to the string processing command `ps = put_string`:

```
> put_string -lexcode "(12 + (3 * 4))"
( 12 + ( 3 * 4 ) )
> put_string -lexcode "(12 + (3 * 4))" | parse
EPlus (EInt 12) (ETimes (EInt 3) (EInt 4))
```

(This follows the grammar defined in Section 8.1.) The inverse problem, getting correct spacing after linearization, is solved by an **unlexer**, which combines tokens into strings in some desired way.

```
> put_string -unlexcode "( 12 + ( 3 * 4 ) )"
(12 + (3 * 4))
```

Figure 18 shows some lexers and unlexers available in the GF shell.

Exercise 2-14.* Make the reversing grammar of the exercise in Section 2.9 work with unspaced strings of letters **a** to **z**, for instance, to convert the input **"abcdefghi"** to **"ihgfedcba"**.

Exercise 2-15.* Write a compiler of the kind described in Section 1.5 that translates addition expressions to JVM assembler. It is enough to

lexer	description
words	(default) tokens as separated by spaces or newlines
chars	treat each character as a token
lexcode	lex as program code (uses Haskell's lex)
lextext	use conventions on punctuation and capital letters

unlexer	description
unwords	(default) separate tokens by spaces
unchars	glue tokens together without spaces
unlexcode	format as code: spacing
unlextext	format as text: punctuation, capitals

FIGURE 18 Some lexers and unlexers available in the GF shell.

have a couple of integer constants. You can either make parentheses obligatory (as in $(2 + 3)$) or leave them out altogether. The latter alternative can either make expressions ambiguous or assume some form of precedence. Chapter 8 will develop the compiler idea further.

2.16 Character encoding

As a multilingual grammar formalism, GF has to support different writing systems, which require different character sets. The simplest set, **ASCII**, works for very few languages except English. Many European languages can be managed with **iso-latin-1**, which provides ASCII letters with various diacritic marks (such as $\tilde{a}$, $\acute{e}$, $\tilde{n}$). But for many languages, **Unicode** is needed. Thus a string in GF is internally a string of Unicode characters consisting of 32 bits each. This is the way in which GF grammars for Arabic, Chinese, Greek, Hebrew, Hindi, Japanese, Russian, Telugu, Thai, and many other languages with different writing systems have been made to work.

To make 32-bit Unicode characters possible to read and write, different **character encodings** are used. Many text editors, terminals, and web browsers support **UTF-8**, which encodes Unicode by means of variable-size sequences of 8-bit characters. All files generated by GF (for instance, the **.gfo** object files) are encoded in UTF-8. Also input and output in the GF shell is by default expected to be in UTF-8.

The default encoding of **.gf** source files is not UTF-8 but iso-latin-1. This means that the compiler decodes source files to Unicode assuming they are in iso-latin-1. Generated files are always encoded in UTF-8. When they are read back to GF, UTF-8 decoding is performed.

However, since iso-latin-1 is not sufficient for many languages, there is a need to write source files in other encodings. The most popular

```

concrete FoodHin of Food = {
  flags coding = utf8 ;
  lincat Comment, Item, Kind, Quality = Str ;
  lin
    Pred item quality = item ++ quality ++ "है" ;
    This kind = "यह" ++ kind ;
    That kind = "वह" ++ kind ;
    Mod quality kind = quality ++ kind ;
    Wine = "मदिरा" ;
    Cheese = "पनीर" ;
    Fish = "मछली" ;
    Very quality = "अति" ++ quality ;
    Fresh = "ताज़ा" ;
    Warm = "गरम" ;
    Italian = "इटली" ;
    Expensive = "बहुमूल्य" ;
    Delicious = "स्वादित" ;
    Boring = "अरुचिकर" ;
}

```

FIGURE 19 A (simple-minded) Hindi grammar in UTF-8 encoding
(adapted from code by Vikash Rauniyar).

choice is UTF-8. But the compiler must then be informed that the encoding is not iso-latin-1. This can be indicated by the flag

```
flags coding=utf8 ;
```

in the source module. Other available encodings can be found with `help set_encoding`. But UTF-8 is sufficient for most purposes. Figure 12 shows a Hindi grammar for Food; this grammar is not quite correct linguistically, since it does not handle gender agreement.

As an alternative to UTF-8, non-ASCII character sets can be implemented with **transliterations**, which are one-to-one mappings between Unicode characters and ASCII characters (possibly strings of two ASCII characters). For instance, the Hindi word for **Fish** in Figure 19 is in GF's standard Devanagari transliteration represented as "maCII". Section 10.6 explains how transliterations are used in GF and how you

can define your own ones. Transliterations are useful if the available tools don't display UTF-8 characters properly. This may happen, for instance, in your favourite text editor or in the Windows terminal.

Parameters, tables, and records

This chapter generalizes string-based GF grammars to grammars with data structures and thereby completes the picture of what can be done in concrete syntax. It covers the following topics:

- morphological variation
- variable vs. inherent features
- agreement
- parameters, tables, and records
- pattern matching
- data structures in linearization types
- functional programming in GF by operation definitions
- discontinuous constituents

3.1 The problem of morphological variation

In English, nouns (like *wine*) are inflected for number, which is either the singular (*wine*) or the plural (*wines*). The number of the noun varies in agreement with the **determiner**, so that for instance *this* requires a noun in the singular and *these* in the plural. When the resulting noun phrase is used as a subject in a sentence, the verb agrees with it in number:

this wine is Italian

these wines are Italian

Number agreement is theoretically possible to express with a BNF grammar: we just have to duplicate the relevant categories and rules to singular and plural versions. Thus the beginning of the grammar in Figure 8 becomes

```
Comment ::= Item_Sg "is"  Quality
Comment ::= Item_Pl "are" Quality
```

```
Item_Sg ::= "this" Kind_Sg
Item_Pl ::= "these" Kind_Pl
```

In Italian, the number of rules becomes twice as large, because also the gender (the masculine and the feminine) has to be taken into account:

```
Comment ::= Item_Sg_Masc "è"   Quality_Sg_Masc
Comment ::= Item_Sg_Fem  "è"   Quality_Sg_Fem
Comment ::= Item_Pl_Masc "sono"  Quality_Pl_Masc
Comment ::= Item_Pl_Fem  "sono"  Quality_Pl_Fem
```

Writing in this way is of course tedious. It can be avoided by treating the suffixes `_Sg`, `_Masc`, etc. as **parameters** of categories. Then the categories are no longer atomic identifiers, but have a structure consisting of a category and a parameter list. The next step is to permit variables in the parameter list. In such a notation, the four Italian predication rules can be compressed to two:

```
Comment ::= Item(Sg,g) "è"      Quality(Sg,g)
Comment ::= Item(Pl,g) "sono"    Quality(Pl,g)
```

An even larger gain is achieved in the Italian modification rule, where four rules become one:

```
Kind(n,g) ::= Kind(n,g) Quality(n,g)
```

The use of categories with parameters is essentially what happens in **unification grammars**. The above rules are ones that could be written in **Definite Clause Grammars**, although the usual syntax for them in the Prolog programming language would look slightly different.

The use of parameters is also the solution provided by GF to deal with agreement. Since we have divided BNF rules into abstract and concrete syntax components, we have to choose which component parameters belong to. Just looking at the BNF-with-parameters notation above suggests that parameters belong to the abstract syntax, since they are a part of the categories and hence of the category skeleton, from which we extracted the abstract syntax function types in Section 2.6. This would even be technically possible by the use of **dependent types**, which will be introduced in Section 6.3. However, since we have decided to keep the abstract syntax language-neutral, it is not the proper place of parameters, because different languages use different parameters. In this chapter, we will therefore show how parameters are used in concrete syntax, which is indeed the standard way of dealing with them in GF.

3.2 Parameters and tables

Parameters, like everything else in GF, are objects of some types. These types are defined by the grammarian in a concrete syntax. For this purpose, a special form of judgement is used: **parameter type definitions**.

An example is a definition of the type **Number** as containing two values, **Sg** and **Pl**:

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

Parameter types can be used in **table types**, such as

```
Number => Str
```

which is read, “table from numbers to strings”. Table types are GF’s way to formalize **inflection tables**, where each value of a certain parameter type is assigned a string, an **inflection form**. A very simple inflection table is the one of the Italian noun *pizza*:

number	form
singular	<i>pizza</i>
plural	<i>pizze</i>

In GF, this table is represented as the term

```
table {Sg => "pizza" ; Pl => "pizze"}
```

whose type is **Number => Str**.

To access a value in a table, the **selection** operator **!** is used. Thus, for instance,

```
table {Sg => "pizza" ; Pl => "pizze"} ! Pl
```

computes to the string **"pizze"**. Notice that selection works very much in the same way as function application; tables are in fact also called **finite functions**. Also the arrow notation **=>** suggests a similarity with function types.

To express the relation of **computation**, we will use the notation

$$t \Downarrow v$$

to mean that the expression t is computed to v . Thus we can write

```
table {Sg => "pizza" ; Pl => "pizze"} ! Pl \Downarrow "pizze"
```

The symbol $\Downarrow$ is not a part of GF notation, but used for talking about GF, just like the syntactic sugar symbol $\equiv$ in Section 2.8.

Expressions with several parameters, such as Italian adjectives, can be represented as tables from one feature to tables from the next feature, and so on, ending up with strings. Thus Italian adjectives have inflection tables of type


```
Gender => Number => Str
```

where the type **Gender** is defined by

```
param Gender = Masc | Fem
```

The inflection of the adjective *caldo* (“warm”) is represented as

```
table {
  Masc => table {Sg => "caldo" ; Pl => "caldi"} ;
  Fem  => table {Sg => "calda" ; Pl => "calde"}
}
```

The parts of a table separated by semicolons are called **branches**. A branch consists of a **pattern**, a double arrow `=>`, and a term assigning a value to the pattern. It is not necessary to give all parameter-value pairs in separate branches: one can also use variables in patterns to compress branches together. Selection is then computed by **pattern matching**. A constant pattern matches itself, and a variable pattern matches whatever has not been matched in earlier branches. Thus

```
table {g => table {Sg => "grave" ; Pl => "gravi"}}
```

is a table for the adjective *grave* (“heavy”), which is inflected in the same way for both genders. Since the variable `g` is not used in the value assigned to it, it can be replaced by a **wildcard** (`_`):

```
table {_ => table {Sg => "grave" ; Pl => "gravi"}}
```

Tables with only one branch are a common special case. GF provides syntactic sugar for writing them concisely:

```
\\p, ... ,q => t ≡ table {p => ... table {q => t} ... }
```

This notation, with a double backslash `\\`, is analogous to lambda abstraction with a single backslash `\` (see Section 3.6) in the same way as the double arrow `=>` for tables is analogous to the single arrow `->` for functions.

Exercise 3-1. The type **Bool** with the values **True** and **False** is a useful parameter type in all programming, also frequently used in grammars. Define this type and write tables of type `Bool => Bool => Bool` that compute the conjunction and disjunction of any two boolean values.

3.3 Variable vs. inherent features

Nouns in both English and Italian have both singular and plural forms: one and the same noun can be rendered in both the singular and the plural. We say therefore that the number is a **variable feature** of nouns. But gender in Italian (as in many other languages) attaches to nouns in a different way: a noun *is* either masculine or feminine,

instead of having both masculine and feminine forms. We say therefore that gender is an **inherent feature** of Italian nouns.

The distinction between variable and inherent features is reflected in the way words are presented in dictionaries. For example, a dictionary entry for *pizza* might look as follows:

pizza, pl. *pizze*: n.f.

In other words: *pizza* is a feminine noun (n.f.) with the plural form (pl.) *pizze*.

Variable and inherent features are connected to each other via the mechanism of **agreement**. In agreement, a variable feature of one word is determined by an inherent feature of another one. For instance, in the adjectival modification of nouns, the variable gender of the adjective is determined by the inherent gender of the noun. Agreement is thus asymmetric: we say that the adjective agrees with the noun, rather than the other way round.

We will shortly see how variable and inherent features and agreement are formalized in GF. For those readers familiar with unification grammars, we should just point out that in those grammars, there is just one kind of features. Thus agreement is symmetric, in the sense that both parts of an adjectival modification get their features in exactly the same way, as the rule

$\text{Kind}(n,g) ::= \text{Kind}(n,g) \text{ Quality}(n,g)$

shows. This symmetry is also reflected in the way in which words are introduced in the grammar. For instance, the word *pizza* has two entries:

$\text{Noun}(\text{Sg}, \text{Fem}) ::= \text{"pizza"}$
 $\text{Noun}(\text{Pl}, \text{Fem}) ::= \text{"pizze"}$

It is not said explicitly in the grammar that *pizza* is inherently feminine: it just lacks the masculine forms. Actually, these rules don't even say that the forms *pizza* and *pizze* belong together; it is up to the grammarian to add information saying that the abstract syntax trees, or, more generally semantic values, are the same or related for the two word forms.

3.4 Records and record types

A **record** is a data structure that gathers together objects of possibly different types. For instance,

$\{\text{s} = \text{"these"} ; \text{n} = \text{Pl}\}$

is a record that contains the string "these" and the number P1. The record

```
{s = table {Sg => "pizza" ; P1 => "pizze"} ; g = Fem}
```

contains a table from numbers to strings and a gender. Records are in GF used for storing all information that belongs to the concrete syntax of an expression. In particular, inherent features are represented as parameter values stored in records, as suggested by the examples just shown. Variable features, in contrast, appear as arguments of tables.

Just like any object in GF, records have types, called **record types**. The types of the two records above are, in the same order,

```
{s : Str ; n : Number}
{s : Number => Str ; g : Gender}
```

The notational difference between records and record types is that record types use : (as in all type declarations) and records use = (as in all value definitions).

The identifiers left of colons and equality signs are called **labels**. Each part of a record or a record type containing one label and its value or type is called a **field**.

To access a value in a record, the **projection** operator (.) is used, with the label as the second operand. Thus

```
{s = "these" ; n = P1}.n ↓ P1
```

and, in a typical combination with selection,

```
{s = table {Sg => "zia" ; P1 => "zie"} ; g = Fem}.s ! Sg
      ↓ "zia"
```

(Italian *zia*, "aunt").

3.5 Linearization types and agreement

By using tables to encode variable features and records to encode inherent features, we can deal accurately with inflection and agreement in concrete syntax, separately for each language. The first place affected is the linearization types of categories. The linearization rules are then type-checked with respect to the linearization types.

Beginning with English, we assign the following linearization types to the categories of the Food grammar (the complete code can be found in Figure 21):

```
lincat
  Comment = {s : Str} ;
  Item    = {s : Str ; n : Number} ;
  Kind    = {s : Number => Str} ;
```

```
Quality = {s : Str} ;
```

We use records $\{s : \text{Str}\}$, instead of plain strings, as the simplest type. This is a good habit in GF programming, because it makes it easier to modify the code if we later want to add information to linearization types.

Some of the combination rules now more or less write themselves:

```
lin
  This kind = {s = kind.s ! Sg ; n = Sg} ;
  Mod qual kind =
    {s = table {n => qual.s ++ kind.s ! n}} ;
```

Notice that the pattern variable n is used on the right hand side of its branch in the modification rule. The rule could be characterized by saying that the variable number of the noun formed in modification is passed to the noun that is modified.

The predication rule is slightly more complex, but it can still be expressed with the machinery we already have:

```
lin Pred item qual = {
  s = item.s ++
    table {Sg => "is" ; Pl => "are"} ! item.n ++
    qual.s
} ;
```

The middle term is in fact the verb *be*, which is expressed as a table from numbers to strings and applied to the number of the subject.

In Italian, where gender is present, the linearization types are slightly more complex. Noun phrases and nouns have an inherent gender. Moreover, adjectives are inflected for number as well as gender.

```
lincat
  Comment = {s : Str} ;
  Item     = {s : Str ; g : Gender ; n : Number} ;
  Kind     = {s : Number => Str ; g : Gender} ;
  Quality  = {s : Gender => Number => Str} ;
```

The combination rules are again straightforward, although more complex than in English. In modification, for instance, the inherent gender is inherited from the modified noun, and also passed to the modifying adjective.

```
lin Mod qual kind = {
  s = table {n => kind.s ++ qual.s ! kind.g ! n}
  g = kind.g
} ;
```

The complete code for Italian can be found in Figure 24 in Chapter 4. Before getting back to the Italian grammar, we need to take a closer look at the lexical rules in both languages. We will also introduce more efficient ways of writing grammars.

Rules for lexical items are easy to define, for example,

```
lin
  Wine = {s = table {Sg => "wine" ; Pl => "wines"}} ;
  Cheese =
    {s = table {Sg => "cheese" ; Pl => "cheeses"}} ;
  Fish = {s = \_ => "fish"} ;
```

But they are tedious to write, and adjectives in Italian become rather verbose:

```
lin Warm = {
  s = table {
    Masc => table {Sg => "caldo" ; Pl => "caldi"} ;
    Fem  => table {Sg => "calda" ; Pl => "calde"}
  }
} ;
```

We don't want to write tables like this for all words. We rather want to use the regularities of inflection that enable us to specify inflection tables with less code than full tables. To do this, we use functional programming.

3.6 Functional programming in GF: operation definitions

One way to define the linearization of the Italian adjectives *delizioso*, *italiano*, etc, is to copy and paste the rule with *caldo* in the previous section and change the stem of the adjective in each new word needed. This copying and pasting is completely mechanical; therefore, a machine would probably do it better than a human programmer, who is likely to forget or mistype something. The software version of such a machine is a **function**: it is a general rule for producing output from input in a determinate way. This is what could be called the **golden rule of functional programming**:

Whenever you find yourself programming by copy and paste, define a function instead.

Most programming languages have some notion of functions, but the characteristic of **functional programming languages** is that almost any case of copying and pasting can be made into a function.

GF is a functional programming language in two ways. We have already seen that the abstract syntax is a system of tree-building functions, marked with the keyword **fun**. These functions are rather special and limited. They are what in functional programming is called **constructors**: they are used for constructing trees, not for computing anything (however, we will say more about **fun** functions in Section 6.11, which will modify this view). Here we will introduce functions in concrete syntax, defined by a new form of judgement, **operation definitions** (**oper**). To make the distinction clear, functions defined in **oper** definitions will usually be called **operations**.

An operation is just like a function in any (typed) functional programming language: it has a name, a type, and a defining expression, as shown by the general form of **oper** judgements,

$$\text{oper } f : t = e$$

Here is an example: an operation defining the regular inflection of nouns in English:

```
oper regNoun : Str -> {s : Number => Str} =
  \word -> {s = table {Sg => word ; Pl => word + "s"}} ;
```

The name of this operation is **regNoun**. Its type is that of functions from strings to records that we have used for linearizing English nouns. Its definition is one of the hallmarks of functional programming: a **lambda abstract**. A lambda abstract, in general, is an expression of the form

$$\lambda x \rightarrow t$$

It is a function that can be applied to an argument and is computed by the **substitution** of that argument for x in t . Thus in the lambda abstract defining **regNoun**, the string "wine" can be substituted for the variable **word**. The result is the record

```
{s = table {Sg => "wine" ; Pl => "wine" + "s"}}
```

It remains to explain the single plus (+). The single plus in GF is the **gluing operator**, which glues two tokens into one. Thus the expression "wine" + "s" can be computed further to the token "wines".

The "single plus" (+), which combines two tokens into one token, is different from the "double plus" (++), which combines two *lists* of tokens into one list, and whose operands remain distinct tokens. This is usually visible in the spacing of the strings finally generated:

- "foo" + "bar" $\Downarrow$ "foobar"
- "foo" ++ "bar" $\Downarrow$ "foo bar"

But spacing is in the end determined by the lexer and unlexer that relate strings to token lists; see Section 2.15. Hence it is not always the

case that tokens are delimited by spaces.

With the `regNoun` operation, we can now define some of the English nouns in `Food` compactly:

```
lin Wine = regNoun "wine" ;
lin Cheese = regNoun "cheese" ;
```

This is indeed the style of programming we will be aiming at in functional programming: most of the code is written as applications of functions defined by using lambda abstraction. It is only in the definitions of these functions that tables and records are seen. In the ideal case, all such functions are defined in **libraries**.

Functions often take more than one argument, and multiple lambda abstraction is then needed. An iterated abstraction can be abbreviated as a single lambda with several variables,

$$\backslash x_1, \dots, x_n \rightarrow t \equiv \backslash x_1 \rightarrow \dots \rightarrow \backslash x_n \rightarrow t$$

But why are lambda abstracts called “lambda”? In their original environment, mathematical logic, the Greek letter λ (lambda) is used instead of the backslash `\`. The use of the backslash for lambda is a common ASCII notation in functional languages such as Haskell.

Exercise 3-2. Continuing the exercise in Section 3.2, define boolean “and” and “or” as `opers` of type `Bool -> Bool -> Bool`.

Exercise 3-3. Define some other inflection functions, e.g. for English nouns of type `fly - flies` and Italian nouns of type `vino - vini`. What are the expected arguments of these functions?

3.7 The Food grammar revisited

We can now present a larger grammar for comments on food, featuring plural noun phrases formed by *these* and *those*, as well as the noun *pizza*. The abstract syntax `Foods`, shown in Figure 20, is otherwise similar to `Food` in the previous chapter.

We are also ready to give an English concrete syntax, building on the discussion above. It is shown in Figure 21. We have used `oper` definitions to add clarity and modularity to the grammar. Thus we have introduced an operation for the verb *be*, called `copula` in accordance with linguistic tradition. We have also removed all explicit use of tables and records in lexical rules by providing the operations `noun` and `adj`; this shortens the code a little, but also makes it more robust since we could now change the linearization types of `Kind` and `Quality` without changing all lexical rules; see Section 4.2 for this.

As the most elaborated idea linguistically, we have captured the

```

abstract Foods = {
  flags startcat = Comment ;
  cat
    Comment ; Item ; Kind ; Quality ;
  fun
    Pred : Item -> Quality -> Comment ;
    This, That, These, Those : Kind -> Item ;
    Mod : Quality -> Kind -> Kind ;
    Wine, Cheese, Fish, Pizza : Kind ;
    Very : Quality -> Quality ;
    Fresh, Warm, Italian,
      Expensive, Delicious, Boring : Quality ;
}

```

FIGURE 20 Abstract syntax Foods, introducing plural noun phrases.

common pattern in the formation of noun phrases with determiners by the operation **det**. Without this operation, we would have four **lin** rules similar to the rule

```
lin This kind = {s = "this" ++ kind.s ! Sg ; n = Sg}
```

This rule is just a one-liner; however, in Italian, we will save several lines with the same idea, as also gender agreement is involved. This rule can also be seen as a step towards treating determiners as a category of its own, as will be done in Section 5.3.

3.8 Testing inflection and operations in GF

When developing a grammar, it is useful to test inflection separately. This can be done in the **linearize** command by using the flag **-table**, which prints all branches of tables (and also all record fields with their labels):

```

Foods> linearize -table Wine
s Sg : wine
s Pl : wines

```

It is also possible to test the evaluation of arbitrary terms, by the command **compute_concrete = cc**. Since evaluation uses the definitions of **oper** operations, which are normally thrown away in the course of compilation, the **import** command must use the flag **-retain** in order to retain the operations. Thus we do:

```
> import -retain FoodsEng.gf
```



```

concrete FoodsEng of Foods = {
  lincat
    Comment, Quality = {s : Str} ;
    Kind = {s : Number => Str} ;
    Item = {s : Str ; n : Number} ;
  lin
    Pred item quality =
      {s = item.s ++ copula ! item.n ++ quality.s} ;
    This = det Sg "this" ;
    That = det Sg "that" ;
    These = det Pl "these" ;
    Those = det Pl "those" ;
    Mod quality kind =
      {s = \n => quality.s ++ kind.s ! n} ;
    Wine = regNoun "wine" ;
    Cheese = regNoun "cheese" ;
    Fish = noun "fish" "fish" ;
    Pizza = regNoun "pizza" ;
    Very a = {s = "very" ++ a.s} ;
    Fresh = adj "fresh" ;
    Warm = adj "warm" ;
    Italian = adj "Italian" ;
    Expensive = adj "expensive" ;
    Delicious = adj "delicious" ;
    Boring = adj "boring" ;
  param
    Number = Sg | Pl ;
  oper
    det : Number -> Str ->
      {s : Number => Str} -> {s : Str ; n : Number} =
        \n,det,noun -> {s = det ++ noun.s ! n ; n = n} ;
    noun : Str -> Str -> {s : Number => Str} =
      \man,men -> {s = table {Sg => man ; Pl => men}} ;
    regNoun : Str -> {s : Number => Str} =
      \car -> noun car (car + "s") ;
    adj : Str -> {s : Str} =
      \cold -> {s = cold} ;
    copula : Number => Str =
      table {Sg => "is" ; Pl => "are"} ;
}

```

FIGURE 21 English concrete syntax for Foods.

```
> compute_concrete (regNoun "wine").s ! Pl
"wines"
```

3.9 Partial application

The rule appearing in Figure 21,

```
lin This = det Sg "this"
```

could be written equivalently

```
lin This kind = det Sg "this" kind
```

However, a variable appearing as the last term on both the left and the right hand side of a rule is not necessary. This is not an *ad hoc* convention, but an instance of **partial application**. A partial application of a function

$$f : A \rightarrow B \rightarrow C$$

is an application of the function only to the first argument; the resulting term is thus itself a function on the remaining argument type,

$$f a : B \rightarrow C$$

Now, a linearization rule

$$\text{lin } f x_1 \dots x_n = t$$

is really a shorthand for defining a function by lambda abstraction,

$$\text{lin } f = \lambda x_1, \dots, x_n \rightarrow t$$

What counts is that the right hand side is a function from the linearization types of the argument types of f to the linearization type of its value type. This function need not be given as a full lambda abstract: any term that denotes a function of correct type is usable.

Not using partial application when possible is usually considered clumsy in functional programming. It is a part of the art of programming to design function types in such a way that partial application can be used as much as possible. For instance, if the type of `det` in Figure 21 had its argument types in a different order, e.g.

```
{s : Number => Str} -> Number -> Str ->
  {s : Str ; n : Number}
```

we could not use partial application in the grammar, which would result in longer and heavier code.

3.10 Discontinuous constituents

The general form of a linearization type (**lincat**) as presented so far is a record type of the form

$$\{s : P \rightarrow \mathbf{Str}; q : Q\}$$

where P represents the variable features and Q the inherent features; this form would actually work with any number of variable and inherent features, by the use of algebraic datatype constructors or records, as shown in Section 4.11. The crucial restriction is that there is just *one* string-valued field in the record, the field **s**. However, there is really no such restriction in GF: a linearization can be *any* combination of strings, parameters, tables, and records.

The use of records with many strings can be used for modelling **discontinuous constituents**. A discontinuous constituent is precisely what is sounds: a constituent (i.e. part of an expression corresponding to a subtree) that consists of several strings, which can be separated by other strings. A prime example is provided by **verb phrases** in many languages, including English. Verb phrases (**VP**) appear in the most well-known rule of linguistic syntax,

$$S ::= NP \ VP$$

This rule says that a sentence (**S**) can be built from a noun phrase (**NP**) and a verb phrase (**VP**). Sentences like *this wine is expensive* have exactly this form: *this wine* is the noun phrase and *is expensive* is the verb phrase. Verb phrases also cover *walks* in *John walks*, *loves Mary* in *John loves Mary*, etc. The NP-VP format covers a vast class of sentences. As we will see in Section 5.15, it is the analysis used in the linguistic resource grammar, as opposed to the semantic grammar we have been building so far.

But what is the syntax of *questions*, such as *is this wine delicious* or *does John love Mary*? If we want to discern a verb phrase in them as well, then it must be discontinuous: *is - delicious* and *does - love Mary*. Since in the resource grammar we want to have a common set of rules that construct verb phrases for both questions and sentences, we have to use records with two strings: one for the verb (**verb**) and one for the complement (**comp**).

Figure 22 shows a minimal grammar that forms declarative sentences and questions from discontinuous verb phrases. It uses the category **C1** of *clauses* to represent sentences in abstraction from whether they are used declaratively or as questions. Thus its linearization is a table over these possible sentence forms, encoded in the parameter type **SForm**.

Figure 22 of course just gives the bare bones of discontinuous verb

phrases. It doesn't even cover the formation of verb phrases with verbs, which in English poses the special problem that questions use an auxiliary *do* whereas declarative sentences only use it if they are negative (*John doesn't walk*). Working out the details of this is left as an exercise.

Alongside with parameters, discontinuous constituents play a crucial role in multilingual grammars, since they enable the maintenance of abstract grammatical concepts such as verb phrases over languages. They are also language-theoretically interesting, since they enable the definition of some famous non-context-free languages, as shown in the exercises below.

Exercise 3-4.* Add a rule for building verb phrases from verbs to the grammar in Figure 22. In the English concrete syntax, you have to find a way to treat the auxiliary *do* correctly.

Exercise 3-5.* Write a grammar that generates the (non-context-free) language $a^n b^n c^n$, i.e. a language whose strings are the empty string, *abc*, *aabbcc*, etc, where there are always as many *a*'s as *b*'s and *c*'s.

Exercise 3-6.* Write a grammar that generates the (non-context-free) language $a^m b^n c^m d^n$, i.e. where the number of *a*'s and *c*'s is the same and so is the number of *b*'s and *d*'s. This language is well-known as a model of Swiss German, originally presented by Shieber in 1985 in his argument that Swiss German is not context-free.

Exercise 3-7.+ Now we have defined a part of GF that is *complete* in the sense that pretty much any GF grammar can be written in it. So you can try and write a concrete syntax of **Foods** for any language you please, and make it correct. This is a useful exercise to repeat after reading Chapter 4 when you have learnt more of GF to make your programming more efficient. And then again after Chapter 5, when you can do it with the minimum of work conceivable!

3.11 Non-concatenative morphology

Once we get rid of the idea that linguistic objects in concrete syntax have to be continuous strings, we have no problems with the **non-concatenative morphology**, as found in Semitic languages such as Arabic, Hebrew, and Maltese.

Thus, for instance, Arabic word inflection is much more complex than just concatenating suffixes or prefixes to word stems. A word like *kataba* ("to write") has forms like *kaatibu*, *yaktubu*, *kattaba*, *iktataba*. The traditional analysis of Arabic words is the following:

```

abstract Discont = {
  cat
    S ; Cl ; NP ; VP ; AP ;
  fun
    DeclCl   : Cl -> S ;
    QuestCl  : Cl -> S ;
    PredVP   : NP -> VP -> Cl ;
    CompAP   : AP -> VP ;
    John    : NP ;
    Old     : AP ;
}

concrete DiscontEng of Discont = {
  param
    SForm = SDecl | SQuest ;
  lincat
    S, NP, AP = Str ;
    Cl = SForm => Str ;
    VP = {verb,comp : Str} ;
  lin
    DeclCl cl = cl ! SDecl ;
    QuestCl cl = cl ! SQuest ;
    PredVP np vp = table {
      SDecl => np ++ vp.verb ++ vp.comp ;
      SQuest => vp.verb ++ np ++ vp.comp
    } ;
    CompAP ap = {
      verb = "is" ;
      comp = ap
    } ;
    John = "John" ;
    Old = "old" ;
}

```

FIGURE 22 A minimal grammar with discontinuous verb phrases.

- A word is formed by applying a **pattern** to a **root**.
- A root is a triple of consonants, which are called the **radicals**.
- A pattern is a function from roots to strings.

Thus, for instance, the triple *ktb* is a root, from which the word *kataba* is obtained by the pattern *FaCaLa*, and the word *yaktubu* by the pattern *yaFCuLu*. The traditional term for patterns is “*fa‘al* patterns”, where *fa‘al* is a verb for “do”. Its three radicals are *f*, the guttural stop ‘, and *l*. We will just use capital letters for these radicals, and the more convenient letter *C* for the guttural stop. One could also read *F* as the first, *C* as the central, and *L* as the last radical.

We can now implement roots as records and patterns as functions:

```
oper
Root      : Type = {F,C,L : Str} ;
Pattern   : Type = Root -> Str ;
```

These operations define **type synonyms**, but they are just ordinary **opers**, with **Type** given as value type.

One convenient pattern is a record that fills in material in each of the four slots before, after, and between the radicals:

```
Filling : Type = {F,FC,CL,L : Str} ;
```

Applying this pattern is by interlacing the records,

```
fill : Filling -> Pattern = \p,r ->
  p.F + r.F + p.FC + r.C + p.CL + r.L + p.L ;
```

Notice that this expression is well-typed because the type **Pattern** is itself a function type and `\p,r` is a shorthand for two lambdas.

There’s basically only one kind of pattern in Arabic that is *not* a pure filling pattern: the duplication of the middle consonant. But it can be defined as a special kind of filling:

```
dfill : Filling -> Pattern = \p,r ->
  p.F + r.F + p.FC + r.C + r.C + p.CL + r.L + p.L ;
```

In the lexicon, we can now build word forms in the following way:

```
yaktubu = fill
  {F = "ya" ; FC = "" ; CL = "u" ; L = "u"}
  {F = "k" ; C = "t" ; L = "b"}
kuttiba = dfill
  {F = "" ; FC = "u" ; CL = "i" ; L = "a"}
  {F = "k" ; C = "t" ; L = "b"}
```

This is quite tedious: what we would really like to write, in accordance with normal dictionary conventions, is just two strings for the pattern

and the root:

```
yaktubu = word "yaFCuLu" "ktb"  
kuttiba = word "FuCCiLa" "ktb"
```

We will return to this problem in Section 4.5 and show how it is solved by the powerful technique of pattern matching over strings.

Modular and scalable grammar writing

This chapter introduces concepts that help in scaling up from small and simple grammars to larger and more structured ones. It covers the following topics:

- reusable resource modules
- data abstraction
- smart paradigms
- pattern matching over strings
- operation overloading
- module extension and inheritance
- algebraic datatypes
- record extension, subtyping, and tuples
- prefix-dependent choices
- compile-time vs. run-time string operations

4.1 Reusable resource modules

Parameter types and operations belong to concrete syntax. Unlike `lincat` and `lin` definitions, they are completely independent of any abstract syntax. They could be used in many concrete syntaxes, for different abstract syntaxes and sometimes even for different languages; for instance, the two-valued parameter type `Number` would work in both English and Italian, as well as many other languages. To make this sharing of code possible in grammars, GF provides a module type entitled `resource`, which can contain `oper` and `param` judgements. Figure 23 shows a `resource` module with parameters and operations usable in the Italian `Foods` grammar. It is very similar to the set of

param and **oper** definitions contained in the English Foods grammar—just slightly more complex because of the presence of gender in Italian.

To use the definitions contained in a **resource** model, a **concrete** module must **open** it. The module **FoodsIta** Figure 24 thus opens **ResIta** and uses the parameters and operations defined in it. Also **resource** modules may themselves open resources; thus **ResIta** opens the module **Prelude**, which is a standard library module distributed with the GF system. **Prelude** contains basic operations on strings, records, and tables; **ResIta** uses the **Prelude** operation **init**, which has the type **Str -> Str**. It returns the initial segment of its argument string, that is, the string minus its last character, so that e.g. **init "nero"** is **"ner"**. Documentation of **Prelude** can be found in Section D.4.1.

The module **ResIta** displays two new forms of expression. The first one is **local definitions**, also known as **let expressions**, in the definition of **regAdj**. A **let** expression has the form

$$\text{let } c : t = d \text{ in } e$$

where the constant c has the type t and the value d in the expression e . The constant c is local in the sense that it is not available outside the e part of the **let** expression. As in all functional programming, local definitions are in GF used for eliminating multiple evaluations of the same expression d , and also for making the code more structured and clear.

The second new form is **function types with variables** (of which wildcards $_$ are a special case), in the type of **adjective**. The type

```
(nero,nera,neri,nere : Str) -> Adjective
```

is actually the same as

```
Str -> Str -> Str -> Str -> Adjective
```

but the use of variables makes it possible to share the argument type. They can also be given mnemonic names, like here, to guide the users of the function about the proper order of arguments. However, since the variable names are not used later and hence make no semantic difference, also wildcards could be used:

```
(_,_,_,_ : Str) -> Adjective
```

Finally, notice that **ResIta** defines **type synonyms** for the linearization types corresponding to noun phrases, nouns, and adjectives. The use of type synonyms is useful for structuring code and making it clearer. It is also a way to implement **abstract datatypes**, which are

```

resource ResIta = open Prelude in {
  param
    Number = Sg | Pl ;
    Gender = Masc | Fem ;
  oper
    NounPhrase : Type =
      {s : Str ; g : Gender ; n : Number} ;
    Noun : Type = {s : Number => Str ; g : Gender} ;
    Adjective : Type = {s : Gender => Number => Str} ;

    det : Number -> Str -> Str -> Noun -> NounPhrase =
      \n,m,f,cn -> {
        s = table {Masc => m ; Fem => f} ! cn.g ++
          cn.s ! n ;
        g = cn.g ;
        n = n
      } ;
    noun : Str -> Str -> Gender -> Noun =
      \vino,vini,g -> {
        s = table {Sg => vino ; Pl => vini} ;
        g = g
      } ;
    adjective :
      (nero,nera,neri,nere : Str) -> Adjective =
        \nero,nera,neri,nere -> {
          s = table {
            Masc => table {Sg => nero ; Pl => neri} ;
            Fem => table {Sg => nera ; Pl => nere}
          }
        } ;
    regAdj : Str -> Adjective = \nero ->
      let ner : Str = init nero
      in
        adjective nero (ner+"a") (ner+"i") (ner+"e") ;
    copula : Number => Str =
      table {Sg => "è" ; Pl => "sono"} ;
}

```

FIGURE 23 A resource module for Italian grammars.

```

concrete FoodsIta of Foods = open ResIta in {
  lincat
    Comment = {s : Str} ;
    Quality = Adjective ;
    Kind = Noun ;
    Item = NounPhrase ;
  lin
    Pred item quality =
      {s = item.s ++ copula ! item.n ++
        quality.s ! item.g ! item.n} ;
    This = det Sg "questo" "questa" ;
    That = det Sg "quel" "quella" ;
    These = det Pl "questi" "queste" ;
    Those = det Pl "quei" "quelle" ;
    Mod quality kind = {
      s = \\n => kind.s ! n ++ quality.s ! kind.g ! n ;
      g = kind.g
    } ;
    Wine = noun "vino" "vini" Masc ;
    Cheese = noun "formaggio" "formaggi" Masc ;
    Fish = noun "pesce" "pesci" Masc ;
    Pizza = noun "pizza" "pizze" Fem ;
    Very qual =
      {s = \\g,n => "molto" ++ qual.s ! g ! n} ;
    Fresh =
      adjective "fresco" "fresca" "freschi" "fresche" ;
    Warm = regAdj "caldo" ;
    Italian = regAdj "italiano" ;
    Expensive = regAdj "caro" ;
    Delicious = regAdj "delizioso" ;
    Boring = regAdj "noioso" ;
  }
}

```

FIGURE 24 Italian concrete syntax for Foods.

a key to scalable code and to which we will turn in the next section.

Exercise 4-1. Factor out a separate **resource** module from **FoodsEng** as defined in Figure 21. Then reimplement **FoodsEng** by using this resource.

Exercise 4-2. Write an abstract syntax similar to **Foods** but with a different vocabulary, possibly describing a different domain. Use the resource module created in the previous exercise to define the concrete syntax. Do the same with Italian, if you know enough Italian.

4.2 Data abstraction

Let us look more closely at the problem of implementing inflection paradigms. We want to make them easy to use for all words, not just for regular ones. The goal is to provide a morphology module that is maximally easy to use when words are added to the lexicon. We should think of a division of labour where a linguistically trained grammarian writes a set of inflection paradigms and hands them over to the lexicon writer who knows much less about the rules of inflection.

We work through English nouns as our example. Although simple, this example requires techniques that scale up to much more complex morphological systems.

We begin with performing **data abstraction** by first defining a type synonym,

```
oper Noun : Type = {s : Number => Str} ;
```

Then we define a **worst-case function** that covers all possible nouns, even the most irregular ones:

```
oper mkNoun : Str -> Str -> Noun = \x,y -> {
  s = table {
    Sg => x ;
    Pl => y
  }
} ;
```

Using **mkNoun**, we can thus define irregular nouns,

```
lin Mouse = mkNoun "mouse" "mice" ;
```

We also use it to define paradigms, including the regular one:

```
oper regNoun : Str -> Noun =
  \word -> mkNoun word (word + "s") ;
```

The user of the module is only supposed to see the types of these operations:

```

oper
  Noun : Type ;
  mkNoun : Str -> Str -> Noun ;
  regNoun : Str -> Noun ;

```

In programming terms, **Noun** is treated as an **abstract datatype**: its definition is not exposed, but only some indirect way of constructing its objects. The set of operation type signatures (hiding the definitions) works as an **interface** for the programmers who use the operations.

The engineering advantage of abstract datatypes is that the author of the resource module may change the definitions of operations without the user's needing to change her use of them. For instance, the resource module author may add **case** (nominative and genitive) to English nouns:

```

param Case = Nom | Gen ;
oper Noun : Type = {s : Number => Case => Str} ;

```

The worst-case function must now be redefined to match the new definition of **Noun**:

```

oper mkNoun : Str -> Str -> Noun = \x,y -> {
  s = table {
    Sg => table {
      Nom => x ;
      Gen => x + "'s"
    } ;
    Pl => table {
      Nom => y ;
      Gen => case y of {
        _ + "s" => y + "'" ;
        _      => y + "'s"
      }
    }
  }
} ;

```

The generalization is straightforward, except for the genitive plural. There we have to perform string matching (by a **case** expression) to decide whether the ending is *'s* (for irregular nouns such as *men-men's*) or just *'* (for regular nouns such as *boys-boys'*). The details of string matching will be explained in next section.

Now, since the interface of **Noun** and **mkNoun** hasn't changed, the old definition

```

oper regNoun : Str -> Noun = \x -> mkNoun x (x + "s") ;

```

is still valid. It will just produce larger tables now.

4.3 Case expressions and string matching

Inside the last definition of `mkNoun` in the previous section, we used a **case expression**, which is a common construct in functional programming:

```
case y of {
  _ + "s" => y + "'s" ;
  _       => y + "'s"
}
```

This expression performs **pattern matching over strings**: the patterns in the table are matched against the string bound to the variable `y`, which in this example is the nominative plural form of a noun.

Just like **table expressions**, **case expressions** have a list of branches with patterns and values. In the above example, the pattern `_ + "s"` in the first branch matches any word that ends with an "s". The pattern `_` in the second branch matches all other words. It is an example of a **concatenative pattern**, which is analogous to the `+` operator used for gluing tokens.

Here is a list of the most important string matching patterns:

- the **disjunctive pattern** $P \mid Q$, which matches everything that P or Q matches
- the **concatenation pattern** $P + Q$, which matches any string of form st where P matches s and Q matches t
- the **variable pattern** x , which matches anything and binds the variable x to this
- the **wildcard pattern** `_`, which matches anything
- the **alias pattern** $x@P$, which matches anything that P matches and binds the variable x to this
- the **string pattern** `"foo"`, which matches just the string `"foo"`
- the **one-character pattern** `?`, which matches any string whose length is exactly one (Unicode) character

The disjunctive and concatenation patterns are also called **regular expression patterns**, since they correspond to regular expressions. Notice that all patterns, except the concatenation, string, and one-character patterns, work for all parameter types as well, and not only for strings.

Case expressions over parameter types are in fact not primitives in GF, but syntactic sugar for table selections:

$$\text{case } e \text{ of } \{ \dots \} \equiv \text{table } \{ \dots \} ! e$$

A case expression is often easier to read and write than a table selection, especially when the list of branches is long. They can be used not only for string matching but also for matching with any parameter types. A typical example is shown in Figure 33 in Section 4.11.

4.4 Smart paradigms

Between the completely regular *dog-dogs* and the completely irregular *mouse-mice*, there are many predictable variations:

- nouns ending with *y*: *fly-flies*, except if a vowel precedes the *y*: *boy-boys*
- nouns ending with *s*, *ch*, and a number of other endings: *bus-buses*, *leech-leeches*

One way to deal with them would be to provide alternative paradigms:

```
noun_y : Str -> Noun = \fly ->
    mkNoun fly (init fly + "ies") ;
noun_s : Str -> Noun = \bus ->
    mkNoun bus (bus + "es") ;
```

But this solution has some drawbacks:

- it can be difficult to select the correct paradigm
- it can be difficult to remember the names of all different paradigms

To help the lexicon builder's task, the morphology programmer can put more intelligence in the regular noun paradigm. She can capture the predictable variations by using pattern matching over strings. The easiest way to express this in GF is by the use of regular expression patterns, as shown in Figure 25. In this definition, we have used a local definition just in order to structure the code, even though the constant is used only once. The patterns in Figure 25 are ordered in such a way that, for instance, the suffix "*oo*" prevents *bamboo* from matching the suffix "*o*" and producing *bambooes*. Similarly, "*boy*" doesn't reach the pattern *x + "y"*, which would produce "*boies*".

Exercise 4-3. The same rules that form plural nouns in English also apply in the formation of third-person singular verbs. Write a regular verb paradigm that uses this idea, but first rewrite **regNoun** so that the analysis needed to build *s*-forms is factored out as a separate **oper**, which is shared with **regVerb**.

Exercise 4-4. Extend the verb paradigms to cover all regular verb forms in English, with special care taken of variations with the suffix *ed* (e.g. *try-tried*, *use-used*).

```

regNoun : Str -> Noun = \w ->
  let
    ws : Str = case w of {
      _ + ("a" | "e" | "i" | "o") + "o" => w + "s" ;
      _ + ("s" | "x" | "sh" | "o")      => w + "es" ;
      _ + ("a" | "e" | "o" | "u") + "y" => w + "s" ;
      x + "y"                          => x + "ies" ;
      _                                => w + "s"
    }
  in
    mkNoun w ws

```

FIGURE 25 A smart paradigm for English nouns using regular expression patterns

Exercise 4-5. Implement the German **Umlaut** operation on word stems. The operation has the type `Str -> Str`. It changes the vowel of the stressed stem syllable as follows: *a* to *ä*, *au* to *äu*, *o* to *ö*, and *u* to *ü*. You can assume that the operation only takes syllables as arguments. Test the operation to verify that it correctly changes *Arzt* to *Ärzt*, *Baum* to *Bäum*, *Topf* to *Töpf*, and *Kuh* to *Küh*.

4.5 Arabic morphology revisited

In Section 3.11, we were left with Arabic lexicon construction functions that were awkward to use, because they used the root and pattern data structures explicitly. What we really want to have is a word-construction operation

```
word : (patt, root : Str) -> Str
```

which allows us to write

```
yaktubu = word "yaFCuLu" "ktb"
```

The code in Figure 26 shows how to define this operation. It uses pattern matching on strings to extract the desired record fields. Thus `getRoot` matches any string of length 3 and extracts each of its characters in order. A string that has another length produces an error, which guards against ill-formed patterns; the non-3-radical roots of Arabic must be treated separately. The `getPattern` operation looks for the letters "F", "C", "L", which are not used as real letters in Arabic words. If "C" is duplicated, the duplicating pattern is returned.


```

Root      : Type = {F,C,L : Str} ;
Pattern   : Type = Root -> Str ;

Filling   : Type = {F,FC,CL,L : Str} ;

fill      : Filling -> Root -> Str = \p,r ->
  p.F + r.F + p.FC + r.C + p.CL + r.L + p.L ;

dfill     : Filling -> Root -> Str = \p,r ->
  p.F + r.F + p.FC + r.C + r.C + p.CL + r.L + p.L ;

getRoot    : Str -> Root = \s -> case s of {
  F@? + C@? + L@? => {F = F ; C = C ; L = L} ;
  _ => Predef.error ("cannot get root from" ++ s)
} ;

getPattern : Str -> Pattern = \s -> case s of {
  F + "F" + FC + "CC" + CL + "L" + L =>
    dfill {F = F ; FC = FC ; CL = CL ; L = L} ;
  F + "F" + FC + "C" + CL + "L" + L =>
    fill {F = F ; FC = FC ; CL = CL ; L = L} ;
  _ => Predef.error ("cannot get pattern from" ++ s)
} ;

word : (patt, root : Str) -> Str = \p,r ->
  getPattern p (getRoot r) ;

```

FIGURE 26 Arabic roots and patterns and their encoding as strings.

4.6 Separating operation types and definitions

In **resource** modules intended as libraries, it is useful to separate operation definitions from their type signatures. The user is only interested in the types, whereas the definitions are kept for the implementor and the maintainer. This is possible by using separate **oper** fragments for the two parts:

```
oper regNoun : Str -> Noun ;
oper regNoun s = mkNoun s (s + "s") ;
```

The type checker combines the two parts into one **oper** judgement and checks that the definition matches the type. Notice that, in this syntax, it is moreover possible to bind the argument variables on the left hand side instead of using lambda abstraction, analogously to the way we have written **lin** judgements.

In a library module, the type signatures are typically placed in the beginning and the definitions in the end. A more radical separation can be achieved by using separate **interface** and **instance** module types: the type signatures are placed in the interface and the definitions in the instance. These module types will be introduced in Section 5.9.

4.7 Overloading of operations

Large libraries, such as the GF Resource Grammar Library, may define thousands of names. This can be unpractical for both the library author and the user: the author has to invent longer and longer names which are not always intuitive, and the user has to learn all these names, or at least find them easily. A solution to this problem, inspired by the programming language C++, is **overloading**: one and the same name can be used for several functions. When such a name is used, the compiler performs **overload resolution** to find out which of the possible functions is meant. Overload resolution is based on the types of the functions: all functions that have the same name must have different types. (In contrast to C++, not only the argument types but also the value type is used in overload resolution.)

In C++, functions with the same name can be scattered everywhere in the program. In GF, they must be grouped together in **overload** groups. Here is an example of an overload group, giving the different ways to define nouns in English:

```
oper mkN = overload {
  mkN : (dog : Str) -> Noun = regNoun ;
  mkN : (mouse,mice : Str) -> Noun = mkNoun ;
}
```

Intuitively, this function comes close to the way in which regular and irregular words are given in most dictionaries. If the word is regular, just one form is needed. If it is irregular, more forms are given.

The `mkN` example gives both the types and definitions of the overloaded operation. If the type is to be given separately, the following format is used:

```
oper mkN : overload {
  mkN : (dog : Str) -> Noun ;           -- regular nouns
  mkN : (mouse,mice : Str) -> Noun ;    -- irregular nouns
}
```

Notice that the types must be repeated in the overload group giving definitions, because it is by means of types that the branches are identified. The order of the branches can be different in the typing group and the corresponding defining group.

Exercise 4-6. Design, implement, and test a system of English verb paradigms presented as an overload group.

4.8 Module extension and inheritance

A grammar of natural language can easily have thousands of rules, in particular in the lexicon. Such grammars are cumbersome to keep in single files. Moreover, splitting a grammar into separate modules makes it possible to recombine such modules and create **module hierarchies**. GF supports this kind of modularity by providing a mechanism of **extension** for modules.

A module can **extend** one or more other modules and thereby **inherit** their judgements. It can of course add its own judgements to the inherited ones. Examples of module extension are shown in Figure 27. The comma-separated list of inherited modules is separated from the module body by the operator ******. If the module body is empty, it can be omitted, as in the last module **Shopping**. If the module has **opens**, they appear after the extensions, as shown in Figure 28. Thus the general syntax of module headers is

*moduletype name = extends ** opens in body*

As exemplified by Figure 27, a typical usage of module extension is the separation of a grammar to a part with syntactic combination rules (a **base grammar**) and a more specific vocabulary part (a **domain lexicon**). Figure 27 shows a base grammar **Comments**, which just contains combination rules. Its extension **Foods** contains food-related vocabulary. Another extension, **Clothes**, contains clothing-related vocabulary. Each of these extensions can be used separately. But they

```

abstract Comments = {
  flags startcat = Comment ;
  cat
    Comment ; Item ; Kind ; Quality ;
  fun
    Pred : Item -> Quality -> Comment ;
    This, That, These, Those : Kind -> Item ;
    Mod : Quality -> Kind -> Kind ;
    Very : Quality -> Quality ;
}

abstract Foods = Comments ** {
  fun
    Wine, Cheese, Fish, Pizza : Kind ;
    Fresh, Warm, Italian,
      Expensive, Delicious, Boring : Quality ;
}

abstract Clothes = Comments ** {
  fun
    Shirt, Jacket : Kind ;
    Comfortable, Elegant : Quality ;
}

abstract Shopping = Foods, Clothes ;

```

FIGURE 27 A module hierarchy with a base grammar, two extensions, and a further extension that merges the two.

```

concrete ClothesIta of Clothes = CommentsIta **
  open SyntaxIta, ParadigmsIta in {
  lin
    Shirt = mkCN (mkN "camicia") ;
    Jacket = mkCN (mkN "giacca") ;
    Comfortable = mkAP (mkA "comodo") ;
    Elegant = mkAP (mkA "elegante") ;
  }

```

FIGURE 28 An extension of a concrete syntax with resources opened.

can also be combined to a larger module, **Shopping**, which comprises comments on both food and clothes.

Inheritance is available for all module types. The inherited modules must be of the same type as the main module. Figure 28 shows an Italian concrete syntax of **Clothes**. Parallel to its abstract syntax **Clothes**, which extends **Comments**, **ClothesIta** extends **CommentsIta**, which is a concrete syntax of **Comments**. The resource modules opened come from the GF Resource Grammar Library, which is introduced in Chapter 5.

Inheritance of several modules is known as **multiple inheritance**. It raises the question what happens if one and the same constant is inherited from more than one module. In GF, the answer is simple: every constant must be unique. Thus the same constant may be declared or defined at most once in the union of the judgements in the same module and all inherited modules. Inheriting the same constant from two modules is a compile-time error, and so is redefining a constant that has been inherited.

The most subtle question with multiple inheritance is the **diamond property**: what happens when the same constant is inherited twice from an underlying module, via two extensions of it? This is actually the case with the **Shopping** module of Figure 27, whose dependencies are visualized in Figure 29. All the categories and functions of **Comments** are inherited twice by **Shopping**! This is, however, not a problem, since the compiler can recognize that these constants come in the end from the same source. As it is forbidden to redefine constants, we can be sure that a constant inherited via **Clothes** is the same as the constant with the same name inherited via **Foods**.

Inheritance provides a way to reuse old modules by putting them together. The granularity of always inheriting a whole module may not be sufficient, however. To inherit just selected constants, **restricted inheritance** can be used. It uses lists of constants, which are either

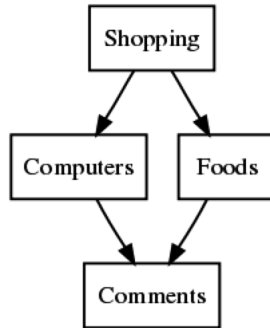


FIGURE 29 Inheritance in a diamond position.

```

abstract SmallShopping =
  Foods - [Wine],
  Clothes [Kind,Quality,Shirt,Elegant] ;

```

FIGURE 30 Restricted inheritance with excluded and included constants.

excluded (list with a minus sign `-`) or solely included (plain list). An example is shown in Figure 30, where `Wine` is excluded from `Foods`, and only `Shirt` and `Elegant` are included from `Clothes`. Notice that the inclusion list in `Clothes` also contains the category names `Kind` and `Quality`. The reason is that the judgements introducing `Shirt` and `Elegant` depend on these categories, and could not be compiled without them being available.

A constant that is not inherited can of course be freely defined in the module body. This has an obvious consequence in the diamond position: in Figure 31, the inheritance of `Very` in `Shopping` does not have a unique source, which makes it illegal.

Exercise 4-7. As yet another variant of the reversing grammar of the exercise in Section 2.9, refactor the grammar so that the linearizations of tokens are inherited from the same concrete syntax in both the forward and the backward version.

```

abstract Comments = ...
abstract Foods = Comments ** {...}
abstract Clothes = Comments - [Very] ** {fun Very ...}
abstract Shopping = Foods, Clothes ; -- ERROR!

```

FIGURE 31 Illegal multiple inheritance of the constant `Very`.

```

resource Auxiliary = {oper aux ...}
resource Library = open Auxiliary in {oper foo = aux ...}
concrete Application of A = open Library in {
  lin f = foo ... ; -- CORRECT
  lin g = aux ...   -- INCORRECT
}

```

FIGURE 32 Information hiding: a public library opening a private auxiliary.

4.9 Inheritance and opening

Both inheritance and opening are ways of **importing**, or **including**, other modules in a module. Some programming languages, for instance C, don't make a distinction between these two mechanisms. In GF, there are two related reasons to make the distinction.

First, GF has different types of modules, and extension is meaningful only for modules of the same type; opening can be between different types, so that in the most common case it is a **concrete** that opens a **resource**.

Secondly, inheritance is transitive, whereas opening is not. Transitivity means that if a module *N* inherits *M* and *M* inherits *L*, then also *N* inherits *L*. Opening does not have this property, which is crucial for **information hiding**. Thus for instance a library module may open some auxiliary modules, but these modules are kept **private** in the sense that they are not available for the user of the library. Figure 32 shows a constellation where a public **Library** opens a private **Auxiliary**, and an **Application** using the **Library** has no access to the operations defined in **Auxiliary**.

Opening, but not inheritance, supports **qualification**. This means that the constants imported from an opened module are marked with the name of the module: **Library.foo** instead of just **foo**. Qualification makes it possible to avoid conflicts in cases where one and the same name is imported from different opened modules.

4.10 Dependency graphs

To help grammar writers in managing module structures, GF produces a visualization tool that shows **dependency graphs**. Such a graph can be created by using the command `dg = dependency_graph` and the graphviz package with the command `dot`. This is how we did to produce the picture in Figure 29:

```

> i -retain Shopping.gf
> dependency_graph

```

```
-- wrote graph in file _gfdepgraph.dot
> ! dot -Tpng _gfdepgraph.dot >diamond.png
```

The `-retain` option is needed to keep track of all source modules, just like in the `compute_concrete` command in Section 3.8.

More involved dependency graphs are shown in Figure 73 in Section 9.9 and in Figure 75 in Section 10.1. In these graphs, modules and dependencies of different kinds are marked with different shapes and arrow styles.

4.11 Algebraic datatypes for parameters

Parameter type definitions are similar to the definitions of **algebraic datatypes** in functional programming languages like Haskell and ML. The examples shown so far all belong to the restricted special case of **enumerated types**. But also more general types are possible in GF. As a generalization of parameter constants, these types involve **constructors** that take arguments from other parameter types. With such constructors, one can express hierarchies of parameters.

In German, the form of a determiner depends on gender, case, and number. Gender and case are enumerated types defined as follows:

```
param Gender = Masc | Fem | Neutr
param Case = Nom | Acc | Dat | Gen
```

However, the plural forms are the same for all genders. If we defined determiner inflection as the type

```
Number => Gender => Case => Str
```

we would generate tables with superfluous forms. To avoid this, we define a special parameter type of determiner forms,

```
param DetForm = DSg Gender Case | DP1 Case
```

Its first constructor `DSg` is for singular forms, and it takes a gender and a case as arguments. Its second constructor `DP1` takes only a case. The type `DetForm` has thus $3 \times 4 + 4 = 16$ distinct values, instead of $2 \times 3 \times 4 = 24$. Figure 33 shows an example of a determiner defined as a table of this type, using pattern matching. Here, as often in practice, a word can have fewer forms than the theoretical maximum, because of **syncretism** (the merger of distinct forms). This is expressed by using disjunctive patterns and wildcards in Figure 33, which shares branches maximally—perhaps at the expense of clarity.

What syncretism means formally is clear: it means that different parameters produce the same value. What is treated as syncretism and what is a part of the type system is up to the grammarian to


```

oper artDef : DetForm => Str = table {
  DSg Masc Acc | DPl Dat => "den" ;
  DSg (Masc | Neutr) Dat => "dem" ;
  DSg (Masc | Neutr) Gen => "des" ;
  DSg Neutr _ => "das" ;
  DSg Fem (Nom | Acc) | DPl (Nom | Acc) => "die" ;
  _ => "der"
}

```

form	Sg Masc	Sg Fem	Sg Neutr	Pl
Nom	<i>der</i>	<i>die</i>	<i>das</i>	<i>die</i>
Acc	<i>den</i>	<i>die</i>	<i>das</i>	<i>die</i>
Dat	<i>dem</i>	<i>der</i>	<i>dem</i>	<i>den</i>
Gen	<i>des</i>	<i>der</i>	<i>des</i>	<i>der</i>

FIGURE 33 The German definite article as a case expression and as an inflection table.

decide. For instance, the merger of genders in the plural of German determiners could be treated as syncretism as well—but this solution would be less elegant than the algebraic parameter type, because the merger of genders in the plural has *no* exceptions. On the other hand, it is also true of German that the nominative and accusative forms differ only in the masculine singular. This *could* be expressed in the type system, but we have treated it as syncretism in the definition of `DetForm`.

