Thesis for the Degree of Doctor of Engineering

Three Tools for Language Processing: BNF Converter, Functional Morphology, and Extract

Markus Forsberg

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden 2007

Three Tools for Language Processing: BNF Converter,
Functional Morphology, and Extract
Markus Forsberg
Göteborg, Sweden, 2007
ISBN 978-91-7291-970-9

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie Nr 2651
ISSN 0346-718X

Technical Report no. 32D
Department of Computer Science and Engineering
Research group: Language Technology

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone + 46 (0)31-772 1000

Abstract

Purely functional programming and meta programming based on declarative models are productive approaches to language processing and language resource building. Three tools are presented as evidence of this: *BNF Converter*, *Functional Morphology*, and *Extract*.

BNF Converter is a multi-lingual compiler tool. BNFC accepts as its input a grammar written in Labelled BNF (LBNF) notation, and generates a compiler front end: an abstract syntax, a lexer, and a parser. Furthermore, it generates a case skeleton usable as the starting point of back end construction, a pretty printer, a test bench, and a LaTeX document usable as a language specification. The program components can be generated in Haskell, Java, C, C++, Objective Caml, and C#, and their standard parser and lexer tools.

Functional Morphology and Extract are tools for creating lexical resources. Lexical resources, i.e. systematic computational descriptions of words in a natural language, are fundamental resources for any language technology application and it is imperative that they are of high quality. Moreover, since the development of lexical resources is such a time-consuming task, it is important that they can be created efficiently. The tools have been created to address these issues.

Functional Morphology (FM) is a Haskell library for defining lexical resources. A lexical resource in FM is defined using the *word-and-paradigm* model. Paradigms are abstractions of inflection tables, represented as functions over hereditarily finite algebraic data types, and the lexicon consists of a list of words in citation form annotated with paradigm identifiers. The runtime system of FM consists of an inflection engine, an analyzer, a synthesizer and a compiler to many standard lexicon formats.

Extract is a special-purpose tool for extracting annotated words from raw text data. The extraction is based on a set of rules, where a rule is a propositional formula where the atoms of the formula are regular expressions. The regular expressions, corresponding to a subset of the word forms in a paradigm, contain variables used for capturing substrings. A recent addition to Extract is *Constraint Grammar* constructs together with the possibility of a structured input format. This addition enables a rule to refer to the contexts of the word forms and additional information added to them, such as part of speech (POS) tags.

The six papers included in this thesis, a peer-reviewed paper and a technical report for each tool, have been published previously as follows:

- **BNF Converter**
  - *Labelled BNF: A High-Level Formalism For Defining Well-Behaved Programming Languages*, Markus Forsberg & Aarne Ranta, Proceedings of the Estonian Academy of Sciences, Special issue on programming theory, NWPT'02, December 2003, pages 356–393
  - *BNF Converter: Multilingual Front-End Generation from Labelled BNF Grammars*, Michael Pellauer, Markus Forsberg & Aarne Ranta, Technical Report no. 2004-09 in Computing Science at Chalmers University of Technology and Göteborg University
- **Functional Morphology**
  - *Functional Morphology*, M. Forsberg & A. Ranta, Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, September 19-21, 2004, Snowbird, Utah, USA, pages 213–223
  - *The Functional Morphology Library.* M. Forsberg, Technical Report no. 2007-09 in Computing Science at Chalmers University of Technology and Göteborg University
- **Extract**
  - *Morphological Lexicon Extraction from Raw Text Data.* M. Forsberg, H. Hammarström, A. Ranta. FinTAL 2006, LNAI 4139, pp.488-499
  - *The Extract Tool.* M. Forsberg. Technical Report no. 2007-10 in Computing Science at Chalmers University of Technology and Göteborg University

Acknowledgements

First of all, I would like to thank my supervisor, Aarne Ranta, for his constant support and guidance throughout my PhD studies.

Secondly, I want to thank Vinnova and GSLT, for their generous fundings.

Furthermore, I thank all readers of this thesis, besides my supervisor, which have helped me a great deal in improving the quality of this thesis: Gérard Huet, Harald Hammarström, Bengt Nordström, Lars Borin, and Björn Bringert.

I would also like to thank all people in the Language Technology group, CLT, GSLT and my PhD committee, for all your inspiring discussions and general good company. Same for all co-workers at the Computing Science department at Chalmers.

Of course, my family and friends, as always, deserves a massive thank you for being there when I needed them, and for bringing joy to my life outside the office.

Finally, a special thank you with love, to my girlfriend Merja and her wonderful children Tova and Dennis, who take the 'K' out of my life.

Contents

Chapter 1

Introduction

1.1 General Introduction

It is our thesis that purely functional programming and meta programming based on declarative models are productive approaches to language processing and language resource building. As evidence of this we present three tools: *BNF Converter*, *Functional Morphology*, and *Extract*.

BNF Converter (BNFC) is a multi-lingual compiler tool for defining formal languages. Given a BNFC specification of a formal language, or a Labelled BNF (LBNF) in BNFC terminology, it can generate a complete front end: a parser, a lexer, an abstract syntax, documentation et cetera. The multi-lingual part of BNFC allows front ends for the specified formal language to be generated in many different programming languages: Haskell, C, C++, Java, C# and Objective Caml. The Haskell generation also allows generalized LR parsing (GLR), which enables ambiguous grammars. This, in turn, opens up for processing of natural languages, which are inherently ambiguous.

BNFC has proved useful as a compiler teaching tool. It encourages clean language design and declarative definitions. But it also lets the teacher spend more time on back end construction and/or the theory of parsing than traditional compiler tools, which require learning tricky and complicated notations.

BNFC also scales up to full-fledged language definitions. Even though real-world languages already have compilers generating machine code, it can be difficult to extract abstract syntax from them. A BNFC-generated parser, case skeleton, and pretty printer is a good starting point for programs doing some new kind of transformation or translation of an existing

language. Program transformation is used, for example, for dealing with legacy code, i.e. source code that is no longer supported. The goal of the program transformation could be to translate the source code into another programming language or to transform it to a more readable format.

However, the clearest case for BNFC is the development of new languages. It is easy to get started: just write a few lines of LBNF, run `bnfc`, and apply the `Makefile` to create a test bench, by which files written in the language can be parsed. Adding or changing a language construct is also easy, since changes only need to be done in one file. When the language design is complete, the implementor perhaps wants to change the implementation language; no work is lost, since the front end can be generated in a new target language, Finally, when the language implementation is ready to be given to users, a reliable and human-readable language definition is ready as well.

Functional Morphology (FM) and Extract were created to suggest a solution to the problem of creating high-quality *lexical resources* efficiently. A lexical resource is *a computational description of words in a human language*, where the computational description can be as simple as a word list or arbitrarily complicated, such as including whatever grammatical or semantic information. Lexical resources are fundamental in most language technology applications: word prediction on mobile phones, spell-checkers, educational material, machine translation and information retrieval among others.

However, many lexical resources already exist, so the question is why one would be interested in developing a new one. First of all, this statement is only true for languages with many speakers, such as English or Spanish, but is not necessarily true for smaller languages. Furthermore, even if resources exist, they may not fit present needs for one reason or another. For example, they may lack some crucial information that is needed, but hard or impossible to add — perhaps the resource is in a proprietary closed format or just structured inconveniently.

Before deciding to develop a lexical resource, it is important to realize that it is a time-consuming task, even just a word list of average quality takes a lot of time to develop. The reason is the size — the number of word forms in a substantial word list, i.e. all inflected forms of the words of the language, may reach several millions, depending on the degree of inflection in the language at hand.

The development of a lexical resource, as suggested in this thesis, consists of defining two things: an *inflection engine* and a *lexicon*. The inflection engine defines what kind of information we may annotate the words with.

The lexicon consists of a list of words annotated with the information stipulated by the inflection engine. The second tool presented in this thesis, *Functional Morphology*, defines the inflection engine together with the lexicon, and the third, *Extract*, helps the development of a lexicon through automated methods.

FM and Extract provide a toolbox for lexical resource development. They allow users to work on different levels, i.e. they allow a division of labour. An *implementer* of Extract or the FM library is required to have good knowledge of Haskell. A *morphology developer* needs less skill in Haskell, since she can learn by example, but must have good insights about the target language. A *lexicographer* needs no skill in Haskell or in FM, and, in fact, just average skills in the target language, since it is the morphology developer who does the most tricky linguistic work. If the lexicographer, on the other hand, wants to use Extract, she must have a good understanding about the target language to write successful paradigm rules.

BNFC has also played a role in the creation of the toolbox, since it has been used to develop Extract. The development of the necessary parts for processing the Extract language, e.g. a parser, a lexer, an abstract syntax, would previously been a substantial task, but with BNFC, this task have been reduced to the definition of the language by a BNF grammar.

1.2 Vocabulary

We will now briefly define some key concepts we will use when discussing lexical resources. We will not endeavor in a philosophical discussion, instead we will only go through the most important concepts and make the necessary distinctions for our purposes.

A *word* corresponds to a dictionary entry. A word has an *inflection table* containing the word's *word forms*. One word form, the *citation form*, *dictionary form*, or *lemma*, identifies a word, and is the one normally found in an ordinary dictionary, e.g. bare infinitives for verbs. The citation form is typically the word form considered as most characteristic, or the least marked, i.e. the word form with the least number of inflectional affixes.

A *paradigm* is an abstraction of an inflection table, which describes how a set of words is inflected.

A *lexical resource* is a computational description of the words of a language.

A *morphology*[1] is a lexical resource consisting of an *inflection engine*

[1]Note that we use the word 'morphology' ambiguously: it may refer to either the

and a *lexicon*. A lexicon is a list of citation forms annotated with paradigm names. An inflection engine is a computational device that translates a paradigm name together with a list of citation forms to a dictionary entry.

1.3 Contributions

The research contributions of the author of this thesis:

- The implementer of the major part of the Haskell generation, the first target language of *BNF Converter*, and together with A. Ranta, the designer of BNFC. The Java, C and C++ generation was developed by M. Pellauer, Java 1.5 by B. Bringert, Objective Caml by K. Johannisson and C# by J. Broberg.
- The main implementer of *Functional Morphology* and *Extract*.
- The author of the technical reports of Functional Morphology and Extract, in Chapter 5 and 7.
- The main author of the FM paper, together with A. Ranta, in Chapter 4.
- The main author of the BNF Converter paper, in Chapter 2.
- Co-author of the Extract paper, together with H. Hammarström and A. Ranta, in Chapter 6, where all authors contributed equally.
- Co-author of the technical report of BNF Converter, in Chapter 3, where M. Pellauer was the main author and A. Ranta co-authored.

1.4 Introduction to BNF Converter

BNF Converter, abbreviated BNFC, is a tool for defining formal languages, in particular programming languages. The idea was to construct a grammar tool for defining formal languages that was as declarative as possible, and from this tool generate necessary components for the front end of the target language, such as a lexical analyzer and a parser.

It started as an experimental study into what extent the Grammatical Framework tool (GF) [18] could be used as a compiler front end generator. Though it was possible, it was soon realized that some extra notation

linguistic object or to a morphology implementation.

was necessary to get a front end comparable to what a programmer would normally expect. For example, comments are usually something treated as white-spaces, instead of having them represented in the abstract syntax, and this requires some special notation. Hence, the BNF Converter tool was born.

Since the paper in Chapter 2 was written, substantial development has been carried out on BNFC, partly described in the technical report of BNFC in Chapter 3. In particular, multi-lingual support has been added and, due to the declarative nature of BNFC and because the back end tools used by BNFC have similar syntax and functionality, this was a relatively easy task. Support has been added for C, C++, C++ with Standard Template Library (STL), Java 1.4, Java 1.5, C# and Objective Caml.

Other functionalities added, since the papers included in this thesis were written, are XML generation, GF to BNFC translation, GADT support, layout support, generalized LR parsing [15], among others.

BNFC is currently part of the stable version of Linux distribution Debian[2] and its derivatives (e.g. Gentoo[3] and Ubuntu[4]). This shows that the tool has been accepted by the open source community.

1.4.1 Overview of BNFC

The grammar format of BNFC is LBNF, an abbreviation of Labelled BNF. The files that are generated in all target languages by BNFC from an LBNF grammar are: an abstract syntax, a lexer, a parser, a pretty-printer, a case-skeleton that recursively traverses the abstract syntax, a test bench that puts everything together in an executable, and a language document.

Let us look at a grammar for Lambda calculus defined in LBNF. An example of a Lambda calculus expression is `(\x -> x x) (\y -> y y)`.

```
Lambda . Exp  ::= "\\" Ident "->" Exp ;
App    . Exp1 ::= Exp1 Exp2 ;
Id     . Exp2 ::= Ident ;
coercions Exp 2 ;
```

The basic structure is a BNF grammar, consisting of three rules, augmented with labels (`Lambda`, `App` and `Id`). The labels corresponds to the constructors of the abstract syntax. The category `Ident` is a built-in token type for identifiers.

[2] `http://www.debian.org`
[3] `http://www.gentoo.org`
[4] `http://www.ubuntu.com`

The statement `coercions Exp 2` is a macro for the following conversion rules. The wild card label `_` means that the use of the rule will not be visible in the abstract syntax.

```
_ . Exp   ::= Exp1 ;
_ . Exp1  ::= Exp2 ;
_ . Exp2  ::= "(" Exp ")" ;
```

BNFC avoids cluttering the abstract syntax with the categories that expresses associativity and precedence by the convention that indexed categories are variants of the same category. The abstract syntax of our grammar, generated by the Haskell back end, is the following, index-free, data types.

```
newtype Ident = Ident String

data Exp =
   Lambda Ident Exp
 | App Exp Exp
 | Id Ident
```

Given that our grammar is in the file `lambda.cf`, we can now compile it with BNFC. Here we use the flag `-m` to also generate a makefile. Since we have not stated which programming language back end to use, BNFC defaults to the Haskell back end. Typing `make` produces an executable binary `Testlambda`, which is used as a test bench for our grammar.

```
$ bnfc -m lambda.cf
...
$ make
...
```

If we try parsing the Lambda calculus expression

```
(\x -> x x) (\y -> y y)
```

with `Testlambda`, then we are successful, and the program prints the raw abstract syntax of the expression together with its pretty-printed version.

```
$ echo "(\\x -> x x) (\\y -> y y)" | ./Testlambda

Parse Successful!

[Abstract Syntax]
```

```
App (Lambda (Ident "x") (App (Id (Ident "x")) (Id (Ident "x"))))
    (Lambda (Ident "y") (App (Id (Ident "y")) (Id (Ident "y"))))

[Linearized tree]

(\ x -> x x)(\ y -> y y)
```

The approach taken in BNFC is to generate code for a set of tools, e.g. parser and lexer generators. This approach has a couple of advantages — first it avoids redoing work already done, such as implementing parsing algorithms, and therefore saves a lot of work. However, there is yet another, more important, motivation: that of maintenance. Bug fixes and development come for free by using existing tools.

The multilinguality of BNFC provides a convenient way of data transfer between different programming languages. A language that describes the data is created in BNFC. The transfer takes place by pretty-printing the data from a program written in one language, which is later parsed by a program written in another language.

1.4.2 Requirements of BNF Converter

Some requirements are put on the languages implemented in BNFC, to be able to generate all the mentioned modules. The requirements follow the guidelines of any modern compiler construction book, and most of today's programming languages have at least a well-defined subset that fulfills all requirements. The requirements are:

The modules in the front end are sequentialized. This means that every task, such as lexing or parsing, is performed as a separate step — the result of one process is fed into the next step in the process.

The lexical structure can be described by a regular expression. This may seem trivially fulfilled since the engine in a lexer is a finite state automaton, equivalent to a regular expression. However, it is possible to execute arbitrary code in the semantic action of a lexer rule, i.e. non-regular phenomena such as nested comments can be defined with a lexer tool.

The only semantic action allowed in the parser is the building of abstract syntax trees. Even though this is highly recommended in the literature, some front end implementations also perform additional tasks in the parser, such as type checking.

White spaces carry no meaning, except for layout information. That is, white spaces are safely removed in the tokenization process. The layout

information is handled in a preprocessing step, where the block structures described by the layout information are made explicit with curly braces.

The grammar must, except for the Haskell generation with GLR support, be LALR(1) parsable. This requirement is not something inherent in BNFC, but rather in the tools that are used to produce the parser. For example, the Happy parser generator has been generalized with the Tomita algorithm [24, 15] so that it will produce a forest of parse trees, instead of a single tree.

1.5 Introduction to Functional Morphology

Functional Morphology (FM) is a computational tool for defining a *lexical resource*, i.e. a systematic description of the words in a language. The aim of FM has been to create a language-independent tool, in the sense that no unnecessary restrictions should be built in that limit the tool to a particular type of language. Naturally, language independence cannot be fully achieved, since a tool like this implements a model that necessarily fits some languages better than others. The lexical resource model of FM is *word-and-paradigm*, a concept coined by Hockett [8], where a *paradigm* can be understood as an abstraction of an inflection table. A grammar book for Latin illustrates this model, since it is organized around inflection tables. This is true for many grammar books, but Latin is the school example.

The inflection tables in a grammar define not only the inflection of the words in the tables but whole classes of words, i.e. they are paradigms. It is the job of the reader of the grammar to extract the inflection pattern from the table. This process must be made explicit in a computer, and this is done in FM through a function that encodes the paradigm.

A *lexicon* in FM consists of words annotated with paradigm names. An paradigm name carries information about the word, for example how it inflects and its part of speech. The inflection engine assigns the paradigm name's meaning by computing words into *inflection tables*. As a concrete example, consider the Latin word *rosa* (Eng. 'rose'). It is a feminine noun in the first declension. This fact can be encoded by annotating the word with an name: `nn1_f_puella rosa`. Even though the name is just an name, it uses a mnemonic encoding: `nn1` stands for 'noun in the first declension', `f` stands for its gender, feminine, and `puella` is the paradigm's example word form.

The inflection engine for a Latin morphology translates the word with its paradigm name into an inflection table. The information generated is exemplified in Fig. 1.1. It includes, besides the word forms, the inflectional

nn1_f_puella rosa ⇒

Dictionary form		rosa
Word class		nn
Gender		f
Number	**Case**	**Word form**
sg	nom	rosa
sg	voc	rosa
sg	acc	rosam
sg	gen	rosae
sg	dat	rosae
sg	abl	rosa
pl	nom	rosae
pl	voc	rosae
pl	acc	rosas
pl	gen	rosarum
pl	dat	rosis
pl	abl	rosis

Figure 1.1: The inflection table of *rosa*.

parameters, number and case, the inherent parameter, feminine gender, the part of speech, noun, and the citation form, rosa.

1.6 Lexical Resources in FM

The definition of a paradigm in FM, i.e. the addition of a paradigm to the inflection engine, requires a predefined *type system*, describing the parameters of the paradigm, an *inflection function* and a *paradigm name*.

An inflection function will correctly generate the inflection table for a class of words, and it is the lexicographer's task to annotate the words of the lexicon with the correct paradigm name. An inflection function assumes that one or more word forms of a particular shape is supplied — typically the citation forms in an ordinary dictionary.

When defining paradigm functions there are, at least, three considerations: the *productivity* of the paradigm function, i.e. that the set of words it covers is as large as possible; that it *requires as little information as possible*, since a paradigm function that requires almost all word forms would be of little use; and that it is *predictable*, to simplify the task of the lexicographer.

1.7 Why Functional Morphology?

1.7.1 FM is an Embedded Domain-specific Language

FM is an embedded domain-specific language in the programming language Haskell, a functional programming language. This means that higher-order functions, types and abstract data types are available, which allows the definition of the lexical resource on a higher level, compared to, for example, the programming language of XFST [2], which is untyped and its only support for abstraction is the possibility to give names to expressions.

1.7.2 FM is Typed

FM uses algebraic data types for defining its type system, i.e. the parameters of the target language. Besides being a convenient way of describing parameters in an exact and concise way, it also gives guarantees: we know that we cannot create a spurious parameter configuration, since that would give rise to a type error; and we also get a check that we have defined all cases, since inflection tables are created by enumerating all parameter configurations and a missing case will be captured at compile time.

1.7.3 The Runtime System of FM

The runtime system of FM provides support for analysis of input text and synthesis of word forms into inflection tables. Furthermore, the inflection engine of FM is usable in batch mode, i.e. other programs can call FM with paradigm names together with citation forms, and it will compute the inflection tables and output them on standard output.

1.7.4 FM is a Compiler

FM is a compiler, acknowledging that one who are ready to invest time in developing a lexical resource wants to maximize its usefulness. One way of achieving this is to generate the resource in as many formats as possible. A lexical resource in FM can be translated to many other formats: fullform, LexC, XFST, SQL, GF (gives a direct connection to syntax) etc.

1.7.5 FM is Open Source

FM is open source software available under GPL. It allows rewriting of those parts of FM that do not fit the present purposes for one reason or another,

or it could be that there is a need for adding a new resource translation format.

1.7.6 FM is Reusable

It is easy to reuse parts of an FM implementation, in particular the type system, for closely related languages, especially if the implementation is created with this in mind. The module system of Haskell together with the clearly separated parts of an FM implementation are the main reasons why reuse is a simple task.

1.7.7 FM supports Compound Analysis

The definition of a lexical resource in FM may include a specification of compound analysis, which is supported by the runtime system of FM. Compound analysis is performed by associating all word forms[5] with an integer, and by a compound function that describes which sequences of integers are valid. The compound analysis divides an input word form into all possible sequences of word forms and filters out all that are not valid according to the compound function.

1.8 FM and the Languages of the World

Morphologies can be classified in one of the following categories [3, 6]. Note that the classification is in no way clear cut — a language may involve many different types, so the classification is rather the main tendency of a language.

- **Analytic languages**
  Example: Chinese
- **Synthetic languages**
  - **Agglutinative languages**
    Example: Turkish
  - **Flexive languages**
    Example: Latin

[5] Actually, it is the objects of the dictionary types, i.e. the inflectional parameters, which are associated with integers, a subtle difference discussed later on.

  - **Non-linear (templatic) languages**
    Example: Arabic

- **Polysynthetic languages**
  Example: Inuktitut

Analytic languages have essentially no morphology, and because of that it may not be very interesting to define a lexicon in FM. However, one potential benefit of defining an analytical language in FM is that the lexicon can be generated in many formats. Many of the analytical languages have a non-separating writing system, a property discussed in Sec. 1.9.5.

Agglutinative and *flexive* languages are examples of *concatenative* morphology, where the inflectional affixes are glued to the word in question. The difference between the two categories is: in agglutinative languages, a grammatical function is typically realized by a specific affix; in flexive languages, on the other hand, an affix can express many functions. An example of agglutination is the Turkish word *el-ler-in* (Eng. 'hand-PL-GEN'), where every morpheme expresses *one* function. An example of flexivity is the Latin word *amo* (Eng. 'I love'), where the suffix '-o' expresses many functions, i.e. that the word is in indicative, first person, singular, present active form.

Non-linear languages, or templatic languages, are languages where the affixes are not necessary a consecutive sequence of characters. E.g. in Modern Hebrew [3], the consonant template *g-d-r* (Eng. 'enclose') can be inflected as: *gadar* (Eng. 'he enclosed') or *gdor* (Eng. 'enclose it!'). FM has a rich type system, where the templates can be represented with a more complex type than a simple string. A possible choice for Modern Hebrew would be to represent the consonant templates with 3-tuples. An example of this for the language Temiar is given in Sec. 1.9.2.

Polysynthetic languages are highly concatenative languages where a word can correspond to a complete sentence in an other, more analytic, language. The language typically mentioned is Inuktitut, where, e.g. the two word sentence [6] *Paasinngilluinnarpara ilaajumasutit* means *I didn't understand at all that you wanted to come along.* It is mainly for the morphologies of the synthetic languages that FM is intended for, where the language's morphology is naturally expressed in terms of inflection tables. Polysynthetic languages are more problematic since the inflection tables are infinite. Polysynthetic languages will be discussed further in Sec. 1.9.3.

1.9 Morphological Phenomena in FM

Since a paradigm function is an ordinary function in the programming language Haskell, all morphological phenomena that are computable are expressible with it. However, it may still be interesting to study some of the trickier, non-concatenative, cases to see how they can be handled in FM.

1.9.1 Vowel harmony

Vowel harmony appears in languages such as Turkish and Finnish, a phenomenon where the stem vowel affects the realization of the vowels in the suffixes. There are also other kinds of vowel harmony, but they will not be discussed here.

Karlsson [11] describes the vowel harmony of Finnish as follows: if the stem contains any of the back vowels u, o, a, then the suffixes should only contain back vowels, otherwise the suffixes should only have front vowels. The translation between back and front vowels can be described with three vowel pairs: (a,ä), (o,ö) and (u,y). A consequence of this is that all suffixes containing back vowels also have a corresponding version containing front vowels and vice versa.

We have two choices for how to treat vowel harmony. We could treat it paradigmatically, i.e. that vowel harmony is a part of the specification of the paradigms. Or, since vowel harmony is rather regular, we can write an auxiliary function for it. The latter is more natural and productive. The function is given below, where we simplify the function to only use front vowels in the suffixes. This simplification follows the convention of Finnish dictionaries. Example of usage is given in the partial definition of the function `decl1`.

```
vowel_harmony :: String -> String -> String
vowel_harmony stem suffix
  | or [elem c "aou" | c <- stem] = stem ++ suffix
  | otherwise                     = stem ++ map vh suffix
 where vh c  = maybe c id (lookup c (zip "aou" "ääy"))

decl1 stem nf =
   case nf of
    ...  -> vowel_harmony stem "lla"
    ...
```

Compounds in Finnish require something more, since it is only the last stem in the compound that dictates the vowel harmony. An example is

the word *kitaristi*, which is either the word *kitaristi* (Eng. 'guitarist') or the compound word *kita-risti* (Eng. 'mouth cross'), where the single word would get the partitive ending *-a* and the compound *-ä*. There are a number of possible solutions for handling the compounds in Finnish: we could remove them from the lexicon and treat compounds with the compound analysis; we could require that the lexicographer marks the compound boundaries in the lexicon; we could supply the paradigm functions with information to specify the vowel harmony.

1.9.2 Reduplication

Reduplication occurs when the stem, or part of it, of a word is in some way repeated. A more complicated instance of reduplication is *internal reduplication* [5] where parts of the stem are repeated in the stem itself. Verbal inflection with internal reduplication in the Austroasiatic language Temiar is presented here with the example word *kɔ̄w* (Eng. 'to call'). The table is taken from Broselow and MacCarthy [5].

	Active	**Causative**
Perfective	kɔ̄w	trkɔ̄w
Simulfactive	kakɔ̄w	trakɔ̄w
Continuative	kwkɔ̄w	trwkɔ̄w

The interesting case is Active Continuative: the last consonant is reduplicated at second position, and the first consonant is duplicated in the third position.

The implementation of this paradigm is given below, where the consonant pattern, including the medial vowel, is represented with a 3-tuple. The result of applying `biconsonantal` with a pattern is a function in the type `Verb`, a function type from a parameter configuration to a string. The inflection table is produced by generating all parameter configurations and applying them consecutively to the function.

```
biconsonantal :: (String, String, String) -> Verb
biconsonantal (k,o,w) (VF v p) =
    case v of
      Active ->
       case p of
        Perfective   -> concat [k,o,w]
        Simulfactive -> concat [k,"a",k,o,w]
        Continuative -> concat [k,w,k,o,w]
      Causative ->
```

```
case p of
 Perfective   -> concat ["tr",k,o,w]
 Simulfactive -> concat ["tra",k,o,w]
 Continuative -> concat ["tr",w,k,o,w]
```

Temiar's verbal inflection only reduplicates consonants, but it is worth noting that the function would look exactly the same for syllables.

1.9.3 Polysynthetic Languages

Polysynthetic languages are interesting since it is no longer natural to talk about words and inflection tables. Instead we have a lexicon of allomorphs and a description of how allomorphs can be assembled.

Inuktitut is a nice example since not only is the language polysynthetic, it also has a rich morphophonology. M. Mallon [14] describes the morphophonology in Inuktitut. The exact details are not interesting in this setting, but rather that the morphophonology is quite complicated, and more importantly, that it is realized in the spelling of words.

- **Consonant deleters** A deleter deletes the consonants in the affix before. Example:
  umiaq+ksaq → *umiaksaq*

- **Vowel deleters** As a consonant deleter, with the additional rule that three vowels may not appear in a row. In that case, an epenthetic element is be inserted. Example ('ra' is the epenthetic element):
  qallunaaq+aluk → *qallunaa**ra**aluk*

- **Regressive assimilation**

  - **Neutral** If a consonant is voiceless after an affix boundary, then the consonant before the affix boundary is left unaffected.
  - **Nasalization** If the consonant is nasalized after an affix boundary, then the consonant before the affix boundary becomes nasalized. Example, where 'r' is a voiced nasalized sound:
    umiaq+mut → *umia**r**mut*.
  - **Voiced** If the consonant after an affix boundary is voiced, then the consonant before the affix boundary will become voiced. Example, where 'r' is a voiced, but here not nasalized, sound:
    niuviq+vik → *niuvi**r**vik*

Mallon presents two additional morphophonological phenomena, namely *consonant variation* and *uvular variation*. They will, however, not be discussed here, since they are a bit more complicated and contribute nothing further to the discussion.

How do we deal with the rich morphophonology of Inuktitut in FM, could it be solved with compound analysis? Huet [9] deals with external sandhi in Sanskrit, a similar problem, through rewrite rules of the form $u|v \rightarrow w$, where u,v and w are strings. Huet reports that since these rules are so concrete (note that u,v and w are strings, not regular expressions), they actually amount to as many as 2790 rules[6]. These rules are later compiled into a segmentation automation.

To use the compound analysis of FM we need to take into account that the morphemes not only affect their environment, but are also affected by it. Both of these properties need to be encoded in the compound parameter. The current compound parameters of FM are encoded with integers. However, a small and natural extension would be to allow arbitrary types for describing compounding. So, instead of having to encode the properties with a complicated integer encoding, we could use a pair of objects of an algebraic data type that reflects their meaning. We can define a type `P`, which is used for both stating how a morpheme affects its environment and how it is affected by it, e.g. the pair `(CD,CD)` associated to a morpheme form states that it is a consonant deleter which, in turn, has been affected by a consonant deleter.

```
data P = CD  | -- consonant deleter
         VD  | -- vowel deleter
         NA  | -- nasalizer
         V   | -- voicer
         NEU | -- neutral
         ...

type CompP = (Param,Param)
```

The compounding model of FM associates compound parameters to the inflectional parameters, i.e. the objects of the dictionary types, not to the actual word forms. That is, if a word form has a variant, there is no way to assign a different compound parameter to that variant. This choice may be unsuitable in the case of rich morphophonology, since it requires that the morphophonology is represented by inflectional types.

[6]The rules were generated from a more abstract specification.

1.9.4 Clitics

Clitic elements are often problematic to deal with. For example, some clitics are unbounded with respect to the syntactic categories they attach to. For example the element *ne* in Latin, which can be attached to any word to express a questioning of that particular word. Clitics cannot be handled in the same way as derived words, i.e. just adding cliticized words to the lexicon, since a clitic such as *ne* would double the size of the lexicon, and more, clitics are not only applied to words but sometimes to phrases, as the *'s* in English: *the men who knit's house*.

Clitics can be handled cleanly with compound analysis in FM. The clitics are represented in the lexicon as word forms, which may only appear in compounds. There is no way to decide that a clitic is an element for a whole phrase in FM, which is unsurprising since FM does not include any syntactic analysis.

1.9.5 Non-Separating Writing Systems

Non-separating writing systems do not use whitespaces to separate words. Examples of languages that uses such systems are Chinese, Korean, Japanese and Thai. These systems require a word segmentation phase, in a sense adding whitespaces to the input, before a morphological analysis can be performed.

What simplifies matters, from a computational point of view, is that some non-separating writing systems, such as Chinese, use typographical symbols such as full stop and comma to delimit sentences and phrases, which greatly simplifies the segmentation task, since it is enough to perform segmentation on the sentence level.

Is segmentation a task that could be solved by compound analysis in FM? The compound analysis can be used to retrieve all possible segmentations, which is the first step, but to select the correct segmentation then we are, in general, required to go beyond the lexicon and the text at hand, i.e. we need some kind of semantic knowledge to retrieve the correct result. Even though it would be possible to add semantic annotations in FM, an FM module would rather be a component in a larger system instead of being able to deal with semantics itself.

1.9.6 Portmanteau and Fixed Phrases

A *Portmanteau* expresses more than one word, and at the same time, the categories expressed exist as independent words. E.g. the French *du* that

corresponds to *de* (of) and *le* (the). Conceptually, it is not clear how Portmanteaus should be treated, i.e. represented in the lexicon. However, they are quite rare even in languages that have them, so maybe we can accept that they appear in the lexicon? A natural treatment of a portmanteau in the morphological analysis would be as a compound.

Fixed phrases are phrases that appear in a dictionary, such as idioms or stagnated phrases containing word forms that do not appear in modern use outside the phrase. They are problematic since the degree of fixation might vary — adding totally fixed phrases poses no problem but some fixed phrases are more loose in the sense that some kind of phrases may be inserted in them such as adverbial phrases.

It is not possible to give a final solution for how to deal with fixed phrases, since they lie on the border of morphology and syntax, but they must, at least, be considered carefully when creating a natural language system.

1.10 Lexical Resources in FM

A number of languages have been implemented in FM to date, and we will shortly present some of them.

1.10.1 Latin

A small resource for Latin has been developed by M. Forsberg, to be used as a tutorial language for FM. Latin is a good choice for illustrating FM since: it is the school example of word-and-paradigm; its grammar is well-documented; it is a good example why types is a good idea (see FM:s technical report for details); and finally, it has a clitic *ne*, a question particle, whose implementation is a nice demonstration of FM's compound analysis.

The Latin Morphology is distributed with the Functional Morphology library at the FM homepage.

1.10.2 Modern and Old Swedish

L. Borin and M. Forsberg are working on adding morphological information to a lexical resource named SAL (Swedish Association Lexicon) developed by Lennart Lönngren. SAL is a rather large resource, comprising 71752 entries. The starting point is an FM implementation for Swedish, implemented by M. Forsberg and A. Ranta, but further developed, based on Hellberg's [7] rather complete description of Swedish paradigms. The goal of the project

is twofold: first, to enrich SAL with morphological information, and second, to make a large, unrestricted morphological lexicon available to the public.

L. Borin, R. Johnsson, M. Forsberg have been working on a project for Old Swedish, where the goal is to connect three Old Swedish dictionaries with real text. The connection will be done by a morphological unit that also deal with the spelling variation of the word forms. The dictionaries are Söderwall [22], Söderwall supplement [23] and Schlyter [19], available in electronic form at Språkbanken[7]. These dictionaries are the main authoritative material for old Swedish.

The main challenge of Old Swedish is to cope with the enormous variation in spelling. There are two reasons why there is such a variation: first, because there were no prescriptive standard on how to spell; second, because the time period covered is 300 years and many natural changes of spelling/pronouncing occur during such a long time.

1.10.3 Master Thesis Projects

There have been three Master's thesis projects to date concerned with developing a lexical resource in FM. I. Andersson and T. Söderberg defined a lexical resource for Spanish [1], L. Bogavac a Russian resource [4], and M. Humayoun a resource for Urdu [10]. The work on Urdu has subsequently been published [13, 12].

Our experiences have been good — all students were able to create substantial resources. This has led us to believe that the development and documentation of a lexical resource is in the scope of a Master's thesis.

1.10.4 Other Implementations

O. Smrz has implemented an extension of FM, which he refers to as ElixirFM [20], for Arabic. This work is the main part of his thesis [21].

A. Ranta [17] implemented the first morphology in FM, a nearly complete paradigm system for Italian nouns and verbs.

M. Pellauer [16] has developed a morphology for Estonian.

1.11 Extended Example: Swedish

We will now give some detailed information taken from the FM implementation of modern Swedish morphology. We will focus on nouns since they

[7]Dictionaries accessible at: `http://spraakbanken.gu.se/fsvldb/`

provide a balance between being complex enough to be interesting, but simple enough to allow a full presentation.

1.11.1 Type System

The type system defines the language model. There are three kinds of types: *inflectional*, *inherent* and *dictionary* types. The inflectional types describes the parameters that govern the inflection, the inherent types concern with parameters not involved in the inflection, such as gender or subcategorization information. The dictionary type stands typically for a word class.

The inflection of Swedish nouns depends on three parameters: number, definiteness and case. These are implemented as data type definitions in Haskell:

```
data Number       = Sg    | Pl

data Definiteness = Indef | Def

data Case         = Nom   | Gen
```

Nouns in Swedish also have a *inherent* parameter: `Gender`. Nouns in Swedish **have** a gender, they are not inflected in gender. The `Gender` type has two values, `Uter` and `Neuter`, with their obvious meanings.

```
data Gender = Uter | Neuter
```

The inflectional types are combined in one type, a dictionary type, typically corresponding to a word class. Besides the inflectional types, there is also another constructor, `Comp`, corresponding to the compound word forms.

```
data NounForm = NF Number Definiteness Case | Comp
```

1.11.2 Paradigm System

The paradigm system of nouns in Swedish looks deceptively simple, consisting of five declensions, all distinguished by their plural form. However, if we aim for completeness and exactness then there are many small differences which give rise to many more paradigms. See Hellberg's description of the Swedish paradigm system [7] for details.

A noun paradigm creates a `Noun`, a function from the dictionary type `NounForm` to a `Str`.

```
type Noun = NounForm -> Str
```

A `Noun` is translated into an inflection table by enumerating all values of `NounForm` and applying it to the `Noun`.

A `Str` is not a single word form, but a list of word forms. The abstract type `Str`, defined in the FM library API, is given below, together with three of its methods: `mkStr` creates a `Str` from a word form; `strings` creates a `Str` from a list of word forms; and `nonExist` is a `Str` describing a missing word form.

```
type Str = [String]

mkStr :: String -> Str
mkStr = (:[])

strings :: [String] -> Str
strings = id

nonExist :: Str
nonExist = []
```

The use of `Str` enables us to describe missing forms, such as in the Swedish word *lat* (Eng. 'lazy'), which is missing its neuter form, and free variants, such as *köps* and *köpes* (Eng. passive of 'buy').

Swedish has two cases, nominative and genitive, which is uniformly defined for all noun paradigms, i.e. the case inflection is not part of the paradigm specification. Hence, we can define it as an auxiliary function. The operator `+?`, which is part of the FM library, is a conditional concatenation that concatenate the input string with a string `s` if and only if it does not already end with the string `s`.

```
mkCase :: Case → String → String
mkCase c w = case c of
  Nom → w
  Gen → w +? "s"
```

The next step is to define a convenient worst-case function. In the worst case, since we could abstract out the case inflection, we need to supply five word forms. The last word form is a `Str`, corresponding to the compound word form, since it is not all of the paradigms that have a special compound word form.

```
mkNoun :: String → String → String → String → Str → Noun
mkNoun apa apan apor aporna ap f = case f of
  NounForm n s c -> mkStr $ mkCase c $ case (n,s) of
```

```
    (Sg,Indef) → apa
    (Sg,Def)   → apan
    (Pl,Indef) → apor
    (Pl,Def)   → aporna
  Comp       → ap
```

We can now define a paradigm function for the declensions of Swedish by using our worst-case function. The function `init` takes all characters except the last one, the function `last` takes the last character and the operator `+?` avoid duplication of adjacent characters.

```
decl1 :: String -> Noun
decl1 apa = mkNoun apa (ap ++ "an") (ap++"or") (ap++"orna") (mkStr ap)
  where ap   = init apa

decl2 :: String -> Noun
decl2 pojke = mkNoun pojke pojken (pojk ++ "ar") (pojk ++ "arna")
                     (pojkC)
  where
    pojk = dropEndIfE pojke
    pojkC = if (last pojke == 'e') then mkStr pojk else nonExist
    pojken = pojke +? "en"

decl3 :: String -> Noun
decl3 sak = mkNoun sak (sak ++ "en") (sak ++ "er") (sak++"erna")
                   nonExist

decl4 :: String -> Noun
decl4 jojo = mkNoun jojo (jojo ++ "n") (jojo ++ "r") (jojo++"rna")
                    nonExist

decl5 :: String -> Noun
decl5 rike = mkNoun rike (rike +? "et") (rike ++ "n") (rike ++ "na")
                    nonExist

decl6 :: String -> Noun
decl6 lik = mkNoun lik (lik ++ "et") lik (lik ++ "en")
                   nonExist
```

Many compound forms of the first declension have variants. The variants are word forms from old Swedish still used in modern Swedish. For example, the word *gata* (Eng. 'street') has not only the compound word form *gat* but also *gatu*. We can express this with a higher-order function `variants`, which is part of the FM library. The paradigm function for words like *gata* is defined in terms of `decl1`.

```
decl1gata :: String → Noun
decl1gata gata = (decl1 gata) `variants` [(Comp, gat++"u")]
 where gat = init gata
```

1.11.3 Dictionary Functions

The next step is to create dictionary functions, which translate paradigm functions into dictionary entries. We start with a general function `noun`, which translates a `Noun` into a dictionary entry. The translation is done with the function `entryI`, and the translation is possible since the dictionary type `NounForm` is an instance of the `Dict` class.

It is at this stage the inherent types enter, since they are a property associated to the whole dictionary entry, not a particular word form.

We can now define the dictionary functions for the two paradigm functions we defined before, with the help of `noun`.

```
noun :: Noun → Gender → Entry
noun n g = entryI n [prValue g]

d1 :: String → Entry
d1 s = noun (decl1 s) Uter

d2 :: String → Entry
d2 s = noun (decl2 s) Uter

d3 :: String → Entry
d3 s = noun (decl3 s) Uter

d4 :: String → Entry
d4 s = noun (decl4 s) Uter

d5 :: String → Entry
d5 s = noun (decl5 s) Neuter

d6 :: String → Entry
d6 s = noun (decl6 s) Neuter

d1gata :: String → Entry
d1gata s = noun (decl1gata s) Utr
```

The dictionary functions are assigned paradigm names (details omitted), typically the same as the dictionary function names, which connects the dictionary functions with the external lexicon. The lexicon consists of a listing of words annotated with their paradigm names.

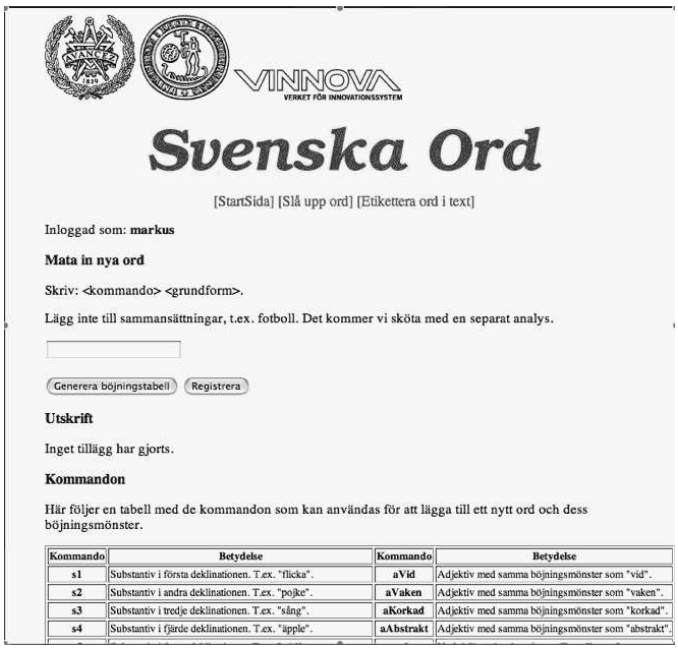
We can now start developing our lexicon, here with only 5 entries. This lexicon is expanded to 43 word forms by the inflection machinery we just defined.

```
d1 flicka
d1 smula
d2 stolpe
d2 al
d6 fik
```

1.12 Introduction to Lexicon Extraction

Given that we have defined an inflection engine, how should we go about building the lexicon? There are usually some trickier cases that are best treated manually, such as irregularly inflected words, but for the regular ones, could we employ some automatic methods?

Our first approach was to create a CGI script that enabled registered users to extend the lexicon. This script was generated from FM, but required some manual work (translation of the written content and the addition of host information). A screen shot of the CGI script is given below.



The idea with the CGI script was to use the Internet community as an automatic method to extend the lexicon. This resulted in only 360 words and the quality was not that great. We realized that no-one was going to do the work for us.

After realizing that we had to do our own spade work, we wrote some scripts that applied a word guessing strategy on the word forms of some text, i.e. to use the inflectional and derivational affixes of word forms to try to identify citation forms tagged with paradigm names. It worked reasonably well, but it was only based on a single word form and it was in no way systematic. The *Extract* tool came as a sudden idea, first tested on the Language Technology course in 2004 and then at a morphology course for computational linguists.

1.13 Swedish Nouns in Extract

The tool Extract is essentially a powerful search tool that uses a combination of regular expressions containing variables and propositional logic to form search templates. Extract takes either unprocessed or preprocessed text data as input together with a file containing lexicon extraction rules. A rule describes how citation forms tagged with identifiers can be identified based on the word forms in the text data. The identifiers encode some linguistic information about the words the citation forms represent, e.g. `regN cat` would indicate that *cat* is inflected as a regular noun in English.

Let us now write a rule for Swedish first declension nouns to extract words to the lexicon of our FM implementation defined in Sec. 1.11. We will use a collection of 94k unique word forms as input data, collected from raw text data.

The rule can be read as follows: if a substring, denoted by `stick`, can be found for one of the singular word forms `stick+"a"`, `stick+"an"`, `stick+"as"`, `stick+"ans"` and one of the plural word forms `stick+"or"`, `stick+"orna"`, `stick+"ors"`, `stick+"ornas"`, then we output `d1 stick+"a"`. E.g. if the input data contains the word forms *strumpas* and *strumporna*, then the tool would output `d1 strumpa`.

```
rule d1
    =
    stick+"a"
   {
     (stick+"a" | stick+"an" |  stick+"as" | stick+"ans") &
     (stick+"or" | stick+"orna" | stick+"ors" | stick+"ornas")
    } ;
```

This rule contains no reference to the context of the word forms, i.e. the input data is treated as a set. Extract supports the possibility to refer to the context and also to any additional information that word forms been augmented with, through so called Constraint Grammar constructs, but this will not be discussed further here.

When we run Extract with the rule on our data set, we end up with 530 words hypothesized to be nouns in the first declension. An examination of the output reveals a couple of spurious short words appearing in the list, e.g. `d1 bra`, which is a hypothesis based, erroneously, on the word forms *bra* (Eng. 'good') and *bror* (Eng. 'brother'). These false positives are caused by the stem variable `stick` being unconstrained, and to improve the extraction we need to require that the substring associated to `stick`, at least, contains one (stem) vowel.

The stem variable is constrained by associating a regular expression to it. This is done by defining a couple of regular expressions and by a small modification in our rule. The regular expression `Stem` states that the variable `stick` can only be associated to a substring consisting of alphabetic letters with at least one vowel.

```
regexp Consonant = ["bcdfghjklmnpqrstvwxz"] ;
regexp Vowel     = ["aeiouyåäö"] ;
regexp Letter    = Consonant | Vowel ;
regexp Stem      = Letter* Vowel Letter* ;

rule d1 [stick:Stem]
    =
    stick+"a"
   {
     (stick+"a" | stick+"an" |  stick+"as" | stick+"ans") &
     (stick+"or" | stick+"orna" | stick+"ors" | stick+"ornas")
    } ;
```

If we rerun Extract with the new rule, we get 505 words instead, of which 14 are still false positives. The false positives originate mostly from verbs in infinitive: `+"a"` or `+"as"` (passive infinitive). Amusingly, we get the false positive `d1 halvbra`, originating from *halvbra* (Eng. 'half good') and *halvbror* (Eng. 'half brother'), the word forms we wanted to get rid of, now appearing in compound word forms.

Bibliography

[1] I. Andersson and T. Söderberg. Spanish Morphology – implemented in a functional programming language. Master's Thesis, Computational Linguistics, Gothenburg University, 2003.

[2] K. R. Beesley and L. Karttunen. *Finite State Morphology*. CSLI Publications, Stanford University, United States, 2003.

[3] B. Bickel and J. Nichols. Inflectional morphology. In T. Shopen, editor, *Complex Constructions*, volume 2 of *Language Typology and Syntactic Description*. 2 edition, 2007.

[4] L. Bogavac. Functional Morphology for Russian. Master's Thesis, Department of Computing Science, Chalmers University of Technology, 2004.

[5] E. Broselow and J. McCarthy. A theory of internal reduplication. *The Linguistic Review*, 3:25–98, 1983.

[6] M. Haspelmath. *Understanding Morphology*. Oxford University Press Inc., Upper Saddle River, New Jersey 07458, 2002.

[7] S. Hellberg. *The Morphology of Present-Day Swedish*. Almqvist & Wiksell International, Stockholm, Sweden, 1978.

[8] C. F. Hockett. Two models of grammatical description. *Word*, 10:210–234, 1954.

[9] G. Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional Programming*, 15,4:573–614, 2005. `http://yquem.inria.fr/~huet/PUBLIC/tagger.pdf`.

[10] M. Humayoun. Urdu Language Morphology in Functional Morphology Toolkit. Master's Thesis, Department of Computing Science, Chalmers University of Technology, 2006.

[11] F. Karlsson. *Finsk Grammatik*. Suomalaisen Kirjallisuuden Seura, 2005.

[12] M. Humayoun, H. Hammarström, and A. Ranta. Implementing Urdu Grammar as Open Source Software - extented abstract. *Conference on Language and Technology, University of Peshawar, Pakistan*, 2007.

[13] 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, LSA 2007 Linguistic Institute, Stanford University*, 2007.

[14] M. Mallon. Inuktitut Linguistics for Technocrats, 2000. `http://www.inuktitutcomputing.ca/Technocrats`, accessed 1 Jan 2007.

[15] P. Callaghan. Ambiguous Parsing with Happy, 2007. Under Consideration for Publication in J. Functional Programming. Available at: `www.dur.ac.uk/p.c.callaghan/happy-glr/callaghan-glr.ps.gz`.

[16] M. Pellauer. A Functional Morphology for Estonian. Term Paper, MIT, 2005.

[17] A. Ranta. 1+n representations of Italian morphology. *Essays dedicated to Jan von Plato on the occasion of his 50th birthday*, `http://www.valt.helsinki.fi/kfil/jvp50.htm`, 2001.

[18] A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

[19] C. Schlyter. *Ordbok till Samlingen af Sweriges Gamla Lagar. (Saml. af Sweriges Gamla Lagar 13.)*. Lund, Sweden, 1887.

[20] O. Smrz. Elixirfm - implementation of functional arabic morphology. *ACL 2007 Workshop on Computational Approaches to Semitic Languages*, 2007.

[21] O. Smrz. *Functional Arabic Morphology. Formal System and Implementation.* PhD thesis, Charles University in Prague, 2007.

[22] K. Söderwall. *Ordbok Öfver svenska medeltids-språket. Vol I-III.* Lund, Sweden, 1884-1918.

[23] K. Söderwall. *Ordbok Öfver svenska medeltids-språket. Supplement. Vol IV—V.* Lund, Sweden, 1953—1973.

[24] M. Tomita. Efficient Parsing of Natural Language. *Kluwer Academic Press*, 1986.

Part I

BNF Converter

Chapter 2

Labelled BNF: A High-Level Formalism for Defining Well-Behaved Programming Languages

Markus Forsberg and Aarne Ranta
Department of Computing Science
Chalmers University of Technology and the University of Göteborg
SE-412 96 Göteborg, Sweden
{*markus,aarne*}*@cs.chalmers.se*

Paper published:
NWPT'03, Proceedings of the Estonian Academy of Sciences: Physics and Mathematics, Special issue on programming theory edited by J. Vain and T. Uustalu, 2003, volume 52, p. 356–377

abstract

This paper introduces the grammar formalism *Labelled BNF* (LBNF), and the compiler construction tool *BNF Converter*. Given a grammar written in LBNF, the BNF Converter produces a complete compiler front end (up to, but excluding, type checking), i.e. a lexer, a parser, and an abstract syntax definition. Moreover, it produces a pretty-printer and a language specification in LaTeX, as well as a template file for the compiler back end.

A language specification in LBNF is completely declarative and therefore portable. It reduces dramatically the effort of implementing a language. The price to pay is that the language must be "well-behaved", i.e. that its lexical structure must be describable by a regular expression and its syntax by a context-free grammar.

Keywords

compiler construction, parser generator, grammar, labelled BNF, abstract syntax, pretty printer, document automation

2.1 Introduction

This paper defends an old idea: a programming language is defined by a BNF grammar [12]. This idea is usually not followed for two reasons. One reason is that a language may require more powerful methods (consider, for example, languages with layout rules). The other reason is that, when parsing, one wants to do other things already (such as type checking etc). Hence the idea of extending pure BNF with semantic actions, written in a general-purpose programming language. However, such actions destroy declarativity and portability. To describe the language, it becomes necessary to write a separate document, since the BNF no longer defines the language. Also the problem of synchronization arises: how to guarantee that the different modules—the lexer, the parser, and the document, etc.—describe the same language and that they fit together?

The idea in LBNF is to use BNF, with construction of syntax trees as the only semantic action. This gives an unique source for all language-related modules, and it also solves the problem of synchronization. Thereby it dramatically reduces the effort of implementing a new language. Generating syntax trees instead of using more complex semantic actions is a natural phase of *multi-phase compilation*, which is recommended by most modern-day text books about compiler construction (e.g. Appel [1]). BNF grammars are an ingredient of all modern compilers. When designing LBNF, we tried to keep it so simple and intuitive that it can be learnt in a few minutes by anyone who knows ordinary BNF.

Of course, the are some drawbacks with our approach. Not all languages can be completely defined, although surprisingly many can (see Section 2.5.1). Another drawback is that the modules generated are not quite as good as handwritten. But this is a general problem when generating code

instead of handwriting it: a problem shared by all compilers, including the standard parser and lexer generation tools.

To use LBNF descriptions as implementations, we have built the *BNF Converter*[4]. Given an input LBNF grammar, the BNF Converter produces a lexer, a parser, and an abstract syntax definition. Moreover, it produces a pretty-printer and a language specification in LaTeX. Since all this is generated from a *single source*, we can be sure that the documentation corresponds to the actual language, and that the lexer, parser and abstract syntax fit seamlessly together.

The BNF Converter is written in the functional programming language Haskell[14], and its target languages are presently Haskell, the associated compiler tools Happy[10] and Alex[2], and LaTeX. Happy is a parser generator tool, similar to YACC[7], which from a BNF-like description builds an LALR(1) parser. Alex is a lexer generator tool, similar to Lex[9], which converts a regular expression into a finite-state automaton. Over the years, Haskell and these tools have proven to be excellent devices for compiler construction, to a large extent because of Haskell's algebraic data types and a convenient method of syntax-directed translation via pattern matching; yet they do not quite remove the need for repetitive and low-level coding. The BNF Converter can be seen as a high-level front end to these tools. However, due to its declarative nature, LBNF does not crucially depend on the target language, and it is therefore possible to redirect the BNF Converter as a front end to another set of compiler tools. This has in fact recently been done for Java, CUP [6], and JLex [3][1]. The only essential difference between Haskell/Happy/Alex and Java/CUP/JLex or C/YACC/Lex is the target language included in the parser and lexer description.

2.2 The LBNF Grammar Formalism

As the first example of LBNF, consider a triple of rules defining addition expressions with "1":

```
EPlus. Exp ::= Exp "+" Num ;
ENum.  Exp ::= Num ;
NOne.  Num ::= "1" ;
```

Apart from the *labels*, `EPlus`, `ENum`, and `NOne`, the rules are ordinary BNF rules, with terminal symbols enclosed in double quotes and nonterminals written without quotes. The labels serve as *constructors* for syntax trees.

[1] Work by Michael Pellauer at Chalmers

From an LBNF grammar, the BNF Converter extracts an *abstract syntax* and a *concrete syntax*. The abstract syntax is implemented, in Haskell, as a system of data type definitions

```
data Exp = EPlus Exp Exp | ENum Num
data Num = NOne
```

(For other languages, including C and Java, an equivalent representation can be given in the same way as in the Zephyr abstract syntax specification tool [16]). The concrete syntax is implemented by the lexer, parser and pretty-printer algorithms, which are defined in other generated program modules.

2.2.1 LBNF in a nutshell

Briefly, an LBNF grammar is a BNF grammar where every rule is given a label. The label is used for constructing a syntax tree whose subtrees are given by the nonterminals of the rule, in the same order.

More formally, an LBNF grammar consists of a collection of rules, which have the following form (expressed by a regular expression; Appendix A gives a complete BNF definition of the notation):

> Ident "`.`" Ident "`::=`" (Ident | String)* "`;`" ;

The first identifier is the *rule label*, followed by the *value category*. On the right-hand side of the production arrow (`::=`) is the list of production items. An item is either a quoted string (*terminal*) or a category symbol (*non-terminal*). A rule whose value category is C is also called a *production* for C.

Identifiers, that is, rule names and category symbols, can be chosen *ad libitum*, with the restrictions imposed by the target language. To satisfy Haskell, and C and Java as well, the following rule is imposed

> An identifier is a nonempty sequence of letters, starting with a capital letter.

LBNF is clearly sufficient for defining any context-free language. However, the abstract syntax that it generates may often become too detailed. Without destroying the declarative nature and the simplicity of LBNF, we have added to it four *ad hoc* conventions, which are described in the following subsection.

2.2.2 LBNF conventions

Predefined basic types

The first convention are predefined basic types. Basic types, such as integer and character, can of course be defined in a labelled BNF, for example:

```
Char_a. Char ::= "a" ;
Char_b. Char ::= "b" ;
```

This is, however, cumbersome and inefficient. Instead, we have decided to extend our formalism with predefined basic types, and represent their grammar as a part of lexical structure. These types are the following, as defined by LBNF regular expressions (see 2.3.3 for the regular expression syntax):

`Integer` of integers, defined
`digit+`

`Double` of floating point numbers, defined
`digit+ '.' digit+ ('e' '-'? digit+)?`

`Char` of characters (in single quotes), defined
`'\'' ((char - ["'\\"]) | ('\\' ["'\\nt"])) '\''`

`String` of strings (in double quotes), defined
`'"' ((char - ["\"\\"]) | ('\\' ["\"\\nt"]))* '"'`

`Ident` of identifiers, defined
`letter (letter | digit | '_' | '\'')*`

In the abstract syntax, these types are represented as corresponding types. In Haskell, we also need to define a new type for Ident:

```
newtype Ident = Ident String
```

For example, the LBNF rules

```
EVar. Exp ::= Ident ;
EInt. Exp ::= Integer ;
EStr. Exp ::= String ;
```

generate the abstract syntax

```
data Exp = EVar Ident | EInt Integer | EStr String
```

where `Integer` and `String` have their standard Haskell meanings. The lexer only produces the high-precision variants of integers and floats; authors of applications can truncate these numbers later if they want to have low precision instead.

Predefined categories may not have explicit productions in the grammar, since this would violate their predefined meanings.

Semantic dummies

Sometimes the concrete syntax of a language includes rules that make no semantic difference. An example is a BNF rule making the parser accept extra semicolons after statements:

```
Stm ::= Stm ";" ;
```

As this rule is semantically dummy, we do not want to represent it by a constructors in the abstract syntax. Instead, we introduce the following convention:

> A rule label can be an underscore _, which does not add anything to the syntax tree.

Thus we can write the following rule in LBNF:

```
_ . Stm ::= Stm ";" ;
```

Underscores are of course only meaningful as replacements of one-argument constructors where the value type is the same as the argument type. Semantic dummies leave no trace in the pretty-printer. Thus, for instance, the pretty-printer "normalizes away" extra semicolons.

Precedence levels

A common idiom in (ordinary) BNF is to use indexed variants of categories to express precedence levels:

```
Exp3 ::= Integer ;
Exp2 ::= Exp2 "*" Exp3 ;
Exp  ::= Exp  "+" Exp2 ;
Exp  ::= Exp2 ;
Exp2 ::= Exp3 ;
Exp3 ::= "(" Exp ")" ;
```

The precedence level regulates the order of parsing, including associativity. Parentheses lift an expression of any level to the highest level.

A straightforward labelling of the above rules creates a grammar that does have the desired recognition behavior, as the abstract syntax is cluttered with type distinctions (between `Exp`, `Exp2`, and `Exp3`) and constructors (from the last three rules) with no semantic content. The BNF Converter solution is to distinguish among category symbols those that are just indexed variants of each other:

> A category symbol can end with an integer index (i.e. a sequence of digits), and is then treated as a type synonym of the corresponding non-indexed symbol.

Thus `Exp2` and `Exp3` are indexed variants of `Exp`.

Transitions between indexed variants are semantically dummy, and we do not want to represent them by constructors in the abstract syntax. To do this, we extend the use of underscores to indexed variants. The example grammar above can now be labelled as follows:

```
EInt.   Exp3 ::= Integer ;
ETimes. Exp2 ::= Exp2 "*" Exp3 ;
EPlus.  Exp  ::= Exp  "+" Exp2 ;
_.      Exp  ::= Exp2 ;
_.      Exp2 ::= Exp3 ;
_.      Exp3 ::= "(" Exp ")" ;
```

Thus the data type of expressions becomes simply

```
data Exp = EInt Integer | ETimes Exp Exp | EPlus Exp Exp
```

and the syntax tree for `2*(3+1)` is

```
ETimes (EInt 2) (EPlus (EInt 3) (EInt 1))
```

Indexed categories *can* be used for other purposes than precedence, since the only thing we can formally check is the type skeleton (see the section 2.2.3). The parser does not need to know that the indices mean precedence, but only that indexed variants have values of the same type. The pretty-printer, however, assumes that indexed categories are used for precedence, and may produce strange results if they are used in some other way.

Polymorphic lists

It is easy to define monomorphic list types in LBNF:

```
NilDef.  ListDef ::= ;
ConsDef. ListDef ::= Def ";" ListDef ;
```

However, compiler writers in languages like Haskell may want to use predefined polymorphic lists, because of the language support for these constructs. LBNF permits the use of Haskell's list constructors as labels, and list brackets in category names:

```
[].  [Def] ::= ;
(:). [Def] ::= Def ";" [Def] ;
```

As the general rule, we have

`[`C`]`, the category of lists of type C,

`[]` and `(:)`, the Nil and Cons rule labels,

`(:[])`, the rule label for one-element lists.

The third rule label is used to place an at-least-one restriction, but also to permit special treatment of one-element lists in the concrete syntax.

In the LaTeX document (for stylistic reasons) and in the Happy file (for syntactic reasons), the category name `[X]` is replaced by `ListX`. In order for this not to cause clashes, `ListX` may not be at the same time used explicitly in the grammar.

The list category constructor can be iterated: `[[X]]`, `[[[X]]]`, etc behave in the expected way.

The list notation can also be seen as a variant of the Kleene star and plus, and hence as an ingredient from Extended BNF.

2.2.3 The type-correctness of LBNF rules

It is customary in parser generators to delegate the checking of certain errors to the target language. For instance, a Happy source file that Happy processes without complaints can still produce a Haskell file that is rejected by Haskell. In the same way, the BNF converter delegates some checking to Happy and Haskell (for instance, the parser conflict check). However, since it is always the easiest for the programmer to understand error messages related to the source, the BNF Converter performs some checks, which are mostly connected with the sanity of the abstract syntax.

The type checker uses a notion of the *category skeleton* of a rule, which is a pair

$$(C, A \ldots B)$$

where C is the unindexed left-hand-side non-terminal and $A \ldots B$ is the sequence of unindexed right-hand-side non-terminals of the rule. In other words, the category skeleton of a rule expresses the abstract-syntax type of the semantic action associated to that rule.

We also need the notions of a *regular category* and a *regular rule label*. Briefly, regular labels and categories are the user-defined ones. More formally, a regular category is none of `[`C`]`,`Integer`, `Double`, `Char`, `String` and `Ident`. A regular rule label is none of `_`, `[]`, `(:)`, and `(:[])`.

The type checking rules are now the following:

> A rule labelled by `_` must have a category skeleton of form (C, C).
>
> A rule labelled by `[]` must have a category skeleton of form $([C],)$.
>
> A rule labelled by `(:)` must have a category skeleton of form $([C], C[C])$.
>
> A rule labelled by `(:[])` must have a category skeleton of form $([C], C)$.
>
> Only regular categories may have productions with regular rule labels.
>
> Every regular category occurring in the grammar must have at least one production with a regular rule label.
>
> All rules with the same regular rule label must have the same category skeleton.

The second-last rule corresponds to the absence of empty data types in Haskell. The last rule could be strengthened so as to require that all regular rule labels be unique: this is needed to guarantee error-free pretty-printing. Violating this strengthened rule currently generates only a warning, not a type error.

2.3 LBNF Pragmas

Even well-behaved languages have features that cannot be expressed naturally in its BNF grammar. To take care of them, while preserving the single-source nature of the BNF Converter, we extend the notation with

what we call *pragmas*. All these pragmas are completely declarative, and the pragmas are also reflected in the documentation.

2.3.1 Comment pragmas

The first pragma tells what kinds of *comments* the language has. Normally we do not want comments to appear in the abstract syntax, but treat them in the lexical analysis. The comment pragma instructs the lexer generator (and the document generator!) to treat certain pieces of text as comments and thus to ignore them (except for their contribution to the position information used in parser error messages).

The simplest solution to the comment problem would be to use some default comments that are hard-coded into the system, e.g. Haskell's comments. But this definition can hardly be stated as a condition for a language to be well-behaved, and we could not even define C or Java or ML then. So we have added a comment pragma, whose regular-expression syntax is

"`comment`" String String? "`;`"

The first string tells how a comment begins. The second, optional, string marks the end of a comment: if it is not given, then the comment expects a newline to end. For instance, to describe the Haskell comment convention, we write the following lines in our LBNF source file:

```
comment "--" ;
comment "{-" "-}" ;
```

Since comments are treated in the lexical analyzer, they must be recognized by a finite state automaton. This excludes the use of nested comments unless defined in the grammar itself. Discarding nested comments is one aspect of what we call well-behaved languages.

The length of comment end markers is restricted to two characters, due to the complexities in the lexer caused by longer end markers.

2.3.2 Internal pragmas

Sometimes we want to include in the abstract syntax structures that are not part of the concrete syntax, and hence not parsable. They can be, for instance, syntax trees that are produced by a type-annotating type checker. Even though they are not parsable, we may want to pretty-print them, for instance, in the type checker's error messages. To define such an internal constructor, we use a pragma

```
"internal" Rule ";"
```

where `Rule` is a normal LBNF rule. For instance,

```
internal EVarT. Exp ::= "(" Ident ":" Type ")";
```

introduces a type-annotated variant of a variable expression.

2.3.3 Token pragmas

The predefined lexical types are sufficient in most cases, but sometimes we would like to have more control over the lexer. This is provided by *token pragmas*. They use regular expressions to define new token types.