Algebraic datatypes with just one constructor that takes many arguments are an alternative to using tables with many arguments. They provide more data abstraction: if we define, as in Section 4.2,

```
lincat N = {s : Number => Case => Str}
```

we “leak” the information that nouns have two variable features. If we instead define

```

param NForm = NF Number Case ;
lincat N = {s : NForm => Str}

```

we can write more robust code. For instance, if adjectival modification just passes the variable noun features to the noun, we can simply write

```
lin Mod adj noun = {s = \\f => adj.s ++ noun.s ! f}
```

If we use algebraic datatypes to guarantee that nouns always have just one variable feature, this definition is immune to changes in what exactly belongs to that feature, e.g. if it includes both number and case or just one of them.

Record types provide an alternative to one-constructor parameter types. Record types whose fields are parameter types are also parameter types, which can be pattern-matched on. Thus yet another way to define the noun linearization type is one that corresponds to **feature structures** (records of features) in unification grammar formalisms:

```
lincat N = {s : {n : Number ; c : Case} => Str}
```

4.12 Record extension and subtyping

Record types and records can be **extended** with new fields. For instance, in German it is natural to treat two-place verbs (V2) as verbs (V) with a case for the complement, which is usually accusative or dative, and is passed to the object of the verb. The symbol ****** is used for both record types and record objects.

```
lincat V2 = V ** {c : Case} ;
lin Follow = regV "folgen" ** {c = Dative} ;
```

To extend a record type or object with a field whose label it already has is a type error. It is also an error to extend a non-record type or object.

A record type T is a **subtype** of another one R , if T has all the fields of R and possibly other fields. For instance, an extension of a record type is always a subtype of it. If T is a subtype of R , then R is a **supertype** of T .

If T is a subtype of R , an object of T can be used whenever an object of R is required. For instance, a transitive verb can be used whenever a verb is required.

Covariance means that, if T is a subtype of R , a function of type $U \rightarrow T$ has also the type $U \rightarrow R$. For example, any function that produces V2's also produces V's. **Contravariance** means that a function of type $R \rightarrow U$ has also the type $T \rightarrow U$. For example, any function that operates on V's also operates on V2's, as defined above.

In pattern matching, a pattern that matches a record also matches all its extensions. Thus one can write an inflection paradigm for English nouns with invariant plurals (such as *fish*) concisely:

```
oper invarPluralN :
  Str -> {s : {n : Number ; c : Case} => Str} = \s -> {
    s = table {
      {c = Gen} => s + "'s" ;
      _ => s
    }
  }
```

4.13 Tuples and product types

Product types and tuples are syntactic sugar for record types and records:

$$T_1 * \dots * T_n \equiv \{\mathbf{p1} : T_1 ; \dots ; \mathbf{pn} : T_n\}$$

$$\langle t_1, \dots, t_n \rangle \equiv \{\mathbf{p1} = t_1 ; \dots ; \mathbf{pn} = t_n\}$$

Thus the labels $\mathbf{p1}$, $\mathbf{p2}$, $\dots$ are hard-coded. When used as patterns, tuples are translated to record patterns in the same way as tuples to records. Partial patterns make it possible to write, logically but slightly surprisingly,

```
case <g,n,p> of {
  <Fem> => t
  ...
}
```

4.14 Prefix-dependent choices and pattern macros

To conclude the presentation of GF's programming constructs, we'll show a couple that are seldom used outside resource grammar libraries. First, **prefix-dependent choice** is an expression form specifying how a string may depend on the beginning (prefix) of the next token. A prime example of its usage is the English indefinite article. A rough rule is that the article is *an* if the next token begins with a vowel, and *a* otherwise. This rule is expressed by the following prefix-dependent choice expression:

```
pre {
  "a" | "e" | "i" | "o" | "u" => "an" ;
  _ => "a"
}
```

Now, the English article rule is approximate, and can be refined by introducing more cases, which are then matched in order:

```
pre {
  "eu"  => "a" ;    -- a euphemism
  "uni" => "a" ;    -- a university
  "un"  => "an" ;   -- an uncle
  "u"   => "a" ;    -- a user
  "a" | "e" | "i" | "o" => "an" ;
  _     => "a"
}
```

Even this rule is very rough: the heart of the problem is that the form of the article depends on *pronunciation*, which is not predictable from

English spelling. In many other languages, prefix-dependent choices are more predictable. Italian is such a language; the most important example is the definite article, defined in Section 9.11.

Both prefix-dependent choice expressions and case expressions in smart paradigms typically repeat the same patterns over and over again. **Pattern macros** are a way to avoid this. Thus, for instance, a macro for vowels can be defined

```
oper vowel : pattern Str = #("a" | "e" | "i" | "o")
```

Notice that `vowel` is not just a macro, but a genuine `oper` function in the sense that it is type checked; actually any types that can have patterns can also have pattern macros. The operator `#` makes a pattern of type `T` into an expression of type `pattern T`.

When a pattern macro is used in actual pattern matching, the operator `#` switches it back from an expression to a pattern:

```
pre {
  #vowel => "an" ;
  _ => "a"
}
```

4.15 Strings at compile time vs. run time

A common difficulty in GF are the conditions under which tokens can be created. To summarize, tokens are created in the following ways:

- quoted string: `"foo"`
- gluing: `t + s`
- predefined operations: `init`, `tail`, `tk`, `dp`
- pattern matching over strings: `"y" => "ies"`
- prefix-dependent choices: `pre {...}`

The general principle is that *all tokens must be known at compile time*. This means that the above operations may not have **run-time variables** in their arguments. Run-time variables are the variables that stand for function arguments in linearization rules.

Hence it is not legal to write

```
cat Noun ;
fun Plural : Noun -> Noun ;
lin Plural n = {s = n.s + "s"} ;
```

because `n` is a run-time variable. Also

```
lin Plural n = {s = (regNoun n).s ! Pl} ;
```

is incorrect with `regNoun` as defined in Section 4.2, because the run-time variable `n` is eventually sent to string pattern matching and gluing. The

way to go here is to define the plural form inside the noun linearization, by using a table. Then the forms with and without *s* are created as separate tokens at compile time. (In this case, this is of course the linguistically valid choice as well, because plural formation cannot be iterated!)

Writing tokens together without a space is an often-wanted behaviour, for instance, with punctuation marks. Thus one might try to write

```
lin Question p = {s = p + "?"} ;
```

which is incorrect. The way to go is to use an **unlexer** that creates correct spacing after linearization—in this case, probably **unlextext** (see Section 2.15). Correspondingly, the **lexer lextext**, which analyses a string like "why?" into two tokens is needed before parsing.

Another way to force the binding of tokens in unlexing is by the **Prelude** operation

```
glue : Str -> Str -> Str
```

Thus

```
lin Question p = {s = glue p "?"}
```

renders the value of **p** followed by **?** without a space, if the unlexer **bind** is used. Internally, **glue** works by using a special token (currently **&+**), but this token should not be used explicitly because the implementation of **glue** may change.

Using the Resource Grammar Library

This chapter introduces the GF Resource Grammar Library, by giving an overview and practical advice for users of the library. The internals of the library are covered in Chapters 9 and 10. The main topics of this chapter are:

- the coverage of the Resource Grammar Library
- the structure and presentation of the library
- lexical vs. phrasal categories
- the resource grammar API (Application Programmer's Interface)
- reimplementing the **Foods** grammar and porting it to new languages
- interfaces, instances, and functors
- the division of labour between resource and application grammars
- functor overriding and compile-time transfer
- resource grammars as a linguistic ontology
- a tour of the Resource Grammar Library
- tense systems
- browsing the library

5.1 The purpose and coverage of the library

The purpose of the GF Resource Grammar Library is to provide the main grammar rules of different languages, in a form readily usable by application programmers. The library defines the low-level details of morphology and syntax, so that application programmers can concentrate on the semantic and stylistic aspects of their grammars. The guiding principle is that

grammar checking becomes type checking

that is, whatever is type-correct in the resource grammar is also grammatically correct.

The intended level of application grammarians is that of a skilled programmer with a practical knowledge of the target languages, but without theoretical knowledge about their grammars. Such a combination of skills is typical of programmers who, for instance, want to localize language software to new languages.

The library covers currently (Summer 2010) 16 languages:

Bulgarian	Catalan	Danish	Dutch
English	Finnish	French	German
Italian	Norwegian	Polish	Romanian
Russian	Spanish	Swedish	Urdu

The first three letters in language names (Bul etc, except Ron for Romanian) give the three-letter codes for languages from the ISO-639 standard. These are the codes that are used in the module names in the library. The 3-letter codes cover many more languages than the more widely used 2-letter codes and are therefore a more scalable choice.

In addition to the above-mentioned 16 languages, the artificial language Interlingua (Ina) is included. In addition, there are some partial implementations already available and sufficient for use in some applications; for instance, Arabic, Latin, and Turkish have fairly complete sets of inflectional paradigms.

5.2 Lexical vs. phrasal rules

So far in this book, we have looked at grammars from a semantic point of view: the abstract syntax defines a system of meanings, and the concrete syntax tells how they are expressed in some language. In resource grammars, as often in the linguistic tradition, the goal is more modest: to specify the **grammatically correct combinations** of words, whatever their meanings are. With this more modest goal, it is possible to achieve a much wider coverage than with semantic grammars without sacrificing precision. High-precision semantics with wide coverage would require much more work, if not be impossible.

Following the distinction between words and their combinations, the resource grammar has two kinds of categories and two kinds of rules:

- lexical:
 - lexical categories, to classify words
 - lexical rules, to define words and their properties
- phrasal (combinatorial, syntactic):
 - phrasal categories, to classify phrases of arbitrary size

- phrasal rules, to combine phrases into larger phrases

In GF, this distinction is purely heuristic: there is no formal distinction between lexical and phrasal rules. In fact, it would be impossible to maintain the distinction in multilingual grammars, where one lexical unit in one language may correspond to a multiword complex in another language, or to no word at all. For instance, the English determiner word *that* has the Swedish equivalent *den där*, and the French equivalent *ce-là*, where the two words are separated by the determined noun (e.g. *ce vin-là*, “that wine”). The English definite article *the* is in Swedish realized not by a word but by an inflectional form (*miraklet* “the miracle” from the noun *mirakel*, “miracle”); however, if the noun is modified, then an article word is added (*det oväntade miraklet*, “the unexpected miracle”).

The only maintainable way to define “lexical” and “combinatorial” is via abstract syntax: **lexical rules** are **fun** functions that take no arguments whereas **phrasal rules** are functions that do take arguments. Thus the “lexical” function **that.Det** : **Det** is linearized to one or more words according to language: *that*, *den där*, *ce-là*, etc.

Even with the reservations made above, it has turned out to be a good discipline to try to maintain a distinction between the lexical and syntactic components in the resource grammar. The same distinction carries over to application grammars: while syntactic structures are more or less the same across applications, words are essentially different, except in closely related languages. This is reflected in the module structure of multilingual applications, where it is often possible to use shared concrete syntax modules for combinations and treat just the lexicon separately in each language.

The distinction between phrasal and lexical rules thus determines the main division of resource library modules. For each language *L*, the library has

- the module **Syntax***L* with phrasal rules,
- the module **Paradigms***L* with morphological paradigms usable in the construction of lexicon.

5.3 Lexical categories and rules

Within lexical categories, there is a further classification into **closed** and **open** categories. The defining property of closed categories is that the words in them can be enumerated completely; it is very rare that any new words are introduced in them. In general, closed categories contain **structural words**, also known as **function words**, whereas open categories contain **content words**.

Examples of closed categories are

```
Det ; -- determiner    e.g. "this"
AdA ; -- adadjective  e.g. "very"
```

We have already used words of both of these categories in the **Food** examples; they have just not been assigned a category, but treated as **syncategorematic**. In GF, a syncategorematic word is one that is introduced in a linearization rule of some construction alongside with some other expressions that are combined; there is no abstract syntax tree for that word alone. Thus in the rules

```
fun That : Kind -> Item ;
lin That k = {s = "that" ++ k.s} ;
```

the word *that* is syncategorematic. In linguistically motivated grammars, syncategorematic words are avoided, because the ambition is to analyse expressions into their smallest constituents. But in application grammars, structural words are typically treated as syncategorematic, to keep the abstract syntax trees as simple as possible.

The Resource Grammar Library provides a fairly comprehensive list of structural words in the **Syntax** module. In the first example, the **Foods** grammar of Figure 20 in Section 3.7, we will use the following ones:

```
this_Det, that_Det, these_Det, those_Det : Det ;
very_AdA : AdA ;
```

The naming convention for lexical rules is that we use a word followed by the underscore `_` followed by the category. In this way we can for instance distinguish the determiner *that* from the conjunction *that*.

As for open categories, we will start with the following two:

```
N ; -- noun          e.g. "pizza"
A ; -- adjective     e.g. "good"
```

Open lexical categories have no expressions in **Syntax**. Such expressions are built as they are needed in applications. The abstract syntax of words in applications is already familiar, e.g.

```
fun Wine : Kind ;
```

Their concrete syntax is given by using the morphological paradigm library **ParadigmsEng**,

```
lin Wine = mkN "wine" ;
```

Note. The distinction between open and closed categories is not quite strict in the GF resource library. For instance, adjectives contain structural words like *very*, but they are not a completely closed cate-

gory, because one can form adjectives from adjectives: for instance, *incredibly (warm)*.

5.4 Phrasal categories and rules

There are just five phrasal categories needed in the Foods grammar:

Utt ;	-- utterance	e.g. "this pizza was warm"
Cl ;	-- clause	e.g. "this pizza is warm"
NP ;	-- noun phrase	e.g. "this warm pizza"
CN ;	-- common noun	e.g. "warm pizza"
AP ;	-- adjectival phrase	e.g. "very warm"

Utterances are top-level units in a discourse, and they can be sentences, questions, commands, or whatever can be said by someone in one utterance. Clauses are, roughly, a kind of sentences. A clause expresses a **proposition**, that is, a thinkable fact such as *this pizza is warm*, but it is neutral as for whether this proposition is true or when it is true. Clauses can be used in utterances in many ways, but the only way we need in the Foods grammar is affirmative clauses in the present tense. **Utt** is a good start category in application grammars, because it covers many of the things one may want to express.

Noun phrases and common nouns look similar, but they are radically different syntactically. Noun phrases can be used as subjects of sentences (*the dog sleeps*), whereas bare common nouns generally cannot (*?dog sleeps*). Noun phrases can be built from common nouns by adding determiners; but there are also other kinds of noun phrases, e.g. pronouns.

The syntactic combinations we need are the following:

mkUtt :	Cl	->	Utt ;	-- e.g. "this pizza is warm"
mkCl :	NP	->	AP -> Cl ;	-- e.g. "this pizza is warm"
mkNP :	Det	->	CN -> NP ;	-- e.g. "this pizza"
mkCN :	AP	->	CN -> CN ;	-- e.g. "warm pizza"
mkAP :	AdA	->	AP -> AP ;	-- e.g. "very warm"

To start building phrases, we need rules of **lexical insertion**, which form phrases from single words:

mkCN :	N	->	CN ;
mkAP :	A	->	AP ;

Notice that all (more properly, as many as possible) operations in the resource library have the name **mkC**, where *C* is the value category of the operation. This means of course heavy overloading. For instance, the library API has more than 20 operations named **mkNP**! Overloading is, however, usually not possible for structural words, because there are

many words of the same type.

Now the sentence

these very warm pizzas are Italian

can be built as follows:

```
mkUtt
  (mkCl
    (mkNP these_Det
      (mkCN (mkAP very_AdA (mkAP warm_A)) (mkCN pizza_N)))
    (mkAP italian_AP))
```

What remains is to define the concrete syntax of **Foods** so that this syntactic tree gives the linearization of the semantic tree

Pred (These (Mod (Very Warm) Pizza)) Italian

5.5 The resource API

The resource library API is divided into language-specific and language-independent parts. These parts are accessible by two different modules, which applications must open:

- **Syntax** L , for each language L , the **syntax API**, is language-independent, i.e. has the same types and functions for all languages.
- **Paradigms** L for each language L , the **morphology API**, is language-specific, i.e. has partly different types and functions for different languages.

A comprehensive documentation of the API is given on line. To implement the **Foods** grammar, we will only need a small fragment of the full API. Let us summarize this fragment by a set of tables of the same form as used in the on-line documentation. The tables belonging to **Syntax** are given in Figure 34.

The morphological paradigms needed in the **Foods** grammar for some languages are given in Figure 35. The smart paradigms will take care of variations such as *vino-vini*, *formaggio-formaggi*, and *pizza-pizze* in Italian, and also infer genders correctly in many cases. German noun inflection and gender are less predictable, which means that the paradigms need more arguments. Finnish, despite its complexity, is so predictable that we only need the one-argument smart paradigms in the **Foods** grammar.

5.6 The library path

If you have installed the precompiled libraries of GF, or compiled the libraries yourself, you will find them in some directory, such as

Category	Explanation	Example
Utt	utterance (sentence, question,...)	<i>who are you</i>
Cl	clause, with all tenses	<i>she looks at this</i>
AP	adjectival phrase	<i>very warm</i>
CN	common noun (without determiner)	<i>red house</i>
NP	noun phrase (subject or object)	<i>the red house</i>
AdA	adjective-modifying adverb,	<i>very</i>
Det	determiner	<i>this</i>
A	one-place adjective	<i>warm</i>
N	common noun	<i>house</i>

Function	Type	Example
mkUtt	Cl -> Utt	<i>John is very old</i>
mkCl	NP -> AP -> Cl	<i>John is very old</i>
mkNP	Det -> CN -> NP	<i>this old man</i>
mkCN	N -> CN	<i>house</i>
mkCN	AP -> CN -> CN	<i>very big blue house</i>
mkAP	A -> AP	<i>old</i>
mkAP	AdA -> AP -> AP	<i>very very old</i>

Function	Type	In English
this_Det	Det	<i>this</i>
that_Det	Det	<i>that</i>
these_Det	Det	<i>this</i>
those_Det	Det	<i>that</i>
very_AdA	AdA	<i>very</i>

FIGURE 34 Resource grammar API presentations for the categories, syntactic functions, and structural words needed in the Foods grammar.

English:

Function	Type
mkN	(dog : Str) -> N
mkN	(man,men : Str) -> N
mkA	(cold : Str) -> A

Italian:

Function	Type
mkN	(vino : Str) -> N
mkA	(caro : Str) -> A

German:

Function	Type
Gender	Type
masculine	Gender
feminine	Gender
neuter	Gender
mkN	(Stufe : Str) -> N
mkN	(Bild,Bilder : Str) -> Gender -> N
mkA	(klein : Str) -> A
mkA	(gut,besser,beste : Str) -> A

Finnish:

Function	Type
mkN	(talo : Str) -> N
mkA	(hieno : Str) -> A

FIGURE 35 Inflection paradigms needed for the Foods grammar.

`/usr/local/lib/gf` in Unix-like environments. GF uses the environment variable `GF_LIB_PATH` to locate this library. This variable should be set to the root of the library. To see if so is the case, you can try

```
$ echo $GF_LIB_PATH
```

which should show the directory. If the variable is not set, you can do

```
$ export GF_LIB_PATH=/usr/local/lib/gf
```

if you are in the Bash shell. This is the most common environment in most Linux and Mac OS X systems; in other shells, the command can be different, for instance, `setenv`. It is a good practice to set this variable once and for all in an initialization script, such as `.bashrc` if you use the Bash shell.

The directory `GF_LIB_PATH` has two subdirectories:

- **alltenses**, containing all tense forms
- **present**, containing only the present tense forms, infinitives, and participles.

These subdirectories contain the same modules, but in two versions. For instance, there is both a file `alltenses/SyntaxEng.gfo` and a file `present/SyntaxEng.gfo`, and so on. These files are produced from the same source, `src/api/SyntaxEng.gf`. But the **present** version is compiled with a preprocessor that ignores a part of the code. The reason for doing this is that, in many applications of GF, only the present tense is needed, and having all the other verb forms available would consume unnecessary space (up to six times more than just including the present tense).

If you have `GF_LIB_PATH` set correctly, you can import a library from wherever you are in the file system. For instance, you can import the German paradigms and test them as follows:

```
> import -retain present/ParadigmsGer.gfo
> compute_concrete -table mkN "Farbe"
```

Recall that you need the `-retain` flag to test the paradigms, as explained in Section 3.8. The directory name **present** in the imported file, is interpreted, primarily, relative to the current directory; if there is no subdirectory named **present** in the current directory, then the value of `GF_LIB_PATH` is prepended. Notice also that the imported file has the suffix `.gfo` (GF object file), since the library consists of compiled files and not their sources.

The code example discussed in Section 5.7 and shown in Figure 36 has a **path directive**

```
--# -path=.:present
```

as its first line, indicating which directories are searched to find all files needed for compiling the grammar. It contains two directories: the current directory (`.`), i.e. the directory in which `FoodsREng.gf` itself resides, and `present`, which is relative to the environment variable `GF_LIB_PATH`. Path directives can be given both in files and in import commands, for instance,

```
> import -path=.:present FoodsREng.gf
```

If the `import` command has a path flag, it overrides the path given in the file. Only the path in the top file (the one explicitly given in the import command) is ever taken into account.

5.7 Example: English

Now we are ready to build an application of the library. We work with the abstract syntax `Foods` from Figure 20 in Section 3.7, and build first an English implementation. We don't need to think about inflection and agreement, since we just pick appropriate functions from the Resource Grammar Library.

The code is given in Figure 36. The concrete syntax opens `SyntaxEng` and `ParadigmsEng` to get access to the resource libraries needed.

As linearization types, we use utterances for `Comment`, noun phrases for `Item`, common nouns for `Kind`, and adjectival phrases for `Quality`. These types fit perfectly with the way we have used these categories in the application; hence the combination rules we need almost write themselves automatically.

The lexical part of the grammar uses resource paradigms. Notice that we have to apply the lexical insertion rules (`mkAP` on `A` and `mkCN` on `N`) to get type-correct linearizations. Notice also that we need to use the two-place noun paradigm for *fish*, but everything else is regular.

It is not recommendable to use lexical categories as linearization types in an application grammar, because they don't scale up to future extensions. For instance, even if the category `Kind` in `Foods` only had atomic constructors such as `Wine` and `Pizza`, which would permit the use of `N`, new atomic constructors might require complex `CN`'s as linearizations. For example, `SparklingWine` might be introduced as an atomic kind, because the adjective *sparkling* does not freely combine with the other kinds and would hence not be suitable as a `Quality`. Moreover, the property of being lexically simple varies from one language to another: in German, for instance, *sparkling wine* does have a single-word expression, *Sekt*.

Exercise 5-1. Compile the grammar `FoodsREng` and generate and parse

```
--# -path=.:present

concrete FoodsREng of Foods =
  open SyntaxEng,ParadigmsEng in {
    lincat
      Comment = Utt ;
      Item = NP ;
      Kind = CN ;
      Quality = AP ;
    lin
      Pred item quality = mkUtt (mkCl item quality) ;
      This kind = mkNP this_Det kind ;
      That kind = mkNP that_Det kind ;
      These kind = mkNP these_Det kind ;
      Those kind = mkNP those_Det kind ;
      Mod quality kind = mkCN quality kind ;
      Wine = mkCN (mkN "wine") ;
      Pizza = mkCN (mkN "pizza") ;
      Cheese = mkCN (mkN "cheese") ;
      Fish = mkCN (mkN "fish" "fish") ;
      Very quality = mkAP very_AdA quality ;
      Fresh = mkAP (mkA "fresh") ;
      Warm = mkAP (mkA "warm") ;
      Italian = mkAP (mkA "Italian") ;
      Expensive = mkAP (mkA "expensive") ;
      Delicious = mkAP (mkA "delicious") ;
      Boring = mkAP (mkA "boring") ;
    }
  }
```

FIGURE 36 A resource-based English concrete syntax of Foods.

some sentences.

Exercise 5-2. Test if `FoodsREng` and the manually built `FoodsEng` given in Figure 21 in Section 3.5 yield exactly the same linearizations. You can do this systematically by importing these two grammar and then producing a set of trees and their linearizations by

```
> generate_trees | linearize -treebank
```

Exercise 5-3. Write a concrete syntax of `Foods` for Italian or some other language included in the resource library. You can compare the results with the hand-written grammars presented earlier in this tutorial.

5.8 Functor implementation of multilingual grammars

If you did the exercise in the previous section and wrote a concrete syntax of `Foods` for some other language, you probably noticed that much of the code looks exactly the same as for English. The reason for this is that the `Syntax` API is the same for all languages. This is in turn possible because all languages in the GF Resource Grammar Library implement the same syntactic structures. Moreover, languages tend to use syntactic structures in similar ways to express meanings: usually, it is only the lexical parts of a concrete syntax that we need to write anew for a new language. Thus, to port a grammar to a new language, you

1. copy the concrete syntax of an already given language
2. change the words (strings and inflection paradigms)

However, we should do better, because programming by copy-and-paste is not worthy of a functional programmer! Thus we will now show how to write a *function* that takes care of the shared parts of grammar modules. It is not a function in the `fun` or `oper` sense, but a function operating on modules, called a **functor**. This construct is familiar from the functional programming languages ML and OCaml, but it does not exist in Haskell. It also bears some resemblance to templates in C++. Functors are also known as **parametrized modules**.

In GF, a functor is a module that **opens** one or more **interfaces**. An **interface** is a module similar to a **resource**, but it contains only the *types* of **opers**, not their definitions. You can think of an interface as a kind of a record type. The **oper** names are the labels of this record type. The corresponding record is called an **instance** of the interface. Thus a functor is a module-level function taking instances as arguments and producing modules as values.

```

incomplete concrete FoodsI of Foods =
  open Syntax, LexFoods in {
    lincat
      Comment = Utt ;
      Item = NP ;
      Kind = CN ;
      Quality = AP ;
    lin
      Pred item quality = mkUtt (mkCl item quality) ;
      This kind = mkNP this_Det kind ;
      That kind = mkNP that_Det kind ;
      These kind = mkNP these_Det kind ;
      Those kind = mkNP those_Det kind ;
      Mod quality kind = mkCN quality kind ;
      Very quality = mkAP very_AdA quality ;

      Wine = mkCN wine_N ;
      Pizza = mkCN pizza_N ;
      Cheese = mkCN cheese_N ;
      Fish = mkCN fish_N ;
      Fresh = mkAP fresh_A ;
      Warm = mkAP warm_A ;
      Italian = mkAP italian_A ;
      Expensive = mkAP expensive_A ;
      Delicious = mkAP delicious_A ;
      Boring = mkAP boring_A ;
  }

```

FIGURE 37 A functor for the concrete syntax of Foods.

Let us now write a functor implementation of the Food grammar. The full code is given in the module `FoodsI` in Figure 37. The module header uses the keyword `incomplete` to indicate that `FoodsI` is a functor. The word “incomplete” suggests that the functor is not a complete concrete syntax by itself, but needs completion. What makes a module incomplete is that it opens one or more interfaces—in this case, two interfaces, `Syntax` and `LexFoods`.

In the functor-function analogy, `FoodsI` could be seen as a function with the following type signature:

```

FoodsI : instance of Syntax ->
         instance of LexFoods -> concrete of Foods

```

```
--# -path=.:present

concrete FoodsIEng of Foods = FoodsI with
  (Syntax = SyntaxEng),
  (LexFoods = LexFoodsEng) ;
```

FIGURE 38 An English functor instantiation for Foods.

Such a function takes, as its arguments, instances of two interfaces:

- **Syntax**, the resource grammar interface
- **LexFoods**, the domain-specific lexicon interface

To obtain a complete concrete syntax, the functor has to be applied to instances of the interfaces. For example,

```
FoodsI SyntaxEng LexFoodsEng : concrete of Foods
FoodsI SyntaxFin LexFoodsFin : concrete of Foods
FoodsI SyntaxGer LexFoodsGer : concrete of Foods
```

The application of a functor is called a **functor instantiation**. The notation for it is shown in Figure 38. Notice that Figure 38 gives the *whole* module, not just its header. The module body is constructed by the compiler from **FoodsI**, by instantiating the interface constants with their definitions given in the English instances **LexFoodsEng** and **SyntaxEng**.

5.9 Interfaces and instances

It remains to show what interfaces and instances exactly look like. The interface **LexFoods** is shown in Figure 39. An English instantiation is shown in Figure 40. In this interface, only lexical items are declared. In general, an interface can declare any functions and also types. The **Syntax** interface does so: it consists mainly of overloaded operations that combine phrases.

Notice that when an interface opens an interface, such as **Syntax** here, then its instance has to open an instance of that interface. But the instance may also open some other resources—very typically, like in Figure 40, a domain lexicon instance opens a **Paradigms** module.

5.10 A design pattern for multilingual grammars

Functors opening **Syntax** and a domain lexicon interface are in fact so typical in GF applications, that this structure could be called a **design pattern** for GF grammars. What makes this pattern so useful is that languages tend to use the same syntactic structures and differ only in

```

interface LexFoods = open Syntax in {
  oper
    wine_N : N ;
    pizza_N : N ;
    cheese_N : N ;
    fish_N : N ;
    fresh_A : A ;
    warm_A : A ;
    italian_A : A ;
    expensive_A : A ;
    delicious_A : A ;
    boring_A : A ;
}
    
```

FIGURE 39 A lexicon interface for Foods.

```

instance LexFoodsEng of LexFoods =
  open SyntaxEng, ParadigmsEng in {
    oper
      wine_N = mkN "wine" ;
      pizza_N = mkN "pizza" ;
      cheese_N = mkN "cheese" ;
      fish_N = mkN "fish" "fish" ;
      fresh_A = mkA "fresh" ;
      warm_A = mkA "warm" ;
      italian_A = mkA "Italian" ;
      expensive_A = mkA "expensive" ;
      delicious_A = mkA "delicious" ;
      boring_A = mkA "boring" ;
  }
    
```

FIGURE 40 An English instance of the lexicon interface for Foods.

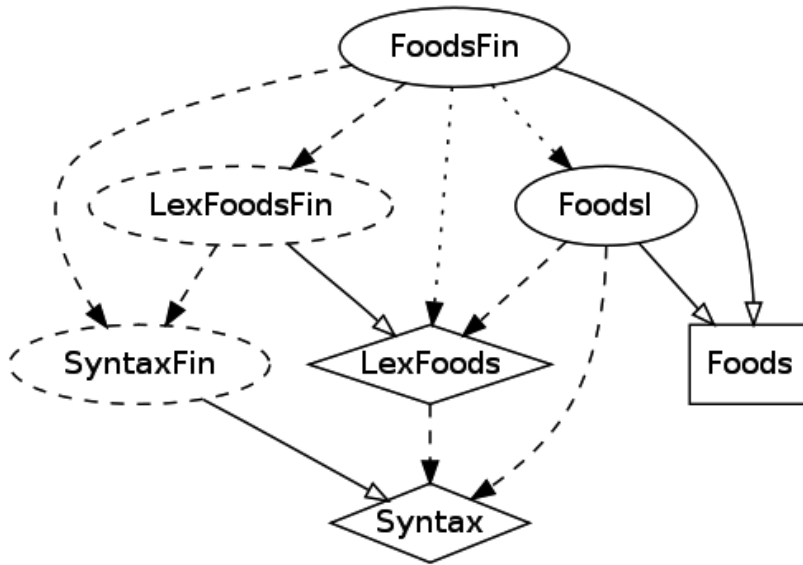


FIGURE 41 The design pattern with a functor over the resource library and a domain lexicon interface.

words. Thus, once we have an application grammar defined by using a functor, adding a new language is simple. Just two modules need to be written:

- a domain lexicon instance
- a functor instantiation

The functor instantiation is completely mechanical to write. The domain lexicon instance requires some knowledge of the words of the language: what words are used for which concepts, how the words are inflected, plus features such as genders. Figure 41 shows the module structure as instantiated to one language. Figure 42 shows a few more instances of the *Foods* lexicon. Figure 43 shows the top-level instantiations of the functor for the same languages.

A variant of the functor-based design pattern follows the structure shown in Section 4.8. In this variant, the grammar itself is divided into a syntax and a lexicon part. The syntax part is implemented with a functor, but one that only needs the *Syntax* interface. The lexicon part is implemented in the ordinary way, without using a functor. This variant is conceptually simpler than the one we have used, but has the disadvantage that it may be difficult to decide what belongs to the

```

instance LexFoodsIta of LexFoods =
  open SyntaxIta, ParadigmsIta in {
    oper
      wine_N = mkN "vino" ;
      pizza_N = mkN "pizza" ;
      cheese_N = mkN "formaggio" ;
      fish_N = mkN "pesce" ;
      fresh_A = mkA "fresco" ;
      warm_A = mkA "caldo" ;
      italian_A = mkA "italiano" ;
      expensive_A = mkA "caro" ;
      delicious_A = mkA "delizioso" ;
      boring_A = mkA "noioso" ;
  }
instance LexFoodsGer of LexFoods =
  open SyntaxGer, ParadigmsGer in {
    oper
      wine_N = mkN "Wein" ;
      pizza_N = mkN "Pizza" "Pizzen" feminine ;
      cheese_N = mkN "Käse" "Käsen" masculine ;
      fish_N = mkN "Fisch" ;
      fresh_A = mkA "frisch" ;
      warm_A = mkA "warm" "wärmer" "wärmste" ;
      italian_A = mkA "italienisch" ;
      expensive_A = mkA "teuer" ;
      delicious_A = mkA "köstlich" ;
      boring_A = mkA "langweilig" ;
  }
instance LexFoodsFin of LexFoods =
  open SyntaxFin, ParadigmsFin in {
    oper
      wine_N = mkN "viini" ;
      pizza_N = mkN "pizza" ;
      cheese_N = mkN "juusto" ;
      fish_N = mkN "kala" ;
      fresh_A = mkA "tuore" ;
      warm_A = mkA "lämmin" ;
      italian_A = mkA "italialainen" ;
      expensive_A = mkA "kallis" ;
      delicious_A = mkA "herkullinen" ;
      boring_A = mkA "tylsä" ;
  }

```

FIGURE 42 Italian, German, and Finnish instances of the Foods lexicon.

```

--# -path=:present

concrete FoodsIta of Foods = FoodsI with
  (Syntax = SyntaxIta),
  (LexFoods = LexFoodsIta) ;

--# -path=:present

concrete FoodsGer of Foods = FoodsI with
  (Syntax = SyntaxGer),
  (LexFoods = LexFoodsGer) ;

--# -path=:present

concrete FoodsFin of Foods = FoodsI with
  (Syntax = SyntaxFin),
  (LexFoods = LexFoodsFin) ;

```

FIGURE 43 Italian, German, and Finnish instantiations of the `FoodsI` functor.

syntax and what belongs to the lexicon.

Exercise 5-4. Instantiate the functor `FoodsI` to some language of your choice.

Exercise 5-5. Divide the `Foods` grammar into a syntax and a lexicon part, and implement it for some language by using the Resource Grammar Library and following the variant suggested at the end of this section.

5.11 Division of labour revisited

The purpose of resource grammars as stated in Section 1.7 is a division of labour between linguists and application grammarians. We can now reflect on what this means more precisely, by asking ourselves what skills are required of grammarians working on different components.

Building a GF application starts from the abstract syntax. Writing an abstract syntax requires

- understanding of the semantic structure of the application domain
- knowledge of the GF fragment with categories and functions

If the concrete syntax is written by using a functor, the programmer has to decide what parts of the implementation are put to the interface and what parts are shared in the functor. This requires

- knowing how the domain concepts are expressed in natural language
- knowledge of the Resource Grammar Library—the categories and operations
- understanding what parts are likely to be expressed in language-dependent ways, so that they are put to an interface and not defined in the functor
- knowledge of the GF fragment with function applications and strings

Instantiating a ready-made functor to a new language is less demanding. It requires essentially

- knowing how the domain words are expressed in the language
- knowing, roughly, how these words are inflected
- knowledge of the paradigms available in the library
- knowledge of the GF fragment with function applications and strings

Notice that none of these tasks requires the use of GF records, tables, or parameters. Thus only a small fragment of GF is needed; the rest of GF is only relevant for those who write the libraries. Essentially, all the machinery introduced in Chapters 3 and 4 is unnecessary! Except for the unfortunate fact that some knowledge of records and tables may be needed for understanding GF’s error messages.

Of course, grammar writing is not always just straightforward usage of libraries. It may happen that a semantic category cannot be simply mapped into a resource category, but needs some extra information or perhaps a combination of many categories, expressible by records and parameters. A typical example is place expressions and their prepositions, in almost any language. In English, for instance, one says *at the airport*, *at the pub*, but *in the church*, *in the restaurant*. In a grammar that uses place expressions both with and without prepositions, the lexicon needs to store both nouns and the associated prepositions:

```
lincat Place = {name : CN ; prep : Prep}
```

We will return to this idea in the exercises of Section 5.18.

5.12 Overriding a functor

A functor implementation using the resource **Syntax** interface works well as long as all concepts are expressed by using the same structures in all languages. If this is not the case, the deviant linearization can be made into a parameter and moved to the domain lexicon interface.

The **Foods** grammar works so well that we have to take a contrived example: assume that English has no word for **Pizza**, but has to use the paraphrase *Italian pie*. This paraphrase is no longer a noun N, but

a complex phrase in the category CN. An obvious way to solve this problem is to change the interface `LexFoods` so that `pizza_N` is replaced by the more general

```
oper pizza_CN : CN
```

The functor must then be changed accordingly. But this solution is unstable: we may end up changing the interface and the functor with each new language, and we must every time also change the interface instances for the old languages to maintain type correctness.

A better solution is to use the technique of restricted inheritance introduced in Section 4.8: the English instantiation inherits the functor implementation but excludes the constant `Pizza`. This is how we write:

```
--# -path=:present

concrete FoodsEng of Foods = FoodsI - [Pizza] with
  (Syntax = SyntaxEng),
  (LexFoods = LexFoodsEng) **
  open SyntaxEng, ParadigmsEng in {

  lin Pizza = mkCN (mkA "Italian") (mkN "pie") ;
}
```

The notation with `**` is a very special case of module inheritance. If the functor is itself built as an extension of some modules, the corresponding inherited modules appear before the functor instantiation:

```
concrete C of A = B1,B2 ** CI with (...) ** open R in...
```

Exercise 5-6. Generalize the functor `FoodsI` and the lexicon interface `Foods` so that it is not necessary to override it as we did in this section.

5.13 Compile-time transfer

Overriding a functor comes out handy in those situations where one language does not preserve the syntactic structure used in the other languages. One typical example is the English verb *like*. The sentence *I like this pizza* in Swedish is *jag gillar den här pizzan*, which has the same structure and can hence be implemented with the functorial rule

```
lin ILike it = mkCl i_NP like_V2 it
```

where the interface constant `like_V2` is declared and instantiated as follows:

```
oper like_V2 : V2
oper like_V2 = mkV2 "like"           -- English
```

```
oper like_V2 = mkV2 (mkV "gilla") -- Swedish
```

Italian, however, performs a structural change: the translation is *questa pizza mi piace*, where the roles of the subject and the object are swapped. The literal correspondent of the Italian sentence would be *this pizza pleases me*, but Italian has no version that literally preserves the structure of *I like this pizza*. In an application grammar, it is still easy to give the linearization rule,

```
lin ILike it = mkC1 it piacere_V2 i_NP
```

but a change of structure has now taken place. If the functor is used in the general grammar, it has to be overridden in the Italian instantiation. An alternative is, again, to make the interface more general:

```
oper i_like : NP -> C1
oper i_like it = mkC1 i_NP (mkV2 "like") it -- English
oper i_like it = mkC1 it piacere_V2 i_NP    -- Italian
```

In many translation systems, structural changes are treated by transformation rules executed at run time. In GF, such transformations can usually be avoided at run time, because they are executed when the grammar is compiled. In the current example, the final outcome of compiling the rules in each language is similar to the hand-written rules

```
lin ILike it = {s = "I like" ++ it.s ! Acc}
lin ILike it = {s = "jag gillar" ++ it.s ! Acc}
lin ILike it = {s = it.s ! Nom ++ "mi piace"}
```

(where we ignore some variation and agreement for simplicity). The term **compile-time transfer** is used in GF-based systems for the selection of different resource grammar structures for the same semantic structures.

Exercise 5-7. Add the `ILike` construct to the `Foods` grammar and implement it for your favourite languages. Also invent some other constructs where compile-time transfer is necessary.

Exercise 5-8. Test word alignment (as explained in Section 2.14) with grammar rules that perform compile-time transfer.

5.14 The resource grammar as a linguistic ontology

Resource grammars are syntactic, rather than semantic, grammars. They aim to cover the syntactic structures of languages, independently of how these structures are used for different purposes and on different domains. How can we make sense of this in GF—in particular, how

can we build an abstract syntax without having a semantic structure in mind?

The way we think of the abstract syntax of resource grammars is as a **linguistic ontology**. It is a domain of linguistic objects, organized in linguistic categories and following strict rules of combination. Two millennia of research and education in the Western grammatical tradition have set up a rather standard set of concepts, which in the 20th century has also been formalized as precise mathematical models, and in later decades, as computer implementations. The GF Resource Grammar Library is an example of such a model and its implementation.

The idea of a language-independent linguistic ontology is one ingredient of this Western tradition. It started with Ancient Greek and was soon adapted to Latin. Since Greek and Latin are closely related languages, this was not difficult. Later on, the same concepts were applied to modern European languages and even to languages from other families, such as Finnish and Arabic. Sometimes these classical grammar concepts are a straight-jacket, which does not really fit all languages. Nevertheless, having a common conceptual framework has both practical and theoretical advantages:

- Foreign language learners are helped by concepts familiar from previous languages.
- Resource grammar implementation can exploit previous work. In GF, the abstract syntax gives a “check list” of things that the grammar must cover. Also concrete syntax code may be reusable.
- Application grammar writing is maximally concise, if the resource grammar is available as an interface and permits the use of functors.
- Language typology is given precise concepts to talk about the similarities and differences of languages.

So, what things are common to languages and where are the differences? When looking at the table of contents of a traditional grammar book, of almost any language, we usually find a chapter on nouns, another one on adjectives, on verbs, and probably on some other parts of speech as well. Often there are also chapters on syntax, covering topics such as sentence formation.

For each part of speech, a typical grammar book has sections on inflectional features (such as case, gender, and tense), and then, as a very substantial part, the different inflection paradigms. These parts vary a lot from one language to another. For instance, gender has three values in some languages, two in others. Some languages, e.g. Finnish, have no gender. The overall picture that grammar books give is that parts of speech are largely common to languages, whereas features and

inflection, and of course words themselves, are different.

The above picture fits GF very nicely: the common parts belong to an abstract syntax, and the distinguishing parts to concrete syntaxes. It also fits the structure of the resource API introduced in Section 5.5, where the main parts are a language-independent **Syntax** module and a set of language-dependent **Paradigms** modules.

Despite the large set of shared syntactic structures, it would be untenable to claim that *all* syntactic structures are shared. To take care of those structures that are not common to all languages, the library has a module named **ExtraL** for each language *L*. These **Extra** modules extend the common **Syntax** in different ways. To take an example, the formation of question words like *how old*, where the interrogative adverb *how* modifies an adjective, is provided for Germanic languages and Finnish but not for Romance languages. Since this construction is not available, its translation must use compile-time transfer. For instance, the question *how old are you* is in French *quelle âge avez-vous*, “what age do you have”. Another example of **Extra** functions is provided by the different tense systems of languages; see Section 5.17. Also see Section 10.1 for the overall module structure and the place of the **Extra** modules in it.

5.15 A tour of the resource API

Let us now make a quick tour of the resource API, to complete the fragmentary picture received when just having the implementation of **Foods** as our goal. We will refer to Appendix D for more details on categories and functions, and just give the general principles that help to navigate in the full API.

The main categories needed by the user are the following, from top to the bottom. The order of presentation follows roughly the category tree shown in Section D.1.

- **Text**, texts, built from phrases and punctuation.
- **Phr**, phrases, utterances with optional leading conjunctions and vocatives.
- **Utt**, utterances, which can be declarative sentences, questions, imperatives, or smaller phrases.
- **Punct**, major punctuation marks, “.”, “?”, “!”.
- **S**, **QS**, **RS**, **Imp**, sentences, questions, relative clauses, and imperatives, built correspondingly from **Cl**, **QC1**, **RC1**, **VP** by fixing tense and polarity
- **Cl**, clauses with variable tense and polarity, built from a noun phrase and a verb phrase by predication

- QCl, question clauses with variable tense and polarity, built from
 - IP, interrogative pronoun, with a verb phrase: *who sleeps*
 - interrogative pronoun corresponding to a non-subject constituent: *whom do you love*
 - clauses, as sentential questions: *does John sleep*
 - IAdv, interrogative adverb modifying a clause: *where does John sleep*
- RCl, relative clauses with variable tense and polarity, built from
 - RP, relative pronoun, with a verb phrase: *that sleeps*
 - relative pronoun for a non-subject constituent: *that you love*
- NP, noun phrases, built from
 - common nouns with Det (determiners): *this man*
 - Pron (pronouns): *she*
 - PN (proper names): *John*
 - plain determiners: *this*
- VP, verb phrases, built from
 - V, one-place verbs: *sleep*
 - V2, two-place verbs with an NP complement: *love her*
 - V3, three-place verbs with two NP complements: *give it to John*
 - VS, verbs with a sentence complement: *say that it is good*
 - VV, verbs with a verb phrase complement: *want to buy it*
 - adjectival phrases: *is good*
 - adverbial phrases: *is in the house*
 - noun phrases: *is a man*
 - from verb phrases by adding an adverb: *sleeps here*
- CN (common nouns), built from
 - N, lexical nouns: *man*
 - N2, relational nouns, with a complement: *son of John*
 - common nouns with adjectival modification: *old man*
 - common nouns with a relative clause: *man that I saw*
- Det, determiners, built from
 - Quant, quantifiers: *this*
 - Numeral, numerals: *five*
 - possessive forms of pronouns: *my*
- AP, adjectival phrases, built from
 - A, lexical adjectives: *old*
 - A2, two-place adjectives, with a complement: *divisible by 3*
 - lexical adjectives in comparative and a complement: *older than you*

- Adv, adverbs and adverbial phrases, built from
 - Prep (prepositions) and noun phrases: *with John*
 - adjectives: *slowly*
 - Subj (subjunctions) with subordinate clauses: *because it is hot*
 - lexical adverbs: *here*

Coordination is an operation available for many categories, with a general scheme for coordinating lists $[C]$ and, as a special case, two-element coordination:

$\text{mk}C : \text{Conj} \rightarrow [C] \rightarrow C$

$\text{mk}C : \text{Conj} \rightarrow C \rightarrow C \rightarrow C$

The list category $[C]$ is definable for any category C , and actually an abbreviation for **List** C , as explained in Section C.4.3.

The naming of functions in the library has the following conventions:

- $\text{mk}C$ is an overloaded function that construct objects of type C
- word_C is a lexical item of type C
- $\text{descr}C$ is any other operation constructing C , with the description *descr*

5.16 Flattening of constructions

The resource grammar description in Appendix D lists the minimal, most general functions. The full API (documented on the GF web page) gives many more functions, which are obtained from the minimal functions in a systematic way by using rules explained in Section D.2. The most important of them is **flattening**, that is, replacing a subtree argument by the sequence of its subtrees.

A prime example of flattening are the different versions of predication. The most general predication rule is

$\text{mk}C1 : \text{NP} \rightarrow \text{VP} \rightarrow C1$

which uses the category of verb phrases and actually covers all forms of predication. To form verb phrases, there is a function

$\text{mkVP} : C \rightarrow X \rightarrow \text{VP}$

for each verb category C and its list of arguments X . For example,

$\text{mkVP} : V2 \rightarrow \text{NP} \rightarrow \text{VP}$

But subtrees of type **VP** can be flattened. Hence each verb category C with the list of arguments X also has the rule

$\text{mk}C1 : \text{NP} \rightarrow C \rightarrow X \rightarrow C1$

An example is

mkC1 : NP -> V2 -> NP -> C1

Thus it is possible to write both of

mkC1 np v2 np'

mkC1 np (mkVP v2 np')

to get the same result. But the former expression is flatter and therefore easier to use, and it also hides the theoretically laden VP category. VP is only necessary in recursive VP forming rules, such as the one that gives complements to verb-complement verbs (VV),

mkVP : VV -> VP -> VP

Predication with adjectives and nouns uses two levels more than predication with verbs. Thus an adjective and its relevant complements are first collected into an **adjectival phrase** (AP), which can be used as a **complement phrase** (Comp) and finally as a verb phrase. The deepest structure of a clause like *this is warm* has thus four levels, and many possible shortcuts. With the minimal constructions, the structure is

mkC1 np (mkVP (mkComp (mkAP a)))

But just like VP, also Comp and AP can be flattened. The flattest expression for the same structure becomes

mkC1 np a

Predication with noun phrases (NP) and adverbs (Adv) has similar flattening options.

5.17 Tense and polarity

A clause (C1) combines a subject with a predicate and thereby expresses what in logic is called a proposition. But it does not determine what exactly is said about this proposition: whether it is in the past, present, or future, or whether it is true or false. These two components are fixed by adding the elements of **tense** and **polarity**, respectively. The expression resulting from fixing the tense and the polarity of a clause is a **sentence** (S).

Polarity is either positive or negative in all languages, whereas tenses differ from one language to another. The GF Resource Grammar Library gives every language its own tense system. However, the common API gives access to a common system, where tenses are built from two components: **anteriority**, expressing whether something is simultaneous or anterior (*to do* vs. *to have done*), and **temporal order**, saying whether something is past, present, future, or conditional (*did* vs. *does* vs. *will do* vs. *would do*). All 16 ($2 \times 2 \times 4$) combinations of polarity,

Form	English	Italian
Sim Pres Pos	<i>I sleep</i>	<i>dormo</i>
Sim Pres Neg	<i>I don't sleep</i>	<i>non dormo</i>
Sim Past Pos	<i>I slept</i>	<i>dormivo</i>
Sim Past Neg	<i>I didn't sleep</i>	<i>non dormivo</i>
Sim Fut Pos	<i>I will sleep</i>	<i>dormirò</i>
Sim Fut Neg	<i>I won't sleep</i>	<i>non dormirò</i>
Sim Cond Pos	<i>I would sleep</i>	<i>dormirei</i>
Sim Cond Neg	<i>I wouldn't sleep</i>	<i>non dormirei</i>
Ant Pres Pos	<i>I have slept</i>	<i>ho dormito</i>
Ant Pres Neg	<i>I haven't slept</i>	<i>non ho dormito</i>
Ant Past Pos	<i>I had slept</i>	<i>avevo dormito</i>
Ant Past Neg	<i>I hadn't slept</i>	<i>non avevo dormito</i>
Ant Fut Pos	<i>I will have slept</i>	<i>avrò dormito</i>
Ant Fut Neg	<i>I won't have slept</i>	<i>non avrò dormito</i>
Ant Cond Pos	<i>I would have slept</i>	<i>avrei dormito</i>
Ant Cond Neg	<i>I wouldn't have slept</i>	<i>non avrei dormito</i>

FIGURE 44 The common system of tense and polarity as shown in English and Italian.

anteriority, and order are thus available. Figure 44 gives examples of each combination for an English and an Italian sentence.

In addition to the common tense and polarity forms given in Figure 44, English also has separate forms for noncontracted negation (such as *I do not sleep*) and inverted clauses (*do I sleep*). Italian has simple past tenses (*dormii* “I (suddenly) slept”) and the conjunctive mood (*dorma* “(that) I sleep”). An easy way to see all forms of a clause in a given language is to use `linearize -table` in the GF shell. For example:

```
> import alltenses/LangEng.gfo
> parse -cat=C1 "I sleep" | linearize -table
```

Each of the three parameters has its default value: anteriority is by default simultaneous, tense is present, and polarity is positive. The function `mkS` that forms sentences from clauses takes each of these features as optional: if none is provided, the default value is used. According to Section D.2.2, the type of `mkS` forming sentences from clauses is the following:

```
mkS : (Tense) -> (Ant) -> (Pol) -> C1 -> S
```

Here parentheses are used to express the optionality of arguments. This use of parentheses is purely for documentation, since the GF compiler

Function	Type	Example
conditionalTense	Tense	(<i>John would walk</i>)
futureTense	Tense	(<i>John will walk</i>)
pastTense	Tense	(<i>John walked</i>)
presentTense	Tense	(<i>John walks</i>) [default]
anteriorAnt	Ant	(<i>John has walked</i>)
simultaneousAnt	Ant	(<i>John walks</i>) [default]
negativePol	Pol	(<i>John doesn't walk</i>)
positivePol	Pol	(<i>John walks</i>) [default]

FIGURE 45 Library constants for the common system of tense and polarity.

does not care about superfluous parentheses. In the library implementation, `mkS` thus has eight separate overloaded function instances, corresponding to whether each of the three arguments is present.

The tense system is one of those places that clearly shows the syntactic nature of the library—that the common abstract syntax does not guarantee translation equivalence. Thus the simultaneous past tense (*he walked*) should in Romance languages and in German be translated by the anterior present, except when the activity is continuous. In English, on the other hand, the use of the simultaneous present (*he walks*) is more limited than in most other languages: the progressive (*he is walking*) must often be used where most other languages would use the simultaneous present. Linguistically the progressive is an example of **aspect**. The past-tense distinctions in Romance and German are likewise examples of aspect distinctions. At the time of writing, the GF Resource Grammar Library does not give a high-level access to aspect. It just provides the different clausal forms, and the user of the library has to select the correct form herself.

The library constants for tenses and polarities are shown in Figure 45. They are examples of **abstract features**: they are not parameters in the sense of concrete syntax, because they are shared by all languages. They are often not realized as clear-cut words, but rather as parameters from the concrete syntax. However, they are still abstract in the sense that they are not in a one-to-one correspondence with features in concrete syntaxes.

Not only is the actual usage of tenses different from one language to another: also the set of available tenses varies. For instance, Romance languages have separate compound and simple past tenses (e.g. *a marché* vs. *marcha* for “(has) walked” in French). These “extra” tenses, which are not available in all languages, are accessible via the

Extra*L* modules such as ExtraFre.

5.18 Browsing the library

The resource grammar is internally implemented as a top-level grammar, whose abstract syntax is called **Lang**. This syntax contains all the syntactic combination rules and structural words, as well as a test lexicon of about 350 content words. Each language *L* has a concrete syntax **Lang*L***, which can be used for parsing and linearization. Thus you can test the English resource syntax as follows:

```
> import alltenses/LangEng.gfo
```

Now you can try parsing sentences in the resource grammar:

```
Lang> p "this wine is good"
PhrUtt NoPConj (UttS (UseCl (TTAnt TPres ASimul) PPos
  (PredVP (DetCN (DetQuant this_Quant NumSg)(UseN wine_N))
    (UseComp (CompAP (PositA good_A)))))) NoVoc
```

The resulting tree is the internal representation in the library. As an application programmer, you will never need this representation; but if you start writing your own resource grammar, you will have to implement all these functions and many more, as described in Chapter 10.

If you are interested in testing the resource API functions themselves, you can import the module **Try*L***, which is a union of **Syntax*L***, **Lexicon*L***, and **Paradigms*L***. This is a resource module, which can be imported with the **-retain** option to enable the command **cc**:

```
> i -retain alltenses/TryEng.gfo
> cc -all mkUtt (mkCl this_NP (mkA "cool"))
this is cool
```

The top-level **Lang** grammars are exported as resource modules by means of the technique of **grammar reuse**, whose internal details are explained in Section C.4.17. The relations between **Lang**, **Syntax**, **Try**, and the other library modules are specified in Section 10.1.

Exercise 5-9. Construct some expressions and their translations by parsing and linearizing in the resource library:

- *is this wine good*
- *I (don't) like this wine, do you like this wine*
- *I want wine, I would like to have wine*
- *I know that this wine is bad*
- *can you give me wine*
- *give me some wine*

- *two apples and wine*
- *he says that this wine is good*
- *she asked which wine was the best*

Exercise 5-10.⁺ Extend the **Foods** grammar with new forms of expressions, corresponding to the examples of the previous exercise. First extend the abstract syntax, then implement it by using the resource grammar and a functor. You can also try to minimize the size of the abstract syntax by using free variation as explained in Section 2.11. For instance, *I would like to have X*, *give me X*, *can you give me X*, and *X* can be variant expressions for one and the same order.

Exercise 5-11.⁺ Design a small grammar that can be used for controlling an MP3 player. The grammar should be able to recognize commands such as *play this song*, with the following variations:

- objects: *song*, *artist*
- modifiers: *this*, *the next*, *the previous*
- verbs with complements: *play*, *remove*
- verbs without complements: *stop*, *pause*

The implementation goes in the following phases:

1. abstract syntax
2. functor and lexicon interface
3. lexicon instance for the first language
4. functor instantiation for the first language
5. lexicon instance for the second language
6. functor instantiation for the second language
7. ...

Exercise 5-12.* Records of resource categories give a natural way to represent families of words that belong together. One example is the complex we might call **Action**, which involves a transitive verb (*play*), a person performing the action (*player*), the action itself (*playing*), and the quality of being a possible object (*playable*). Define the type **Action** as a record of resource grammar categories. Also define a smart paradigm

```
mkAction : Str -> Action
```

which produces such records in the most regular way. Notice that, since this paradigm is not bound to one part of speech, it implements what is usually called **derivational morphology**, as opposed to inflectional morphology.

Semantic actions and conditions in abstract syntax

The abstract syntax of grammars shown until now is still no more expressive than the abstract syntax implicit in context-free grammars. Logically, they correspond to zeroth-order simple type theory, where all types have the form

$$C_1 \rightarrow \dots \rightarrow C_n \rightarrow C$$

corresponding to category skeletons of context-free rules, as defined in Section 2.5. In this chapter, the expressivity is increased, in particular, by strengthening the type system with the notions of dependent types and higher-order abstract syntax. This means that grammars can express more semantics than before.

Since dependent types are not widely used in GF applications and now always supported by the embedded grammar tools of Chapter 7, this chapter and the examples shown in it have a more theoretical nature than the other chapters of this book. We also presuppose more background knowledge about, in particular, symbolic logic. If the material feels hard to understand, it is good to know that later chapters of this book don't presuppose this chapter, except some passages in Chapter 8.

The topics included in this chapter are:

- GF as a logical framework
- dependent types
- selection restrictions
- polymorphism
- proof objects and proof-carrying documents
- variable binding and higher-order abstract syntax
- semantic definitions, paraphrasing, and run-time transfer

- built-in categories
- tree probabilities

6.1 GF as a logical framework

Type theory is the basis of many systems known as **logical frameworks**, which are used for representing mathematical theorems and their proofs on a computer. In fact, GF has a logical framework as its proper part: this part is the abstract syntax.

In a logical framework, the formalization of a mathematical theory is a set of type and function declarations. The following is an example of such a theory, represented as an **abstract module** in GF.

```
abstract Arithm = {
  cat
    Prop ;                      -- proposition
    Nat ;                      -- natural number
  fun
    Zero : Nat ;                -- 0
    Succ : Nat -> Nat ;         -- the successor of x
    Even : Nat -> Prop ;        -- x is even
    And  : Prop -> Prop -> Prop ; -- A and B
}
```

This example does not show any new type-theoretical constructs yet, but it could nevertheless be used as a part of a proof system for arithmetic and will be extended in later sections of this chapter.

Exercise 6-1. Give a concrete syntax of **Arithm**, preferably by using the resource library.

6.2 Dependent types

Dependent types in GF are inherited from the **constructive type theory** of Martin-Löf (Martin-Löf 1984, Nordström & al. 1990). They distinguish GF from most other grammar formalisms and functional programming languages.

A dependent type is a type that takes arguments from other types. For instance, **Nats** n , the type of natural numbers from 0 to n is a dependent type, where the argument n is of type **Nat**. Dependent types can be used for stating stronger **conditions of well-formedness** than ordinary types. A function typed as

```
minus : Nat -> Nat -> Nat
```

cannot really express the normal “minus” operation, because the second argument y can be greater than the first x , and **minus** x y should then

be negative. However, a function typed as

$$\text{minus} : (\mathbf{n} : \mathbf{Nat}) \rightarrow \mathbf{Nats} \ \mathbf{n} \rightarrow \mathbf{Nat}$$

does guarantee that the second argument is at most the size of the first one. Actually, the accuracy of the typing could be increased to

$$\text{minus} : (\mathbf{n} : \mathbf{Nat}) \rightarrow \mathbf{Nats} \ \mathbf{n} \rightarrow \mathbf{Nats} \ \mathbf{n}$$

because $x - y$ can never be greater than x .

Dependent types need some new notation, which actually just generalizes the notation used so far. Hence the form of judgements for **category declarations** is generalized to

$$\text{cat } C \ G$$

where G is a **context**. A context is a sequence of **hypotheses**, of the form

$$(x : T)$$

where x is a variable and T is a type. Hypotheses are written without any separator symbols, which means that a non-dependent category declaration $\text{cat } C$ is simply a declaration with an empty context.

As syntactic sugar,

- variables can share a type;

$$(x, y : T) \equiv (x : T) (y : T)$$

- a **wildcard** can be used for a variable not occurring in types later in the context;

$$(\_ : T) \equiv (x : T)$$

- if the variable does not occur later, it can be omitted altogether, and parentheses are not used;

$$T \equiv (\_ : T)$$

Of course, if T is a more complex expression than an identifier, it needs parentheses to be separated from the rest of the context.