If we, for example, want to make a finer distinction for identifiers, a distinction between lower- and upper-case letters, we can introduce two new token types, `UIdent` and `LIdent`, as follows.

```
token UIdent (upper (letter | digit | '\_')*) ;
token LIdent (lower (letter | digit | '\_')*) ;
```

The regular expression syntax of LBNF is specified in the Appendix. The abbreviations with strings in brackets need a word of explanation:

`["abc7%"]` denotes the union of the characters `'a' 'b' 'c' '7' '%'`

`{"abc7%"}` denotes the sequence of the characters `'a' 'b' 'c' '7' '%'`

The atomic expressions `upper`, `lower`, `letter`, and `digit` denote the character classes suggested by their names (letters are isolatin1). The expression `char` matches any character in the 8-bit ASCII range, and the "epsilon" expression `eps` matches the empty string.[2]

2.3.4 Entry point pragmas

The BNF Converter generates, by default, a parser for every category in the grammar. This is unnecessarily rich in most cases, and makes the parser larger than needed. If the size of the parser becomes critical, the *entry points pragma* enables the user to define which of the parsers are actually exported:

```
entrypoints (Ident ",")* Ident ;
```

For instance, the following pragma defines `Stm` and `Exp` to be the only entry points:

```
entrypoints Stm, Exp ;
```

[2] If we want to describe full Java, we must extend the character set to Unicode. This is currently not supported by Alex, however.

2.4 BNF Converter code generation

2.4.1 The files

Given an LBNF source file `Foo.cf`, the BNF Converter generates the following files:

- `AbsFoo.hs`: The abstract syntax (Haskell source file)
- `LexFoo.x`: The lexer (Alex source file)
- `ParFoo.y`: The parser (Happy source file)
- `PrintFoo.hs`: The pretty printer (Haskell source file)
- `SkelFoo.hs`: The case Skeleton (Haskell source file)
- `TestFoo.hs`: A test bench file for the parser and pretty printer (Haskell source file)
- `DocFoo.tex`: The language document (LaTeXsource file)
- `makefile`: A makefile for the lexer, the parser, and the document

In addition to these files, the user needs the Alex runtime file `Alex.hs` and the error monad definition file `ErrM.hs`, both included in the BNF Converter distribution.

2.4.2 Example: `JavaletteLight.cf`

The following LBNF grammar defines a small C-like language, Javalette Light[3].

```
Fun.      Prog     ::= Typ Ident "(" ")" "{" [Stm] "}" ;
SDecl.    Stm      ::= Typ Ident ";"  ;
SAss.     Stm      ::= Ident "=" Exp ";"  ;
SIncr.    Stm      ::= Ident "++" ";"  ;
SWhile.   Stm      ::= "while" "(" Exp ")" "{" [Stm] "}" ;
ELt.      Exp0     ::= Exp1 "<" Exp1 ;
EPlus.    Exp1     ::= Exp1 "+" Exp2 ;
ETimes.   Exp2     ::= Exp2 "*" Exp3 ;
EVar.     Exp3     ::= Ident ;
EInt.     Exp3     ::= Integer ;
```

[3]It is a fragment of the language Javalette used at compiler construction courses at Chalmers University

```
EDouble.  Exp3     ::= Double ;
TInt.     Typ      ::= "int" ;
TDouble.  Typ      ::= "double" ;
[].       [Stm]    ::= ;
(:).      [Stm]    ::= Stm [Stm] ;

-- coercions
_.  Stm      ::= Stm ";" ;
_.  Exp      ::= Exp0 ;
_.  Exp0     ::= Exp1 ;
_.  Exp1     ::= Exp2 ;
_.  Exp2     ::= Exp3 ;
_.  Exp3     ::= "(" Exp ")" ;

-- pragmas
internal ExpT. Exp ::= Typ Exp ;
comment "/*" "*/" ;
comment "//" ;
entrypoints Prog, Stm, Exp ;
```

The abstract syntax `AbsJavaletteLight.hs`

The abstract syntax of Javalette generated by the BNF Converter is essentially what a Haskell programmer would write by hand:

```
data Prog =
   Fun Typ Ident [Stm]
  deriving (Eq,Show)

data Stm =
   SDecl Typ Ident
 | SAss Ident Exp
 | SIncr Ident
 | SWhile Exp [Stm]
  deriving (Eq,Show)

data Exp =
   ELt Exp Exp
 | EPlus Exp Exp
 | ETimes Exp Exp
 | EVar Ident
 | EInt Integer
 | EDouble Double
 | ExpT Typ Exp
  deriving (Eq,Show)
```

```
data Typ =
   TInt
 | TDouble
  deriving (Eq,Show)
```

The lexer `LexJavaletteLight.x`

The lexer file (in Alex) consists mostly of standard rules for literals and identifiers, but has rules added for reserved words and symbols (i.e. terminals occurring in the grammar) and for comments. Here is a fragment with the definitions characteristic of Javalette.

```
{ %s =  ^( | ^) | ^{ | ^} | ^; | ^= | ^+ ^+ | ^< | ^+ | ^*}

"tokens_lx"/"tokens_acts":-
 <>       ::= ^/^/ [.]* ^n
 <>       ::= ^/ ^* ([^u # ^*] | ^* [^u # ^/])* (^*)+ ^/

 <>       ::= ^w+
 <pTSpec> ::=  %s %{ pTSpec p = PT p . TS    %}
 <ident>  ::= ^l ^i*   %{ ident  p = PT p . eitherResIdent TV %}
 <int>    ::= ^d+      %{ int    p = PT p . TI    %}
 <double> ::= ^d+ ^. ^d+ (e (^-)? ^d+)? %{ double  p = PT p . TD %}

 eitherResIdent :: (String -> Tok) -> String -> Tok
 eitherResIdent tv s = if isResWord s then (TS s) else (tv s) where
   isResWord s = elem s ["double","int","while"]
```

The lexer file moreover defines the token type `Tok` used by the lexer and the parser.

The parser `ParJavaletteLight.y`

The parser file (in Happy) has a large number of token definitions (which we find it extremely valuable to generate automatically), followed by parsing rules corresponding closely to the source BNF rules. Here is a fragment containing examples of both parts:

```
%token
 '('      { PT _ (TS "(") }
 ')'      { PT _ (TS ")") }
 'double' { PT _ (TS "double") }
 'int'    { PT _ (TS "int") }
 'while'  { PT _ (TS "while") }
```

```
L_integ  { PT _ (TI $$) }
L_doubl  { PT _ (TD $$) }

%%

Integer : L_integ  { (read $1) :: Integer }
Double  : L_doubl  { (read $1) :: Double }

Stm :: { Stm }
Stm : Typ Ident ';'                    { SDecl $1 $2 }
  | Ident '=' Exp ';'                  { SAss $1 $3 }
  | Ident '++' ';'                     { SIncr $1 }
  | 'while' '(' Exp ')' '{' ListStm '}' { SWhile $3 (reverse $6) }
  | Stm ';'                            { $1 }

Exp0 :: { Exp }
Exp0 : Exp1 '<' Exp1 { ELt $1 $3 }
  | Exp1 { $1 }
```

The exported parsers have types of the following form, for any abstract syntax type `T`,

```
[Tok] -> Err T
```

returning either a value of type `T` or an error message, using a simple error monad. The input is a token list received from the lexer.

The pretty-printer `PrintJavaletteLight.hs`

The pretty-printer consists of a Haskell class `Print` with instances for all generated data types, taking precedence into account. The class method `prt` generates a list of strings for a syntax tree of any type.

```
instance Print Exp where
  prt i e = case e of
    ELt    exp0 exp ->
     prPrec i 0 (concat [prt 1 exp0 , ["<"] , prt 1 exp])
    EPlus  exp0 exp ->
     prPrec i 1 (concat [prt 1 exp0 , ["+"] , prt 2 exp])
    ETimes exp0 exp ->
     prPrec i 2 (concat [prt 2 exp0 , ["*"] , prt 3 exp])
```

The list is then put in layout (identation, newlines) by a *rendering* function, which is generated independently of the grammar, but written with easy modification in mind.

The case skeleton `SkelJavaletteLight.hs`

The case skeleton can be used as a basis when defining the compiler back end, e.g. type checker and code generator. The same skeleton is actually also used in the pretty printer. The case branches in the skeleton are initialized to show error messages saying that the case is undefined.

```
transExp :: Exp -> Result
transExp x = case x of
  ELt exp0 exp    -> failure x
  EPlus exp0 exp  -> failure x
  ETimes exp0 exp -> failure x
```

The language document `DocJavaletteLight.tex`

We show the main parts of the generated JavaletteLight document in a typeset form. The grammar symbols in the document are produced by LaTeX macros, with easy modification in mind.

The lexical structure of JavaletteLight

Identifiers

Identifiers ⟨*Ident* ⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

Integer literals ⟨*Int* ⟩ are nonempty sequences of digits.

Double-precision float literals ⟨*Double* ⟩ have the structure indicated by the regular expression ⟨*digit* ⟩ + '.' ⟨*digit* ⟩ + ('e' '-'? ⟨*digit* ⟩+)? 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 JavaletteLight are the following:

```
double   int   while
```

The symbols used in JavaletteLight are the following:

```
(     )     {
 }    ;     =
++    <     +
*
```

Comments

Single-line comments begin with `//`.
Multiple-line comments are enclosed with `/*` and `*/`.

The syntactic structure of JavaletteLight

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Prog* ⟩ ::= ⟨*Typ* ⟩ ⟨*Ident* ⟩ `(` `)` `{` ⟨*ListStm* ⟩ `}`

⟨*Stm* ⟩ ::= ⟨*Typ* ⟩ ⟨*Ident* ⟩ `;`
| ⟨*Ident* ⟩ = ⟨*Exp* ⟩ `;`
| ⟨*Ident* ⟩ ++ `;`
| `while` `(` ⟨*Exp* ⟩ `)` `{` ⟨*ListStm* ⟩ `}`
| ⟨*Stm* ⟩ `;`

⟨*Exp0* ⟩ ::= ⟨*Exp1* ⟩ < ⟨*Exp1* ⟩
| ⟨*Exp1* ⟩

⟨*Exp1* ⟩ ::= ⟨*Exp1* ⟩ + ⟨*Exp2* ⟩
| ⟨*Exp2* ⟩

⟨*Exp2* ⟩ ::= ⟨*Exp2* ⟩ `*` ⟨*Exp3* ⟩
| ⟨*Exp3* ⟩

⟨*Exp3* ⟩ ::= ⟨*Ident* ⟩
| ⟨*Integer* ⟩
| ⟨*Double* ⟩
| `(` ⟨*Exp* ⟩ `)`

⟨*ListStm* ⟩ ::= ϵ
| ⟨*Stm* ⟩ ⟨*ListStm* ⟩

⟨*Exp* ⟩ ::= ⟨*Exp0* ⟩

⟨*Typ* ⟩ ::= `int`
| `double`

The `makefile`

The makefile is used to run Alex on the lexer, Happy on the parser, and LaTeX on the document, by simply typing `make`. The `make clean` command removes the generated files.

The test bench file `TestJavaletteLight.hs`

The test bench file can be loaded in the Haskell interpreter hugs to run the parser and the pretty-printer on terminal or file input. The test functions display a syntax tree (or an error message) and the pretty-printer result from the same tree.

2.4.3 An optimization: left-recursive lists

The BNF representation of lists is right-recursive, following Haskell's list constructor. Right-recursive lists, however, are an inefficient way of parsing lists in an LALR parser. The smart programmer would implement a pair of rules such as JavaletteLight's

```
[].    [Stm] ::= ;
(:).   [Stm] ::= Stm [Stm] ;
```

not in the direct way,

```
ListStm : {- empty -} { [] }
  | Stm ListStm { (:) $1 $3 }
```

but under a left-recursive transformation:

```
ListStm : {- empty -} { [] }
  | ListStm Stm { flip (:) $1 $2 }
```

Then the smart programmer would also be careful to reverse the list when it is used:

```
Prog : Typ Ident '(' ')' '{' ListStm '}' { Fun $1 $2 (reverse $6) }
```

As reported in the Happy manual, this transformation is vital to avoid running out of stack space with long lists. Thus we have implemented the transformation in the BNF Converter for pairs of rules of the form

```
[].  [C] ::= ;
(:). [C] ::= C x [C] ;
```

where `C` is any category and `x` is any sequence of terminals (possibly empty).

There is another important parsing technique, recursive descent, which cannot live with left recursion at all, but loops infinitely with left-recursive grammars (cf. e.g. [1]). The question sometimes arises if, when designing a grammar, one should take into account what method will be used for parsing it. The view we are advocating is that the designer of the grammar should in the first place think of the abstract syntax, and let the parser generator perform automatic grammar transformations that are needed by the parsing method.

2.5 Discussion

2.5.1 Results

LBNF and the BNF Converter[4] were introduced as a teaching tool at the fourth-year compiler course in Spring 2003 at Chalmers. The goal was, on the one hand, to advocate the use of declarative and portable language definitions, and on the other hand, to leave more time for back-end construction in a compiler course. The students of the course had as a project to build a compiler in small groups, and grading was based on how much (faultless) functionality the compiler had, e.g. how many language features and how many back ends. The first results were encouraging: a majority (12/20) of the groups that finished their compiler used the BNF Converter. They all were able to produce faultless front ends and, in average, more advanced back ends than the participants of the previous year's edition of the course. In fact, the lexer+parser part of the compiler was estimated only to be 25 % of the work at the lowest grade, and 10 % at the highest grade—far from the old times when the parser was more than 50 % of a student compiler.

One worry about using the LBNF in teaching was that students would not really learn parsing, but just to write grammars. We found that this concern is not relevant when comparing LBNF with a parser tool like Happy and YACC: students writing their parsers in YACC are equally isolated from the internals of LR parsing as those writing in LBNF. In fact, as learning the formalism takes less time in the case of LBNF, the teacher can allocate more time for explaining how the LR parser works. The lexer was a bigger concern, though: since all of the token types needed for the project were predefined types in LBNF, the students did not need to write a single regular expression to finish their compiler! An obvious solution to this is to add some more exotic token types to the project specification.

The main conclusion drawn from the teaching experiment was that the

tool should be ported to C and Java, so that the students who don't use Haskell would have the same facilities as those who do.

Students in a compiler class usually implement toy languages. What about real-world languages? As an experiment, a complete LBNF definition of ANSI C, with [8] as reference, has been written[4]. The length of the LBNF source file is approximately the same as the length of the specification. Here is a word count comparison between the source file and what is generated:

```
$ wc C.cf
   288    1248   10203 C.cf

$ wc ?*C.* makefile
   287     707    5635 AbsC.hs
   518    1795   23062 DocC.tex
    72     501    2600 LexC.x
   477    2675   13761 ParC.y
   423    3270   18114 PrintC.hs
   336    1345    9178 SkelC.hs
    22     103     677 TestC.hs
     7      22     320 makefile
  2142   10418   73347 total
```

Another real-world example is the object-oriented specification language OCL [17][5]. And of course, the BNF Converter has been implemented by using modules generated from an LBNF grammar of LBNF (see the Appendix).

2.5.2 Well-behaved languages

A language that can be defined in LBNF is one whose syntax is context-free.[6] Its lexical structure can be described by a regular expression. Modern languages, like Java and C, are close to this ideal; Haskell, with its layout syntax and infix declarations, is a little farther. To rescue the maximum of existing Haskell or some other language would be a matter of detail handwork rather than general principles; and we have opted for keeping the LBNF formalism simple, sacrificing completeness.

We do not need to sacrifice *semantic completeness*, however: languages usually have a well-behaved subset that is enough for expressing everything that is expressible in the language. When designing new languages—and

[4]Work by Ulf Persson at Chalmers

[5]Work by Kristofer Johannisson at Chalmers

[6]Due to the parser tool used by the BNF converter, it moreover has to be LALR(1)-parsable; but this is a limitation not concerning LBNF as such.

even when using old ones—we find it a virtue to avoid exotic features. Such features are often included in the name of user-friendliness, but for *new* users, they are more often an obstacle than a help, since they violate the users' expectations gained from other languages.

2.5.3 Related work

The BNF Converter belongs largely to the YACC [7] tradition of compiler compilers, since it compiles a higher-level notation into the YACC-like notation of Happy, and since the parser is the most demanding part of a language front-end implementation. Another system on this level up from YACC is Cactus [11], which uses an EBNF-like notation to generate a Happy parser, an Alex lexer, and a data type definition for abstract syntax. Cactus, unlike the BNF Converter, aims for completeness, and it is indeed possible to define Haskell 98 (without layout rules) in it [5]. The price to pay is that the notation is less simple than LBNF. Moreover, because of Cactus's higher level of generality, it is no longer possible to extract a pretty-printer from a grammar. Nor does Cactus generate documentation.

For abstract syntax alone, the Zephyr definition language [16] defines a portable format and translations into program code in SML, Haskell, C, C++, Java, and SGML. Zephyr also generates functions for displaying syntax trees in these languages. But it does not support the definition of concrete syntax.

A survey of compiler tools on the web and in the literature tells that their authors almost invariably opt for expressivity rather than declarativity. The situation is different with grammar tools used in linguistic: there the declarativity and *reversibility* (i.e. usability for both parsing and generation) of grammar formalisms is highly valued. A major example of this philosophy are Definite Clause Grammars (DCG) [13]. In practice, DCGs are implemented as an embedded language in Prolog, and thereby some features of full Prolog are sometimes smuggled into grammars to improve expressivity; but this is usually considered harmful since it destroys declarativity and reversibility.

2.5.4 Future work

In addition to the obvious task of writing LBNF back ends to other languages than Haskell, there are many imaginable ways to extend the formalism itself. One direction is to connect LBNF with the Grammatical Framework GF [15]. GF is a rich grammar formalism originally designed to describe natural languages. LBNF was originally a spin-off of GF, customizing a subset of GF

to combine with standard compiler tools. The connection between LBNF and GF is close, with the difference that GF makes an explicit distinction between abstract and concrete syntax. Consider an LBNF rule describing multiplication:

```
Mult. Exp2 ::= Exp2 "*" Exp3 ;
```

This rule is in GF divided into two judgements: an abstract syntax function definition, and a concrete syntax linearization rule,

```
fun Mult : Exp -> Exp -> Exp ;
lin Mult e1 e2 =
    {s = parIf P2 e1 ++ "*" ++ parIf P3 e2 ; p = P2} ;
```

Precedence is treated as a parameter that regulates the uses of parentheses. In GF, the user can define new parameter types, and thus the precedences `P2` and `P3`, as well as the function `parIf`, are defined in the source code instead of being built in into the system, as in LBNF. GF moreover includes higher-order abstract syntax and dependent types, and a GF grammar can therefore define the type system of a language.

2.6 Conclusion

We see Labelled BNF as a natural step to a yet higher level in the development that led machine programmers to create assemblers, assembler programmers to create Fortran and C, and C programmers to create YACC and Lex. A high-level notation always hides details that can be considered well-understood and therefore uninteresting; this lets the users of the new notation to concentrate on new things. At the same time, it creates quality by eliminating certain errors. Inevitably, it also precludes some smart decisions that a human would make if hand-writing the generated code.

It would be too big a claim to say that LBNF can replace tools like YACC and Happy. It can only replace them if the language to be implemented is simple enough. Even though this is not always the case with legacy programming languages, there is a visible trend towards simple and standardized, "well-behaved" languages, and LBNF has proved useful in reducing the effort in implementing such languages.

2.7 Appendix: LBNF Specification

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which

guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of LBNF

Identifiers

Identifiers ⟨*Ident* ⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

String literals ⟨*String* ⟩ have the form `"`x`"`, where x is any sequence of characters.

Character literals ⟨*Char* ⟩ have the form `'`c`'`, where c is any single character.

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 LBNF are the following:

```
char         comment  digit
entrypoints  eps      internal
letter       lower    token
upper
```

The symbols used in LBNF are the following:

```
;    .   ::=
[    ]    _
(    :   )
|    -   *
+    ?    {
 }   ,
```

Comments

Single-line comments begin with $--$.
Multiple-line comments are enclosed with $\{-$ and $-\}$.

The syntactic structure of LBNF

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Grammar* ⟩ ::= ⟨*ListDef* ⟩

⟨*ListDef* ⟩ ::= ϵ
| ⟨*Def* ⟩ `;` ⟨*ListDef* ⟩

⟨*ListItem* ⟩ ::= ϵ
| ⟨*Item* ⟩ ⟨*ListItem* ⟩

⟨*Def* ⟩ ::= ⟨*Label* ⟩ `.` ⟨*Cat* ⟩ `::=` ⟨*ListItem* ⟩
| `comment` ⟨*String* ⟩
| `comment` ⟨*String* ⟩ ⟨*String* ⟩
| `internal` ⟨*Label* ⟩ `.` ⟨*Cat* ⟩ `::=` ⟨*ListItem* ⟩
| `token` ⟨*Ident* ⟩ ⟨*Reg* ⟩
| `entrypoints` ⟨*ListIdent* ⟩

⟨*Item* ⟩ ::= ⟨*String* ⟩
| ⟨*Cat* ⟩

⟨*Cat* ⟩ ::= `[` ⟨*Cat* ⟩ `]`
| ⟨*Ident* ⟩

⟨*Label* ⟩ ::= ⟨*Ident* ⟩
| `_`
| `[]`
| `(:)`
| `(:[])`

⟨*Reg2* ⟩ ::= ⟨*Reg2* ⟩ ⟨*Reg3* ⟩
| ⟨*Reg3* ⟩

⟨*Reg1* ⟩ ::= ⟨*Reg1* ⟩ `|` ⟨*Reg2* ⟩
| ⟨*Reg2* ⟩ `-` ⟨*Reg2* ⟩
| ⟨*Reg2* ⟩

⟨*Reg3* ⟩ ::= ⟨*Reg3* ⟩ `*`
| ⟨*Reg3* ⟩ +
| ⟨*Reg3* ⟩ `?`
| `eps`
| ⟨*Char* ⟩
| `[` ⟨*String* ⟩ `]`
| `{` ⟨*String* ⟩ `}`
| `digit`
| `letter`
| `upper`
| `lower`
| `char`
| `(` ⟨*Reg* ⟩ `)`

⟨*Reg* ⟩ ::= ⟨*Reg1* ⟩

⟨*ListIdent* ⟩ ::= ⟨*Ident* ⟩
| ⟨*Ident* ⟩ `,` ⟨*ListIdent* ⟩

Bibliography

[1] A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] C. Dornan. Alex: a Lex for Haskell Programmers, 1997. `http://www.cs.ucc.ie/dornan/alex.html`.

[3] C. Dornan. JLex: A Lexical Analyzer Generator for Java, 2000. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[4] M. Forsberg and A. Ranta. Labelled BNF: a high-level formalism for defining well-behaved programming languages. *Proceedings of the Estonian Academy of Sciences: Physics and Mathematics*, 52:356–377, 2003. Special issue on programming theory edited by J. Vain and T. Uustalu.

[5] T. Hallgren. The Haskell 98 grammar in Cactus, 2001. `http://www.cs.chalmers.se/~hallgren/CactusExample/`.

[6] S. E. Hudson. CUP Parser Generator for Java, 1999. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[7] S. C. Johnson. Yacc — yet another compiler compiler. Technical Report CSTR-32, AT & T Bell Laboratories, Murray Hill, NJ, 1975.

[8] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1988. 2nd edition.

[9] M. E. Lesk. Lex — a lexical analyzer generator. Technical Report 39, Bell Laboratories, Murray Hill, N.J., 1975.

[10] S. Marlow. Happy, The Parser Generator for Haskell, 2001. `http://www.haskell.org/happy/`.

[11] N. Martinsson. Cactus (Concrete- to Abstract-syntax Conversion Tool with Userfriendly Syntax) . Master's Thesis in Computer Science, 2001. `http://www.mdstud.chalmers.se/~md6nm/cactus/`.

[12] P. Naur. Revised Report of the Algorithmic Language Algol 60. *Comm. ACM*, 6:1–17, 1963.

[13] F. Pereira 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.

[14] S. Peyton Jones and J. Hughes. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available at `http://www.haskell.org`, February 1999.

[15] A. Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 2004.

[16] D. C. Wang, A. W. A. J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. *Proceedings of the Conference on Domain-Specific Languages*, 1997.

[17] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Addison-Wesley, 1999.

Chapter 3

Multilingual Front-End Generation from Labelled BNF Grammars

Michael Pellauer, Markus Forsberg, and Aarne Ranta
Chalmers University of Technology
Department of Computing Science
SE-412 96 Göteborg, Sweden
{*pellauer, markus, aarne*}*@cs.chalmers.se*

Paper published:
Technical Report no. 2004-09 in Computing Science at Chalmers University of Technology and Göteborg University

The paper has been updated by M. Forsberg with Sec. 3.10 that gives a brief up-to-date view of the tool since this report was written.

Abstract

The BNF Converter is a compiler-construction tool that uses a Labelled BNF grammar as the single source of definition to extract the abstract syntax, lexer, parser and pretty printer of a language. The added layer of abstraction allows it to perform multilingual code generation. As of version 2.0 it is able to output front ends in Haskell, Java, C or C++.

3.1 Introduction

Language implementors have long used generative techniques to implement parsers. However, with advances in language design the focus of the compiler front end has shifted from the parsing of difficult languages to the definition of a complex abstract-syntax-tree data structure. It is the common practise for modern implementors to use one tool to generate an abstract syntax tree, another to generate a lexer, and a third to generate a parser.

Yet this requires that the implementor learn three separate configuration syntaxes, and maintain disparate source files across changes to the language definition. The BNF Converter[1] is a compiler-construction tool based on the idea that from a single source grammar it is possible to generate both an abstract syntax tree definition, including a traversal function, and a concrete syntax, including lexer, parser and pretty printer.

The decoupling of the grammar description from the implementation language allows our tool to perform multilingual code generation. As of version 2.0 the BNF Converter is able to generate a front end in Haskell, Java, C, or C++. This continues the tradition of Andrew Appel [1, 2, 3], whose textbooks apply the same compiler methodology across three widely different target languages.

The BNF Converter Approach

With the BNF Converter the user specifies a grammar using an enhanced version of Backus Naur Form called Labelled BNF (LBNF), described in Section 3.2. This grammar is language independent and serves as a single source for all language definition changes, increasing maintainability. After the user selects a target language it is used generate the following:

- Abstract syntax tree data structure
- Lexer and parser specification
- Pretty printer and traversal skeleton
- Test bench and Makefile
- Language documentation

This unified approach to generation offers many advantages. First of all, the increased level of abstraction allows our tool to check the grammar for problems, rather than attempting to check code written directly in an implementation language like C. Secondly, the components are generated to interoperate correctly together with no additional work from the user. Packages such as the abstract syntax and pretty printer can be supplied as development frameworks to encourage applications to make use of the new language.

[1] Available from the BNF Converter website [8]

Combined with BNF Converter 2.0's multiingual generation this facilitates interesting possibilities, such as using a server application written in C++ to pretty-print output that will be parsed by a Java application running on a PDA. The language maintainers themselves can experiment with implementing the same methodology over multiple languages, even creating a prototype language implementation in Haskell, then switching to C for development once the language definition has been finalized.

This paper gives an overview of the LBNF grammar formalism. We then compare the methodology the BNF Converter uses to produce code in Haskell, Java, C++, and C, highlighting some of the differences of generating a compiler in these languages. Finally, we conclude with a discussion of our practical experiences using the tool in education and language prototyping.

Language Describability

The requirements that the BNF Converter puts on a language in order to describe it are simple and widely accepted: the syntax must be definable by a context-free grammar and the lexical structure by a regular expression. The parser's semantic actions are only used for constructing abstract syntax trees and can therefore not contribute to the definition of the language. Toy languages in compiler text books are usually designed to meet these criteria, and the trend in real languages is to become closer to this ideal.

Often it is possible to use preprocessing to turn a language that almost meets the criteria into one that meets them completely. Features such as layout syntax, for example, can be handled by adding a processing level between the lexer and the parser. Our experiences with real-world languages are discussed in Section 3.8.

3.2 The LBNF Grammar Formalism

The input to the BNF Converter is a specification file written in the LBNF grammar formalism. LBNF is an entirely declarative language designed to combine the simplicity and readability of Backus Naur Form with a handful of features to hasten the development of a compiler front-end.

Besides declarativity, we find it important that LBNF has its own semantics, instead of only getting its meaning through translations to Haskell, Java, C, etc. This means, among other things, that LBNF grammars are type checked on the source, so that semantic errors do not appear unexpectedly in the generated code. Full details on LBNF syntax and semantics are given in [9], as well as on the BNF Converter homepage [8].

3.2.1 Rules and Labels

At the most basic level, an LBNF grammar is a BNF grammar where every rule is given a *label*. The label is an identifier used as the constructor of syntax trees whose subtrees are given by the non-terminals of the rule; the terminals are just

ignored. As a first example, consider a rule defining assignment statements in C-like languages:

```
SAssign. STM ::= Ident "=" EXP ;
```

Apart from the label `SAssign`, the rule is an ordinary BNF rule, with terminal symbols enclosed in double quotes and non-terminals written without quotes. A small, though complete example of a grammar is given in Section 3.2.4.

Some aspects of the language belong to its lexical structure rather than its grammar, and are described by regular expressions rather than by BNF rules. We have therefore added to LBNF two rule formats to define the lexical structure: *tokens* and *comments* (Section 3.2.2).

Creating an abstract syntax by adding a node type for every BNF rule may sometimes become too detailed, or cluttered with extra structures. To remedy this, we have identified the most common problem cases, and added to LBNF some extra conventions to handle them (Section 3.2.3).

Finally, we have added some *macros*, which are syntactic sugar for potentially large groups of rules and help to write grammars concisely, and some *pragmas*, such as the possibility to limit the entrypoints of the parser to a subset of nonterminals.

3.2.2 Lexer Definitions

The `token` definition format

The `token` definition form enables the LBNF programmer to define new lexical types using a simple regular expression notation. For instance, the following defines the type of identifiers beginning with upper-case letters.

```
token UIdent (upper (letter | digit | '\_')*) ;
```

The type `UIdent` becomes usable as an LBNF nonterminal and as a type in the abstract syntax. Each token type is implemented by a `newtype` in Haskell, as a `String` in Java, and as a `typedef` to `char*` in C/C++.

Predefined token types

To cover the most common cases, LBNF provides five predefined token types:

`Integer`, `Double`, `Char`, `String`, `Ident`

These types have predefined lexer rules, but could also be defined using the regular expressions of LBNF (see [9]). In the abstract syntax, the types are represented as corresponding types in the implementation language; `Ident`

is treated like user-defined token types. Only those predefined types that are actually used in the grammar are included in the lexer and the abstract syntax.

The `comment` definition format

Comments are segments of source code that include free text and are not passed to the parser. The natural place to deal with them is in the lexer. A `comment` definition instructs the lexer generator to treat certain pieces of text as comments.

The comment definition takes one or two string arguments. The first string defines how a comment begins. The second, optional string marks the end of a comment; if it is not given then the comment is ended by a newline. For instance, the Java comment convention is defined as follows:

```
comment "//" ;
comment "/*" "*/" ;
```

3.2.3 Abstract Syntax Conventions

Semantic dummies

Sometimes the concrete syntax of a language includes rules that make no semantic difference. For instance, the C language accepts extra semicolons after statements. We do not want to represent these extra semicolons in the abstract syntax. Instead, we use the following convention:

> If a rule has only one non-terminal on the right-hand-side, and this non-terminal is the same as the value type, then it can have as its label an underscore (`_`), which does not add anything to the syntax tree.

Thus, we can write the following rule in LBNF:

```
_ . STM ::= STM ";" ;
```

Precedence levels

A common idiom in (ordinary) BNF is to use indexed variants of categories to express precedence levels, e.g. `EXP, EXP2, EXP3`. The precedence level regulates the order of parsing, including associativity. An expression belonging to a level n can be used on any level $< n$ as well. Parentheses lift an expression of any level to the highest level.

Distinctions between precedence levels and moving expressions between them can be defined by BNF rules, but we do not want these rules to clutter the abstract syntax. Therefore, we can use semantic dummies (`_`) for the transitions, together with the following convention:

> A category symbol indexed with a sequence of digits is treated as a type synonym of the corresponding non-indexed symbol.

A non-indexed symbol is treated as having the level 0. The following grammar shows how the convention works in a familiar example with arithmetic sums and products:

```
EPlus.  EXP  ::= EXP  "+" EXP2 ;
ETimes. EXP2 ::= EXP2 "*" EXP3 ;
EInt.   EXP3 ::= Integer ;
_.      EXP  ::= EXP2 ;
_.      EXP2 ::= EXP3 ;
_.      EXP3 ::= "(" EXP ")" ;
```

The indices also guide the pretty-printer to generate a correct, minimal number of parentheses.

The `coercions` macro provides a shorthand for generating the dummy transition rules concisely. It takes as its arguments the unindexed category and the highest precedence level. So the final three rules in the above example could be replaced with:

```
coercions EXP 3 ;
```

Polymorphic lists

It is easy to define monomorphic list types in LBNF:

```
NilDEF.  ListDEF ::= ;
ConsDEF. ListDEF ::= DEF ";" ListDEF ;
```

But LBNF also has a *polymorphic list notation*. It follows the Haskell syntax but is automatically translated to native representations in Java, C++, and C.

```
[].  [DEF] ::= ;
(:). [DEF] ::= DEF ";" [DEF] ;
```

The basic ingredients of this notation are

> `[`C`]`, the category of lists of type C,
>
> `[]` and `(:)`, the Nil and Cons rule labels,
>
> `(:[])`, the rule label for one-element lists.

The list notation can also be seen as a variant of the Kleene star and plus, and hence as an ingredient from *Extended BNF* in LBNF.