A **basic type** is a category applied to all arguments specified by its context. Thus, in the general case, if

$$\text{cat } C (x_1 : T_1) \dots (x_n : T_n)$$

we can form the basic type

$$C a_1 \dots a_n$$

if the arguments are well typed with respect to the context, that is,

$$a_1 : T_1, \dots, a_n : T_n \{x_1 := a_1, \dots, x_{n-1} := a_{n-1}\}$$

Dependent function types are function types whose value type depends on the argument. The general form is

$$(x : A) \rightarrow B$$

where the variable x may occur in the type B . There is syntactic sugar analogous to contexts, and actually defined above in Section 4.1 where variables were used as a means to document the code and to eliminate repetitions of the argument type.

6.3 Selectional restrictions

A simple natural language example where dependent types are useful is a “smart house” system, which defines voice commands for household appliances. This example is borrowed from the Regulus Book (Rayner & al. 2006).

One who enters a smart house can use a spoken **Command** to dim lights, switch on the fan, etc. For **Devices** of each **Kind**, there is a set of **Actions** that can be performed on them; thus one can dim the lights but not the fan, for example. Thus there are constraints for which verbs can meaningfully combine with which kinds of nouns. These constraints are known as **selectional restrictions**. They can be easily expressed with dependent types: we just make the types **Device** and **Action** dependent on **Kind**. We express these dependencies in **cat** declarations that have a context with one argument type:

```
cat
  Command ;
  Kind ;
  Device Kind ;
  Action Kind ;
```

The crucial use of the dependencies is made in the rule for forming commands, expressed by a dependent function type:

```
fun Act :
  (k : Kind) -> Action k -> Device k -> Command ;
```

In other words: an action and a device can be combined into a command only if they are of the same kind k . If we have the functions

```
The : (k : Kind) -> Device k ; -- the light
Light, Fan : Kind ;
Dim : Action Light ;
```

we can form the syntax tree

```
Act Light Dim (The Light)
```

but we cannot form the trees

```
Act Light Dim (The Fan)
Act Fan   Dim (The Light)
Act Fan   Dim (The Fan)
```

Linearization rules are written as usual: the concrete syntax does not know if a category is a dependent type. In English, one could write as follows:

```
lincat Action = {s : Str} ;
lin Act _ act dev = {s = act.s ++ dev.s} ;
```

Notice that the argument for `Kind` does not appear in the linearization; it is good practice to make this clear by using a wild card for it, rather than a real variable. As we will show, the type checker can reconstruct the kind from the `dev` argument. This is one of the main uses of the technique of suppression introduced in Section 2.9.

Parsing with dependent types can be conceptually divided to two phases:

1. context-free parsing
2. filtering through type checker

The `parse` command performs both phases automatically:

```
> parse "dim the light"
Act Light Dim (The Light)
```

Context-free parsing alone would just return a tree with a metavariable (?) for the suppressed argument:

```
Act ? Dim (The Light)
```

When the type checker inspects a tree with metavariables, it forms a set of **constraints**, which are equations involving the metavariables. In this case, there are two constraints, resulting from the types of `Dim` and the argument of `The`:

```
? = Light, ? = Light
```

This set of constraints has a trivial solution, setting ? to `Light`.

But constraint solving can also fail, as is the case with incorrect trees:

```
> parse "dim the fan"
The parsing is successful but the type checking failed:
Couldn't match expected type Device Light
      against inferred type Device Fan
```

Here the context-free tree is

Act ? Dim (The Fan)

which yields the constraints

? = Light, ? = Fan

This leads to the derived constraint **Fan = Light**, which is unsolvable and produces the error message printed by GF.

Solving can also fail even though the constraints are not found to be unsolvable. This happens, for instance, when a function with non-dependent types suppresses an argument. In such a case, metavariables are left after type checking, and GF type checking itself has no clue about how to solve them. But the tree can of course be sent to further processing such as anaphora resolution, as mentioned in Section 2.10, or to interactive editing by the user, as described in Section 7.12.

Exercise 6-2. Write an abstract syntax module with the contents above and an appropriate English concrete syntax. Try out parsing and generation.

Exercise 6-3. Add some device kinds and actions to the grammar.

Exercise 6-4.* Define some rules involving agreement in abstract syntax by using dependent types—for instance, the number agreement in NP-VP predication in English. To this end, you need to introduce **Number** as a type in the abstract syntax and make **NP** and **VP** dependent on it.

Exercise 6-5.*+ In the GF Resource Grammar Library, there are several categories of verbs—**V**, **V2**, **V3**, **VS**, etc—and corresponding complementation rules for building verb phrases; see Section 5.15 and Appendix D. An alternative analysis, perhaps more elegant, would be to have just one category of verbs, but make it dependent on another category, say **VSub**, which is the type of verb subcategorizations. We would also need a type of complement lists, say **Comps**, dependent on **VSub**. With this machinery, just one complementation rule is needed, of type

(vs : VSub) -> V vs -> Comps vs -> VP

Work out the details of this idea, covering at least the four verb categories listed above.

6.4 Polymorphism

Sometimes an action can be performed on all kinds of devices. It would be possible to introduce separate **fun** constants for each kind-action pair, but this would be tedious. Instead, one can use **polymorphic** actions, i.e. actions that take a **Kind** as an argument and produce an

Action for that Kind:

```
fun SwitchOn, SwitchOff : (k : Kind) -> Action k ;
```

Functions that are not polymorphic are **monomorphic**. However, the dichotomy into monomorphism and full polymorphism is not always sufficient for good semantic modelling: very typically, some actions are defined for a proper subset of devices, for more than one. For instance, both doors and windows can be opened, whereas lights cannot. We will return to this problem by introducing the concept of **restricted polymorphism** in Section 6.8.

Exercise 6-6. The grammar *Shopping* in Section 4.8 permits the formation of phrases such as *this shirt is delicious* and *that fish is comfortable*. A way to prevent this would be to distinguish between clothing and food items. To avoid duplicated code for these items, the best way to go is by making the categories of *Item* and *Quality* dependent on *Kind*. The predicates *elegant*, *expensive*, and *Italian* can be made polymorphic. Rewrite the abstract syntax in Figure 27 in accordance with these ideas.

6.5 Dependent types in concrete syntax

The **functional fragment** of GF terms and types comprises function types, applications, lambda abstracts, constants, and variables. This fragment is the same in abstract and concrete syntax. In particular, dependent types are also available in concrete syntax. We have not made use of them yet, but we will now look at one example of how they can be used.

Those readers who are familiar with functional programming languages like ML and Haskell, may already have missed **polymorphic** functions. For instance, Haskell programmers have access to the functions

```
const :: a -> b -> a
const c _ = c

flip :: (a -> b -> c) -> b -> a -> c
flip f y x = f x y
```

which can be used for any given types *a*, *b*, and *c*.

The GF counterpart of polymorphic functions are **monomorphic** functions with explicit **type variables** — a technique that we already used in abstract syntax for modelling actions that can be performed on all kinds of devices. Thus in GF the above definitions can be written

```
oper const :(a,b : Type) -> a -> b -> a =
```

```

  \_,_,c,_ -> c ;

oper flip :
  (a,b,c : Type) -> (a -> b -> c) -> b -> a -> c =
    \_,_,_,f,x,y -> f y x ;

```

When the operations are used, the type checker requires them to be equipped with all their arguments. This may be a nuisance for a Haskell or ML programmer, who are used to implicit type arguments, but familiar to Java programmers using generics.

The GF Resource Grammar Library uses dependent types to implement **coordination**, that is, phrases of the form $X \dots X \ c \ X$ where c is a conjunction and X is one of the categories that supports coordination (see Section D.2.9). The generalization that is captured is that, whenever X has a variable feature P , it is passed to all conjuncts. The crucial operation is

```

conjunctTable : (P : Type) -> Conj ->
  {s1,s2 : P => Str} -> {s : P => Str} =
  \_,co,xs -> {s = \p => xs.s1 ! p ++ co.s ++ xs.s2 ! p}

```

where the list is represented by a record of two tables, where `s1` contains the initial elements and the last one is stored in `s2`. The conjunction is placed between these two. The library module **Coordination** makes a set of such operations available for programmers who need to handle cases not covered by the current resource grammars—for instance, when implementing the resource for new languages.

6.6 Proof objects

Perhaps the most radical idea in constructive type theory is the **Curry-Howard isomorphism**, also known as the **propositions as types principle**. Its earliest formulations were meant to give semantics to the logical systems of propositional and predicate calculus. In this section, we will consider a more elementary example, but also show how the notion of proof makes sense outside mathematics.

We use the already shown category of unary (also known as Peano-style) natural numbers:

```

cat Nat ;
fun Zero : Nat ;
fun Succ : Nat -> Nat ;

```

The **successor function** `Succ` generates an infinite sequence of natural numbers, beginning from `Zero`.

We then define what it means for a number x to be *less than* a

number y . Our definition is based on two axioms:

- For any y , **Zero** is less than **Succ** y .
- If x is less than y , then **Succ** x is less than **Succ** y .

The most straightforward way of expressing these axioms in type theory is with a dependent type **Less** x y , and two functions constructing its objects:

```
cat Less Nat Nat ;
fun LessZ : (y : Nat) -> Less Zero (Succ y) ;
fun LessS : (x,y : Nat) -> Less x y ->
    Less (Succ x) (Succ y) ;
```

Objects formed by **LessZ** and **LessS** are called **proof objects**: they establish the truth of certain mathematical propositions. For instance, the fact that 2 is less than 4 is encoded by the type

```
Less (Succ (Succ Zero)) (Succ (Succ (Succ (Succ Zero))))
```

which has the proof object

```
LessS (Succ Zero) (Succ (Succ (Succ Zero)))
  (LessS Zero (Succ (Succ Zero)) (LessZ (Succ Zero)))
```

Selectional expressions were our first example of how dependent types can be used to provide a **semantic control** of the well-formedness of expressions. Proof objects provide an even more powerful technique for expressing semantic conditions.

A simple example of the use of proof objects is the definition of well-formed *time spans*: a time span is expected to be from an earlier to a later time:

```
from 3 to 8
```

is thus well-formed, whereas

```
from 8 to 3
```

is not. The following rules for spans impose this condition by using the **Less** predicate:

```
cat Span ;
fun FromTo : (m,n : Nat) -> Less m n -> Span ;
```

Exercise 6-7. Write an abstract and concrete syntax with the concepts of this section, and experiment with it in GF.

Exercise 6-8.* Define the notions of “even” and “odd” by using proof objects. **Hint.** You need one function for proving that 0 is even, and

two other functions for propagating the properties from each number to its successor.

6.7 Proof-carrying documents

One possible application of proof objects is **proof-carrying documents**: to be semantically well-formed, the abstract syntax of a document must contain a proof of some property, although the proof is not shown in the concrete document. Think, for instance, of small documents describing flight connections:

To fly from Gothenburg to Prague, first take LH3043 to Frankfurt, then OK0537 to Prague.

The well-formedness of this text is partly expressible by a dependent type of flights, which indicates the depart and destination cities of each flight:

```
cat
  City ;
  Flight City City ;
fun
  Gothenburg, Frankfurt, Prague : City ;
  LH3043 : Flight Gothenburg Frankfurt ;
  OK0537 : Flight Frankfurt Prague ;
```

These rules are enough to reject texts saying *take OK0537 from Gothenburg to Prague*. However, there is a further condition involved: it must be possible to change from LH3043 to OK0537 in Frankfurt. This condition can be modelled as a proof object of the following category, expressing that it is possible to change from one flight to another:

```
cat IsPossible (x,y,z : City)(Flight x y)(Flight y z) ;
```

The context of this category is more complex than the ones seen so far, as its argument list has dependent types with growing dependencies from the left: three cities are needed (departure, intermediate destination, final destination), as well as two flights involving these cities in the appropriate manner.

By requiring a proof of type `IsPossible`, we can finally define the constructor that connects two flights, in a manner that is guaranteed to be possible:

```
fun Connect : (x,y,z : City) ->
  (u : Flight x y) -> (v : Flight y z) ->
    IsPossible x y z u v -> Flight x z ;
```

6.8 Restricted polymorphism

In the first version of the smart house grammar **Smart**, all Actions were either of

- monomorphic: defined for one Kind
- polymorphic: defined for all Kinds

To make this scale up better for new Kinds, we can refine this dichotomy to the more general notion of **restricted polymorphism**: a function can be defined for a subset of all Kinds, defined as a **class** of Kinds. The notion of class can be expressed in abstract syntax by using the Curry-Howard isomorphism as follows:

- a class is a **predicate** of Kinds — i.e. a type depending on Kind
- a Kind is in a class if there is a proof object of this type

Here is an example with switching and dimming. The classes are called **Switchable** and **Dimmable**.

```
cat
  Switchable Kind ;
  Dimmable   Kind ;
```

The inclusion of a Kind in a class is encoded by a proof object. Thus we have

```
fun
  switchable_light : Switchable Light ;
  switchable_fan   : Switchable Fan   ;
  dimmable_light   : Dimmable Light   ;
```

Now the actions of switching on and off can be defined for the switchable class, whereas dimming is defined for the dimmable class:

```
SwitchOn  : (k : Kind) -> Switchable k -> Action k ;
SwitchOff : (k : Kind) -> Switchable k -> Action k ;
Dim       : (k : Kind) -> Dimmable k   -> Action k ;
```

One advantage of restricted polymorphism is that classes for new actions, as well as proofs for new kinds, can be added incrementally. A pitfall with restricted polymorphism is that it may reflect expressions rather than their semantics, and hence work differently in different languages. For instance, the action of “opening” works in English for windows and doors, but in Finnish, also TV’s and radios can be “opened”, which simply means switching them on.

The proof objects for class inclusion (as well as for other semantic conditions) are usually suppressed in linearization. If they are *always* suppressed, it is tedious to write the linearization rules at all, as is

required by the completeness of a concrete syntax. The effort can be minimized in two ways:

- Use the empty record type (`{}`) as the linearization type of proof object categories; this type carries no information, and all linearization rules can be defined as empty tuples (`<>`). In fact, linearization rules that return the type `{}` are constructed automatically by the compiler, because this type has only one possible value.
- Keep the linearization types and rules for dummy objects in one module, which is inherited by all languages.

Exercise 6-9.* Write a new version of the **Smart** grammar with classes, and test it in GF.

Exercise 6-10.* Add some actions, kinds, and classes to the grammar. Try to port the grammar to a new language.

Exercise 6-11.* A better structure can be given for the code by treating classes as zero-place **fun**s of category **Class**, and making all proof objects have the category saying that a kind is an instance of a class:

```
cat Instance Class Kind ;
```

Actions can then be made dependent on just classes, and devices on kinds. Work out the details of this alternative, in particular, the **Act** function that forms commands.

6.9 Variable bindings and higher-order abstract syntax

Mathematical notation and programming languages have expressions that **bind** variables. For instance, a universally quantifier proposition

$$(\forall x)B(x)$$

consists of the **binding** $(\forall x)$ of the variable x , and the **body** $B(x)$, where the variable x can have **bound occurrences**.

Variable bindings appear in informal mathematical language as well, for instance,

for all x , x is equal to x

the function that for any numbers x and y returns the maximum of $x + y$ and xy

Let x be a natural number. Assume that x is even. Then $x + 3$ is odd.

In type theory, variable-binding expression forms can be formalized as functions that take functions as arguments. The universal quantifier is defined

```
fun All : (Ind -> Prop) -> Prop
```

where **Ind** is the type of individuals and **Prop**, the type of propositions. If we have, for instance, the equality predicate

```
fun Eq : Ind -> Ind -> Prop
```

we may form the tree

```
All (\x -> Eq x x)
```

which corresponds to the ordinary notation

$$(\forall x)(x = x)$$

An abstract syntax where trees have functions as arguments, as in the two examples above, has turned out to be precisely the right thing for the semantics and computer implementation of variable-binding expressions. The advantage lies in the fact that only one variable-binding expression form is needed, the lambda abstract $\backslash x \rightarrow b$. All other bindings can be reduced to it. This makes it easier to implement mathematical theories and reason about them, since variable binding is tricky to implement and to reason about. The idea of using functions as arguments of syntactic constructors is known as **higher-order abstract syntax**.

The question now arises: how to define linearization rules for variable-binding expressions? Let us first consider universal quantification,

```
fun All : (Ind -> Prop) -> Prop
```

In GF, we can write

```
lin All B = {s = "(" ++ "\forall" ++ B.$0 ++ ")" ++ B.s}
```

to obtain the form shown above. This linearization rule brings in a new GF concept: the **bound variable label** $\$0$, which selects bound variable symbols in higher-order abstract syntax. The general rule is that, if an argument type of a function is itself a function type $A \rightarrow C$, the linearization type of this argument is the linearization type of C together with a new field $\$0 : \mathbf{Str}$. In the linearization rule for **All**, the argument B thus has the linearization type

```
{\$0 : Str ; s : Str},
```

since the linearization type of **Prop** is

```
{s : Str}
```

In other words, the linearization of a function consists of a linearization of the body together with a field for a linearization of the bound variable. Those familiar with type theory or lambda calculus should notice

that GF requires trees to be in **eta-expanded** form in order for this to make sense: for any function of type

$A \rightarrow B$

an eta-expanded syntax tree has the form of a lambda abstract

$\backslash x \rightarrow b$

where $b : B$ under the assumption $x : A$. It is in this form that an expression can be analysed as having a bound variable and a body, which can be put into a linearization record.

Given the linearization rule

$\text{lin Eq } a \ b = \{s = "(" ++ a.s ++ "=" ++ b.s ++ ")\}$

the linearization of the lambda abstract

$\backslash x \rightarrow \text{Eq } x \ x$

is the record

$\{\$0 = "x", s = "(x = x)"\}$

Thus the linearization of the formula (abstract syntax tree)

$\text{All } (\backslash x \rightarrow \text{Eq } x \ x)$

becomes

$\{s = "(\forall x) (x = x)"\}$

But how did we get the linearization of the variable x into the string " x "? GF grammars have no rules for this: it is just hard-wired in GF that variable symbols are linearized into the same strings that represent them in the print-out of the abstract syntax.

Exercise 6-12.* Write an abstract syntax of the whole **predicate calculus**, with the **connectives** “and”, “or”, “implies”, and “not”, and the **quantifiers** “exists” and “for all”. Use higher-order functions to guarantee that unbounded variables do not occur.

Exercise 6-13.* Write a concrete syntax for your favourite notation of predicate calculus. You can use $\text{\LaTeX}$ as target language if you want nice output. You can also try producing boolean expressions of some programming language. Use as many parenthesis as you need to guarantee non-ambiguity. You can also minimize the number of parentheses by using a precedence level parameter, as described in Section 8.1.

Exercise 6-14.* Write an English concrete syntax for predicate calculus, to translate between formulas and natural language. Even better, use a functor over the resource grammar **Syntax** interface to write a generic natural-language grammar for predicate calculus. You can

thereby use the `Symbolic` library to enable the use of variables and other mathematical symbols inside natural language text; see Section D.4.3.

6.10 Anaphoric expressions

Anaphoric expressions such as pronouns (*it*, *he*, *she*) can be used for referring to objects without naming them. Sometimes the object does have a name. For instance, in *if John walks he talks*, the pronoun *he* can be replaced by *John*. But sometimes this is not the case. In

if a man walks he talks

replacing *he* with *a man* would change the meaning, because the sentence *if a man walks a man talks* could be about two different men. One possible replacement is *the man*, but this phrase, a **definite description**, is no less anaphoric than the pronoun *he*!

A widely applied idea to explain what happens in anaphora is to use bound variables. Ranta (1994) develops this idea in the context of constructive type theory. As shown in Ranta (2004b), the idea can be implemented in GF. Without going to the full details here, let us give a summary of its main ingredients:

1. Anaphoric expressions stand for individuals that are present in the abstract syntax but suppressed in the concrete syntax.
2. The individuals can be constants or variables.
3. The hidden arguments can be found by solving metavariables.

Point (1) is the same as in Section 2.10. Generalizing from what was introduced there, we consider pronouns for individuals of different kinds. We also distinguish between explicit names (**Name**) and hidden referents (**Ref**). The reason is that we don't want to include bare referents, which are typically variables, in natural language: a **Ref** can only be used as a name via an anaphoric expression.

```
cat Kind ; Name Kind ; Ref Kind ;
```

The following functions build pronouns and definite descriptions:

```
fun Pron, Def : (k : Kind) -> Ref k -> Name k ;
```

Their linearizations use the `Kind` argument either to determine the gender (**Pron**) or as the noun following the definite article (**Def**)

```
lin Pron k _ =
  case k.g of {Masc => "he" ; Fem => "she" ; _ => "it"} ;
lin Def k _ = "the" ++ k.s ;
```

The gender of pronouns is of course language-dependent. Thus English *it* may in French become *il* or *elle*, which in turn can also correspond to English *he* and *she*, respectively. To translate a pronoun, one must restore its Kind argument to find the gender in the target language.

The hidden individuals in the category **Ref** are often called **discourse referents**. The simplest kind of a discourse referent is a bound variable. Consider a function for universal quantification defined by

```
fun Univ : (k : Kind) -> (Ref k -> Prop) -> Prop ;
lin Univ k p =
  {s = "it holds for every" ++ k.s ++ p.$0 ++ "that" ++ p.s}
```

This could give, for instance,

it holds for every number x that x is even or odd

A variable-free expression for the same proposition is

it holds for every number that it is even or odd

It can be obtained by just omitting the variable:

```
lin Univ k p =
  {s = "it holds for every" ++ k.s ++ "that" ++ p.s}
```

With the above definition of (**Pron**), and a suitable predicate **Eo0** for “even or odd”, this sentence has the abstract syntax tree

```
Univ Number (\x -> Eo0 (Pron Number x))
```

Now, what about the original example, *if a man walks he talks*? The solution in Ranta (1994) is the **progressive implication** of constructive type theory. It generalizes ordinary implication by letting the succedent depend on a *proof* of the antecedent (cf. Section 6.6):

```
fun If : (p : Prop) -> (Proof p -> Prop) -> Prop ;
lin p q = {s = "if" ++ p.s ++ q.s}
```

Furthermore in constructive type theory, existential propositions have proofs from which one can extract a **witness**—an individual for which the existing property holds. The sentence *a man walks* expresses an existential proposition, whose witness is an individual of type **man**. If we, for simplicity, restrict ourselves to this form of existentials,

```
fun Exist : (k : Kind) -> Verb k -> Prop
lin Exist k v = {s = "a" ++ k.s ++ v.s}
```

the witness function is

```
fun wit : (k : Kind) -> (v : Verb k) ->
  Proof (Exist k v) -> Ref k
```

We of course need predication for noun phrases and verb phrases,

```
fun Pred : (k : Kind) -> Name k -> Verb k -> Prop
```

Assuming the verb phrases **Walk** and **Talk**, we can now build an abstract syntax tree for our original example:

```
If (Exist Man Walk)
  (\x -> Pred (Pron Man (wit Man Walk x)) Talk)
```

Exercise 6-15. Build an abstract and concrete syntax with the rules given in this section and experiment with the parsing and generation of sentences with anaphoric expressions. Verify, in particular, that you can parse and type check (filling all metavariables) *if a man walks the man talks* but not *if a man walks the woman talks*.

Exercise 6-16.* Generalize the grammar of this section to a multilingual one, possibly by using the resource grammar library. By extending the lexicon of kinds (nouns), construct examples in which one and the same source language pronoun is translated in different ways depending of its kind.

Exercise 6-17.* Definite descriptions can carry more information than just the kind. For instance, *the man who walks* is a valid expression in the context of *a man walks*. Define the abstract syntax and linearization of this form of expressions (i.e. *the Kind who Verb*), which require both a witness and a proof that the witness satisfies the relative clause. Verify in GF that you can parse and type check *if a man walks the man who walks talks* but not *if a man walks the man who talks walks*.

6.11 Semantic definitions

Just like any functional programming language, abstract syntax in GF has declarations of functions, telling what the type of a function is. But we have not yet shown how to **compute** these functions: all we can do is provide them with arguments and linearize the resulting terms. Since our main interest is the well-formedness of expressions, this has not yet bothered us very much. However, as we will see in the next section, computation does play a role even in the well-formedness of expressions when dependent types are present.

GF has a form of judgement for **semantic definitions**, marked by the key word **def**. At its simplest, it is just the definition of one constant, e.g.

```
fun one : Nat ;
def one = Succ Zero ;
```

We follow here the convention of Haskell and use small initials for defined functions (such as **one**) and capitals for constructors (such as

`Succ` and `Zero`); this convention is not compulsory in GF, but it adds clarity.

We can also define a function with arguments,

```
fun twice : Nat -> Nat ;
def twice x = plus x x ;
```

This is still a special case of the most general notion of definition, that of a group of **pattern equations**. This is how we compute the sum of two numbers by pattern matching on the second argument:

```
fun plus : Nat -> Nat -> Nat ;
def
  plus x Zero = x ;
  plus x (Succ y) = Succ (plus x y) ;
```

To compute a term is, as in functional programming languages, simply to follow a chain of computation steps until no definition can be applied. For instance, we compute

```
plus one one ↓
plus (Succ Zero) (Succ Zero) ↓
Succ (plus (Succ Zero) Zero) ↓
Succ (Succ Zero)
```

Computation in GF is performed with the `put_tree` command and the `compute` option, e.g.

```
> parse -tr "1 + 1" | put_tree -compute -tr | linearize
plus one one
Succ (Succ Zero)
s(s(0))
```

where we use the trace option `-tr` to see the intermediate results.

The `def` definitions of a grammar induce a notion of **definitional equality** among trees: two trees are definitionally equal if they compute into the same tree. Thus, trivially, all trees in a chain of computation (such as the one above) are definitionally equal to each other. In general, there can be infinitely many trees definitionally equal to each other.

In addition to computation, `def` definitions can be used in **paraphrasing** trees with other trees. Paraphrasing applies definitions by one step both backwards and forwards, in all parts of the tree separately:

```
> pt -paraphrase plus (Succ Zero) one
plus (Succ Zero) one
plus (Succ Zero) (Succ Zero)
plus one one
```

```
plus one (Succ Zero)
```

More remote paraphrases can be generated by piping this paraphrasing command into another one, which takes all paraphrases as starting points of new ones.

When pattern matching is performed with **def** equations, it is crucial to distinguish between **constructors** and other functions (cf. Section 3.2 on pattern matching in concrete syntax). GF has a judgement form **data** to tell that a category has certain functions as constructors:

```
data Nat = Succ | Zero ;
```

Unlike in Haskell and ML, new constructors can be added to a type with new **data** judgements. The type signatures of constructors are given separately, in ordinary **fun** judgements. One can also write directly

```
data Succ : Nat -> Nat ;
```

which is syntactic sugar for the pair of judgements

```
fun Succ : Nat -> Nat ;
data Nat = Succ ;
```

Since constructors and variables are syntactically similar in GF, **data** declarations are essential when resolving patterns in equations. If we did not mark **Zero** as **data**, it would be interpreted as a variable in the definition

```
fun isZero : Nat -> Bool ;
def isZero Zero = True ;
def isZero _ = False ;
```

The function **isZero** would then return **True** for all arguments, because the pattern **Zero** would match all values! This is a common pitfall in GF.

Exercise 6-18.*+ Implement an interpreter of a small functional programming language with natural numbers, lists, pairs, lambda abstracts, etc. Use higher-order abstract syntax with semantic definitions. As concrete syntax, use the notation of your favourite functional programming language.

6.12 Intensional and extensional equality

An important property of definitional equality is that it is **extensional**, i.e. has to do with the sameness of semantic value. Linearization, on the other hand, is an **intensional** operation, i.e. respects the syntactic identity of expressions. This means that **def** definitions are *not* evaluated as linearization steps. Intensionality is a crucial property of linearization,

since we want to use it for things like tracing a chain of evaluation. For instance, each of the steps of the computation shown in the previous section has a different linearization into standard arithmetic notation:

```
1 + 1
s(0) + s(0)
s(s(0) + 0)
s(s(0))
```

In most programming languages, the operations that can be performed on expressions are extensional, i.e. give equal values to equal arguments. But GF has both extensional and intensional operations. Type checking is extensional: in the type theory with dependent types, types may depend on terms, and types depending on definitionally equal terms are equal types. For instance,

```
Less Zero one
Less Zero (Succ Zero))
```

are equal types. Hence, any tree that type checks as a proof that 0 is less than 1 also type checks as a proof that 0 is less than the successor of 0.

In the smart house example (Section 6.3), we defined dimming to be an action applicable to lights:

```
fun Dim : Action Light
```

Now, suppose we introduce the lamps as a device equal to lights:

```
fun Lamp : Kind
def Lamp = Light
```

The equality implies that `Dim` also works as `Action Lamp`.

Equalities in abstract syntax can often be expressed alternatively by the use of free variation in concrete syntax (Section 2.11). For instance,

```
lin Light = mkN "lamp" | mkN "light"
```

The advantage of using abstract syntax is that the trees are distinct, and translation can be controlled to produce literal counterparts. Thus *light* in will always give *Licht* in German and *lamp* will always give *Lampe*. On the other hand, this treatment is not so natural in cases where the languages have different sets of synonyms to express things.

6.13 Semantic actions and run-time transfer

The term **transfer** is used in machine translation for structure-changing operations, which are often needed when going from one language to another. In GF, transfer can often be performed at compile time,

as we have seen in Section 5.13. However, this is not always possible, because transfer may depend on subtrees that are known only at run time. Fortunately, transfer functions can be defined as **fun** functions with **def** definitions.

An example of transfer is **aggregation**, which means the sharing of common constituents. For instance, *John runs or John walks* can be transformed to *John runs or walks* by sharing the common subject. Aggregation is used in many systems of **natural language generation** to improve style and to eliminate ambiguities. The abstract syntax in Figure 46 defines a minimal aggregation system, where both noun phrases (as above) and verb phrases (as in *John or Mary runs*) can be shared.

To use a transfer function at run time, the **put_tree** command can be used with the flag **-transfer** indicating the transfer function. The following command shows traces of each phase in the aggregation pipeline, assuming an appropriate concrete syntax of the **Aggregation** grammar:

```
> p -tr "John runs or Mary runs" |
    pt -tr -transfer=aggr | l
ConjS (PredVP John Run) (PredVP Mary Run)
PredVP (ConjNP John Mary) Run
John or Mary runs
```

While the code in Figure 46 shows that aggregation and other run-time transfer functions are *possible* to define in GF, it also shows how primitive the **def** definitions are as a programming language. For example, booleans, if-then-else expressions, and equality predicates have to be defined by the programmer, and only the applicative functional notation is available. A more productive way of writing transfer functions may thus be via the use of a host language such as Haskell with embedded grammars. We return to this topic in Section 7.8.

So, when is run-time transfer needed? The general answer follows from a fundamental property of GF: that linearization is **compositional**. Compositionality means that the linearization of any tree is a function of the *linearizations* of its subtrees. In other words,

$$(f\ x_1 \dots x_n)^* = f^*\ x_1^* \dots x_n^*$$

where t^* is the linearization of a tree t , and f^* is the linearization function of a function f . This means that linearization cannot inspect the structure of the subtrees themselves, but only of their linearizations. Hence, whenever an operation is not compositional, it cannot be encoded as linearization in GF, and hence needs run-time transfer. Aggregation is an example of such an operation, even though *finite* in-


```

abstract Aggregation = {
  cat S ; NP ; VP ;
  data
    PredVP : NP -> VP -> S ;
    ConjS   : S -> S -> S ;
    ConjVP  : VP -> VP -> VP ;
    ConjNP  : NP -> NP -> NP ;
    Run, Walk : VP ;
    John, Mary : NP ;

  fun aggr : S -> S ; -- main aggregation function
  def aggr (ConjS (PredVP x X) (PredVP y Y)) =
    ifS (eqNP x y)
      (PredVP x (ConjVP X Y))
      (ifS (eqVP X Y)
        (PredVP (ConjNP x y) X)
        (ConjS (PredVP x X) (PredVP y Y))) ;
  fun ifS : Bool -> S -> S -> S ; -- if then else
  def
    ifS True  x _ = x ;
    ifS False _ y = y ;
  fun eqNP : NP -> NP -> Bool ; -- x == y
  def
    eqNP John John = True ;
    eqNP Mary Mary = True ;
    eqNP _ _ = False ;
  fun eqVP : VP -> VP -> Bool ; -- X == Y
  def
    eqVP Run Run = True ;
    eqVP Walk Walk = True ;
    eqVP _ _ = False ;
  cat Bool ; data True, False : Bool ;
}

```

FIGURE 46 A minimal system for aggregation.

stances of aggregation (i.e. with a limited size of conjuncts), as in the example above, could in theory be encoded by parameters. Translations between number systems of different bases are a very clear example of non-compositional operations (cf. exercise below).

Exercise 6-19.* Write a concrete syntax of **Aggregation** and test the grammar in the GF shell. Notice that we have used *or* as the only conjunction to simplify number agreement for noun phrases; however, there is no extra problem to use *and* if linearization is defined by using the resource grammar.

Exercise 6-20.* English phrases of the form *A and B or C* are ambiguous. Generalize the **Aggregation** grammar by using the conjunction word, either *and* or *or*, as an extra argument of conjunctions, and generalize aggregation accordingly. Give an example where aggregation actually resolves an ambiguity.

Exercise 6-21.* Generalize the **aggr** function to arbitrary iterations, so that e.g. *John walks or John runs or John walks* becomes *John walks or runs or walks*.

Exercise 6-22.* Languages sometimes put particular restrictions to the co-occurrences of words. For example, in the **Food** grammar (Figure 12 in Section 2.6), the word *very* in *very delicious* might be superfluous and clumsy in some languages. But it can be removed by a suitable transfer function, without changing the grammar. Define this function, operating on the top level **Comments**, and test it in the translation from English to Italian under the assumption that Italian doesn't permit *molto* in combination with *delizioso*.

Exercise 6-23.*+ The GF web page contains a multilingual grammar of the numeral systems of 88 languages ([GF/examples/numerals](#)). This is possible because the numeral systems in all those languages are decimal (base 10), at least for numbers above a certain size. If we want to translate between binary numbers (base 2) and decimals, we cannot use a common abstract syntax with linearizations, but need a transfer function. Now, such a function would be tedious to write in GF, because of its limited programming constructs—in particular, the lack of built-in arithmetic. However, the same idea is illustrated by the easier task of defining a translator between binary and unary numbers (base 1, that is, the type **Nat** with the constructors **Zero** and **Succ**). Implement this in GF, by means of two separate categories **Nat** and **Bin**, and the transfer functions **bin2nat** and **nat2bin**. Test your conversions by random-generating numbers and converting them back and forth.

6.14 Predefined categories

In addition to the categories defined by `cat` judgements, GF has three built-in abstract syntax categories:

- `Int` of integers, e.g. 987031434, -6
- `Float` of floating-point numbers, e.g. 7.21, -6.0
- `String` of string literals, e.g. "foo"

The syntax trees of these types are **literals**, as shown in the examples. Such literals are the only objects of the built-in types: grammars are not allowed to declare `fun` functions with these types as value types. The literals can be of any length, and spaces in string literals count as real space characters.

The linearization type of every built-in category is the record type `{s : Str}`. Thus, for instance, the integer literal 5457 is linearized as

```
{s = "5457"}
```

The string literal "hello world" is linearized as

```
{s = "\"hello world\""}
```

where, as customary, literal quotes inside quotes are escaped by backslashes.

6.15 Probabilistic GF grammars

In **probabilistic context-free grammars**, each rule has an associated probability. The probability is a real number between 0 and 1, and the sum of the probabilities of rules with the same value category must sum up to 1. This notion has a natural generalization to **probabilistic GF grammars**, obtained by assigning probabilities to the `fun` functions of an abstract syntax.

The GF language has no notation for probabilities, but the GF shell supports the use of **probability configuration files**. These files have a simple syntax: each line consists of a function name and a floating-point number, separated by spaces. For instance, the `Food` grammar (Figure 12 in Section 2.6) can be given the following probability configuration file:

```
This 0.8
Mod 0.2
Wine 0.0
Fresh 0.4
```

As this example shows, probability configuration files can be partial: the remaining probability mass is divided evenly between the remain-

ing functions. Thus, for example, since **This** and **That** are the two constructors of **Item**, **That** receives the probability 0.2.

The most obvious place to use probabilities is in random generation. This is done by using the flag `-probs`, whose value is the name of a configuration file. If the four lines above are written in the file `Food.probs`, we can write

```
> generate_random -probs=Food.probs
```

A derived use is in the translation quiz (Section 2.7). The syntax is, naturally,

```
> translation_quiz -from=Eng -to=Ita -probs=Food.probs
```

In addition to trees with metavariables (Section 2.10), this gives a method for language learners to restrict the quiz to a subset of the whole language.

Starting with the probabilities of functions, we can define **tree probability** as the product of all functions used for constructing an abstract syntax tree. When many trees are given, they can be **ranked** by the command `rank_trees = rt`, with a flag for the probability configuration file. A typical use of ranking is in the **disambiguation** of parsing results. Thus

```
> parse "STRING" | rt -probs=foo.probs | pt -number=5
```

shows just the five best trees, in the order of ranking. Another use is the ranking of paraphrases (Section 6.11). For instance, a natural language generation system may use paraphrasing with language-dependent tree ranking to select the best expressions in each target language.

When using probabilities, an obvious question is where they come from. In typical computational linguistics tasks, they are extracted from frequencies in a natural collection of examples. In an interactive GF system, such as a web service (cf. Section 7.10), the actual user inputs can be recorded and used for building such collections. But even without the use of real frequencies, probability configurations files can be used as a cheap way to customize grammars for tasks such as the generation of exercises.

Exercise 6-24. Generate 100 random trees in the Food grammar and count the frequencies of **This** and **That**, to verify that the generator obeys the probabilities.

Part II

Larger Grammars and Applications

Embedded grammars and code generation

Embedded grammars are GF grammars that are used as parts of programs written in other languages, such as Haskell, Java, and JavaScript. These other languages, in which the main part of the programs are written, are called **host languages**. Embedded grammars make it possible to build systems where grammar-based processing is combined with components written in a host language.

In addition to grammars embedded in general-purpose host languages, interoperability may require grammars in special formats—in particular, language models for speech recognizers. It is important that such formats can be automatically generated from the same grammar as the other processing components, to guarantee consistency between the components and increase productivity by reducing manual work.

This chapter will cover the following issues:

- PGF, a portable format for multilingual GF grammars
- host-language API's for PGF
- manipulation of abstract syntax trees in the host language
- stand-alone translation programs
- question-answering systems
- multilingual syntax editors
- web services
- mobile phone applications
- language models for speech recognition
- multimodal dialogue systems

7.1 The portable grammar format

GF grammars are compiled to a low-level binary format called **PGF**, **Portable Grammar Format**. The GF shell performs the compilation

purely in memory, but PGF files can also be saved for later use by running GF as a **batch compiler**. This is done by calling `gf` with the option `-make` and a list of grammar files:

```
$ gf -make CONCRETE1.gf ... CONCRETEn.gf
```

The CONCRETE grammars must have the same abstract syntax ABS. The result of the command is a file `ABS.pgf`, which is a binary file containing a pre-compiled and optimized multilingual grammar as defined by the CONCRETE grammars.

Files in the `.pgf` format can also be imported in the GF system, which recognizes the suffix `.pgf` automatically. PGF is, in fact, the recommended format in which final grammar products are distributed, because they are stripped from superfluous information and can be started and applied much faster than sets of separate modules. As they don't require the GF grammar compiler (the program `gf`), they can work on a wider variety of platforms, for example, on mobile phones.

The use of GF as a batch compiler and the associated options are explained in Section E.2.

7.2 The embedded interpreter and its API

A PGF interpreter is a kind of a miniature GF system, which can parse, linearize, random-generate, and type check with grammars. But it can perform only a subset of the actions of the GF system. For instance, it cannot compile source grammars into the PGF format; the compiler is the most heavy-weight component of the GF system, and should not be carried around in end-user applications. Since PGF is much simpler than source GF, building an interpreter is relatively easy. A full-scale interpreter currently (Summer 2010) exists in Haskell, and a one without dependent types in Java. These interpreters take `.pgf` files as input. There is also an interpreter in JavaScript. It doesn't read `.pgf`, but JavaScript files, which can be generated by `gf -make` from both `.gf` and `.pgf` files, as shown in Section 7.11.

Application programmers usually don't need to read or modify the embedded GF interpreter. They only need to access it via its API. We will now look into the API's available in Haskell, Java, and JavaScript, and show some applications that use them.

7.3 Embedded GF applications in Haskell

Haskell is a typed functional programming language closely connected with GF in many ways. The creators of GF have been involved in teaching and developing Haskell. Thus Haskell has been an inspiration for GF, and GF actually borrows many of its constructs and notations

```

readPGF :: FilePath -> IO PGF

linearize :: PGF -> Language -> Tree -> String
parse :: PGF -> Language -> Category -> String -> [Tree]

linearizeAll      :: PGF -> Tree -> [String]
linearizeAllLang :: PGF -> Tree -> [(Language,String)]

parseAll :: PGF -> Category -> String -> [[Tree]]
parseAllLang ::
  PGF -> Category -> String -> [(Language,[Tree])]

languages :: PGF -> [Language]
categories :: PGF -> [Category]
startCat   :: PGF -> Category

```

FIGURE 47 A part of the API of the embedded PGF interpreter in Haskell.

from Haskell. Haskell is also the implementation language of the GF grammar compiler.

Unlike GF, Haskell is a general-purpose language, which has compilers all the way down to native machine code. However, language processing is one of the strongest applications of Haskell—traditionally, programming language processing in compilers, but more recently also natural language processing. An introductory book accessible to those with a background in computational linguistic is Doets and van Eijck (2006). The most recent Haskell text, also manifesting the growing general interest to Haskell, is Sullivan & al. (2008).

We hope that even readers unfamiliar with Haskell will be able to follow the discussion below. Haskell is a concise language, notationally close to both GF and to mathematical symbolism. In particular, recursive types of syntax trees and pattern matching over them are very similar in Haskell and GF. In a language like Java, these constructs require a complex encoding in terms of advanced concepts such as abstract classes and visitor patterns.

7.4 The module PGF

The main functions of the embedded Haskell API are given in Figure 47. This module, named `PGF`, is the only GF-related module that needs to be imported in most applications.

The first function, `readPGF`, is for reading a PGF grammar from a file. It can be used in any IO-enabled Haskell function; a typical

```

module Main where

import PGF
import System (getArgs)

main :: IO ()
main = do
    file:_ <- getArgs
    gr      <- readPGF file
    interact (translate gr)

translate :: PGF -> String -> String
translate gr s = case parseAllLang gr (startCat gr) s of
    (lg,t:_):_ -> unlines
        [linearize gr l t | l <- languages gr, l /= lg]
    _ -> "NO PARSE"

```

FIGURE 48 A stand-alone multilingual translator in Haskell.

example of its usage is shown in Figure 48. The next two functions, `linearize` and `parse`, perform linearization and parsing with given PGF grammars and languages. The `All` variants of these functions use all languages simultaneously; of these, the `AllLang` variants moreover show the language names attached to each string or tree. The last three functions extract information from the PGF grammar: what languages and categories there are, and what is the start category.

7.5 A stand-alone translator

Let us first build a stand-alone translator, which can translate in any multilingual grammar between any languages in the grammar. The entire code for this translator is the file `Translator.hs` shown in Figure 48.

To run the translator, we first compile it by the Haskell compiler GHC (Glasgow Haskell Compiler, freely available from <http://haskell.org/ghc/>). We use the `-o` flag to produce an executable named `trans`.

```
$ ghc --make -o trans Translator.hs
```

Then produce a PGF file. For instance, the `Food` grammar set can be compiled into PGF as follows:

```
$ gf -make FoodEng.gf FoodIta.gf
```

This produces the file `Food.pgf`.

The Haskell library function `interact` makes the `trans` program work like a Unix filter, which reads from standard input and writes to standard output. Therefore it can be a part of a pipe and read and write files. The simplest way to translate is to `echo` input to the program:

```
$ echo "this wine is delicious" | ./trans Food.pgf
questo vino è delizioso
```

The result is shown in all languages except the input language, as is defined on the most complex lines of the translator program:

```
(lg,t:_) :-> unlines
  [linearize gr l t | l <- languages gr, l /= lg]
```

The pattern `(lg,t:_) :->` matches a list of parse results that has at least one element, and returns from it the language `lg` and the first parse tree `t`. It returns, on separate lines (using `unlines`), the list of linearizations of `t` in each language `l` that is not equal (`/=`) to the source language `lg`.

Exercise 7-1. Compile and test the translator for the Food grammar and some other grammars.

7.6 A translator loop

If the user wants to translate many expressions in a sequence, it is cumbersome to have to start the translator over and over again, because reading the grammar from a PGF file always takes some time. The translator of the previous section is easy to modify to enable this: just change `interact` in the main function to `loop`, which is not a standard Haskell function, but defined as follows:

```
loop :: (String -> String) -> IO ()
loop trans = do
  s <- getLine
  if s == "quit" then putStrLn "bye" else do
    putStrLn $ trans s
    loop trans
```

The loop keeps on translating line by line until the input line is `quit`.

7.7 A question-answer system

The next application is also technically a translator, but it adds a **transfer** component to the grammar. Transfer is a function that takes the input syntax tree into some other syntax tree, which is then linearized and shown back to the user. The transfer function we are going

to use is one that computes a question into an answer. The program accepts simple questions about arithmetic and answers “yes” or “no” in the language in which the question was made:

```
is 123 prime ?
No.
77 est impair ?
Oui.
```

Figure 49 shows an abstract syntax of the query language. It has categories for questions and answers. Answers are just *yes* or *no*, and queries are just about integers and their properties of being *even*, *odd*, and *prime*.

The main change needed to the pure translator is an extra argument in the type of `translate`: a transfer function of type `Tree -> Tree`,

```
translate :: (Tree -> Tree) -> PGF -> String -> String
```

You can think of ordinary translation as a special case where transfer is the identity function (`id` in Haskell).

Also the behaviour of returning the reply in different languages should be changed so that the reply is returned in the *same* language. Here is the complete definition of `translate` in the new form.

```
translate trf gr =
  case parseAllLang gr (startCat gr) s of
    (lg,t:_):_ -> linearize gr lg (trf t)
    _ -> "NO PARSE"
```

To complete the system, we have to define the transfer function. So, how can we write a function from from abstract syntax trees to abstract syntax trees? This will be the topic of next section.

Exercise 7-2. Write a couple of concrete syntaxes of *Query*, either string-based or by using the resource grammar.

7.8 Exporting GF datatypes

The PGF API does not make the constructors of the type `Tree` available for users. Even if it did, it would be quite complicated to use this type, and programmers might mistakenly produce trees that are ill-typed in GF and could therefore not be linearized. The solution to this problem is to export GF categories as first-class datatypes in the host language. In Haskell, the idea is simply to convert GF’s tree constructors (the `fun` functions) into data constructors (branches in `data` definitions) in Haskell. There is enough resemblance between GF and Haskell to make this possible in most cases; only dependent types and higher-order

abstract syntax are not supported. A similar conversion is possible in Java and C as well, but it is less direct.

What we do is translate every category in an abstract syntax in GF into a Haskell datatype. The functions producing a value of that category are treated as constructors. For example the category of natural numbers defined in the previous chapter is converted as follows:

<code>cat Nat ;</code>		<code>data GNat =</code>
<code>fun Zero : Nat ;</code>	$\implies$	<code>GZero</code>
<code>fun Succ : Nat -> Nat ;</code>		<code> GZero GNat</code>

All type and constructor names are prefixed with a `G` to prevent clashes with standard Haskell functions and keywords and to guarantee that all these names have a capital initial, which is required by Haskell but not by GF.

A more complex conversion is shown in Figure 49. It covers a system of several types, as defined in the GF source file `Query.gf`; in general, such types can be mutually recursive. The translation is obtained by GF by using the batch compiler with the `-output-format` flag:

```
$ gf -make -output-format=haskell Query.gf
```

The result is a file named `Query.hs`, containing a module named `Query`.

Now it is possible to define Haskell functions on abstract syntax trees by using the Haskell datatypes in Figure 49. Haskell's type checker guarantees that the functions are well-typed also with respect to GF. A question-to-answer function for this language is shown in Figure 50. The main transfer function, which we apply to convert input to output, is `answer`.

The only problem is the *type* of the `answer` function: the parsing and linearization methods of the PGF API work with `Trees` and not with `GQuestions` and `GAnswers`. Fortunately the Haskell translation of GF takes care of translating between the type `Tree` and the generated datatypes. This is done by using a class with the required translation methods:

```
class Gf a where
  gf :: a -> Tree
  fg :: Tree -> a
```

The Haskell code generator also generates this class and its instances as part of the Haskell file shown in Figure 49. Needless to say, the generated file `Query.hs` is a file that a programmer never needs to look into, let alone change: it is enough to know that it contains a systematic encoding and decoding between an abstract syntax and Haskell datatypes, where

Source file Query.gf

```

abstract Query = {
  flags startcat=Question ;
  cat
    Answer ; Question ; Object ;
  fun
    Even   : Object -> Question ;
    Odd    : Object -> Question ;
    Prime  : Object -> Question ;
    Number : Int -> Object ;
    Yes    : Answer ;
    No     : Answer ;
}

```

Generated file Query.hs

```

module Query where

data GAnswer = GYes | GNo
data GObject = GNumber GInt
data GQuestion =
  GPrime GObject | GOdd GObject | GEven GObject

instance Gf GQuestion where
  gf (GEven x1) = mkApp (mkCId "Even") [gf x1]
  gf (GOdd x1) = mkApp (mkCId "Odd") [gf x1]
  gf (GPrime x1) = mkApp (mkCId "Prime") [gf x1]
  fg t = case unApp t of
    Just (i,[x1]) | i == mkCId "Even" -> GEven (fg x1)
    Just (i,[x1]) | i == mkCId "Odd" -> GOdd (fg x1)
    Just (i,[x1]) | i == mkCId "Prime" -> GPrime (fg x1)
    _ -> error ("no Question " ++ show t)

```

FIGURE 49 The abstract syntax of a simple query system and the Haskell datatypes exported, with some of the Gf class instances.

```

module Answer where

import PGF (Tree)
import Query

transfer :: Tree -> Tree
transfer = gf . answer . fg

answer :: GQuestion -> GAnswer
answer p = case p of
    GOdd x    -> test odd x
    GEven x   -> test even x
    GPrime x  -> test prime x

value :: GObject -> Int
value e = case e of
    GNumber (GInt i) -> i

test :: (Int -> Bool) -> GObject -> GAnswer
test f x = if f (value x) then GYes else GNo

prime :: Int -> Bool
prime x = elem x primes where
    primes = sieve [2 .. x]
    sieve (p:xs) = p : sieve [n | n <- xs, mod n p > 0]
    sieve [] = []

```

FIGURE 50 A question-answer function for the Query language.

- all GF names are in Haskell prefixed with `G`
- `gf` translates from Haskell objects to GF trees
- `fg` translates from GF trees to Haskell objects

To get access to these facilities, `Answer` must just import the generated `Query` module.

Exercise 7-3.* Define the aggregation function of Section 6.13 in Haskell, by using the abstract syntax of Figure 46 only up to the definition of `aggr`. Now it should be easy to generalize it to iterated aggregation and to an arbitrarily large lexicon.

7.9 Putting it all together

The complete code of the top-level Haskell module `QuerySystem` is shown in Figure 51. In addition to the PGF API, it needs only the function `transfer` from `Answer` in order to compile. The grammar `Query.pgf` is hard-coded in the `main` function; the `Answer` module of course relies on a Haskell datatype system generated from the abstract syntax `Query`.

To automate the production of the system, we write a `Makefile`, which generates the PGF file and the Haskell datatype file (first line) and then compiles the Haskell files (second line):

```
all:
    gf -make -output-format=haskell Query???.gf
    ghc --make -o query QuerySystem.hs
```

(Notice that the empty segments starting the command lines in a Makefile must be tabs.) Now we can compile the whole system by just typing

```
$ make
```

Then we can run it by typing

```
$ ./query
```

To be able to parse any questions, we need at least one concrete syntax of `Query`—but notice that all code we have shown is independent of how many concrete syntaxes there are and for what languages. The file pattern `Query???.gf` compiles them all.

To summarize, the source of the application consists of the following files:

<code>Makefile</code>	-- commands to compile the application
<code>Query.gf</code>	-- abstract syntax
<code>Query???.gf</code>	-- concrete syntaxes for each language
<code>Answer.hs</code>	-- the question-to-answer function

```

module Main where

import PGF
import Answer (transfer)

main :: IO ()
main = do
    gr <- readPGF "Query.pgf"
    loop (translate transfer gr)

loop :: (String -> String) -> IO ()
loop trans = do
    s <- getLine
    if s == "quit" then putStrLn "bye" else do
        putStrLn $ trans s
    loop trans

translate :: (Tree -> Tree) -> PGF -> String -> String
translate tr gr s =
    case parseAllLang gr (startCat gr) s of
        (lg,t:_):_ -> linearize gr lg (tr t)
        _ -> "NO PARSE"

```

FIGURE 51 File `QuerySystem.hs`, the main Haskell module of the query system.

```
QuerySystem.hs    -- Haskell Main module
```

All the other files needed—`Query.hs` and `Query.pgf`—are generated by the command `make`.

Exercise 7-4.* Compile the query system and test it with different grammars and queries.

Exercise 7-5.*⁺ Extend the query system with new question forms, such as

- *what is the product of 2, 3 and 4*
- *which numbers is 91 divisible by*

The answers must correspondingly cover integers and lists of integers.

7.10 Web server applications

In addition to applications running locally, PGF files can be used in web servers. The web server uses a Haskell library enabling the PGF interpreter to communicate with clients via HTTP. On the client side, JavaScript is used for building generic user interfaces that work for all PGF grammars; however, these interfaces are based on generic libraries that can be customized further for particular applications.

One of the servers that can be readily built with the available library (without any programming required) is **fridge poetry magnets**. It is an application that uses an incremental parser to suggest grammatically correct next words (cf. Section 2.3 for incremental parsing). The upper picture in Figure 52 shows an example of its application to the **Foods** grammars. Due to incremental parsing, only feminine forms of adjectives are suggested after *quella pizza*. The lower picture in Figure 52 shows an interface to the same server with character input, which suggests possible continuations of already started words.

The GF website provides the libraries for the server and the web interfaces, including the fridge magnets. Since this is a field of rapid development, we refer to the book web page for pointers to up-to-date documentation, and for the complete source code for the fridge magnet application.

7.11 Embedded grammars in other host languages

Java is an alternative to Haskell as a host language for embedded GF applications. The Java API provides similar functionalities as Haskell in Figure 47, for instance, using PGF grammars as Java objects, reading grammars from files, linearization, and parsing. Figure 53 lists some of the relevant classes and methods, as they are shown to the Java programmer.

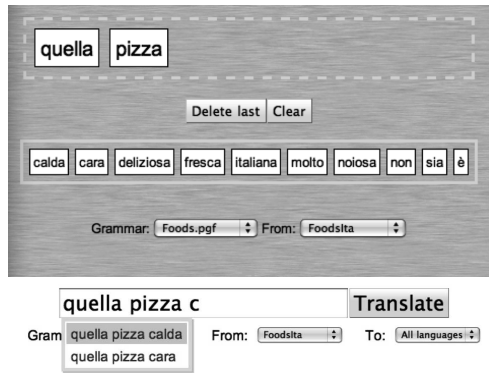


FIGURE 52 Incremental parsing for the Italian foods grammar via two input modes: fridge magnets and text input.

```

public class PGF { ... }

public class PGFBuilder
{... public static PGF fromFile (String filename) ...}

public class Linearizer
{... public String linearizeString(Tree absyn) ...}

public class Parser
{... public ParseState parse(String phrase) ...}

```

FIGURE 53 Some embedded GF interpreter facilities in Java.



FIGURE 54 (Left) Java-based fridge magnet translator running on an Android phone; the “Say it” button launches speech synthesis; (right) JavaScript-based numeral translator running on an iPhone.

At the time of writing, the embedded PGF interpreter in Java is not so complete as the interpreter in Haskell. Its aim is to make it smooth to integrate GF in Java programs. A particular use for this is in mobile phones running the Android operating system, because they support Java programs executed in the phone itself. Figure 54 (left) shows a fridge magnet application running on an Android phone.

Even though Java is a widely used language and enhances the usability of GF outside the functional programming community, it is not always an option. For instance, mobile phones don’t always support Java; the Apple iPhone is an example. One of the most widely supported languages on different platforms is actually JavaScript—a simple, untyped language, which is supported in almost all web browsers, including those on mobile platforms. JavaScript runnable in a web browser is hence one of the most portable formats for application programs, even if not the most efficient one. It is also the most straightforward way to create programs for hand-held devices. Figure 54 (right) shows a JavaScript-based translator running on an Apple iPhone. It translates sequences of digits into numeric expressions in 15 languages.

The JavaScript embedded grammar interpreter doesn’t read PGF files but expects the PGF grammar to be compiled to JavaScript code. This can be made in GF by using the flag `-printer=js` in the GF

batch compiler. The generated grammar code can then be called from an HTML page together with a supporting JavaScript run-time library.

In addition to Haskell, Java, and JavaScript, embedded grammars are currently available for C and Python. The C bindings currently use foreign function calls to the interpreter compiled from Haskell source code. The Python bindings in turn link to the C code, as is customary in Python. They can be used in connection to the Natural Language Toolkit, NLTK (Bird & al. 2009).

7.12 Multilingual syntax editing

One way to use multilingual grammars is via a **syntax editor**. It is a program in which the user can build an abstract syntax tree step by step. The evolving tree is simultaneously displayed in different languages via linearizations. Three screenshots of a syntax editor (implemented in JavaScript and running in a web browser) are shown in Figure 55.

The syntax editing process is started by choosing a category from the “New” menu. Thus choosing **Comment** creates a new tree of type **Comment**. A new tree is in general completely unknown: it consists of just a metavariable **?**. Screenshot (a) in Figure 55 shows the editor at this stage.

Editing goes on by **refinements**, i.e. choices of constructors from the menu, until no metavariables remain. Screenshot (b) in Figure 55 shows a tree with a metavariable for **Quality** and the menu of available refinements. Screenshot (c) shows the editor with a complete tree.

Editing can be continued even when the tree is complete. The user can shift the **focus** to some of the subtrees by clicking at it or the corresponding part of a linearization. In Screenshot (c) in Figure 55, the focus is on “fish”. Since there are no metavariables, refinement is not possible, but there are other possible actions:

- to **replace** “fish” by “cheese” or “wine”
- to **delete** “fish”, i.e. change it to a metavariable
- to **wrap** “fish” in a modification, i.e. change it to **Mod ? Fish**, where the choice of the modifying quality is postponed to a later refinement

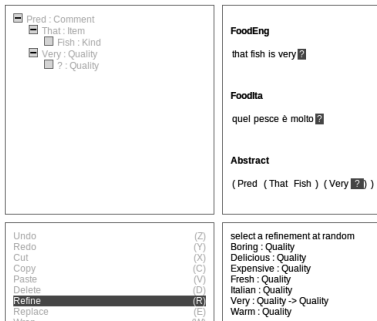
In addition to menu-based editing, the syntax editor supports refinement by parsing. If the parser gives more results than one, the user is shown a menu with the possible trees.

Exercise 7-6. Construct the sentence *this very expensive cheese is very very delicious* and its Italian translation by using the syntax editor (available via the book’s web page).

(a) Initial state of editing.



(b) A refinement menu.



(c) Changing a subtree.



FIGURE 55 Three states of a syntax editor embedded in a web page.

7.13 Language models for speech recognition

In systems with speech input, the first step of processing is **speech recognition**, also known as **speech to text** conversion. A speech recognition system has, for each language, an **acoustic model**, which defines the sounds of the language and their sequences. It also has a dictionary specifying which sound combinations are actually words. Finally, there is a **language model** defining which word combinations are possible. The language model rules out meaningless combinations, which are left as possible interpretations of the input if only the acoustic model and the dictionary are considered. For instance, all of the following strings of words sound the same when spoken:

their must be ice

there must be eyes

there must be ice

However, a very general language model based on grammar or statistics can rule out the first interpretation. A more specific language model, taking into account the context in which the utterance is used, may rule out the second or the third, depending on the context.

The main division of language models is to statistical and grammar-based ones. **Statistical language models** are usually based on *n*-grams derived from a corpus (cf. Section 1.3). **Grammar-based language models** are systems of rules, typically context-free grammar rules. One widely used rule format is **GSL**, which is used in the Nuance speech recognizer (<http://www.nuance.com/>).

In a system that both converts speech to text and then processes text further by a parser, it is useful if the spoken language model and the text parser are consistent with each other. The ultimate way to guarantee this is to use a **single source** for both components. If the parser is derived from a GF grammar, it is natural to do the same with the spoken language model. To this end, GF has compilers into several formats that are used in speech recognition systems, including GSL. They are produced by `gf -make` with the appropriate output format specification, such as `gsl`:

```
$ gf -make -output-format=gsl FoodsEng.gf
```

The result of this command is the context-free Nuance grammar shown in Figure 56.