Using the polymorphic list type makes BNF Converter perform an automatic optimization: *left-recursive lists*. Standard lists in languages like Haskell are right-recursive, but LR parsers favor left-recursive lists because they save stack space. BNF Converter allows programmers to define familiar right-recursive lists, but translates them into left-recursive variants in parser generation. When used in another construction, the list is automatically reversed. The code examples below, generated from the grammar in Section 3.2.4, show how this works in the different parser tools.

The `terminator` and `separator` macros

The `terminator` macro defines a pair of list rules by what token terminates each element in the list. For instance,

```
terminator STM ";" ;
```

is shorthand for the pair of rules

```
[].  [STM] ::= ;
(:). [STM] ::= STM ";" [STM] ;
```

The `separator` macro is similar, except that the separating token is not expected after the last element of the list. The qualifier `nonempty` can be used in both macros to make the one-element list the base case.

3.2.4 Example Grammar

A small example LBNF grammar is given in Figure 3.1. It describes a language of boolean expressions, perhaps written as part of a larger grammar. In this small language a PROGRAM is simply a list of expressions terminated by semicolons. The expressions themselves are just logical AND and OR of true, false, or variable names represented by the LBNF built-in type `Ident`.

This example, though small, is representative because it uses both polymorphic lists and precedence levels (the AND operator having higher precedence than OR). We will use this single source example to explore BNF Converter's generation methodology across multiple implementation languages.

```
PROGRAM. PROGRAM ::= [EXP] ;

EOr.    EXP  ::=
           EXP "||" EXP1 ;
EAnd.   EXP1 ::=
           EXP1 "&&" EXP2 ;
ETrue.  EXP2 ::= "true" ;
EFalse. EXP2 ::= "false" ;
EVar.   EXP2 ::= Ident ;
terminator EXP ";" ;
coercions EXP 2 ;
```

Figure 3.1: LBNF Source code for all examples

3.3 Haskell Code Generation

The process the BNF Converter uses to generate Haskell code is quite straightforward. Here we will only present an overview of this process, for comparison with the methods used for Java and C. For a more complete look at this process see the documentation on the BNF Converter Homepage [8].

The Abstract Syntax

Consider the example grammar given in Section 3.2.4.

The Haskell abstract syntax generated by the BNF Converter, shown in Figure 3.2A, is essentially what a Haskell programmer would write by hand, given the close relationship between a declarative grammar and Haskell's algebraic data types.

The Lexer and Parser

The BNF Converter generates lexer and parser specifications for the Alex [6] and Happy [17] tools. The lexer file (omitted for space considerations) consists mostly of standard rules for literals and identifiers, but has rules added for reserved words and symbols (i.e. terminals occurring in the grammar), regular expressions defined in token definitions, and comments.

The Happy specification (Figure 3.2B) has a large number of token definitions, followed by parsing rules corresponding closely to the source BNF rules. Note the left-recursive list transformation, as defined in Section 3.2.3.

The Pretty Printer and Case Skeleton

The pretty printer consists of a Haskell class `Print` with instances for all generated data types, taking precedence into account. The class method `prt` generates a list of strings for a syntax tree of any type (Figure 3.2C).

The list of strings is then put in layout (indentation, newlines) by a *rendering* heuristic, which is generated independently of the grammar. This function is designed to make C-like languages look good by default, but it is written with easy modification in mind.

The case skeleton (Figure 3.2D) is a simple traversal of the abstract syntax tree representation that can be used as a template when defining the compiler back end, e.g. type checker and code generator. The same methodology is also used to generate the pretty printer. The case branches in the skeleton are initialized to fail, and the user can simply replace them with something more interesting.

The Makefile and Test Bench

The generated test bench file can be loaded in the Haskell interpreter `hugs` to run the parser and the pretty printer on terminal or file input. If parsing succeeds the test functions display a syntax tree, and the pretty printer linearization. Otherwise an error message is displayed.

A simple makefile is created to run Alex on the lexer, Happy on the parser, and LaTeX on the document, by simply typing `make`. The `make clean` command removes the generated files.

Translation Summary

Overall, it is easy to represent an LBNF grammar as a Haskell data type—a straightforward translation between source productions and algebraic data types. Language implementors have long known that the similarities between algebraic data types and grammar specifications make functional programming a good choice for compilers.

3.4 Java Code Generation

Translating an LBNF grammar into an object-oriented language is less straightforward. Appel outlines two possible approaches to abstract syntax representation in *Modern Compiler Implementation in Java* [2].

In the first method, which Appel refers to as the "Object-Oriented method," there is one Java class for each rule in the language grammar.

A. Abstract Syntax

```
data PROGRAM = PROGRAM [EXP]
   deriving (Eq, Show)
data EXP =
    EOr EXP EXP
  | EAnd EXP EXP
  | ETrue
  | EFalse
  | EVar
   deriving (Eq, Show)
```

B. Happy Parser

```
PROGRAM :: { PROGRAM }
PROGRAM : ListEXP { PROGRAM (reverse $1) }
EXP :: { EXP }
EXP : EXP '||' EXP1 { EOr $1 $3 }
  | EXP1 { $1 }
EXP1 :: { EXP }
EXP1 : EXP1 '&&' EXP2 { EAnd $1 $3 }
  | EXP2 { $1 }
EXP2 :: { EXP }
EXP2 : 'true' { ETrue }
  | 'false' { EFalse }
  | Ident { EVar $1 }
  | '(' EXP ')' { $2 }
ListEXP :: { [EXP] }
ListEXP : {- empty -} { [] }
  | ListEXP EXP ';' { flip (:) $1 $2 }
```

C. Pretty Printer

```
instance Print PROGRAM where
  prt i e = case e of
   PROGRAM exp -> prPrec i 0
                  (concat [prt 0 exp])
instance Print EXP where
  prt i e = case e of
   EOr exp0 exp -> prPrec i 0
    (concat
      [prt 0 exp0 , ["||"] , prt 1 exp])
   EAnd exp0 exp -> prPrec i 1
    (concat
      [prt 1 exp0 , ["&&"] , prt 2 exp])
   ETrue  -> prPrec i 2
      (concat [["true"]])
   EFalse  -> prPrec i 2
      (concat [["false"]])
   EVar id -> prPrec i 2
      (concat [prt 0 id])
  prtList es = case es of
   [] -> (concat [])
   x:xs ->
    (concat
      [prt 0 x , [";"] ,
       prt 0 xs])
```

D. Case Skeleton

```
transPROGRAM :: PROGRAM -> Result
transPROGRAM x = case x of
  PROGRAM exp  -> failure x
transEXP :: EXP -> Result
transEXP x = case x of
  EOr exp0 exp  -> failure x
  EAnd exp0 exp  -> failure x
  ETrue  -> failure x
  EFalse  -> failure x
  EVar id  -> failure x
```

Figure 3.2: Haskell source code fragments generated from Figure 3.1

Each class inherits from a single superclass, and each class defines operations on itself. For instance, if our compiler were to translate to SPARC and Intel assembly code each class would have a method `toSPARC()` and `toIntel()` that would translate itself to the appropriate representation. The advantage of this method is that it is easy to add new language categories. The user may add new classes containing the appropriate methods without altering existing definitions. The disadvantage is that it can be hard to add new syntax tree traversals. Adding a function `toAlpha()` for instance, could result in editing hundreds of classes.

In the second "syntax separate from interpretations" method, there is still one Java class for each grammar rule, but now classes are simply empty data structures with no methods aside from a constructor. Translation functions are removed from the data structure, and traverse the tree by straightforward manner. With this method it is easy to add new traversals, and these functions can make better use of context information than single objects' methods. The disadvantage is that adding new language constructs requires editing all existing traversal functions to handle the new cases.

However, the BNF Converter, which makes the grammar the central point of all language changes, lessens this disadvantage. Additionally, since translation functions are now traversals, it is easy for our tool to generate skeleton functions as we do in Haskell and for the user to reuse the template in all transformations.[2] Therefore the BNF Converter uses this method in generating Java (and C++) abstract syntax.

Java Abstract Syntax Generation

Let us return to our example of Boolean Expressions from earlier (Section 3.2.4). Given this grammar, the BNF Converter will generate the abstract syntax found in Figure 3.3A, following Appel's method.

There are several differences between this transformation and the Haskell version that should be highlighted. First, experienced Java programmers will quickly notice that all the generated classes are public, and in Java public classes must go into their own `.java` file, with class name matching the file name. Since it common to have hundreds of productions in an LBNF grammar, the user's source directory can quickly become cluttered, so Abstract Syntax classes are placed into a sub-package called `Absyn`, and

[2] Of course, if the user implements a translation and then modifies the language definition they must still change the implemented code to reflect the modifications. However, they can refer to the template function in order to locate the differences.

thus must be kept in a file-system subdirectory of the same name, which the tool creates.

There is a second difference in the code in Figure 3.3A: names. Classes in Java have instance variables and parameters, and all of these require unique names (whereas in Haskell data structures the names are only optional). First, we realize that parameter names generally are not important—we can simply give them the name "p" plus a unique number. The names of instance variables, on the other hand, do matter. The BNF Converter converts the type name to lowercase and adds an underscore to prevent conflicts with reserved words. If there is more than one variable of a type then they are numbered. Thus, the classes `EPlus` and `ETimes` have members `exp_1` and `exp_2`.

Notice that Appel's method uses public instance variables, which may be regarded as bad style by object-oriented programmers today. We have chosen to remain with the original method, both to keep a higher correspondence to the textbook, and to ease the generation of the pretty printer and other traversals.

Finally recall that Java 1.4 does not support polymorphic lists. Generic types is supported in the Java 2 Platform, Standard Edition 1.5 release, also implemented in BNF Converter (see section 3.5). The BNF Converter Java 1.4 backend generates simple null-terminated linked lists for each list that the grammar uses. These special classes are prefixed with "List," such as the class `ListEXP` above, which takes the place of Haskell's [EXP].

The Lexer and Parser

The BNF Converter generates specification files for the JLex [7] and CUP [13] tools which create a lexer and parser in a manner similar to the Haskell version. The difference between the tools is mainly a matter of syntax. For example, CUP cannot work with strings directly but requires terminal symbols be defined for each language symbol or reserved word. Also, CUP does not refer to variables with $ variables like Bison, but rather by assigning names to all possibly-used values. Specifications equivalent to the Happy code in Figure 3.2B is shown in Figures 3.3B.

The Java Pretty Printer and Skeleton Function

Similar to the Haskell version, the Java pretty printer linearizes the abstract syntax tree using some easily-modifiable heuristics. It follows the method Appel outlines, using Java's `instanceof` operator to determine which sub-

class it is dealing with, then down-casting and working with the public variables. For example, the code to pretty-print an EXP is found in Figure 3.3D.

However, the pretty printer alone is not enough to test the correctness of a parse. In the Haskell version the built-in `show` function is used to print out the abstract syntax tree so that the programmer can confirm its correctness. We could use Java's `toString()` method in a similar role, but this is not satisfying, as it is generally used for debugging purposes. Instead, the BNF Converter adds a second method to the pretty printer, similar to Haskell's `show` function, shown in Figure 3.3E.

Throughout both methods the generated code makes use of Java's `StringBuffer` class to efficiently build the result of the linearization.

This `instanceof` method is also used to generate a code skeleton. However, this method may seem awkward to many object-oriented programmers, who are often taught to avoid `instanceof` wherever possible.

Much more familiar is the *Visitor Design Pattern* [12]. In it each member of the abstract syntax tree implements an `accept` method, which then calls the appropriate method in the visiting class (double-dispatch).

There is no reason that these two methods cannot live side by side. Therefore the BNF Converter generates code skeletons using both Appel's method and a Visitor interface and skeleton (Figure 3.3F).

Most familiar Visitor Design Patterns use a Visitee-traversal algorithm. That is to say, visiting the top member of a list will automatically visit all the members of the list. However, the BNF Converter-generated pattern uses Visitor-traversal. This means that it is the Visitor's responsibility, when visiting a list, to visit all the members in turn. This is because certain algorithms that compilers want to implement are not compositional, so performing a transformation on a single member may be quite different than performing that transformation on a certain pattern of nodes. For example, during peephole analysis a compiler may wish to merge to subsequent additions into a single operation, but may want to leave single additions unchanged. In our experience, these types of algorithms are easier to implement if the Visitor itself is in control of the traversal.

The Test Bench and Makefile

With the pretty printer defined it is trivial to define a test bench and makefile to compile the code. However, the lack of an interactive environment such as Haskell's `hugs` means that the user is not able to specify which parser is used. Instead the first-defined entry-point of the grammar is used by default.

A. Abstract Syntax

```
public class PROGRAM {
  public ListEXP listexp_;
  public PROGRAM(ListEXP p1)
           { listexp_ = p1; }
}
public abstract class EXP {}
public class EAnd extends EXP {
  public EXP exp_1, exp_2;
  public EAnd(EXP p1, EXP p2)
     { exp_1 = p1; exp_2 = p2; }
}
public class EOr extends EXP {
  public EXP exp_1, exp_2;
  public EOr(EXP p1, EXP p2)
     { exp_1 = p1; exp_2 = p2; }
}
public class ETrue extends EXP {
  public ETrue() { }
}
public class EFalse extends EXP {
  public EFalse() { }
}
public class EVar extends EXP {
  public String ident_;
  public EVar(String p1)
              { ident_ = p1; }
}
public class ListEXP {
  public EXP exp_;
  public ListEXP listexp_;
  public ListEXP(EXP p1, ListEXP p2)
    { exp_ = p1; listexp_ = p2; }
}
```

B. CUP Parser

```
terminal _SYMB_0;    //   ||
terminal _SYMB_1;    //   &&
terminal _SYMB_2;    //   ;
terminal _SYMB_3;    //   (
terminal _SYMB_4;    //   )
terminal _SYMB_5;    //   false
terminal _SYMB_6;    //   true
terminal String _IDENT_;
PROGRAM ::= ListEXP:p_1  {:
    if (p_1 != null) p_1 = p_1.reverse();
      RESULT = new Absyn.PROGRAM(p_1); :}
;
EXP ::= EXP:p_1 _SYMB_0 EXP1:p_3  {:
     RESULT = new Absyn.EOr(p_1, p_3); :}
  | EXP1:p_1  {:  RESULT = (p_1); :}
;
EXP1 ::= EXP1:p_1 _SYMB_1 EXP2:p_3  {:
    RESULT = new Absyn.EAnd(p_1, p_3); :}
  | EXP2:p_1  {:  RESULT = (p_1); :}
;
EXP2 ::= _SYMB_6 {:
           RESULT = new Absyn.ETrue(); :}
  | _SYMB_5 {:
          RESULT = new Absyn.EFalse(); :}
  | _IDENT_:p_1 {:
        RESULT = new Absyn.EVar(p_1); :}
```

CUP Parser (continued)

```
  | _SYMB_3 EXP:p_2 _SYMB_4 {:
                      RESULT = (p_2); :}
;
ListEXP ::= /*empty*/{: RESULT = null; :}
  | ListEXP:p_1 EXP:p_2 _SYMB_2  {:
 RESULT = new Absyn.ListEXP(p_2, p_1); :}
;
```

C. Pretty Printer

```
  private static void
  pp(Absyn.EXP exp, int _i_) {
    if (exp instanceof Absyn.EOr) {
       Absyn.EOr eor = (Absyn.EOr) exp;
       if (_i_ > 0) render(_L_PAREN);
       pp(eor.exp_1, 0);
       render("||");
       pp(eor.exp_2, 1);
       if (_i_ > 0) render(_R_PAREN);
    }
    if (exp instanceof Absyn.EAnd) {
       Absyn.EAnd eand = (Absyn.EAnd) exp;
       if (_i_ > 1) render(_L_PAREN);
       pp(eand.exp_1, 1);
       render("&&");
    ...
```

D. Abstract Syntax Viewer

```
  private static void sh(Absyn.EXP exp)
  {
    if (exp instanceof Absyn.EOr) {
       Absyn.EOr eor = (Absyn.EOr) exp;
       render("(");
       render("EOr");
       sh(eor.exp_1);
       sh(eor.exp_2);
       render(")");
    }
    if (exp instanceof Absyn.EAnd) {
       Absyn.EAnd eand = (Absyn.EAnd) exp;
       render("(");
       render("EAnd");
    ...
```

E. Visitor Design Pattern

```
  public void visitEOr(Absyn.EOr eor) {
    /* Code For EOr Goes Here */
    eor.exp_1.accept(this);
    eor.exp_2.accept(this);
  }
  public void visitEAnd(Absyn.EAnd eand) {
    /* Code For EAnd Goes Here */
    ...
  public void
  visitListEXP(Absyn.ListEXP listexp) {
    while(listexp!= null) {
      /* Code For ListEXP Goes Here */
      listexp.listexp_.accept(this);
      listexp = listexp.listexp_;
      ...
```


Figure 3.3: Java source code fragments generated from Figure 3.1

However it is easy for the user to specify another entry point directly in the test bench source code.

Translation Summary

Overall, translating from a declarative grammar to an object-oriented abstract syntax definition is possible, however the translation introduces a number of new complications such as the names of instance variables. A comparison of Figure 3.2A and Figure 3.3A emphasizes the challenges of implementing a compiler in Java.

The BNF Converter tries to deal with these complications in a consistent way to ease the implementation of the rest of the compiler. Appel's syntax-separate-from-interpretations method introduces several conventions that object-oriented programmers may find confusing at first. However, in practice the ease of using the generated transformation templates should help users to quickly overcome these difficulties.

3.5 Java 1.5 Generation

The Java backend has been adapted to Java 1.5 by Björn Bringert at Computing Science, Chalmers. The main difference is *generic types*. Generic types ensure type safety without having to resort to monomorphic types. For example, the container types in Java 1.5 are parameterized by a type `T`. Compare this with Java 1.4 where all objects in a container are of type `Object`. Furthermore, some adaptions were needed to reflect these changes, in particular in the syntax tree traversal.

3.6 C++ Code Generation

With the Java version implemented it was straightforward to add support for C++ generation, using Flex [11] and Bison [10]. This translation is similar to the Java version—the main difference being the additional complications of destructors and the separation of interface (.H) and implementation (.cpp) files. The details of this translation have been omitted for space considerations but may be found on the BNF Converter homepage [8].

3.7 C Code Generation

The Abstract Syntax

The translation to C code is quite different than the other languages. It follows the methodology used by Appel in the C Version of his textbook [1].

In this methodology, each grammar category is represented by a C `struct`. Each struct has an enumerated type indicating which LBNF label it represents, and a `union` of pointers to all corresponding non-terminal categories. Our boolean-expressions example generates the structs shown in Figure 3.4A. Structs are originally named with an underscore, and `typdef` declarations clean up the code by making the original grammar name refer to a pointer to that struct.

Data structure instances are created by using constructor functions, which are generated for each struct (Figure 3.4B). These functions are straightforward to generate and take the place of the `new` operator and constructors in an object-oriented language.

The Lexer and Parser

The BNF Converter also generates a lexer specification file for Flex and a parser specification file for Bison. Figure 3.4C shows specification code equivalent to the examples in Figures 3.2B and 3.3B.

One complication is that there is no way to access the result of the parse without storing a global pointer to it. This means that every potential entry point production must store a pointer to the parse (the `YY_RESULT` variables in Figure 3.4C), in case they are the final successful category. Users can limit the performance impact of this by using the `entrypoints` pragma.

The Pretty Printer and Case Skeleton

Any algorithm that wishes to traverse the tree must switch on the `kind` field of each node, then recurse to the appropriate members. For example, Figure 3.4E shows the pretty-printer traversal. The abstract syntax tree viewer and skeleton template are similar traversals.

Translation Summary

While it is straightforward to generate a parser and a data structure to represent the results of a parse in C, the combination of pointers and unions (seen in Figures 3.4B and 3.4D) results in code that can be sometimes hard

A. Abstract Syntax

```
struct PROG_ {
  enum {is_PROG} kind;
  union {
    struct { ListEXP listexp_; } prog_;
  } u;
};
typedef struct PROG_ *PROG;
struct EXP_ {
  enum { is_EOr, is_EAnd, is_ETrue,
         is_EFalse, is_EVar } kind;
  union {
    struct { EXP exp_1, exp_2; } eor_;
    struct { EXP exp_1, exp_2; } eand_;
    struct { Ident ident_; } evar_;
  } u;
};
typedef struct EXP_ *EXP;
struct ListEXP_ {
  EXP exp_;
  ListEXP listexp_;
};
typedef struct ListEXP_ *ListEXP;
```

B. Constructor Functions

```
EXP make_EOr(EXP p1, EXP p2) {
  EXP tmp = (EXP) malloc(sizeof(*tmp));
  if (!tmp) {
    fprintf(stderr,
            "Error: out of memory!\n");
    exit(1);
  }
  tmp->kind = is_EOr;
  tmp->u.eor_.exp_1 = p1;
  tmp->u.eor_.exp_2 = p2;
  return tmp;
}
EXP make_EAnd(EXP p1, EXP p2)
{
...
```

C. Bison Parser

```
PROGRAM YY_RESULT_PROGRAM_ = 0;
PROGRAM pPROGRAM(FILE *inp) {
  initialize_lexer(inp);
  if (yyparse()) /* Failure */
    return 0;
  else /* Success */
    return YY_RESULT_PROGRAM_;
}
...
%token _ERROR_   /* Terminal */
%token _SYMB_0    /*   ||   */
%token _SYMB_1    /*   &&   */
%token _SYMB_2    /*   ;    */
%token _SYMB_3    /*   (    */
%token _SYMB_4    /*   )    */
%token _SYMB_5    /* false  */
%token _SYMB_6    /*  true  */
...
```

Bison Parser Continued

```
%%
PROGRAM : ListEXP {
   $$ = make_PROGRAM(reverseListEXP($1));
                YY_RESULT_PROGRAM_= $$; }
;
EXP : EXP _SYMB_0 EXP1 {
                   $$ = make_EOr($1, $3);
                    YY_RESULT_EXP_= $$; }
  | EXP1 { $$ = $1; YY_RESULT_EXP_= $$; }
;
EXP1 : EXP1 _SYMB_1 EXP2 {
                  $$ = make_EAnd($1, $3);
                    YY_RESULT_EXP_= $$; }
  | EXP2 { $$ = $1; YY_RESULT_EXP_= $$; }
;
EXP2 : _SYMB_6 { $$ = make_ETrue();
                    YY_RESULT_EXP_= $$; }
  | _SYMB_5 { $$ = make_EFalse();
                    YY_RESULT_EXP_= $$; }
  | _IDENT_ { $$ = make_EVar($1);
                    YY_RESULT_EXP_= $$; }
  | _SYMB_3 EXP _SYMB_4 { $$ = $2;
                    YY_RESULT_EXP_= $$; }
;
ListEXP : /* empty */ { $$ = 0;
                 YY_RESULT_ListEXP_= $$; }
  | ListEXP EXP _SYMB_2 {
               $$ = make_ListEXP($2, $1);
                YY_RESULT_ListEXP_= $$; }
;
```

D. Pretty Printer

```
...
void ppEXP(EXP _p_, int _i_) {
  switch(_p_->kind) {
  case is_EOr:
    if (_i_ > 0) renderC(_L_PAREN);
    ppEXP(_p_->u.eor_.exp_1, 0);
    renderS("||");
    ppEXP(_p_->u.eor_.exp_2, 1);
    if (_i_ > 0) renderC(_R_PAREN);
    break;
  case is_EAnd:
    if (_i_ > 1) renderC(_L_PAREN);
    ppEXP(_p_->u.eand_.exp_1, 1);
    renderS("&&");
    ...
void ppListEXP(ListEXP listexp, int i) {
  while(listexp!= 0) {
    if (listexp->listexp_ == 0) {
      ppEXP(listexp->exp_, 0);
      renderC(';');
      listexp = 0;
    } else {
      ppEXP(listexp->exp_, 0);
      renderC(';');
      listexp = listexp->listexp_;
    }
  }
}
```


Figure 3.4: C source code fragments generated from Figure 3.1

for the user to work with. We are currently looking into ways to make the generated code more friendly through the use of macros or other methods.

3.8 Discussion

Productivity Gains

The source code of the Boolean expression grammar in Section 3.2.4 is 8 lines. The size of the generated code varies from 425 lines of Haskell/Happy/Alex to 1112 lines of C++/Bison/Flex. The generated code is not superfluously verbose, but similar to what would be written by hand by a programmer following Appel's methodology [1, 2, 3]. This amounts to a gain of coding effort by a factor of 50–100, which is comparable to the effort saved by, for instance, writing an LR parser in Bison instead of directly in C.[3] In addition to decreasing the number of lines, the single-source approach alleviates synchronization problems, both when creating and when maintaining a language.

The BNF Converter as a Teaching Tool

The BNF Converter has been used as a teaching tool in a fourth-year compiler course at Chalmers University in 2003 and 2004. The goal is, on the one hand, to advocate the use of declarative and portable language definitions, and on the other hand, to leave more time for back-end construction. The generated code follows the format recommended in Appel's text books [1, 2, 3], which makes is coherent to use the tool as a companion to those books. The results are encouraging: the lexer/parser part of the compiler was estimated only to be 25 % of the work at the lowest grade, and 10 % at the highest grade—at which point the student compiler had to include several back ends. This was far from the times when the parser was more than 50 % of a student compiler. About 50 % of the laboration groups use Haskell as implementation language, the rest using Java, C, or C++. In 2004, when the BNF Converter was available for all these languages, 16 groups of the 19 accepted ones used it in their assignment. The main discouraging factor were initial problems with Bison versions: older versions than 1.875 do compile the generated Bison file, but the parser fails with all input.

[3] In the present example, the Flex and Bison code generated by the BNF Converter is 172 lines, from which these tools generate 2600 line of C.

In Autumn 2003, the BNF Converter was also used in a second-year Chalmers course on Programming Languages. It is replacing the previously-used parser combinator libraries in Haskell. The main motivation at this level is to teach the correspondence between parsers and grammars, and to provide a high-level parser tool also for programmers who do not know Haskell.

One concern about using the BNF Converter was that students would not really learn parsing, but just to write grammars. However, students writing their parsers in YACC are equally isolated from the internals of LR parsing as those writing in LBNF. In fact, as learning the formalism takes less time in the case of LBNF, the teacher can allocate more time for explaining how an LR parser works.

Real-World Languages

Students in a compiler class usually implement toy languages. What about real-world languages? As an experiment, a complete LBNF definition of ANSI C, with [15] as reference, was written.[4] The result was a complete front-end processor for ANSI C, with the exception, mentioned in [15] of type definitions, which have to be treated with a preprocessor. The grammar has 229 LBNF rules and 15 token definitions (to deal with different numeral literals, such as octals and hexadecimals).

The BNF Converter has also been applied in an industrial application producing a compiler for a telecommunications protocol description language. [5]

Another real-world example is the object-oriented specification language OCL [24].[5] Finally, the BNF Converter itself is implemented by using modules generated from an LBNF grammar of the LBNF formalism.

A Case Study in Language Prototyping

A strong case for BNF Converter is the prototyping of new languages. It is easy to add and remove language features, and to test the updated language immediately. Since standard tools are used, the step from the prototype to a production-quality front end is small, typically involving some fine-tuning of the abstract syntax and the pretty printer. We have a large-scale experience of this in creating a new version of the language GF (Grammatical Framework, [21]).

[4] BSc thesis of Ulf Persson at Chalmers.

[5] Work by Kristofer Johannisson at Chalmers (private communication).

The main novelties added to GF were a module system added on top of the old GF language, and a lower-level language GFC, playing the role of "object code" generated by the GF compiler. The GF language has constructions mostly familiar from functional programming languages, and the size of the full grammar is similar to ANSI C; GFC is about half this size. We wrote the LBNF grammar from scratch, one motivation being to obtain reliable documentation of GF. This work took a few hours. We then used the skeleton file to translate the generated abstract syntax into the existing hand-written Haskell datatypes; in this way, we did not need to change the later phases of the existing compiler (apart from the changes due to new language features). In a couple of days, we had a new parser accepting all old GF files as well as files with the new language features. Working with later compilation phases suggested some changes in the new features, such as adding and removing type annotations. Putting the changes in place never required changing other things than the LBNF grammar and some clauses in the skeleton-based translator.

The development of GFC was different, since the language was completely new. The crucial feature was the symmetry between the parser and the pretty printer. The GF compiler generates GFC, but it also needs to parse GFC, so that it can use precompiled modules instead of source files. It was reassuring to know that the parser and the pretty printer completely matched. As a last step, we modified the rendering function of the GFC pretty printer so that it did not generate unnecessary spaces; GFC code is not supposed to be read by humans. This step initially created unparsable code (due to some necessary spaces having been omitted), which was another proof of the value of automatically generated pretty-printers.

In addition to the GF compiler written in Haskell, we have been working on GF-based applets ("gramlets") written in Java. These applications use precompiled GF. With the Java parser generated by the BNF Converter, we can guarantee that the GFC code generated by the Haskell pretty-printer can be read in by the Java application.

Related Work

The BNF Converter adds a level of abstraction to the YACC [14] tradition of compiler compilers, since it compiles a yet higher-level notation into notations on the level of YACC. Another system on this level up from YACC is Cactus [18], which uses an EBNF-like notation to generate front ends in Haskell and C. Unlike the BNF Converter, Cactus aims for completeness, and the notation is therefore more complex than LBNF. It is not possible

to extract a pretty printer from a Cactus grammar, and Cactus does not generate documentation.

The Zephyr definition language [23] defines a portable format for abstract syntax and translates it into SML, Haskell, C, C++, Java, and SGML, together with functions for displaying syntax trees. It does not support the definition of concrete syntax.

In general, compiler tools almost invariably opt for expressive power rather than declarativity and simplicity. The situation is different in linguistics, where the declarativity and *reversibility* (i.e. usability for both parsing and generation) of grammar formalisms are highly valued. A major example of this philosophy are Definite Clause Grammars (DCG) [20]. Since DCGs are usually implemented as an embedded language in Prolog, features of full Prolog are sometimes smuggled into DCG grammars; but this is usually considered harmful since it destroys declarativity.

3.9 Conclusions and Future Work

BNF Converter is a tool implementing the Labelled BNF grammar formalism (LBNF). Given that a programming language is "well-behaved", in a rather intuitive sense, an LBNF grammar is the only source that is needed to implement a front end for the language, together with matching LaTeX documentation. Since LBNF is purely declarative, the implementation can be generated in different languages: these currently include Haskell, Java, C++, and C, each with their standard parser and lexer tools. Depending on the tools, the size of the generated code is typically 50–100 times the size of the LBNF source.

The approach has proven to be useful both in teaching and in language prototyping. As for legacy real-world languages, complete definitions have so far been written for C and OCL. Often a language is almost definable, but has some exotic features that would require stronger tools. We have, however, opted to keep LBNF simple, at the expense of expressivity; and we believe that there are many good reasons behind a trend toward more and more well-behaved programming languages.

One frequent request has been a possibility to retain some of the position information in the abstract syntax tree, so that error messages from later compiler phases can be linked to the source code. This has been partly solved by extending the `token` pragma with the keyword `position` that enable position information to be retained in that particular token. However, further generalizations are needed at this point. Other requests are increased

control of the generated abstract syntax and some means of controlling the output of the pretty-printing.

3.10 BNF Converter in Year 2007

Since this technical report has been written, BNFC has been actively developed and many people have contributed. This section will give a brief overview of what is new.

3.10.1 New Back Ends

- Kristofer Johannisson has added a back end for Objective Caml.
- Johan Broberg has added a back end for C#.
- Björn Bringert has updated the Java back end to support Java 1.5. The old Java back end is still available.
- Aarne Ranta has updated the C++ back end with STL support. The old C++ back end is still available.

3.10.2 Natural Language Support

Two new features have been added to BNFC that support the treatment of natural languages.

The first new feature is ambiguous parsing in the Haskell back end, which has been added by Paul Callaghan through his work on generalized LR parsing in Happy [19]. Since natural language is know to be ambiguous, this has been an important addition to allow processing of natural languages.

The second new feature is *profiles* in the Haskell back end, added by Aarne Ranta. Profiles allow a grammar written in Grammatical Framework (GF) [22], a much more powerful grammar formalism, to be translated into a LBNF grammar augmented with profiles. This translation allows the creation of a stand-alone program from a GF grammar. For more details, see the LBNF report [16].

3.10.3 Layout Support

Aarne Ranta has added support for layout in the Haskell back end, i.e. the language feature where the indentation of a program is used for grouping. The details of the layout support is given in the LBNF report [16].

3.10.4 Support for Definitions

Ulf Norell has added support for a feature that we refer to as *defines*. Defines are used to add syntactic sugar to a language without it being represented in the abstract syntax.

Define Example

Here we provide an example[6] of the use of defines in a fragment of an imperative language. We start with the core statement language, expressed in normal LBNF.

```
Assign. Stm ::= Ident "=" Exp ;
Block.  Stm ::= "{" [Stm] "}" ;
While.  Stm ::= "while" "(" Exp ")" Stm ;
If.     Stm ::= "if" "(" Exp ")" Stm "else" Stm "endif" ;
```

We now want to have some syntactic sugar. Note that the labels for these rules all start with a lowercase letter, indicating that they correspond to defined functions rather than nodes in the abstract syntax tree.

```
if.     Stm ::= "if" "(" Exp ")" Stm "endif" ;
for.    Stm ::= "for" "(" Stm ";" Exp ";" Stm ")" Stm ;
inc.    Stm ::= Ident "++" ;
```

Functions are defined using the 'define' keyword. Definitions have the form `'define f x1 .. xn = e'` where `e` is an expression on applicative form using labels, other defined functions, lists and literals.

```
define if e s       = If e s (Block []) ;
define for i c s b  = Block [i, While c (Block [b, s])] ;
define inc x        = Assign x (EOp (EVar x) Plus (EInt 1)) ;
```

3.10.5 Multi Views

A *multi view*[7] in BNFC is an experimental feature in the Haskell back end that allows languages to be defined in parallel, sharing abstract syntax.