There are many other formats available, including the Java Speech Grammar Format (JSGF), the SRGS ABNF and XML formats, which are context-free, and finite automata in the HTK SLF format. All currently available formats are listed in GF with `help pg`.


```

;GSL2.0
; Nuance speech recognition grammar for FoodsEng
; Generated by GF
.MAIN Comment_cat
Item_1 [("that" Kind_1) ("this" Kind_1)]
Item_2 [("these" Kind_2) ("those" Kind_2)]
Item_cat [Item_1 Item_2]
Kind_1 ["cheese" "fish" "pizza" (Quality_1 Kind_1)
        "wine"]
Kind_2 ["cheeses" "fish" "pizzas"
        (Quality_1 Kind_2) "wines"]
Kind_cat [Kind_1 Kind_2]
Comment_1 [(Item_1 "is" Quality_1)
            (Item_2 "are" Quality_1)]
Comment_cat Comment_1
Quality_1 ["boring" "delicious" "expensive"
           "fresh" "italian" ("very" Quality_1) "warm"]
Quality_cat Quality_1

```

FIGURE 56 A speech recognition grammar generated from GF in the GSL/Nuance format.

The HTK SLF format is a finite-state approximation of the grammar. The GF-HTK compiler also generates graphviz code for the visualization of automaton diagrams; an example is shown in Figure 5 in Section 1.5.

The generation of context-free and regular grammars in various formats is explained in detail in Bringert (2008).

7.14 Statistical language models

An alternative to grammar-based language models are **statistical language models (SLM)**. An SLM is built from a **corpus**, i.e. a set of utterances. It specifies the probability of each ***n*-gram**, i.e. sequence of *n* words. Typical values of *n* are 2 (bigrams) and 3 (trigrams).

One advantage of SLM's over grammar-based models is that they are **robust**, i.e. they can be used to recognize sequences that would be out of the grammar or the corpus. Another advantage is that an SLM can be built "for free" if a corpus is available.

However, collecting a corpus can require a lot of work, and writing a grammar can be less demanding, especially with tools such as GF or Regulus (Rayner & al. 2006). This advantage of grammars can be combined with robustness by creating a back-up SLM from a **synthesized**

corpus. This means simply that the grammar is used for generating a set of strings. In GF, this can be done with the `generate_trees` command piped into linearization.

```
> generate_trees | linearize
```

As with grammar-based models, the quality of the SLM is better if meaningless utterances are excluded from the corpus. Thus a good way to generate an SLM from a GF grammar is by using semantically motivated application grammars. One can also use random generation with biased probabilities to control the distribution of constructs in the corpus; see Section 6.15 for how to do this.

The method of creating statistical language model from corpora synthesized from GF grammars is applied and evaluated in (Jonson 2006).

7.15 Multimodal dialogue systems

Dialogue systems are among the most complex language processing systems conceivable, since they involve the whole pipeline of language understanding and production:

- speech recognition, from speech to text input
- parsing, from text input to a formal representation of it
- dialogue manager, from the input to a response
- linearization, from the formal response to text
- speech synthesis, from text to speech

Notice the analogy with a compiler pipeline. In a modern compiler, the main part of processing is on the level of abstract syntax trees. In the same way, the most substantial component of a dialogue system is the **dialogue manager**, which performs semantic interpretation, consults a database, updates the information state of the dialogue, and so on. Since the dialogue manager manipulates formal representations and they can be made independent of language, a GF-like technology can be used for making dialogue systems multilingual: for each new language, one just needs to localize the outermost parts but not the dialogue manager. As we have seen, a domain-specific language model for speech recognition can be produced from the same grammar as the parser. Moreover, as shown in Cooper and Ranta (2004), Bringert (2008), and Larsson and Ljunglöf (2008), a substantial part of the dialogue manager can be derived from an abstract syntax by using ideas similar to syntax editors presented in Section 7.12.

As if dialogue systems were not complex enough as described above, yet another component is often introduced to the architecture: **multimodality**. This means that the user has more input modalities than

just speech—typically, mouse clicks. Integrating speech and mouse clicks to one input is an instance of the traditional linguistic notion of **demonstratives**, expressions that get their meaning from pointing gestures. Thus someone can point on a map and say,

*I want to go from **this place** to **this place***

The speech part indicates just *that* the person wants to go, and the pointing gestures add the information *where*.

In demonstrative expressions,

meaning-bearing unit = words + pointing

In dialogue systems this is called **integrated multimodality**. The pointing means a mouse click or something else machine-observable. The system records the coordinates of the pointings and can from them build a meaning representation, something like

want_go (place (123,45)) (place (98,10))

Experience from dialogue systems shows that speech and pointing gestures need not be *simultaneous*: the clicks can appear just any time around the spoken utterance:

I want to go from this place CLICK to this place CLICK

CLICK CLICK I want to go from this place to this place

I want to go from this place to this place CLICK CLICK

In fact, devices such as mice make it difficult to click at exactly the same time as one says “this”. All that matters is *order*: the first click is associated with the first “this”, the second click with the second “this”.

How do we build the semantic value of a multimodal input compositionally in parsing? We need a notion of **asynchronous parsing**. We need to *fuse* the two asynchronous input strings—speech and clicks—into one input object. In GF we can do this simply by using records! A demonstrative expression, such as the utterance *this place* with a click at a coordinate, has a record as its concrete syntax:

this place (coord 123 45)	$\iff$	<pre> {s = "this place" ; p = "(123,45)" }</pre>

The **s** field comes from the speech recognizer and the **p** field from the click sensor, which maps coordinates into locations. The grammar in Figure 57 gives a minimal configuration with integrated multimodality. Notice that the representation of demonstratives is similar to **discontinuous constituents**, which we know perfectly well how to parse, as seen in Section 3.10.

```

abstract Map = {
cat
  Query ; Input ; Place ; Click ;
fun
  GoFromTo    : Dep -> Dest -> Input ;
  ThisPlace   : Click -> Place ;
  QueryInput  : Input -> Query ;
  ClickCoord  : Int -> Int -> Click ;
}

concrete MapEng of Map = {
lincat
  Query      = {s : Str} ;
  Input, Place = {s : Str ; p : Str} ;
  Click      = {p : Str} ;
lin
  GoFromTo x y = {
    s = "I want to go from" ++ x.s ++ "to" ++ y.s ;
    p = x.p ++ y.p
  } ;
  ThisPlace c = {
    s = "this place" ;
    p = c.p
  } ;
  QueryInput i = {s = i.s ++ ";" ++ i.p} ;
  ClickCoord x y = {p = "(" ++ x.s ++ "," ++ y.s ++ ")" } ;
}

```

FIGURE 57 A multimodal grammar with clicks.

The system works as follows: the input comes asynchronously from the speech recognizer and the click sensor, which don't know about each other. The input is collected into two separate record fields, **s** for the speech and **p** for the click. When the input is terminated, these two strings are concatenated with the separator **;** between them. The resulting string can be parsed in an ordinary way by a GF parser generated from the grammar in Figure 57. The order of clicks is known to the parser, and this is enough for building the abstract syntax tree.

Interfacing formal and natural languages

Formal languages are, generally speaking, less challenging than natural languages for grammar writers. They are often context-free by design. Nevertheless, they have some special features whose treatment in GF it is useful to know, since some of the most typical GF applications are natural-language interfaces to formal systems. In such an interface, the translation between natural and formal language is defined by a multilingual grammar. Grammars of formal languages also open the way for non-linguistic applications, such as compilers implemented in GF and the generation of graphics by grammars.

The main contents of this chapter are:

- arithmetic expressions and precedences
- code generation as linearization
- strict vs. liberal abstract syntax
- natural language generation from logic
- logical semantics of natural language
- graftals: grammars for fractals

8.1 Arithmetic expressions

Arithmetic expressions are a subset of many languages, both formal languages and informal mathematical text. Such expressions are built from integers by the binary operations of addition, subtraction, multiplication, and division. Their abstract syntax is easy to write, as shown in Figure 58. We give it the name **Calculator**, since it can be used as the basis of a calculator.

Arithmetic expressions should be unambiguous. If we parse

$2 + 3 * 4$

```

abstract Calculator = {
  flags startcat = Exp ;
  cat Exp ;
  fun
    EPlus, EMinus, ETimes, EDiv : Exp -> Exp -> Exp ;
    EInt : Int -> Exp ;
}

```

FIGURE 58 The abstract syntax of a simple calculator.

we should receive one, but not both, of

```

EPlus (EInt 2) (ETimes (EInt 3) (EInt 4))
ETimes (EPlus (EInt 2) (EInt 3)) (EInt 4)

```

Under normal conventions, the former is chosen, because multiplication has **higher precedence** than addition. If we want to express the latter tree, we have to use parentheses:

$$(2 + 3) * 4$$

Here is a summary of the usual precedence rules in mathematics and programming languages:

- Integer constants and expressions in parentheses have the highest precedence.
- Multiplication and division have equal precedence, lower than the highest but higher than addition and subtraction, which are again equal.
- All the four binary operations are **left-associative**, which means that e.g. $1 + 2 + 3$ means the same as $(1 + 2) + 3$.

One way of dealing with precedences in compiler books is by dividing expressions into three categories:

- expressions: addition and subtraction
- terms: multiplication and division
- factors: constants and expressions in parentheses

A context-free grammar, which also takes care of associativity, is the following:

```

Exp ::= Exp "+" Term | Exp "-" Term | Term ;
Term ::= Term "*" Fact | Term "/" Fact | Fact ;
Fact ::= Int | "(" Exp ")" ;

```

The compiler, however, doesn't make a semantic distinction between the three categories. Hence it doesn't want to clutter the syntax trees

with constructors that just convert expressions between precedence levels. In compiler tools such as YACC, building abstract syntax trees is performed as a **semantic action**. For instance, if the parser recognizes an expression in parentheses, the action is to return only the abstract syntax tree, without encoding the parentheses.

In GF, semantic actions could be encoded by using `def` definitions. But there is a more straightforward way of thinking about precedences: we introduce a parameter for precedence, and treat it as an inherent feature of expressions:

```
oper
  param Prec = Ints 2 ;
  TermPrec : Type = {s : Str ; p : Prec} ;
lincat
  Exp = TermPrec ;
```

This example shows another way to use built-in integers in GF: the type `Ints 2` is a parameter type, whose values are the integers 0,1,2. These are the three precedence levels we need. The main idea is to compare the inherent precedence of an expression with the context in which it is used. If the precedence is higher than or equal to the expected, then no parentheses are needed. Otherwise they are. The result is similar to defining `Fact` in the above grammar as `Exp` with precedence 2, `Term` as `Exp` with precedence 1, and `Exp` itself as `Exp` with precedence 0. Notice that the treatment of precedences in concrete syntax rather than on the category level is similar to the parametrization of morphological distinctions in Section 3.1.

The general rule for using parameters as a function of precedence is implemented by the operation

```
oper usePrec : TermPrec -> Prec -> Str = \x,p ->
  case lessPrec x.p p of {
    True  => "(" ++ x.s ++ ")" ;
    False => x.s
  } ;
```

where `lessPrec` defines the ordering of integers. With this low-level operation, we can define other ones, such as the operation that forms left-associative infix expressions:

```
infixl : Prec -> Str -> (_,_ : TermPrec) -> TermPrec =
  \p,f,x,y -> {
    s = usePrec x p ++ f ++ usePrec y (nextPrec p) ;
    p = p
  } ;
```



```

concrete CalculatorC of Calculator =
  open Formal, Prelude in {
lincat
  Exp = TermPrec ;
lin
  EPlus  = infixl 0 "+" ;
  EMinus = infixl 0 "-" ;
  ETimes = infixl 1 "*" ;
  EDiv   = infixl 1 "/" ;
  EInt i = constant i.s ;
  }

```

FIGURE 59 Arithmetic expressions with precedences.

where `nextPrec` gives the next precedence level above. Constant-like expressions (the highest level) are built by

```
constant : Str -> TermPrec = \s -> {s = s ; p = maxPrec}
```

where `maxPrec` = 2 in the current example. All these operations can be found in the library module `Formal` (see Section D.4.2) so we don't need to define them in our own code. Thus we can express the whole concrete syntax of `Calculator` compactly as shown in Figure 59. The linearization rules in fact look very much like the infix directives of the Haskell programming language.

Let us just take one more look at the operation `usePrec`, which decides whether to put parentheses around a term or not. The case where parentheses are not needed around a string was defined as the string itself. However, this would imply that superfluous parentheses are never correct. A more liberal grammar is obtained by using the operation

```
parenthOpt : Str -> Str = \s -> s | "(" ++ s ++ ")"
```

which is the one actually used in the `Formal` library. But even in this way, we can only allow one pair (or some other finite number or pairs) of superfluous parentheses. Thus the parameter-based grammar has not quite reached the goal of implementing the same language as the expression-term-factor grammar. But it has the advantage of eliminating precedence distinctions from the abstract syntax.

Exercise 8-1. Define non-associative and right-associative infix operations analogous to `infixl`.

Exercise 8-2. Add a constructor that puts parentheses around expressions to raise their precedence, but is eliminated by a `def` definition (see

Section 6.11). Test parsing with and without a pipe to `pt -compute`.

Exercise 8-3.* Implement a calculator that returns numeric results as an embedded grammar application. You can even extend this to a natural-language calculator that uses number words, which are implemented in GF currently for 88 languages (see `examples/numerals` on the GF web page).

8.2 Code generation as linearization

The standard use of grammars of programming languages is in **compilers**, which translate one language into another. Typically the source language of a compiler is a high-level language and the target language is a machine language. The hub of a compiler is abstract syntax: the **front end** of the compiler parses source language strings into abstract syntax trees, and the **back end** linearizes these trees into target language strings. This processing model is of course what GF uses for natural language translation; the main difference is that, in GF, the compiler can run in the opposite direction as well, that is, function as a **decompiler**. (In full-size compilers, however, the abstract syntax is transformed by several layers of semantic analysis and optimizations, before the target code is generated; this can destroy reversibility and hence decompilation.)

More for the sake of illustration than as a serious compiler, let us write a concrete syntax of **Calculator** that generates machine code similar to JVM (Java Virtual Machine). JVM is a so-called **stack machine**, whose code follows the **postfix** notation, also known as **reverse Polish** notation. Thus the expression

$$2 + 3 * 4$$

is translated to

```
ldc 2 : ldc 3 ; ldc 4 ; imul ; iadd
```

The linearization rules are straightforward, and given in Figure 60. The use of the instruction `ldc` is suboptimal; it does work for all integers of the Java type `int`, but real Java compilers use special instructions for small integers.

8.3 Programs with variables

A useful extension of arithmetic expressions is a **straight code** programming language. The programs of this language consist of **assignments** of the form `x = exp`, which give values to variables. Expressions can contain variables that have been given values in previous assignments; thus the assignment

```

concrete CalculatorJ of Calculator = open Prelude in {
lincat
  Exp = SS ;
lin
  EPlus  = postfix "iadd" ;
  EMinus = postfix "isub" ;
  ETimes = postfix "imul" ;
  EDiv   = postfix "idiv" ;
  EInt i = ss ("ldc" ++ i.s) ;
oper
  postfix : Str -> SS -> SS -> SS = \op,x,y ->
    ss (x.s ++ ";" ++ y.s ++ ";" ++ op) ;
}

```

FIGURE 60 JVM-style arithmetic expressions.

```
x = x + 1
```

is legal if and only if there is an earlier assignment to `x`.

In this language, we use two new categories: programs and variables. A program is a sequence of assignments, where a variable is given a value. Logically, we want to distinguish **initializations** from other assignments: these are the assignments where a variable is given a value for the first time. The concrete syntax follows C and prefixes the type of the variable to the initializing assignment. Here is an example of a piece of code written in the language:

```

int x = 2 + 3 ;
int y = x + 1 ;
x = x + 9 * y ;

```

We define programs by the following constructors:

```

fun
  PEmpty : Prog ;
  PInit  : Exp -> (Var -> Prog) -> Prog ;
  PAss   : Var -> Exp -> Prog -> Prog ;

```

The most interesting constructor is `PInit`, which uses higher-order abstract syntax (cf. Section 6.9) for making the initialized variable available in the **continuation** of the program. The abstract syntax tree for the above code is

```

PInit (EPlus (EInt 2) (EInt 3)) (\x ->
  PInit (EPlus (EVar x) (EInt 1)) (\y ->
    PAss x (EPlus (EVar x) (ETimes (EInt 9) (EVar y)))
  )
)

```

```
PEmpty))
```

Since we want to prevent the use of uninitialized variables in programs, we don't give any constructors for **Var**! We just have a rule for using variables as expressions:

```
fun EVar : Var -> Exp ;
```

Hence all variables are created by bindings in **PInit** in the higher-order abstract syntax, which guarantees that there are no unbound or uninitialized variables.

The rest of the grammar is just the same as for arithmetic expressions in Section 8.1. The best way to implement it is perhaps by writing a module that extends the expression module. The most natural start category of the extension is **Prog**.

Exercise 8-4.* Extend the straight-code language to expressions of type **float**. To guarantee type safety, you can define a category **Typ** of types, and make **Exp** and **Var** dependent on **Typ**. Basic floating point expressions can be formed from literals of the built-in GF type **Float**. The arithmetic operations should be made polymorphic, as in Section 6.4.

8.4 The concrete syntax of assignments

We can define a C-like concrete syntax for the straight-code language by using GF's \$ variables explained in Section 6.9. In a JVM-like syntax, we need two more instructions: **iload** *x*, which loads (pushes on the stack) the value of the variable *x*, and **istore** *x*, which stores the value of the currently topmost expression in the variable *x*. Thus the code for the example in the previous section is

```
ldc 2    ; ldc 3 ; iadd ; istore x ;
iload x  ; ldc 1 ; iadd ; istore y ;
iload x  ; ldc 9 ; iload y ; imul ; iadd ; istore x ;
```

Readers familiar with JVM will notice that we are using **symbolic addresses**, i.e. variable names, instead of integer offsets in the memory as in real JVM. Neither real JVM nor our variant makes any distinction between the initialization and reassignment of a variable.

Exercise 8-5.* Finish the implementation of the C-to-JVM compiler by extending the expression modules to straight code programs. Would it be possible to use numeric addresses instead of symbolic ones?

Exercise 8-6.* If you made the exercise of adding floating point numbers to the language, you can now cash out the main advantage of type checking for code generation: selecting type-correct JVM instructions. The floating point instructions are precisely the same as the integer

ones, except that the prefix is `f` instead of `i`. The `ldc` instruction can take both integers and floating point numbers as arguments.

8.5 A liberal syntax of variables

In many applications, the task of GF is just linearization and parsing; keeping track of bound variables and other semantic constraints is the task of other parts of the program. For instance, if we want to write a natural language interface that converts C code to English text, we can quite as well use a context-free grammar of C, and leave it to the C compiler to check that variables are well-typed. In such a program, we may want to treat variables as strings, i.e. to have a constructor

```
fun VString : String -> Var ;
```

As specified in Section 6.14, the built-in category `String` has as its values **string literals**, which are strings in double quotes. The lexer and unlexer can be defined in such a way that they restore and remove the quotes: when the lexer finds a token that is neither a terminal in the grammar nor an integer literal, it should send it to the parser as a string literal.

Exercise 8-7. Write a grammar for straight code without higher-order abstract syntax.

Exercise 8-8.* Extend the liberal straight code grammar to `while` loops and some other imperative program constructs.

8.6 Is GF useful for defining formal languages?

Since formal languages are syntactically simpler than natural languages, it is no wonder that their grammars can be defined in GF. Some thought is needed for dealing with precedences and spacing, but much of it is encoded in GF's libraries and built-in lexers and unlexers. However, if the sole purpose of a grammar is to implement a programming language, then the **BNF Converter** tool (BNFC) is more appropriate than GF. BNFC is a spin-off of GF, using a labelled BNF notation and generating parsers via standard YACC-like parser tools (see Forsberg 2007).

The most common applications of GF grammars of formal languages are in natural-language interfaces of various kinds. These systems don't usually need semantic control in GF abstract syntax. However, the situation can be different if the interface also comprises an interactive syntax editor, as in the GF-Key system (Beckert & al. 2006, Burke & Johannisson 2005). In that system, the editor is used for guiding programmers to write only type-correct code.

Exercise 8-9.*+ The book *C programming language* by Kernighan and Ritchie (p. 123, 2nd edition, 1988) describes an English-like syntax for pointer and array declarations, and a C program for translating between English and C. The following example pair shows all the expression forms needed:

```
char (*(x[3]))[5]
```

```
x: array[3] of pointer to function returning
   pointer to array[5] of char
```

Implement these translations by a GF grammar.

Exercise 8-10.*+ Extend the natural language translator to other programming language constructs. Investigate if you can build a reasonable text generator for this fragment. One potentially useful application of such a text generator is the conversion of code into speech, by the use of speech synthesis software. When combined with speech recognition, it could also enable programming by speech.

Exercise 8-11.*+ Design a natural-language interface to Unix command lines. It should be able to express verbally commands such as `cat`, `cd`, `grep`, `ls`, `mv`, `rm`, `wc` and also pipes built from them. For instance,

```
ls ~ | wc -l
```

might be expressed as *list the files in my home directory and count the lines* or, more ambitiously, *how many files are there in my home directory*.

8.7 Natural language generation from logic

Translating between logic and natural language has been a typical task in projects applying GF. It is often combined with tools for reasoning in a logic. Natural language is used both for rendering logical expressions into a form readable by layman users and for helping with the construction of logical representations, often in connection with a syntax editor (see Section 7.12). The GF-Alfa proof editor (Hallgren and Ranta 2000), the GF-KeY software specification editor (Johannisson 2005), and the WebALT mathematical exercise translator (Caprotti and Seppälä 2006) are examples of this.

There are many systems and notations for logic, but many of their differences disappear in abstract syntax. The core of such a syntax consists of the types for propositions (formulas) and individuals (terms), and the constructors of logically complex propositions: connectives and

quantifiers. Some crucial decisions have to be made already at this level. First of all, do we want to capture semantic well-formedness in the abstract syntax? If the answer is yes, then we should use higher-order abstract syntax for bindings in quantifiers, for instance,

```
fun All : (Ind -> Prop) -> Prop
```

as in Section 6.9. One may even build a syntax that implements domain-specific quantification and type restrictions of predicates, as in full-scale constructive type theory:

```
fun All : (A : Set) -> (Ind A -> Prop) -> Prop
```

The advantage of this syntax is that we can then get more functionality out of GF's type checker and syntax editor; for instance, Johannisson (2005) could in this way build an editor that guided the user to build only well-typed formal specifications. Figure 64 in Section 8.8 shows an abstract syntax built in this way. This syntax contains the full set of connectives and quantifiers of first-order logic. As it is designed to give the semantics to the miniature resource grammar in Chapter 9 and Appendix A, it moreover contains a temporal operator, **Past**.

One disadvantage of full semantic typing is that bindings and dependent types make the syntax difficult to extend. For instance, a natural-language-like universal quantifier should allow several variables to be bound simultaneously, as in *for all x, y and $z, \dots$* , but implementing this with higher-order abstract syntax needs a heavy machinery of auxiliary constructs for dealing with lists of bindings (see exercise at the end of this section). Another disadvantage is that the grammar may be difficult to integrate with other tools. For instance, the exportation of datatypes (see Section 7.8) to Haskell no more works, because GF's higher order and dependent types cannot be interpreted.

Fortunately, semantic checking on GF side is often unnecessary, because the logical tool with which the grammar is integrated has its own checking mechanisms. The abstract syntax can then be made liberal in the sense of Section 8.5, and it is easily extensible with natural-language-like constructs. Such an abstract syntax is shown in Figure 61. It divides logical formulas to top-level statements (**Stm**), propositions (**Prop**), and atomic formulas (**Atom**). Singular terms are divided to individuals (**Ind**) and variables (**Var**). The category **Dom** is used for restricting quantification to domains. The grammar moreover uses two **list categories**: **[Prop]** to permit list conjunctions, and **[Var]** to permit the binding of several variables at once. The notation for list categories is a GF shorthand, which is explained in Section C.4.3.

The **Logic** module can be extended with lexical modules, such as the

```

abstract Logic = {
flags startcat = Stm ;
cat
  Stm ;      -- top-level statement
  Prop ;     -- proposition
  Atom ;     -- atomic formula
  Ind ;      -- individual term
  Dom ;      -- domain expression
  Var ;      -- variable
  [Prop] {2} ; -- list of propositions, 2 or more
  [Var]  {1} ; -- list of variables, 1 or more
fun
  SProp      : Prop -> Stm ;
  And, Or    : [Prop] -> Prop ;
  If         : Prop -> Prop -> Prop ;
  Not        : Prop -> Prop ;
  PAtom      : Atom -> Prop ;
  All, Exist : [Var] -> Dom -> Prop -> Prop ;
  IVar       : Var -> Ind ;
  VString    : String -> Var ;
}

```

FIGURE 61 Liberal abstract syntax of a basic logic system.


```

abstract Geometry = Logic ** {
fun
  Line, Point, Circle : Dom ;
  Intersect, Parallel : Ind -> Ind -> Prop ;
  Vertical : Ind -> Prop ;
  Centre : Ind -> Ind ;
}

concrete GeometryEng of Geometry = LogicEng **
  open SyntaxEng, ParadigmsEng in {
lin
  Line = mkN "line" ;
  Point = mkN "point" ;
  Circle = mkN "circle" ;
  Intersect = pred (mkV2 "intersect") ;
  Parallel = pred (mkA2 (mkA "parallel")(mkPrep "to")) ;
  Vertical = pred (mkA "vertical") ;
  Centre = app (mkN2 (mkN "centre") (mkPrep "of")) ;
}

```

FIGURE 62 Lexicon for geometry with English concrete syntax.

lexicon for geometry in Figure 62 (module **Geometry**). A typical lexicon defines domains (*line*, *point*), predicates (*x intersects y*, *x is parallel to y*), and individual terms and functions (*the centre of x*).

In the concrete syntax, atomic formulas are naturally treated as clauses, domains as common nouns, individuals as noun phrases, and predicates as functions that use verbal or adjectival predication. The resource grammar module **Combinators** (see Section D.4.4) contains a set of predication and application functions that are designed for this purpose. Unlike the “ordinary” resource functions, they take the predicate as their first argument, which enables concise definitions using partial application. Figure 62 gives examples of this.

It is easy to write a base-line concrete syntax of **Logic**, as shown in Figure 63. This syntax can be generalized into a functor over the Resource Grammar Library, by just moving the constant **case.CN** into an interface. The comments in the code show how each logical form is expressed. The resource grammar expressions for negation and quantification are slightly complex, but not too difficult to figure out.

The concrete syntax in Figure 63 does not, however, generate very good language. First, the connectives **And** and **Or** are rendered with plain infix conjunctions *and* and *or*, which produces ambiguous sen-

```

concrete LogicEng of Logic = open
  SyntaxEng, (P=ParadigmsEng), SymbolicEng, Prelude in {
lincat
  Stm  = Text ;
  Prop = S ;
  Atom = Cl ;
  Ind  = NP ;
  Dom  = CN ;
  Var  = NP ;
  [Prop] = [S] ;
  [Var]  = NP ;
lin
  SProp = mkText ;
  And = mkS and_Conj ;          -- A, B ... and C
  Or  = mkS or_Conj ;          -- A, B ... or C
  If A B =                      -- if A B
    mkS (mkAdv if_Subj A) B ;
  Not A =                      -- it is not the case that A
    mkS negativePol (mkCl
      (mkVP (mkNP the_Quant
        (mkCN case_CN A)))) ;
  All xs A B =                  -- for all A's xs, B
    mkS (mkAdv for_Prep
      (mkNP all_Predet (mkNP a_Quant
        plNum (mkCN A xs)))) B ;
  Exist xs A B =                -- for some A's xs, B
    mkS (mkAdv for_Prep
      (mkNP somePl_Det (mkCN A xs))) B ;
  PAtom = mkS ;
  IVar x = x ;
  VString s = symb s ;
  BaseProp A B = mkListS A B ;
  ConsProp A As = mkListS A As ;
  BaseVar x = x ;
  ConsVar x xs = mkNP and_Conj (mkListNP x xs) ;
oper
  case_CN : CN = mkCN (P.mkN "case") ;
}

```

FIGURE 63 A base-line concrete syntax of Logic.

tences, starting from those of the form *A and B or C*. Secondly, the negation is clumsy, as it is not able to use verb negations (*x is not parallel to y*) but only the prefix *it is not the case that...* Thirdly, quantification is always with explicit variables (*for all points x...*), and in-place quantifiers cannot be used (*every line is parallel to some line*). All these problems are familiar from the field of **natural language generation (NLG)**. NLG is usually considered separately from grammars: given a set of grammatically correct expressions, its task is to find the best ones in terms of style, fluency, clarity, and so on.

Many NLG techniques, such as aggregation (cf. Section 6.13), are non-compositional, and thus cannot be defined as part of linearization. They are more properly implemented with embedded grammars or with transfer functions defined by `def` definitions in GF. However, the use of suitable parameters in linearization sometimes does suffice for solving NLG problems smoothly. Let us show how this technique applies to the problem of negation.

The negation rule we want to implement is the following (adapted from Burke & Johannisson 2005):

- to negate an atomic proposition, negate its verb
- to negate a complex proposition, use the prefix *it is not the case*

To enforce this rule, we have to keep track of whether a proposition (`Prop`) is atomic or complex. This suggests the linearization type

```
lincat Prop = {s : S ; isAtom : Bool}
```

In the `Logic` grammar, the atomic propositions are exactly those formed by the constructor `PAtom`. Thus we put

```
lin PAtom p = {s = mkS p ; isAtom = True}
```

All other `Prop` constructors return `isAtom = False`.

To negate an atomic proposition, we want to negate the verb of the sentence. However, the `S` category in the library already has a fixed polarity. Only *clauses* (`Cl`) have polarity as variable feature. Thus the negation must be formed *before* `PAtom` is applied, and remembered in the linearization record:

```
lincat Prop = {pos, neg : S ; isAtom : Bool}
lin PAtom p = {
  pos = mkS p ;
  neg = mkS negativePol p ;
  isAtom = True
}
```

As all other `Prop`-forming constructors produce complex propositions,

they can be implemented by wrapping the simple-minded linearizations in Figure 63 with the operation

```
oper complexProp : S -> {pos,neg : S ; isAtom : Bool} =
  \s -> {
    pos = s ;
    neg = mkS negativePol (mkCl (mkVP
      (mkNP the_Quant (mkCN case_CN s)))) ;
    isAtom = False
  }
```

The problem of the ambiguity of conjunctions can be solved by using *both-and* and *either-or*. For instance, *(A and B) or C* is *either A and B or C*, whereas *A and (B or C)* is *A and either B or C*. These constructs are available in the resource library, but of course give clumsy results when iterated. Another solution is given in Burke & Johannisson (2005), who use bulleted lists. For instance, *A and (B or C)* can be expressed as

both of the following hold:

- *A*
- *B or C*

The production of in-site quantification from variable binding constructs is an interesting problem, which probably cannot be solved without an external transfer component. In the next section, we will look at it from the opposite perspective: define how the in-site quantifiers available in the resource grammar are interpreted in logic.

Exercise 8-12. Build a domain lexicon of your own on top of **Logic** in Figure 61.

Exercise 8-13.* Write a symbolic concrete syntax for **Logic** in Figure 61 and **Geometry** in Figure 62, to produce formulas in your favourite formalism of predicate logic.

Exercise 8-14.* Define the abstract and concrete syntax for a universal quantifier that binds a list of one or more variables, each of the same type. This syntax should permit formulas such as

$$(\text{All } x,y,z : \text{Nat})(x < y \ \& \ y < z \rightarrow x < z)$$

and guarantee the well-typedness of formulas by higher-order abstract syntax.

Exercise 8-15. Complete the improvement of the **LogicEng** module for negations using the **isAtom** parameter.

Exercise 8-16.* Negation expressions in `LogicEng` can be further simplified if we assume the laws of double negation ($\sim\sim A \Leftrightarrow A$) and the De Morgan laws ($\sim(A \vee B) \Leftrightarrow \sim A \& \sim B$, $\sim(A \& B) \Leftrightarrow \sim A \vee \sim B$). Implement this idea in the `LogicEng` module.

8.8 Logical semantics of natural language

We saw in the previous section how multilingual grammars can be used for a baseline translation between logic and natural language. We also found that language generated in this way can be clumsy and unnatural. One of the problems is **quantification**: while formal logic uses variable bindings (*for all numbers x , x is even or odd*), natural language prefers in-place quantifier phrases (*every number is even or odd*).

The problem of quantification in natural language is usually approached in a direction opposite to starting from logic and generating language: starting from natural language structures, how to produce the corresponding formulas in logic? This approach is probably older and more widely studied than the natural language generation problem. It started with the work of Montague (collected in Montague 1974), and is therefore often called **Montague semantics**. A more descriptive term is **logical semantics**. From the computer science point of view, logical semantics is an instance of **denotational semantics**, as it is based on the idea that the meaning of an expression is its **denotation**, that is, an object in some mathematical universe.

We will now show how to implement logical semantics in GF. We will work with a grammar that is a part of the original Montague semantics and also of the GF resource grammar library. This grammar is the *miniature resource grammar*, which is thoroughly discussed in Chapter 9. It is implemented for Italian, with all details given in Appendix A. But as the GF Resource Grammar Library covers the same fragment in 16 languages (in Summer 2010), the semantics can be easily ported to all these languages. A more comprehensive account of logical semantics, comprising anaphora (Section 6.10), can be found in Ranta 2004b.

A system of logical semantics covering the entire resource grammar was built by Bringert (2008), who moreover connected it with a theorem proving system (Equinox, Claessen 2005) and a large English lexicon (the *Oxford Advanced Learner's Dictionary*). Bringert's system was implemented by embedded grammars in Haskell, and tested against the *FRACAS Test Suite* (Kamp & al. 1994). While using a powerful host language certainly is the right choice in a large-scale system like Bringert's, we will here show a GF implementation because of its simplicity and familiarity to all readers of this book.

```

abstract Logic = {
  cat
    Prop ; Ind ;
  data
    And, Or, If : Prop -> Prop -> Prop ;
    Not          : Prop -> Prop ;
    All, Exist   : (Ind -> Prop) -> Prop ;
    Past         : Prop -> Prop ;
}

```

FIGURE 64 Higher-order abstract syntax of predicate logic with a tense.

Our semantics maps the abstract syntax of natural language in the module **Grammar** (Section A.1) to the abstract syntax of logic in the module **Logic** (Figure 64). The semantics itself is defined in the module **Semantics** (Figure 65).

The **Semantics** module begins with declarations of the **interpretation functions** iC , one for each category C in **Grammar**. Each of these functions takes a tree of type C as its argument, and returns its denotation. The denotation is usually a function type, except for **S** and **Cl**, which use **Prop**. One-place verbs (**V**), adjectives (**A**), and common nouns (**N**) are interpreted as one-place propositional functions (**Ind** $\rightarrow$ **Prop**). This extends from the lexical categories to the corresponding phrasal ones (**VP**, **AP**, **CN**). Two-place verbs (**V2**) are two-place propositional functions. Conjunctions (**Conj**) are two-place functions on propositions. Tense and polarity (**Tense**, **Pol**) are one-place functions on propositions.

The rest of the categories are interpreted as higher-order functions. For adjectives (**AdA**), this is clear: they convert **AP** to **AP** and are thus interpreted as functions from **AP** denotations to **AP** denotations. Noun phrases (**NP**) have the same type as quantifiers in **Logic**. This is what is needed to interpret noun phrases like *every man*. If the noun phrase happens to denote an individual and correspond to a singular term in logic, the semantics applies the argument propositional function **f** to that individual: assuming **john_NP** denotes **john**, we have

```
def iNP john_NP f = f john
```

Determiners are naturally interpreted as functions from **CN** denotations to **NP** denotations. Their syntax always involves a domain, and this is encoded in **Logic** in the usual way by restricting the quantification with **And** in the existential case and **If** in the universal case.

The **def** definitions of the interpretation functions use pattern

```

abstract Semantics = Grammar, Logic ** {
fun
  iS   : S   -> Prop ;
  iCl  : Cl  -> Prop ;
  iNP  : NP  -> (Ind -> Prop) -> Prop ;
  iVP  : VP  -> Ind -> Prop ;
  iAP  : AP  -> Ind -> Prop ;
  iCN  : CN  -> Ind -> Prop ;
  iDet : Det -> (Ind -> Prop) -> (Ind -> Prop) -> Prop ;
  iN   : N   -> Ind -> Prop ;
  iA   : A   -> Ind -> Prop ;
  iV   : V   -> Ind -> Prop ;
  iV2  : V2  -> Ind -> Ind -> Prop ;
  iAdA : AdA -> (Ind -> Prop) -> Ind -> Prop ;
  iPol : Pol -> Prop -> Prop ;
  iTense : Tense -> Prop -> Prop ;
def
  iS (UseCl t p cl) = iTense t (iPol p (iCl cl)) ;
  iCl (PredVP np vp) = iNP np (iVP vp) ;
  iVP (ComplV2 v2 np) i = iNP np (iV2 v2 i) ;
  iNP (DetCN det cn) f = iDet det (iCN cn) f ;
  iCN (ModCN ap cn) i = And (iAP ap i) (iCN cn i) ;
  iVP (CompAP ap) i = iAP ap i ;
  iAP (AdAP ada ap) i = iAdA ada (iAP ap) i ;
  iS (ConjS co x y) = iConj co (iS x) (iS y) ;
  iNP (ConjNP co x y) f = iConj co (iNP x f)(iNP y f) ;
  iVP (UseV v) i = iV v i ;
  iAP (UseA a) i = iA a i ;
  iCN (UseN n) i = iN n i ;
  iDet a_Det d f = Exist (\x -> And (d x) (f x)) ;
  iDet every_Det d f = All (\x -> If (d x) (f x)) ;
  iPol Pos t = t ;
  iPol Neg t = Not t ;
  iTense Pres t = t ;
  iTense Perf t = Past t ;
  iConj and_Conj a b = And a b ;
  iConj or_Conj a b = Or a b ;
}

```

FIGURE 65 Denotational semantics of Grammar in Logic.

matching on their first arguments. They are in fact **compositional**, in the sense of Section 6.13, since they never do case analysis inside their immediate arguments. As they are not defined for all constructors, they are just partial functions. In particular, we have left most of the lexicon uninterpreted. Thus an interpretation of a **Grammar** tree may compute into a **Logic** tree that has remaining calls to interpretation functions.

To test the grammar, we need a test lexicon, the module **Test** defined in Section A.5 on top of **Grammar**. To test the semantics, we need a grammar that has both **Semantics** and **Test** in scope. As we don't need to linearize the interpretation functions, nor to give semantics to the test lexicon, we just need two dummy modules in addition to the once already shown:

```
abstract TestSemantics = Test, Semantics ;
```

```
concrete TestSemanticsIta of TestSemantics = TestIta ;
```

If we now import **TestSemanticsIta** in GF, we can parse Italian sentences and interpret them in logic:

```
> i TestSemanticsIta.gf
> p -tr "ogni donna cammina" | pt -transfer=iS
UseC1 Pres Pos (PredVP
  (DetCN every_Det (UseN woman_N)) (UseV walk_V))
All (\x -> If (iN woman_N x) (iV walk_V x))
```

Since logical semantics can be given to all constructs of the GF resource grammar, it can also be given to all application grammars implemented by using the resource. This semantics could actually be produced mechanically: if an application defines

$$\text{lin } f = t$$

using the resource grammar API, we can from this derive

$$\text{def } f = t'$$

where t' is the term obtained from t by performing overload resolution and using the **oper** definitions of the resource as **def** definitions. Such semantics is, however, not always interesting in applications. For instance, in a dialogue system for travel booking, we don't want to compute *I would like to go from Gothenburg to Munich* into a complex formula involving the denotations of *I* and *would* and so on, but just the abstract syntax tree in the style **Ticket Got Muc**, as specified in Figure 16 in Section 2.11.

Exercise 8-17. Extend the semantics of **Test** by giving denotations (at least some dummy ones) to all constructs, so that you can compute

denotations free from interpretation function applications.

Exercise 8-18. Give a concrete syntax of **Logic** as defined in Figure 64, so that you can show denotations as decent logical formulas.

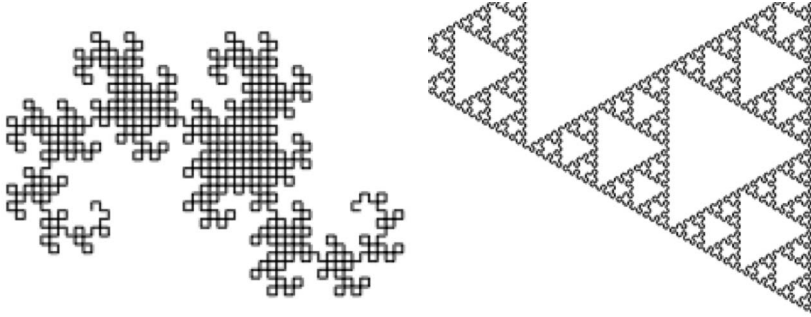
8.9 Grammars for fractals

Graftals are grammars used in computer graphics for generating recursive branching patterns, **fractals**. Graftals are not context-free, but easily definable in GF. Figure 66 shows two figures produced from the same abstract syntax tree: a dragon fractal and a Sierpinski triangle. It also shows the complete code and the concrete syntax producing Sierpinski triangles as PostScript code. The abstract syntax tree is

```
c (s (s (s (s (s (s (s (s (s z))))))))))
```

Exercise 8-19. What makes the Sierpinski grammar non-context-free?

Exercise 8-20.*⁺ Find out more about graftals and create your own ones in GF. The Wikipedia article <http://en.wikipedia.org/wiki/Graftal> is a good starting point.



```
-- (c) Krasimir Angelov 2009
abstract Graftal = {
  cat N; S;
  fun z : N ;
    s : N -> N ;
    c : N -> S ;
}

concrete Sierpinski of Graftal = {
  lincat N = {a : Str; b : Str} ;
  lincat S = {s : Str} ;

  lin z = {a = A; b = B} ;
  lin s x = {
    a = x.b ++ R ++ x.a ++ R ++ x.b ;
    b = x.a ++ L ++ x.b ++ L ++ x.a
  } ;
  lin c x = {s = "newpath 300 550 moveto" ++ x.a ++ "stroke showpage"} ;

  oper A : Str = "0 2 rlineto" ;
  oper B : Str = "0 2 rlineto" ;
  oper L : Str = "+60 rotate" ;
  oper R : Str = "-60 rotate" ;
}
```

FIGURE 66 Dragon and Sierpinski fractals and the grammar for Sierpinski.

Getting started with resource grammar programming

In this chapter, we will take the implementor’s perspective on resource grammar libraries. The primary purpose is to give an accessible introduction for those who want to write their own resource grammar implementations. But even those who are not doing this can read the chapter to gain insight into the linguistic concepts underlying the GF Resource Grammar Library. On a more general level, this chapter contains lots of examples of how linguistic problems are solved in the GF way by the use of parameters and data structures. The main contents are:

- key categories and functions: a miniature resource grammar
- principles of feature design for the key categories and functions
- an Italian implementation, shown in its entirety in Appendix A

9.1 Overview

In Chapter 5, we looked at the GF Resource Grammar Library from the user’s point of view. Appendix D completes that picture by showing the full set of categories and functions. But we have said very little about how the library is implemented. In Chapters 3 and 4, we have introduced all the GF machinery that is needed for this task. In this chapter, we will finally dive into the details.

We will first identify a **miniature resource grammar**, a small number of categories and functions that display the key grammatical structures in miniature. We will discuss the implementation issues for the miniature, both in a general, language-neutral way and by giving a detailed implementation of one language, Italian. The reader is encouraged, as a continuous exercise, to work out the corresponding details for some other language of her choice. The miniature resource has a

module structure similar to the full resource, and it is in fact sufficient for implementing the **Foods** grammar of Chapter 5, in the same way as the real resource.

When the miniature resource is completed, we will in the next chapter take a look at the full resource grammar, showing how its module structure grows out from the miniature and discussing what problems remain to be solved. We will also give statistics of the work effort behind the current library, and hands-on guidance on what steps are involved when adding a new language to the library. While application grammars can be written for new languages directly, without an intervening Resource Grammar Library, implementing the library is the best way to go in the long run—not only because it will serve an unlimited number of applications, but also because the existing library specification will help to identify the linguistic issues and avoid pitfalls.

9.2 The miniature resource grammar

The full Resource Grammar Library, as of Summer 2010, has around 80 categories and 200 abstract syntax functions. The miniature resource grammar, whose complete abstract syntax is shown in Section A.1, has 14 categories and 23 functions. Of these, we will focus on just 7 categories and 4 functions; the rest are included in order to make the grammar produce something useful that can be tested, but they are otherwise straightforward to implement when the really crucial ones are in place.

The key phrasal categories and functions should be familiar at this point, but let us recapitulate them, in Figure 67. What we haven't shown before are the internal names of the functions: as users of the resource, we have worked with the overloaded API operations.

To make the system work, we also need a handful of other categories:

- lexical categories: nouns (**N**), adjectives (**A**), verbs (**V** and **V2**) and adjectives (**AdA**; not very central, but included here to enable the implementation of **Foods**),
- the category (**S**) of sentences with fixed tense and polarity,
- the abstract feature categories tense (**Tense**) and polarity (**Po1**).

Likewise, we need some more functions:

- function **UseC1** that fixes tense and polarity,
- adjectival predication **CompAP**,
- adjectival modification **AdAP**,
- the lexical insertion functions **UseN**, **UseA**, and **UseV**,
- values of the abstract features,

cat	name	example
C1	clause	<i>she loves us</i>
VP	verb phrase	<i>loves us</i>
V2	two-place verb	<i>loves</i>
NP	noun phrase	<i>this man</i>
CN	common noun	<i>small man</i>
AP	adjectival phrase	<i>young</i>
Det	determiner	<i>this</i>

fun	type	construction	example
PredVP	NP -> VP -> C1	predication	<i>she loves us</i>
ComplV2	V2 -> NP -> VP	complementation	<i>loves us</i>
DetCN	Det -> CN -> NP	determination	<i>this man</i>
AdjCN	AP -> CN -> CN	modification	<i>small man</i>

FIGURE 67 The key phrasal categories and functions.

- a random selection of structural words, for illustration,
- constructs for coordination, to be discussed in Section 9.15.

The entire abstract syntax **Grammar** is shown in Section A.1. We will now start our work by some general principles and observations about the key categories and functions.

9.3 Feature design

To make a concrete syntax work, it is crucial to define the variable and inherent features of each category carefully. Fortunately, these features are quite uniform across languages. Figure 68 shows the way variable and inherent features typically attach to syntactic categories. Many languages may lack some of the features, for instance, gender; but if a language does have gender, it tends to be used in the way shown in Figure 68.

An important role in syntax is played by the so-called **agreement features**, which in predication are passed from the subject noun phrase to the verb phrase. Agreement features usually include gender, number, and person.

9.4 Predication

To implement the predication rule

```
fun PredVP : NP -> VP -> C1
```

let us just directly copy the features from Figure 68 into GF code. We don't assume any particular values of the features types yet, but just

cat	variable	inherent
Cl	tense	-
VP	tense, agreement	-
V2	tense, agreement	case
NP	case	agreement
CN	number, case	gender
AP	gender, number, case	-
Det	gender, case	number

FIGURE 68 A typical attachment of features to categories.

that they exist. Thus we just assume that there is a special case used for the subject; in many languages, this is called the nominative.

```

param
  Tense, Case, Agr
lincat
  Cl = {s : Tense      => Str      }
  NP = {s : Case      => Str ; a : Agr}
  VP = {s : Tense => Agr => Str      }
oper
  nom : Case

```

The predication function passes the tense of the clause to the VP. The variable agreement feature of the VP is received from the NP, and the case of the NP is set to the nominative. Thus

```

lin PredVP np vp =
  {s = \\t => np.s ! nom ++ vp.s ! t ! np.a}

```

Notice that in this rule, like in most combination rules in well-designed grammars, features are just *passed* from one part to the other, so that the table abstraction notation `\\p => t` can be used instead of case analysis (that is, instead of **table** expressions with several branches).

The two tables in Figure 69 illustrate predication in English and Italian. The rows show the agreement features of the subject that may affect the predicate. In English, only number and person are relevant, whereas Italian also has gender agreement.

Now, all languages show deviations from the schematic predication rule shown above. The **word order** may vary, as in Latin which often has the verb as the last word (*ego fessus sum* “I tired am”). This happens even in English, if the clause is used for questions (*am I tired*); in the resource grammar, sentential questions are built from **Cl** trees.

np.agr	present	perfect
Sg Per1	<i>I am tired</i>	<i>I have been tired</i>
Sg Per3	<i>she is tired</i>	<i>she has been tired</i>
Pl Per1	<i>we are tired</i>	<i>we have been tired</i>

np.agr	present	perfect
Masc Sg Per1	<i>io sono stanco</i>	<i>io sono stato stanco</i>
Fem Sg Per3	<i>lei è stanca</i>	<i>lei è stata stanca</i>
Fem Pl Per1	<i>noi siamo stanche</i>	<i>noi siamo stati stanchi</i>

FIGURE 69 The variation of PredVP i_NP (ComplAP (UseAP tired_A)) in English and Italian.

Another deviation is **pro-drop**, i.e. the omission of an unstressed subject pronoun. Thus Italian normally says just *sono stanco* instead of *io sono stanco*, if the subject is not emphasized.

Ergativity is the phenomenon where the subject, rather than the object, of a transitive verb has a specially marked case; the object has the same case as the subject of an intransitive verb. As an accompanying phenomenon, the rest of the verb phrase may then agree with the object rather than the subject. This happens in Hindi, in the perfective tense. The agreement-to-object phenomenon is even present in Italian: in the past compound tense (that we are calling the “perfect”), the participle agrees with the object if it is a so-called direct clitic object: *lei ci ha amati* (“she has loved us”), with *amati* (“loved”) in plural case inherited from *ci* (“us”).

Finally, the subject case need not always be the same, but vary according to the verb. This is one way to describe Finnish, where “I am a child” is expressed by *minä olen lapsi*, with the subject *minä* in the nominative case, and “I have a child” with *minulla on lapsi*, where the subject *minulla* is in the adessive. An alternative way to analyse this would be to treat *lapsi* (“a child”, in the nominative) as the subject of the sentence, but this would create some other problems, for instance, in reflexivization.

9.5 Complementation

Continuing under the guidance of Figure 68, we get the following linearization type for two-place verbs:

```
lincat V2 = {s : Tense => Agr => Str ; c : Case}
```

Complementation passes the variable tense and agreement features of the VP to the verb. The case of the complement NP is received from

the verb:

```
lin ComplV2 v np =  
  {s = \\t,a => v.s ! t ! a ++ np.s ! v.c}
```

The **Case** parameter, inherent for V2, has a rich variation in languages like Finnish, as shown in Figure 70. Cases are functionally similar to prepositions. If the preposition is not a part of the **Case** type (which is usually a possible GF implementation), it can be made into yet another field of V2 linearization:

```
lincat V2 =  
  {s : Tense => Agr => Str ; c : Case ; prep : Str}  
lin ComplV2 v2 vp =  
  {s = \\t,a => v2.s ! t ! a ++ v2.prep ++ np.s ! v2.c}
```

In many languages, V2 and other verb categories don't have variation in gender but only in number and person. But in some languages, e.g. Arabic, verbs do vary in gender and thus need the full array of agreement features.

v2.case	VP, infinitive
Acc	<i>love me</i>
at + Acc	<i>look at me</i>

v2.case	VP, infinitive	translation
Accusative	<i>tavata minut</i>	“meet me”
Partitive	<i>rakastaa minua</i>	“love me”
Elative	<i>pitää minusta</i>	“like me”
Genitive + <i>perään</i>	<i>katsoa minun perääni</i>	“look after me”

FIGURE 70 Complementation in English and Finnish with different cases and prepositions.

Clitics are a common variation to the complementation rule. They are complements that are put before their verbs, rather than after. Thus in Italian, unstressed pronouns may appear before verbs (*Maria mi ama*, “Mary me loves”), whereas other noun phrases come after (*Maria ama Giovanni*, “Mary loves John”).

9.6 Determination

Determination builds a noun phrase from a determiner and a common noun. The interplay of features is interesting. The NP needs to have both gender and number as agreement features. The CN has an inherent gender but a variable number. The Det has a variable gender but an

Det.num	NP		
Sg	<i>this house</i>		
Pl	<i>these houses</i>		

Det.num	CN.gen	NP	translation
Sg	Masc	<i>questo vino</i>	“this wine”
Sg	Fem	<i>questa casa</i>	“this house”
Pl	Masc	<i>questi vini</i>	“these wines”
Pl	Fem	<i>queste case</i>	“these houses”

Det.num	case	NP	translation
Sg	nominative	<i>tämä talo</i>	“this house”
Sg	genitive	<i>tämän talon</i>	“of this house”
Sg	inessive	<i>tässä talossa</i>	“in this house”
Pl	nominative	<i>nämä talot</i>	“these houses”
Pl	genitive	<i>näiden talojen</i>	“of these houses”
Pl	inessive	<i>näissä taloissa</i>	“in these houses”

FIGURE 71 Determination in English, Italian, and Finnish.

inherent number. The CN and the Det give their inherent features to each other, and the resulting NP inherits them both. The case is variable throughout the construction, and the person is the third.

```
lincat
  NP = {s :          Case => Str ; a : Agr   }
  CN  = {s : Number => Case => Str ; g : Gender}
  Det = {s : Gender => Case => Str ; n : Number}
lin DetCN det cn = {
  s = \\c => det.s ! cn.g ! c ++ cn.s ! det.n ! c ;
  a = agr cn.g det.n Per3
}
oper agr : Gender -> Number -> Person -> Agr
```

Figure 71 shows examples of determination in three languages, which have different sets of actualized features.

The inherent number of determiners is contested by the fact that many determiners have a systematic number variation, exemplified by *this-these* in English. For the definite article, there is just one form, *the*, but it is used for both the singular and the plural. Italian has separate singular and plural forms of the definite article, for instance, *il-i* in the masculine. As shown by Appendix D, this systematic variation is captured in the full resource grammar by the category **Quant**, which contains those determiners that occur in both numbers. In the

		singular	plural
		<i>new house</i>	<i>new houses</i>

cn.g	singular	plural	translation
Masc	<i>vino rosso</i>	<i>vini rossi</i>	“red wine”
Fem	<i>casa rossa</i>	<i>case rosse</i>	“red house”

sg, nominative	sg, ablative	pl, essive	translation
<i>iso talo</i>	<i>isolta talolta</i>	<i>isoina taloina</i>	“big house”

FIGURE 72 Adjectival modification in English, Italian, and Finnish.

miniature resource, however, we will not enter this detail.

Another common phenomenon with determiners is the absence of explicit determiner words. Thus Finnish has no explicit articles, but *talo* (“house”) expresses both “the house” and “a house”. In English, there is no word for the plural indefinite article (*houses*), whereas in French there is (*des maisons*, “houses”).

The “missing determiners” are often expressed by other means. Thus Swedish uses a morphological feature for definiteness: (*huset*, “the house”, is an inflectional form of *hus*, “house”). The definite form is used variably with other determiners as well, which can be expressed by using definiteness as an inherent parameter of determiners. Thus *varje* (“every”) takes the indefinite form (*varje hus*, “every house”) whereas *det här* (“this”) takes the definite form (*det här huset*, “this house”).

9.7 Modification

Adjectival modification is straightforward on the level of the predicted features: the inherent gender of CN is passed to the AP and inherited by the resulting construction; number and case are variable throughout:

```
lincat
  AP = {s : Gender => Number => Case => Str
        CN = {s :          Number => Case => Str ; g : Gender}
lin AdjCN ap cn = {
  s = \n,c => ap.s ! cn.g ! n ! c ++ cn.s ! n ! c ;
  g = cn.g
}
```

Figure 72 shows examples in different languages.

A common variation is the place of the adjective. Some languages have the adjective before the noun, some after. But this is seldom uniform in a language. Thus Italian places most adjectives after the noun

(*casa rossa*, “house red”), but some very common ones before (*vecchia casa*, “old house”). English places simple adjectives before the noun (*old house*) but complex adjectival phrases after (*house similar to this*). Finnish uses the pre-noun position even for complex adjectival phrases (*tätä muistuttava talo*, “to-this similar house”).

The definiteness parameter of nouns discussed in Section 9.6 is usually passed to adjectives as well. Thus in Swedish, *ett rött hus* (“a red house”) has both the adjective and the noun in the indefinite form, whereas *det röda huset* (“the red house”) has them in the definite form. German, which doesn’t inflect nouns for definiteness, still has a related distinction between strong and weak adjective declensions: *ein rotes Haus* (“a red house”, strong declension) vs. *das rote Haus* (“the red house”, weak declension).

9.8 Lexical insertion

To get started with the construction of phrases in each category, words from the lexicon are used. This is done by means of **lexical insertion functions**, one for each lexical category:

```
fun
  UseN : N -> CN
  UseA : A -> AP
  UseV : V -> VP
```

Their linearization rules are often trivial, because the linearization types match:

```
lin
  UseN n = n
  UseA a = a
  UseV v = v
```

However, for *UseV* in particular, the rule will often be more complex, because a VP record contains fields not present in V, such as objects and other complements.

The inserted word is called the **head** of the phrases built from it. Thus *house* is the head of *house*, *big house*, *big old house*, etc. The notion of head is useful because it comes with a powerful generalization:

- variable features are passed from the whole phrase to its head,
- inherent features of the head are inherited by the whole phrase.

As most other generalizations, this one has exceptions and modifications. But it works well for **endocentric** constructions: constructions that have a “nucleus” of the same (or nearly the same) category as the full phrase.

Lexical insertions are by definition endocentric, and so are complementation and modification. In contrast to this, predication and determination are **exocentric**: they have no part that has the same type as the whole. This can be seen by the **substitution test**: a part has the same type as the whole, if both can be used in the same contexts without loss of grammaticality. For instance, *house* and *big house* pass the substitution test but *house* and *the house* don't.

The names of the categories suggest that nouns (N) are heads of noun phrases (NP). However, in an NP of the form (DetCN det cn), neither of the parts can in general pass the substitution test. What is more, none of them alone has all the features needed for using an NP, as we saw in Section 9.6: the number comes from the determiner and the gender from the noun.

As we have seen, the phrase category really corresponding to N is not NP but CN. In many linguistic theories, this category is called N-bar, as a special case of using the name *X-bar* for phrasal categories with lexical heads in a category *X*; this idea is known as the **X-bar theory** (Jackendoff 1977).

There are, however, two lexical categories working as heads for noun phrases: pronouns (like *she*) and proper names (like *John*). We haven't distinguished them from NP in the miniature resource, but the full resource grammar does so. In addition, many determiners can be used alone as noun phrases, for instance *this*, as in the sentence *this is delicious*.

9.9 The miniature resource in Italian

The miniature resource consists of the following modules:

- `abstract Grammar`, the core syntax
- `abstract Test`, a test lexicon
- `interface Syntax`, the syntax API
- `concrete GrammarIta`, the core syntax in Italian
- `concrete TestIta`, the test lexicon in Italian
- `instance SyntaxIta`, the syntax API for Italian
- `resource ParadigmsIta`, the Italian paradigms API
- `resource ResIta`, auxiliary operations for Italian

The full code in these modules is given in Appendix A. It is around 400 lines of GF code, of which 300 lines are Italian-specific. Figure 73 shows the dependencies between the modules.

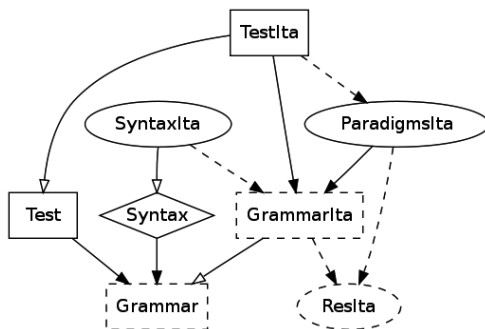


FIGURE 73 The miniature resource module dependencies.

9.10 Implementing morphology

Usually the best way to start resource grammar writing is with the morphology. Compared with syntax, this is a rather concrete task that gives visible results quickly. Moreover, in order to get visible results in syntax, some morphology is needed anyway. And thinking about morphology is an efficient way to find out what parameter types a language has.

We define the parameters in the beginning of `ResIta.gf`, in Section A.2. Nouns and adjectives, defined later in the same module, follow the general pattern given in Section 9.3. The only addition is the boolean parameter in adjectives, indicating whether the adjective is prefixed or not; cf. Section 9.7. Thus

```
lincat A = {s : Gender => Number => Str ; isPre : Bool}
```

In the module `GrammarIta`, this definition is given via a type synonym `Adj` defined in `ResIta`. The use of this synonym helps us to keep the `lincat` definition in synchrony with what is defined for adjectives in morphology.