An example of the syntax for multi views is provided here.

[6] The example together with the explanations is provided by U. Norell.

[7] Multi views have been designed by M. Forsberg, B. Nordström, and A. Ranta, and implemented by A. Ranta.

```
views C, JVM ;

C: EAdd. Exp  ::= Exp  "+" Exp1 ;
C: EMul. Exp1 ::= Exp1 "*" Exp2 ;
C: EInt. Exp2 ::= Integer ;
C: coercions Exp 2 ;

JVM: EAdd. Exp ::= Exp Exp "iadd" ";" ;
JVM: EMul. Exp ::= Exp Exp "imul" ";" ;
JVM: EInt. Exp ::= "bipush" Integer ";" ;
```

There are a new keyword `views`, which is used to declare the language names. Furthermore, the rules of the LBNF are prefixed with a language name followed by a colon (`LANG:`), which defines syntax of the languages in a multi view. A multi view is correct if all languages in the multi view have identical abstract syntax.

3.10.6 Haskell GADT Support

Björn Bringert and Aarne Ranta have implemented support for generalized algebraic data types (GADT) in the Haskell back end. The details can be found in their article *A Pattern for Almost Compositional Functions* [4].

Bibliography

[1] A. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[2] A. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[3] A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[4] Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 216–226, New York, NY, USA, 2006. ACM Press.

[5] C. Däldborg and O. Noreklint. ASN.1 Compiler. Master's Thesis, Department of Computing Science, Chalmers University of Technology, 2004.

[6] C. Dornan. Alex: a Lex for Haskell Programmers, 1997. http://www.cs.ucc.ie/dornan/alex.html.

[7] C. Dornan. JLex: A Lexical Analyzer Generator for Java, 2000. http://www.cs.princeton.edu/~appel/modern/java/JLex/.

[8] M. Forsberg, P. Gammie, M. Pellauer, and A. Ranta. BNF Converter site. Program and documentation, http://www.cs.chalmers.se/~markus/BNFC/, 2004.

[9] M. Forsberg and A. Ranta. Labelled BNF: a high-level formalism for defining well-behaved programming languages. *Proceedings of the Estonian Academy of Sciences: Physics and Mathematics*, 52:356–377, 2003. Special issue on programming theory edited by J. Vain and T. Uustalu.

[10] Free Software Foundation. Bison - GNU Project, 2003. http://www.gnu.org/software/bison/bison.html.

[11] Free Software Foundation. Flex - GNU Project, 2003. http://www.gnu.org/software/flex/flex.html.

[12] E. Gamma, R. Hehn, R. Johnson, and J. Viissides. *Design Patterns*. Addison Wesley, 1995.

[13] Scott E. Hudson. CUP Parser Generator for Java, 1999. http://www.cs.princeton.edu/~appel/modern/java/CUP/.

[14] S. C. Johnson. Yacc — yet another compiler compiler. Technical Report CSTR-32, AT & T Bell Laboratories, Murray Hill, NJ, 1975.

[15] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1988. 2nd edition.

[16] M. Forsberg and A. Ranta. The Labelled BNF Grammar Formalism, For BNF Converter Version 2.2, Feb. 11, 2005. http://www.cs.chalmers.se/~markus/BNFC/LBNF-report.pdf.

[17] S. Marlow. Happy, The Parser Generator for Haskell, 2001. http://www.haskell.org/happy/.

[18] N. Martinsson. Cactus (Concrete- to Abstract-syntax Conversion Tool with Userfriendly Syntax) . Master's Thesis in Computer Science, 2001. http://www.mdstud.chalmers.se/~md6nm/cactus/.

[19] P. Callaghan. Ambiguous Parsing with Happy, 2007. Under Consideration for Publication in J. Functional Programming. Available at: `www.dur.ac.uk/p.c.callaghan/happy-glr/callaghan-glr.ps.gz`.

[20] F. Pereira 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.

[21] A. Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 2004.

[22] A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

[23] Daniel C. Wang, Andrew W. Appel Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. *Proceedings of the Conference on Domain-Specific Languages*, 1997.

[24] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Addison-Wesley, 1999.

Part II

Functional Morphology

Chapter 4

Functional Morphology

Authors:
Markus Forsberg and Aarne Ranta
Department of Computing Science
Chalmers University of Technology
and the University of Gothenburg
{*markus, aarne*}*@cs.chalmers.se*

Paper published:
ICFP'04, Proceedings of the Ninth ACM SIGPLAN International Conference of Functional Programming, September 19-21, 2004, Snowbird, Utah

Abstract

This paper presents a methodology for implementing natural language morphology in the functional language Haskell. The main idea behind is simple: instead of working with untyped regular expressions, which is the state of the art of morphology in computational linguistics, we use finite functions over hereditarily finite algebraic data types. The definitions of these data types and functions are the language-dependent part of the morphology. The language-independent part consists of an untyped dictionary format which is used for synthesis of word forms, and a decorated trie, which is used for analysis.

Functional Morphology builds on ideas introduced by Huet in his computational linguistics toolkit Zen, which he has used to implement the morphology of Sanskrit. The goal has been to make it easy for linguists, who are

not trained as functional programmers, to apply the ideas to new languages. As a proof of the productivity of the method, morphologies for Swedish, Italian, Russian, Spanish, and Latin have already been implemented using the library. The Latin morphology is used as a running example in this article.

4.1 Introduction

This paper presents a systematic way of developing natural language morphologies in a functional language. We think of functions and linguistic abstractions as strongly related in the sense that given a linguistic abstraction, it is, in most cases, natural and elegant to express it as a function. We feel that the methodology presented is yet another proof of this view.

An implementation of the methodology is presented, named *Functional Morphology* [9]. It can be viewed as an embedded domain-specific language in Haskell. Its basis are two type classes: `Param`, which formalizes the notion of a parameter type, and `Dict`, which formalizes the notion of a part of speech as represented in a dictionary. For these classes, Functional Morphology gives generic Haskell functions for morphological analysis and synthesis, as well as generation of code that presents the morphology in other formats, including Xerox Finite State Tools and relational databases.

The outline of the paper is the following: The morphology task is described in section 4.2, and then the contemporary approaches are discussed in section 4.3. The main part of the paper, section 4.4, focuses on describing the *Functional Morphology* library. We conclude in sections 4.5 and 4.6 with some results and a discussion.

4.2 Morphology

A morphology is a systematic description of words in a natural language. It describes a set of relations between words' *surface forms* and *lexical forms*. A word's surface form is its graphical or spoken form, and the lexical form is an analysis of the word into its *lemma* (also known as its *dictionary form*) and its *grammatical description*. This task is more precisely called *inflectional* morphology.

Yet another task, which is outside the scope of this paper, is *derivational* morphology, which describes how to construct new words in a language.

A clarifying example is the English word *functions'*. The graphical form *functions'* corresponds to the surface form of the word. A possible lexical

form for the word *functions'* is *function +N +Pl +Gen*. From the analysis it can be read that the word can be analyzed into the lemma *function*, and the grammatical description *noun*, in *plural*, *genitive* case.

A morphological description has many applications, to mention a few: *machine translation*, *information retrieval*, *spelling and grammar checking* and *language learning*.

A morphology is a key component in machine translation, assuming that the aim is something more sophisticated than string to string translation. The grammatical properties of words are needed to handle linguistic phenomena such as *agreement*. Consider, for example, the subject-verb agreement in English — *Colorless green ideas* ***sleep*** *furiously*, not **Colorless green ideas* ***sleeps*** *furiously*.

In information retrieval, the use of a morphology is most easily explained through an example. Consider the case when you perform a search in a text for the word *car*. In the search result, you would also like to find information about *cars* and *car's*, but no information about *carts* and *careers*. A morphology is useful to do this kind of distinctions.

4.3 Implementations of Morphology

4.3.1 Finite State Technology

The main contemporary approach within computational morphology is finite state technology - the morphology is described with a regular expression [18, 26, 16, 2] that is compiled to a finite state transducer, using a finite state tool. Some of the tools available are: the commercial tool *XFST* [29], developed at Xerox, van Noord's [28] *finite state automata utilities*, AT&T's [19] *FSM library* and Forsberg's [7] *FST Studio*.

Finite state technology is a popular choice since finite state transducers provide a compact representation of the implemented morphology, and the lookup time is close to constant in the size of the lexicon.

Finite state technology is based on the notion of a *regular relation*. A regular relation is a set of n-tuples of words. Regular languages are a special case, with $n = 1$. Morphology tools such as XFST work with 2-place relations. They come with an extended regular expression notation for easy manipulation of symbol and word pairs. Such expressions are compiled into *finite-state transducers*, which are like finite-state automata, but their arcs are labelled by pairs of symbols rather than just symbols. Strings consisting of the first components of these pairs are called the *upper language* of the transducer, and strings consisting of the second components are called

the *lower language*. A transducer is typically used so that the upper language contains structural descriptions of word forms and the lower language contains the forms themselves.

A trivial example of a regular relation is the description of the inflection of three English nouns in number. The code is XFST source code, where the `|` is the union operator, and `.x.` is the cross product of the strings. Concatenation is expressed by juxtaposition.

```
NOUN = "table" | "horse" | "cat"

INFL = NOUN .x. "Sg" | NOUN "s" .x. "Pl"
```

If a transducer is compiled from this regular relation, and applied *upward* with the string `"tables"`, it will return {"Pl"}. If the built transducer is applied *downward* with the string `"Sg"`, it will return {"table", "horse", "cat"}.

One problem with finite-state tranducers is that cycles (corresponding to Kleene stars in regular expressions), can appear anywhere in them. This increases the complexity of compilation so that it can be exponential. Compiling a morphological description to a transducer has been reported to last several days, and sometimes small changes in the source code can make a huge difference. Another problem is that transducers cannot generally be made deterministic for sequences of symbols (they are of course deterministic for sequences of symbol pairs). This means that analysis and synthesis can be worse than linear in the size of the input.

4.3.2 The Zen Linguistic Toolkit

Huet has used the functional language Caml to build a Sanskrit dictionary and morphological analysis and synthesis. [13]. He has generalized the ideas used for Sanskrit to a toolkit for computational linguistics, Zen [14]. The key idea is to exploit the expressive power of a functional language to define a morphology on a high level, higher than regular expressions. Such definitions are moreover safe, in the sense that the type checker guarantees that all words are defined correctly as required by the definitions of different parts of speech.

The analysis of words in Zen is performed by using *tries*. A trie is a special case of a finite-state automaton, which has no cycles. As Huet points out, the extra power added by cycles is not needed for the morphological description inside words, but, at most, between words. This extra power is needed in languages like Sanskrit where word boundaries are not visible and adjacent words can affect each other (this phenomenon is known as *sandhi*).

It is also needed in languages like Swedish where compound words can be formed almost *ad libitum*, and words often have special forms used in compounds. Compositions of tries, with cycles possible only on word boundaries, have a much nicer computational behaviour than full-scale transducers.

4.3.3 Grammatical Framework

The Grammatical Framework GF [25] is a special-purpose functional language for defining grammars, including ones for natural languages. One part of a grammar is a morphology, and therefore GF has to be capable of defining morphology. In a sense, this is trivial, since morphology requires strictly less expressive power than syntax (regular languages as opposed to context-free languages and beyond). At the same time, using a grammar formalism for morphology is overkill, and may result in severely suboptimal implementations.

One way to see the Functional Morphology library described in this paper is as a fragment of GF embedded in Haskell. The `Param` and `Dict` classes correspond to constructs that are hard-wired in GF: *parameter types* and *linearization types*, respectively. Given this close correspondence, it is no wonder that it is very easy to generate GF code from a Functional Morphology description. On the other hand, the way morphological analysis is implemented efficiently using tries has later been adopted in GF, so that the argument on efficiency is no longer so important. Thus one can see the morphology fragment of GF as an instance of the methodology of Functional Morphology. However, complicated morphological rules (such as stem-internal vowel changes) are easier to write in Haskell than in GF, since Haskell provides more powerful list and string processing than GF.

4.4 Functional morphology

4.4.1 Background

The goal of our work is to provide a freely available open-source library that provides a high level of abstraction for defining natural language morphologies. The examples used in this article are collected from Latin morphology. Our Latin morphology is based on the descriptions provided by [20, 6, 17, 3].

Our work is heavily influenced by Huet's functional description of Sanskrit [13] and his Zen Toolkit [14]. The analyzer provided by Functional Morphology can be seen as a Haskell version of Huet's "reference implementation" in Caml. At the same time, we aim to provide a language-

independent high-level *front-end* to those tools that makes it possible to define a morphology with modest training in functional programming.

The idea of using an embedded language with a support for code generation is related to Claessen's hardware description language Lava [5], which is compiled into VHDL. For the same reasons as it is important for Lava to generate VHDL—the needs of the main stream community—we generate regular expressions in the XFST and LEXC formats.

Functional Morphology is based on an old idea, which has been around for over 2000 years, that of *inflection tables*. An inflection table captures an inflectional regularity in a language. A morphology is a set of tables and a dictionary. A dictionary consists of lemmas, or dictionary forms, tagged with pointers to tables.

An inflection table displaying the inflection of regular nouns in English, illustrated with the lemma *function*, is shown below.

	Case	
Number	Nominative	Genitive
Singular	*function*	*function's*
Plural	*functions*	*functions'*

Different ways of describing morphologies were identified by Hockett [10] in 1950's. The view of a morphology as a set of inflection tables he calls *word and paradigm*. The paradigm is an inflection table, and the word is an example word that represents a group of words with the same inflection table.

In a sense, the research problem of describing inflectional morphologies is already solved: how to fully describe a language's inflectional morphology in the languages we studied is already known. But the are still problematic issues which are related to the size of a typical morphology. A morphology covering a dictionary of a language, if written out in full form lexicon format, can be as large as 1-10 million words, each tagged with their grammatical description.

The size of the morphology demands two things: first, we need an efficient way of describing the words in the morphology, generalize as much as possible to minimize the effort of implementing the morphology, and secondly, we need a compact representation of the morphology that has an efficient lookup function.

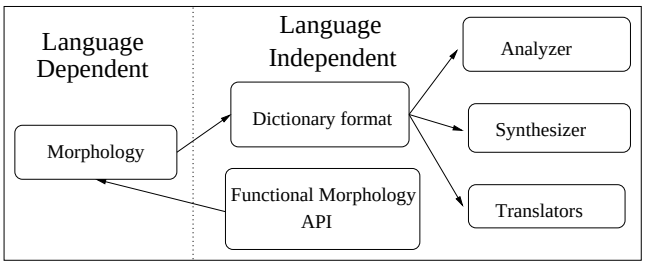


Figure 4.1: Functional Morphology system overview

4.4.2 Methodology

The methodology suggests that paradigms, i.e. inflection tables, should be defined as finite functions over an enumerable, hereditarily finite, algebraic data type describing the parameters of the paradigm. These functions are later translated to a *dictionary*, which is a language-independent datastructure designed to support analyzers, synthesizers, and generation of code in other formats than Haskell.

All parameter types are instances of the `Param` class, which is an extension of the built-in `Enum` and `Bounded` class, to be able to define enumerable, finite types over hierarchical data types.

Parts of speech are modelled by instances of the class `Dict`, which automate the translation from a paradigm to the `Dictionary` type.

4.4.3 System overview

A Functional Morphology system consists of two parts, one language dependent part, and one language independent part, illustrated in figure 4.1.

The language dependent part is what the morphology implementor has to provide, and it consists of a *type system*, an *inflection engine* and a *dictionary*. The type system gives all word classes and their inflection and inherent parameters, and instances of the `Param` class and the `Dict` class. The inflection machinery defines all valid inflection tables, i.e. all *paradigms*, as finite functions. The dictionary lists all words in dictionary form with its paradigm in the language.

Defining the type system and the inflection machinery can be a demanding task, where you not only need to be knowledgeable about the language in question, but also have to have some understanding about functional programming. The libraries provided by Functional Morphology simplifies this step.

However, when the general framework has been defined, which is actually a new library built on top of ours, it is easy for a lexicographer to add new words, and this can be done with limited or no knowledge about functional programming. The lexicographer does not even have to be knowledgeable about the inner workings of a morphology, it is sufficient that she knows the inflectional patterns of words in the target language.

4.4.4 Technical details

Parameter types

In grammars, words are divided into classes according to similarity, such as having similar inflection patterns, and where they can occur and what role they play in a sentence. Examples of classes, the *part of speech*, are nouns, verbs, adjectives and pronouns.

Words in a class are attributed with a set of parameters that can be divided into two different kinds of categories: *inflectional* parameters and *inherent* parameters.

Parameters are best explain with an example. Consider the Latin noun *causa* (Eng. *cause*). It is inflected in number and case, i.e. number and case are the inflectional parameters. It also has a gender, which is an inherent parameter. The inflection of *causa* in plural nominative is *causae*, but it *has* feminine gender.

These parameters are described with the help of Haskell's data types. For example, to describe the parameters for Latin noun, the types `Gender`, `Case` and `Number` are introduced.

```
data Gender = Feminine  |
              Masculine |
              Neuter
 deriving (Show,Eq,Enum,Ord,Bounded)

data Case   = Nominative |
              Genitive   |
              Dative     |
              Accusative |
              Ablative   |
              Vocative
 deriving (Show,Eq,Enum,Ord,Bounded)

data Number = Singular |
              Plural
 deriving (Show,Eq,Enum,Ord,Bounded)
```

The inflectional parameter types `Case` and `Number` are combined into one type, `NounForm`, that describes all the inflection forms of a noun. Note that `Gender` is not part of the inflection types, it is an inherent parameter.

```
data NounForm = NounForm Number Case
 deriving (Show,Eq)
```

The parameter types of a language are language-dependent. A class `Param` for parameters has been defined, to make it possible to define language independent methods, i.e. implement generic algorithms.

```
  class (Eq a, Show a) ⇒ Param a where
   values   :: [a]
   value    :: Int → a
   value0   :: a
   prValue  :: a → String
   value n  = values !! n
   value0   = value 0
   prValue  = show
```

The most important method — the only one not defined by default — is `values`, giving the complete list of all objects in a `Param` type. The parameter types are, in a word, hereditarily finite data types: not only enumerated types but also types whose constructors have arguments of parameter types.

An instance of `Param` is easy to define for bounded enumerated types by the function `enum`.

```
  enum :: (Enum a, Bounded a) ⇒ [a]
  enum = [minBound .. maxBound]
```

The parameters of Latin nouns are made an instance of `Param` by the following definitions:

```
instance Param Gender  where values = enum
instance Param Case    where values = enum
instance Param Number  where values = enum
instance Param NounForm where
 values =
  [NounForm n c | n <- values ,
                  c <- values]
 prValue (NounForm n c) =
   unwords $ [prValue n, prValue c]
```

The default definition for `prValue` has been redefined for `NounForm` to remove the `NounForm` constructor. Usually, a more sophisticated printing scheme is preferred, using a particular *tag set*, i.e. adopting a standard for describing the parameters of words.

Latin nouns can now be defined as a finite function, from a NounForm to a `String`. The choice of `String` as a return type will be problematized in section 4.4.4 and another type, `Str`, will be introduced.

```
type Noun = NounForm -> String
```

More generally, a finite function in Functional Morphology, is a function `f` from a parameter type `P` to strings.

```
f :: P -> String
```

Note that the finite functions have a single argument. This is, however, not a limitation, because we can construct arbitrarily complex single types with tuple-like constructors.

Type hierarchy

A naive way of describing a class of words is by using the cross product of all parameters. This would in many languages lead to a serious over-generation of cases that do not exist in the language.

An example is the Latin verbs, where the cross product of the inflection parameters generates 1260 forms (three persons, two numbers, six tenses, seven moods and five cases[1]), but only 147 forms actually exist, which is just about a ninth of 1260.

This problem is easily avoided in a language like Haskell that has algebraic data types, where data types are not only enumerated, but also complex types with constructors that have type parameters as arguments.

The type system for Latin verbs can be defined with the data types below, that exactly describes the 147 forms that exist in Latin verb conjugation:

```
data VerbForm =
    Indicative Person Number Tense Voice   |
    Infinitive TenseI Voice                |
    ParticiplesFuture  Voice               |
```

[1]The verb inflection in case only appears in the gerund and supine mood, and only some of the six cases are possible.

```
  ParticiplesPresent                      |
  ParticiplesPerfect                      |
  Subjunctive Person Number TenseS Voice  |
  ImperativePresent Number Voice          |
  ImperativeFutureActive  Number PersonI  |
  ImperativeFuturePassiveSing PersonI     |
  ImperativeFuturePassivePl               |
  GerundGenitive                          |
  GerundDative                            |
  GerundAcc                               |
  GerundAbl                               |
  SupineAcc                               |
  SupineAblative
```

This representation gives a correct description of what forms exist, and it is hence linguistically more satisfying than a cross-product of features. The type system moreover enables a completeness check to be performed.

Tables and finite functions

The concept of inflection tables corresponds intuitively, in a programming language, to a list of pairs. Instead of using list of pairs, a functional counterpart of a table — a finite function could be used, i.e. a finite set of pairs defined as a function.

To illustrate the convenience with using finite functions instead of tables, consider the inflection table of the Latin word *rosa* (Eng. *rose*):

	Singular	**Plural**
Nominative	*rosa*	*rosae*
Vocative	*rosa*	*rosae*
Accusative	*rosam*	*rosas*
Genitive	*rosae*	*rosarum*
Dative	*rosae*	*rosis*
Ablative	*rosa*	*rosis*

The word has two inflection parameters, case and number, that, as discussed in section 4.4.4, can be described in Haskell with algebraic data types.

```
data Case     = Nominative | Vocative |
                Accusative | Genitive |
                Dative     | Ablative
data Number   = Singular   | Plural
data NounForm = NounForm Case Number
```

The inflection table can be viewed as a list of pairs, where the first component of a pair is an inflection parameter, and the second component is an inflected word. The inflection table of *rosa* is described, in the definition of `rosa` below, as a list of pairs.

```
rosa :: [(NounForm,String)]
rosa =
 [
  (NounForm Singular Nominative,"rosa"),
  (NounForm Singular Vocative,"rosa"),
  (NounForm Singular Accusative,"rosam"),
  (NounForm Singular Genitive,"rosae"),
  (NounForm Singular Dative,"rosae"),
  (NounForm Singular Ablative,"rosa"),
  (NounForm Plural Nominative,"rosae"),
  (NounForm Plural Vocative,"rosae"),
  (NounForm Plural Accusative,"rosas"),
  (NounForm Plural Genitive,"rosarum"),
  (NounForm Plural Dative,"rosis"),
  (NounForm Plural Ablative,"rosis")
 ]
```

The type `NounForm` is finite, so instead of writing these kinds of tables, we can write a finite function that describes this table more compactly. We could even go a step further, and first define a function that describes all nouns that inflects in the same way as the noun *rosa*, i.e. defining a paradigm.

```
rosaParadigm :: String → Noun
rosaParadigm rosa (NounForm n c) =
 let rosae = rosa ++ "e"
     rosis = init rosa ++ "is"
  in
   case n of
    Singular → case c of
                  Accusative → rosa + "m"
                  Genitive   → rosae
                  Dative     → rosae
                  _          → rosa
    Plural   → case c of
                  Nominative → rosae
                  Vocative   → rosae
                  Accusative → rosa ++ "s"
                  Genitive   → rosa ++ "rum"
                  _          → rosis
```

It may seem that not much has been gained, except that the twelve cases have been collapsed to nine, and we have achieved some sharing of `rosa` and `rosae`.

However, the gain is clearer when defining the paradigm for *dea* (Eng. *goddess*), that inflects in the same way, with the exception of two cases, plural dative and ablative.

```
dea :: Noun
dea nf =
  case nf of
   NounForm Plural Dative   → dea
   NounForm Plural Ablative → dea
   _                        → rosaParadigm dea nf
 where dea = "dea"
```

Given the paradigm of *rosa*, `rosaParadigm`, we can describe the inflection tables of other nouns in the same paradigm, such as *causa* (Eng. *cause*) and *barba* (Eng. *beard*).

```
rosa, causa, barba :: Noun
rosa  = rosaParadigm "rosa"
causa = rosaParadigm "causa"
barba = rosaParadigm "barba"
```

Turning a function into a table

The most important function of Functional Morphology is `table`, that translates a finite function into a list of pairs. This is done by ensuring that the parameter type is of the `Param` class, which enables us to generate all forms with the class function `values`.

```
table :: Param a ⇒ (a → Str) → [(a,Str)]
table f = [(v, f v) | v ← values]
```

A function would only be good for generating forms, but with `table`, the function can be compiled into lookup tables and further to tries to perform analysis as well.

String values

The use of a single string for representing a word is too restricted, because words can have *free variation*, i.e. that two or more words have the same

morphological meaning, but are spelled differently. Yet another exception is *missing forms*, some inflection tables may have missing cases.

Free variation exists in the Latin noun *domus* (Eng. *home*) in singular dative, *domui* or *domo*, in plural accusative, *domus* or *domos*, and in plural genitive, *domuum* or *domorum*.

Missing forms appear in the Latin noun *vis* (Eng. *violence, force*), a noun that is *defective* in linguistic terms.

	Singular	**Plural**
Nominative	*vis*	*vires*
Vocative	-	*vires*
Accusative	*vim*	*vires*
Genitive	-	*virium*
Dative	-	*viribus*
Ablative	*vi*	*viribus*

These two observations lead us to represent a word with the abstract type `Str`, which is simply a list of strings. The empty list corresponds to the missing case.

```
type Str = [String]
```

The `Str` type is kept abstract, to enable a change of the representation. The abstraction function is called `strings`.

```
strings :: [String] → Str
string = id
```

The normal case is singleton lists, and to avoid the increased complexity of programming with lists of strings, we provide the `mkStr` function, that promotes a `String` to a `Str`.

```
mkStr :: String → Str
mkStr = (:[])
```

The description of missing cases is handled with the constant `nonExist`, which is defined as the empty list.

```
nonExist :: Str
nonExist = []
```

The inflection table of *vis* can be described with the function `vis` below.

```
vis :: Noun
vis (NounForm n c) =
  case n of
   Singular → case c of
                 Nominative → mkStr $ vi ++ "s"
                 Accusative → mkStr $ vi ++ "m"
                 Ablative   → mkStr vi
                 _          → nonExist
   Plural   → mkStr $
                case c of
                 Genitive   → vir ++ "ium"
                 Dative     → viribus
                 Ablative   → viribus
                 _          → vir ++ "es"
 where vi = "vi"
       vir = vi ++ "r"
       viribus = vir ++ "ibus"
```

String operations

Functional Morphology provides a set of string operation functions that captures common phenomena in word inflections. Some of them are listed below to serve as examples.

The string operations cannot be quite complete, and a morphology implementer typically has to write some functions of her own, reflecting the peculiarities of the target language. These new functions can be supplied as an extended library, that will simplify the implementation of a similar language. The goal is to make the library so complete that linguists with little knowledge of Haskell can find it comfortable to write morphological rules without recourse to full Haskell.

Here is a sample of string operations provided by the library.

The Haskell standard functions `take` and `drop` take and drop prefixes of words. In morphology, it is much more common to consider *suffixes*. So the library provides the following dual versions of the standard functions:

```
tk :: Int → String → String
tk i s = take (max 0 (length s - i)) s

dp :: Int → String → String
dp i s = drop (max 0 (length s - i)) s
```

It is a common phenomenon that, if the last letter of a word and the first letter of an ending coincide, then one of them is dropped.

```
(+?) :: String → String → String
s +? e = case (s,e) of
          (_:_,c:cs) | last s == c → s ++ cs
          _ → s ++ e
```

More generally, a suffix of a word may be dependent of the last letter of its stem.

```
ifEndThen :: (Char → Bool) → String → String
                           → String → String
ifEndThen cond s a b = case s of
                        _:_ | cond (last s) → a
                        _ → b
```

A more language dependent function, but interesting because it is difficult to define on this level of generality with a regular expression, is the *umlaut* phenomenon in German, i.e. the transformation of a word's stem vowel when inflected in plural.

```
findStemVowel :: String → (String, String, String)
findStemVowel sprick =
             (reverse rps, reverse i, reverse kc)
  where (kc, irps) = break isVowel $ reverse sprick
        (i, rps)   = span  isVowel $ irps

umlaut :: String → String
umlaut man = m ++ mkUm a ++ n
  where
   (m,a,n) = findStemVowel man
   mkUm v = case v of
             "a" → "ä"
             "o" → "ö"
             "u" → "u"
             "au" → "äu"
             _   → v
```

The plural form of *Baum*, can be describe with the function `baumPl`.

```
baumPl :: String → String
baumPl baum = umlaut baum ++ "e"
```

Applying the function `baumPl` with the string `"Baum"` computes to the correct plural form `"Bäume"`.

Obviously, the function *umlaut* is a special case of a more general *vowel alternation* function, that is present in many language, for instance, in English in the thematic alternation of verbs such as *drink-drank-drunk*:

```
vowAltern :: [(String,String)] → String → String
vowAltern alts man = m ++ a' ++ n
  where
   (m,a,n) = findStemVowel man
   a' = maybe a id $ lookup a alts
```

A general lesson from vowel alternations is that words are not just strings, but data structures such as tuples.[2] If regular expressions are used, these data structures have to be encoded as strings with special characters used as delimiters, which can give rise to strange errors since there is no type checking.

Exceptions

Exceptions are used to describe paradigms that are similar to another paradigm, with the exception of one or more case. That is, instead of defining a completely new paradigm, we use the old definition only marking what is different. This is not only linguistically more satisfying, it saves a lot of work. Four different kinds of exceptions, `excepts`, `missing`, `only` and `variants`, are listed below.

The exception `excepts`, takes a finite function, or a paradigm in other words, and list of exceptions, and forms a new finite function with exceptions included.

```
excepts :: Param a ⇒ (a → Str) → [(a,Str)] → (a → Str)
excepts f es p = maybe (f p) id $ lookup p es
```

The paradigm of *dea* defined in section 4.4.4 can be described with the function `dea` using the exception `excepts`.

```
dea :: Noun
dea =
  (rosaParadigm dea) 'excepts'
    [(NounForm Plural c, dea) | c <- [Dative, Ablative]]
 where dea = "dea"
```

The exception functions `missing` and `only` are used to express missing cases in a table; `missing` enumerates the cases with missing forms, and `only` is used for highly defective words, where it is easier to enumerate the cases that actually exists.

[2] E.g. in Arabic, triples of consonants are a natural way to represent the so-called roots of words.

```
missing :: Param a ⇒ (a → Str) → [a] → (a → Str)
missing f as = excepts f [(a,nonExist) | a ← as]

only :: Param a ⇒ (a → Str) → [a] → (a → Str)
only f as = missing f [a | a ← values, notElem a as]
```

The paradigm of *vis* described in section 4.4.4, can be described with the `only` exception and the paradigm of *hostis* (Eng. enemy).

```
vis :: Noun
vis =
 (hostisParadigm "vis") 'missing'
  [
  NounForm Singular c | c <- [Vocative, Genitive, Dative]
  ]
```

An often occurring exception is additional variants, expressed with the function `variants`. That is, that a word is in a particular paradigm, but have more than one variant in one or more forms.

```
variants :: Param a ⇒ (a → Str) → [(a,String)] →
                                        (a → Str)
variants f es p =
  maybe (f p) (reverse . (: f p)) $ lookup p es
```

Dictionary

The `Dictionary` type is the core of Functional Morphology, in the sense that the morphology description denotes a `Dictionary`. The `Dictionary` is a language-independent representation of a morphology, that is chosen to make generation to other formats easy.

A `Dictionary` is a list of `Entry`, where an `Entry` corresponds to a specific dictionary word.

```
type Dictionary = [Entry]
```

An `Entry` consists of the dictionary word, the part of speech (category) symbol, a list of the inherent parameters, and the word's, lacking a better word, untyped inflection table.

```
type Dictionary       = [Entry]
type Entry = (Dictionary_Word, Category,
              [Inherent], Inflection_Table)
```

```
type Dictionary_Word  = String
type Category         = String
type Inherent         = String
type Parameter        = String
type Inflection_Table = [(Parameter,(Attr,Str))]
```

The `Attr` type and definitions containing this type concerns the handling of composite forms, that will be explained later in section 4.4.6.

To be able to generate the `Dictionary` type automatically, a class `Dict` has been defined. Only composite types, describing the inflection parameters of a part of speech, should normally be an instance of the `Dict` class.

```
class Param a ⇒ Dict a where
  dictword :: (a → Str) → String
  category :: (a → Str) → String
  defaultAttr :: (a → Str) → Attr
  attrException :: (a → Str) → [(a,Attr)]
  dictword f = concat $ take 1 $ f value0
  category = const "Undefined"
  defaultAttr = const atW
  attrException = const []
```

Note that all class functions have a default definition, but usually we have to at least give a definition of `category`, that gives the name of the part of speech of a particular parameter type. It's impossible to give a reasonable default definition of `category`; it would require that we have types as first class objects.

It may be surprising that `category` and `defaultAttr` are higher-order functions. This is simply a type hack that forces the inference of the correct class instance without the need to provide an object of the type. Normally, the function argument is an inflection table (cf. the definition of `entryI` below).

The most important function defined for types in `Dict` is `entryI`, which, given a paradigm and a list of inherent features, creates an `Entry`. However, most categories lack inherent features, so the function `entry` is used in most cases, with an empty list of inherent features.

```
entryI :: Dict a ⇒ (a → Str) → [Inherent] → Entry
entryI f ihs = (dictword f, category f, ihs, infTable f)

entry  :: Dict a ⇒ (a → Str) → Entry
entry f = entryI f []
```

Returning to the noun example, `NounForm` can be defined as an instance of the class `Dict` by giving a definition of the `category` function.

```
instance Dict NounForm
  where category _ = "Noun"
```

Given that `NounForm` is an instance of the `Dict` class, a function `noun` can be defined, that translates a `Noun` into an dictionary entry, including the inherent parameter `Gender`, and a function for every gender.

```
noun :: Noun → Gender → Entry
noun n g = entryI n [prValue g]

masculine :: Noun → Entry
masculine n = noun n Masculine

feminine :: Noun → Entry
feminine n = noun n Feminine

neuter :: Noun → Entry
neuter n = Noun n Neuter
```

Finally, we can define a set of *interface functions* that translates a dictionary word into a dictionary entry: `d2servus` (Eng. *servant, slave*), `d1puella` (Eng. *girl*) and `d2donum` (Eng. *gift, present*).

```
d2servus :: String -> Entry
d2servus = masculine . decl2servus

d1puella :: String -> Entry
d1puella = feminine . decl1puella

d2donum :: String -> Entry
d2donum s = neuter . decl2donum
```

Given these interface function, a dictionary with words can be created. Note that the function `dictionary` is an abstraction function that is presently defined as `id`.

```
latinDict :: Dictionary
latinDict =
 dictionary $
  [
   d2servus  "servus",
```

```
   d2servus  "somnus",
   d2servus  "amicus",
   d2servus  "animus",
   d2servus  "campus",
   d2servus  "cantus",
   d2servus  "caseus",
   d2servus  "cervus",
   d2donum   "donum",
   feminine $ (d1puella "dea") `excepts`
     [(NounForm Plural c,"dea") | c <- [Dative, Ablative]]
  ]
```

The dictionary above consists of 11 dictionary entries, which defines a lexicon of 132 full form words. Note that when using exceptions, the use of interface functions has to be postponed. We could define exceptions on the entry level, but we would then lose the type safety.

Even more productive are the interface functions for Latin verbs. Consider the dictionary `latinVerbs` below, that uses the interface functions `v1amare` (Eng. *to love*) and `v2habere` (Eng. *to have*).

```
latinVerbs :: Dictionary
latinVerbs =
 dictionary $
  [
   v1amare  "amare",
   v1amare  "portare",
   v1amare  "demonstrare",
   v1amare  "laborare",
   v2habere "monere",
   v2habere "admonere",
   v2habere "habere"
   ]
```

The dictionary `latinVerbs` consists of 7 dictionary entries, that defines a lexicon of as many as 1029 full form words.

External dictionary

When a set of interface functions have been defined, we don't want to recompile the system every time we add a new regular word. Instead, we define an external dictionary format, with a translation function to the internal `Dictionary`. The syntax of the external dictionary format is straightforward: just a listing of the words with their paradigms. The first entries of the dictionary `latinVerbs` are written

```
v1amare  amare
v1amare  portare
v1amare  demonstrare
v1amare  laborare
```

Notice that the external dictionary format is a very simple special-purpose language implemented on top of the morphology of one language. This is the only language that a person extending a lexicon needs to learn.

Code generation

The `Dictionary` format, described in section 4.4.4, has been defined with generation in mind. It is usually easy to define a translation to another format. Let us look at an example of how the *LEXC* source code is generated. The size of the function `prLEXC`, not the details, is the interesting part. It is just 18 lines. The functions not defined in the function, is part of Haskell's standard Prelude or the standard API of Functional Morphology.

```
prLEXC :: Dictionary → String
prLEXC = unlines . (["LEXICON Root",[]] ++) . (++ ["END",[]]) .
                   map (uncurry prLEXCRules) . classifyDict

prLEXCRules :: Ident → [Entry] → String
prLEXCRules cat entries =
 unlines $ [[],"! category " ++ cat,[]] ++
             (map (prEntry . noAttr) entries)
  where
    prEntry (stem,_,inhs,tbl) =
     concat (map (prForm stem inhs) (existingForms tbl))
    prForm stem inhs (a,b) =
     unlines
      [x ++ ":" ++ stem ++ prTags (a:inhs) ++ " # ;" | x <- b]
    prTags ts =
     concat
      ["+" ++ w | t <- ts, w <- words (prFlat t)]
    altsLEXC cs =
     unwords $ intersperse " # ;" [ s  | s <- cs]
```

Currently, the following formats are supported by Functional Morphology.

Full form lexicon .A full form lexicon is a listing of all word forms with their analyses, in alphabetical order, in the lexicon.

Inflection tables. Printer-quality tables typeset in LaTeX

GF grammar source code. Translation to a Grammatical Framework grammar.

XML. An XML[27] representation of the morphological lexicon.

XFST source code. Source code for a simple, non-cyclic transducer in the Xerox notation.

LEXC source code. Source code for LEXC format, a version of XFST that is optimized for morphological descriptions.

Relational database. A database described with SQL source code.

Decorated tries. An analyzer for the morphology as a decorated trie.

CGI. A web server for querying and updating the morphological lexicon.[3]

4.4.5 Trie analyzer

The analyzer is a key component in a morphology system — to analyze a word into its lemma and its grammatical description. Synthesizers are also interesting, that is, given an analysis, produce the word form. In a trivial sense, an analyzer already exists through the XFST/LEXC formats, but Functional Morphology also provides its own analyzer.

Decorated tries is currently used instead of transducers for analysis in our implementation. Decorated tries can be considered as a specialized version of one of the languages in a transducer, that is deterministic with respect to that language, hence prefix-minimal. If we have an undecorated trie, we can also achieve total minimality by sharing, as described by Huet [15]; full-scale transducers can even achieve suffix sharing by using decorated edges. This approach has been used by Huet [13], when defining a morphology for Sanskrit. The trie is size-optimized by using a symbol table for the return value (the grammatical description).

[3] In a previous version, a CGI morphology web server was generated. Meijer's [21] CGI library was used, further modified by Panne. There exists a prototype web server [8] for Swedish. However, the CGI implementation scaled up poorly, so it is no longer generated. This is to be replaced by a SQL database and PHP.

4.4.6 Composite forms

Some natural languages have compound words — words composed from other words. A typical example is the (outdated) German word for a computer, *Datenverarbeitungsanlage*, composed from *Daten*, *Verarbeitung*, and *Anlage*. If such words are uncommon, they can be put to the lexicon, but if they are a core feature of the language (as in German), this productivity must be described in the morphology. Highly inspired by Huet's glue function [15], we have solved the problem by tagging all words with a special type `Attr` that is just a code for how a word can be combined with other words. At the analysis phase, the trie is iterated, and words are decomposed according to these parameters.

The `Attr` type is simply a integer. Together with a set of constants `atW`, `atP`, `atWP` and `atS`, we can describe how a word can be combined with another. The `atW` for stand-alone words, `atP` for words that can only be a prefix of other words, `atWP` for words that can be a stand-alone word and a prefix, and finally, `atS`, for words that can only be a suffix of other words.

```
type Attr = Int

atW, atP, atWP, atS :: Attr
(atW, atP, atWP, atS) = (0,1,2,3)
```

As an example, we will describe how to add the productive question particle *ne* in Latin, that can be added as a suffix to any word in Latin, and has the interrogative meaning of a questioning the word.

We begin by defining a type for the particle, and instantiate it in `Param`. The `Invariant` type expresses that the particle is not inflected.

```
data ParticleForm = ParticleForm Invariant
 deriving (Show,Eq)

type Particle    = ParticleForm  -> Str

instance Param ParticleForm where
    values     = [ParticleForm p | p <- values]
    prValue _  = "Invariant"
```

We continue by instantiating `ParticleForm` in `Dict`, where we also give a definition for `defaultAttr` with `atS`, that expresses that the words of this form can only appear as a suffix to another word, not as a word on its own.

```
instance Dict ParticleForm
    where category    _ = "Particle"
          defaultAttr _ = atS
```

We then define an interface function `particle` and add *ne* to our dictionary.

```
makeParticle :: String -> Particle
makeParticle s _ = mkStr s

particle :: String -> Entry
particle = entry . makeParticle

dictLat :: Dictionary
dictLat = dictionary $
          [
          ...
          particle "ne"
          ]
```

Analyzing the word *servumne*, the questioning that the object in a phrase is a slave or a servant, gives the following analysis in Functional Morphology:

```
[ <servumne>
  Composite:
  servus Noun - Singular Accusative - Masculine
  |  # ne Particle - Invariant -]
```

4.5 Results

The following morphologies have been implemented in Functional Morphology: a Swedish inflection machinery and a lexicon of 15,000 words [9]; a Spanish inflection machinery + lexicon of 10,000 words [1]; major parts of the inflection machinery + lexicon for Russian [4], Italian [24], and Latin [9]. Comprehensive inflection engines for Finnish and French have been written following the same method but using GF as source language [23]

One interesting fact is that the Master students had very limited knowledge of Haskell before they started their projects, but still managed to produce competitive morphology implementations.

An interface between morphology and syntax, through the Grammatical Framework, exists. An implemented morphology can directly be used as a resource for a grammatical description.

The analyzer tags words with a speed of 2k-50k words/second (depending on how much compound analysis is involved), a speed that compares with finite state transducers. The analyzer is often compiled faster than XFST's finite state transducers, because Kleene's star is disallowed within a word description.

4.6 Discussion

One way of viewing Functional Morphology is as a domain specific embedded language [11, 12], with the functional programming language Haskell [22] as host language.

There are a lot of features that make Haskell suitable as a host language, to mention a few: *a strong type system*, *polymorphism*, *class system*, and *higher-order functions*. Functional Morphology uses all of the mentioned features.

One could wonder if the power and freedom provided by a general-purpose programming language does not lead to problems, in terms of errors and inconsistency. Functional Morphology avoids this by requiring from the user that the definition denotes an object of a given type, i.e. the user has full freedom to use the whole power of Haskell as long as she respects the type system of Functional Morphology.

Embedding a language into another may also lead to efficiency issues — an embedded language cannot usually compete with a DSL that has been optimized for the given problem domain. This is avoided in Functional Morphology by generating other formats which provide the efficiency needed.

A simple representation of the morphology in the core system has been chosen, which enables easy generation of other formats. This approach makes the framework more easily adaptable to future applications, which may require new formats. It also enforces the *single-source* idea, i.e. a general single format is used that generates the formats of interest. A single source solves the problems of maintainability and inconsistency.

Programming constructs and features available in a functional framework make it is easier to capture generalizations that may even transcend over different languages. It is no coincidence that Spanish, French and Italian are among the languages we have implemented: the languages' morphology are relatively close, so some of the type systems and function definitions could be reused.

We believe that we provide a higher level of abstraction than the mainstream tools using regular relations, which results in a faster development

and easier adaption. Not only does the morphology implementor have a nice and flexible framework to work within, but she gets a lot for free through the translators, and will also profit from further development of the system.

Bibliography

[1] I. Andersson and T. Söderberg. Spanish Morphology – implemented in a functional programming language. Master's Thesis in Computational Linguistics, 2003. `http://www.cling.gu.se/theses/finished.html`.

[2] K. R. Beesley and L. Karttunen. *Finite State Morphology*. CSLI Publications, Stanford University, United States, 2003.

[3] C. E. Bennett. *A Latin Grammar*. Allyn and Bacon, Boston and Chicago, 1913.

[4] L. Bogavac. Functional Morphology for Russian. Master's Thesis in Computing Science, 2004.

[5] K. Claessen. *An Embedded Language Approach to Hardware Description and Verification*. PhD thesis, Chalmers University of Technology, 2000.

[6] E. Conrad. Latin grammar. `www.math.ohio-state.edu/~econrad/lang/latin.html`, 2004.

[7] M. Forsberg. Fststudio. `http://www.cs.chalmers.se/~markus/fstStudio`.

[8] M. Forsberg and A. Ranta. Svenska ord. `http://www.cs.chalmers.se/~markus/svenska`, 2002.

[9] M. Forsberg and A. Ranta. Functional morphology. `http://www.cs.chalmers.se/~markus/FM`, 2007.

[10] C. F. Hockett. Two models of grammatical description. *Word*, 10:210–234, 1954.

[11] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), 1996.

[12] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[13] G. Huet. Sanskrit site. Program and documentation, http://sanskrit.inria.fr/, 2000.

[14] G. Huet. The Zen Computational Linguistics Toolkit. http://pauillac.inria.fr/~huet/, 2002.

[15] G. Huet. Transducers as lexicon morphisms, phonemic segmentation by euphony analysis, application to a Sanskrit tagger. *Available: http://pauillac.inria.fr/~huet/FREE/*, 2003.

[16] L. Karttunen, J.-P. Chanond, G. Grefenstette, and A. Schiller. Regular expressions for language engineering. *Natural Language Engineering*, 2:305–328, 1996.

[17] G. Klyve. *Latin Grammar*. Hodder & Stoughton Ltd., London, 2002.

[18] K. Koskenniemi. *Two-level morphology: a general computational model for word-form recognition and production*. PhD thesis, University of Helsinki, 1983.

[19] A. Labs-Research. At&t fsm library. http://www.research.att.com/sw/tools/fsm/.

[20] J. Lambek. A mathematician looks at the latin conjugation. *Theoretical Linguistics*, 1977.

[21] E. Meijer and J. van Dijk. Perl for swine: Cgi programming in haskell. *Proc. First Workshop on Functional Programming*, 1996.

[22] S. Peyton Jones and J. Hughes. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available at http://www.haskell.org, February 1999.

[23] A. Ranta. Grammatical Framework Homepage, 2000–2004. www.cs.chalmers.se/~aarne/GF/.

[24] A. Ranta. 1+n representations of Italian morphology. *Essays dedicated to Jan von Plato on the occasion of his 50th birthday*, http://www.valt.helsinki.fi/kfil/jvp50.htm, 2001.

[25] A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

[26] M. K. Ronald M. Kaplan. Regular Models of Phonological Rule Systems. *Computational lingustics*, pages 331–380, 1994.

[27] The World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML/, 2000.

[28] G. van Noord. Finite state automata utilities. http://odur.let.rug.nl/~vannoord/Fsa/.

[29] Xerox. The Xerox Finite-State Compiler. http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/.

Chapter 5

The Functional Morphology Library

Author:
Markus Forsberg
Department of Computing Science
Chalmers University of Technology
and the University of Gothenburg
markus@cs.chalmers.se

Paper published:
Technical Report no. 2007-09 in Computing Science at Chalmers University of Technology and Göteborg University

5.1 Introduction

This document contains the technical report of *Functional Morphology (FM) version 2.0*. The aim of the text is to supply a detailed description on how to use FM and to provide some insights into the implementation of FM.

FM is a library for **programming** lexical resources. It is **not** a new linguistic formalism. It helps creating a lexical resource in a structured and efficient way. It is also a compiler, able to translate a lexical resource, defined in FM, into many other lexical resource formats, such as SQL or XFST source code.

Note that to be able to fully benefit from this document, a basic knowledge of the functional programming language Haskell is required.

5.2 FM Tutorial

This section presents a detailed walk-through of a fragment of a Latin morphology implemented in FM. Even though the choice of Latin is arbitrary, it works as a nice example for FM since it is a highly inflected language, which fits perfectly with the word-and-paradigm model of FM.

5.2.1 Overview

An implementation of a lexical resource in FM consists of clearly distinct components, which is naturally put into different Haskell modules. The components, listed below with short explanations, will be presented one by one in the following sections.

Type system The type system consists of inflectional, inherent and dictionary types, i.e. the parameters of the lexical resource, defined with algebraic data types.

Paradigm functions The paradigms of the lexical resource expressed as finite functions over the algebraic data types.

Interface functions The paradigm functions are translated to interface functions. An interface function connects a paradigm function to the dictionary, which includes the production of the inflection table and the addition of the inherent parameters.

Lexicon There are two kinds of lexicon for a lexicon resource, an *internal* and an *external* one. We will see that even though it may be convenient with internal lexica, there are reasons to only use external lexica.

Paradigm names The words in the external lexicon are annotated with paradigm names. The paradigm name module connects these names with interface functions.

Compound analysis (optional) If compound analysis is used, all word forms are given an attribute that defines how they can be combined with other words forms. The compound analysis function defines which of the attribute sequences correspond to possible compounds.

Main module The main module puts everything together into a runtime system.

5.2.2 Type System

The type system defines all inflectional and inherent parameters of the morphology. The parameters are defined with algebraic data types. Inflectional parameters, for example number and case, are parameters dictating how a word is inflected. Inherent parameters are attributes associated to a word, such as gender or subcategorization frame. Inherent parameters differ from inflectional parameters in that inflectional parameters are associated to a word form, but an inherent parameter is associated to the word — e.g. a feminine noun is not inflected in feminine, it *is* feminine.

The use of algebraic data types instead of ordinary strings gives many advantages. It gives a guarantee that the correct parameters are used for a paradigm as long as the correct word class is chosen. Furthermore, it is possible to define the types in such a way that only valid parameter configurations are possible to construct. For example, the cross product of the inflectional parameters of Latin verbs generates 1260 forms, but only 147 forms are existing — with algebraic types we can define a type system which disallows the 1113 non-existing forms.

Furthermore, if *incomplete pattern detection* is activated in the Haskell compiler, we can get information about missing cases. That is, if we forget to define the word forms for some parameter configurations, then the compiler will complain.

As an example, consider the Latin noun *causa* (Eng. 'cause'). It is inflected in number and case, i.e. number and case are the inflectional parameters. It also has one inherent parameter: *gender*. The inflection of *causa* in plural nominative is *causae*, but it has a feminine gender.

The parameters of Latin nouns are described with the help of Haskell's data types. To describe them, we introduce the types: `Gender`, `Case` and `Number`, given in Fig. 5.1. The `deriving` part is needed to ensure that the type is finite and enumerable — it gives us a way to enumerate all objects in a type.

The inflectional parameter types `Case` and `Number` are combined into one dictionary type `NounForm` describing all inflection forms of a Latin noun. `Gender` is not part of the dictionary type, since it is an inherent parameter.

```
data NounForm = NounForm Number Case
 deriving (Show,Eq,Ord,Bounded)
```

The `NounForm` type is missing a `deriving` for the `Enum` class. This is because the current main Haskell compiler is unable to derive the enumeration of a data type containing constructors with arguments. The reason for this

```
data Gender = Feminine  |
              Masculine |
              Neuter
 deriving (Show,Eq,Enum,Ord,Bounded)

data Case   = Nominative |
              Genitive   |
              Dative     |
              Accusative |
              Ablative   |
              Vocative
 deriving (Show,Eq,Enum,Ord,Bounded)

data Number = Singular |
              Plural
 deriving (Show,Eq,Enum,Ord,Bounded)
```

Figure 5.1: Type system

is obvious for arguments that are not `Enum` and `Bounded`, since there is no obvious enumeration strategy in that case, but there is a natural strategy if they are: simply enumerating a constructor's arguments from left to right.

An important class in FM is `Param`, defined in `General.hs`. The most important method — the only one not defined by default — is `values`. It gives the complete list of the objects in a `Param` type. An instance of `Param` is easy to define for bounded enumerated types with the function `enum`, using the member functions of `Enum` (the list generator) and `Bounded` (`minBound` and `maxBound`).

```
enum :: (Enum a, Bounded a) => [a]
enum = [minBound .. maxBound]
```

We continue by instantiating the parameters of the Latin nouns in the class `Param`. The default definition for `prValue` has been redefined for `NounForm`, to remove the `NounForm` constructor. Usually, a more sophisticated printing scheme is preferred, using a particular *tag set*, i.e. adopting to a standard for describing the types of a language.

```
instance Param Gender  where values = enum
instance Param Case    where values = enum
instance Param Number  where values = enum
instance Param NounForm where
```

```
  values =
   [NounForm n c | n <- values ,
                   c <- values]
  prValue (NounForm n c) =
    unwords $ [prValue n, prValue c]
```

A paradigm of Latin nouns is defined as a finite function `Noun`, from a `NounForm` to a `Str`; from a parameter configuration to a word form. The type for word forms is actually a list of strings, instead of a single one. The reason for this is to be able to describe missing word forms and word form variants.

```
type Noun = NounForm -> Str
```

A `Noun` is translated to an inflection table by generating all `NounForm` objects and applying them to the `Noun`. This is done by the function `table`.

```
table :: Param a => (a -> Str) -> [(a,Str)]
table f = [(a,f a) | a <- values]
```

Note that this function is polymorphic — the only restriction of `a` is that it is an instance of the `Param` class.

5.2.3 String operations

FM provides a set of string operation functions capturing common phenomena in word inflection. For a complete reference, see `General.hs` in the FM API (see Sec. 5.15.1).

The set of string operations is by no means complete. An implementer of a lexical resource typically writes new functions reflecting some specifics of the target language. For example, if it is common in a language that the second last letter is dropped while inflecting a word, it is reasonable to write a function that does exactly that. These new functions can be delivered as an extended library, which will simplify the implementation of a similar language.

For example, among the string operations are the functions `tk` and `dp`, similar to Haskell's standard functions `take` and `drop`, but they focus on suffixes instead of prefixes. `tk` takes all but the `n` last characters and `dp` drops all but the last `n` characters.

```
tk :: Int -> String -> String
tk n s = take (max 0 (length s - n)) s
```

```
dp :: Int -> String -> String
dp n s = drop (max 0 (length s - n)) s
```

Yet another example is the operator `(+?)`, which implements the common phenomenon: if the last letter of a word and the first letter of an ending coincide, then one of them is dropped. An example of the usage is given in the function `mkCase`, where the genitive case of Swedish nouns is formed by adding an 's' to word forms, unless it does not already end in 's'. In that case, nothing is added.

```
(+?) :: String -> String -> String
s +? e = case (s,e) of
            (_:_,c:cs) | last s == c -> s ++ cs
            _ -> s ++ e

mkCase :: Case -> String -> String
mkCase c w = case c of
  Nom -> w
  Gen -> w +? "s"
```

The strings operations all share the property that they perform a small, specific task. And more, that their definitions are compact and easily understood.

5.3 Paradigms as functions

Let us start by considering the first declension noun paradigm, illustrated with the inflection table of the word `rosa` (Eng. 'rose'), in Fig. 5.2. The concept of inflection tables corresponds intuitively to a list of pairs in a programming language, but FM takes an indirect approach and uses finite functions, which is later translated to a list of pairs. The use of finite functions has many advantages: it allows the use of higher-order functions, e.g. see *exceptions* in Sec. 5.3.1; it allows us to divide the paradigm definitions into sets of finite functions, each solving a specific task, which in the end are combined into one function with function compositions; and it allows the use of pattern matching, which permits common cases to be defined simultaneously.

The paradigm function for the first declension paradigm, `decl1`, is directly defined based on the inflection table. It is defined as a single function.

	Singular	Plural
Nominative	*rosa*	*rosae*
Vocative	*rosa*	*rosae*
Accusative	*rosam*	*rosas*
Genitive	*rosae*	*rosarum*
Dative	*rosae*	*rosis*
Ablative	*rosa*	*rosis*

Figure 5.2: The inflection table of *rosa*

```
decl1rosa :: String -> Noun
decl1rosa rosa (NounForm n c) =
 mkStr $
  case n of
   Singular ->
    case c of
     Accusative -> rosa ++ "m"
     Genitive   -> rosa ++ "e"
     Dative     -> rosa ++ "e"
     _          -> rosa
   Plural   ->
    case c of
     Nominative -> rosa ++ "e"
     Vocative   -> rosa ++ "e"
     Accusative -> rosa ++ "s"
     Genitive   -> rosa ++ "rum"
     _          -> rosa ++ "is"
```

Note that the paradigm function requires one argument, a citation word form. The functions `rosa` and `puella` are two nouns created by the application of the citation forms `"rosa"` and `"puella"` (Eng. 'girl').

```
rosa :: Noun
rosa   = decl1 "rosa"

puella :: Noun
puella = decl1 "puella"
```

5.3.1 Exceptions

Many paradigms of the same type are similar, just differing in one or two word forms. When defining a class of similar paradigms, its is convenient to use FM:s *exceptions*. Exceptions are used to describe inflection functions

in terms of other inflection functions. Instead of defining a completely new paradigm, we use the old definition and only mark what is different. This is not only linguistically more satisfying, it saves a lot of work.

There are four different kinds of exception: `excepts`, `missing`, `only` and `variants`. All exceptions are higher-order functions that take a finite inflection function as an argument.

The exceptions `except` and `excepts`, take a finite inflection function and list of exceptions, and constructs a new finite function with the exceptions included. An example of its usage is given in the definition of `decl2gladius`.

```
except  :: Param a => (a -> Str) -> [(a,String)] -> (a -> Str)
excepts :: Param a => (a -> Str) -> [(a,Str)]    -> (a -> Str)

decl2gladius :: String -> Noun
decl2gladius gladius =
  except (decl2servus gladius)
     [(NounForm Singular Genitive, gladi),
      (NounForm Singular Vocative, gladi)]
 where gladi =  tk 2 gladius
```

The exception functions `missing` and `only` are used to express missing cases in a table. `missing` enumerates the cases with missing forms, and `only`, used for highly defective words, enumerates the cases that exists. An example is the paradigm of *vis* (Eng. 'force'), which inflects in the same manner as *hostis* (Eng. 'enemy'), with the exception that it is missing the singular vocative, genitive and dative case.

```
missing :: Param a => (a -> Str) -> [a] -> (a -> Str)
only    :: Param a => (a -> Str) -> [a] -> (a -> Str)

vis_paradigm :: String -> Noun
vis_paradigm s = (hostisParadigm s) 'missing'
   [NounForm Singular c | c <- [Vocative, Genitive, Dative]]
```

A very common exception is additional variants, i.e. that two paradigms differ only in the number of word forms for one or more parameter configuration. This type of exception is expressed with the functions `variant` and `variants`.

An example is given with the function `decl3parti`, a Swedish paradigm function. The function is defined in terms of a worst-case function `mkNoun`, which takes `String`s as arguments. This function is then augmented with two variant word forms through the use of `variant`.

```
variant  :: Param a => (a -> Str) -> [(a,String)] -> (a -> Str)
variants :: Param a => (a -> Str) -> [(a,Str)]    -> (a -> Str)

decl3parti :: String -> Substantive
decl3parti parti =
 mkNoun parti (parti ++ "et") (parti ++ "er") (parti ++ "erna")
  `variant`
  [(SF Sg Def c, mkCase c (parti++"t") | c <- values]
```

Note that we use `values` to generate the values of `c`. The type system is able to infer that `c` is of type `Case`, and since `Case` is an instance of the class `Param`, we can use the function `values` to generate the constructors `Nom` and `Gen`.

5.4 Interface Functions

A lexical resource has its own type system, so to be able to use generic translations, we need to translate it into an intermediary format, a `Dictionary`. A `Dictionary` is an untyped ADT consisting of a list of `Entry`:s. An `Entry` corresponds to dictionary entry, specifying information about a word, e.g. the inflection table and the inherent parameters.

The translation is done by first instantiating the dictionary types in the `Dict` class, defined in `Dictionary.hs` (see Sec. 5.15.2). Typically, the only information we need to supply is the name of the word class that the dictionary type represents. Note that since we have no access to the names of the types within Haskell, we must require that this information is supplied by the user.

Let us return to the Latin noun example with the dictionary type `NounForm`. When the `NounForm` type is made an instance of `Dict`, we also give the name of the word class that `NounForm` represents, i.e. `Noun`.

```
instance Dict NounForm where
 category _ = "Noun"
```

The next step is to define interface functions, i.e. functions that create `Entry`:s. We start by a general interface function, `noun`, which transforms a `Noun` together with its inherent parameter, `Gender`, and its paradigm identifier, to an `Entry`. The identifier is not the same as the paradigm names in a command map (see Sec. 5.6), it is used in the *word identifier*. A word identifier is built up from the citation form, the word class, the inherent

parameters and the identifier. For example, the word *rosa* has the identifier `rosa_Noun_Feminine__n1`. The main difference between the command map identifiers and the ones in the word identifiers is that command map identifiers must be unique.

We also define a function for every gender: `masculine`, `feminine` and `neuter`.

```
noun :: Noun -> Gender -> Paradigm -> Entry
noun n g p = entryIP n [prValue g] p

feminine :: Noun -> Paradigm -> Entry
feminine n = noun n Feminine

masculine :: Noun -> Paradigm -> Entry
masculine n = noun n Masculine

neuter :: Noun -> Paradigm -> Entry
neuter n = noun n Neuter
```

We can now define interface functions for all our paradigm functions. Let us have a look at three interface functions, one for each gender: `d1rosa` (Eng. 'rose'), `d2servus` (Eng. 'servant'), and `d2bellum` (Eng. 'war').

```
d1rosa :: DictForm -> Entry
d1rosa w = feminine (decl1rosa w) "n1"

d2servus :: DictForm -> Entry
d2servus w = masculine (decl2servus w) "n2"

d2bellum :: DictForm -> Entry
d2bellum w = neuter (decl2bellum w) "n2"
```

We can now create a small lexicon with the interface functions we have defined. The function `dictionary` is an abstraction function that creates a `Dictionary` ADT from a list of `Entry`:s.

```
latinDict :: Dictionary
latinDict =
 dictionary $
  [
   d1rosa    "puella",
   d2servus  "somnus",
   d2servus  "amicus",
   d2bellum  "donum"
  ]
```

The lexicon we just defined is referred to as an *internal lexicon*, since it is defined within Haskell. If we add a new word, we need to recompile our FM implementation, but on the other hand, we have the full power of Haskell at our disposal. This is contrasted with an *external lexicon*, which is simply a text file (discussed in Sec. 5.6). If we add a new word to the *external lexicon* there is no need to recompile. We are, however, more restricted in what we can express, since we no longer have access to Haskell.

In previous documentations of FM, we recommended that the irregularly inflected words should be defined in the internal lexicon, and the regularly ones in the external lexicon. We have reconsidered this somewhat, since it is very convenient to have all words listed in the external lexicon — all in the same place, in the same format. A definite preference, however, depends on the intended usage of the lexical resource in question.

5.5 Compound Analysis

FM offers the possibility to perform compound analysis. By default, all words are assumed to appear outside compounds, so the compound analysis is invisible to someone who does not use it.

We will use the particle *ne* (Eng. approximately '?') as our example, a clitic element that can be placed on any word, and by that, it expresses that the word it attaches to is in some way questioned.

The first thing we need to do is to define *attributes*, which will be used to describe the compound behaviour of *ne*. Attributes are integers greater than one and we use the type `Attr` to refer to them.

For the purpose of our example, we will only define one attribute, `atS`, which will be used for words that may only appear as a suffix on another word form.

```
atS :: Attr
atS = 1
```

The next step is to associate the attribute to the dictionary type. This is done in the instantiation of the class `Dict`, or more precisely, in the class function `defaultAttr`.

```
instance Dict ParticleForm where
 category    _ = "Particle"
 defaultAttr _ = atS
```

All words with the dictionary type `ParticleForm` now have the default attribute `atS`. As the name `defaultAttr` implies, it is also possible to associate an attribute value to any parameter configuration.

All word forms have an attribute associated to them. If no attribute association has been defined for a word form, it receives the default attribute value. The default attribute value is `0`, which explains why a user-defined attribute value must be larger than `0`.

The compound analysis try to divide an input word form into all possible sequences of word forms with their associated attributes. Some of these sequences will, of course, not be valid. It is the *compound function*, defined by the implementer of the lexical resource, which decides what attributes are valid.

The compound function `latin_compound` defines the valid attribute sequences of our Latin lexical resource.

```
latin_compound :: [Attr] -> Bool
latin_compound [x,y] = (x /= y) && atS == y
latin_compound [x]   = x /= atS
latin_compound _     = False
```

Word forms with the attribute `atS` may only occur as suffixes. All other word forms may only occur as single words.

5.6 Paradigm Identifiers and External Lexicon

We do not want to recompile the whole system every time we add a new word, in fact, we may not want to recompile at all. This is where the external lexicon comes into picture. An external lexicon is a text file containing a list of citation forms marked with *paradigm names*. A paradigm name refers to an interface function.

The first thing we need to do to get things working is to define a *command map*. The command map defines the mapping between paradigm name and interface functions. It consists of a list of triplets, where the first element is the paradigm name, the second element is example citation forms for the paradigm, and the third element is the interface function.

The interface functions are applied to a function `app1` that requires a special explanation. First of all, we want to be able to have interface functions that have more than one argument. But then we have a problem, since the type system of Haskell does not allow functions of different types to appear at the same position in a list. The solution provided by FM is to

use one of a set of wrapper functions, named `app1`, `app2`, `app3`, et cetera, where the number corresponds to the argument count. These wrapper functions encapsulate the interface functions, creating new functions of the type `[String] -> Entry`. Since all wrapper functions create a function of the same type, we can have interface functions of different argument count appearing in the command list.

```
commands =
 [
  ("d1puella",     ["rosa"],   app1 d1puella),
  ("d1puellaMasc", ["poeta"],  app1 d1puellaMasc),
  ("d2servus",     ["servus"], app1 d2servus),
  ("d2servusFem",  ["pinus"],  app1 d2servusFem),
  ("d2servusNeu",  ["virus"],  app1 d2servusNeu),
  ("d2bellum",     ["bellum"], app1 d2bellum),
  ("d2puer",       ["puer"],   app1 d2puer),
  ("d2liber",      ["liber"],  app1 d2liber),
  ("prep",         ["ad"],     app1 prep),
  ("v1amare",      ["amare"],  app1 v1amare),
  ("v2habere",     ["habere"], app1 v2habere)
  ]
```

Given that we have defined our command map (and our runtime system, see Sec. 5.7), then we can start developing our external lexicon.

```
latin.lexicon:
 v1amare  amare
 v1amare  portare
 v1amare  demonstrare
 v1amare  laborare
```

The external lexicon is in a file `latin.lexicon`, where we have defined four words in the first conjugation: *amare* (Eng. 'to love'), *portare* (Eng. 'to carry'), *demonstrare* (Eng. 'to point out'), and *laborare* (Eng. 'to work'). Note that the format of an external lexicon is simple — it consists of paradigm names and citation forms. Single line comments are allowed, triggered by `--`, but besides that, there is nothing more.