The noun and adjective morphology itself is slightly complex but straightforward. Italian doesn't have purely suffixable number and case endings; instead, the final vowels are changed. The changes are predictable from the last vowel, sometimes together with the foregoing character. The gender of nouns is predictable for nouns ending with *a* (feminine) or *o* (masculine); for nouns ending with *e*, the smart paradigm `regNoun` uses the masculine as default.

For verbs, the miniature resource has a simplified parameter type:

```
param VForm =
  VInf | VPres Number Person | VPart Gender Number
```

arrive_V	form
VInf	<i>arrivare</i>
VPres Sg Per1	<i>arrivo</i>
VPres Sg Per2	<i>arrivi</i>
VPres Sg Per3	<i>arriva</i>
VPres Pl Per1	<i>arriviamo</i>
VPres Pl Per2	<i>arrivate</i>
VPres Pl Per3	<i>arrivano</i>
VPart Masc Sg	<i>arrivato</i>
VPart Fem Sg	<i>arrivata</i>
VPart Masc Pl	<i>arrivati</i>
VPart Fem Pl	<i>arrivate</i>

FIGURE 74 The miniature inflection table for the Italian verb *arrivare*.

The full Italian verb conjugation has 56 forms, but in the miniature we need just 11. Figure 74 shows these forms for the regular verb *arrivare* (“arrive”). The forms can be produced by the operation **regVerb** in **ResIta**, which recognizes the infinitive ending *are* to produce the so-called first conjugation (and likewise, *ere* for the second conjugation and *ire* for the third). Irregular verbs, such as *piacere* in Section A.5, are produced by using the worst-case function **mkVerb**, which takes eight arguments; the four forms of the participle can always be produced by the regular adjective paradigm.

In addition to the conjugation, Italian verbs have an inherent feature indicating which **auxiliary verb** is used for compound tenses, such as the perfect (**Perf**) in the miniature resource. We thus have

```
lincat V = {s : VForm => Str ; aux : Aux}
```

where **Aux** is a parameter type with two values. The default auxiliary is *avere* (“have”), but some verbs use *essere* (“be”). In **ParadigmsIta**, the choice of the auxiliary is made by a separate operation

```
essereV : V -> V
```

which changes the auxiliary of a verb to *essere* for verbs like *arrivare* (“arrive”).

9.11 Implementing modification and determination

The adjectival modification rule is straightforward, because the features match as expected. The inherent **isPre** feature is used for deciding the order between the noun and the adjective. The **PreIude** operation **preOrPost**, which can be defined

```
oper preOrPost : Bool -> Str -> Str = \p,a,b ->
  case p of {
    True  => a ++ b ;
    False => b ++ a
  }
```

is used for this task as follows:

```
lin AdjCN cn ap = {
  s = \\n =>
    preOrPost ap.isPre (ap.s ! cn.g ! n) (cn.s ! n) ;
  g = cn.g
}
```

But NP is a category that requires some thought. First of all, even though Italian common nouns don't have different cases, some noun phrases do. Thus personal pronouns have separate nominative, accusative, and dative cases, for instance *lei-la-le* ("she-her-to her"). Since the NP category includes personal pronouns, we have to introduce a variable case for it. As we will see later in this section, this decision also helps us with the syntax of the definite article.

Another problem with NP is that some forms of pronouns are clitic, i.e. appear before the verb when used as objects. Being a clitic depends on the case parameter. The way we implement this in **ResIta** is by using two separate fields in the NP: a "clitic" field and an "object" field. In predication, as we will see in Section 9.12, the clitic field is placed before the verb and the object field after the verb. Thus:

```
lincat NP = {s : Case => {clit,obj : Str} ; a : Agr}
```

A pronoun can thus be implemented as follows:

```
lin she_NP = {
  s = table {
    Nom => {clit = [] ; obj = "lei"} ;
    Acc => {clit = "la" ; obj = []} ;
    Dat => {clit = "le" ; obj = []}
  } ;
  a = Ag Fem Sg Per3
}
```

Instead of repeating this for all pronouns, **ResIta** defines the operation **pronNP**, which enables us to write the rules for all pronouns in a concise and uniform way. Other noun phrases than pronouns are uniformly linearized without using clitics:

```
lin John_NP = {
```



```

s = table {
  Nom | Acc => {clit = [] ; obj = "Giovanni"} ;
  Dat      => {clit = [] ; obj = "a Giovanni"}
} ;
a = Ag Fem Sg Per3
}

```

As an alternative to the separate clitic and object fields, a boolean feature similar to the one in adjectives could be used:

```
NP = {s : Case => {s : Str ; isPre : Bool} ; a : Agr}
```

However, in our experience separate fields are easier to use when there are more than two clitic positions (as in the full Italian resource). Now, the actual NP type in *ResIta* does contain a Boolean feature *isClit*, but this feature is used only for deciding where agreement with the clitic is possible; see Section 9.12.

Determination presents no surprises: it comes out as another instance of uncliticized noun phrases,

```

lin DetCN det cn = {
  s = \\c => {
    obj = det.s ! cn.g ! c ++ cn.s ! det.n ;
    clit = []
  } ;
  a = Ag cn.g det.n Per3
}

```

Determiners like *questo* (“this”) can usually be built by adjective paradigms. Their case inflection is uniformly provided by the operation *prepCase*, which produces the preposition *a* in the dative and [] for the other cases:

```

oper adjDet : Adj -> Number -> Determiner = \adj,n -> {
  s = \\g,c => prepCase c ++ adj.s ! g ! n ;
  n = n
}

```

The definite article is more complex, because of its prefix-dependent forms. The masculine singular article, for instance, is *l’* before vowels (*l’uomo* “the man”), *lo* before *z* and various combinations with *s* (*lo zio* “the uncle”; known as *s impuro*, “impure *s*”), and finally *il* as the default case (*il ragazzo* “the boy”). This three-way variation is so common that *ResIta* introduces a special operation for it:

```

elisForms : (_,_,_ : Str) -> Str = \lo,l',il ->
  pre {#s_impuro => lo ; #vowel => l' ; _ => il} ;

```

The pattern macros `s_impuro` and `vowel` are defined in `ResIta` in Appendix A.2.

The preposition *a* has special prefix-dependent contractions with the definite article, which can be solved by using `elisForms` separately in the dative case. Thus we arrive at the following pattern in the masculine case:

```
table {
  Nom | Acc => elisForms "lo" "l'" "il" ;
  Dat      => elisForms "allo" "all'" "al"
}
```

The full inflection can be seen in the `the_Det` rule in Section A.3.

9.12 Implementing verb phrases and complementation

The definition of the `VP` type is perhaps the most difficult task in resource grammar programming. There are considerations pulling to two directions. First, the general considerations in Section 9.3 suggest that `VP` have variable tense and agreement features. Then it will be close to the `C1` type (in a sense, like `C1` without a subject)

```
lincat VP = {s : Tense => Agr => Str}
```

and predication is easy to implement:

```
lin PredVP np vp =
  {s = \\t => np.s ! Nom ++ vp.s ! t ! np.a}
```

On the other hand, lexical insertion has to map `V` to `VP`. Since verbs in morphology don't have straightforward `Tense` and `Agr` paradigms but something more complex that combines the two (such as `VForm` in Italian), a natural `VP` type just adds a clitic and an object field to a verb field:

```
lincat VP = {v : Verb ; clit : Str ; obj : Str}
```

We have chosen the latter alternative. Lexical insertion is hence trivial: just add empty clitic and object fields,

```
lin UseV v = {v = v ; clit, obj = []}
```

The complementation rule uses the clitic and object fields of the `NP`, selected by using the inherent case of the `V2`:

```
lin ComplV2 np = let nps = np.s ! v2.c in
  {v = v2 ; clit = nps.clit ; obj = nps.obj}
```

Notice that the `V2` itself works as the verb field, due to record subtyping (see Section 4.12).

In the real code in A.2, the type `VP` is slightly more complicated than shown above. First, the `obj` field is made dependent on `Agr`, so that it can store adjectival predicates. Secondly, a field `clitAgr` is added for reasons explained in the next section.

9.13 Implementing predication

In the previous section, we chose to implement `VP` so that it is close to `V` rather than to `Cl`. This gave us easy lexical insertion but more work in predication. In Italian, there are two crucial things: find the correct places of the object and the clitic, and select the correct verb form.

Placing the object and the clitic is easy, due to our design of the `NP` type in Section 9.11. It remains to select the verb form, as the function of the subject (agreement features) and the tense. But this is not yet enough, since the verb form may *also* depend on the clitic object, as seen in Section 9.4. The informal rules are the following:

1. In the present tense, the verb form depends only on the number and the person of the subject.
2. In the perfect tense,
 - (a) there is an auxiliary verb, whose form depends on the subject in the same way as the main verb in the present tense;
 - (b) the main verb is in the participle, whose gender and number
 - i. come from the subject, if the auxiliary is *essere* (*lei è arrivata* “she has arrived”, *noi siamo arrivati* “we have arrived”);
 - ii. come from the clitic, if there is one and it is in the accusative (*lei ci ha trovati* “she has found us”);
 - iii. are masculine and singular in all other cases (*noi abbiamo trovato Maria* “we have found Maria”).

The operation `argV` in `ResIta` takes care of the simple cases 1 and 2(a). The operation `AgrPart` takes care of 2(b). It uses an extra feature, `ClitAgr`, which is defined as “either agreement or not”,

```
param ClitAgr = CAgr Agr | CAgrNo
```

A feature of this type is stored in the `VP`, and it is set whenever a complement is added, thus in the linearization rule of `CompLV2` in `GrammarIta`. There the definition of this field just checks if there is a clitic and if the case is accusative:

```
clitAgr = case <nps.isClit,v2.c> of {
  <True,Acc> => CAgr np.a ;
  _ => CAgrNo
}
```

With this machinery in place, we can finally implement predication:

```
lin PredVP np vp =
  let
    subj = (np.s ! Nom).obj ;
    obj  = vp.obj ! np.a ;
    clit = vp.clit ;
    verb = table {
      Pres => agrV vp.v np.a ;
      Perf => agrV (auxVerb vp.v.aux) np.a ++
                 agrPart vp.v np.a vp.clitAgr
    }
  in {s = \\t => subj ++ clit ++ verb ! t ++ obj}
```

This rule displays one more aspect that we haven't shown yet: the dependence of the object on the agreement feature of the subject. This is needed in adjectival predication, since the adjective is stored in the `obj` field, as we will show in the next section. In the full resource grammar, even the `clit` field must be made subject-dependent, because reflexive verbs introduce clitics that depend on the subject (*io mi figuro* “I imagine”, *tu ti figuri* “you imagine”).

9.14 Implementing the rest

Most of the contents of Appendix A should now be understandable, at least to readers who try to reproduce the implementation for some other language and thereby have to experiment on a computer. As for the syntax rules, the treatment of tenses may raise a question: why do the abstract features have a field for a string, which then is always empty?

```
lincat Tense = {s : Str ; t : ResIta.Tense} ;
lin Pres = {s = [] ; t = ResIta.Pres} ;
lin Perf = {s = [] ; t = ResIta.Perf} ;
```

(Notice the qualified names `ResIta.Tense` etc, to avoid clashes with the same names introduced in `GrammarIta`.) In the `UseCl` rule, which fixes the tense of a sentence, the dummy `t.s` field is prefixed to the clause:

```
lin UseCl t cl = {s = t.s ++ cl.s ! t.t}
```

There is no case even in the full Italian grammar where the `s` field of `Tense` has any content. The reason to include it is in the parsing algorithm, which would otherwise believe that the `t` argument is suppressed, and introduce a metavariable! The presence of a string, even if empty, prevents this. This is a trick needed in GF programming that

may be hard to figure out without experience.

The morphology and syntax API in Appendix A are defined following the principles explained in Chapter 5.14. Some more smartness could easily be included in the inflection paradigms, especially in those for verbs.

At this point, if you haven't done it yet, it is warmly recommended that you follow the model of Appendix A and implement the miniature resource for some other language. It can very well be a language already included in the resource library. Even stealing code from the existing source files and adapting it to the miniature format will certainly be useful! Many of the languages have implementations with some parts dating back to the 20th century, and the structure is likely to be less nice and clear than in the miniature.

The miniature is also a recommended starting point for anyone aiming to implement the full resource. The work will not be wasted, but can be smoothly embedded as a part of the full grammar. The details are shown in Section 10.4.

Exercise 9-1. Modify the pronoun rules in such a way that the pro-drop property is implemented, i.e. that subject pronouns (which you can always assume to be unstressed) are not shown. How does this affect parsing?

9.15 Coordination and extraction

The key phenomena we identified in Section 9.2 did not cover two of the traditional touchstones of natural language syntax:

- **Coordination**, the combination of phrases of different categories with conjunctions, e.g. *John and Mary*.
- **Extraction**, the “movement” of relative and interrogative pronouns from their “normal” places in predication, e.g. *(the woman) whom John loves; whom does John love*.

The reason why we didn't discuss them is that they have not created any particular problems in the resource grammar libraries. This has to do with two properties of GF. One of these properties is, of course, the very division between abstract and concrete syntax. Thus there is no such thing as the “normal” place of a constituent from which it has to be “moved”. The other property is that GF is a *programming language*, in which problems can be solved by “brute force”, using data structures and algorithms. Moreover, since GF is a functional language, even “brute force” solutions can often be made simple and elegant.

As an example of coordination, let us add the binary noun phrase coordination (as in *you and me*) to the miniature resource. The main problem to solve is the distribution of agreement features. Coordination is endocentric in the sense that it contains an argument of the same type as the value—but the problem is that there are *two* such arguments, which can have different features. None of the arguments is clearly a head of the coordinated phrase. However, the grammatical rules can be made clear:

- the gender is feminine, if and only if both noun phrases are feminine;
- as for the number, with *and* (*e* in Italian), the result is always plural; with *or* (*o*), the result is singular if and only if both are singular;
- as for the person, the “lowest number” wins.

This set of rules is very much like a program, and it is indeed easy to write by using **case** expressions. Mathematically, the calculations for gender, number, and person are similar to the minimum function, with **Masc**, **Pl**, and **Per1**, respectively, as the smallest elements. The actual rules may well be slightly more complicated—for instance, that the choice of person depends on number—but such rules can likewise be written with **case** expressions.

For extraction, the resource grammar uses the idea of **slash categories** familiar from the GPSG grammar formalism (Gazdar & al. 1985). An example is the category **CISlash** (**C1/NP** in GPSG notation), which is “a clause lacking a noun phrase”. The linearization type of a slash category **CSlash** is

$$C \text{ ** } \{c : \text{Case}\}$$

where the inherent case **c** is used when, for instance, a relative pronoun “unslashes” a **CISlash**. The different combinations with slashes can be seen in the Resource API (Appendix D). The linguistic details are discussed in a survey article about the Resource Grammar Library (Ranta 2009).

Exercise 9-2.* From the semantic point of view, a category **CSlash** is similar to the function type $NP \rightarrow C$. But using this type in the resource grammar would involve higher-order abstract syntax (Section 6.9) and hence have a linearization type that would not really do the job required. But you can try out this solution and see how far you can get in implementing phrases like *whom does John love*.

Exercise 9-3.*+ The NP coordination rule in Section A.3 doesn’t treat clitics properly: if coordinated, clitics should become stressed pronouns that appear in the same object position as other noun phrases. For

instance, from *io ti amo* (“I love you”) and *io amo Maria* (“I love Maria”) we should get *io amo te e Maria* (“I love you and Maria”). Getting this right requires a distinction between stressed and unstressed pronouns. If you are familiar with this topic, you can try to fix the rules; else take a look at the actual GF resource grammar sources.

Exercise 9-4.*⁺ Implement the miniature resource grammar for a language of your choice. If it is a language already included in the library, compare its linearizations with those of the library.

This is a major assignment, which may take one or two weeks, and provides a good start for a full-scale resource grammar implementation.

Extending the Resource Grammar Library

This chapter gives practical advice on how to add a new language to the Resource Grammar Library. The focus is on the module structure, workflow, and evaluation; the exact details about each category and function can be found in the actual abstract syntax sources. The main contents are:

- the module structure
- statistics on the required effort
- the steps for adding a new language
- testing and evaluation
- non-ASCII alphabets
- the use of functors
- extending a resource grammar for wide-coverage parsing
- bootstrapping a big lexicon

10.1 The module structure of a resource grammar

The main modules and their dependencies are given in Figure 75. This figure can be compared with Figure 73, which shows the miniature resource. The most important change is that the **Grammar** module of the miniature has been split to 16 different modules. One of them has the name **Grammar**, which is really just a collection of the rest of them; see Figure 75. For simplicity, the figure shows only the abstract syntax modules; each of them has a corresponding concrete syntax module in every language.

The graphics of the figure carry the following information:

- solid contours: visible module, including the API
- dashed contours: internal module

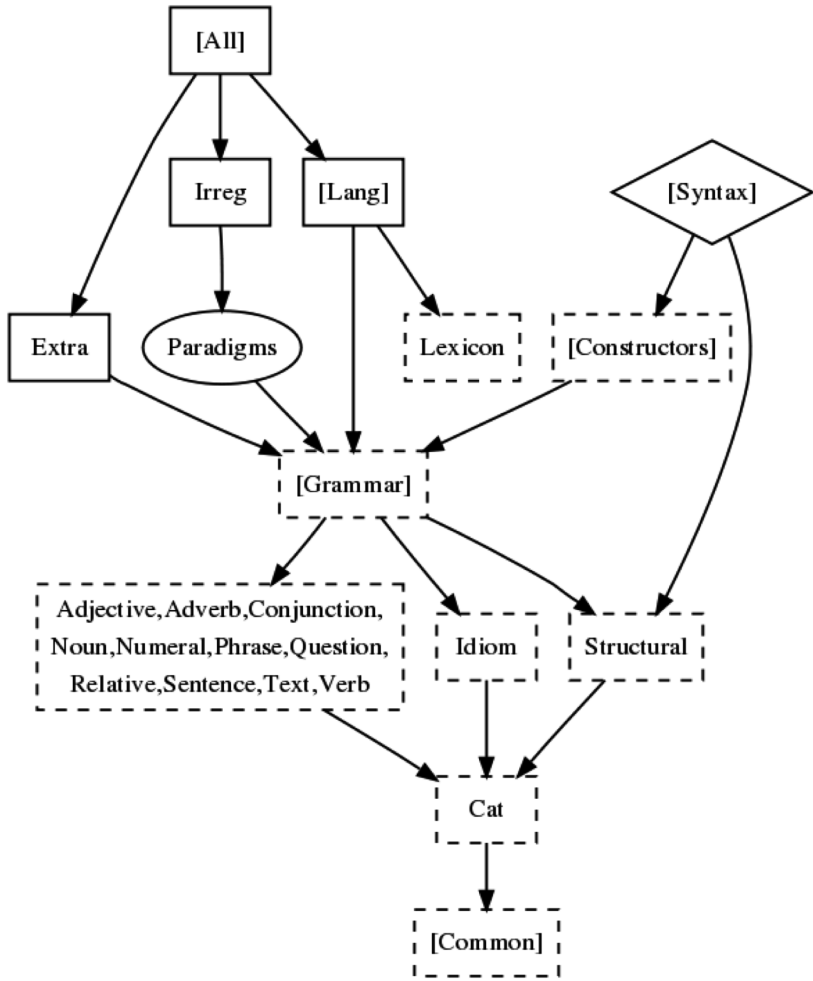


FIGURE 75 The main modules of a resource grammar.

- rectangle: abstract+concrete pair
- ellipse: resource or instance
- diamond: interface
- name in brackets: already given or mechanical to produce

The grammarian for a new language doesn't need to put effort on those modules whose names are in brackets: some of them are already there, whereas some need to be copied with new language names. More precisely,

- given: all abstract modules except **Extra**, **Irreg**
- mechanical: concrete of **Common**, **Grammar**, **Lang**, **All**
- mechanical: resources **Constructors**, **Syntax**, **Try**

The real work is thus to write

- concrete syntaxes of the row from **Adjective** to **Structural**
- concrete syntaxes of **Cat** and **Lexicon**
- the resource module **Paradigms**
- abstract and concrete of **Extra** and **Irreg**

Here is a summary of the roles of the modules. First the API:

- **Syntax**: syntactic combinations and structural words
- **Paradigms**: morphological paradigms
- **Constructors**: syntactic combinations only
- **Irreg**: irregularly inflected words (often mostly verbs)
- **Extra**: language-specific extra syntactic constructs

Then the top-level grammar with abstract+concrete pairs:

- **Lang**: common syntax and lexicon
- **All**: common grammar plus language-dependent extensions
- **Grammar**: abstract syntax functions common to languages
- **Idion**: idiomatic expressions
- **Structural**: lexicon of structural words
- **Lexicon**: test lexicon of 350 content words
- **Cat**: the type system common to languages
- **Common**: concrete syntax common to (most) languages

The most substantial part is the box containing the ten modules from **Adjective** to **Verb**. These modules are called **phrase category modules**. Each of them defines the constructors for one or a few related parts of speech, as summarized in Figure 76.

In addition to the modules shown in Figure 75, the resource grammarian will use a number of auxiliary modules. Some of these are already given in the library:

module	scope	value categories
Adjective	adjectives	AP
Adverb	adverbial phrases	AdN, Adv
Conjunction	coordination	Adv, AP, NP, RS, S
Noun	noun phrases, nouns	Card,CN,Det,NP,Num,Ord
Numeral	cardinals, ordinals	Dig, Digits, Numeral
Phrase	suprasentential units	PConj, Phr, Utt, Voc
Question	questions	IAdv, IComp, IDet, IP, QCl
Relative	relative clauses	RCl, RP
Sentence	clauses and sentences	Cl, Imp, QS, RS, S, SC, SSslash
Text	many-phrase texts	Text
Verb	verb phrases	Comp, VP, VPSlash

FIGURE 76 The roles of the phrase category modules.

- **Prelude** and **Predef**: string operations, booleans
- **Coordination**: generic operations for list conjunctions
- **ParamX**: commonly used parameters, such as **Number** = Sg | Pl

The following are written by the grammarian herself:

- **Res**: language specific parameter types, morphology, VP formation
- **Morpho**, **Phono**: auxiliary modules taken apart from **Res** if desired, for instance when the morphological rules or the phonological changes are substantial in a language.

10.2 Effort statistics

Figure 77 shows some statistics of the hitherto completed languages of the Resource Grammar Library. The figures are from Summer 2010; the person months are very rough estimates. The asterisk marks line counts that contains generated code; the real hand-written source is much smaller in those cases.

As the library has stabilized, experience has grown, and GF implementation itself has improved, the effort needed to implement a new language has gone down. In the recent and future projects, the interval of 3 to 6 person months seems sufficient for a new resource grammar implementation. The Bulgarian, Catalan, Polish, Romanian, Russian, and Urdu grammars were implemented by native speakers of their languages. The other grammars were mostly implemented by the author of this book, a native speaker of Finnish, but with contributions and bug fixes from native speaker programmers and informants.

In addition to the core resource code included in the table, there are large morphological dictionaries for (at the time of writing) Bulgarian,

language	syntax	morpho	lex	total	pms	start
<i>common</i>	413	-	-	413	2	2001
<i>abstract</i>	729	-	468	1197	24	2001
Bulgarian	1200	2329	502	4031	3	2008
Dutch	1077	1244	876	3197	1	2009
English	1025	772	506	2303	6	2001
Finnish	1471	1490	703	3664	6	2003
German	1337	604	492	2433	6	2002
Polish	1935	5152	507	8424	6	2007
Romanian	1713	3513	666	5892	3	2009
Russian	1492	3668	534	5694	18	2002
Urdu	903	1257	497	2657	4	2009
<i>Romance</i>	1346	-	-	1346	10	2003
Catalan	521	*9000	518	*10039	4	2006
French	468	1789	514	2771	6	2002
Italian	423	*7423	500	*8346	3	2003
Spanish	417	*6549	516	*7482	3	2004
<i>Scandinavian</i>	1293	-	-	1293	4	2005
Danish	262	683	486	1431	2	2005
Norwegian	281	676	488	1445	2	2005
Swedish	280	717	491	1488	4	2001
total	18173	*46866	9264	*74303	117	2001

FIGURE 77 Resource grammar effort in lines of source code and person months; (*) means that some of the code is generated.

English, Finnish, Swedish, and Turkish. These dictionaries together contain hundreds of thousands of lines of GF code, mostly machine-generated from available sources (see Section 10.10 for how to do this).

10.3 Workflow for a new language

Here is a summary of the steps to be taken when implementing a new language. We use Marathi as a place holder: you should replace it with your own language.

1. Fetch the latest GF sources from the developers' repository (see GF homepage).
2. Create a directory `GF/lib/src/marathi`.
3. Check out the ISO 639-3 language code: `Mar`.
4. Copy over the files from the closest related language, e.g. `hindi`.
5. Rename files `marathi/*Hin.gf` to `marathi/*Mar.gf`.
6. In module headers, replace references to `Hin` modules by references to `Mar` modules,
7. In each module, comment out every line in the module body.
8. Now you can import your (empty) grammar in GF: `import marathi/LangMar.gf`.

The rest is debugging the empty grammar! In this task, there are two possible ways to start:

- **Syntax-driven:** build the miniature resource first, then move the definitions to the full resource as described in Section 10.4.
- **Morphology-driven:** with the goal of defining `LexiconMar` by the use of smart paradigms, complete `ParadigmsMar` and, while doing this, those parts of `ResMar` and `CatMar` that are needed.

The syntax-driven approach is top-down in the sense that it starts from the largest units of language (sentences) and gives, from the beginning, a total picture of the language—how everything fits together. The morphology-driven approach is bottom-up, as it starts from the smallest units of language (the words) before going to larger units. An advantage with the morphology-driven approach is that the morphology alone is a useful linguistic resource, which has a permanent value even if the resource syntax is not finished. The syntax-driven approach has the advantage that it involves more “thinking” and less attention on particular details. Both approaches have been used successfully, and you can make the choice depend on your personal preferences.

At some point of the work, you should send your code to the public GF code repository. You shouldn't wait until it is completely finished

and polished, but exploit the possibility to get feedback and contributions from the open-source resource grammar community!

10.4 Reusing code from the miniature resource

If you have started with the miniature resource grammar (previous chapter), you will have to reorganize the code a bit. The miniature is not just smaller, but its structure is simpler. We have already seen what this means for the module structure. Another simplification is that a couple of shortcuts are made in the miniature, replacing more primitive concepts of the full resource. These concepts permit a more compact representation of the different combinatorial possibilities, but they are often more theoretical and abstract than the ones used in the miniature; they are also often hidden from the resource API users by flattening, as explained in Section 5.16. What distinguishes the miniature from the full resource is that

- VP's with complements are built via an intervening category `VPSlash`, which supports extraction (Section 9.15)
- VP's with a copula and a non-verbal category are built via an intervening category `Comp`
- Det's are often built from the category `Quant` of variable-number quantifiers together with a number `Num`
- NP doesn't contain personal pronouns directly, but as expressions of the category `Pron`

In more detail, the contents of the `Grammar` module are divided as follows:

- all `cat` definitions go to `Cat`
- the categories `Comp`, `Num`, `Pron`, `Quant`, and `VPSlash` are added
- `UseC1` and `PredVP` go to `Sentence`.
- `ComplV2` is split to `SlashV2` and `CompSlash`, both defined in `Verb`
- `DetCN` and `AdjCN` go to `Noun`
- `CompAP` is split to `UseComp` and `CompAP`, both defined in `Verb`
- `AdAP` goes to `Adjective`
- `a_Det` is defined as `IndefArt` and `the_Det` as `DefArt` in `Noun`
- `DetQuant`, `NumSg`, and `NumPl` are defined in `Noun`, to enable the use of `Quant` as `Det`
- `this_Det` and `these_Det` are merged to `this_Quant` in `Structural`
- `that_Det` and `those_Det` are merged to `that_Quant` in `Structural`
- `i_NP`, `she_NP`, and `we_NP` are moved to `Pron` constants in `Structural`
- `UsePron` is defined in `Noun`, to enable the use of `Pron` as NP

10.5 The development-test cycle

In the process of “debugging the empty grammar”, make sure you have a compilable **LangMar** at all times! Then you can use the GF shell as a diagnostic tool. The command

```
> print_gammar -missing
```

tells you which functions are not implemented yet.

As for the linguistic correctness of your code, random generation is a useful tool. It should be used on all levels, not just the top level. Thus

```
> gr -cat=C | l -table
```

should be used to test each category **C** separately. Even more accurate tests using trees with metavariables may be appropriate; see Section 2.10.

A test suite should be created for systematic **regression testing**, to make sure that bug fixes don’t generate new bugs. You should build and maintain a **treebank** as a test suite: a set of trees with their linearizations. This is how a treebank can be produced and used:

1. Create a file **test.trees** with just trees, one by line.
2. Linearize each tree to all forms, possibly with English for comparison.

```
> i english/LangEng.gf
> i marathi/LangMar.gf
> rf -lines -tree -file=test.trees |
  l -all -treebank | wf test.treebank
```

3. Create a **gold standard** **gold.treebank** from **test.treebank** by manually correcting the Marathi linearizations.
4. Compare with the Unix command

```
$ diff test.treebank gold.treebank
```

5. Rerun (2.) and (4.) after every change in concrete syntax.
6. Extend the tree set and the gold standard after every new implemented function.

The easiest way to create the initial set of trees is by parsing a set of sentences, written e.g. in English. If **test.txt** is such a file, a set of trees can be produced by

```
> rf -lines -file=test.txt | parse | wf -file=test.trees
```

The GF library sources actually contain a test suite, which can be used as a starting point for your language as well.

10.6 Non-ASCII alphabets and transliterations

An often forthcoming practical question is character encoding. GF uses internally the 32-bit unicode standard, and generates files (`.gfo`, `.pgf`) with the UTF-8 encoding. But `.gf` source files can use whatever encoding you want, indicated by a flag. If you use UTF-8 in your sources, each module body with string literals should contain the line

```
flags coding = utf8 ;
```

Now, while UTF-8 is more and more common and supported by tools such as text editors, it may still be difficult to get full support for it. For instance, the Windows shell doesn't support it, and the Emacs editor may have difficulties in displaying Unicode characters. The input of these characters can also be time-consuming, since you may have to use a virtual keyboard with an unfamiliar layout, or even raw hex-codes.

Transliterations are a traditional solution to the character set problem, and GF has some built-in support for them. There is a set of ready-made transliterations, which you can use. The command `ut=unicode_table` displays transliteration tables. You can use

```
> help unicode_table
```

to get a list of the available transliterations and how to view them.

If you need to define a transliteration of your own, you will have to edit the GF source file `GF/Text/Transliterations.hs` and recompile GF. Here is an example of how a transliteration is defined:

```
transHebrew :: Transliteration
transHebrew = mkTransliteration allTrans allCodes where
  allTrans = words $
    "A b g d h w z H T y K k l M m N " ++
    "n S O P p Z. Z q r s t - - - - " ++
    "w2 w3 y2 g1 g2"
  allCodes = [0x05d0..0x05f4]
```

Each transliteration a letter followed by any number (including 0) of non-letter characters. Capital and small letters are treated as distinct.

Strings in the GF shell and in GF source files can be converted in both directions between UTF-8 and transliterations, by using the `ps = put_string` command. For instance,

```
> put_string -to_hebrew "Obryt"
```

produces the Hebrew word for “Hebrew” in UTF-8. If you want to convert a whole source file `Heb.gf` written in the transliteration to real Hebrew UTF-8, you can write

```
> read_file -file=Heb.gf | ps -env=quotes -to_hebrew
```


The flag `-env=quotes` says that only string literals are converted, not the rest of the file, which would of course be disastrous. The addition of a transliteration in the file `Transliterations.hs` is sufficient for updating all the commands involving transliterations, including the `help` commands that show the available transliterations. Please contribute your new transliteration to the GF community!

10.7 Coding discipline

Resource grammar programming is a complex and demanding programming effort, both as a GF task and as a programming project in general. GF's module and type system do their best to help in this task, but on top of them, an informal discipline is needed as well. Let us discuss two aspects of this: type discipline and abstraction.

First, **type discipline**. The `Grammar` module in Figure 75 is just a proxy for the 13 modules that it extends. These 13 are separated in order to divide the otherwise too heavy task of implementing the whole syntax. This division makes the compilation of the grammar faster, because each module is compiled separately, and in a rapid development-test loop only those modules need to be recompiled that have been changed after the previous compilation. Moreover, this module structure makes it possible to divide the labour between different grammarians working on different categories.

Each phrase structure module in Figure 76 is actually a **producer** for its value categories: it is here, and only here (except possibly in `Structural`, `Conjunction`, `Extra`, and `Idiom`), that the rules producing each category are implemented.

At the same time, every module is potentially a **consumer** for any category, in the sense that it may use that category as an argument type. The producer-consumer dependencies can be mutual, as is seen from the following configuration:

```
Sentence.UseCl   : Temp -> Pol -> Cl -> S   -- S uses Cl
Sentence.PredVP  : VP -> NP -> Cl           -- Cl uses VP
Verb.ComplVS     : VS -> S -> VP           -- VP uses S
```

How can this be made to work? How can the producers know what the consumers need, and how can the consumers be sure that they get what they need? The solution to this is to set up a **contract** between producers and consumers, which is the system of linearization types defined *below* the phrase category level, in the module `Cat`. Since all other modules inherit their type system from `Cat`, they can be sure that they are working on objects of the same types.

In addition to the type discipline, resource grammarians should

strive after **abstraction** and follow the **functional programming style**, with its golden rule,

Whenever you find yourself programming by copy and paste, write a function instead!

Now, writing functions, just like copying and pasting, is a way to avoid **repetition**. Writing functions when there is *no* repetition can lead to over-engineering and code that is unnecessarily hard to understand. The definition of a function is always at some distance from the places where the function is used, and having to look up a definition slows down the process of reading the code. Thus we would primarily encourage writing functions where they are actually needed. This means that they should be responses to observed repetition, or, at least, measures preventing expected repetition.

Repetition may occur on different levels of code, and should be met on a level as local as possible, to enhance the readability and maintainability of the code. This suggests the following recipe:

- repetition inside one definition: use a **let** expression
- repetition inside one module: define an **oper** in the same module
- repetition in many modules: define an **oper** in the **Res** module
- repetition of an entire module: write a functor

The miniature resource shows examples of the first three kinds. Thus, for instance, **CompLV2** in **GrammarIta** in Section A.3 has a **let** expression defining a constant **nps**, which is then used twice. The same module has an **oper** **quello.A**, which is likewise used twice in the module. **ResIta** in Section A.2 has several **opers** that are used throughout the code. Many more distributed uses arise in the full resource grammar, where **Grammar** is divided into 13 modules.

10.8 Functors in the resource grammar

In Section 5.8 we saw a prime example of the use of functors in GF: application grammars written by using the resource grammar API. But functors are also useful inside the resource grammar itself. There they are used on a lower level of linguistic detail, to implement **families of languages**. Thus in the version of 2010, two families are implemented by means of functors:

- Romance: Catalan, French, Italian, Spanish
- Scandinavian: Danish, Norwegian, Swedish

Both of these functor implementations have the following structure:

- **resource** **CommonFam**, a common resource for the family

- `interface DiffFam`, a minimal interface declaring all constants that can be different
- `instance DiffLang`, the definition of the constants in *Lang*
- `interface ResFam`, a comprehensive resource interface extending `Diff`
- `incomplete concrete CatFam`, functor over `ResFam`
- `incomplete concrete NounFam` and other phrase structure modules are functors over `ResFam`

When a language *Lang* is added to the family, only `DiffLang` needs some real work: all the functor instantiations can be produced mechanically.

What is said above concerns the syntax part of the grammar. No effort has been made to functorize the morphology or the lexicon, and hence `Idiom`, `Structural`, `Lexicon`, `Numeral`, and `Paradigms` are ordinary modules separate for each language.

To get a better idea of what things may distinguish the languages within a family, let us look at `DiffRomance`. It contains some lexical material that is used in the syntax module, for instance, the auxiliary verbs. But for syntax proper, there are just eight constants that in the end make the difference between Catalan, French, Italian, and Spanish:

- Prepositions that fuse with the article (Fre, Spa *de*, *a*; Ita also *con*, *da*, *in*, *su*).

`param Prepos`

- Which types of verbs exist, in terms of auxiliaries. (Fre, Ita *avoir*, *être*, and reflexive; Spa only *haber* and reflexive).

`param VType`

- Derivatively, if/when the participle agrees with the subject. (Fre *elle est partie*, Ita *lei è partita*, Spa never)

`partAgr : VType -> VPAgr`

- Whether participle agrees with foregoing clitic. (Fre yes: *je l'ai vue*, Spa not: *yo la he visto*)

`vpAgrClit : Agr -> VPAgr`

- Whether a preposition is repeated in conjunction (Fre *la somme de 3 et de 4*, Ita *la somma di 3 e 4*).

`conjunctCase : NPForm -> NPForm`

- How infinitives and clitics are placed relative to each other (Fre *la voir*, Ita *vederla*). The `Bool` is used for indicating if there are any clitics.

```
clitInf : Bool -> Str -> Str -> Str
```

- To render pronominal arguments as clitics and/or ordinary complements. Returns `True` if there are any clitics.

```
pronArg :
```

```
Number -> Person -> CAgr -> CAgr -> Str * Str * Bool
```

- To render imperatives (with their clitics etc).

```
mkImper :
```

```
Bool -> Person -> VP -> {s : Polarity => Agr => Str}
```

Now, setting up a `Diff` interface like this for a family of languages is hardly a way to start an implementation with! It is only through experience that one can find out what there should be in the `Diff`, and if `Diff` remains sufficiently small to make the functor worth-while at all.

It is mathematically possible to implement *any* set of languages by means of a functor: just write a `Diff` interface that contains the whole abstract syntax, with instances containing the concrete syntaxes! But this would certainly be over-engineering. Thus the library hasn't, for instance, tried to implement all Germanic languages (Scandinavian, German, Dutch, English) with a functor: the `Diff` would probably be too extensive to be really useful. Also the recently added Romanian grammar was built separately from Romance, by using ideas from it but not by using the functor.

Let us summarize the pros and cons of functors in the resource grammar, and conclude with some practical recommendations:

- + Intellectual satisfaction: linguistic generalizations.
- + Code sharing: of syntax code, 75% in Romance and 85% in Scandinavian is common.
- + Bug fixes and maintenance can usually be shared as well.
- + Adding a new language of the same family can be very easy.
- It is difficult to get started with proper abstractions.
- New languages may require extensions of the `Diff` interface.

The recommended workflow is:

1. Don't start with a functor, but first do one language in the normal way.
2. Refactor the resulting grammar into an interface, functor, and instance; the `Diff` interface is initialized by an empty module.
3. Add the second language by writing a morphology and the test lexicon, still keeping the `Diff` interface empty.

4. Test the resulting grammar, and add constants to the `Diff` interface when needed.

All this said, we see it as a potentially rewarding research programme to increase the use of functors and create hierarchies of functors on different levels. For instance, Germanic could be a functor on a higher level than Scandinavian.

Restricted inheritance (Section 4.8) was not available when the Romance and Scandinavian languages were implemented. It makes the use of functors more flexible, since differences can be treated by means of exceptions rather than interface constants. However, it also blurs the generalizations and theoretical consequences one can draw from a pure functor implementation.

Exercise 10-1.*+ Refactor the Italian grammar in Appendix A into a functor and an interface permitting a smooth implementation of a grammar for French or some other Romance language. Then implement this language.

10.9 Widening the coverage for parsing text

The ultimate wide-coverage grammar is one that can parse general text, not just text on some limited domain. Resource grammars, even though they are large-scale and aim to cover entire languages, are not suitable for parsing general text. There are four main reasons for this: lexicon, syntax coverage, ambiguity, and efficiency. The first two problems could of course be solved by extending the resource grammars themselves. The last two problems are created by the abstractness and deepness of the resource grammar. They are solved by making the grammar more concrete and shallow.

One recommended technique for building an efficient grammar for parsing is the **flattening** of its structures. This involves, in particular, the elimination of verb phrases. The parsing grammar should be implemented as an application grammar, with rules such as

```
fun PredV2 : NP -> V2 -> NP -> C1
lin PredV2 subj verb obj = PredVP subj (Comp1V2 verb obj)
```

Much of this flattening is readily given in the Resource Grammar Library, so that it is enough to write

```
lin PredV2 = mkC1
```

What requires special care is the elimination of ambiguities caused by **PP attachment**. Parsing the sentence *I saw her in the city today* gives seven different trees, corresponding to whether *today* attaches to *the city* or to *I saw her in the city*, and so on. The number of such

groupings grows exponentially with the length of a sentence. One way to eliminate this problem is to leave out the normal adverb combination rules from the parsing grammar and insert adverb lists to predication rules:

```
fun PredV2Adv : NP -> V2 -> NP -> ListAdv -> Cl
```

This is at the same time yet another example of flattening.

Large parts of shallow parsing grammars can be built as functors, independently of target language. But eventually some language-dependent work is needed as well. What it is can only be revealed by experimenting with a corpus. This is an area where GF remains to be properly applied.

Exercise 10-2.⁺ Use a resource grammar to generate random sentences and parse them back to trees. Pick a set of sentences with very many parsing results and analyse what functions produce the ambiguities. Write a wrapper grammar that excludes some of these functions (by restricted inheritance), and try to reach a level on which the same sentences can still be parsed but get only one tree each.

Exercise 10-3.*⁺ Write a grammar that inherits most of the resource grammar for some language, but uses flattened predication instead of verb phrases. Measure the performance differences compared to the original resource grammar.

10.10 Bootstrapping a resource lexicon

The test lexicon of the Resource Grammar Library is a set of words meant to cover some of the most common ones. Thus it includes the **Swadesh list** of 207 primitive words (Swadesh 1955) but also some modern words. It is presented as a multilingual lexicon with a common abstract syntax, and it thereby permits translation. However, even this very small list of originally English words contains many ambiguous ones that have several equivalents in other languages. This shows that there can hardly be such a thing as a universal multilingual lexicon. As a more realistic goal, the GF resource grammar project is building monolingual lexica, which give the inflection and other grammatical information of words for each language separately. The words from these lexica can then be used as library functions that help in defining domain lexica. They can also be used in large-scale monolingual parsers.

The easiest way to build a large-scale lexicon is to use an open-source morphological word list. Such word lists can sometimes be found, for instance, in the OpenOffice project where they are used for spell checking (<http://www.openoffice.org/>). These lists may give the full sets

of inflection forms of words. Then they can provide the full lists of arguments for worst-case paradigms (see Section 4.2) and can thereby be used for constructing complete lexical entries. For instance, a German word list might give six forms of verbs,

```
folgen folgt folgte gefolgt folgte folge
```

(“follow”). These forms can be ordered to match the arguments requested by the worst-case mkV in German and thereby to generate a pair of GF rules:

```
fun w02948_folgen_V : V ;
lin w02948_folgen_V =
  "folgen" "folgt" "folgte" "gefolgt" "folgte" "folge" ;
```

It is a good practice to generate a unique identifier (like the number 02948 here) for each word, since the word list can contain homonyms.

A variant of explicit full-form word lists are lists that just give the base form but annotate it with a paradigm identifier. The Finnish KOTUS list (<http://kaino.kotus.fi/>) is an example of this. Two approaches are available then. One is to expand the word list to a full-form lexicon, if the list is distributed with an inflection engine of some kind. Another one is to implement the paradigms by using the resource grammar paradigms. This is what has been done in the case of KOTUS. It is an XML file with entries such as

```
<st><s>aines</s><t><tn>39</tn></t></st>
```

for the word *aines* (“ingredient”). We can define the KOTUS paradigm 39 as

```
kotus39_N : Str -> N = \jalas ->
  mkN jalas (init jalas + "ksia") ;
```

Then we can derive the entry

```
lin w00651_aines_N = kotus39_N "aines" ;
```

with the corresponding abstract syntax entry.

For many languages, no morphological word lists are available. There may still be plain word lists with just the basic forms. If such a list has part-of-speech tags, it can be bootstrapped by using the regular paradigms. For instance, if we find a German word marked

```
Reich, noun
```

(“kingdom”) we can generate the entry

```
w00702_reich_N = mkN "Reich" ;
```

We can then use the smart paradigm mkN with one argument to generate

the arguments for the next `mkN`, and get

```
w00702_reich_N = mkN "Reich" "Reiche" masculine ;
```

Now, this result is not correct, but we have to change it to

```
w00702_reich_N = mkN "Reich" "Reiche" neuter ;
```

The next iteration, with the 7-argument `mkN`, gives

```
w00702_reich_N = mkN "Reich" "Reich" "Reich"  
"Reiches" "Reiche" "Reichen" neuter ;
```

which is correct, so we don't need any new forms.

The number of corrections needed in the bootstrapping process gives a measure of the predictability of the morphology in the language, and certainly a measure of the smartness of the paradigms. An experiment using the Finnish KOTUS list as gold standard (Ranta 2009) showed that for Finnish nouns, 82% needed only one form, and 96% needed at most two forms. 1.42 forms per noun were needed in average. Since every Finnish noun has 30 forms, this proved to be a very productive way to bootstrap a lexicon.

If a word list has no part-of-speech tags, the paradigms of the resource grammar cannot be used directly. But even then, smart paradigms may be useful for extracting the parts of speech. This is particularly easy for a language like German, where nouns have capital initials and verbs have the infinitive ending *en*. A smart paradigm that just classifies words into nouns, verbs, and the rest can then provide a long step towards a morphological lexicon.

The most difficult case for bootstrapping is the usage of plain text as the only available material. This is worse than a plain word list, because it doesn't tell what words in the text are base forms. But even then, a technique related to smart paradigms can be used. The **lexicon extraction tool** (Forsberg & al. 2006) supports rules such as

```
rule mkN x (x + "ar") =  
  {( x      | x+"en"   | x+"s"    | x+"ens") &  
    ( x+"ar" | x+"arna" | x+"ars"  | x+"arnas") } ;
```

This rule applies to Swedish, targeting the smart paradigm `mkN` that produces all the eight noun forms from two arguments. These two arguments are the singular and plural nominative indefinite forms, for instance, *bil* and *bilar* ("car"). However, to establish that something is a noun inflected in this way, it is sufficient to find any pair of a singular and a plural form, whatever their case and definiteness. The rule above expresses this by using a conjunction (&) of two disjunctions (|). Thus the two forms

`kompisens kompisarna`

(“mate”) produce the lexical entry

`w12345_kompis_N = mkN "kompis" "kompisar"`

Humayoun & al. (2007) used the same technique to extract a lexicon of 4,816 entries in Urdu from texts on the Internet.

Exercise 10-4.⁺ Make a rough estimate of the predictability of inflection in some languages in the Resource Grammar Library by studying the number of forms needed to implement the resource **Lexicon**. This number may be pessimistic, because the **Lexicon** has a high percentage of irregular words.

Exercise 10-5.^{*+} Write extraction rules for some language of your choice. The extraction tool can be downloaded via the book web page.

Appendix A

A miniature resource grammar

A.1 Abstract syntax

```
abstract Grammar = {

  flags startcat = S ;

  cat
    S ; C1 ; NP ; VP ; AP ; CN ;
    Det ; N ; A ; V ; V2 ; AdA ; Tense ;
  fun
    UseC1   : Tense -> C1 -> S ;
    PredVP  : NP -> VP -> C1 ;
    ComplV2 : V2 -> NP -> VP ;
    DetCN   : Det -> CN -> NP ;
    AdjCN   : AP -> CN -> CN ;
    CompAP  : AP -> VP ;
    AdAP    : AdA -> AP -> AP ;

    UseV    : V -> VP ;
    UseN    : N -> CN ;
    UseA    : A -> AP ;

    Pres, Perf : Tense ;

    a_Det, every_Det, the_Det : Det ;
    this_Det, these_Det : Det ;
    that_Det, those_Det : Det ;
    i_NP, she_NP, we_NP : NP ;
    very_AdA : AdA ;

-- coordination
  cat Conj ;
  fun
    ConjS : Conj -> S -> S -> S ;
    ConjNP : Conj -> NP -> NP -> NP ;
    and_Conj, or_Conj : Conj ;
}
```

A.2 Auxiliary resource module for Italian

```

resource ResIta = open Prelude in {

param
  Number = Sg | Pl ;
  Gender = Masc | Fem ;
  Case   = Nom | Acc | Dat ;
  Agr     = Ag Gender Number Person ;
  Aux     = Avere | Essere ;
  Tense   = Pres | Perf ;
  Person  = Per1 | Per2 | Per3 ;

  VForm   = VInf | VPres Number Person | VPart Gender Number ;

  ClitAgr = CAgrNo | CAgr Agr ;

oper
  VP = {
    v : Verb ;
    clit : Str ;
    clitAgr : ClitAgr ;
    obj : Agr => Str
  } ;
  NP = {
    s : Case => {clit,obj : Str ; isClit : Bool} ;
    a : Agr
  } ;

-- the preposition word of an abstract case

prepCase : Case -> Str = \c -> case c of {
  Dat => "a" ;
  _ => []
} ;

-- for predication

auxVerb : Aux -> Verb = \a -> case a of {
  Avere  => mkVerb "avere" "ho" "hai" "ha"
          "abbiamo" "avete" "hanno" "avuto" Avere ;
  Essere => mkVerb "essere" "sono" "sei" "è"
          "siamo" "siete" "sono" "stato" Essere
} ;
essere_V = auxVerb Essere ;

agrPart : Verb -> Agr -> ClitAgr -> Str = \v,a,c -> case v.aux of {
  Avere  => case c of {
    CAgr (Ag g n _) => v.s ! VPart g n ;
    _ => v.s ! VPart Masc Sg
  } ;
  Essere => case a of {
    Ag g n _ => v.s ! VPart g n
  }
}

```

```

    }
  } ;
agrV : Verb -> Agr -> Str = \v,a -> case a of {
  Ag _ n p => v.s ! VPres n p
} ;

-- morphology

Noun : Type = {s : Number => Str ; g : Gender} ;
Adj  : Type = {s : Gender => Number => Str ; isPre : Bool} ;
Verb : Type = {s : VForm => Str ; aux : Aux} ;

mkNoun : Str -> Str -> Gender -> Noun = \vino,vini,g -> {
  s = table {Sg => vino ; Pl => vini} ;
  g = g
} ;

regNoun : Str -> Noun = \vino -> case vino of {
  fuo + c@("c"|"g") + "o" => mkNoun vino (fuo + c + "hi") Masc ;
  ol + "io" => mkNoun vino (ol + "i") Masc ;
  vin + "o" => mkNoun vino (vin + "i") Masc ;
  cas + "a" => mkNoun vino (cas + "e") Fem ;
  pan + "e" => mkNoun vino (pan + "i") Masc ;
  _ => mkNoun vino vino Masc
} ;

mkAdj : (_,_,_,_ : Str) -> Bool -> Adj =
\buono,buona,buoni,buone,p -> {
  s = table {
    Masc => table {Sg => buono ; Pl => buoni} ;
    Fem  => table {Sg => buona ; Pl => buone}
  } ;
  isPre = p
} ;

regAdj : Str -> Adj = \nero -> case nero of {
  ner + "o"  => mkAdj nero (ner + "a") (ner + "i") (ner + "e") False ;
  verd + "e" => mkAdj nero nero (verd + "i") (verd + "i") False ;
  _ => mkAdj nero nero nero nero False
} ;

mkVerb : (_,_,_,_,_,_,_ : Str) -> Aux -> Verb =
\amare,amo,ami,ama,amiamo,amate,amano,amato,aux -> {
  s = table {
    VInf          => amare ;
    VPres Sg Per1 => amo ;
    VPres Sg Per2 => ami ;
    VPres Sg Per3 => ama ;
    VPres Pl Per1 => amiamo ;
    VPres Pl Per2 => amate ;
    VPres Pl Per3 => amano ;
    VPart g n     => (regAdj amato).s ! g ! n
  }
}

```

```

    } ;
    aux = aux
  } ;

regVerb : Str -> Verb = \amare -> case amare of {
  am + "are" => mkVerb amare (am+"o") (am+"i") (am+"a")
    (am+"iamo") (am+"ate") (am+"ano") (am+"ato") Avere ;
  tem + "ere" => mkVerb amare (tem+"o") (tem+"i") (tem+"e")
    (tem+"iamo") (tem+"ete") (tem+"ono") (tem+"uto") Avere ;
  fin + "ire" => mkVerb amare (fin+"isco") (fin+"isci") (fin+"isce")
    (fin+"iamo") (fin+"ite") (fin+"iscono") (fin+"ito") Avere
  } ;

-- for structural words

adjDet : Adj -> Number -> {s : Gender => Case => Str ; n : Number} =
\adj,n -> {
  s = \g,c => prepCase c ++ adj.s ! g ! n ;
  n = n
} ;

pronNP : (s,a,d : Str) -> Gender -> Number -> Person -> NP =
\s,a,d,g,n,p -> {
  s = table {
    Nom => {clit = [] ; obj = s ; isClit = False} ;
    Acc => {clit = a ; obj = [] ; isClit = True} ;
    Dat => {clit = d ; obj = [] ; isClit = False}
  } ;
  a = Ag g n p
} ;

-- phonological auxiliaries

vowel : pattern Str = #("a" | "e" | "i" | "o" | "u" | "h") ;
s_impuro : pattern Str =
  #("z" | "s" + ("b"|"c"|"d"|"f"|"m"|"p"|"q"|"t")) ;

elisForms : (_,_,_ : Str) -> Str = \lo,l',il ->
  pre {#s_impuro => lo ; #vowel => l' ; _ => il} ;

-- for coordination

oper
conjAgr : Number -> Agr -> Agr -> Agr = \n,xa,ya ->
  let
    x = agrFeatures xa ; y = agrFeatures ya
  in Ag
    (conjGender x.g y.g)
    (conjNumber (conjNumber x.n y.n) n)
    (conjPerson x.p y.p) ;

agrFeatures : Agr -> {g : Gender ; n : Number ; p : Person} = \a ->

```

```

case a of {Ag g n p => {g = g ; n = n ; p = p}} ;

conjGender : Gender -> Gender -> Gender = \g,h ->
  case g of {Masc => Masc ; _ => h} ;

conjNumber : Number -> Number -> Number = \m,n ->
  case m of {Pl => Pl ; _ => n} ;

conjPerson : Person -> Person -> Person = \p,q ->
  case <p,q> of {
    <Per1,_> | <_,Per1> => Per1 ;
    <Per2,_> | <_,Per2> => Per2 ;
    _ => Per3
  } ;
}

```

A.3 Italian concrete syntax

```

concrete GrammarIta of Grammar = open ResIta, Prelude in {
  lincat
    S = {s : Str} ;
    Cl = {s : ResIta.Tense => Str} ;
    NP = ResIta.NP ;
    VP = ResIta.VP ;
    AP = {s : Gender => Number => Str ; isPre : Bool} ;
    CN = ResIta.Noun ;
    Det = {s : Gender => Case => Str ; n : Number} ;
    N = ResIta.Noun ;
    A = ResIta.Adj ;
    V = ResIta.Verb ;
    V2 = ResIta.Verb ** {c : Case} ;
    AdA = {s : Str} ;
    Tense = {s : Str ; t : ResIta.Tense} ;
    Conj = {s : Str ; n : Number} ;
  lin
    UseCl t cl = {s = t.s ++ cl.s ! t.t} ;
    PredVP np vp =
      let
        subj = (np.s ! Nom).obj ;
        obj = vp.obj ! np.a ;
        clit = vp.clit ;
        verb = table {
          Pres => agrV vp.v np.a ;
          Perf => agrV (auxVerb vp.v.aux) np.a ++
            agrPart vp.v np.a vp.clitAgr
        }
      in {
        s = \t => subj ++ clit ++ verb ! t ++ obj
      } ;
    ComplV2 v2 np =
      let
        nps = np.s ! v2.c
      in {

```

```

    v = v2 ;
    clit = nps.clit ;
    clitAgr = case <nps.isClit,v2.c> of {
        <True,Acc> => CAgr np.a ;
        _ => CAgrNo
    } ;
    obj = \_ => nps.obj
  } ;
UseV v = {
  v = v ;
  clit = [] ;
  clitAgr = CAgrNo ;
  obj = \_ => []
} ;

DetCN det cn = {
  s = \c => {
    obj = det.s ! cn.g ! c ++ cn.s ! det.n ;
    clit = [] ;
    isClit = False
  } ;
  a = Ag cn.g det.n Per3
} ;
AdjCN ap cn = {
  s = \n => preOrPost ap.isPre (ap.s ! cn.g ! n) (cn.s ! n) ;
  g = cn.g
} ;
CompAP ap = {
  v = essere_V ;
  clit = [] ;
  clitAgr = CAgrNo ;
  obj = \ag => case ag of {
    Ag g n _ => ap.s ! g ! n
  }
} ;
AdAP ada ap = {
  s = \g,n => ada.s ++ ap.s ! g ! n ;
  isPre = ap.isPre ;
} ;

UseN n = n ;
UseA a = a ;

a_Det = adjDet (mkAdj "un" "una" [] [] True) Sg ;
every_Det = adjDet (regAdj "ogni") Sg ;
the_Det = {
  s = table {
    Masc => table {
      Nom | Acc => elisForms "lo" "l'" "il" ;
      Dat => elisForms "allo" "all'" "al"
    } ;
    Fem => table {

```

```

    Nom | Acc => elisForms "la" "'l" "la" ;
    Dat => elisForms "alla" "all'" "alla"
  }
} ;
n = Sg
} ;
this_Det = adjDet (regAdj "questo") Sg ;
these_Det = adjDet (regAdj "questo") Pl ;
that_Det = adjDet quello_A Sg ;
those_Det = adjDet quello_A Pl ;

Pres = {s = [] ; t = ResIta.Pres} ;
Perf = {s = [] ; t = ResIta.Perf} ;

i_NP = pronNP "io" "mi" "mi" Masc Sg Per1 ;
she_NP = pronNP "lei" "la" "le" Fem Sg Per3 ;
we_NP = pronNP "noi" "ci" "ci" Masc Pl Per1 ;
very_AdA = ss "molto" ;
and_Conj = {s = "e" ; n = Pl} ;
or_Conj = {s = "o" ; n = Sg} ;

ConjNP co nx ny = {
  s = \\c => {
    obj = (nx.s ! c).obj ++ co.s ++ (ny.s ! c).obj ;
    clit = [] ;
    isClit = False
  } ;
  a = conjAgr co.n nx.a ny.a
} ;
ConjS co x y = {s = x.s ++ co.s ++ y.s} ;

oper
quello_A : Adj = mkAdj
  (elisForms "quello" "quell'" "quel") "quella"
  (elisForms "quegli" "quegli" "quei") "quelle"
  True ;
}

```

A.4 Morphological paradigms API for Italian

```

resource ParadigmsIta = GrammarIta [N,A,V] **
  open ResIta, GrammarIta, Prelude in {
oper
  masculine : Gender = Masc ;
  feminine : Gender = Fem ;
  accusative : Case = Acc ;
  dative : Case = Dat ;

  mkN = overload {
    mkN : (vino : Str) -> N
      = \n -> lin N (regNoun n) ;
    mkN : (uomo, uomini : Str) -> Gender -> N
      = \s,p,g -> lin N (mkNoun s p g) ;
  }

```



```

    } ;
mkA = overload {
  mkA : (nero : Str) -> A
    = \a -> lin A (regAdj a) ;
  mkA : (buono,buona,buoni,buone : Str) -> Bool -> A
    = \sm,sf,pm,pf,p -> lin A (mkAdj sm sf pm pf False) ;
} ;
preA : A -> A
  = \a -> lin A {s = a.s ; isPre = True} ;
mkV = overload {
  mkV : (finire : Str) -> V
    = \v -> lin V (regVerb v) ;
  mkV : (andare,vado,vadi,va,andiamo,andate,vanno,andato : Str) -> V
    = \i,p1,p2,p3,p4,p5,p6,p ->
      lin V (mkVerb i p1 p2 p3 p4 p5 p6 p Avere) ;
} ;
essereV : V -> V
  = \v -> lin V {s = v.s ; aux = Essere} ;
mkV2 = overload {
  mkV2 : Str -> V2
    = \s -> lin V2 (regVerb s ** {c = accusative}) ;
  mkV2 : V -> V2
    = \v -> lin V2 (v ** {c = accusative}) ;
  mkV2 : V -> Case -> V2
    = \v,c -> lin V2 (v ** {c = c}) ;
} ;
}

```

A.5 Test lexicon

```

abstract Test = Grammar ** {
fun
  man_N, woman_N, house_N, tree_N : N ;
  big_A, small_A, green_A : A ;
  walk_V, arrive_V : V ;
  love_V2, please_V2 : V2 ;
}

concrete TestIta of Test = GrammarIta ** open ParadigmsIta in {
lin
  man_N = mkN "uomo" "uomini" masculine ;
  woman_N = mkN "donna" ;
  house_N = mkN "casa" ;
  tree_N = mkN "albero" ;
  big_A = preA (mkA "grande") ;
  small_A = preA (mkA "piccolo") ;
  green_A = mkA "verde" ;
  walk_V = mkV "camminare" ;
  arrive_V = essereV (mkV "arrivare") ;
  love_V2 = mkV2 "amare" ;
  please_V2 = mkV2 (essereV (mkV "piacere" "piaccio" "piaci" "piace"
    "piacciamo" "piacete" "piacciono" "piaciuto"))) dative ;
}

```

A.6 Syntax API

```

interface Syntax = Grammar -
  [UseCl,PredVP,ComplV2,UseV,DetCN,AdjCN,CompAP,AdAP,
   UseN,UseA,Pres,Perf] **
  open Grammar in {
oper
  mkS = overload {
    mkS : Cl -> S = UseCl Pres ;
    mkS : Tense -> Cl -> S = UseCl ;
    mkS : Conj -> S -> S -> S = ConjS ;
  } ;
  mkCl = overload {
    mkCl : NP -> V -> Cl = \np,v -> PredVP np (UseV v) ;
    mkCl : NP -> V2 -> NP -> Cl = \np,v,o -> PredVP np (ComplV2 v o) ;
    mkCl : NP -> A -> Cl = \np,a -> PredVP np (CompAP (UseA a)) ;
    mkCl : NP -> AP -> Cl = \np,ap -> PredVP np (CompAP ap) ;
    mkCl : NP -> VP -> Cl = PredVP ;
  } ;
  mkAP = overload {
    mkAP : A -> AP = UseA ;
    mkAP : AdA -> AP -> AP = AdAP ;
  } ;
  mkNP = overload {
    mkNP : Det -> N -> NP = \d,n -> DetCN d (UseN n) ;
    mkNP : Det -> CN -> NP = \d,n -> DetCN d n ;
    mkNP : Conj -> NP -> NP -> NP = ConjNP ;
  } ;
  mkCN = overload {
    mkCN : N -> CN = UseN ;
    mkCN : A -> N -> CN = \a,n -> AdjCN (UseA a) (UseN n) ;
    mkCN : A -> CN -> CN = \a,n -> AdjCN (UseA a) n ;
    mkCN : AP -> N -> CN = \a,n -> AdjCN a (UseN n) ;
    mkCN : AP -> CN -> CN = \a,n -> AdjCN a n ;
  } ;
  presTense : Tense = Pres ;
  perfTense : Tense = Perf ;
}

instance SyntaxIta of Syntax = GrammarIta -
  [UseCl,PredVP,ComplV2,UseV,DetCN,AdjCN,CompAP,AdAP,
   ConjS,ConjNP,UseN,UseA,Pres,Perf] **
  open GrammarIta in {} ;

```


Appendix B

A glossary of linguistic terms

Agreement. The dependence of the form of a **constituent** on other **constituents**. Example: in English, the form of a verb depends on the number and person of the **subject**. In GF: typically, an inherent feature of one constituent determines a variable feature of another one, e.g. $\text{lin Pred np vp} = \text{np.s ++ vp.s ! np.n}$.