5.7 Runtime System

The last thing we need to do is to connect our lexical resource with the runtime system of FM. For this, FM uses a class `Language`, defined in

`Frontend.hs` (see Sec. 5.15.4), which gives the language-specific parts of the runtime system. We start by defining a type consisting of a single constructor, which is the default name of our lexical resource.

```
data Latin = Latin
 deriving Show
```

Next, we make our data type an instance of of the `Language` class, where we define functions needed for the runtime system. All class functions have a default definition, e.g. the internal dictionary may be empty; the list of commands may be empty; or there may be no compound analysis. In this instance, we define our internal dictionary, our compound function, and our command map, which is folded into a more efficient lookup table.

```
instance Language Latin where
 internDict   _ = latinDict
 composition  _ = latin_compound
 paradigms    _ = foldr insertCommand emptyC commands
```

We can now define our `main` function with the help of the FM library `commonMain` applied to our constructor `Latin`.

```
main :: IO ()
main = commonMain Latin
```

The constructor `Latin` is used to retrieve the information provided in the instance of the `Language` class. It is a convenient way to avoid having many optional arguments in the `commonMain` function.

This concludes the FM tutorial — we have now defined a complete fragment of a lexical resource for Latin. For information on how to compile FM, see Sec. 5.10, and on how to run FM, see Sec. 5.11.

5.8 Extending the Translator

This section lists how to add a new output format called `FORMAT`. For additional help, have a look at how another format is defined.

1. Define a function in `Print.hs`:
   `prFORMAT :: Dictionary -> String`

2. Define two functions in `GeneralIO.hs`:
   `writeFORMAT :: FilePath -> Dictionary -> IO()`
   `outputFORMAT :: Dictionary -> IO()`
   These functions, responsible of writing the output of `prFORMAT` to the file `Filepath` and standard output respectively, typically add an header to the output.

3. In `CommonMain.hs`: add a new command-line flag (e.g. `-format`) and document it in the help message `help_text`.

5.9 Compound Analysis in FM

A compound in FM is a word $w = w_1 w_2 ... w_n$ where $dictionary(w_i)$ and $valid(attr(w_1)...attr(w_n))$. *dictionary* is a boolean function defining the word forms of a language. *attr* is function that for every word form in the dictionary assigns a set of parameter values. The parameter values defines how the word forms can be composed with other words. *valid* is a boolean function that accepts as input a list of sets of attribute values and gives as result a boolean value that states if the sequence of attribute values is valid or not.

The compound analysis of FM consists of two functions, `unglue` and `valid`. The `unglue` function is a rewritten version of Huet's unglueing function [3], which splits an input word, based on a dictionary, into all possible compounds. Note that it is essential to have the dictionary check in the generator, since the generator would otherwise be subject to a combinatorial explosion.

```
unglue [] dictionary = [[]]
unglue  w dictionary = [map (pre:) (unglue suf) |
                          (pre,suf) <- zip (prefixes w)
                                           (suffixes w)
                          dictionary pre]
```

The `valid` function uses the compound function to filter out only the valid compound.

```
valid c_fun cs = filter_valid attr_values
  where
   attr_values = flatten $ map lookup_attr cs
   filter_valid [] = []
   filter_valid (as:ass)
    | c_fun (extract_attr as) = as : filter_valid ass
    | otherwise               = filter_valid ass
```

These two functions could be combined to avoid duplicate work, but are held separate for the presentation. It may seem inefficient to separate the validity test with the actual unglueing, but since Haskell is a lazy programming language, the situation is better. The laziness ensures that the unglueing process only continues on the suffix if the prefix is in the lexicon.

Since a word form may be a homograph, it can be associated with a set of attributes. Because of this, we need to use the function `flatten` to flatten the sequences of sets of attributes into a set of attribute sequences.

The function `compound_analysis` puts everything together.

```
compound_analysis c_fun w =
  concat $ map (valid c_fun) (unglue w)
```

Huet [3] uses a different approach to compounds: he uses rewrite rules to describe internal and external sandhi of Sanskrit. The rules are compiled into a dictionary trie with the addition of choice points, which encodes the rules. The sandhi of Sanskrit is complicated since the spelling exactly reflects the pronunciation of the sentences.

It is not clear that it would be possible to handle Sanskrit's sandhi in FM. It may be the case that the number of word forms would be to great to be feasible to define in FM style.

The translation in FM of a lexical resource with compounding to other systems is not complete — even though the compound information is exported, the compound function is missing. The compound function is currently a function in Haskell that is not readily translatable. The situation could be improved by an algebraic representation of the compound function, which in turn would be translatable. How such an algebraic representation is best implemented requires some future work.

5.10 Compiling FM

The source code is downloadable at FM's homepage[1].

FM requires the GHC compiler[2] to be built. Since FM is a command-line program, it should work on all platforms supported by the GHC compiler.

1. Unpack the source code: `tar -xvfz FM_LAT_v2.0.tgz`
2. Change directory: `cd ./FM_LAT_v2.0/`

[1] `http://www.cs.chalmers.se/~markus/FM`
[2] `http://www.haskell.org/ghc`

3. Compile FM: `make`

4. This produces a binary `morpho_lat`

5.11 Running FM

We assume that the tutorial language is downloaded — the lexical resource of Latin, and compiled in the manner described in Sec. 5.10. Other FM implementations are handled in an analogous way.

Before FM implementation can be run, one needs to refer to the external lexicon, `latin.lexicon`. This is done by either running the program in the same directory as the external lexicon, or by pointing the environment variable `FM_LAT` to it. Environment variables are set differently depending on which shell are in use, but in a Bash shell, and given that the lexicon file is placed in the directory `/home/dictionary`, we would write the command below. Or better, put the command in one of the system files declaring the environment variables.

```
$ export FM_LAT="/home/dictionary/latin.lexicon"
```

The runtime system of an FM implementation consists of four parts: the analyzer, the synthesizer, the inflection engine, and the translator. We will describe each of these parts in the rest of this section. An overview of the command-line flag of FM is printed with the help command `morpho_lat -h`. The output is given in Fig. 5.3.

5.11.1 The Analyzer

The *analyzer*, also referred to as the *tagger*, annotates the word forms of an input text with information collected from the current lexical resource. The analyzer is divided into two phases: *word segmentation*, or *tokenization*, and *word analysis*. The word segmentation splits the string of the input text into tokens, and the word analysis, which may be compound analysis, does the actual annotation.

An example of the analyzer in action is given in Fig. 5.4, where two word forms is being tagged: *servi* and *servusne*. The first word, *servi*, is the inflection form of *servus* (Eng. 'servant'). It is ambiguous: it may be singular genitive, plural nominative or plural vocative. The second word, *servusne*, is a compound word consisting of *servus* and the question particle *ne*.

```
|--------------------------------------|
|        Program parameters            |
|--------------------------------------|
| -h             | Display this message |
|--------------------------------------|
| <None>         | Enter tagger mode    |
|--------------------------------------|
| -s             | Enter interactive    |
|                | synthesiser mode     |
|--------------------------------------|
| -i             | Enter inflection     |
|                | mode                 |
|--------------------------------------|
| -lex    [file] | Full form lexicon    |
| -tables [file] | Tables               |
| -gf     [file] | GF source code       |
| -latex  [file] | LaTeX source code    |
| -xml    [file] | XML source code      |
| -lexc   [file] | LexC source code     |
| -xfst   [file] | XFST source code     |
| -sql    [file] | SQL source code      |
|--------------------------------------|
```

Figure 5.3: FM help

The part about *Morphology Statistics* contains information about the lexical resource — here we see that 11 paradigms have been implemented; the lexicon consists of 196 entries, 173 in the external lexicon and 23 in the internal; these entries are expanded into 8131 word forms (yes, Latin is a highly inflected language!) of which 5417 are unique. Finally, the compile time for the dictionary and the building time of the analysis data structure sums to 1.00 seconds.

FM is, of course, capable of analyzing a complete text. Given a Latin text `Latina_Vulgate.txt`, we analyze the whole text by simply piping the text to the FM program, as illustrated below.

```
$ cat Latina_Vulgate.txt | morpho_lat
```

The analysis is reasonably fast. The analysis of 'Latina Vulgate', consisting of approximately 1.4 million word forms, took around 25 seconds on a Macbook (including the compile time of the lexical resource), which gives us an analysis speed in the ballpark of 56k word forms per second.

5.11.2 The Synthesizer

The synthesizer is used to retrieve all dictionary entries that include the input word form in their inflection table. An example is provided in Fig. 5.5

```
$ morpho_lat

************************************
*   Functional Morphology v2.0     *
* (c) M. Forsberg & A. Ranta 2007  *
* under GNU General Public License. *
************************************

Morphology Statistics:
 # language id: latin
 # 11 paradigms
 # 0k entries (e: 173, i: 23)
 # 8k word forms (c: 8131, u: 5417)
 # compile time: 1.00 seconds

servi
[ <servi>
1. servus (servus_Noun_Masculine__n2:1)
          Noun - Plural Vocative - Masculine [0]
2. servus (servus_Noun_Masculine__n2:1)
          Noun - Plural Nominative - Masculine [0]
3. servus (servus_Noun_Masculine__n2:1)
          Noun - Singular Genitive - Masculine [0]
]
servusne
[ <servusne>
1. Composite: servus (servus_Noun_Masculine__n2:1) Noun -
              Singular Nominative - Masculine [0]  |
              ne (ne_Particle__inv:1) Particle - Invariant [1]
]
```

Figure 5.4: FM analysis example

with the word form *puellae*. In this example, we only got one entry, since the word form *puellae* only appears in one entry. If it were an homograph, on the other hand, appearing in more than one entry, then those entries would be listed also.

The current version of FM does not include compound words in the synthesizer: it only retrieves word forms that exists in the lexicon. How to include compound words in the synthesis is by no means obvious — we need a method to decide which word form in the compound that is the main word form, to be able to select the correct word class and the correct inherent parameters. And more, this method must be valid for any language, and preferably invisible to an FM implementer that do not use compound analysis.

A possible solution would be to strengthen the compound function so that it would, as a result, not only give true or false, but also which attribute corresponds to the main word form. Such an addition, however, would break the backward compatibility of FM.

5.12 The Inflection Engine

The inflection engine of FM translates paradigm names applied to citation word forms to dictionary entries. The inflection engine is also runnable in batch mode, i.e. the input can be piped to the program. An example of the inflection engine in interactive mode is given in Fig. 5.6, where the word *porta* (Eng. 'door') is marked as being in the first declension. The same result is achieved in batch mode with the following command.

```
$ echo "d1rosa porta" | ./morpho_lat -ib
```

Typing `'c'` in interactive mode gives the list of all paradigm names together with their example word forms.

5.13 The Translator

An important aspect of FM is its use as a compiler. The idea is that the user of FM should never get "stuck" in FM, but instead have the ability to translate the lexical resource to many other lexicon formats, and by doing that, maximize the usefulness of the resource. In fact, FM has been designed so that adding a new format is a relatively small task (see Sec. 5.8 for details).

The formats currently supported by FM will now be exemplified one by one. We will use the same example word, and only the part of the

```
$ morpho_lat -s

************************************
*   Functional Morphology v2.0     *
* (c) M. Forsberg & A. Ranta 2007  *
* under GNU General Public License. *
************************************


Synthesiser mode

Enter a Latin word in any form

or a [paradigm name] with [word forms].

Type 'c' to list paradigms.

Type 'q' to quit.

Morphology Statistics:
 # language id: latin
 # 11 paradigms
 # 0k entries (e: 173, i: 23)
 # 8k word forms (c: 8131, u: 5417)
 # compile time: 0.00 seconds

> puellae
[ <puellae>
{
lemma: puella
pos: Noun
inherent(s):  Feminine
paradigm id: n1

Singular Nominative : puella
Singular Vocative : puella
Singular Accusative : puellam
Singular Genitive : puellae
Singular Dative : puellae
Singular Ablative : puella
Plural Nominative : puellae
Plural Vocative : puellae
Plural Accusative : puellas
Plural Genitive : puellarum
Plural Dative : puellis
Plural Ablative : puellis
}
]
```


Figure 5.5: FM synthesis example

```
$ morpho_lat -i

************************************
*   Functional Morphology v2.0     *
* (c) M. Forsberg & A. Ranta 2007  *
* under GNU General Public License. *
************************************


[Inflection mode]

Enter [paradigm name] with [word forms].

Type 'c' to list paradigms.

Type 'q' to quit.

> d1rosa porta
porta
Noun
Feminine
Singular Nominative: porta
Singular Vocative: porta
Singular Accusative: portam
Singular Genitive: portae
Singular Dative: portae
Singular Ablative: porta
Plural Nominative: portae
Plural Vocative: portae
Plural Accusative: portas
Plural Genitive: portarum
Plural Dative: portis
Plural Ablative: portis

>
```

Figure 5.6: FM inflection mode example

generation that refers to that word, to enable comparisons between the different formats. The word used is *filius* (Eng. 'son').

5.13.1 Full Form Lexicon

A fundamental format in FM is the *full form lexicon*, which is the format that the analyzer builds on. The format consists of all word forms annotated with their analyses (separated by ':').

```
filius:filius (filius_Noun_Masculine__n2:1) Noun -
              Singular Nominative - Masculine [0]
fili:filius (filius_Noun_Masculine__n2:1) Noun -
            Singular Vocative - Masculine [0]
filium:filius (filius_Noun_Masculine__n2:1) Noun -
              Singular Accusative - Masculine [0]
fili:filius (filius_Noun_Masculine__n2:1) Noun -
            Singular Genitive - Masculine [0]
filio:filius (filius_Noun_Masculine__n2:1) Noun -
             Singular Dative - Masculine [0]
filio:filius (filius_Noun_Masculine__n2:1) Noun -
             Singular Ablative - Masculine [0]
filii:filius (filius_Noun_Masculine__n2:1) Noun -
             Plural Nominative - Masculine [0]
filii:filius (filius_Noun_Masculine__n2:1) Noun -
             Plural Vocative - Masculine [0]
filios:filius (filius_Noun_Masculine__n2:1) Noun -
              Plural Accusative - Masculine [0]
filiorum:filius (filius_Noun_Masculine__n2:1) Noun -
                Plural Genitive - Masculine [0]
filiis:filius (filius_Noun_Masculine__n2:1) Noun -
              Plural Dative - Masculine [0]
filiis:filius (filius_Noun_Masculine__n2:1) Noun -
              Plural Ablative - Masculine [0]
```

5.13.2 Inflection Tables

Inflection tables can be generated in two formats, either as text or Latex source code. The text version is given below.

```
$ morpho_lat -tables

filius
Noun
Masculine
```

```
Singular Nominative: filius
Singular Vocative: fili
Singular Accusative: filium
Singular Genitive: fili
Singular Dative: filio
Singular Ablative: filio
Plural Nominative: filii
Plural Vocative: filii
Plural Accusative: filios
Plural Genitive: filiorum
Plural Dative: filiis
Plural Ablative: filiis
```

The generation of tables in Latex source code enables us to create nicer, formatted tables with the program `latex`.

```
$ morpho_lat -latex

filius, Noun Masculine
\begin{center}
\begin{tabular}{|l|l|}\hline
Singular Nominative & {\em filius} \\
Singular Vocative & {\em fili} \\
Singular Accusative & {\em filium} \\
Singular Genitive & {\em fili} \\
Singular Dative & {\em filio} \\
Singular Ablative & {\em filio} \\
Plural Nominative & {\em filii} \\
Plural Vocative & {\em filii} \\
Plural Accusative & {\em filios} \\
Plural Genitive & {\em filiorum} \\
Plural Dative & {\em filiis} \\
Plural Ablative & {\em filiis} \\
\hline
\end{tabular}
\end{center}
% \newpage
```

Grammatical framework (GF)

Grammatical Framework [4] is a multilingual grammar formalism, and because of the translation, we have a direct connection between a lexical resource and syntax, i.e. a GF grammar. GF also requires a type system, which is not exported from FM. The type system, which should corresponds to the type system of the FM implementation, must be in the file `types.latin.gf`.

We have actually cheated a bit with the generation — in our FM implementation we used the possibility to have pretty-printed versions of our types, through the function `prValue` in the class `Param`, where we have removed the constructor `NounForm`. For the generation to be correct, we needed to remove our `prValue` declarations, and recompile FM. It may be reasonable to extend FM to support two kinds of `Dictionary`:s, one with the pretty-printed types and one without, to avoid having to remove `prValue` declarations prior to GF generation.

```
$ morpho_lat -gf

include types.latin.gf ;

cat Noun;

fun filius_1 : Noun ;

lin filius_1 = {s = table {
  NounForm Singular Nominative => "filius" ;
  NounForm Singular Vocative   => "fili" ;
  NounForm Singular Accusative => "filium" ;
  NounForm Singular Genitive   => "fili" ;
  NounForm Singular Dative     => "filio" ;
  NounForm Singular Ablative   => "filio" ;
  NounForm Plural Nominative   => "filii" ;
  NounForm Plural Vocative     => "filii" ;
  NounForm Plural Accusative   => "filios" ;
  NounForm Plural Genitive     => "filiorum" ;
  NounForm Plural Dative       => "filiis" ;
  NounForm Plural Ablative     => "filiis" };
  h1 = Masculine
} ;
```

5.13.3 XML

The XML [5] format is a way of representing structured information in ASCII. It is the most verbose format of FM, which is typical to an XML representation. However, it is not as bad as it seems, since an XML file may be heavily compressed.

```
$ morpho_lat -xml

<lexicon_entry>
 <dictionary_form value="filius" />
```

```
 <inherent value="Masculine" />
 <inflection_table>
  <inflection_form pos="Singular Nominative">
   <variant word="filius" />
  </inflection_form>
  <inflection_form pos="Singular Vocative">
   <variant word="fili" />
  </inflection_form>
  <inflection_form pos="Singular Accusative">
   <variant word="filium" />
  </inflection_form>
  <inflection_form pos="Singular Genitive">
   <variant word="fili" />
  </inflection_form>
  <inflection_form pos="Singular Dative">
   <variant word="filio" />
  </inflection_form>
  <inflection_form pos="Singular Ablative">
   <variant word="filio" />
  </inflection_form>
  <inflection_form pos="Plural Nominative">
   <variant word="filii" />
  </inflection_form>
  <inflection_form pos="Plural Vocative">
   <variant word="filii" />
  </inflection_form>
  <inflection_form pos="Plural Accusative">
   <variant word="filios" />
  </inflection_form>
  <inflection_form pos="Plural Genitive">
   <variant word="filiorum" />
  </inflection_form>
  <inflection_form pos="Plural Dative">
   <variant word="filiis" />
  </inflection_form>
  <inflection_form pos="Plural Ablative">
   <variant word="filiis" />
  </inflection_form>
 </inflection_table>
</lexicon_entry>
```

5.13.4 XFST

XFST [2] (Xerox Finite State Transducer) source code defines a regular relation, i.e. a relation between two regular languages. A regular relation can be compiled into a finite state transducer, which is an automaton providing a compact and efficient structure for lexical resources. XFST source code is compiled by the XFST tool.

```
$ morpho_lat -xfst

[ {filius} %+Singular  %+Nominative  %+Masculine .x. {filius}]  |
[ {filius} %+Singular  %+Vocative  %+Masculine .x. {fili}]  |
[ {filius} %+Singular  %+Accusative  %+Masculine .x. {filium}]  |
[ {filius} %+Singular  %+Genitive  %+Masculine .x. {fili}]  |
[ {filius} %+Singular  %+Dative  %+Masculine .x. {filio}]  |
[ {filius} %+Singular  %+Ablative  %+Masculine .x. {filio}]  |
[ {filius} %+Plural  %+Nominative  %+Masculine .x. {filii}]  |
[ {filius} %+Plural  %+Vocative  %+Masculine .x. {filii}]  |
[ {filius} %+Plural  %+Accusative  %+Masculine .x. {filios}]  |
[ {filius} %+Plural  %+Genitive  %+Masculine .x. {filiorum}]  |
[ {filius} %+Plural  %+Dative  %+Masculine .x. {filiis}]  |
[ {filius} %+Plural  %+Ablative  %+Masculine .x. {filiis}]
```

5.13.5 LexC

LexC [2] source code is another, but more restricted, regular relation format designed by Xerox. The restrictions of the format enable the XFST tool to compile the regular relation to a finite state transducer faster and allow better optimizations to be done on the resulting finite state transducer.

```
$ morpho_lat -lexc

filius:filius+Singular+Nominative+Masculine # ;
fili:filius+Singular+Vocative+Masculine # ;
filium:filius+Singular+Accusative+Masculine # ;
fili:filius+Singular+Genitive+Masculine # ;
filio:filius+Singular+Dative+Masculine # ;
filio:filius+Singular+Ablative+Masculine # ;
filii:filius+Plural+Nominative+Masculine # ;
filii:filius+Plural+Vocative+Masculine # ;
filios:filius+Plural+Accusative+Masculine # ;
filiorum:filius+Plural+Genitive+Masculine # ;
filiis:filius+Plural+Dative+Masculine # ;
filiis:filius+Plural+Ablative+Masculine # ;
```

5.13.6 SQL

SQL, Structured Query Language [1], is a popular source format for defining databases. The first part of the generation creates a table `LEXICON` and defines the types of the elements in the table. We use integers here instead of word identifiers to identify words. The second part simply consists of insertions of data into the table.

```
$ morpho_lat -sql

CREATE TABLE LEXICON
(
ID INTEGER NOT NULL,
DICTIONARY VARCHAR(50) NOT NULL,
CLASS VARCHAR(50) NOT NULL,
WORD VARCHAR(50) NOT NULL,
POS VARCHAR(50) NOT NULL);

INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filius','Singular Nominative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','fili','Singular Vocative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filium','Singular Accusative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','fili','Singular Genitive - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filio','Singular Dative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filio','Singular Ablative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filii','Plural Nominative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filii','Plural Vocative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filios','Plural Accusative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filiorum','Plural Genitive - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filiis','Plural Dative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filiis','Plural Ablative -Masculine');
```

5.14 Other Commands

5.14.1 Precompiled Dictionary

When we run FM we always rebuild our dictionary, and if it is large it may take some time. However, if we are only going to use the analyzer, there is a shortcut. A full form lexicon, i.e. a precompiled list of word forms, can be read with the following command.

```
./morpho_lat -f latin.fullform
```

The file `latin.fullform` was generated with the full form generation of FM, i.e. with the command `morpho_lat -lex`.

5.14.2 Print Paradigms

The paradigms of FM are printed with the `-p` flag. The result is similar to generating inflection tables, but with the crucial difference that every paradigm is only printed once and only for those paradigms that have been defined in the command map.

5.15 The Functional Morphology API

5.15.1 General.hs

The type for word forms. The type is a list to allow variants and missing word forms.

```
newtype Str = Str [String]
```

The type for polymorphic inflection tables.

```
type Table a = [(a, Str)]
```

The type for finite inflection functions.

```
type Finite a = a -> Str
```

The class for finite parameters.

```
class (Eq a, Show a) => Param a where
 values :: [a]
 value :: Int -> a
 value0 :: a
 prValue :: a -> String
```

The type for token: `W` for normal tokens, `P` for symbols and `D` for digits.

```
data Tok = W String
         | P String
         | D String
```

The attribute type for compounds.

```
type Attr = Int
```

The default attribute value (0).

```
noComp :: Attr
```

Promotes `String` to `Str`.

```
mkStr :: String -> Str
```

Sharing of `Str`:s, achieved by the use of a global hash table.

```
shareStr :: Str -> Str
```

Translate a `Str` to `[String]`.

```
unStr :: Str -> [String]
```

Promotes `[String]` to `Str`.

```
strings :: [String] -> Str
```

Apply function to `a` and promote the resulting `String` to `Str`.

```
mkStr1 :: (a -> String) -> a -> Str
```

Apply function to all variants in `Str`.

```
mapStr :: (String -> String) -> Str -> Str
```

The union of two `Str`.

```
unionStr :: Str -> Str -> Str
```

Prepend a string to all variants in `Str`.

```
(+*) :: String -> Str -> Str
```

Concatenation that marks the morpheme boundaries.

```
(+/) :: String -> String -> String
```

Variants listed in a string. Translated into a list of word by the function `words`.

```
mkStrWords :: String -> Str
```

Takes all but `Int` characters in the end of the string.

```
tk :: Int -> String -> String
```

Drops all but `Int` character in the end of the string.

```
dp :: Int -> String -> String
```

Gets the `Int`:th character from the end of String.

```
ch :: Int -> String -> String
```

Prevents duplication, e.g. `"mus" +? "s" = "mus"`.

```
(+?) :: String -> String -> String
```

Chooses suffix (second and third `String`) depending on the last letter of the first `String`.

```
ifEndThen :: (Char -> Bool) -> String ->
             String -> String -> String
```

Conditionally drops the last letter.

```
dropEndIf :: (Char -> Bool) -> String -> String
```

Apply substitution table to string.

```
changes :: [(String, String)] -> String -> String
```

Like `changes`, but applies only to the prefix.

```
changePref :: [(String, String)] -> String -> String
```

Single word form exception.

```
except :: Param a => Finite a -> [(a, String)] -> Finite a
```

Multiple word form exception.

```
excepts :: Param a => Finite a -> [(a, Str)] -> Finite a
```

Merge two paradigm functions.

```
combine :: Param a => Finite a -> Finite a -> Finite a
```

Missing forms exception.

```
missing :: Param a => Finite a -> [a] -> Finite a
```

Only exception, for highly degenerate paradigms.

```
only :: Param a => Finite a -> [a] -> Finite a
```

Single word form variant exception.

```
variant :: Param a => Finite a -> [(a, String)] -> Finite a
```

Multiple word form variants exception.

```
variants :: Param a => Finite a -> [(a, Str)] -> Finite a
```

Missing word form.

```
nonExist :: Str
```

Filters missing forms from inflection table.

```
existingForms :: Table a -> Table a
```

Translates a finite function to a table.

```
table :: Param a => (a -> Str) -> Table a
```

Used to define Param instances.

```
enum :: (Enum a, Bounded a) => [a]
```

A function with the same functionality as `fromEnum`, but for Param.

```
indexVal :: (Eq a, Param a) => a -> Int
```

Lookup in an inflection table.

```
appTable :: Param a => Table a -> a -> Str
```

Selects the first word form in an inflection table.

```
firstForm :: Param a => Table a -> Str
```

Creates a function from list of values (sensitive to order).

```
giveValues :: (Eq a, Param a) => [Str] -> a -> Str
```

Longest common prefix for a list of strings.

```
longestPrefix :: [String] -> String
```

Collects all word forms into a `Str`.

```
formsInTable :: Table a -> Str
```

Apply function to all word forms in table.

```
mapInTable :: (String -> String) -> Table a -> Table a
```

5.15.2 Dictionary.hs

An instance of the `Dict` class provides information on how to construct an entry for a given dictionary type. In particular, it associates a word class identifier to the dictionary type.

```
class Param a => Dict a where
 dictword :: (a -> Str) -> String
 category :: (a -> Str) -> String
 defaultAttr :: (a -> Str) -> Attr
 attrException :: (a -> Str) -> [(a, Attr)]
```

The type for dictionaries.

```
data Dictionary = D [Entry]
```

The type for a dictionary entry.

```
type Entry = (Dictionary_Word,
      Paradigm,
              Category,
              [Inherent],
              Inflection_Table,
              Extra)
```

The type for paradigm identifiers.

```
type Paradigm = String
```

Transforms a typed table to an untyped.

```
prTable :: Param a => Table a -> Table String
```

Removes attributes and extra information from a dictionary.

```
removeAttr :: Dictionary -> [EntryN]
```

The type for full form lexica: a list of word forms together with their analyses and compound attributes.

```
type FullFormLex = [(String, [(Attr, String)])]
```

Group a dictionary into categories; reverses the entries.

```
classifyDict :: Dictionary -> [(Category, [Entry])]
```

Removes attributes and extra information from `Entry`.

```
noAttr :: Entry -> EntryN
```

Translates an inflection function to an `Entry`.

```
entry :: Dict a => (a -> Str) -> Entry
```

Translates an inflection function with inherent information to an `Entry`.

```
entryI :: Dict a => (a -> Str) -> [Inherent] -> Entry
```

Translates an inflection function with extra information to an `Entry`.

```
entryWithInfo :: Dict a => (a -> (Str, Str)) -> Entry
```

Translates an inflection function with extra information and inherent information to an `Entry`.

```
entryWithInfoI :: Dict a => (a -> (Str, Str)) ->
                            [Inherent] -> Entry
```

Translates an inflection function with paradigm identifier to an `Entry`.

```
entryP :: Dict a => (a -> Str) -> Paradigm -> Entry
```

Translates an inflection function with inherent information and paradigm identifier to an `Entry`.

```
entryIP :: Dict a => (a -> Str) ->
                     [Inherent] -> Paradigm -> Entry
```

Translates an inflection function with extra information and paradigm identifier to an `Entry`.

```
entryWithInfoP :: Dict a => (a -> (Str, Str)) ->
                            Paradigm -> Entry
```

Translates an inflection function with extra information, inherent information, and paradigm identifier to an `Entry`.

```
entryWithInfoIP :: Dict a => (a -> (Str, Str)) ->
                             [Inherent] -> Paradigm -> Entry
```

An `Entry` without attributes and extra information.

```
type EntryN = (Dictionary_Word,
               Category,
               [Inherent],
               [(Untyped, Str)])
```

Creates a `Dictionary`.

```
dictionary :: [Entry] -> Dictionary
```

Translate a `Dictionary` to a list of `Entry`:s.

```
unDict :: Dictionary -> [Entry]
```

The number of entries in a dictionary.

```
size :: Dictionary -> Int
```

The number of word forms in a dictionary.

```
sizeW :: Dictionary -> Int
```

Concatenates two dictionaries.

```
unionDictionary :: Dictionary -> Dictionary -> Dictionary
```

Concatenates a list of Dictionaries.

```
unionDictionaries :: [Dictionary] -> Dictionary
```

An empty Dictionary.

```
emptyDict :: Dictionary
```

Translates a `Dictionary` to a `FullFormLex`.

```
dict2fullform :: Dictionary -> FullFormLex
```

A full form lexicon structured around the word identifier.

```
dict2idlex :: Dictionary -> FullFormLex
```

Performs sharing on the strings in the `Dictionary`.

```
shareDictionary :: Dictionary -> Dictionary
```

5.15.3 Print.hs

Prints word forms in `Str`, separated with '/'.

```
prStr :: Str -> String
```

Similar `prStr`, but outputs '*' for missing word forms.

```
prAlts :: Str -> String
```

Creates a constant table.

```
consTable :: Str -> Table String
```

Creates an attributed constant table.

```
consTableW :: Str -> [(String, (Attr, Str))]
```

Print a `show`:ed inflection function to standard output.

```
putFun0 :: Param a => (a -> Str) -> IO ()
```

Print an inflection function to standard output.

```
putFun :: Param a => (a -> Str) -> IO ()
```

Translate a `show`:ed parameter value to one without parenthesis.

```
prFlat :: String -> String
```

Shows all values for the first parameter.

```
prFirstForm :: Param a => Table a -> String
```

Shows one value for the first parameter.

```
prDictForm :: Param a => Table a -> String
```

Another `Str` printing function.

```
prDictStr :: Str -> String
```

Prints a dictionary, removing the attributes.

```
prDictionary :: Dictionary -> String
```

Prints a dictionary in a structured format.

```
prNewDictionary :: Dictionary -> String
```

Writes a full form lexicon to handle.

```
prFullFormLex :: Handle -> FullFormLex -> IO ()
```

Prints attribute to handle.

```
prCompAttr :: Handle -> Attr -> IO ()
```

Generates GF paradigm functions.

```
prGFRes :: Dictionary -> String
```

Prints GF source code.

```
prGF :: Dictionary -> String
```

Generates XML source code.

```
prXML :: Dictionary -> String
```

Prints LexC source code.

```
prLEXC :: Dictionary -> String
```

Prints XFST source code.

```
prXFST :: Dictionary -> String
```

Prints latex tables.

```
prLatex :: Dictionary -> String
```

Prints SQL Code.

```
prSQL :: Dictionary -> String
```

5.15.4 Frontend.hs

The runtime system class.

```
class Show a => Language a where
 name        :: a -> String
 dbaseName   :: a -> String
 composition :: a -> [Attr] -> Bool
 env         :: a -> String
 paradigms   :: a -> Commands
 internDict  :: a -> Dictionary
 tokenizer   :: a -> String -> [Tok]
 wordGuesser :: a -> String -> [String]
```

The type for command maps.

```
type Commands = Map String ([String], [String] -> Entry)
```

An empty command map.

```
emptyC :: Commands
```

Inserts a command into a set of commands.

```
insertCommand :: (String, [String], [String] -> Entry) ->
                 Commands -> Commands
```

Constructs a command map.

```
mkCommands :: [(String, [String], [String] -> Entry)] ->
              Commands
```

Creates a dictionary from the list of paradigms.

```
command_paradigms :: Language a => a -> Dictionary
```

Parses commands.

```
parseCommand :: Language a => a -> String -> Err Entry
```

Lists paradigm names.

```
paradigmNames :: Language a => a -> [String]
```

The number of paradigms.

```
paradigmCount :: Language a => a -> Int
```

Reading external lexicon. Creates empty lexicon if the file does not exist.

```
parseDict :: Language a => a -> FilePath ->
                           IO (Dictionary, Int)
```

Is input string a paradigm name?

```
isParadigm :: Language a => a -> String -> Bool
```

Reads external lexicon.

```
readdict :: Language a => a -> FilePath ->
                          IO ([Entry], Int)
```

Removes comments in String.

```
remove_comment :: String -> String
```

Wrapper functions for the command map.

```
app1 :: (String -> Entry) -> [String] -> Entry
app2 :: (String -> String -> Entry) -> [String] -> Entry
app3 :: (String -> String -> String ->
         Entry) -> [String] -> Entry
app4 :: (String -> String -> String ->
         String -> Entry) -> [String] -> Entry
app5 :: (String -> String -> String -> String ->
         String -> Entry) -> [String] -> Entry
app6 :: (String -> String -> String -> String ->
         String -> String -> Entry) -> [String] -> Entry
app7 :: (String -> String -> String -> String ->
         String -> String -> String -> Entry) ->
         [String] -> Entry
```

Prints to `stderr`.

```
prErr :: String -> IO ()
```

5.15.5 GeneralIO.hs

Outputs UTF8-encoded string.

```
putStrLnUTF8 :: String -> IO ()
```

Writes source format to file.

```
writeLex    :: FilePath -> Dictionary -> IO ()
writeTables :: FilePath -> Dictionary -> IO ()
writeGF     :: FilePath -> FilePath -> Dictionary -> IO ()
writeGFRes  :: FilePath -> FilePath -> Dictionary -> IO ()
writeXML    :: FilePath -> Dictionary -> IO ()
writeXFST   :: FilePath -> Dictionary -> IO ()
writeLEXC   :: FilePath -> Dictionary -> IO ()
writeLatex  :: FilePath -> Dictionary -> IO ()
writeSQL    :: FilePath -> Dictionary -> IO ()
```

The analysis function.

```
analysis :: ([Attr] -> Bool) -> String -> [[String]]
```

Lookup identifiers for a word form.

```
lookupId :: String -> [String]
```

The synthesiser function.

```
synthesiser :: Language a => a -> IO ()
```

The inflection mode function.

```
infMode :: Language a => a -> IO ()
```

The batch inflection mode function.

```
imode :: Language a => a -> IO ()
```

5.15.6 CommonMain.hs

The 'main' function of FM.

```
commonMain :: Language a => a -> IO ()
```

A type for statistics.

```
data Stats = Stats {totalWords :: Int,
                    coveredWords :: Int}
```

Empty statistics.

```
initStats :: Stats
```

5.15.7 CTrie.hs

Constructs a c-trie from a file containing a full form lexicon.

```
buildTrie :: FilePath -> Bool -> IO ()
```

Constructs a C-trie from a Dictionary ADT. Note that the trie is not handled in Haskell, it's a global object in C.

```
buildTrieDict :: Dictionary -> Bool -> IO ()
```

```
buildTrieDictSynt :: Dictionary -> Bool -> IO ()
```

Builds an undecorated trie.

```
buildTrieWordlist :: [String] -> Bool -> IO ()
```

```
trie_lookup :: String -> [(Attr, String)]
```

Is the string a member in the trie?

```
isInTrie :: String -> Bool
```

Function for compound analysis.

```
decompose :: ([Attr] -> Bool) -> String -> [[(Attr, String)]]
```

Bibliography

[1] *ISO/IEC 9075 Information Technology–Database Languages–SQL*. 1999.