Aspect. A feature of verb phrases expressing the progression or completion of action. For instance, *she is running* has a progressive aspect, as opposed to *she runs*. Aspect is realized in different ways in different languages, and often intertwined with **tense** in the morphological and syntactic variation of verbs, verb phrases, and sentences.

Case. A feature expressing the role of a noun phrase in a sentence; typically attached to nouns, adjectives, and pronouns; either morphological or expressed syntactically by prepositions. Example: German has the cases nominative, accusative, dative, and genitive. In GF: definable as a parameter type $\text{Case} = \text{Nom} \mid \text{Acc} \mid \text{Dat} \mid \text{Gen}$.

Clause. A unit of language consisting of a **subject** and a **predicate**. In GF Resource Grammar Library, the category C1, which has undetermined **tense** and **polarity**.

Complement. Argument of a verb, an adjective, or a noun, other than subject.

Complementation. The process of providing the **complement** to form a phrase.

Conjugation. An inflection paradigm for verbs.

Constituent. A subtree of a syntax tree, or a string corresponding to a subtree. Example: if in the sentence *John is old*, the substring *is old* is a constituent, then *John is* isn't. In GF: a subtree of an abstract syntax tree, or the corresponding set of substrings; if this is not a singleton, then it is a **discontinuous constituent**.

Coordination. A list of constituents of the same category grouped together by the use of a conjunction. Example: *John, Mary, and me* with the conjunction *and*.

Copula. A verb, such as English *be*, used for building verb phrases from non-verb phrases such as adjectives.

Corpus. A collection of language samples, usually text but also speech recordings.

Declension. An inflection paradigm for nouns or adjectives.

Discontinuous constituent. A set of strings corresponding to a subtree, but separated by other strings. Example: in *switch it off*, *switch - off* is a discontinuous constituent. In GF: a linearization that is a record with more strings than one used simultaneously.

Endocentric. A syntactic construction with a **head** that has the same type

as the construction itself. Example: modification of a noun with an adjective, to construct a noun. In GF: an abstract syntax function whose type has the form $\dots \rightarrow C \rightarrow \dots \rightarrow C$.

Ergative. A construction in which the **subject** of a transitive verb has a special case, and the **object** has the same case as the subject of an intransitive verb. It may also happen that the verb then agrees with the object. Examples of a language that have ergative constructions are Basque and Hindi.

Exocentric. A syntactic construction with no argument having the same type as the construction itself, and hence without a **head**. Example: formation of a sentence from a noun phrase and a verb phrase. In GF: an abstract syntax function whose type has the form $\dots \rightarrow C$ where no argument type is C .

Gender. A morphological feature expressing classification based on properties such as sex; typically attached to nouns and adjectives. Example: German and Latin have the features masculine, feminine, and neuter. In GF: definable as a parameter type **Gender** = Masc | Fem | Neut.

Head. In an **endocentric** construction, the **constituent** that has the same type as the construction itself. Example: *man* is the head of *old man*. In GF: the argument of type C of a function whose value type is C and which has exactly one argument of type C .

Inflection. The production of different forms of a word. Example: the English verb *die* is inflected *dies*, *died*, *dying*.

Intransitive. A verb that doesn't take an **object**. In GF Resource Grammar Library, **V** is the category of intransitive verbs. But also a **V2** verb may be called intransitive, if its complement is not in the accusative case.

Lexeme. A word in an abstract sense, independently of its inflectional forms.

Lexicon. A list of words, either forms or lexemes.

Morphology. The study of inflection and word formation.

Number. A morphological feature expressing the difference between one and many, sometimes also other numbers; typically attached to nouns and often also to adjectives and verbs. Example: most languages have the features singular and plural, many even dual. In GF: definable as a parameter type **Number** = Sg | Pl | D1.

Object. The second argument (after the **subject**) of a **transitive verb**. Example: in *John loves Mary*, *Mary* is the object.

Paradigm. A function for producing the **inflection** forms of a word. Example: English regular verb paradigm.

Part of speech. Class of words or, by generalization, phrases. Example: noun, adjective, determiner.

Polarity. The property of a sentence of being positive or negative.

Predicate. In traditional grammar, the part of a sentence saying something about the **subject**; in modern logic, a function forming propositions from one or more arguments. Example: traditionally, *John loves Mary* has the predicate *loves Mary*; in modern logic, *love* could be seen as a predicate that takes two arguments.

Predication. The syntactic operation of putting together a **subject** and a **predicate**, or a **predicate** with its arguments. In GF, the resource grammar function **PredVP** : NP -> VP -> C1 expresses predication.

Subject. Traditionally, the part of a sentence about which something is said by a **predicate**; in linguistic syntax, the constituent (normally noun phrase) that gives the agreement features to the verb (except in some **ergative** languages).

Syncretism. In morphology, different form descriptions producing the same form. Example: for regular English verbs, past indicative and participle have the same form, as in *kill-killed-killed*.

Syntax. The part of grammar telling how words and phrases are combined to larger phrases. Example: the rule for combining a **subject** and a **predicate** into a clause belongs to syntax. In GF, inspired by the theory of programming languages, a distinction is made between **abstract syntax** and **concrete syntax**.

Tense. A feature of verb phrases expressing temporal order (past, present, future). Often intertwined with **aspect** in the verb form systems of languages.

Transitive. A verb that takes an **object**. In GF Resource Grammar Library, V2 is the category of transitive verbs, in particular if the complement is in the accusative case.

Utterance. Unit of spoken language, or its representation in written language. In GF Resource Grammar Library, the category **Utt**.

Vocative. An expression used for addressing a hearer, for instance, *John* in *Come here, John*. Expressed by a special **case** in some languages, e.g. Latin. In GF Resource Grammar Library, the category **Voc**.

Appendix Part III

GF Reference Manual

Appendix C

The GF Programming Language

GF, Grammatical Framework, is a special-purpose programming language designed to support the implementation of grammars. This Appendix is a reference manual of the GF language.

The reference manual is not an introduction to GF; such an introduction can be found in the first part of this book. This manual covers only the language. The libraries will be covered in Appendix D and the GF system in Appendix E.

This manual is meant to be fully compatible with and complete for GF version 3.2 released in 2010. A more theoretical and concise description is found in the reference article on GF,

A. Ranta, “Grammatical Framework. A Type Theoretical Grammar Formalism”, *The Journal of Functional Programming* 14(2), 2004, pp. 145-189.

This article will be referred to as “the JFP article”. It covers only the expressions, types, and judgements, not the module level.

As metalinguistic notation, we will use the symbols

- $a \equiv b$ to say that a is syntactic sugar for b
- $a \Downarrow b$ to say that a is computed (or compiled) to b

C.1 Overview of GF

GF is a typed functional language, borrowing many of its constructs from ML and Haskell: algebraic datatypes, higher-order functions, pattern matching. The module system bears resemblance to ML (functors) but also to object-oriented languages (inheritance). The type theory used in the abstract syntax part of GF is inherited from logical frameworks, in particular ALF (“Another Logical Framework”; in a sense, GF is Yet Another ALF). From ALF comes also the use of dependent types, including the use of explicit type variables instead of Hindley-Milner polymorphism.

The look and feel of GF is close to Java and C, due to the use of curly brackets and semicolons in structuring the code; the expression syntax, however, follows Haskell in using juxtaposition for function application and parentheses only for grouping.

To understand the constructs of GF, and especially their limitations in comparison to general-purpose programming languages, it is essential to keep in mind that GF is a special-purpose and non-Turing-complete language. Every GF program is ultimately compiled to a **multilingual grammar**, which consists of an **abstract syntax** and a set of **concrete syntaxes**. The abstract syntax defines a system of **syntax trees**, and each concrete syntax defines a mapping from those syntax trees to tuples of strings. This mapping is **compositional**, i.e. **homomorphic**, and moreover **reversible**: given a string, there exists an effective way of finding the set of syntax trees that map to this string (as a component of a tuple).

The procedure of applying the mapping to a tree to produce a tuple is called **linearization**, and the reverse search procedure is called **parsing**. In practice, parsing is applied to strings extracted from tuples, not to complete tuples themselves. It is ultimately the requirement of reversibility that restricts GF to be less than Turing-complete. This is reflected in restrictions to recursion in concrete syntax. Tree formation in abstract syntax, however, is fully recursive.

Even though run-time GF grammars manipulate just tuples of strings, at compile time these are represented by the more fine-grained labelled records and finite functions over algebraic datatypes. This enables the programmer to write code on a higher abstraction level, and also adds type distinctions and hence raises the accuracy of static type checking.

C.2 The module system

C.2.1 Top-level and supplementary module structure

The big picture of GF as a programming language for multilingual grammars explains its principal module structure. Any GF grammar (i.e. what will be called a top-level GF grammar) must have an **abstract syntax** module; it can in addition have any number of **concrete syntax** modules matching that abstract syntax. Before going to details, we give a simple example: a module with two **judgements**, which define one **category**, **A** (intuitively, adjectives) and one **function**, the zero-place function **Even** that forms an adjective. We give the module the name **Adj**. The GF code for the module looks as follows:

```
abstract Adj = {
  cat A ;
  fun Even : A ;
}
```

Here are two concrete syntax modules, one intended for mapping the trees to English, the other to Swedish. The mapping is defined by **lincat** definitions assigning a **linearization type** to each category, and **lin** definitions assigning a **linearization** to each function.

```
concrete AdjEng of Adj = {
  lincat A = {s : Str} ;
```

```

  lin Even = {s = "even"} ;
}

concrete AdjSwe of Adj = {
  lincat A = {s : AForm => Str} ;
  lin Even = {s = table {
    ASg Utr => "jänn" ;
    ASg Neutr => "jämnt" ;
    AP1 => "jämna"
  }}
} ;
param AForm = ASg Gender | AP1 ;
param Gender = Utr | Neutr ;
}

```

These examples illustrate the main ideas of multilingual grammars:

- the concrete syntax must match the abstract syntax:
 - every **cat** is given a **lincat**
 - every **fun** is given a **lin**
- the concrete syntax is internally coherent:
 - the **lin** rules respect the types defined by **lincat** rules
- concrete syntaxes are independent of each other
 - they can use different **lincat** and **lin** definitions
 - they can define their own **parameter types** (**param**)

The first two ideas form the core of the **static checking** of GF grammars, eliminating the possibility of run-time failures in linearization and parsing. The third idea gives GF the expressive power needed to map abstract syntax to vastly different languages.

Abstract and concrete modules are called **top-level grammar modules**, since they are the ones that are preserved in grammar systems at run time. However, in order to support **modular grammar engineering**, GF provides much more module structure than strictly required by top-level grammars.

Inheritance, also known as **extension**, means that a module can inherit the contents of one or more other modules to which new judgements are added, e.g.

```

abstract MoreAdj = Adj ** {
  fun Odd : A ;
}

```

Resource modules define parameter types and **operations** usable in several concrete syntaxes,

```

resource MorphoFre = {
  param Number = Sg | Pl ;
  param Gender = Masc | Fem ;
}

```

```

oper regA : Str -> {s : Gender => Number => Str} =
  \fin -> {s = table {
    Masc => table {Sg => fin ; Pl => fin + "s"} ;
    Fem => table {Sg => fin + "e" ; Pl => fin + "es"}
  }
  } ;
}

```

By **opening**, a module can use the contents of a resource module without inheriting them, e.g.

```

concrete AdjFre of Adj = open MorphoFre in {
  lincat A = {s : Gender => Number => Str} ;
  lin Even = regA "pair" ;
}

```

Interfaces and **instances** separate the contents of a resource module to type signatures and definitions, in a way analogous to abstract vs. concrete modules, e.g.

```

interface Lexicon = {
  oper Adjective : Type ;
  oper even_A : Adjective ;
}

instance LexiconEng of Lexicon = {
  oper Adjective = {s : Str} ;
  oper even_A = {s = "even"} ;
}

```

Functors i.e. **parametrized modules** i.e. **incomplete modules**, defining a concrete syntax in terms of an interface.

```

incomplete concrete AdjI of Adj = open Lexicon in {
  lincat A = Adjective ;
  lin Even = even_A ;
}

```

A functor can be **instantiated** by providing instances of its open interfaces.

```

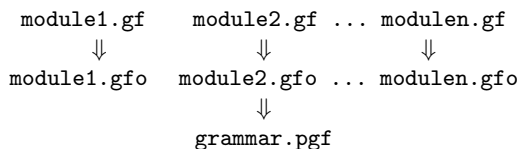
concrete AdjEng of Adj = AdjI with
  (Lexicon = LexiconEng) ;

```

C.2.2 Compilation units

The compilation unit of GF source code is a file that contains a module. Every source file, suffixed **.gf**, is compiled to a **GF object file**, suffixed **.gfo**. For runtime grammar objects used for parsing and linearization, a set of **.gfo** files is linked to a single file suffixed **.pgf**, the **Portable Grammar Format**. While **.gf** and **.gfo** files may contain modules of any kinds, a **.pgf** file always contains a multilingual grammar with one abstract and a set of concrete syntaxes.

The following diagram summarizes the files involved in the compilation process.



Files with the suffix `.gf` are written in the GF source language, while `.gfo` and `.pgf` files use low-level binary formats. The process of translating `.gf` to `.gfo` involves **name resolution**, **type annotation**, **partial evaluation**, and **optimization**.

There is a great advantage in performing separate compilation for each GF module and saving the results in `.gfo` files. The partial evaluation phase, in particular, is time and memory consuming, and GF libraries should therefore be distributed in `.gfo` to make their use less arduous.

The standard compiler has a built-in **make facility**, which finds out exactly what modules are needed when compiling any given module that may depend on other modules. This facility builds a dependency graph and decides which of the involved modules need recompilation (from `.gf` to `.gfo`), and for which an already existing GF object can be used directly.

C.2.3 Names

Each module M defines a set of **names**, which are visible in M itself, in all modules extending M (unless excluded, as explained in Section C.2.7) and in all modules opening M . These names can stand for abstract syntax categories and functions, parameter types and parameter constructors, and operations. All these names live in the same **name space**, which means that a name entering a module more than once due to inheritance or opening can lead to a **conflict**. It is specified in Section C.2.9 how these conflicts are resolved.

The names of modules live in a name space separate from the other names. Even here, all names must be distinct in a set of files compiled to a multilingual grammar. In particular, even files residing in different directories must have different names, since GF has no notion of hierarchic module names.

Lexically, names belong to the class of **identifiers**. An identifier is a letter followed by any number of letters, digits, underscores (`_`) and apostrophes (`'`). Upper- and lower-case letters are treated as distinct. Nothing dictates the choice of upper or lower-case initials, but the standard libraries follow conventions similar to Haskell:

- upper case is used for modules, abstract syntax categories and functions, parameter types and constructors, and type synonyms
- lower case is used for non-type-valued operations and for variables

“Letters” as mentioned in the identifier syntax include all 7-bit ASCII letters (i.e. `[A..Z]` `[a..z]`). Iso-latin-1 letters with diacritics are supported in varying degrees by different tools and platforms, and are hence not recommended

in identifiers.

C.2.4 The structure of a module

Modules of all types have the following structure:

moduletype name = extends opens body

The part of the module preceding the body is its **header**. The header defines the type of the module and tells what other modules it inherits and opens. The body consists of the judgements that introduce all the new names defined by the module.

Any of the parts *extends*, *opens*, and *body* may be empty. If they are all filled, delimiters and keywords separate the parts in the following way:

*moduletype name = extends ** open opens in { body }*

The part *moduletype name* looks slightly different if the type is **concrete** or **instance**: the *name* intrudes between the type keyword and the name of the module being implemented:

concrete name of abstractname

Logically, the type of this module is **concrete of abstractname**.

The only exception to the schema of header syntax is functor instantiations: the instantiation list is given in a special way between *extends* and *opens*:

**incomplete concrete name of abstractname = extends ** functormame
with instantiations ** open opens in { body }**

Logically, the part “*functormame with instantiations*” is really a part of the *extends*. This is also shown by the fact that it can have restricted inheritance (concept defined in Section C.2.7).

C.2.5 Module types, headers, and bodies

The *extends* and *opens* parts of a module header are lists of module names (with possible qualifications, as defined below in Section C.2.8). The first step of type checking a module consists of verifying that these names stand for modules of appropriate module types. As a rule of thumb,

- the *extends* of a module must have the same *moduletype*
- the *opens* of a module must be of type **resource**

However, the precise rules are a little more fine-grained, because of the presence of interfaces and their instances, and the possibility to reuse abstract and concrete modules as resources. The following table gives, for all module types, the possible module types of their *extends* and *opens*, as well as the forms of judgement legal in that module type.

module type	extends	opens	body
abstract	abstract	-	cat,fun,def,data
concrete of <i>abstr</i>	concrete	resource*	lin*,oper,param
resource	resource*	resource*	oper,param
interface	resource+	resource*	oper,param
instance of <i>interf</i>	resource*	resource*	oper,param
incomplete concrete	concrete+	resource+	lin*,oper,param

The table uses the following shorthands for lists of module types and judgement forms:

- **resource***: resource, instance, concrete
- **resource+**: resource*, interface, abstract
- **concrete+**: concrete, incomplete concrete
- **lin***: lin, lincat, lindef, printname cat, printname fun

The validity of judgements in the body is checked with respect to judgement forms before the judgements themselves are checked.

In addition to the judgements listed in the above table, **flags** judgements are valid in all types of modules. The forms of judgement are explained in Section C.3.1.

C.2.6 Digression: the logic of module types

Why are the legality conditions of opens and extends so complicated? The best way to grasp them is probably to consider a simplified logical model of the module system, replacing modules by types and functions. This model could actually be developed towards treating modules in GF as first-class objects; so far, however, this step has not been taken since no practical needs have been encountered.

module	object and type
abstract $A = B$	$A = B : \text{type}$
concrete C of $A = B$	$C = B : A \rightarrow S$
interface $I = B$	$I = B : \text{type}$
instance J of $I = B$	$J = B : I$
incomplete concrete C of $A = \text{open } I \text{ in } B$	$C = B : I \rightarrow A \rightarrow S$
concrete K of $A = C$ with $(I = J)$	$K = C(J) : A \rightarrow S$
resource $R = B$	$R = B : I$
concrete C of $A = \text{open } R \text{ in } B$	$C = B(R) : A \rightarrow S$

The definition of modules as first-class objects would use GADT's and record types:

- an abstract syntax is a Generalized Algebraic Datatype (GADT)
- the target type **S** of concrete syntax is the universe of strings, parameters, records, and tables
- an interface is a labelled record type
- an instance is a record of the type defined by the interface
- a functor, with a module body opening an interface, is a function on the instances of the interface

- the instantiation of a functor is an application of the function to some instance
- a resource is a typed labelled record, putting together an interface and an instance of it
- the body of a module opening a resource is a function on the interface implicit in the resource; this function is immediately applied to the instance defined in the resource

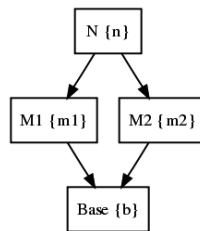
Slightly unexpectedly, interfaces and instances are easier to understand in this way than resources—a resource is, indeed, more complex, since it fuses together an interface and an instance.

When an abstract is used as an interface and a concrete as its instance, they are actually reinterpreted so that they match the model. Then the abstract is no longer a GADT, but a system of *abstract* datatypes, with a record field of type **Type** for each category, and a function among these types for each abstract syntax function. A concrete syntax instantiates this record with linearization types and linearizations.

C.2.7 Inheritance

After checking that the *extends* of a module are of appropriate module types, the compiler adds the inherited judgements to the judgements included in the body. The inherited judgements are not copied entirely, but only their names with links to the inherited module. Conflicts may arise in this process: a name can have two definitions in the combined pool of inherited and added judgements. Such a conflict is always an error: GF provides no way to redefine an inherited constant.

Simple as the definition of a conflict may sound, it has to take care of the inheritance hierarchy. A very common pattern of inheritance is the **diamond pattern**: inheritance from two modules which themselves inherit a common base module. Assume that we have four modules in a diamond position, each defining one new name:



Now, *N* inherits *b* from both *M1* and *M2*, so is there a conflict? The answer in GF is *no*, because the “two” *b*’s are in the end the same: the one defined in *Base*. The situation is thus simpler than in **multiple inheritance** in languages like C++, because definitions in GF are **immutable**: neither *M1* nor *M2* can possibly have changed the definition of *b* given in *Base*. In practice, the compiler manages inheritance through hierarchy in a very simple way, by

just always creating a link not to the immediate parent, but the original ancestor; this ancestor can be read from the link provided by the immediate parent. Here is how links are created from source modules by the compiler:

Base {b}	↓	Base {b}
M1 {m1}	↓	M1 {Base.b, m1}
M2 {m2}	↓	M2 {Base.b, m2}
N {n}	↓	N {Base.b, M1.m1, M2.m2, n}

Inheritance can be **restricted**. This means that a module can be specified as inheriting *only* explicitly listed constants, or all constants *except* ones explicitly listed. The syntax uses constant names in brackets, prefixed by a minus sign in the case of an exclusion list. In the following configuration, N inherits a,b,c from M1, and all names but d from M2

N = M1 [a,b,c], M2-[d]

Restrictions are performed as a part of inheritance linking, module by module: the link is created for a constant if and only if it is both included in the module and compatible with the restriction. Thus, for instance, an inadvertent usage can exclude a constant from one module but inherit it from another one. In the following configuration, f is inherited via M1, if M1 inherits it.

N = M1 [a,b,c], M2-[f]

Uncautious restricted inheritance may result in errors in the judgement-level dependency analysis phase. For instance, suppose a function f has category C as its type in M, and we only include f. The exclusion has the effect of creating an ill-formed module, because C is used in it but not defined:

```
abstract M = {cat C ; fun f : C ;}
M [f] ↓ {fun f : C ;}
```

One might expect inheritance restriction to be transitive: if an included constant b depends on some other constant a, then a should be included automatically. However, this rule would leave to hard-to-detect inheritances. And it could only be applied later in the compilation phase, when the compiler has not only collected the names defined, but also resolved the names used in definitions.

C.2.8 Opening

Opening makes constants from other modules usable in judgements, without inheriting them. This means that, unlike inheritance, opening is not transitive.

Opening cannot be restricted as inheritance can, but it can be **qualified**. This means that the names from the opened modules cannot be used as such, but only as prefixed by a qualifier and a dot (.). The qualifier can be any identifier, including the name of the module. Here is an example of an *opens* list:

open A, (X = XSLTS), (Y = XSLTS), B

If A defines the constant a, it can be accessed by the names

```
a      A.a
```

If `XSLTS` defines the constant `x`, it can be accessed by the names

```
X.x      Y.x      XSLTS.x
```

Thus qualification by real module name is always possible, and one and the same module can be qualified in different ways at the same time (the latter can be useful if you want to be able to change the implementations of some constants to a different resource later). Since the qualification with real module names is always possible, it is not possible to “swap” the names of modules locally:

```
open (A=B), (B=A) -- NOT LEGAL!
```

C.2.9 Name resolution

Name resolution is the compiler phase taking place after inheritance linking. It qualifies all names occurring in the definition parts of judgements (that is, just excluding the defined names themselves) with the names of the modules they come from. If a name can come from different modules (that is, not from their common ancestor), a conflict is reported; this decision is hence not dependent on e.g. types, which are known only at a later phase.

The qualification of names is the main device for avoiding conflicts in name resolution. No other information is used, such as priorities between modules. However, if a name is just defined in different opened modules but never used in the module body, a conflict does not arise: conflicts arise only when names are used. Also in this respect, opening is thus different from inheritance, where conflicts are checked independently of use.

As usual, inner scope has priority in name resolution. This means that if an identifier is in scope as a bound variable, it will not be interpreted as a constant, unless qualified by a module name (variable bindings are explained in Section C.4.2).

C.2.10 Functor instantiations

We have dealt with the principles of module headers, inheritance, and names in a general way that applies to all module types. The exception is functor instantiations, that have an extra part of the instantiating equations, assigning an instance to every interface. Here is a typical example, displaying the full generality:

```
concrete FoodsEng of Foods = CommentsEng **
  FoodsI-[Pizza] with
    (Syntax = SyntaxEng),
    (LexFoods = LexFoodsEng)
  open SyntaxEng, ParadigmsEng in {
    lin Pizza = mkCN (mkA "Italian") (mkN "pie") ;
  }
```

The instantiation syntax is similar to qualified *opens*. The left-hand-side names must be interfaces, the right-hand-side names their instances. (Re-

call, moreover, that **abstract** can be used as **interface** and **concrete** as its **instance**.) Inheritance from the functor can be restricted, typically in the purpose of defining some excluded functions in language-specific ways in the module body.

C.2.11 Completeness

(This section refers to the forms of judgement introduced in Section C.3.1.)

A **concrete** is complete with respect to an **abstract**, if it contains a **lincat** definition for every **cat** declaration, and a **lin** definition for every **fun** declaration.

The same completeness criterion applies to functor instantiations. It is not possible to use a partial functor instantiation, leading to another functor.

Functors do not need to be complete in the sense concrete modules need. Any missing definitions can be provided in the body of each functor instantiation. This means that a functor implementation can also be used for an abstract syntax that is not completely functorial—for instance, which contains functions that require different resource grammar structures in different languages.

A **resource** is complete, if all its **oper** and **param** judgements have a definition part. While a **resource** must be complete, an **interface** need not. In an **interface**, the definition parts of judgements are optional.

An **instance** is complete with respect to an **interface**, if it gives the definition parts of all **oper** and **param** judgements that lack definition parts in the **interface**. Giving definitions to judgements that have already been defined in the **interface** is illegal. Type signatures, on the other hand, can be repeated if the same types are used.

In addition to completing the definitions in an **interface**, an instance of it may contain other judgements, but these must all be complete with definitions.

Here is an example of an instance and its interface showing the above variations:

```
interface Pos = {
  param Case ;                -- no definition
  param Number = Sg | Pl ;    -- definition given
  oper Noun : Type = {        -- relative definition given
    s : Number => Case => Str
  } ;
  oper regNoun : Str -> Noun ; -- no definition
}

instance PosEng of Pos = {
  param Case = Nom | Gen ;    -- definition of Case
                                -- Number and Noun inherited
  oper regNoun = \dog -> {    -- type of regNoun inherited
    s = table {               -- definition of regNoun

```

judgement	where	module	default	base
<code>cat CT</code>	Γ context	abstract	N/A	yes
<code>fun $f : A$</code>	A type	abstract	N/A	yes
<code>def $fps = t$</code>	f fun, ps patterns, t term	abstract	yes	no
<code>data $C = f \mid \dots \mid g$</code>	C cat, $f \dots g$ fun	abstract	yes	no
<code>lincat $C = T$</code>	C cat, T type	concrete*	yes	yes
<code>lin $f = t$</code>	f fun, t term	concrete*	no	yes
<code>lindef $f = t$</code>	f fun, t term	concrete*	yes	no
<code>printname cat $C = t$</code>	C cat, t term	concrete*	yes	no
<code>printname fun $f = t$</code>	f fun, t term	concrete*	yes	no
<code>param $P = C \mid \dots \mid D$</code>	$C \dots D$ constructors	resource*	N/A	yes
<code>oper $f : T = t$</code>	T type, t term	resource*	N/A	yes
<code>flags $o = v$</code>	o flag, v value	any	yes	N/A

FIGURE 78 Forms of judgement in GF.

```

Sg => table {
  Nom => dog
  -- etc
}
};
oper house_N : Noun =          -- new definition
  regNoun "house" ;
}

```

C.3 Judgements

C.3.1 Overview of the forms of judgement

A module body in GF is a set of **judgements**. Judgements are definitions or declarations, sometimes combinations of the two; the common feature is that every judgement introduces a name, which is available in the module and whenever the module is extended or opened. Judgements of the **param** form introduce more than one name: both the type name and the constructor names.

There are several different **forms of judgement**, identified by different **judgement keywords**. The table in Figure 78 gives a list of all these forms, together with syntax descriptions and the types of modules in which each form can occur. The table moreover indicates whether the judgement has a default value, and whether it contributes to the **name base**, i.e. introduces a new name to the scope.

Judgements that have default values are rarely written by the user, except **lincat** and **flags**, which often need values different from the defaults.

Introducing a name twice in the same module is an error. In other words, all judgements that have a “yes” in the name base column, must have distinct identifiers on their left-hand sides.

All judgements are terminated with semicolons (;).

In addition to the syntax given in the table, many of the forms have

syntactic sugar. This sugar will be explained below in connection to each form. There are moreover two kinds of syntactic sugar common to all forms:

- the judgement keyword is shared between consecutive judgements until a new keyword appears:

$$\text{keyw } J ; K ; \equiv \text{keyw } J ; \text{keyw } K ;$$

- the right-hand sides of colon (:) and equality (=) can be shared, by using comma (,) as separator of left-hand sides, which must consist of identifiers

$$c, d : T \equiv c : T ; d : T ;$$

$$c, d = t \equiv c = t ; d = t ;$$

These conventions, like all syntactic sugar, are performed at an early compilation phase, directly after parsing. This means that e.g.

$$\text{lin } f, g = \backslash x \rightarrow x ;$$

can be correct even though f and g have different types.

Within a module, judgements can occur in any order. In particular, a name can be used before it is introduced.

The explanations of judgement forms refer to the notions of **type** and **term** (the latter also called **expression**). These notions will be explained in detail in Section C.4.1.

C.3.2 Category declarations, cat

Category declarations

$$\text{cat } C \ G$$

define the **basic types** of abstract syntax. A basic type is formed from a category by giving values to all variables in the **context** G . If the context is empty, the basic type looks the same as the category itself. Otherwise, application syntax is used:

$$C \ a_1 \dots a_n$$

C.3.3 Hypotheses and contexts

A context is a sequence of **hypotheses**, i.e. variable-type pairs. A hypothesis is written

$$(x : T)$$

and a sequence does not have any separator symbols. As syntactic sugar,

- variables can share a type;

$$(x, y : T) \equiv (x : T) (y : T)$$

- a **wildcard** can be used for a variable not occurring in types later in the context;

$$(\_ : T) \equiv (x : T)$$

- if the variable does not occur later, it can be omitted altogether, and parentheses are not used;

$$T \equiv (\_ : T)$$

(but of course, if T is a more complex expression than an identifier, it needs parentheses to be separated from the rest of the context).

A **dependent type** is a type built with a category whose context is non-empty.

C.3.4 Function declarations, fun

Function declarations,

$$\text{fun } f : T$$

define the **syntactic constructors** of abstract syntax. The type T of f is built from basic types (formed from categories) by using the function type constructor $\rightarrow$ (which we will also denote by the symbol $\rightarrow$ when speaking about code). Thus its form is

$$(x_1 : A_1) \rightarrow \dots \rightarrow (x_n : A_n) \rightarrow B$$

where $A_1, \dots, A_n$ are types, called the **argument types**, and B is a basic type, called the **value type** of f . The **value category** of f is the category that forms the type B .

A **syntax tree** is formed by f by applying f to a full list of arguments, so that the result is of a basic type. (See Section C.4.5 for more details on syntax trees.)

A **higher-order function** is one that has a function type as an argument. The concrete syntax of GF does not support displaying the bound variables of functions of higher than second order, but they are legal in abstract syntax.

An abstract syntax is **context-free**, if it has neither dependent types nor higher-order functions. Grammars with context-free abstract syntax are an important subclass of GF, with more limited complexity than full GF. Whether the *concrete* syntax is context-free in the sense of the Chomsky hierarchy is independent of the context-freeness of the abstract syntax.

C.3.5 Function definitions, def

Function definitions,

$$\text{def } f \ p_1 \ \dots \ p_n = t$$

where f is a **fun** function and p_i are patterns, impose a relation of **definitional equality** on abstract syntax trees. They form the basis of **computation**, which is used when comparing whether two types are equal; this notion is relevant only if the types are dependent. Computation can also be used for the **normalization** of syntax trees, which applies even in context-free abstract syntax.

The set of **def** definitions for f can be scattered around the module in which f is introduced as a function. The compiler builds the set of pattern equations in the order in which the equations appear; this order is significant in the case of overlapping patterns. All equations must appear in the same module in which f itself is declared.

The syntax of patterns will be specified in Section C.4.13, commonly for abstract and concrete syntax. In abstract syntax, **constructor patterns** are

those of the form

$$C \ p_1 \ \dots p_n$$

where C is declared as **data** for some abstract syntax category (see next section). A **variable pattern** is either an identifier or a wildcard.

A common pitfall is to forget to declare a constructor as data, which causes it to be interpreted as a variable pattern in definitions.

Computation is performed by applying definitions and beta conversions, and in general by using **pattern matching**. Computation and pattern matching are explained commonly for abstract and concrete syntax in Section C.4.13.

In contrast to concrete syntax, abstract syntax computation is completely **symbolic**: it does not produce a value, but just another term. Hence it is not an error to have incomplete systems of pattern equations for a function. In addition, the definitions can be **recursive**, and the abstract syntax is hence a Turing-complete language. This means that computation can fail to terminate, which can never happen in concrete syntax.

C.3.6 Data constructor declarations, data

A **data constructor declaration**,

$$\mathbf{data} \ C = f_1 \mid \dots \mid f_n$$

defines the functions $f_1 \dots f_n$ to be **constructors** of the category C . This means that they are recognized as constructor patterns when used in function definitions.

In order for the data constructor definition to be correct, $f_1 \dots f_n$ must be functions with C as their value category.

The complete set of constructors for a category C is the union of all its data constructor definitions. Thus a category can be “extended” by new constructors “afterwards”. However, all these constructor definitions must appear in the same module in which the category is itself defined.

There is syntactic sugar for declaring a function as a constructor at the same time as introducing it:

$$\begin{aligned} \mathbf{data} \ f : A_1 \rightarrow \dots \rightarrow A_n \rightarrow C \ t_1 \dots t_m \\ \equiv \\ \mathbf{fun} \ f : A_1 \rightarrow \dots \rightarrow A_n \rightarrow C \ t_1 \dots t_m ; \mathbf{data} \ C = f \end{aligned}$$

C.3.7 The semantic status of an abstract syntax function

There are three possible statuses for a function declared in a **fun** judgement:

- primitive notion: the default status
- constructor: the function appears on the right-hand side in a **data** judgement
- defined: the function has a **def** definition

The “constructor” and “defined” statuses are in contradiction with each other, whereas the primitive notion status is overridden by any of the two others.

This distinction is relevant for the semantics of abstract syntax, not for concrete syntax. It shows in the way patterns are treated in equations in **def** definitions: a constructor in a pattern matches only itself, whereas any other name is treated as a variable pattern, which matches anything.

C.3.8 Linearization type definitions, **lincat**

A linearization type definition,

$$\text{lincat } C = T$$

defines the **linearization type** of category C , i.e. the type of the linearizations of trees whose type has category C . Type dependences have no effect on the linearization type.

The type T must be a **legal linearization type**, which means that it is built from the type **Str** of strings, from parameter types, and from table and record types. In particular, function types may not appear in T . A detailed explanation of types in concrete syntax will be given in Section C.4.7.

If K is the concrete syntax of an abstract syntax A , then K must define the linearization types of all categories declared in A . However, the definition can be omitted from the source code, in which case the type $\{s : \text{Str}\}$ is used by default.

C.3.9 Linearization definitions, **lin**

A linearization definition,

$$\text{lin } f = t$$

defines the **linearization function** of function f , i.e. the function used for linearizing trees formed by f .

The type of the linearization function t must be the homomorphic image of the type of f . In other words, if

$$\text{fun } f : A_1 \rightarrow \dots \rightarrow A_n \rightarrow A$$

then

$$\text{lin } f : A_1^* \rightarrow \dots \rightarrow A_n^* \rightarrow A^*$$

where the type T^* is defined as follows depending on T :

- $(C \ t_1 \dots t_n)^* = T$, if $\text{lincat } C = T$
- $(B_0 \rightarrow \dots \rightarrow B_m \rightarrow B)^* = B^* ** \{\$0, \dots, \$m : \text{Str}\}$

The second case is relevant for higher-order functions only. It says that the linearization type of the value type is extended by adding a string field for each argument types; these fields store the variable symbol used for the binding of each variable.

Since the arguments of a function argument are treated as bare strings, orders higher than the second are irrelevant for concrete syntax.

There is syntactic sugar for binding the variables of the linearization of a function on the left-hand side:

$$\text{lin } f \ p = t \equiv \text{lin } f = \backslash p \rightarrow t$$

The pattern p must be either a variable or a wildcard ($\_$), which follows the syntax of lambda abstracts.

If K is the concrete syntax of an abstract syntax A , then K must define the linearizations of all functions declared in A . However, if the value type of linearization is the empty record type $\{\}$, the empty record $\langle \rangle$ is inserted automatically, as it is the only possible linearization in this type.

C.3.10 Linearization default definitions, `lindef`

A linearization default definition,

$$\text{lindef } C = t$$

defines the **default linearization** of category C , i.e. the function applicable to a string to make it into an object of the linearization type of C .

Linearization defaults are invoked when linearizing variable bindings in higher-order abstract syntax. A variable symbol is then presented as a string, which must be converted to correct type in order for linearization not to fail with an error. The defaults are also used for linearizing metavariables.

Usually, linearization defaults are generated by using the default rule that uses the symbol itself for every string, and the first value of the parameter type for every parameter. The precise definition is by structural recursion on the type:

- $\text{default}(\text{Str}, s) = s$
- $\text{default}(P, s) = \#1(P)$
- $\text{default}(P \Rightarrow T, s) = \_ \Rightarrow \text{default}(T, s)$
- $\text{default}(\{\dots; r : R; \dots\}, s) = \{\dots; r : \text{default}(R, s); \dots\}$

The notion of the first value of a parameter type ($\#1(P)$) is defined in Section C.3.13 below.

C.3.11 Printname definitions, `printname cat/fun`

A category printname definition,

$$\text{printname cat } C = s$$

defines the **printname** of category C , i.e. the name usable by application programs to show information to the user.

Likewise, a **function printname definition**,

$$\text{printname fun } f = s$$

defines the printname of function f , i.e. the name used in some abstract syntax information shown to the user.

The most common use of printnames is in the interactive syntax editor, where printnames are displayed in menus. It is possible e.g. to adapt them to each language, or to embed HTML tooltips in them (as is used in some HTML-based editor GUI's).

By default, printnames are generated automatically from the symbol and/or concrete syntax information.

C.3.12 Parameter type definitions, `param`

A **parameter type definition**,

$$\text{param } P = C_1 \ G_1 \mid \dots \mid C_n \ G_n$$

defines a parameter type P with the **parameter constructors** $C_1 \dots C_n$, with their respective contexts $G_1 \dots G_n$.

Contexts have the same syntax as in `cat` judgements, explained in Section C.3.2. Since dependent types are not available in parameter type definitions, the use of variables is never necessary. The types in the context must themselves be **parameter types**, which are defined as follows:

- Given the judgement `param P ...`, P is a parameter type.
- A record type of parameter types is a parameter type.
- `Ints n` (integers from 0 to n) is a parameter type.

The names defined by a parameter type definition include both the type name P and the constructor names C_i . Therefore all these names must be distinct in a module.

A parameter type may not be recursive, i.e. P itself may not occur in the contexts of its constructors. This restriction extends to mutual recursion: we say that P **depends** on the types that occur in the contexts of its constructors and on all types that those types depend on, and state that P may not depend on itself.

In an **interface** module, it is possible to declare a parameter type without defining it,

$$\text{param } P ;$$

C.3.13 Parameter values

All parameter types are finite, and the GF compiler will internally compute them to **lists of parameter values**. These lists are formed by traversing the `param` definitions, usually respecting the order of constructors in the source code. For records, bibliographical sorting is applied. However, both the order of traversal of `param` definitions and the order of fields in a record are specified in a compiler-internal way, which means that the programmer should not rely on any particular order.

The order of the list of parameter values can affect the program in two cases:

- in the default `lindex` definition (see Section C.3.10), the first value is chosen
- in course-of-value tables (see Section C.4.12), the compiler-internal order is followed

The first usage implies that, if `lindex` definitions are essential for the application, they should be given manually. The second usage implies that course-of-value tables should be avoided in hand-written GF code.

In run-time grammar generation, all parameter values are translated to integers denoting positions in these parameter lists.

C.3.14 Operation definitions, `oper`

An **operation definition**,

$$\text{oper } h : T = t$$

defines an **operation** h of type T , with the computation rule

$$h \Downarrow t$$

The type T can be any concrete syntax type, including function types of any order. The term t must have the type T , as defined in Section C.4.1.

As syntactic sugar, the type can be omitted,

$$\text{oper } h = t$$

which works if either of these two conditions hold:

- the type can be inferred from t (compiler-dependent);
- the definition occurs in an **instance** and the type is given in the **interface**.

It is also possible to give the type and the definition separately:

$$\text{oper } h : T ; \text{oper } h = t \equiv \text{oper } h : T = t$$

The order of the type part and the definition part is free, and there can be other judgements in between. However, they must occur in the same **resource** module for it to be complete (as defined in Section C.2.11). In an **interface** module, it is enough to give just the type without a definition.

When only the definition is given, it is possible to use a shorthand similar to `lin` judgements:

$$\text{oper } h \ p = t \equiv \text{oper } h = \backslash p \rightarrow t$$

The pattern p is either a variable or a wildcard (`_`).

Operation definitions may not be recursive, not even mutually recursive. This condition ensures that functions can be eliminated from concrete syntax code at compile time (as explained in Section C.4.16).

C.3.15 Operation overloading

One and the same operation name h can be used for different operations, which have to have different types. For each call of h , the type checker selects one of these operations depending on what type is expected in the context of the call. The syntax of overloaded operation definitions doesn't allow separate **oper** definitions, but requires them to be included in one and the same **overload group**, which has the following record-like syntax:

$$\text{oper } h = \text{overload } \{h : T_1 = t_1 ; \dots ; h : T_n = t_n\}$$

Notice that h must be the same in all cases. This format can be used to give the complete implementation; to give just the types, e.g. in an interface, one can use the form

$$\text{oper } h : \text{overload } \{h : T_1 ; \dots ; h : T_n\}$$

The implementation of this operation typing is given by a judgement of the first form. The order of branches need not be the same.

The compiler performs **overload resolution** to choose the correct instance of an overloaded function and replace it by its definiens. This process is guided by the type of the function application. The rules for resolving an application ($f\ a_1 \dots a_n$) are as follows:

1. Infer the types of the arguments $a_1 \dots a_n$
2. If there is a unique instance of f with the same list of argument types, use this instance.
3. If there are several instances, use the expected value type of the application to resolve overloading.

This resolution method always gives a unique result, because an overload group cannot contain more than one function of the same type.

Notice that overload resolution requires exact equality of types, and doesn't therefore accept subtypes (cf. Section C.4.11).

C.3.16 Flag definitions, flags

A **flag definition**,

flags $o = v$

sets the value of the flag o , to be used when compiling or using the module.

The flag o is an identifier, and the value v is either an identifier or a quoted string.

Flags are a kind of metadata, which do not strictly belong to the GF language. For instance, compilers do not necessarily check the consistency of flags, or the meaningfulness of their values. The inheritance of flags is not well-defined; it is not even guaranteed that flags set in the module body override the settings from inherited modules.

The following flags are commonly included in grammars:

- **coding**, defining the character encoding of Unicode string literals (in any module containing string literals)
- **startcat**, define the default target category of parsing (in an **abstract** module)

Some possible values of these flags are specified in Section C.5.1.

C.4 Types and expressions

C.4.1 Overview of expression forms

Like many other dependently typed languages, GF makes no syntactic distinction between expressions and types. An illegal use of a type as an expression or vice versa comes out as a type error. Whether a variable, for instance, stands for a type or an expression value, can only be resolved from its context of use. Types are declared as expressions of type **Type**.

One practical consequence of the common syntax is that global and local definitions (**oper** judgements and **let** expressions, respectively) work in the same way for types and expressions. Thus it is possible to abbreviate a type occurring in a type expression:

let A = {s : Str ; b : Bool} in A -> A -> A

Type and other expressions have a system of **precedences**. The table in Figure C.4.1 summarizes all expression forms, from the highest to the lowest precedence. Some expressions are moreover left- or right-associative. Any expression in parentheses (*exp*) is in the highest precedence class.

C.4.2 The functional fragment: expressions in abstract syntax

The expression syntax is the same in abstract and concrete syntax, although only a part of the syntax is actually usable in well-typed expressions in abstract syntax. An abstract syntax is used for defining a set of types and a set of functions between those types. Therefore it needs only the **functional fragment** of the syntax. This fragment comprises two kinds of types:

- basic types, of form $C\ a_1 \dots a_n$, where
 - **cat** $C\ (x_1 : A_1) \dots (x_n : A_n)$, including the predefined categories **Int**, **Float**, and **String** explained in Section C.4.6
 - $a_1 : A_1, \dots, a_n : A_n \{x_1 = a_1, \dots, x_{n-1} = a_{n-1}\}$
- function types, **function type** of form $(x : A) \rightarrow B$, where
 - A is a type
 - B is a type possibly depending on $x : A$

When defining basic types, we used the notation $t\{x_1 = t_1, \dots, x_n = t_n\}$ for the **substitution** of values to variables. This is a metalevel notation, which denotes a term that is formed by replacing the free occurrences of each variable x_i by t_i . Abstract syntax has six kinds of expressions:

- constants, $f : A$, where **fun** $f : A$
- literals, for integers, floats, and strings (defined in Section C.4.6)
- variables, $x : A$, where x has been introduced by a binding
- applications, $f\ a : B\{x=a\}$, where
 - $f : (x : A) \rightarrow B$
 - $a : A$
- abstractions, $\backslash x \rightarrow b : (x : A) \rightarrow B$, where $b : B$ possibly depending on $x : A$
- metavariables, $?n$, as introduced in intermediate phases of incremental type checking; metavariables are not permitted in GF source code. Bare $?$ is syntactic sugar for $?0$.

The notion of **binding** is defined for occurrences of variables in subexpressions as follows:

- in $(x : A) \rightarrow B$, x is bound in B
- in $\backslash x \rightarrow b$, x is bound in b
- in **def** $f\ p_1 \dots p_n = t$, any pattern variable introduced in any p_1 is bound in t (as defined in Section C.4.13)

As syntactic sugar, function types have sharing of types and suppression of variables, in the same way as contexts (defined in Section C.3.3):

prec	expression example	explanation
6	c	constant or variable
6	Type	the type of types
6	PType	the type of parameter types
6	Str	the type of strings/token lists
6	"foo"	string literal
6	123	integer literal
6	0.123	floating point literal
6	?32	metavariable
6	[]	empty token list
6	[C a b]	list category
6	["foo bar"]	token list
6	{s : Str ; n : Num}	record type
6	{s = "foo" ; n = Sg}	record
6	<Sg,Fem,Gen>	tuple
6	<n : Num>	type-annotated expression
5 left	t.r	projection or qualification
4 left	f a	function application
4	table {Sg => [] ; _ => "xs"}	table
4	table P [a ; b ; c]	course-of-values table
4	case n of {Sg => [] ; _ => "xs"}	case expression
4	variants {"color" ; "colour"}	free variation
4	pre {"a" "i" => "an" ; _ => "a"}	prefix-dependent choice
4	lincat C {s : Str}	linearization type of C
4	lin C {s = "even"}	object of lincat C
4	pattern P	pattern of type P
3 left	t ! v	table selection
3 left	A * B	product type
3 left	R ** {b : Bool}	record (type) extension
3	#("a" "i")	pattern in macro definition
2 right	t + s	token gluing
1 right	t ++ s	token list concatenation
0 right	\x,y -> t	function abstraction
0 right	\x,y => t	table abstraction
0 right	(x : A) -> B	dependent function type
0 right	A -> B	function type
0 right	P => T	table type
0 right	let x = v in t	local definition
0 left	"color" "colour"	free variation
0	t where {x = v}	local definition

FIGURE 79 GF expressions with precedences and associativities.

- variables can share a type,

$$(x, y : A) \rightarrow B \equiv (x : A) \rightarrow (y : A) \rightarrow B$$

- a **wildcard** can be used for a variable not occurring later in the type,

$$(\_ : A) \rightarrow B \equiv (x : A) \rightarrow B$$

- if the variable does not occur later, it can be omitted altogether, and parentheses are not used,

$$A \rightarrow B \equiv (\_ : A) \rightarrow B$$

There is analogous syntactic sugar for multiple lambda abstractions

$$\backslash x, y \rightarrow t \equiv \backslash x \rightarrow \backslash y \rightarrow t$$

where x and y are variables or wildcards ($\_$), and for lambda abstracts denoting constant functions,

$$\backslash \_ \rightarrow t \equiv \backslash x \rightarrow t$$

C.4.3 List categories

Since categories of lists of elements of another category are a common idiom, the following syntactic sugar is available:

```
cat [C] {n}
```

abbreviates a set of three judgements:

```
cat List C ;
fun Base C : C → ... → C → List C ; -- n C's
fun Cons C : C → List C → List C
```

The functions **Base** C and **Cons** C are automatically generated in the abstract syntax, but their linearizations, as well as the linearization type of **List** C , must be defined manually. The type expression $[C]$ is in all contexts interchangeable with **List** C .

C.4.4 Conversions

Among expressions, there is a relation of **definitional equality** defined by four **conversion rules**:

- alpha conversion: $\backslash x \rightarrow b \Downarrow \backslash y \rightarrow b\{x=y\}$
- beta conversion: $(\backslash x \rightarrow b) a \Downarrow b\{x=a\}$
- delta conversion: $f a_1 \dots a_n \Downarrow tg$, if
 - there is a definition **def** $f p_1 \dots p_n = t$
 - this definition is the first for f that matches the sequence $a_1 \dots a_n$, with the substitution g
- eta conversion: $c \Downarrow \backslash x \rightarrow c x$, if $c : (x : A) \rightarrow B$

Pattern matching substitution used in delta conversion is defined in Section C.4.13.

An expression is in **beta-eta normal form** if

- it has no subexpressions to which beta conversion applies (beta normality)

- each constant or variable whose type is a function type is **eta-expanded**, i.e. made into an abstract equal to it by eta conversion (eta normality)

Notice that a second iteration of eta expansion would lead to an expression that is no longer in beta-normal form.

C.4.5 Syntax trees

The **syntax trees** defined by an abstract syntax are well-typed expressions of basic types in beta-eta normal form. Linearization defined in concrete syntax applies to all and only these expressions.

There is also a direct definition of syntax trees, which does not refer to beta and eta conversions: keeping in mind that a type always has the form

$$(x_1 : A_1) \rightarrow \dots \rightarrow (x_n : A_n) \rightarrow B$$

where A_i are types and B is a basic type, a syntax tree is an expression

$$b \ t_1 \dots t_n : B'$$

where

- B' is the basic type $B\{x_1 = t_1, \dots, x_n = t_n\}$
- **fun** $b : (x_1 : A_1) \rightarrow \dots \rightarrow (x_n : A_n) \rightarrow B$
- each t_i has the form $\backslash z_1, \dots, z_m \rightarrow c$ if A_i is $(y_1 : B_1) \rightarrow \dots \rightarrow (y_m : B_m) \rightarrow C$

C.4.6 Predefined types in abstract syntax

GF provides three predefined categories for abstract syntax, with predefined expressions:

category	expressions	example
Int	integer literals	123
Float	floating point literals	12.34
String	string literals	"foo"

These categories take no arguments, and they can be used as basic types in the same way as if they were introduced in **cat** judgements. However, it is not legal to define **fun** functions that have any of these types as value type: their only well-typed expressions are literals as defined in the above table (as well as, like for all types, bound variables and metavariables).

C.4.7 Overview of expressions in concrete syntax

Concrete syntax is about defining mappings from abstract syntax trees to **concrete syntax objects**. These objects comprise

- records
- tables
- strings
- parameter values

Thus functions are not concrete syntax objects; however, the mappings themselves are expressed as functions, and the source code of a concrete syntax can use functions under the condition that they can be eliminated from the final

compiled grammar (which they can; this is one of the fundamental properties of compilation, as explained in more detail in the *JFP* article).

Concrete syntax thus has the same function types and expression forms as abstract syntax, specified in Section C.4.2. The basic types defined by categories (**cat** judgements) are available via grammar reuse explained in Section C.4.17; this also comprises the predefined categories **Int**, **Float**, and **String**.

C.4.8 Values, canonical forms, and run-time variables

In abstract syntax, the conversion rules given in Section C.4.4 define a computational relation among expressions, but there is no separate notion of a **value** of computation: the value (the end point) of a computation chain is simply an expression to which no more conversions apply (except *eta*). In general, we are interested in expressions that satisfy the conditions of being syntax trees (as defined in Section C.4.5), but there can be many computationally equivalent syntax trees which nonetheless are distinct syntax trees and hence have different linearizations. The main use of computation in abstract syntax is to compare types in dependent type checking. But it can also be used for paraphrasing and transfer.

In concrete syntax, the notion of values is central. At run time, we want to compute the values of linearizations; at compile time, we want to perform **partial evaluation**, which computes expressions as far as possible. To specify what happens in computation we therefore have to distinguish between **canonical forms** and other forms of expressions. The canonical forms are defined separately for each form of type, whereas the other forms may usually produce expressions of any type.

What is done at compile time is the elimination of any noncanonical forms, except for those depending on **run-time variables**. Run-time variables are the same as the **argument variables** of linearization rules, i.e. the variables $x_1, \dots, x_n$ in

$$\text{lin } f = \setminus x_1, \dots, x_n \rightarrow t$$

where

$$\text{fun } f : (x_1 : A_1) \rightarrow \dots \rightarrow (x_n : A_n) \rightarrow B$$

Notice that this definition refers to the **eta-expanded** linearization term, which has one abstracted variable for each argument type of f . These variables are not necessarily explicit in GF source code, but introduced by the compiler.

Since certain expression forms should be eliminated in compilation but cannot be eliminated if run-time variables appear in them, errors can appear late in compilation. This is an issue with the following expression forms:

- gluing (**s** + **t**), defined in Section C.4.9
- pattern matching on strings, defined in Section C.4.13
- predefined string operations, defined in Section C.4.18 (those taking **Str** arguments)

C.4.9 Token lists, tokens, and strings

The most prominent basic type is **Str**, the type of **token lists**.

This type is often sloppily referred to as the type of **strings**; but it should be kept in mind that the objects of **Str** are *lists* of strings rather than single strings.

Expressions of type **Str** have the following canonical forms:

- tokens, i.e. **string literals**, in double quotes, e.g. `"foo"`
- the empty token list, `[]`
- concatenation, $s ++ t$, where $s, t : \mathbf{Str}$
- prefix-dependent choice, **pre** $\{ p_1 \Rightarrow s_1 ; \dots ; p_n \Rightarrow s_n ; \_ \Rightarrow s \}$, where
 - $s_1, \dots, s_n, s : \mathbf{Str}$
 - $p_1, \dots, p_n$ are patterns of type **Str**

For convenience, string literals are overloaded so that tokens are identified with singleton token lists, and there is no separate type of tokens (this is a change from the *JFP* article). The notion of a token is still important for compilation: all tokens introduced by the grammar must be known at compile time. This, in turn, is required by the parsing algorithms used for parsing with GF grammars.

In addition to string literals, tokens can be formed by a specific non-canonical operator:

- **gluing**, $s + t$, where $s, t : \mathbf{Str}$

Being noncanonical, gluing is equipped with a computation rule: string literals are glued by forming a new string literal, and empty token lists can be ignored:

- `"foo" + "bar"  $\Downarrow$  "foobar"`
- `t + []  $\Downarrow$  t`
- `[] + t  $\Downarrow$  t`

Since tokens must be known at compile time, the operands of gluing may not depend on run-time variables, as defined in Section C.4.8.

As syntactic sugar, token lists can be given as string literals, where spaces separate tokens:

- token lists, `"one two three"  $\equiv$  "one" ++ "two" ++ "three"`

Notice that there are no empty tokens, but the expression `[]` can be used in a context requiring a token, in particular in gluing expression below. Since `[]` denotes an empty token list, the following computation laws are valid:

- `t ++ []  $\Downarrow$  t`
- `[] ++ t  $\Downarrow$  t`

Moreover, concatenation and gluing are associative:

- `s + (t + u)  $\Downarrow$  s + t + u`
- `s ++ (t ++ u)  $\Downarrow$  s ++ t ++ u`

For the programmer, associativity and the empty token laws mean that the compiler can use them to simplify string expressions. It also means that these laws are respected in pattern matching on strings.

A prime example of prefix-dependent choice operation is the following approximative expression for the English indefinite article:

```
pre {"a" | "e" | "i" | "o" => "an" ; _ => "a"}
```

This expression can be computed in the context of a subsequent token:

- **pre** { $p_1 \Rightarrow s_1 ; \dots ; p_n \Rightarrow s_n ; \_ \Rightarrow s$ } $++ t \Downarrow$
 - s_i for the first i such that the pattern p_i matches some prefix of t , if it exists
 - s otherwise

The computation rule can sometimes be applied at compile time, but in general, prefix-dependent choices need to be passed to the run-time grammar, because they are not given a subsequent token to compare with, or because the subsequent token depends on a run-time variable.

The prefix-dependent choice expression itself may not depend on run-time variables.

C.4.10 Records and record types

A **record** is a collection of objects of possibly different types, accessible by **projections** from the record with **labels** pointing to these objects. A record is also itself an object, whose type is a **record type**. Record types have the form

$$\{ r_1 : A_1 ; \dots ; r_n : A_n \}$$

where $n \geq 0$, each A_i is a type, and the labels r_i are distinct. A record of this type has the form

$$\{ r_1 = a_1 ; \dots ; r_n = a_n \}$$

where $a_i : A_i$ for every i . A limiting case is the **empty record type** $\{\}$, which has the object $\langle \rangle$, the **empty record**. (Logically, also $\{\}$ could be used to denote the empty record, but the empty tuple notation $\langle \rangle$ is recommended since it is clearly distinct from the record type in all contexts.) The **fields** of a record type are its parts of the form $r : A$, also called **typings**. The **fields** of a record are of the form $r = a$, also called **value assignments**. Value assignments may optionally indicate the type, as in $r : A = a$.

The order of fields in record types and records is insignificant: two record types (or records) are equal if they have the same fields, in any order, and a record is an object of a record type, if it has type-correct value assignments for all fields of the record type. The latter definition implies the even stronger principle of **record subtyping**: a record can have any type that has some subset of its fields. This principle is explained further in Section C.4.11.

All fields in a record must have distinct labels. Thus it is not possible e.g. to “redefine” a field “later” in a record. This follows from the principle that the order of fields doesn’t matter.

Lexically, labels are identifiers (defined in Section C.2.3). This is with the exception of the labels selecting bound variables in the linearization of higher-order abstract syntax, which have the form $\$i$ for an integer i , as

specified in Section C.3.9. In source code, these labels may not appear in record expressions, but only in projections.

Labels occur only in syntactic positions where they cannot be confused with constants or variables. Therefore it is safe to write, as in **Prelude**,

```
ss : Str -> {s : Str} = \s -> {s = s} ;
```

A **projection** is an expression of the form

$$t.r$$

where t must have a record type and r must be a label defined in it. The type of the projection is the type of that field. The computation rule for projection returns the value assigned to that field:

$$\{ \dots ; r = a ; \dots \}.r \Downarrow a$$

Notice that the dot notation $t.r$ is also used for qualified names as specified in Section C.2.8. This ambiguity follows tradition and convenience. It is resolved by the following rules (before type checking):

1. if t is a bound variable or a constant in scope, $t.r$ is type-checked as a projection
2. otherwise, $t.r$ is type-checked as a qualified name

As syntactic sugar, types and values can be shared:

- $\{ \dots ; r, s : A ; \dots \} \equiv \{ \dots ; r : A ; s : A ; \dots \}$
- $\{ \dots ; r, s = a ; \dots \} \equiv \{ \dots ; r = a ; s = a ; \dots \}$

Another syntactic sugar are **product types** (also known as **tuple types**) and **tuples**, which are translated by endowing their unlabelled fields by the labels **p1**, **p2**,... in the order of appearance of the fields:

- $A_1 * \dots * A_n \equiv \{ \mathbf{p1} : A_1 ; \dots ; \mathbf{pn} : A_n \}$
- $\langle a_1, \dots, a_n \rangle \equiv \{ \mathbf{p1} = a_1 ; \dots ; \mathbf{pn} = a_n \}$

A **record extension** is formed by adding fields to a record or a record type. The general syntax involves two expressions,

$$R ** S$$

The result is a record type or a record with a union of the fields of R and S . It is therefore well-formed if

- both R and S are either records or record types
- the labels in R and S are distinct

C.4.11 Subtyping

The possibility of having superfluous fields in a record forms the basis of the **subtyping** relation. That A is a subtype of B means that $a : A$ implies $a : B$. This is clearly satisfied for records with superfluous fields:

- if R is a record type without the label r , then $R ** \{ r : A \}$ is a subtype of R

The GF grammar compiler extends subtyping to function types by **covariance** and **contravariance**:

- covariance: if A is a subtype of B , then $C \rightarrow A$ is a subtype of $C \rightarrow B$
- contravariance: if A is a subtype of B , then $B \rightarrow C$ is a subtype of $A \rightarrow C$

The logic of these rules is natural: if a function returns a value in a subtype, then this value is *a fortiori* in the supertype. If a function is defined for some type, then it is *a fortiori* defined for any subtype.

In addition to the well-known principles of record subtyping and co- and contravariance, GF implements subtyping for initial segments of integers:

- if $m < n$, then **Ints** m is a subtype of **Ints** n
- **Ints** n is a subtype of **Integer**

As the last rule, subtyping is transitive:

- if A is a subtype of B and B is a subtype of C , then A is a subtype of C .

C.4.12 Tables and table types

One of the most characteristic constructs of GF is **tables**, also called **finite functions**. That these functions are finite means that it is possible to finitely enumerate all argument-value pairs; this, in turn, is possible because the argument types are finite.

A **table type** has the form

$$P \Rightarrow T$$

where P must be a parameter type in the sense defined in Section C.3.12, whereas T can be any type.

Canonical expressions of table types are **tables**, of the form

$$\text{table } \{ V_1 \Rightarrow t_1 ; \dots ; V_n \Rightarrow t_n \}$$

where $V_1, \dots, V_n$ is the complete list of the parameter values of the argument type P (defined in Section C.3.13), and each t_i is an expression of the value type T .

In addition to explicit enumerations, tables can be given by **pattern matching**,

$$\text{table } \{ p_1 \Rightarrow t_1 ; \dots ; p_m \Rightarrow t_m \}$$

where $p_1, \dots, p_m$ is a list of patterns that covers all values of type P . Each pattern p_i may bind some variables, on which the expression t_i may depend. A complete account of patterns and pattern matching is given in Section C.4.13.

A **course-of-values table** omits the patterns and just lists all values. It uses the enumeration of all values of the argument type P to pair the values with arguments:

$$\text{table } P [t_1 ; \dots ; t_n]$$

This format is not recommended for GF source code, since the ordering of parameter values is not specified and therefore a compiler-internal decision.

The argument type can be indicated in ordinary tables as well, which is sometimes needed for type inference:

$$\text{table } P \{ \dots \}$$

The **selection** operator $!$, applied to a table t and to an expression v of its argument type

$$t ! v$$

returns the first pattern matching result from t with v , as defined in Section C.4.13. The order of patterns is thus significant whenever patterns may overlap, for instance, by containing variables or wildcards. When the compiler reorders the patterns following the enumeration of all values of the argument type, this order no longer matters, because no overlap remains between patterns.

The GF compiler performs **table expansion**, i.e. an analogue of eta expansion defined in Section C.4.4, where a table is applied to all values to its argument type:

$$t : P \Rightarrow T \Downarrow \text{table } P [t ! V_1 ; \dots ; t ! V_n]$$

As syntactic sugar, one-branch tables can be written in a way similar to lambda abstractions:

$$\backslash \backslash p \Rightarrow t \equiv \text{table } \{p \Rightarrow t\}$$

where p is either a variable or a wildcard ($\_$). Multiple bindings can be abbreviated:

$$\backslash \backslash p, q \Rightarrow t \equiv \backslash \backslash p \Rightarrow \backslash \backslash q \Rightarrow t$$

Case expressions are syntactic sugar for selections:

$$\text{case } e \text{ of } \{\dots\} \equiv \text{table } \{\dots\} ! e$$

Pattern matching over strings uses the syntax of case expressions, but is technically speaking different from table selection, because **Str** is not a parameter type whose values can be finitely enumerated. When such a case expression, of form

$$\text{case } e \text{ of } \{\dots p \Rightarrow t \dots\}$$

is checked with respect to a type A , each pattern p is checked with respect to the type of the expression e , and each t is checked with respect to the type A . But no reordering of the branches takes place, and completeness is not checked. Notice that the type of e can be not just **Str**, but also a record (or a tuple) with **Str** and parameter type components.

C.4.13 Pattern matching

We will list all forms of patterns that can be used in table branches. We define their **variable bindings** and **matching substitutions**.

We start with the patterns available for all parameter types, as well as for the types **Integer** and **Str**.

- A **constructor pattern** $C p_1 \dots p_n$ binds the union of all variables bound in the subpatterns $p_1, \dots, p_n$. It matches any value $C V_1 \dots V_n$ where each p_i matches V_i , and the matching substitution is the union of these substitutions.

- A **record pattern** $\{ r_1 = p_1 ; \dots ; r_n = p_n \}$ binds the union of all variables bound in the subpatterns $p_1, \dots, p_n$. It matches any value $\{ r_1 = V_1 ; \dots ; r_n = V_n ; \dots \}$ where each p_i matches V_i , and the matching substitution is the union of these substitutions.
- A **variable pattern** x (identifier other than parameter constructor) binds the variable x . It matches any value V , with the substitution $\{x = V\}$.
- The **wild card pattern** $_$ binds no variables. It matches any value, with the empty substitution.
- A **disjunctive pattern** $p \mid q$ binds the intersection of the variables bound by p and q . It matches anything that either p or q matches, in the order of priority p, q . Those variables that are not bound by both patterns are not bound.
- A **negative pattern** $- p$ binds no variables. It matches anything that p does *not* match, with the empty substitution.
- An **alias pattern** $x @ p$ binds x and all the variables bound by p . It matches any value V that p matches, with the same substitution extended by $\{x = V\}$.
- A **pattern macro** $\#c$ is expanded to a pattern by looking up the definition of the **oper** constant c . If it is a **pattern expression** $\#p$, matching is performed by p .

The following patterns are only available for the type **Str**:

- A **string literal pattern**, e.g. `"s"`, binds no variables. It matches the same string, with the empty substitution.
- The **one-character pattern**, `?`, matches a string whose length is exactly one (32-bit Unicode) character.
- A **concatenation pattern**, $p + q$, binds the union of variables bound by p and q . It matches any string that consists of a prefix matching p and a suffix matching q , with the union of substitutions corresponding to the first match (see below).
- A **repetition pattern** p^* binds no variables. It matches any string that can be decomposed into strings that match p , with the empty substitution.

The following pattern is only available for the types **Ints** n :

- An **integer literal pattern**, e.g. `214`, binds no variables. It matches the same integer, with the empty substitution.

All patterns must be **linear**: the same pattern variable may occur only once in them. This is what makes it straightforward to speak about unions of binding sets and substitutions.

Pattern matching is performed in the order in which the branches appear in the source code: the branch of the first matching pattern is followed. When matching is done on parameter types, the type checker rejects sets of patterns that are not exhaustive, and warns for completely overshadowed patterns. It always checks the type correctness of patterns with respect to the argument type. In abstract syntax, only type correctness is checked, no exhaustiveness or overshadowing.

It follows from the definition of record pattern matching that it can utilize partial records: the branch

```
{g = Fem} => t
```

in a table of type `{g : Gender ; n : Number} => T` means the same as

```
{g = Fem ; n = _} => t
```

Variables in regular expression patterns are always bound to the **first match**, which is the first in the sequence of binding lists. For example:

- `x + "e" + y` matches "peter" with `x = "p"`, `y = "ter"`
- `x + "er"*` matches "burgerer" with `x = "burg"`

Pattern macros are defined as **opers** of type **pattern** *T* for any type *T* that can have patterns. The operator `#` makes a pattern of type *T* into an expression of type **pattern** *T*. Thus, for instance, a macro for vowels can be defined

```
oper vowel : pattern Str = #("a" | "e" | "i" | "o")
```

When a pattern macro is used in actual pattern matching, the operator `#` switches it back from an expression to a pattern:

```
pre {
  #vowel => "an" ;
  _ => "a"
}
```

The use of `#` as a switch eliminates the need to complicate the name resolution rules for patterns. Following the usual conventions of functional programming languages, an identifier that is not a constructor is a variable when used as a pattern, even if it has been defined as an **oper** constant.

C.4.14 Free variation

An expression of the form

$$t_1 \mid \dots \mid t_n$$

or equivalently (for backward compatibility)

```
variants {t1 ; ... ; tn}
```

where all t_i are of the same type *T*, has itself type *T*. This expression presents $t_i, \dots, t_n$ as being in **free variation**: the choice between them is not determined by semantics or parameters. A limiting case is

```
variants {}
```

which encodes a rule saying that there is no way to express a certain thing. Notice that empty variants in a record, table, or concatenation expression has the effect that the whole expression is computed to empty variants.

Permitting **variants** in all types involves a major modification of the semantics of GF expressions. All computation rules have to be lifted to deal with lists of expressions and values. For instance,

$$t ! (t_1 \mid \dots \mid t_n) \Downarrow (t ! t_1) \mid \dots \mid (t ! t_n)$$

This is done in such a way that variation does not distribute to records (or other product-like structures). For instance, variants of records,

`{s = "Auto" ; g = Neutr} | {s = "Wagen" ; g = Masc}`

is *not* the same as a record of variants,

`{s = "Auto" | "Wagen" ; g = Neutr | Masc}`

Variants of variants are flattened,

`variants {...; variants {t1 ;...; tn} ;...} \Downarrow`
`variants {...; t1 ;...; tn ;...}`

which in the `|` notation is automatically expressed as associativity. Singleton variants are eliminated,

$$\text{variants } \{t\} \Downarrow t$$

C.4.15 Local definitions

A **local definition**, i.e. a **let expression** has the form

`let x : T = t in e`

The type of t must be T , which also becomes the type of x in e . Computation is performed by substituting t for x in e :

`let x : T = t in e \Downarrow e {x = t}`

As syntactic sugar, the type can be omitted if the type checker is able to infer it:

`let x = t in e`

It is possible to compress several local definitions into one block:

`let x : T = t ; y : U = u in e`

$\equiv$

`let x : T = t in let y : U = u in e`

Another notational variant is a definition block appearing after the main expression:

`e where {...} \equiv let {...} in e`

Curly brackets are obligatory in the **where** form, and can also be optionally used in the **let** form.

Since a block of definitions is treated as syntactic sugar for a nested **let** expression, a constant must be defined before it is used: the scope is not mutual, as in a module body. Furthermore, unlike in **lin** and **oper** definitions, it is *not* possible to bind variables on the left of the equality sign.

C.4.16 Function applications in concrete syntax

Fully compiled concrete syntax may not include expressions of function types except on the outermost level of **lin** rules, as defined in Section C.4.8. However, in the source code, and especially in **oper** definitions, functions are the

main vehicle of code reuse and abstraction. Thus function types and functions follow the same rules as in abstract syntax, as specified in Section C.4.2. In particular, the application of a lambda abstract is computed by beta conversion.

To ensure the elimination of functions, GF uses a special computation rule for pushing function applications inside tables, since otherwise expressions containing run-time variables could block their applications:

$$\begin{array}{c} (\text{table } \{p_1 \Rightarrow f_1 ; \dots ; p_n \Rightarrow f_n\} ! e) a \\ \Downarrow \\ \text{table } \{p_1 \Rightarrow f_1 a ; \dots ; p_n \Rightarrow f_n a\} ! e \end{array}$$

Also parameter constructors with non-empty contexts, as defined in Section C.3.12, result in expressions in application form. These expressions are never a problem if their arguments are just constructors, because they can then be translated to integers corresponding to the position of the expression in the enumeration of the values of its type. However, a constructor applied to a run-time variable needs to be converted as follows:

$$C \dots x \dots \Downarrow \text{case } x \text{ of } \{y \Rightarrow C \dots y \dots\}$$

for a variable y not occurring in $C \dots x \dots$. The resulting expression, when processed by table expansion as explained in Section C.4.12, results in C being applied to just values of the type of x , and the application thereby disappears.

C.4.17 Reusing top-level grammars as resources

As explained in Section C.2.6, abstract syntax modules can be opened as interfaces and concrete syntaxes as their instances. This means that judgements are translated in the following way:

$$\begin{array}{lll} \text{cat } C \ G & \Downarrow & \text{oper } C : \text{Type} \\ \text{fun } f : Cs \rightarrow C & \Downarrow & \text{oper } f : Cs \rightarrow C \\ \text{lincat } C = T & \Downarrow & \text{oper } C : \text{Type} = \text{lincat } C \ T \\ \text{lin } f = h & \Downarrow & \text{oper } f = \backslash xs \rightarrow \text{lin } C \ (h \ xs) \end{array}$$

Notice that the definiens T of `lincat` definitions is **boxed** inside the construct `lincat` $C \ T$. This means that objects of the type T cannot be used as such, but must be wrapped within the `lin` C construct:

$$\text{if } t : T \text{ then } \text{lin } C \ t : \text{lincat } C \ T$$

The type `lincat` $C \ T$ is a subtype of T , which makes the above translation of `lin` judgements type-correct.

For most purposes, a category C exported as an operation is used as an abstract datatype: the only ways of building an object of type C are the operations f obtained from `fun` and `lin` rules by the translation.

The purpose of boxed linearization types is to enforce **grammar checking via type checking**. This means that any well-typed operation application is also well-typed in the sense of the original grammar. If the direct type synonym were used, then we could for instance easily confuse all categories that have the linearization type $\{s : \text{Str}\}$.

Library writers will occasionally need to construct new values of linearization types—most notably, when writing morphological paradigms that construct linearizations of lexical types. This can be accomplished by using the operator `lin C` to wrap terms into boxed terms.

Notice. The constructs `lincat C T` and `lin C t` are implemented internally by using **lock fields**, which are record labels of form `lock_C` of the dummy type `{}` added to the record type `T`. By using the `lin C` construct, lock fields can, and should, be avoided altogether in the source code.

C.4.18 Predefined concrete syntax types

The following concrete syntax types are predefined:

- **Str**, the type of tokens and token lists (defined in Section C.4.9)
- **Integer**, the type of nonnegative integers
- **Ints** *n*, the type of integers from 0 to *n*
- **Type**, the type of (concrete syntax) types
- **PType**, the type of parameter types

The last two types are extended by user-written grammars, since new parameter types can be defined in the way shown in Section C.3.12, and every parameter type is also a type. From the point of view of the values of expressions, however, a **param** declaration does not extend **PType**, since all parameter types get compiled to initial segments of integers.

Notice the difference between the concrete syntax types **Str** and **Integer** on the one hand, and the abstract syntax categories **String** and **Int**, on the other. As *concrete syntax* types, the latter are treated in the same way as any reused categories: their objects can be formed by using syntax trees (string and integer literals), as specified in Section C.4.17.

C.4.19 Predefined concrete syntax operations

The table in Figure 80 shows the predefined operations that are declared in the library module **Predef**. Their computation is implemented as a part of the GF grammar compiler. Compilation eliminates these operations, and they may therefore not take arguments that depend on run-time variables.

The opening directive (**Predef = Predef**) is automatically included in all concrete and resource modules. Expressions of form **Predef.c** are hence available for all constants *c* listed in Figure 80, and the module need not be opened explicitly.

C.5 Flags and pragmas

C.5.1 Some flags and their values

The flag **coding** in concrete syntax sets the **character encoding** used in the grammar. Internally, GF uses unicode, and compiler-generated files are always written in UTF8 encoding. The presence of the flag **coding=utf8** prevents GF from encoding an already encoded file.

The flag **startcat** in abstract syntax sets the default start category for parsing, random generation, and any other grammar operation that depends

operation	type	explanation
PBool	PType	PTrue PFalse
Error	Type	the empty type
Integer	Type	the type of integers
Ints	Integer -> Type	integers from 0 to n
error	Str -> Error	forms error message
length	Str -> Int	length of string
drop	Integer -> Str -> Str	drop prefix of length
take	Integer -> Str -> Str	take prefix of length
tk	Integer -> Str -> Str	drop suffix of length
dp	Integer -> Str -> Str	take suffix of length
eqInt	Integer -> Integer -> PBool	test if equal integers
lessInt	Integer -> Integer -> PBool	test order of integers
plus	Integer -> Integer -> Integer	add integers
eqStr	Str -> Str -> PBool	test if equal strings
occur	Str -> Str -> PBool	test if occurs as substring
occurs	Str -> Str -> PBool	test if any char occurs
show	(P : Type) -> P -> Str	convert param to string
toStr	(L : Type) -> L -> Str	find the “first” string

FIGURE 80 Predefined types and operations.

on category. Its legal values are the categories defined or inherited in the abstract syntax.

The flag **opt** in a concrete or resource module selects the optimization strategy when producing **.gfo** files. Currently its only values are **all** and **noexpand**. The default is **all**, which means that all **oper** definitions are expanded at compile time. The **noexpand** flag prevents the expansion of definitions. This flag is useful if the **.gfo** file becomes very large otherwise, but it can slow down the use of the resource module in other modules. **optimization flag**

C.5.2 Compiler pragmas

Compiler pragmas are a special form of comments prefixed with **--#**. Currently GF interprets just one pragma, which gives the path list for searching modules. For instance, the line

```
--# -path=.:present:/home/aarne/GF/tmp
```

in the top of **FILE.gf** causes the GF compiler, when invoked on **FILE.gf**, to search through the current directory (**.**) and the directories **present** and **/home/aarne/GF/tmp**, in this order. If a directory **DIR** is not found relative to the working directory, also **\$(GF_LIB_PATH)/DIR** is searched. The default path list is **.:prelude**, and the directory **prelude** is automatically included in all path lists.

C.6 The grammar of GF

C.6.1 The lexical structure of GF

Identifiers

Identifiers $\langle \text{Ident} \rangle$ are unquoted strings beginning with a letter or an underscore `_`, followed by any combination of letters, digits, and the characters `_ ' ,`, reserved words excluded.

Literals

Integer literals $\langle \text{Integer} \rangle$ are nonempty sequences of digits.

String literals $\langle \text{String} \rangle$ have the form `"x"`, where *x* is any sequence of any characters except `"` unless preceded by `\`.

Double-precision float literals $\langle \text{Double} \rangle$ have the structure indicated by the regular expression $\langle \text{digit} \rangle + \cdot \langle \text{digit} \rangle + (\text{'e'-'?'}\langle \text{digit} \rangle +)^?$ i.e. two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in GF, which may not be used as identifiers, are the following:

PType	Str	Strs
Type	abstract	case
cat	concrete	data
def	flags	fun
in	incomplete	instance
interface	let	lin
lincat	lindef	of
open	oper	param
pre	printname	resource
strs	table	transfer
variants	where	with

The reserved symbols used in GF are the following:

```

;   =   :   ->  {   }
**  ,   (   )   [   ]
-   .   |   ?   <   >
!   *   +   ++  \   =>
_   @   $   /   #

```

Comments

Single-line comments begin with `--`. They reach till the end of the same line.

Multiple-line comments are enclosed with `{--` and `--}`. These comments can spread over many lines, or within a single line.

C.6.2 The syntactic structure of GF

Non-terminals are enclosed between $\langle$ and $\rangle$. The symbols $::=$ (production), $|$ (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

$\langle \text{Grammar} \rangle$	$::=$	$\langle \text{ListModDef} \rangle$
$\langle \text{ListModDef} \rangle$	$::=$	ϵ
	$ $	$\langle \text{ModDef} \rangle \langle \text{ListModDef} \rangle$
$\langle \text{ModDef} \rangle$	$::=$	$\langle \text{ModDef} \rangle ;$
	$ $	$\langle \text{ComplMod} \rangle \langle \text{ModType} \rangle = \langle \text{ModBody} \rangle$
$\langle \text{ModType} \rangle$	$::=$	abstract $\langle \text{Ident} \rangle$
	$ $	resource $\langle \text{Ident} \rangle$
	$ $	interface $\langle \text{Ident} \rangle$
	$ $	concrete $\langle \text{Ident} \rangle$ of $\langle \text{Ident} \rangle$
	$ $	instance $\langle \text{Ident} \rangle$ of $\langle \text{Ident} \rangle$
$\langle \text{ModBody} \rangle$	$::=$	$\langle \text{Extend} \rangle \langle \text{Opens} \rangle \{ \langle \text{ListTopDef} \rangle \}$
	$ $	$\langle \text{ListIncluded} \rangle$
	$ $	$\langle \text{Included} \rangle$ with $\langle \text{ListOpen} \rangle$
	$ $	$\langle \text{Included} \rangle$ with $\langle \text{ListOpen} \rangle$ ** $\langle \text{Opens} \rangle \{ \langle \text{ListTopDef} \rangle \}$
	$ $	$\langle \text{ListIncluded} \rangle$ ** $\langle \text{Included} \rangle$ with $\langle \text{ListOpen} \rangle$
	$ $	$\langle \text{ListIncluded} \rangle$ ** $\langle \text{Included} \rangle$ with $\langle \text{ListOpen} \rangle$
	$ $	** $\langle \text{Opens} \rangle \{ \langle \text{ListTopDef} \rangle \}$
$\langle \text{ListTopDef} \rangle$	$::=$	ϵ
	$ $	$\langle \text{TopDef} \rangle \langle \text{ListTopDef} \rangle$
$\langle \text{Extend} \rangle$	$::=$	ϵ
	$ $	$\langle \text{ListIncluded} \rangle$ **
$\langle \text{ListOpen} \rangle$	$::=$	ϵ
	$ $	$\langle \text{Open} \rangle$
	$ $	$\langle \text{Open} \rangle , \langle \text{ListOpen} \rangle$
$\langle \text{Opens} \rangle$	$::=$	ϵ
	$ $	open $\langle \text{ListOpen} \rangle$ in
$\langle \text{Open} \rangle$	$::=$	$\langle \text{Ident} \rangle$
	$ $	$(\langle \text{Ident} \rangle)$
	$ $	$(\langle \text{Ident} \rangle = \langle \text{Ident} \rangle)$
$\langle \text{ComplMod} \rangle$	$::=$	ϵ
	$ $	incomplete
$\langle \text{ListIncluded} \rangle$	$::=$	ϵ
	$ $	$\langle \text{Included} \rangle$
	$ $	$\langle \text{Included} \rangle , \langle \text{ListIncluded} \rangle$
$\langle \text{Included} \rangle$	$::=$	$\langle \text{Ident} \rangle$
	$ $	$\langle \text{Ident} \rangle [\langle \text{ListIdent} \rangle]$
	$ $	$\langle \text{Ident} \rangle - [\langle \text{ListIdent} \rangle]$
$\langle \text{Def} \rangle$	$::=$	$\langle \text{ListName} \rangle : \langle \text{Exp} \rangle$
	$ $	$\langle \text{ListName} \rangle = \langle \text{Exp} \rangle$
	$ $	$\langle \text{Name} \rangle \langle \text{ListPatt} \rangle = \langle \text{Exp} \rangle$
	$ $	$\langle \text{ListName} \rangle : \langle \text{Exp} \rangle = \langle \text{Exp} \rangle$

$\langle \text{TopDef} \rangle$	::=	$\text{cat } \langle \text{ListCatDef} \rangle$ $\text{fun } \langle \text{ListFunDef} \rangle$ $\text{data } \langle \text{ListFunDef} \rangle$ $\text{def } \langle \text{ListDef} \rangle$ $\text{data } \langle \text{ListDataDef} \rangle$ $\text{param } \langle \text{ListParDef} \rangle$ $\text{oper } \langle \text{ListDef} \rangle$ $\text{lincat } \langle \text{ListPrintDef} \rangle$ $\text{lindef } \langle \text{ListDef} \rangle$ $\text{lin } \langle \text{ListDef} \rangle$ $\text{printname cat } \langle \text{ListPrintDef} \rangle$ $\text{printname fun } \langle \text{ListPrintDef} \rangle$ $\text{flags } \langle \text{ListFlagDef} \rangle$
$\langle \text{CatDef} \rangle$	::=	$\langle \text{Ident} \rangle \langle \text{ListDDecl} \rangle$ $[\langle \text{Ident} \rangle \langle \text{ListDDecl} \rangle]$ $[\langle \text{Ident} \rangle \langle \text{ListDDecl} \rangle] \{ \langle \text{Integer} \rangle \}$
$\langle \text{FunDef} \rangle$	::=	$\langle \text{ListIdent} \rangle : \langle \text{Exp} \rangle$
$\langle \text{DataDef} \rangle$	::=	$\langle \text{Ident} \rangle = \langle \text{ListDataConstr} \rangle$
$\langle \text{DataConstr} \rangle$	::=	$\langle \text{Ident} \rangle$ $\langle \text{Ident} \rangle . \langle \text{Ident} \rangle$
$\langle \text{ListDataConstr} \rangle$	::=	ϵ $\langle \text{DataConstr} \rangle$ $\langle \text{DataConstr} \rangle \mid \langle \text{ListDataConstr} \rangle$
$\langle \text{ParDef} \rangle$	::=	$\langle \text{Ident} \rangle = \langle \text{ListParConstr} \rangle$ $\langle \text{Ident} \rangle = (\text{in } \langle \text{Ident} \rangle)$ $\langle \text{Ident} \rangle$
$\langle \text{ParConstr} \rangle$	::=	$\langle \text{Ident} \rangle \langle \text{ListDDecl} \rangle$
$\langle \text{PrintDef} \rangle$	::=	$\langle \text{ListName} \rangle = \langle \text{Exp} \rangle$
$\langle \text{FlagDef} \rangle$	::=	$\langle \text{Ident} \rangle = \langle \text{Ident} \rangle$
$\langle \text{ListDef} \rangle$	::=	$\langle \text{Def} \rangle ;$ $\langle \text{Def} \rangle ; \langle \text{ListDef} \rangle$
$\langle \text{ListCatDef} \rangle$	::=	$\langle \text{CatDef} \rangle ;$ $\langle \text{CatDef} \rangle ; \langle \text{ListCatDef} \rangle$
$\langle \text{ListFunDef} \rangle$	::=	$\langle \text{FunDef} \rangle ;$ $\langle \text{FunDef} \rangle ; \langle \text{ListFunDef} \rangle$
$\langle \text{ListDataDef} \rangle$	::=	$\langle \text{DataDef} \rangle ;$ $\langle \text{DataDef} \rangle ; \langle \text{ListDataDef} \rangle$
$\langle \text{ListParDef} \rangle$	::=	$\langle \text{ParDef} \rangle ;$ $\langle \text{ParDef} \rangle ; \langle \text{ListParDef} \rangle$
$\langle \text{ListPrintDef} \rangle$	::=	$\langle \text{PrintDef} \rangle ;$ $\langle \text{PrintDef} \rangle ; \langle \text{ListPrintDef} \rangle$
$\langle \text{ListFlagDef} \rangle$	::=	$\langle \text{FlagDef} \rangle ;$ $\langle \text{FlagDef} \rangle ; \langle \text{ListFlagDef} \rangle$
$\langle \text{ListParConstr} \rangle$	::=	ϵ $\langle \text{ParConstr} \rangle$ $\langle \text{ParConstr} \rangle \mid \langle \text{ListParConstr} \rangle$
$\langle \text{ListIdent} \rangle$	::=	$\langle \text{Ident} \rangle$ $\langle \text{Ident} \rangle , \langle \text{ListIdent} \rangle$
$\langle \text{Name} \rangle$	::=	$\langle \text{Ident} \rangle$ $[\langle \text{Ident} \rangle]$

$\langle \text{ListName} \rangle$	$::=$	$\langle \text{Name} \rangle$
		$\mid \langle \text{Name} \rangle , \langle \text{ListName} \rangle$
$\langle \text{LocDef} \rangle$	$::=$	$\langle \text{ListIdent} \rangle : \langle \text{Exp} \rangle$
		$\mid \langle \text{ListIdent} \rangle = \langle \text{Exp} \rangle$
		$\mid \langle \text{ListIdent} \rangle : \langle \text{Exp} \rangle = \langle \text{Exp} \rangle$
$\langle \text{ListLocDef} \rangle$	$::=$	ϵ
		$\mid \langle \text{LocDef} \rangle$
		$\mid \langle \text{LocDef} \rangle ; \langle \text{ListLocDef} \rangle$
$\langle \text{Exp6} \rangle$	$::=$	$\langle \text{Ident} \rangle$
		$\mid \langle \text{Sort} \rangle$
		$\mid \langle \text{String} \rangle$
		$\mid \langle \text{Integer} \rangle$
		$\mid \langle \text{Double} \rangle$
		$\mid ? \langle \text{Integer} \rangle$
		$\mid ?$
		$\mid []$
		data
		$[\langle \text{Ident} \rangle \langle \text{Exps} \rangle]$
		$[\langle \text{String} \rangle]$
		$\{ \langle \text{ListLocDef} \rangle \}$
		$< \langle \text{ListTupleComp} \rangle >$
		$< \langle \text{Exp} \rangle : \langle \text{Exp} \rangle >$
		$(\langle \text{Exp} \rangle)$
$\langle \text{Exp5} \rangle$	$::=$	$\langle \text{Exp5} \rangle . \langle \text{Label} \rangle$
		$\mid \langle \text{Exp6} \rangle$
$\langle \text{Exp4} \rangle$	$::=$	$\langle \text{Exp4} \rangle \langle \text{Exp5} \rangle$
		table $\{ \langle \text{ListCase} \rangle \}$
		table $\langle \text{Exp6} \rangle \{ \langle \text{ListCase} \rangle \}$
		table $\langle \text{Exp6} \rangle [\langle \text{ListExp} \rangle]$
		case $\langle \text{Exp} \rangle$ of $\{ \langle \text{ListCase} \rangle \}$
		variants $\{ \langle \text{ListExp} \rangle \}$
		pre $\{ \langle \text{ListCase} \rangle \}$
		lincat $\langle \text{Ident} \rangle$
		lin $\langle \text{Ident} \rangle \langle \text{Exp6} \rangle$
		$\langle \text{Exp5} \rangle$
$\langle \text{Exp3} \rangle$	$::=$	$\langle \text{Exp3} \rangle ! \langle \text{Exp4} \rangle$
		$\mid \langle \text{Exp3} \rangle * \langle \text{Exp4} \rangle$
		$\mid \langle \text{Exp3} \rangle ** \langle \text{Exp4} \rangle$
		$\mid \# \langle \text{Patt2} \rangle$
		$\mid \langle \text{Exp4} \rangle$
$\langle \text{Exp2} \rangle$	$::=$	$\langle \text{Exp3} \rangle + \langle \text{Exp2} \rangle$
		$\mid \langle \text{Exp3} \rangle$
$\langle \text{Exp1} \rangle$	$::=$	$\langle \text{Exp2} \rangle ++ \langle \text{Exp1} \rangle$
		$\mid \langle \text{Exp2} \rangle$
$\langle \text{Exp} \rangle$	$::=$	$\backslash \langle \text{ListBind} \rangle -> \langle \text{Exp} \rangle$
		$\backslash \backslash \langle \text{ListBind} \rangle => \langle \text{Exp} \rangle$
		$\langle \text{Decl} \rangle -> \langle \text{Exp} \rangle$
		$\langle \text{Exp3} \rangle => \langle \text{Exp} \rangle$
		let $\{ \langle \text{ListLocDef} \rangle \}$ in $\langle \text{Exp} \rangle$
		let $\langle \text{ListLocDef} \rangle$ in $\langle \text{Exp} \rangle$
		$\langle \text{Exp3} \rangle$ where $\{ \langle \text{ListLocDef} \rangle \}$
		$\langle \text{Exp0} \rangle \mid \langle \text{Exp1} \rangle$
		$\langle \text{Exp1} \rangle$

$\langle \text{ListExp} \rangle$	$::=$	ϵ
		$\mid$
		$\langle \text{Exp} \rangle$
		$\mid$
		$\langle \text{Exp} \rangle ; \langle \text{ListExp} \rangle$
$\langle \text{Exps} \rangle$	$::=$	ϵ
		$\mid$
		$\langle \text{Exp6} \rangle \langle \text{Exps} \rangle$
$\langle \text{Patt2} \rangle$	$::=$	$-$
		$\mid$
		$\langle \text{Ident} \rangle$
		$\mid$
		$\langle \text{Ident} \rangle . \langle \text{Ident} \rangle$
		$\mid$
		$\langle \text{Integer} \rangle$
		$\mid$
		$\langle \text{Double} \rangle$
		$\mid$
		$\langle \text{String} \rangle$
		$\mid$
		$\{ \langle \text{ListPattAss} \rangle \}$
		$\mid$
		$< \langle \text{ListPattTupleComp} \rangle >$
		$\mid$
		$\# \langle \text{Ident} \rangle$
		$\mid$
		$?$
		$\mid$
		$(\langle \text{Patt} \rangle)$
$\langle \text{Patt1} \rangle$	$::=$	$\langle \text{Ident} \rangle \langle \text{ListPatt} \rangle$
		$\mid$
		$\langle \text{Ident} \rangle . \langle \text{Ident} \rangle \langle \text{ListPatt} \rangle$
		$\mid$
		$\langle \text{Patt2} \rangle *$
		$\mid$
		$\langle \text{Ident} \rangle @ \langle \text{Patt2} \rangle$
		$\mid$
		$- \langle \text{Patt2} \rangle$
		$\mid$
		$\langle \text{Patt2} \rangle$
$\langle \text{Patt} \rangle$	$::=$	$\langle \text{Patt} \rangle \mid \langle \text{Patt1} \rangle$
		$\mid$
		$\langle \text{Patt} \rangle + \langle \text{Patt1} \rangle$
		$\mid$
		$\langle \text{Patt1} \rangle$
$\langle \text{PattAss} \rangle$	$::=$	$\langle \text{ListIdent} \rangle = \langle \text{Patt} \rangle$
$\langle \text{Label} \rangle$	$::=$	$\langle \text{Ident} \rangle$
		$\mid$
		$\$ \langle \text{Integer} \rangle$
$\langle \text{Sort} \rangle$	$::=$	Type
		$\mid$
		PType
		$\mid$
		Str
		$\mid$
		Strs
$\langle \text{ListPattAss} \rangle$	$::=$	ϵ
		$\mid$
		$\langle \text{PattAss} \rangle$
		$\mid$
		$\langle \text{PattAss} \rangle ; \langle \text{ListPattAss} \rangle$
$\langle \text{ListPatt} \rangle$	$::=$	$\langle \text{Patt2} \rangle$
		$\mid$
		$\langle \text{Patt2} \rangle \langle \text{ListPatt} \rangle$
$\langle \text{Bind} \rangle$	$::=$	$\langle \text{Ident} \rangle$
		$\mid$
		$-$
$\langle \text{ListBind} \rangle$	$::=$	ϵ
		$\mid$
		$\langle \text{Bind} \rangle$
		$\mid$
		$\langle \text{Bind} \rangle , \langle \text{ListBind} \rangle$
$\langle \text{Decl} \rangle$	$::=$	$(\langle \text{ListBind} \rangle : \langle \text{Exp} \rangle)$
		$\mid$
		$\langle \text{Exp4} \rangle$
$\langle \text{TupleComp} \rangle$	$::=$	$\langle \text{Exp} \rangle$
$\langle \text{PattTupleComp} \rangle$	$::=$	$\langle \text{Patt} \rangle$
$\langle \text{ListTupleComp} \rangle$	$::=$	ϵ
		$\mid$
		$\langle \text{TupleComp} \rangle$
		$\mid$
		$\langle \text{TupleComp} \rangle , \langle \text{ListTupleComp} \rangle$

$\langle ListPattTupleComp \rangle$	$::=$	ϵ
	$ $	$\langle PattTupleComp \rangle$
	$ $	$\langle PattTupleComp \rangle , \langle ListPattTupleComp \rangle$
$\langle Case \rangle$	$::=$	$\langle Patt \rangle => \langle Exp \rangle$
$\langle ListCase \rangle$	$::=$	$\langle Case \rangle$
	$ $	$\langle Case \rangle ; \langle ListCase \rangle$
$\langle DDecl \rangle$	$::=$	$(\langle ListBind \rangle : \langle Exp \rangle)$
	$ $	$\langle Exp6 \rangle$
$\langle ListDDecl \rangle$	$::=$	ϵ
	$ $	$\langle DDecl \rangle \langle ListDDecl \rangle$

Appendix D

The GF Resource Grammar Library

This Appendix gives a complete listing of the categories and functions available in the language-independent part of the GF Resource Grammar Library, as released with GF Version 3.2 in 2010. The listing covers the **minimal set of functions** and excludes the **shortcuts**, most of which are derived from the minimal set by the expansion rules of Section D.2. Thus it is aimed to serve two kinds of readers:

- resource implementors and linguists, by giving a concise overview of the scope and structure of the library
- resource library users, by displaying the complete set of available operations, as a product of the minimal functions and the expansion rules

This Appendix will mostly not show the names of the functions, but their API names can be easily constructed by following the conventions of Section D.2. Their internal names have to be looked up from the library sources. Both the API and the internal modules, with documentation, can be found on the GF web page.

In addition to the core resource syntax in Section D.1 and Section D.2, we will show the lexical paradigms that are available in all languages in Section D.3 and some other library modules in Section D.4.

D.1 The category system

The chart in Figure 81 shows the categories in a hierarchical top-down order. The edges do not define the complete dependency structure; if they did, the graph would have many more edges, and also cycles. The precise meaning of a directed edge from *C* to *D* is: there is a constructor of *C* that takes *D* as an argument. What the constructors exactly are, and what other arguments they take, is described by separate tables for each category.

The rectangular boxes mark open lexical categories, which have constructors also in the **Paradigms** modules. The abbreviations **A***, **N***, and **V*** stand for groups of adjective, noun, and verb categories, respectively, differing in their expected complements. These groups are listed and explained in full in Section D.1.2.

D.1.1 Phrasal and closed lexical categories

Category	Explanation	Example (default)
Ant	anteriority	simultaneous , anterior
CAdv	comparative adverb	<i>more</i>
CN	common noun (without determiner)	<i>red house</i>
Card	cardinal number	<i>seven</i>
Cl	declarative clause, with all tenses	<i>she looks at this</i>
ClSlash	clause missing noun phrase	<i>she looks at</i>
Comp	complement of copula, such as AP	<i>very warm</i>
Conj	conjunction	<i>and</i>
Det	determiner phrase	<i>those seven</i>
Digits	cardinal or ordinal in digits	<i>1,000/1,000th</i>
IAdv	interrogative adverb	<i>why</i>
IComp	interrogative complement of copula	<i>where</i>
IDet	interrogative determiner	<i>how many</i>
IP	interrogative pronoun	<i>who</i>
Imp	imperative	<i>look at this</i>
NP	noun phrase (subject or object)	<i>the red house</i>
Num	number determining element	<i>seven</i> , singular , plural
Numeral	cardinal or ordinal in words	<i>five/fifth</i>
Ord	ordinal number (used in Det)	<i>seventh</i>
PConj	phrase-beginning conjunction	<i>therefore</i> , no pconj
Phr	phrase in a text	<i>but be quiet please</i>
Pol	polarity	positive , negative
Predet	predeterminer (prefixed Quant)	<i>all</i>
Prep	preposition, or just case	<i>in</i>
Pron	personal pronoun	<i>she</i>
QCl	question clause, with all tenses	<i>why does she walk</i>
QS	question	<i>where did she live</i>
Quant	quantifier ('nucleus' of Det)	<i>this/these</i>
RCl	relative clause, with all tenses	<i>in which she lives</i>
RP	relative pronoun	<i>in which</i>
RS	relative sentence	<i>in which she lived</i>
S	declarative sentence	<i>she looked at him</i>
SC	embedded sentence or question	<i>that it rains</i>
SSlash	sentence missing noun phrase	<i>she looked at</i>
Subj	subjunction	<i>if</i>
Temp	temporal and aspectual features	past anterior
Tense	tense	present , past, future
Text	text consisting of several phrases	<i>He is here. Why?</i>
Utt	utterance: sentence, question...	<i>be quiet</i>
VP	verb phrase	<i>is very warm</i>
VPSlash	verb phrase missing noun phrase	<i>look at</i>
Voc	vocative	<i>my darling</i> , no voc
[C]	list of category $C = \text{Adv, AP, NP, RS, S}$	X, Y, Z

D.1.2 Open lexical categories

Category	Explanation	Example
AdA	adjective-modifying adverb	<i>very</i>
AdN	numeral-modifying adverb	<i>more than</i>
Adv	adverb directly attached to verb	<i>always</i>
Adv	verb-phrase-modifying adverb	<i>in the house</i>
A	one-place adjective	<i>warm</i>
A2	two-place adjective	<i>divisible</i>
AP	adjectival phrase	<i>very warm</i>
N	common noun	<i>house</i>
N2	relational noun	<i>son</i>
N3	three-place relational noun	<i>connection</i>
PN	proper name	<i>Paris</i>
V	one-place verb	<i>sleep</i>
V2	verb with an NP complement	<i>love</i>
VA	verb with an AP complement	<i>look</i>
VQ	verb with a QS complement	<i>wonder</i>
VS	verb with an S complement	<i>claim</i>
VV	verb with a VP complement	<i>want</i>
V2A	verb with NP and AP complement	<i>paint</i>
V2Q	verb with NP and QS complement	<i>ask</i>
V2S	verb with NP and S complement	<i>tell</i>
V2V	verb with NP and VP complement	<i>cause</i>
V3	verb with two NP complements	<i>show</i>

D.2 Syntax rules

The syntax rules are given in a tabular format, where each line defines an abstract syntax constructor. Thus, for instance,

C1	NP VP	<i>John walks</i>

means that there is a constructor

`mkC1 : NP -> VP -> C1`

Identifiers in angle brackets, such as `<fullstopPunct>`, are not categories but constructors. Thus

Pol	<code><positivePol></code>	(as in <i>he walks</i>)

means that there is a constructor

`positivePol : Pol`

After each rule, an English example is given. Most examples can be automatically translated to all library languages, since they use the vocabulary included in the common test lexicon. Examples are in *italics*; roman font in this column, usually in parentheses, is an explanation of a construct that doesn't have straightforward examples.

The tables give only the minimal syntax rules, i.e. the ones that correspond to **fun** functions in the core resource grammar. This set of rules expands to the full library API by the following **expansion rules**:

- **Defaults.** Arguments given in parentheses can be omitted, in which case defaults are used. These arguments belong to the types **Ant**, **Pol**, **PConj**, **Tense** and **Voc**. Their defaults are given in Section D.1.1.
- **Flattening.** Arguments of the following types can be expanded to all non-recursive lists of arguments that have a constructor resulting in the same type:

- ClSlash, Comp, SC, Tense, VP, VPSlash
- **Lexical flattening.** Arguments can be replaced by their lexical heads. This covers the following cases:
 - AP by A
 - CN by N
 - Num by Numeral, Card, and Digits

D.2.1 Suprasentential level: texts and utterances

value	arguments	example
Text	-	(empty text)
Punct	Phr Punct Text	<i>Who walks? John.</i>
	<fullstopPunct>	.
	<questionPunct>	?
Phr	<exclamationPunct>	!
	(PConj) Utt (Voc)	<i>but walk, my friend</i>
	Conj	<i>and</i>
PConj	<noPConj>	(empty)
Voc	NP	<i>my friend</i>
Utt	<noVoc>	(empty)
	S	<i>John walks</i>
	QS	<i>who walks</i>
	Imp	<i>don't walk</i>
	NP	<i>this man</i>
	Adv	<i>here</i>
	VP	<i>to sleep</i>
	IP	<i>who</i>
	IAAdv	<i>why</i>

D.2.2 Sentential level: polarity, tense, and mood

value	arguments	example
S	(Tense) (Ant) (Pol) Cl	<i>John wouldn't have walked</i>
QS	(Tense) (Ant) (Pol) QC1	<i>wouldn't John have walked</i>
RS	(Tense) (Ant) (Pol) RC1	<i>that wouldn't have walked</i>
Pol	<positivePol>	(as in <i>he walks</i>)
	<negativePol>	(as in <i>he doesn't walk</i>)
Temp	Tense Ant	(tense and aspect together)
Tense	<presentTense>	(as in <i>he walks</i>)
	<pastTense>	(as in <i>he walked</i>)
	<futureTense>	(as in <i>he will walk</i>)
	<conditionalTense>	(as in <i>he would walk</i>)
Ant	<simultaneousAnt>	(as in <i>he walks</i>)
	<anteriorAnt>	(as in <i>he has walked</i>)
Imp	ImpForm Pol VP	<i>don't forget yourselves</i>
ImpForm	<famImpForm>	(as in <i>help yourself</i>)
	<polImpForm>	(as in <i>help yourself, sir</i>)
	<pluralImpForm>	(as in <i>help yourselves</i>)

D.2.3 Predication, complementation, and extraction

value	arguments	example
C1	NP VP	<i>John walks</i>
	SC VP	<i>that he walks is good</i>
	NP	<i>there are five flowers</i>
	VP	<i>it is raining</i>
	NP RS	<i>it is John who walks</i>
	Adv S	<i>it is here John walks</i>
VP	V	<i>sleep</i>
	VV VP	<i>want to run</i>
	VS S	<i>know that she runs</i>
	VQ QS	<i>wonder if she runs</i>
	VA AP	<i>look red</i>
	VPSlash NP	<i>use it</i>
	VPSlash	<i>love himself</i>
	<passiveVP> VPSlash	<i>be loved</i>
	Comp	<i>be small</i>
	VP Adv	<i>sleep here</i>
Comp	VP Adv	<i>always sleep</i>
	AP	<i>(be) small</i>
	NP	<i>(be) a man</i>
	Adv	<i>(be) here</i>
SC	S	<i>that she goes</i>
	QS	<i>who goes</i>
	VP	<i>to go</i>
SSlash	Tense Pol C1Slash	<i>he used (it)</i>
C1Slash	NP VPSlash	<i>he uses (it)</i>
	C1 Prep	<i>he walks with (her)</i>
	C1Slash Adv	<i>he uses (it) today</i>
	NP VS SSlash	<i>she says that he uses (it)</i>
	VPSlash	<i>use (it)</i>
	V3 NP	<i>send it (to her)</i>
	V3 NP	<i>send (it) to her</i>
	V2V VP	<i>force (her) to run</i>
	V2S S	<i>tell (us) that she runs</i>
	V2Q QS	<i>ask (us) if she runs</i>
VPSlash	V2A AP	<i>paint (it) red</i>
	VV VPSlash	<i>want to use (it)</i>
	V2V NP VPSlash	<i>force me to use (it)</i>

D.2.4 Question and relative clause formation

value	arguments	example
QC1	C1	<i>does John walk</i>
	IP VP	<i>who walks</i>
	IP C1Slash	<i>whom does John love</i>
	IAdv C1	<i>why does John walk</i>
	IComp NP	<i>where is John</i>
	IP	<i>which houses are there</i>
RC1	RP VP	<i>who walks</i>
	RP C1Slash	<i>whom John loves</i>

D.2.5 Interrogative and relative pronouns

value	arguments	example
IP	IDet CN	<i>which five songs</i>
	IDet	<i>which five</i>
	IP Adv	<i>who in Paris</i>
	IDet	<i>which five</i>
	IAdv	<i>with whom</i>
IComp	Prep IP	<i>where (is it)</i>
	IAdv	<i>who (is it)</i>
	IP	<i>who (is it)</i>
RP	<which_RP>	<i>which</i>
	Prep NP RP	<i>only one of which</i>

D.2.6 Noun phrases and determiners

value	arguments	example
NP	Det CN	<i>the man</i>
	PN	<i>John</i>
	Pron	<i>he</i>
	Predet NP	<i>only the man</i>
	NP V2	<i>the man seen</i>
	NP Adv	<i>Paris today</i>
	NP RS	<i>Paris, which is here</i>
	Det	<i>these five</i>
	CN	<i>beer</i>
	Det	<i>these five best</i>
Quant	Quant Num Det	<i>these five</i>
	Pron Quant	<i>my</i>
	<the_Quant>	<i>the</i>
	<a_Quant>	<i>a</i>
Num	Card	<i>fifty-six</i>
	<sgNum>	<i>(as in this)</i>
	<plNum>	<i>(as in these)</i>
Card	Numeral	<i>fifty-six</i>
	Digits	<i>56</i>
Ord	Numeral	<i>fifty-sixth</i>
	Digits	<i>56th</i>
	A	<i>best</i>

D.2.7 The numeral system

The numeral system uses auxiliary categories `Sub1000`, `Sub100`, and `Unit`, which are not displayed in the category table in Section D.1.1.

value	arguments	example
Digits	Dig Dig Digits	5 528
Dig	<n0_Dig>...<n9_Dig>	0,1,2,3,4,5,6,7,8,9
Numeral	<thousandfoldNumeral> Sub1000 Sub1000 Sub1000 Sub1000	$m * 1000$ $m * 1000 + n$ coerce 1..999
Sub1000	Unit Unit Sub100 Sub100	$m * 100$ $m * 100 + n$ coerce 1..99
Sub100	<tenfoldSub100> Unit Unit Unit Unit	$m * 10$ $m * 10 + n$ coerce 1..9
Unit	<n1_Unit>,...,<n9_Unit>	one,...,nine

D.2.8 Common nouns, adjectives, and adverbs

value	arguments	example
CN	N	man
	CN AP	old man
	CN RS	man who walks
	N2 NP	mother (of x)
	N3 NP NP	distance (from x) (to y)
	CN SC	question where she sleeps
AP	A	warm
	A NP	warmer than Paris
	A2 NP	married to her
	A2	married to himself
	A2	married
	AP SC	good that she sleeps
	AdA AP	very good
AdN	CAdv	more (than five)
AdA	A	extremely
Adv	A	quickly
	Prep NP	in the house
	CAdv A NP	less quickly than John
	CAdv A S	less quickly than I ran
	AdA Adv	very quickly
	Subj S	when he arrives

D.2.9 Coordination

For $C = \text{Adv, AP, NP, RS, S}$:

value	arguments	example
C	Conj [C]	X, Y and Z
[C]	$C C$	Y, Z
	$C [C]$	X, Y, Z

D.2.10 Structural words

The structural words are identified by English words, for which each language uses the closest equivalent. Digits (7, 8) are used as separators in multi-word expressions and for appending explanations. The function names are formed by appending *_C* to the English word given below.

category	words
AdA	almost, quite, so, too, very
Adv	always
Adv	everywhere, here, here7from, here7to, somewhere, there there7from, there7to
CAdv	less, more
Conj	and, both7and, either7or, or
Det	every, few, many, much, somePl, someSg
IAdv	how, when, where, why
IDet	how-many
IP	whatPl, whatSg, whoPl, whoSg
IQuant	which
NP	everybody, everything, somebody, something
PConj	but, otherwise, therefore
Predet	all, most, only
Prep	above, after, before, behind, between, by8agent by8means, during, for, from, in8front, in, on part (partitive <i>of</i>), possess (possessive <i>of</i>), through to, under, with, without
Pron	i, youSg, youPl, he, she, it, we, youPl, they
Quant	that, this
Subj	although, because, if, when
Utt	no, yes
VV	can, can8know, must, want

D.3 Lexical Paradigms

The paradigms in this section are available for all resource grammar languages. In addition to them, the `Paradigms` module for each language has a set of paradigms for less regular words.

D.3.1 Paradigms for regular words

Function	Type
mkN	Str -> N
mkPN	Str -> PN
mkA	Str -> A
mkV	Str -> V
mkAdv	Str -> Adv
mkPrep	Str -> Prep

D.3.2 Paradigms for verb, adjective, and noun subcategories

Function	Type
mkV2	V -> V2
mkV2	V -> Prep -> V2
mkV3	V -> V3
mkV3	V -> Prep -> Prep -> V3
mkVS	V -> VS
mkV2S	V -> Prep -> V2S
mkVV	V -> VV
mkV2V	V -> Prep -> Prep -> V2V
mkVA	V -> VA
mkV2A	V -> Prep -> V2A
mkVQ	V -> VQ
mkV2Q	V -> Prep -> V2Q
mkAS	A -> AS
mkA2S	A -> Prep -> A2S
mkAV	A -> AV
mkA2V	A -> Prep -> A2V
mkN2	N -> Prep -> N2
mkN3	N -> Prep -> Prep -> N3

D.4 Other library modules

D.4.1 The Prelude module

The `Prelude` defines commonly used utility functions, in particular for strings and booleans.

Oper	Type	Explanation
SS	Type	the type {s : Str}
ss	Str -> SS	record from string
nonExist	Str	missing form
optStr	Str -> Str	optional string
bothWays	(x,y : Str) -> Str	either x ++ y or y ++ x
Bool	PType	values True and False
andB	(_,_ : Bool) -> Bool	conjunction
orB	(_,_ : Bool) -> Bool	disjunction
notB	Bool -> Bool	negation
if_then_else	(A:Type)->Bool->A->A->A	conditional
init	Str -> Str	drop last character
last	Str -> Str	return last character
glue	Str -> Str -> Str	glue tokens together

D.4.2 The Formal module

This module is used for defining formal languages, in particular ones that use precedence levels and parentheses for grouping subexpressions.

Oper	Type	Explanation
Prec	PType	precedence levels 0..4
TermPrec	Type	string with precedence
mkPrec	Prec -> Str -> TermPrec	construct a TermPrec
top	TermPrec -> Str	top term (lowest prec.)
constant	Str -> TermPrec	atomic (highest prec.)
infixl	Prec->Str->(.,.:TermPrec)->TermPrec	left-assoc. infix
infixr	Prec->Str->(.,.:TermPrec)->TermPrec	right-assoc. infix
infixn	Prec->Str->(.,.:TermPrec)->TermPrec	non-assoc. infix
usePrec	Prec -> TermPrec -> Str	use term on given level

D.4.3 The Symbolic module

This module is used for embedding symbolic notation in natural-language text constructed by the resource grammar API. It works for all resource languages.

Function	Type	Example
symb	Str -> NP	<i>x</i>
symb	Int -> NP	<i>23</i>
symb	Float -> NP	<i>0.99</i>
symb	CN -> Numeral -> NP	<i>level four</i>
symb	Det -> CN -> Numeral -> NP	<i>the level four</i>
symb	Det -> CN -> [Symb] -> NP	<i>the levels i, j and k</i>
symb	Symb -> S	<i>A (formula)</i>
symb	Symb -> Card	<i>n (number)</i>
mkSymb	Str -> Symb	<i>x</i>

D.4.4 The Combinators module

This module gives shortcuts for defining predicates (**pred**) and function expressions (**app**). It works for all resource languages.

Function	Type	Example
pred	V -> NP -> C1	<i>x converges</i>
pred	V2 -> NP -> NP -> C1	<i>x intersects y</i>
pred	V -> NP -> NP -> C1	<i>x and y intersect</i>
pred	A -> NP -> C1	<i>x is even</i>
pred	A2 -> NP -> NP -> C1	<i>x is divisible by y</i>
pred	A -> NP -> NP -> C1	<i>x and y are equal</i>
pred	N -> NP -> C1	<i>x is a maximum</i>
pred	N -> NP -> NP -> C1	<i>x and y are inverses</i>
pred	Adv -> NP -> C1	<i>x is in scope</i>
pred	Prep -> NP -> NP -> C1	<i>x is outside y</i>
app	N -> NP	<i>the bottom</i>
app	N2 -> NP -> NP	<i>the successor of x</i>
app	N3 -> NP -> NP -> NP	<i>the distance from x to y</i>
app	N2 -> NP -> NP -> NP	<i>the sum of x and y</i>
app	N2 -> N -> CN	<i>set of integers</i>
app	N2 -> NP -> CN	<i>divisor of x</i>
app	N3 -> NP -> NP -> CN	<i>path from x to y</i>
app	N2 -> NP -> NP -> CN	<i>path between x and y</i>

Appendix E

The GF Software System

The GF software system implements the GF programming language. Its components are

- the **compiler**, translating `.gf` source files to `.gfo` object files, to `.pgf` run-time grammars, and to various other formats
- the **run-time system**, performing parsing, generation, translation and other functions with `.pgf` grammars
- the **command interpreter**, also known as the **GF shell**, executing user commands by calling the compiler and the run-time system

The GF shell is used in this book as the main tool for learning GF and experimenting with it. This Appendix summarizes the most important commands of the GF shell, as well as the use of the compiler in batch mode.

E.1 The GF shell

The GF shell is invoked by the command `gf`, which takes arguments and options according to the following syntax:

```
gf (OPTION | FLAG)* FILE*
```

The shell maintains a **state**, to which belong

- a multilingual PGF grammar
- optionally, a set of compiled GF modules (retaining `oper` definitions)
- a history of previous commands
- a set of string, tree, and command macros

Unless file arguments are provided to the `gf` command, the shell starts in an empty state, with no grammars and no history.

In the shell, a set of commands is available. Some of these commands may change the grammars in the state. The general syntax of commands is given by the following BNF grammar:

```
COMMAND_LINE ::= COMMAND_PIPE
COMMAND_LINE ::= COMMAND_PIPE ";" COMMAND_LINE
COMMAND_PIPE ::= COMMAND
COMMAND_PIPE ::= COMMAND "|" COMMAND_PIPE
COMMAND      ::= COMMAND_ID (OPTION | FLAG)* ARGUMENT?
OPTION       ::= "-"OPTION_ID
FLAG         ::= "-"OPTION_ID "=" VALUE
ARGUMENT     ::= QUOTED_STRING | TREE
VALUE        ::= IDENT | QUOTED_STRING
```


A command pipe is a sequence of commands interpreted in such a way that the output of each command is sent as input to the next. The option `-tr` causes GF to show a trace, i.e. the intermediate result of the command to which it is attached.

A command line is a sequence of pipes separated by `;`. These pipes are executed one by one, in the order of appearance.

The following list of commands is a subset of the full set, but it includes all commands used in this book, giving some representative examples of each. A complete, updated list of commands with more explanations and options can be obtained in the GF shell by the `help` command.

!: *system command: escape to system shell.*

```
! ls *.gf -- list all GF files in the working directory
```

?: *system pipe: send value from previous command to a system command.*

```
gt | l | ? wc -- generate, linearize, word-count
```

aw = `align_words`: *show word alignments between languages graphically.*

```
gr | aw -- generate tree, show alignment graph code
gr | aw -view=open -- generate tree, display alignment on Mac
gr | aw -view=eog -- generate tree, display alignment on Ubuntu
```

cc = `compute_concrete`: *computes concrete syntax term using a source grammar.*

```
cc -table mkN "talo" -- show inflection table of noun "talo"
```

dc = `define_command`: *add a command macro to the state.*

```
dc table linearize -table -lang=Fin ?0 -- define command 'table'
%table house_N -- apply the command 'table'
```

e = `empty`: *empty the environment.*

```
e
```

gr = `generate_random`: *generate random trees in the current abstract syntax.*

```
gr -- one tree in the startcat of current grammar
gr -cat=NP -number=16 -- 16 trees in the category NP
gr -lang=Hin,Tha -- trees implemented both in Hin and Tha
gr -probs=FILE -- generate with bias
gr (AdjCN ? (UseN ?)) -- generate trees of form (AdjCN ? (UseN ?))
```

gt = `generate_trees`: *generates a list of trees, by default exhaustive.*

```
gt -- all trees in the startcat, to depth 4
gt -cat=NP -number=16 -- 16 trees in the category NP
gt -cat=NP -depth=2 -- trees in the category NP to depth 2
gt (AdjCN ? (UseN ?)) -- trees of form (AdjCN ? (UseN ?))
```

h = `help`: *get description of a command, or a the full list of commands.*

```
h gr -- help on command gr
h -full | wf -file=HELP -- write file with full help on all commands
```

i = `import`: *import a grammar from source code or compiled .pgf file.*

```
i -v LangEng.gf LangSwe.gf -- import Eng and Swe, be verbose
```

l = `linearize`: *convert an abstract syntax expression to string.*

```
l -lang=Swe,Nor no_Utt -- tree in Swe and Nor
gr -cat=C1 | l -lang=Hin -table -to_devanagari -- random hindi table
```

`ma = morpho_analyse`: *print the morphological analyses of all words in the string.*

```
rf -file=bible.txt | ma -- analyse the words in the Bible
```

`mq = morpho_quiz`: *start a morphology quiz.*

```
mq -lang=Swe (AdjCN (PositA ?) (UseN ?)) -- train N+A forms in Swe
```

`p = parse`: *parse a string to abstract syntax expression.*

```
p -cat=NP -lang=Fin "tuo talo" -- parse NP in Finnish
```

`pg = print_grammar`: *print the actual grammar with the given printer.*

```
pg -funs | ? grep " S ;" -- show functions with value cat S
```

`ps = put_string`: *return a string, possibly processed with a function.*

```
l (EAdd 3 4) | ps -lexcode      -- linearize code-like output
ps -lexcode | p -cat=Exp       -- parse code-like input
ps -to_devanagari "A-p"       -- show Devanagari in real font
rf -file=Hin.gf | ps -env=quotes -to_devanagari
                        -- convert transliterated gf source file to devanagari
rf -file=Ara.gf | ps -env=quotes -from_arabic
                        -- transliterate strings in gf source file from Arabic
```

`pt = put_tree`: *return a tree, possibly processed with a function.*

```
pt -compute (plus one two) -- compute value
p "foo" | pt -paraphrase   -- paraphrase parse results
```

`q = quit`: *exit the GF shell.*

```
q
```

`rf = read_file`: *read string or tree input from a file.*

```
rf -file=tb -lines -trees | l -- linearize trees given in file
```

`rt = rank_trees`: *show trees in the order of decreasing probability.*

```
p "foo" | rt -probs=FILE | pt -number=1 -- most probable parse
```

`tq = translation_quiz`: *start a translation quiz.*

```
tq -from=Eng -to=Swe -- startcat trees
tq -from=Eng -to=Swe (AdjCN (PositA ?2) (UseN ?)) -- only this form
```

`ut = unicode_table`: *show a transliteration table for a unicode character set.*

```
ut -devanagari
```

`vd = visualize_dependency`: *show dependency parse tree graphically.*

```
gr | vd -- generate random dependency tree in .dot
gr | vd -view=open -- generate and display dependency tree on Mac
gr -number=1000 | vd -file=deps -output=malt -- training corpus
gr -number=100 | vd -file=deps -output=malt_input -- test corpus
```

`vp = visualize_parse`: *show a parse tree graphically.*

```
p "hello" | vp -view=open -- parse a string and display tree on Mac
```

`vt = visualize_tree`: *show a set of trees graphically.*

```
p "hello" | vt -view=eog -- parse a string and display tree on Ubuntu
```

`wf = write_file`: *send string or tree to a file.*

```

wf -file=tb "My Treebank"           -- write one line to file tb
gr | l -treebank | wf -append -file=tb -- append random tree+lin.

```

E.2 The GF batch compiler

With the option `-batch`, GF can be invoked in batch mode, i.e. without opening the shell, to compile files from `.gf` to `.gfo`. The `-s` option (“silent”) eliminates all messages except errors.

```
$ gf -batch -s LangIta.gf
```

With the option `-make`, and as a set of top-level grammar files (with the same abstract syntax) as arguments, GF produces a `.pgf` file. The flag `-optimize-pgf` minimizes the size of the `.pgf` file, and is recommended for grammars that are shipped as end user applications.

```
$ gf -make -optimize-pgf LangIta.gf LangEng.gf LangGer.gf
```

The flag `-output-format` adds another output format to be generated in addition to `.pgf`. For instance

```
$ gf -make -output-format=js LangEng.pgf LangGer.pgf
```

Notice that the arguments can be `.pgf` files, which in this case are merged and written into a JavaScript grammar file.

More options and instructions are obtained with

```
$ gf -help
```

To run GF from a **script**, redirection of standard input can be used:

```
$ gf <script.gfs
```

The file `script.gfs` should then contain a sequence of GF commands, one per line. Unrecognized command lines are skipped without terminating GF.

Appendix F

Bibliography

Following the normal usage of text books, as opposed to research monographs, this book gives very few explicit references. However, the work has not been done in a vacuum. This Appendix gives a list of relevant literature, divided into two sections

- publications on GF: a fairly complete list, with brief summaries of each publication
- background and related work: a list of literature that has inspired GF or is otherwise related, with brief comments on how the publication is related to GF

The book's web page <http://www.grammaticalframework.org/gf-book> gives links to those publications that are available electronically.

F.1 Publications on GF

In reverse temporal order:

- J. Camilleri, Gordon J. Pace, and Mike Rosner. Playing Nomic using a Controlled Natural Language. *CNL 2010, Controlled Natural Language*, Marettimo, 2010. *Using GF for defining the rules of a game.*
- K. Angelov and R. Enache. Typeful Ontologies with Direct Multilingual Verbalization. *CNL 2010, Controlled Natural Language*, Marettimo, 2010. *The SUMO ontology implemented in GF.*
- S. Virk, M. Humayoun and A. Ranta. An Open-Source Urdu Resource Grammar, *COLING 20010/The 8th Workshop on Asian Language Resources*, Beijing, 2010. *A report on the Urdu resource grammar implementation.*
- A. Ranta, K. Angelov and T. Hallgren. Tools for Multilingual Grammar-Based Translation on the Web. *ACL 2010 System Demo*, Uppsala, 2010. *An overview of GF for developers and users of translation systems.*
- D. Dannélls and J. Camilleri. Verb Morphology of Hebrew and Maltese—Towards an Open Source Type Theoretical Resource Grammar in GF. *Proceedings of the Language Resources (LRs) and Human Language Technologies (HLT) for Semitic Languages Status, Updates, and Prospects, LREC-2010 Workshop*, Malta, pp. 57-61. 2010. *A study of Semitic non-concatenative morphology from the GF point of view.*
- M. Humayoun and C. Raffalli. MathNat—Mathematical Text in a Controlled Natural Language. *Special issue: Natural Language Processing and its Applications. Journal on Research in Computing Science*, Volume 46. 2010. *Natural language interface to a proof system, implemented in GF.*
- D. Dannélls. Discourse Generation from Formal Specifications Using the Grammatical Framework, GF. *Special issue: Natural Language Processing and its Ap-*

- lications. *Journal on Research in Computing Science (RCS)*, volume 46. pp. 167-178, 2010. *Interfacing GF with ontology, with a natural language generation perspective.*
- R. Enache, A. Ranta and K. Angelov. An open-source computational grammar for Romanian. *CICLing-2010*, LNCS, Vol. 6096, Springer, 2010. *A report on the Romanian resource grammar implementation.*
- K. Angelov and A. Ranta. Implementing Controlled Languages in GF, Proceedings of *CNL-2009*, LNCS, Vol. 5972, Springer, 2009. *On the use of GF for controlled languages, exemplified by an implementation of Attempto Controlled English also ported to three other languages.*
- A. Ranta. The GF Resource Grammar Library, *Linguistics in Language Technology*, Vol. 2(2), 2009. *A systematic presentation of the library from the linguistic point of view.*
- A. Ranta. Grammatical Framework: A Multilingual Grammar Formalism, *Language and Linguistics Compass*, Vol.3, 2009. *An overview of GF for readers with a general academic background.*
- A. Ranta. Grammars as Software Libraries. In Y. Bertot, G. Huet, J-J. Lévy, and G. Plotkin (eds.), *From Semantics to Computer Science*, Cambridge University Press, Cambridge, 2009. *Grammar libraries from the software engineering point of view, with an example application to mathematical language.*
- K. Angelov. Incremental Parsing in Parallel Multiple Context-Free Grammars. *EACL 2009*. *Describes the algorithm used in parsing with GF.*
- A. Ranta, B. Bringert, and K. Angelov. The GF Grammar Development Environment. System demo. Proceedings of *EACL-2009*, 2009. *An overview of GF from the grammarian's point of view.*
- B. Bringert, K. Angelov, and A. Ranta. Grammatical Framework Web Service, System demo. Proceedings of *EACL-2009*, 2009. *An overview of how to build web services on top of PGF using the Google Web Toolkit.*
- B. Bringert. Semantics of the GF Resource Grammar Library. Report, Chalmers University. 2008. *Montague-style semantics for the Resource Grammar Library, applied to the FraCaS test suite of computational semantics via an extension of the parsing grammar and the use of the Oxford English Dictionary.*
- B. Bringert. *Programming Language Techniques for Natural Language Applications*. PhD Thesis, University of Gothenburg, 2008. *Collection of articles about grammar-related algorithms and grammar-based speech applications.*
- B. Bringert and A. Ranta. A Pattern for Almost Compositional Functions. *Journal of Functional Programming*, 18(5-6), pp. 567-598, 2008; an extended version of a conference paper in *ICFP 2006, The 11th ACM SIGPLAN International Conference on Functional Programming, Portland, Oregon, 2006*. *A functional programming technique for defining transfer-like functions with the minimum of effort.*
- R. Cooper and A. Ranta. Natural Languages as Collections of Resources. In *Language in Flux: Dialogue Coordination, Language Variation, Change*, ed. by R. Cooper and R. Kempson, pp. 109-120. College Publications, London, 2008. *The resource grammar idea applied to language learning and evolution.*
- M. S. Meza Moreno and B. Bringert. Interactive Multilingual Web Applications with Grammatical Framework. In B. Nordström and A. Ranta (eds), *Advances in Natural Language Processing (GoTAL 2008)*, LNCS/LNAI 5221, Springer, 2008. *Shows how GF compiled to JavaScript is used in dynamic multilingual web pages.*
- P. Ljunglöf and S. Larsson. A grammar formalism for specifying ISU-based dialogue systems. In B. Nordström and A. Ranta (eds), *Advances in Natural Language*

- Processing (GoTAL 2008)*, LNCS/LNAI 5221, Springer, 2008. *Explains how GoDiS dialogue systems are specified by GF grammars.*
- K. Angelov. Type-Theoretical Bulgarian Grammar. In B. Nordström and A. Ranta (eds), *Advances in Natural Language Processing (GoTAL 2008)*, LNCS/LNAI 5221, Springer, 2008. *Explains the implementation of a Bulgarian resource grammar in GF.*
- B. Bringert. High-precision Domain-specific Interlingua-based Speech Translation with Grammatical Framework. *Coling 2008 Workshop on Speech Translation for Medical and Other Safety-Critical Applications*, Manchester, UK, August 23, 2008. *Shows how to build spoken language translators based on GF grammars and their compilation to Nuance.*
- A. Ranta. How predictable is Finnish morphology? An experiment on lexicon construction. In J. Nivre, M. Dahllöf and B. Megyesi (eds), *Resourceful Language Technology: Festschrift in Honor of Anna Sägvall Hein*, University of Uppsala, 2008. *Presents an experiment on smart paradigms in Finnish.*
- R. Cooper. The abstract-concrete syntax distinction and unification in multilingual grammar. In J. Nivre, M. Dahllöf and B. Megyesi (eds), *Resourceful Language Technology: Festschrift in Honor of Anna Sägvall Hein*, University of Uppsala, 2008. *A comparison between GF and unification grammars.*
- M. Forsberg. *Three Tools for Language Processing: BNF Converter, Functional Morphology, and Extract*. PhD Thesis, Chalmers University of Technology, 2007. *Articles and detailed documentation about GF Spin-Off projects.*
- A. Ranta. Example-based grammar writing. To appear in L. Borin and S. Larsson (eds), *Festschrift for Robin Cooper*, 2007. *Presents and discusses the ideas of grammar composition and example-based grammar writing.*
- B. Bringert. Rapid Development of Dialogue Systems by Grammar Compilation. *8th SIGdial Workshop on Discourse and Dialogue*, Antwerp, Belgium, September 1-2, 2007. *Shows how to build a web-based spoken dialogue system by generating VoiceXML and JavaScript.*
- A. El Dada and A. Ranta. Implementing an Open Source Arabic Resource Grammar in GF. In M. Mughazy (ed), *Perspectives on Arabic Linguistics XX. Papers from the Twentieth Annual Symposium on Arabic Linguistics, Kalamazoo, March 26* John Benjamins Publishing Company. 2007. *An outline of the Arabic resource grammar project, focusing on linguistic aspects.*
- A. El Dada. Implementation of the Arabic Numerals and their Syntax in GF. *Computational Approaches to Semitic Languages: Common Issues and Resources*, ACL-2007 Workshop, June 28, 2007, Prague. 2007. *A case study with the resource grammar, focusing on the morphosyntax and agreement of constructions with numerals.*
- A. Ranta. Modular Grammar Engineering in GF. *Research on Language and Computation*, 5:133-158, 2007. *Adapts library-based software engineering methods to grammar writing and introduces the module system of GF.*
- A. Ranta. The GF Grammar Compiler. *Workshop on New Directions in Type-theoretic Grammars*, Dublin, August 2007 (ESSLLI workshop). 2007. *Describes the compilation of GF source code to lower-level run-time formats.*
- M. Humayoun, H. Hammarström, and A. Ranta. Urdu Morphology, Orthography and Lexicon Extraction. *CAASL-2: The Second Workshop on Computational Approaches to Arabic Script-based Languages*, July 21-22, 2007, LSA 2007 Linguistic Institute, Stanford University. 2007. *Fairly complete open-source Urdu morphology and elementary syntax in GF.*
- N. Perera and A. Ranta. Dialogue System Localization with the GF Resource Grammar Library. *SPEECHGRAM 2007: ACL Workshop on Grammar-Based*

- Approaches to Spoken Language Processing*, June 29, 2007, Prague. 2007. *An experiment in porting an in-car dialogue system from two to six languages.*
- B. Bringert. Speech Recognition Grammar Compilation in Grammatical Framework *SPEECHGRAM 2007: ACL Workshop on Grammar-Based Approaches to Spoken Language Processing*, June 29, 2007, Prague. 2007. *Generation of speech recognition language models from GF to several formats: GSL (Nuance), SRGS, JS GF, and HTK SLF, with embedded semantic interpretation.*
 - A. Ranta. Features in Abstract and Concrete Syntax. *The 2nd International Workshop on Typed Feature Structure Grammars*, Tartu, 24 May 2007 (NODALIDA workshop). 2007. *Explores the design choices of incorporating features in a GF-like grammar, with comparisons to feature-based unification grammars.*
 - O. Lemon and X. Liu. DUDE: a Dialogue and Understanding Development Environment, mapping Business Process Models to Information State Update dialogue systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics.*, 2006. *A GF-based system for rapid development of dialogue systems.*
 - P. Ljunglöf, G. Amores, R. Cooper, D. Hjelm, O. Lemon, P. Manchón, G. Pérez, and A. Ranta. Multimodal Grammar Library. TALK. Talk and Look: Tools for Ambient Linguistic Knowledge. IST-507802. Deliverable 1.2b, 2006. *An extension of the Resource Grammar Library with multimodality (pointing), with an application to dialogue systems.*
 - O. Caprotti and M. Seppälä. Multilingual Delivery of Online Tests in mathematics. Proceedings of Online Educa Berlin 2006. Berlin, 2006. *This papers shows screenshots of multilingual generation in the WebALT project, using GF and the Resource Grammar Library.*
 - J. Khagai. Language engineering in Grammatical Framework (GF). PhD thesis, Computer Science, Chalmers University of Technology, 2006. *Collection of articles and technical reports on multilingual authoring and the Russian resource grammar.*
 - M. Forsberg, H. Hammarström, and A. Ranta. Morphological Lexicon Extraction from Raw Text Data. *FinTAL 2006*, Turku, August 23-25, 2006. Springer LNCS/LNAI 4139, pp. 488-499, 2006. *A method for automatic production of morphological lexica based on inflection engines such as those of GF Resource Grammar Library.*
 - A. Ranta. Type Theory and Universal Grammar. *Philosophia Scientiae, Constructivism: Mathematics, Logic, Philosophy and Linguistics*, cahier spécial 6, pp. 115-131, 2006. *A philosophical study of the medieval thesis that grammar is the same in all languages and the difference is only in words.*
 - J. Khagai. GF parallel resource grammars and Russian. In proceedings of ACL2006 (The joint conference of the International Committee on Computational Linguistics and the Association for Computational Linguistics) (pp. 475-482), Sydney, Australia, July 2006. *Gives an outline of the Russian resource grammar project.*
 - J. Khagai. Grammatical Framework (GF) for MT in sublanguage domains. *Proceedings of 11th Annual conference of the European Association for Machine Translation*, Oslo. pp. 95-104, 2006. *Shows how GF is used in controlled language translation.*
 - W. Ng'ang'a. Multilingual content development for eLearning in Africa. eLearning Africa: 1st Pan-African Conference on ICT for Development, Education and Training. 24-26 May 2006, Addis Ababa, Ethiopia. 2006. *Presents a programme for producing educational material in African languages via multilingual generation in GF.*
 - R. Jonson. Generating statistical language models from interpretation grammars

- in dialogue system. In Proceedings of EACL'06, Trento, Italy, 2006. *Uses GF grammars to generate statistical language models for speech recognition.*
- A. El Dada and A. Ranta. Arabic Resource Grammar. *Arabic Language Processing (JETALA)*, 5-6 June 2006, IERA, Rabat, Morocco, 2006. *An outline of the Arabic resource grammar project, focusing on software aspects.*
 - D. A. Burke and K. Johannisson. Translating Formal Software Specifications to Natural Language. A Grammar-Based Approach. In P. Blache, E. Stabler, J. Bustquets and R. Moot (eds), *Logical Aspects of Computational Linguistics (LACL 2005)*, Springer LNAI 3402, pp. 51-66, 2005. *A paper explaining how a multilingual GF grammar is completed with Natural Language Generation techniques to improve text quality.*
 - B. Bringert, R. Cooper, P. Ljunglöf, A. Ranta, Multimodal Dialogue System Grammars. *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France, June 9-11, 2005*, 2005. *Shows how mouse clicks can be integrated in GF grammars alongside with speech input.*
 - K. Johannisson, Formal and Informal Software Specifications. PhD thesis, Computer Science, Göteborg University, 2005. *Collection of articles in the GF-KeY project, with an introduction.*
 - A. Ranta. Declarative Language Definitions and Code Generation as Linearization. TYPES Meeting, Jouy-en-Jossas, 2004c. Available through GF homepage. *An implementation of a compiler from a fragment of C to JVM, using a multilingual GF grammar and dependent types.*
 - P. Ljunglöf. Expressivity and Complexity of the Grammatical Framework. PhD thesis, Computer Science, Göteborg University, 2004. *Language-theoretical study of GF and its parsing problem.*
 - A. Ranta. Computational semantics in type theory. *Mathematics and Social Sciences*, 165, pp. 31-57, 2004b. *Shows how Montague-style grammars are implemented in GF and extends this to type-theoretical grammars for anaphoric expressions.*
 - A. Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2), pp. 145-189, 2004. *Theoretical paper explaining the GF formalism and its implementation. The standard reference on GF, but doesn't cover the module system.*
 - H. Hammarström and A. Ranta. Cardinal Numerals Revisited in GF. *Workshop on Numerals in the World's Languages*. Dept. of Linguistics Max Planck Institute for Evolutionary Anthropology, Leipzig, 2004. *An overview of the numeral grammar project, covering 88 languages.*
 - A. Ranta. Grammatical Framework Tutorial. In A. Beckmann and N. Preining, editors, *ESSLLI 2003 Course Material I, Collegium Logicum*, volume V, pp. 1-86. Kurt Gödel Society, Vienna, 2004. *A revised version of the on-line GF tutorial, v1.0.*
 - J. Khagai and A. Ranta. Building and Using a Russian Resource Grammar in GF. In A. Gelbukh (ed), *Intelligent Text Processing and Computational Linguistics (CICLing-2004)*, Seoul, Korea, November 2004, Springer LNCS 2945, pp. 38-41, 2004. *An introduction to the GF resource grammar project, with Russian as prime example.*
 - A. Ranta and R. Cooper. Dialogue Systems as Proof Editors. *Journal of Logic, Language and Information*, 13, pp. 225-240, 2004. *Shows a close analogy between task-oriented dialogue systems and metavariable-based proof editors.*
 - J. Khagai, B. Nordström, and A. Ranta. Multilingual Syntax Editing in GF. *Intelligent Text Processing and Computational Linguistics (CICLing-2003)*, Mexico City, February 2003, Springer LNCS 2588, pp. 453-464, 2003. *Explains how*

GF is used in multilingual syntax editing and discusses how new grammars are created.

- R. Hähnle, K. Johannisson, and A. Ranta. An Authoring Tool for Informal and Formal Requirements Specifications. In R. D. Kutsche and H. Weber (eds), *ETAPS/FASE-2002: Fundamental Approaches to Software Engineering*, Springer LNCS, vol. 2306, pp. 233–248, 2002. *Describes a GF-based authoring tool for object-oriented specifications in OCL and English. Carries out in full the work proposed in the position paper (Hähnle & Ranta 2001).*
- K. Johannisson and A. Ranta, Formal Verification of Multilingual Instructions. *Proceedings of the Joint Winter Meeting 2001*. Departments of Computer Science and Computer Engineering, Chalmers University of Technology and Göteborg University. 2001. *Instructions for an alarm system in four languages, verified in the proof editor Alfa.*
- R. Hähnle and A. Ranta, Connecting OCL with the Rest of the World. *ETAPS 2001 Workshop on Transformations in UML (WTUML)*, Genova, 2001. *A position paper explaining how GF can be used to help in object-oriented modelling, with some examples on natural-language interaction with OCL (Object Constraint Language).*
- T. Hallgren, The Correctness of Insertion Sort, Manuscript, Chalmers University, Göteborg, 2001. Available in *A seven-page text generated by GF-Alfa.*
- A. Ranta. *Bescherelle bricolé*, 2001. *A machine-generated book on French conjugation implemented in GF.*
- T. Hallgren and A. Ranta, An Extensible Proof Text Editor. In M. Parigot and A. Voronkov (eds), *Logic for Programming and Automated Reasoning (LPAR'2000)*, LNCS/LNAI 1955, pp. 70–84, Springer Verlag, Heidelberg, 2000. *Describes an interface to the proof editor Alfa written in GF.*
- M. Dymetman, V. Lux, and A. Ranta, XML and multilingual document authoring: converging trends. *Proceedings of the The 18th International Conference on Computational Linguistics (COLING 2000)*, pp. 243–249, Saarbruecken, 2000. *Abstract syntax as interlingua, as implemented by XML, GF, and definite clause grammars.*
- P. Mäenpää and A. Ranta. The type theory and type checker of GF. *Colloquium on Principles, Logics, and Implementations of High-Level Programming Languages, Workshop on Logical Frameworks and Meta-languages, Paris, 28 September 1999*. 1999. *Concise theoretical presentation of GF, using the old notation prior to v0.9.*

F.2 Background and related work

In alphabetical order:

- Alshawi, H. *The Core Language Engine*. MIT Press, Cambridge, Ma. 1992. *A set of parallel grammars written in Prolog and used as a library; inspiration of GF resource grammars.*
- Appel, A. *Modern Compiler Implementation in ML*. Cambridge University Press. 1998. *Compiler text book presenting the main techniques underlying language processing in GF: abstract syntax, pattern matching, parser generation.*
- Axelsson, J., C. Cross, J. Ferrans, G. McCobb, T. V. Raman, and L. Wilson XHTML+Voice profile 1.2. Specification, VoiceXML Forum. 2004. *One of the formats generated from GF, to specify dialogue systems.*
- Beckert, B., R. Hähnle, and P. H. Schmitt (Eds.). *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag. 2007. *A system that includes a GF-plugin for syntax editing formal and informal software specifications.*

- Beesley, K. and L. Karttunen. *Finite State Morphology*. CSLI Publications. 2003.
- Xerox *Finite State Tool*, the most influential tool in computational morphology.
- Bender, E. M. and D. Flickinger. Rapid prototyping of scalable grammars: Towards modularity in extensions to a language-independent core. In *Proceedings of the 2nd International Joint Conference on Natural Language Processing IJCNLP-05 (Posters/Demos)*, Jeju Island, Korea. 2005. *The LinGO Matrix project for parallel grammars in HPSG*.
- Bescherelle. *La conjugaison pour tous*. Hatier. 1997. *The authoritative source for French verb conjugation, also available for Italian, Spanish, Arabic, and some other languages*.
- Bevocal Inc. Nuance GSL Grammar Format. URL <http://cafe.bevocal.com/docs/grammar/gsl.html>. 2005. *One of the speech recognition formats generated from GF*.
- Bird, S., E. Klein, and E. Loper. *Natural Language Processing with Python. Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, 2009. *Building natural language processing applications in Python; these may include GF grammars via Python bindings to GF run time*.
- Blackburn, P. and J. Bos. *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI Publications. 2005. *Applications of syntactic and semantic analysis implemented in Prolog; an example of the embedded grammar idea*.
- Bos, J., S. Clark, M. Steedman, J. Curran, and J. Hockenmaier. Wide-Coverage Semantic Representations from a CCG Parser. In *Coling 2004*. 2004. *Benchmark for wide-coverage grammar-based parsing with semantics*.
- Bresnan, J. (Ed.). *The Mental Representation of Grammatical Relations*. MIT Press. 1982. *The standard text on the LFG grammar formalism*.
- Brown, P.F., J. Cocke, S.A. Della Pietra, V.J. Della Pietra, F. Jelinek, J.D. Lafferty, R.L. Mercer, and P.S. Roossin. A statistical approach to machine translation. *Computational Linguistics* 16 (2), pp. 79–85, 1990. *The “IBM Approach” pioneering the modern era of statistical machine translation*.
- de Bruijn, N. G. Mathematical Vernacular. In *Collected Papers on Automath*, North-Holland, 1994. *A research programme for building a natural-language-like notation for type theory to express mathematics*.
- Burden, H. and P. Ljunglöf. Parsing Linear Context-Free Rewriting Systems. In *Proceedings of the Ninth International Workshop on Parsing Technology*, Vancouver, British Columbia, pp. 11–17. Association for Computational Linguistics. 2005. *Underlying ideas for parsing GF*.
- Burton, R. R. and J. S. Brown. Semantic grammar: A technique for constructing natural language interface to instructional systems. In *BBN Report 3587*. Cambridge, Ma.: Bolt, Beranek, and Newman, Inc. 1977. *The pioneering work on what GF calls application grammars*.
- Butt, M., H. Dyvik, T. H. King, H. Masuichi, and C. Rohrer. The Parallel Grammar Project. In *COLING 2002, Workshop on Grammar Engineering and Evaluation*, pp. 1–7. 2002. *The multilingual grammar project using LFG*.
- Chomsky, N. Three models for the description of language. *IRE Transactions on Information Theory*, Vol. 2, pp. 113–124, 1956. *The first definition of the “Chomsky hierarchy” of formal languages*.
- Chomsky, N. *Syntactic Structures*. The Hague: Mouton. 1957. *The pioneering work on formalizing the syntax of natural language*.
- Claessen, K. Equinox, A New Theorem Prover for Full First-Order Logic with Equality. In *Dagstuhl Seminar 05431 on Deduction and Applications*. 2005. *A theorem prover used in computing with the first-order logic semantics of the resource*

- grammar in Bringert 2008.*
- Copestake, A., D. Flickinger, C. Pollard, and I. Sag. Minimal Recursive Semantics: An Introduction. *Research on Language and Computation*, Vol. 3, pp. 281–332, 2005. *Meaning representation formalism used for translation in HPSG, LFG, and TAG.*
- Copestake, A. *Implementing Typed Feature Structure Grammars*. CSLI Publications, 2002. *The LKB grammar development system widely used for HPSG.*
- Curry, H. B. Some logical aspects of grammatical structure. In R. Jakobson (Ed.), *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pp. 56–68. American Mathematical Society, 1961. *The origin of the idea of tectogrammatics and phenogrammatics, also known as abstract and concrete syntax.*
- de Groote, P. Towards Abstract Categorical Grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter*, Toulouse, France, pp. 148–155, 2001. *A grammar formalism reminiscent of GF, based on linear lambda calculus.*
- Dean, M. and G. Schreiber. OWL Web Ontology Language Reference, 2004. URL <http://www.w3.org/TR/owl-ref/>. *Meaning representation for the semantic web.*
- Diderichsen, P. *Elementaer dansk grammatik*. Kobenhavn, 1962. *The topological structure of Scandinavian languages, a prime example of discontinuous constituents.*
- Doets, K. and J. van Eijck. *The Haskell Road to Logic, Maths and Programming*. College Publications, 2004. *Introduction to Haskell for those interested in logic.*
- Dowty, D. *Word Meaning and Montague Grammar*. Dordrecht: D. Reidel, 1979. *Points out that Montague grammar (Montague 1974) is an instance of Curry's distinction between tectogrammatics and phenogrammatics (Curry 1961).*
- Forsberg, M. and A. Ranta. Functional Morphology. In *ICFP 2004*, Snowbird, Utah, pp. 213–223, 2004. *A spin-off of GF: a Haskell library for implementing morphology.*
- Frost, R. Realization of natural language interfaces using lazy functional programming. *ACM Comput. Surv.* 38(4), 2006. *A survey of NLP in Haskell-like languages.*
- Gazdar, G., E. Klein, G. Pullum, and I. Sag. *Generalized Phrase Structure Grammar*. Basil Blackwell, Oxford, 1985. *Slash categories and lots of other ideas using categorial grammar and unification.*
- Grevisse, M. *Le bon usage, 13me edition*. Paris: Duculot, 1993. *The authoritative French grammar, also systematically using the distinction between inherent and variable features.*
- Harper, R., F. Honsell, and G. Plotkin. A Framework for Defining Logics. *JACM* 40(1), 143–184, 1993. *The standard logical framework based on type theory, used for the abstract syntax in GF.*
- Hockett, C. F. Two models of grammatical description. *Word* 10, 210–233, 1954. *The item-and-arrangement and item-and-process models—plus a third one, word-and-paradigm, which is the one mostly followed in GF.*
- Huet, G. A Functional Toolkit for Morphological and Phonological Processing, Application to a Sanskrit Tagger. *The Journal of Functional Programming* 15(4), 573–614, 2005. *Functional programming in OCAML for large-scale and efficient morphology implementations.*
- Ishii M., K. Ohta, and H. Saito. An Efficient Parser Generator for Natural Language. *15th International Conference on Computational Linguistics*, Kyoto, Japan, pp. 417–420, 1994. *The NLYACC parser generator for context-free grammars.*

- Jackendoff, R. *X-Bar Syntax: A Study of Phrase Structure*. MIT Press, 1977. *One of the most elegant and influential theories in linguistic syntax.*
- Johnson, S. C. Yacc—yet another compiler compiler. Technical Report CSTR-32, AT&T Bell Laboratories, Murray Hill, NJ, 1975. *Despite its name, the first still known implementation of a grammar formalism for programming languages.*
- Joshi, A. Tree-adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions. In D. Dowty, L. Karttunen, and A. Zwicky (Eds.), *Natural Language Parsing*, pp. 206–250. Cambridge University Press, 1985. *The standard paper on TAG, a mildly context-sensitive grammar formalism.*
- Jurafsky, D. and J. Martin. *Speech and Language Processing*. Prentice Hall, 2000. *Standard reference to natural language processing, covering almost everything; new edition in 2008. Cf. Nuges 2006.*
- Kernighan, B. W. and D.M. Ritchie. *The C Programming Language*. Prentice Hall. First edition 1978. *One of the most influential programming language texts: tutorial, advanced applications, and reference manual in less than 300 pages; a model for this book as well (together with Koenig and Moo).*
- Kamp, H. A theory of truth and semantic representation. In J.A.G. Groenendijk, T.M.V. Janssen, and M.B.J. Stokhof (eds.), *Formal Methods in the Study of Language. Mathematical Centre Tracts 135*, Amsterdam, pp. 277–322, 1981. *Introduces Discourse Representation Theory to solve problems with the logical semantics of anaphora.*
- Kamp, H., R. Crouch, J. van Genabith, R. Cooper, M. Poesio, J. van Eijck, J. Jaspars, M. Pinkal, E. Vestre, and S. Pulman. Specification of linguistic coverage. FRACAS Deliverable D2, 1994. *The computational semantics test suite used for GF by Bringert 2008.*
- Koenig, A. and Moo, B. E. *Accelerated C++*. Addison-Wesley, 2000. *An excellent introduction to C++, presenting the standard libraries before the low-level language details; another model for this book (with Kernighan and Ritchie).*
- Kotimaisten Kielten Tutkimuskeskus. KOTUS Wordlist, 2006. URL <http://kaino.kotus.fi/sanat/nykysuomi>. *Comprehensive open-source morphological lexicon of Finnish, reused in GF Resource Grammar Library.*
- Larsson, S. *Issue-based Dialogue Management*. PhD thesis, University of Gothenburg, 2002. *Standard reference on dialogue systems based on an information state updates.*
- Lopez, A. Statistical Machine Translation. *ACM Computing Surveys* 40 (3), 2008. *A survey of contemporary techniques and issues in statistical machine translation, including the use of grammatical information.*
- Lyons, J. *Introduction to Theoretical Linguistics*. Cambridge University Press, 1968. *If you want to read just one book about linguistics, you get both far and deep with this one.*
- Mäenpää, P. Semantical BNF. In *Types for Proofs and Programs 1996*, LNCS 1512, Springer, 1998. *Using dependently typed syntax for program semantics, in comparison to attribute grammars.*
- Magnusson, L. *The Implementation of ALF—a Proof Editor based on Martin-Löf’s Monomorphic Type Theory with Explicit Substitutions*. PhD Thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, 1994. *Back in 1992 the most wonderful program in the world, ALF is a model that GF directly follows: GF is Yet ALF.*
- Martin-Löf, P. *Intuitionistic Type Theory*. Napoli: Bibliopolis, 1984. *A lucid exposition of type theory on less than 100 pages.*
- McCarthy, J. Towards a mathematical science of computation. In *Proceedings of*

- the *Information Processing Cong.* 62, Munich, West Germany, pp. 21–28. North-Holland, 1962. *Introduces the notion of abstract syntax in connection to LISP.*
- Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17, 348–375, 1978. *Simple type theory generalized with type parameters.*
- Milner, R., M. Tofte, and R. Harper. *Definition of Standard ML*. MIT Press, 1990. *The first major typed functional programming language.*
- Montague, R. *Formal Philosophy*. Collected papers edited by Richmond Thomason. New Haven: Yale University Press, 1974. *The pioneering formalization of syntax and semantics together; GF can be seen as a framework for Montague-style grammars.*
- Muskens, R. Lambda Grammars and the Syntax-Semantics Interface. In R. van Rooy and M. Stokhof (Eds.), *Proceedings of the Thirteenth Amsterdam Colloquium*, Amsterdam, pp. 150–155, 2001. *Another Curry-style grammar formalism, aimed for semantic descriptions.*
- Nordström, B., K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: an Introduction*. Oxford University Press, 1990. *Standard reference on the subject.*
- Nugues, P. *An Introduction to Language Processing with Perl and Prolog*. Springer 2006. *As a book that aims to cover all NLP, serious competitor to Jurafsky and Martin: more concise and less English-bound.*
- Pereira, F. and S. Shieber. *Prolog and Natural-Language Analysis*. CSLI, Stanford, 1987. *Using Prolog and embedded definite clause grammars for natural language processing.*
- Pereira, F. and D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence* 13, 231–278, 1980. *Standard reference on definite clause grammars.*
- Peyton Jones, S (ed.). *Haskell 98 language and libraries*. Cambridge University Press, 2003. *Standard reference to Haskell.*
- Pollard, C. Higher-Order Categorical Grammar. In M. Moortgat (Ed.), *Proceedings of the Conference on Categorical Grammars (CG2004)*, Montpellier, France, pp. 340–361, 2004. *From the creator of HPSG, a Curry-style grammar formalism.*
- Pollard, C. and I. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, 1994. *The first standard book on HPSG.*
- Power, R. and D. Scott. Multilingual authoring using feedback texts. In *COLING-ACL*, 1998. *An interactive tool that inspired the GF syntax editor.*
- Ranta, A. Intuitionistic categorial grammar. *Linguistics and Philosophy* 14, 203–239, 1991. *Using constructive type theory for natural language syntax and semantics.*
- Ranta, A. *Type Theoretical Grammar*. Oxford University Press, 1994. *Using type theory for natural language syntax and semantics, with an early implementation of linearization (called "sugaring").*
- Ranta, A. Syntactic categories in the language of mathematics. In P. Dybjer, B. Nordström, and J. Smith, eds., *Types for Proofs and Programs*, pp. 162–182, Lecture Notes in Computer Science 996, Springer-Verlag, Heidelberg, 1995. *Predecessor of GF: a grammar defined in type theory and implemented in ALF.*
- Ranta, A. Context-relative syntactic categories and the formalization of mathematical text. In S. Berardi and M. Coppo, eds., *Types for Proofs and Programs*, pp. 231–248, Lecture Notes in Computer Science 1158, Springer-Verlag, Heidelberg, 1996. *Extending the theory of the previous paper. The implementation in ALF eventually became so heavy that the need arose for GF.*

- Ranta, A. Structures grammaticales dans le français mathématique. *Mathématiques, informatique et Sciences Humaines.*, vol. 138 pp. 5-56 and 139 pp. 5-36, 1997. *A French grammar presented in a type-theoretical style—together with a parallel English grammar, a predecessor of GF.*
- Ranta, A. Syntactic calculus with dependent types. *Journal of Logic, Language and Information*, vol. 4, pp. 413-431, 1998. *Interprets Lambek Calculus in type theory and defines some extensions.*
- Rayner, M., D. Carter, P. Bouillon, V. Digalakis, and M. Wirén. *The Spoken Language Translator*. Cambridge University Press, 2000. *Another book on CLE (Alshawi 1992), with very useful descriptions of grammar coverage.*
- Rayner, M., B. A. Hockey, and P. Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications, 2006. *The Regulus formalism that continues the tradition of CLE (previous entry).*
- Reichenbach, H. *Elements of Symbolic Logic*. The MacMillan Company, New York, 1948. *Famous for a tense system, which inspired the tenses in the GF Resource Grammar Library.*
- Sapir, E. *Language. An Introduction to the Study of Speech*. Harcourt, Brace & co., New York, 1921. *Pioneering work in modern empirical linguistics.*
- de Saussure, F. *Cours de linguistique générale*. Payot, Paris, 1913. *Establishes linguistics as the study of systems involving both expressions and meaning.*
- Seki, H., T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science* 88, 191–229, 1991. *The computational model of GF parsing.*
- Shieber, S. Evidence against the context-freeness of natural language. *Linguistics and Philosophy* 8, pp. 333–343, 1985. *The famous argument using Swiss German.*
- Shieber, S. *An Introduction to Unification-Based Approaches to Grammars*. University of Chicago Press, 1986. *Clear and concise introduction to unification grammars.*
- Shieber, S. M. and Y. Schabes. Synchronous tree-adjoining grammars. In *COLING*, pp. 253–258, 1990. *Translation in the TAG grammar formalism.*
- Stallman, R. *Using and Porting the GNU Compiler Collection*. Free Software Foundation, 2001. *GCC, a multi-source multi-target compiler.*
- Steedman, M. Combinators and grammars. In R. Oehrle, E. Bach, and D. Wheeler (Eds.), *Categorial Grammars and Natural Language Structures*, pp. 417–442. D. Reidel, Dordrecht, 1988. *Categorial grammar without variable bindings—one of the early inspirations of GF.*
- Steedman, M. *The Syntactic Process*. The MIT Press, 2001. *The standard book about combinatory categorial grammar.*
- O’Sullivan, B., Stewart, B., and Goerzen, D. *Real World Haskell: Code You Can Believe In*. O’Reilly 2008. *A modern textbook on Haskell for heavy-duty programmers.*
- Swadesh, M. Towards Greater Accuracy in Lexicostatistic Dating. *International Journal of American Linguistics*, 21, pp. 121–137, 1955. *A lexicon of 207 basic words included in the test lexicon of the GF Resource Grammar Library.*
- Tapanainen, P. and Voutilainen, A. Tagging accurately—Don’t guess if you know. *Proceedings of ANLP ’94*, pp. 47–52, 1994. *A famous comparison between statistical and symbolic methods.*
- Union Mundial pro Interlingua. Interlingua Homepage, 2001. URL <http://www.interlingua.com/>. *An artificial language implemented in the GF Resource Grammar Library.*
- Warmer, J. and A. Kleppe. *The Object Constraint Language: Precise Modelling*

- with UML*. Addison-Wesley, 1999. *A formal specification language used in the GF-KeY project*.
- Young, S., G. Evermann, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland. The HTK Book, Version 3.2. Cambridge University Engineering Dept, 2002, URL <http://www.htk.eng.cam.ac.uk>. *One of the speech recognition frameworks supported by GF*.

Index

- ↓ (computation), 59, 253
- ≡ (syntactic sugar), 44, 253
- @ (alias pattern), 81, 283
- | (disjunctive pattern), 81, 283
- | (free variation), 47, 284
- | (separator in data), 145, 267
- | (separator in parameter types), 59, 270
- ! (selection), 59, 282
- { }(empty record type), 279
- {- (comment), 44
- # (pattern expression), 95, 283
- # (pattern macro), 283
- \$0 (bound variable label), 139
- ** (module extension), 86, 258
- ** (record extension), 93, 280
- * (product type), 94
- * (repetition pattern), 283
- ++ (concatenation), 39, 278
- + (concatenation pattern), 81, 283
- + (gluing), 65, 278
- # (path directive), 105, 288
- (comment), 44
- (exclusion of inheritance), 89
- (negative pattern), 283
- }(comment), 44
- > (function type), 38, 273
- > (in lambda abstract), 65, 273
- . (projection), 62, 280
- ::= (BNF production), 30
- <...> (tuple), 94
- <> (empty tuple), 279
- => (table type), 59, 281
- ? (metavariable), 47, 273
- ? (one-character pattern), 81, 283
- Base (list category), 275
- Cons (list category), 275
- Float (category), 150, 276
- GF_LIB_PATH , 105, 288
- Int (category), 150, 276
- Integer (concrete type), 281, 287
- Ints (parameter type), 270, 281, 287
- PType (type of parameter types), 287
- Predef (predefined resource), 287
- Str (token list), 44
- String (category), 150, 276
- Type (type of concrete types), 73, 287
- [...] (restricted inheritance), 89
- [C] (list category), 121, 275
- [] (empty token list), 45, 278
- \ (lambda), 65, 273
- \ \ (one-branch table), 60
- _ (wildcard), 60, 81, 275, 283
- abstract (module), 43, 259
- case (case expression), 81
- cat (judgement), 43, 265
- coding (flag), 55, 287
- concrete (module), 43, 259
- data (constructor declaration), 145, 267

- def** (semantic definition), 143, 266
- flags** (judgement), 43, 272
- fun** (judgement), 43, 266
- in** (let expression), 76, 285
- incomplete** (functor module), 109, 259
- instance** (module), 108, 259
- interface** (module), 108, 259
- let** (local definition), 76, 285
- lin** (judgement), 43, 268
- lincat** (judgement), 43, 268
- lindef** (judgement), 269
- lock_** (lock field), 287
- noexpand** (optimization strategy), 288
- of** (case expression), 81
- open** (in module header), 76, 258, 259
- oper** (judgement), 65, 271
- opt** (flag), 288
- overload** (operation overloading), 86, 271
- p1** (tuple label), 94
- param** (judgement), 59, 270
- pre** (prefix-dependent choice), 94, 278
- printname** (judgement), 269
- resource** (module), 76, 259
- startcat** (flag), 43, 288
- table** (table expression), 59, 281
- variants** (free variation), 284
- where** (local definition), 285
- with** (functor instantiation), 110
- abstract datatype, 76, 80
- abstract feature, 124
- abstract syntax, 11, 39, 254
- abstract syntax tree, 33
- abstraction, 229, 273
- acoustic model, 171
- adjectival phrase, 122
- aggregation, 147
- agreement, 14, 50, 61, 247
- agreement feature, 201
- algebraic datatype, 91
- alias pattern, 81, 283
- alpha conversion, 275
- ambiguity, 48
- anaphora, 141
- anaphora resolution, 47
- anaphoric expression, 141
- Android, 168
- anteriority, 122
- API, 21
- application, 273
- application grammar, 18
- Application Programmer's Interface, 21
- Arabic, 54, 71, 74, 83, 98, 118, 204
- argument type, 38, 266
- argument variable, 277
- ASCII, 54
- aspect, 124, 247
- assignment, 181
- asynchronous parsing, 174
- auxiliary verb, 210
- back end, 181
- Backus-Naur Format, 16, 29
- base grammar, 86
- basic type, 129, 265, 273
- Basque, 248
- batch compiler, 156
- beta conversion, 275
- beta-eta normal form, 275
- binding, 138, 273
- BNF, 16, 29
- BNF Converter, 184
- body, 138
- bound occurrence, 138
- bound variable label, 139
- boxed, 286
- branches, 60
- Bulgarian, 98, 222
- canonical form, 277
- case, 80, 247
- case expression, 81, 282
- Catalan, 98, 222, 229, 231
- Catalan numbers, 50

- category, 30, 254
- category declaration, 43, 129, 265
- category printname definition, 269
- category skeleton, 38
- character encoding, 54, 287
- Chinese, 54
- class, 137
- clause, 101, 247
- clitic, 211
- clitics, 204
- closed, 99
- command interpreter, 307
- comment, 44
- compile-time transfer, 117
- compiler, 4, 9, 181, 307
- compiler pragma, 288
- complement, 247
- complement phrase, 122
- complementation, 247
- completeness of concrete syntax, 43
- complexity, 8
- compositionality, 147, 195, 254
- computation, 266
- computation, we will use the, 59
- computational linguistics, 1
- compute, 143
- concatenation, 39, 278
- concatenation pattern, 81, 283
- concatenative pattern, 81
- concrete syntax, 11, 39, 254
- concrete syntax object, 276
- conditions of well-formedness, 128
- conflict, 257
- conjugation, 247
- connective, 140
- constant, 273
- constituency, 14
- constituent, 247
- constraint, 131
- constructive type theory, 128
- constructor, 65, 91, 145, 267
- constructor pattern, 266, 282
- consumer, 228
- content word, 99
- context, 129, 265
- context-free, 8, 266
- context-free grammar, 29
- context-sensitive, 8
- continuation, 182
- contract, 228
- contravariance, 93, 280
- conversion rule, 275
- coordination, 121, 134, 216, 247
- copula, 247
- copy language, 45
- corpus, 172, 247
- course-of-values table, 281
- covariance, 93, 280
- coverage, 7
- Curry-Howard isomorphism, 134
- currying, 38
- Danish, 98, 222, 229
- data, 4
- data abstraction, 79
- data constructor declaration, 267
- declension, 247
- decompiler, 181
- default linearization, 269
- Definite Clause Grammar, 17, 58
- definite description, 141
- definitional equality, 144, 266, 275
- delta conversion, 275
- demonstrative, 174
- denotation, 192
- denotational semantics, 192
- depend, 270
- dependency graph, 90
- dependency tree, 51
- dependent function type, 130
- dependent type, 58, 128, 266
- derivational morphology, 126
- design pattern, 110
- determiner, 57
- dialogue manager, 173
- dialogue system, 17, 173
- diamond pattern, 260

- diamond property, 88
- disambiguation, 151
- discontinuous constituent, 13, 70, 174, 247
- discontinuous constituents, 6
- discourse referent, 142
- Discourse Representation Theory, 24
- disjunctive pattern, 81, 283
- division of labour, 21
- domain lexicon, 86
- DRT, 24
- Dutch, 98, 222, 231
- embedded grammar, 17
- empty record, 279
- empty record type, 279
- empty token list, 45, 278
- emptying the shell environment, 43
- endocentric, 207, 247
- English, 5, 10, 11, 13, 14, 16, 18, 19, 21–23, 26, 30, 36, 38, 39, 46–48, 54, 57, 62, 65–67, 70, 71, 79, 80, 82, 86, 93, 94, 98, 99, 102, 104, 106, 110, 115, 116, 123, 124, 132, 137, 188, 192, 202, 204–207, 222, 224, 231
- enumerated type, 91
- ergative, 248
- ergativity, 203
- eta conversion, 275
- eta-expanded, 140, 276, 277
- exhaustive generation, 36
- exocentric, 208, 248
- expression, 265
- extended, 93
- extension, 86, 255
- extensional, 145
- extraction, 216
- families of language, 229
- feature structure, 93
- field, 62, 279
- finite function, 59, 281
- Finnish, 10, 11, 14, 16, 19, 22, 98, 102, 104, 112, 118, 119, 137, 203–207, 222, 224, 234, 235
- first match, 284
- flag definition, 43, 272
- flattening, 121, 232
- focus, 169
- forms of judgement, 43, 264
- fractal, 196
- free variation, 47, 284
- French, 5, 11, 13, 14, 18, 49, 98, 99, 119, 124, 206, 222, 229, 231, 232
- fridge poetry magnet, 166
- front end, 181
- function, 38, 64, 254
- function declaration, 43, 266
- function definition, 266
- function printname definition, 269
- function type, 273
- function types with variables, 76
- function word, 99
- functional fragment, 133, 273
- functional programming, 1, 9
- functional programming language, 64
- functional programming style, 229
- functor, 108, 256
- functor instantiation, 110
- GCC, 14
- gender, 50, 248
- generation, 30
- German, 6, 10, 13, 14, 21–23, 49, 50, 83, 91, 93, 98, 102, 104–106, 112, 124, 146, 207, 222, 231, 234, 235
- Germanic, 119, 231, 232
- GF command, 31
- GF object file, 256
- GF shell, 31, 307
- gluing, 278
- gluing operator, 65
- GNU Compiler Collection, 14

- gold standard, 226
- golden rule of functional programming, 64, 229
- graftal, 196
- grammar, 3
- grammar checking via type checking, 286
- grammar composition, 21
- grammar reuse, 125
- grammar-based language model, 171
- grammatical operation, 18
- grammatically correct combination, 98
- graphviz, 51
- Greek, 54, 118
- GSL, 171
- head, 207, 248
- header, 258
- Hebrew, 54, 71, 227
- hidden argument, 46
- higher precedence, 178
- higher-order abstract syntax, 139
- higher-order function, 266
- Hindi, 54, 55, 203, 248
- hint, 135
- homomorphicity, 254
- host language, 16, 155
- hybrid system, 8
- hypothesis, 129, 265
- identifier, 31, 257
- immutable, 260
- import, 33
- importing, 90
- including, 90
- incomplete, 5
- incomplete module, 256
- incremental parsing, 35
- inflection, 13, 248
- inflection form, 59
- inflection table, 59
- inflectional morphology, 22
- information extraction, 9
- information hiding, 90
- inherent feature, 61
- inheritance, 86, 255
- initialization, 182
- instance, 108, 256
- instantiated, 256
- integer literal pattern, 283
- integrated multimodality, 174
- intensional, 145
- interface, 80, 108, 256
- Interlingua, 98
- interoperability, 17
- interpretation function, 193
- intransitive, 248
- iPhone, 168
- ISO 639-3 language code, 42
- iso-latin-1, 54
- Italian, 10, 36–39, 42, 46–50, 58, 59, 61–64, 66, 67, 75, 76, 79, 88, 95, 98, 102, 104, 112, 117, 123, 166, 195, 202–206, 208–210, 212–215, 222, 229, 231, 232
- Japanese, 54
- Java, 166
- JavaScript, 168
- judgement, 43, 254, 264
- judgement keyword, 264
- label, 62, 279
- labelled rule, 30
- lambda abstract, 65
- language model, 171
- Latin, 98, 118, 249
- left-associative, 178
- legal linearization type, 268
- let expression, 76, 285
- lexeme, 248
- lexer, 53, 96
- lexical category, 98
- lexical insertion, 101
- lexical insertion function, 207
- lexical rule, 98
- lexical rules, 99
- lexicon, 248
- lexicon extraction tool, 235

- libraries, 66
- library, 9, 21
- linear, 283
- linearization, 11, 35, 39, 254
- linearization default definition, 269
- linearization definition, 268
- linearization function, 268
- linearization rule, 43
- linearization type, 39, 254, 268
- linearization type definition, 43, 268
- linguistic ontology, 118
- list category, 186, 275
- lists of parameter value, 270
- literal, 150, 273
- local definition, 76, 285
- lock field, 287
- logical framework, 128
- logical semantics, 192
- long-distance dependencies, 6
- machine learning, 4
- make facility, 257
- Maltese, 71
- matching substitution, 282
- metavariable, 46, 169, 273
- mildly context-sensitive grammar, 8
- miniature resource grammar, 199
- modification, 31
- modular grammar engineering, 255
- module body, 43
- module header, 43
- module hierarchy, 86
- module system, 9
- module type, 43
- monomorphic, 133, 137
- Montague grammar, 24
- Montague semantics, 192
- morphology, 13, 248
- morphology AP, 102
- multi-source, 14
- multi-target, 14
- multilingual grammar, 39, 254
- multilinguality, 10
- multimodality, 173
- multiple inheritance, 88, 260
- n-gram, 6, 172
- name, 44, 257
- name base, 264
- name resolution, 257, 262
- name space, 257
- natural language generation, 147, 151, 190
- negative pattern, 283
- NLG, 190
- non-concatenative morphology, 71
- normalization, 266
- Norwegian, 98, 222, 229
- noun phrase, 101
- number, 248
- object, 248
- one-character pattern, 81, 283
- ontology, 18
- open, 99
- opening, 256
- operation, 65, 255, 271
- operation definition, 65, 271
- optimization, 16, 257
- optimization flag, 288
- overgenerating, 5
- overload group, 271
- overload resolution, 85, 272
- overloading, 85
- paradigm, 248
- Parallel Multiple Context-Free Grammar, 10
- parameter, 58
- parameter constructor, 270
- parameter type, 255, 270
- parameter type definition, 59, 270
- parametrized module, 108, 256
- paraphrase, 151
- paraphrasing, 144
- parse tree, 33
- parsing, 11, 30, 254

- part of speech, 248
- partial application, 69
- partial evaluation, 257, 277
- path directive, 105
- pattern, 60
- pattern equation, 144
- pattern expression, 283
- pattern macro, 95, 283, 284
- pattern matching, 60, 267, 281
- pattern matching over strings, 81, 282
- permutation, 45
- PGF, 155
- phenogrammatical, 24
- phrasal category, 98
- phrasal rules, 99
- phrase category module, 221
- pipe, 35
- PMCFG, 10
- polarity, 122, 248
- Polish, 98, 222
- polymorphic, 132, 133, 137
- portability, 17
- Portable Grammar Format, 155, 256
- postfix, 181
- PP attachment, 48, 232
- precedence, 273
- precision, 7
- predefined category, 150, 276
- predicate, 137, 248
- predicate calculus, 140
- predication, 31, 248
- prefix-dependent choice, 94, 278
- printname, 269
- printname definition, 269
- private, 90
- pro-drop, 203, 216
- probabilistic context-free grammar, 150
- probabilistic GF grammar, 150
- probability configuration file, 150
- producer, 228
- product type, 280
- production, 30
- progressive implication, 142
- projection, 62, 279, 280
- prompt, 31
- pronoun, 46
- proof object, 135
- proof-carrying document, 136
- proposition, 101
- propositions as types principle, 134
- qualification, 90
- qualified, 261
- quantification, 192
- quantifier, 140
- question answering system, 16
- random generation, 35
- ranking of trees, 151
- record, 61, 279
- record extension, 280
- record pattern, 283
- record subtyping, 279
- record type, 62, 279
- recursive, 267
- reduplication, 45
- referent, 46
- refinement, 169
- regression testing, 226
- regular expression pattern, 81
- repetition, 229
- repetition pattern, 283
- resource grammar, 21
- Resource Grammar API, 21
- resource module, 255
- restricted, 261
- restricted inheritance, 88
- restricted polymorphism, 133, 137
- reverse Polish, 181
- reversibility, 254
- reversible, 12
- robust, 172
- Romance, 119, 124, 229, 231
- Romanian, 98, 222, 231
- run-time system, 307
- run-time variable, 95, 277

- Russian, 54, 98, 222
- Scandinavian, 10, 229, 231
- scope, 44
- script, 310
- selection, 59, 282
- selectional restriction, 130
- semantic action, 16, 179
- semantic control, 135
- semantic definition, 143
- semantic grammar, 18
- sentence, 122
- separate compilation, 41
- shell escape, 51
- single source, 171
- slash categories, 217
- SLM, 172
- Spanish, 98, 222, 229, 231
- speech recognition, 171
- speech to text, 171
- spurious ambiguity, 49
- stack machine, 181
- state, 307
- static checking, 255
- static type system, 9
- statistical approach, 4
- statistical language model, 171, 172
- statistical model, 4
- straight code, 181
- string, 278
- string literal, 184, 278
- string literal pattern, 283
- string pattern, 81
- string-based grammar, 50
- strong generative capacity, 46
- structural word, 99
- subject, 248
- substitution, 65, 273
- substitution test, 208
- subtype, 93
- subtyping, 280
- successor function, 134
- supertype, 93
- suppression, 45
- Swadesh list, 233
- Swedish, 98, 99, 116, 206, 207, 222, 224, 229, 235
- Swiss German, 71
- symbolic, 267
- symbolic address, 183
- symbolic approach, 4
- syncategorematic, 100
- syncretism, 91, 248
- syntactic constructor, 266
- syntactic grammar, 18
- syntactic structure, 18
- syntactic sugar, 44
- syntax, 248
- syntax API, 102
- syntax editor, 169
- syntax tree, 254, 266, 276
- synthesized corpus, 172
- system pipe, 52
- table, 281
- table expansion, 282
- table type, 59, 281
- tectogrammatical, 24
- Telugu, 54
- temporal order, 122
- tense, 122, 249
- term, 265
- Thai, 54
- token, 30, 53, 278
- token list, 278
- top-level grammar module, 255
- trace option, 35
- transfer, 146, 159
- transfer-based translation, 16
- transitive, 249
- translation quiz, 42, 151
- transliteration, 55, 227
- tree, 31, 33
- tree probability, 151
- treebank, 226
- tuple, 280
- tuple type, 280
- Turkish, 98, 224
- type, 38, 265
- type annotation, 257
- type discipline, 228

- type synonym, 73, 76
- type theory, 128
- type variable, 133
- typed functional programming
 - language, 9
- typing, 279

- umlaut, 83
- Unicode, 54
- unification, 25
- unification grammar, 58
- unlexer, 53, 96
- Urdu, 98, 222, 236
- UTF-8, 54
- utterance, 101, 119, 249

- value, 59, 277
- value assignment, 279
- value category, 266
- value type, 38, 266
- variable, 39, 273
- variable binding, 282
- variable feature, 60
- variable pattern, 81, 267, 283
- verb phrase, 70
- vocative, 249

- weak generative capacity, 45
- well-typed, 44
- wild card pattern, 283
- wildcard, 60, 129, 265, 275
- wildcard pattern, 81
- witness, 142
- word, 53
- word alignment, 13, 51
- word order, 202
- word sense disambiguation, 18
- worst-case function, 79
- wrap, 169

- X-bar theory, 208

GRAMMATICAL FRAMEWORK is a programming language designed for writing grammars, which has the capability of addressing several languages in parallel. This thorough introduction demonstrates how to write grammars in Grammatical Framework and use them in applications such as tourist phrasebooks, spoken dialogue systems, and natural language interfaces. The examples and exercises presented here address several languages, and the readers are shown how to look at their own languages from the computational perspective.

Requiring no previous knowledge of linguistics, the book is particularly suited for computer scientists and programmers. But it would also be of interest to linguists, as it presents a novel approach to multilingual grammars inspired by the theory of programming languages.

Aarne Ranta is professor of computer science at the University of Gothenburg, Sweden. He is the acting coordinator of the European Union research project MOLTO (Multilingual On-Line Translation), which develops techniques for high-quality translation among fifteen languages.