[2] K. R. Beesley and L. Karttunen. *Finite State Morphology*. CSLI Publications, Stanford University, United States, 2003.

[3] G. Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional Programming*, 15,4:573–614, 2005. `http://yquem.inria.fr/~huet/PUBLIC/tagger.pdf`.

[4] A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

[5] The World Wide Web Consortium. Extensible Markup Language (XML). `http://www.w3.org/XML/`, 2000.

Part III

Extract

Chapter 6

Morphological Lexicon Extraction from Raw Text Data

Authors:
Markus Forsberg, Harald Hammarström and Aarne Ranta
{markus,harald2,aarne}@cs.chalmers.se
Department of Computing Science
Chalmers University of Technology
Sweden

Paper published:
FinTAL 2006, LNAI 4139, pp.488-499

abstract

The tool *extract* enables the automatic extraction of lemma-paradigm pairs from raw text data. The tool uses search patterns that consist of regular expressions and propositional logic. These search patterns define sufficient conditions for including lemma-paradigm pairs in the lexicon, on the basis of word forms occurring in the data. This paper explains the search pattern syntax of *extract* as well as the search algorithm, and discusses the design of search patterns from the recall and precision point of view.

The *extract* tool was developed for morphologies defined in the *Functional Morphology* tool [1], but it is systems that implement a word-and-paradigm description of a morphology.

The usefulness of the tool is demonstrated by a case study on the Canadian Hansards Corpus of French. The result is evaluated in terms of precision of the extracted lemmas and statistics on coverage and rule productiveness. Competitive extraction figures show that human-written rules in a tailored tool is a time-efficient approach to the task at hand.

6.1 Introduction

A wide-coverage morphological lexicon is a key part of any information retrieval system, machine translation engine and of a variety of other Natural Language Processing applications. The demand is high not only for low-density languages, since existing lexica for major languages are often not publicly available. Moreover, even if they were, running text – especially newspapers and technical texts – will always contain new, not necessarily hapax, words.

Manual development of a full-scale lexicon is a time-consuming task, so it is natural to investigate how the lexicon development can be automated. The situation is usually such that access to large collections of raw language data is cheap, so cheap that it is tempting to look at ways to exploit the raw data to obtain the sought after high-quality morphological lexicon. Clearly, attempts to fully automatize the process (e.g [2, 3] – most other systems for unsupervised learning of morphology cannot be used directly to build a lexicon) do not reach the kind of quality we are generally interested in. However, instead of using humans for supervised learning of lexicon extraction in some form, we believe there is a more advantageous placement of the human role. With a suitable tool, humans can use their knowledge to guide a computerized extraction from raw text, with comparatively little time spent.

To be more specific, we intend to show that a profitable role for the human is to write intelligent extraction rules. The *extract* tool has been developed with this in mind. The idea behind *extract* is simple: start with a large-sized corpus and a description of the word forms in the paradigms with the varying parts, which we refer to as *technical stems*, represented with variables. In the tool's syntax, we could describe the first declension noun of Swedish with the following definition.

```
paradigm decl1 =
  x+"a"
  { x+"a"  & x+"as"  & x+"an"  & x+"ans" &
    x+"or" & x+"ors" & x+"orna & x+"ornas" } ;
```

⟨*Def* ⟩ ::= `paradigm` ⟨*Name* ⟩ ⟨*VarDef* ⟩ =
⟨*Head* ⟩ { ⟨*Logic* ⟩ }

| `regexp` ⟨*Name* ⟩ = ⟨*Reg* ⟩

Figure 6.1: Regexp and paradigm definitions

Given that all forms in the curly brackets, called the *constraint*, are found for some prefix `x`, the tool outputs the *head* `x+"a"` tagged with the name of the paradigm. E.g., if these forms exist in the text data: `ärta`, `ärtas`, `ärtan`, `ärtans`, `ärtor`, `ärtors`, `ärtorna` and `ärtornas`, the tool will output `decl1 ärta`. Given that we have the lemma and the paradigm class label, it is a relatively simple task to generate all word forms.

The paradigm definition has a major drawback: very few lemmas appear in all word forms. It could in fact be relaxed to increase recall without sacrificing precision: to identify a Swedish word as a noun of the first declension it is often enough to find one instance of the four singular forms and one of the four plural forms. The tool offers a solution by supporting propositional logic in the constraint, further described in Sec. 6.2.1. Various issues of the extraction process are discussed in Sec. 6.3.

Another problem with the given definition is the lack of control over what the variable `x` might be. Sec. 6.2.2 describes how the tool improves this situation by allowing variables to be associated with regular expressions.

The stems of first declension nouns in Swedish are the same for all word forms, but this is not the case for many paradigms, e.g. German nouns with umlaut. Sec. 6.2.3 presents the tool's use of multiple variables as a solution to this problem.

6.2 Paradigm File Format

A paradigm file consists of two kinds of definitions: `regexp` and `paradigm`. The syntax is given in Fig. 6.1.

A `regexp` definition associates a name (`Name`) with a regular expression (`Reg`). A `paradigm` definition consists of a name (`Name`), a set of variable-regular expression associations (`VarDef`), a set of output constituents (`Head`) and a constraint (`Logic`).

The basic unit in `Head` and `Logic` is a *pattern* that describes a word form. A pattern consists of a sequence of variables and string literals glued together with the '+' operator. An example of a pattern given previously

⟨*Logic* ⟩ ::= ⟨*Logic* ⟩ `&` ⟨*Logic* ⟩
| ⟨*Logic* ⟩ | ⟨*Logic* ⟩
| ⟨*Logic* ⟩
| `~` ⟨*Logic* ⟩
| ⟨*Pattern* ⟩
| `(` ⟨*Logic* ⟩ `)`

Figure 6.2: Propositional logic grammar

was `x+"a"`.

Both definitions will be discussed in detail in the following sections.

6.2.1 Propositional Logic

Propositional logic appears in the constraint to enable a more fine-grained description of what word forms the tool should look for. The basic unit is a pattern, corresponding to a word form, which is combined with the operators `&` (*and*), `|` (*or*), and `~` (*not*).

The syntax for propositional logic is given in Fig. 6.2, where *Pattern* refers to one word form.

The addition of new operators allow the paradigm in Sec. 6.1 to be rewritten with disjunction to reflect that it is sufficient to find one singular and one plural word form.

```
paradigm decl1 =
      x+"a"
      { (x+"a"   | x+"as"  | x+"an"  | x+"ans") &
        (x+"or"  | x+"ors" |x+"orna  | x+"ornas") } ;
```

6.2.2 Regular Expressions

It was mentioned in Sec. 6.1 that control over the variable part of a paradigm description was desired. The solution provided by the tool is to enable the user to associate every variable with a regular expression. The association dictates which (sub-)strings a variable can match. An unannotated variable can match any string, i.e. its regular expression is Kleene star over any symbol.

As a simple example, consider German, where nouns always start with an uppercase letter. This can be expressed as follows.

```
regexp UpperWord = upper letter*;

paradigm n [x:UpperWord] = ... ;
```

⟨*Reg* ⟩ ::= ⟨*Reg* ⟩ | ⟨*Reg* ⟩
| ⟨*Reg* ⟩ − ⟨*Reg* ⟩
| ⟨*Reg* ⟩ ⟨*Reg* ⟩
| ⟨*Reg* ⟩ `*`
| ⟨*Reg* ⟩ +
| ⟨*Reg* ⟩ `?`
| `eps`
| ⟨*Char* ⟩
| `digit`
| `letter`
| `upper`
| `lower`
| `char`
| ⟨*String* ⟩
| `(` ⟨*Reg* ⟩ `)`

Figure 6.3: Regular expression

The syntax of the tool's regular expressions is given in Fig. 6.3, with the normal connectives: union, concatenation, set minus, Kleene star, Kleene plus and optionality. *eps* refers to the empty string, *digit* to 0−9, *letter* to an alphabetic Unicode character, *lower* and *upper* to a lowercase respectively an uppercase letter. *char* refers to any character. A regular expression can also contain a double-quoted string, which is interpreted as the concatenation of the characters in the string.

6.2.3 Multiple Variables

Not all paradigm definitions are as neat as the initial example — phenomena like *umlaut* require an increased control over the variable part. The solution the tool provides is to allow multiple variables, i.e. a pattern may contain more than one variable. This is best explained with an example, where two German noun paradigms are described, both with umlaut. The change of the stem vowel is captured by introducing two variables and by letting the stem vowel be a constant string.

```
regexp Consonant = ... ;

regexp Pre = upper letter*;

regexp Aft = Consonant+ ;
```

```
paradigm n2 [F:Pre, ll:Aft] =
  F+"a"+ll
    { F+"a"+ll & F+"ä"+ll+"e" } ;

paradigm n3 [W:Pre, rt:Aft] =
   W+"o"+rt
    { W+"o"+rt & W+"ö"+rt+"er" } ;
```

The use of variables may reduce the time-performance of the tool, since every possible variable binding is considered. The use of multiple variables should be moderate, and the variables should be restricted as much as possible by their regular expression association to reduce the search space.

A variable does not need to occur in every pattern, but the tool only performs an initial match with patterns containing all variables. The reason for this is efficiency — the tool only considers one word at the time, and if the word matches one of the patterns, it searches for all other patterns with the variables instantiated by the initial match. For obvious reasons, an initial match is never performed under a negation, since this would imply that the tool searches for something it does not want to find.

It is allowed to have repeated variables, i.e. non-linear patterns, which is equivalent to *back reference* in the programming language Perl. An example where a sequence of bits is reduplicated is given. This language is known to be non-context-free [4].

```
regexp ABs = (0|1)*;

paradigm reduplication [x:ABs] =
   x+x { x+x } ;
```

6.2.4 Multiple Arguments

The head of a paradigm definition may have multiple arguments to support more abstract paradigms. An example is Swedish nouns, where many nouns can be correctly classified by just detecting the word forms in nominative singular and nominative plural. An example is given below, where the first and second declension is handled with the same paradigm function, where the head consists of two output forms. The constraints are omitted.

```
paradigm regNoun =
flick+"a" flick+"or"
{...} ;
```

```
paradigm regNoun =
pojk+"e" pojk+"ar"
{...} ;
```

6.2.5 The Algorithm

The underlying algorithm of the tool is presented in pseudo-code notation.

```
let L be the empty lexicon.
let P be the set of extraction paradigms.
let W be all word types in the corpus.
for each w : W
 for each p : P
  for each constraint C with which w matches p
   if W satisfies C with the result H,
    add H to L
   endif
  end
 end
end
```

The algorithm is initialized by reading the word types of the corpus into an array W. A word w *matches* a paradigm p, if it can match any of the patterns in the paradigm's constraint that contains all variables occurring in the constraint. The result of a successful match is an *instantiated constraint* C, i.e. a logical formula with words as atomic propositions. The corpus W *satisfies* a constraint C if the formula is true, where the truth of an atomic proposition a means that the word a occurs in W.

6.2.6 The Performance of the Tool

The extraction tool is implemented in Haskell. It is available as an open-source free software [1]. A typical example of using the tool, the experiment reported in Sec. 6.4 extracted a lexicon of 19,295 lemmas from a corpus of 66,853 word types, by using 43 paradigms. The execution time was 11min 23s on a computer with an AMD 3600+ CPU and 1 GB memory, running Kubuntu Linux 5.10. The memory consumption was 34 MB.

6.3 The Art of Extraction

The constraint of a paradigm describes a sub-paradigm, a subset of the word forms, considered to be evidence enough to be able to judge that the lemmas in the head are in that paradigm class. The identification of appropriate sub-paradigms requires good insights into the target language

[1] Extract homepage: `http://www.cs.chalmers.se/~markus/extract/`

and intuitions about the distributions of the word forms. However, these insights and intuitions may be acquired while using the tool by trial and error.

Lexicon extraction is a balance between *precision*, i.e. the percentage of the extracted lemmas that are correctly classified, and *recall*, i.e. the percentage of the lemmas in the text data that are extracted. Precision, however, is by far the most important, since poor recall can be compensated with more text data, but poor precision requires more human labor.

How about extracting the paradigm descriptions from a set of paradigms automatically? We use the term *minimum-size sub-paradigm* to describe the minimum-sized set of word forms needed to uniquely identify a paradigm P. More formally, a minimum-sized sub-paradigm is a minimum-size set of word forms $P' \subseteq P$ such that for any other paradigm Q, $P' \not\subseteq Q$. It turns out that the problem of finding the minimum-size sub-paradigm for a paradigm P is NP-complete[2]. Furthermore, the minimum-size sub-paradigm need not be of high practical interest since it may contain forms that are very uncommon in actual usage. Therefore there is all the more reason to let a human choose which forms to require and also weigh in which forms are likely to be common or uncommon in actual usage.

Also, some natural languages have *overshadowed paradigms*, i.e. paradigms where the form of one paradigm is a subset of another paradigm. For example, in Latin some noun paradigms are overshadowed by adjective paradigms. The distinction of Latin nouns and adjectives can be done through the use of negation where a second declension noun paradigm is defined by also stating that the feminine endings, which would indicate that it is an adjective, should not be present. This definition, however, misses e.g. *filius* where the feminine parallel *filia* does exist.

```
paradigm decl2fungus =
       fung++"us"
   { fung+"us" & fung+"i" & ~(fung+"a" | fung+"ae") };
```

Negation is similar with *negation as failure* in Prolog, with the same problems associated with it. The main problem is that negation rests on the absence, not the presence, of information, which in turn means that the extraction process with negation is non-monotonic: the use of a larger corpus may lead to an extracted lexicon which is smaller. A worst-case scenario is a misspelt or foreign word that, by negation, removes large parts of the correctly classified lemmas in the extracted lexicon.

[2]The minimum-size sub-paradigm problem (MSS) is equivalent to the well-known set-cover problem. Proof omitted.

In most cases, a better alternative to negation is a more careful use of regular expressions, and in cases like Latin nouns, a rudimentary POS tagger that resolves the POS ambiguity may outperform negation.

6.3.1 Manual Verification

Almost all corpora have misspellings which may lead to false conclusions. Added to that are word forms that incidentally coincide. One possible solution to handle misspellings is to only consider words that occur at some frequency. However, that would remove a lot of unusual but correctly spelled words (to an extent which is unacceptable). Coincidences are in practice impossible to avoid.

Misspellings, foreign words and coincidences are the reason why manual verification of the extracted lexicon cannot be circumvented even with "perfect" paradigm definitions. However, browse-filtering a high-precision extracted lexicon requires much less time than building the same lexicon by hand. Also, nothing in principle prohibits statistical techniques to be applied in collaboration here. For instance, one can sort the extracted lemmas heuristically according to how many forms and with what frequencies they occur (cf. Sec. 6.5). In general, this is productive for poly-occurring lemmas but helps little for the (typically many) hapax lemmas.

6.4 Experiments

We will evaluate our proposed extraction technique with a study of real-world extraction on the Hansards corpus of Canadian French [5]. All words were manually annotated to enable a thorough evaluation. However, the intended practical usage of the extraction tool is to simply run the tool on the raw text data and eye-browse the output list for erroneous extractions.

The corpus consisted of approximately 15 million running tokens of 66853 types. From these 66853 types we manually removed all junk – foreign words, proper names, misspellings, numeric expressions, abbreviations as well as pronouns, prepositions, interjections and non-derived adverbs – so that a 49477 true lexical items remained. 27681 lemmas account for the 49477 forms, where verb lemmas tended to occur in more forms than noun and adjective lemmas. Of course, not all these lemmas occurred in such forms that their morphological class could be recognized by their endings alone. Many lemmas occur in only one form – usually not enough to infer its morphological class – unless, as is often the case, they contain a derivational morpheme which, together with its inflectional ending, does suffice. For

Tokens	15 000 000
Types	66 853
Non-junk types	49 477
Lemmas	27 681

Figure 6.4: Statistics on the corpus of Canadian French Hansards used in the experiment

example, a single occurrence of a word ending in *-e* is hardly conclusive, whereas one ending in *-tude* is almost certainly a feminine noun with a plural in *-s*. Nouns without derivational ending cannot be reliably distinguished from adjectives even when they occur in all their forms, i.e. both the singular and plural. The table in Fig. 6.4 summarizes these data.

We now turn to the question of precision and coverage of rule-extraction of the targeted 27 681 lemmas. We quickly devised a set of 43 rules to extract French nouns (18 rules), verbs (7 rules) and adjectives (18 rules). The verb-rules aimed at *-ir* and *-er* verbs by requiring salient forms for these paradigms, whereas the noun- and adjective rules make heavy use of regularities in derivational morphology to overcome the problems of overlapping forms. Two typical example groups are given below:

```
regexp NOTi = char* (char-"i") ;

paradigm Ver [regard:NOTi]
  = regard+"er"
    {regard+"e" &
     (regard+"é"   | regard+"ée"  |
      regard+"ez"  | regard+"ont" |
      regard+"ons" | regard+"a" )} ;

 paradigm Aif
  = sport+"if"
    {sport+"if"  | sport+"ifs" |
     sport+"ive" | sport+"ives"} ;
```

The results of the extraction are shown in Fig. 6.5. If possible, one would like to know where one's false positives come from – sloppy rules or noisy data? At least one would like to know roughly what to expect. Since we have already annotated this corpus we can give some indicative quantitative data. To assess the impact of misspellings and foreign words – the two main sources for spurious extractions – we show the results of the same extraction

	Extr. All	Extr. Non-Junk
False Positives	2 031	664
Correctly Indentified	17 264	17 264
	19 295	17 928
Precision	89.5%	96.3%

Figure 6.5: Extraction results on raw text vs. text with junk removed first.

performed on the corpus *with all junk removed beforehand*. As expected, false positives increase when junk is added. To be more precise, we get a lot of spurious verbs from English words and proper names in *-er* (e.g farmer, worchester) as well as many nouns, whose identification requires only one form, from misspellings (e.g qestion). Non-junk-related cases of confusion worth mentioning are nouns in *-ment* – the same ending as adverbs – and verbs which have spelling changes (manger-mangeait, appeler-appelle etc).

The rule productiveness, i.e a rule on average catches $17264/43 \approx 401$, must be considered very high. As for coverage, we can see that our rules catch the lions share of the available lemmas, 17 264 out of 27 681 (again, not all of which occur in enough forms to predict their morphological class), in the corpus. This is relevant because even if we can always find more raw text cheaply, we want our rules to make maximal use of whatever is available and more raw data is of little help unless we can actually extract a lot of its lemmas with reasonable effort. It is also relevant because a precision figure without a rule productiveness figure is meaningless. It would be easy to tailor 43 rules to perfect precision, perhaps catching one lemma per rule, so what we show is that precision and rule productiveness can be simultaneously high. In general it is of course up to the user how much of the raw-data lemmas to sacrifice for precision and rule-writing effort, which are usually more important objectives.

6.5 Related Work

The most important work dealing with the very same problem as addressed here, i.e. extracting a morphological lexicon given a morphological description, is the study of the acquisition of French verbs and adjectives in Clément et al. [6]. Likewise, they start from an existing inflection engine and exploit the fact that a new lemma can be inferred with high probability if it occurs in raw text in predictable morphological form(s). Their algorithm ranks

hypothetical lemmas based on the frequency of occurrence of its (hypothetical) forms as well as part-of-speech information signalled from surrounding closed-class words. They do not make use of human-written rules but reserve an unclear, yet crucial, role for the human to hand-validate parts of output and then let the algorithm re-iterate. Given the many differences, the results cannot be compared directly to ours but rather illustrate a complementary technique.

Tested on Russian and Croat, Oliver et al. [7, 8, Ch. 3] describe a lexicon extraction strategy very similar to ours. In contrast to human-made rules, they have rules extracted from an existing (part of) a morphological lexicon and use the number of inflected forms found to heuristically choose between multiple lemma-generating rules (additionally also querying the Internet for existence of forms). The resulting rules appear not at all as sharp as hand-made rules with built-in human knowledge of the paradigms involved and their respective frequency (the latter being crucial for recall). Also, in comparison, our search engine is much more powerful and allows for greater flexibility and user convenience.

For the low-density language Assamese, Sharma et al. [3] report an experiment to induce both morphology, i.e. the set of paradigms, and a morphological lexicon at the same time. Their method is based on segmentation and alignment using string counts only – involving no human annotation or intervention inside the algorithm. It is difficult to assess the strength of their acquired lexicon as it is intertwined with induction of the morphology itself. We feel that inducing morphology and extracting a morphological lexicon should be performed and evaluated separately. Many other attempts to induce morphology, usually with some human tweaking, from raw corpus data (notably Goldsmith [9]), do not aim at lexicon extraction in their current form.

There is a body of work on inducing verb subcategorization information from raw or tagged text (see [10, 11, 12] and references therein). However, the parallel between subcategorization frame and morphological class is only lax. The latter is a simple mapping from word forms to a paradigm membership, whereas in verb subcategorization one also has the onus discerning which parts of a sentence are relevant to a certain verb. Moreover, it is far from clear that verb subcategorization comes in well-defined paradigms – instead the goal may be to reduce the amount of parse trees in a parser that uses the extracted subcategorization constraints.

6.6 Conclusions and Further Work

We have shown that building a morphological lexicon requires relatively little human work. Given a morphological description, typically an inflection engine and a description of the closed word classes, such as pronouns and prepositions, and access to raw text data, a human with knowledge of the language can use a simple but versatile tool that exploits word forms alone. It remains to be seen to what extent syntactic information, e.g part-of-speech information, can further enhance the performance. A more open question is whether the suggested approach can be generalized to collect linguistic information of other kinds than morphology, such as e.g verb subcategorization frames.

Bibliography

[1] Forsberg, M., Ranta, A.: Functional Morphology. Proceedings of the Ninth ACM SIGPLAN International Conference of Functional Programming, Snowbird, Utah (2004) 213–223

[2] Creutz, M., Lagus, K.: Inducing the morphological lexicon of a natural language from unannotated text. In: Proceedings of the International and Interdisciplinary Conference on Adaptive Knowledge Representation and Reasoning (AKRR '05), 15-17 June, Espoo, Finland, Espoo (2005) 106–113

[3] Utpal Sharma, J.K., Das, R.: Unsupervised learning of morphology for building lexicon for a highly inflectional language. In: Proceedings of the 6th Workshop of the ACL Special Interest Group in Computational Phonology (SIGPHON), Philadelphia, July 2002, Association for Computational Linguistics (2002) 1–10

[4] Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)

[5] Germann, U.: Corpus of hansards of the 36th parliament of canada. Provided by the Natural Language Group of the Universtity of Southern California Information Sciences Institute. Downloadable at `http://www.isi.edu/natural-language/download/hansard/`, accessed 1 Nov 2005. (2003) 15 million words.

[6] Clément, L., Sagot, B., Lang, B.: Morphology based automatic acquisition of large-coverage lexica. In: Proc. of LREC'04, Lisboa, Portugal (2004) 1841–1844

[7] Oliver, A., Tadić, M.: Enlarging the croatian morphological lexicon by automatic lexical acquisition from raw corpora. In: Proc. of LREC'04, Lisboa, Portugal (2004) 1259–1262

[8] Oliver, A.: Adquisició d'informació lèxica i morfosintàctica a partir de corpus sense anotar: aplicació al rus i al croat. PhD thesis, Universitat de Barcelona (2004)

[9] Goldsmith, J.: Unsupervised learning of the morphology of natural language. Computational Linguistics **27**(2) (2001) 153–198

[10] Kermanidis, K.L., Fakotakis, N., Kokkinakis, G.: Automatic acquisition of verb subcategorization information by exploiting minimal linguistic resources. International Journal of Corpus Linguistics **9**(1) (2004) 1–28

[11] Faure, D., Nédellec, C.: Asium: Learning subcategorization frames and restrictions of selection. In Kodratoff, Y., ed.: 10th Conference on Machine Learning (ECML 98) – Workshop on Text Mining, Chemnitz, Germany, Avril 1998. Springer-Verlag, Berlin (1998)

[12] Gamallo, P., Agustini, A., Lopes, G.P.: Learning subcategorisation information to model a grammar with "co-restrictions". Traitement Automatique des Langues **44**(1) (2003) 93–177

Chapter 7

The Extract Tool

Author:
Markus Forsberg
Department of Computing Science
Chalmers University of Technology
and the University of Gothenburg
markus@cs.chalmers.se

Paper published:
Technical Report no. 2007-10 in Computing Science at Chalmers
University of Technology and Göteborg University

7.1 Introduction

This technical report describes *Extract v2.0*, a tool for extracting linguistic information from raw text data, in particular inflectional information on words, based on the word forms appearing in the text data. This document is a revision of the *Extract manual* describing *Extract v1.0*, a previous version of the tool [1]. The main differences between v1.0 and v2.0 are the addition of Constraint Grammar together with a new data format, and an upgrade of the regular expression engine. The new engine has changed the semantics of the regular expressions: `letter`, `upper`, and `lower` no longer refer to Unicode letters, but to English letters. However, this small loss of functionality is well compensated by the improved efficiency.

The input of Extract is a file containing a, possibly unannotated, corpus and a file containing Extract rules. Each rule provides a *search template* for some linguistic information, such as regular nouns of English. If a rule's

search template is applicable to a set of word forms in the text data then the tool outputs the *head* of the rule. The head of the rule specifies the rule identifier and the output word forms.

The tool's output is a list of analyses, each analysis consists of a sequence of words annotated with an identifier. The identifier states some linguistic information, e.g. `regNoun hat` may encode that the word `hat` is a regular noun, and `v2 eat` may encode that `eat` is a transitive verb.

The previous version of the tool viewed the input data as a set of words with no contextual information. The focus was solely on annotating words with paradigm identifiers, hence the rules were marked with the keyword `paradigm`. The new version of Extract, presented in this document, allows contextual information in the rules, expressed with a variant of the *Constraint Grammar* formalism [3]. This addition allows more involved linguistic information to be extracted, such as subcategorization frames for verbs. This change is reflected through the change of the keyword `paradigm` to the more generic keyword `rule`. The old keyword is still usable for backward compatibility reasons.

7.2 Lexicon Extraction

We start our discussion with *lexicon extraction* — how *citation forms* marked with *inflectional information* can be extracted from a corpus. We will present our tool incrementally, trying to motivate its different features by presenting problems that need to be resolved.

Citation forms, or dictionary forms, are those word forms typically found in a normal dictionary. They represent a word, or a word's inflection table, and are usually the word forms that are the most unmarked, i.e. perceived as most neutral, or the most characteristic.

Paradigms are abstractions from inflection tables. A paradigm identifier together with the citation forms is enough to produce the complete inflection table of a word.

The objective is to extract citation forms annotated with identifiers.

A first approach is to search for all word forms of a paradigm, where the stem is replaced with a variable. We then traverse the input data searching for instantiations of the variable. This idea is illustrated in the syntax of Extract with the paradigm of Swedish first declension noun as example.

```
rule decl1 =
 x+"a"
 { x+"a"  & x+"as"  & x+"an"   & x+"ans" &
   x+"or" & x+"ors" & x+"orna" & x+"ornas" } ;
```

The rules of Extract consists of an *identifier*, a *body* and a *head*. The body consists of a search template inside curly brackets. Given that all word forms are found in the search template for some string `x`, then the head `x+"a"` will be output tagged with the identifier `decl1`.

Stated more concretely, if we have the inflection table of the Swedish word *smula* (Eng. 'crumb') in our input data, that is, the word forms *smula*, *smulas*, *smulan*, *smulans*, *smulor*, *smulors*, *smulorna* and *smulornas*, then the output would be `decl1 smula`.

If the rule `decl1` is defined in a file `rules`, and if we collect some Swedish text and put it in a file `Swedish_text`, then Extract is runnable with those files as arguments (see Fig. 7.1). Two additional flags are supplied, `-u`, for no duplicates, and `-utf8`, for UTF-8 encoding. The text we are using in this example is the complete set of word forms for the words *smula* (Eng. 'crumb'), *människa* (Eng. 'human') and *flicka* (Eng. 'girl'), all first declension nouns.

The result of the run is, as expected, the three words in the input data.

This looks straightforward enough, but unfortunately, words in a language with non-trivial morphology, such as Swedish, rarely occur in all word forms. Furthermore, it is often difficult to select a subset of word forms that are the most representative for a particular paradigm, if at all possible. For example, in the rule `decl1`, it does not matter if a word form is in the nominative or genitive case, since the case inflection is the same for all declensions. But, restricting the search to just one of the cases would be completely arbitrary. The tool supports propositional logic in the constraints to allow more fine-grained descriptions, such as using disjunction for the case distinction. The use of propositional logic in Extract is described in more detail in Sec. 7.2.1.

In the rule `decl1`, there is no control over which substrings may be associated to the variable `x`. We may, for example, want to state that `x` should at least be monosyllabic to avoid spurious outputs. Sec. 7.2.2 describes how the tool improves this situation by allowing variables to be associated with a regular expression.

The stem in rule `decl1` is the same for all word forms. This is not the case for many paradigms, such as paradigms with umlaut. In Sec. 7.2.3 we describes how such paradigms can be defined by the use of multiple variables.

```
$ extract -u -utf8 rules Swedish_text

*****************************************
*           Lexicon Extraction          *
*                 with                  *
*           Constraint Grammar          *
*****************************************
* (c) Markus Forsberg & Aarne Ranta 2007 *
* under GNU General Public License.      *
*****************************************

1 rule read from 'rules'.

Reading raw text data from 'Swedish_text'...

decl1 flicka
decl1 människa
decl1 smula

Unique tokens     : 24
Corpus Usage      : 100.00%
Words Extracted   : 3
```

Figure 7.1: Example run

⟨*Logic* ⟩ ::= ⟨*Logic* ⟩ `&` ⟨*Logic* ⟩
| ⟨*Logic* ⟩ `|` ⟨*Logic* ⟩
| ⟨*Logic* ⟩
| `~` ⟨*Logic* ⟩
| ⟨*Pattern* ⟩
| `(` ⟨*Logic* ⟩ `)`

Figure 7.2: Propositional logic

7.2.1 Propositional Logic

Propositional logic is used in the body of a rule to enable a more fine-grained description of which word forms the tool should look for. The basic unit is a *pattern*, corresponding to a word form, which are the atoms of a propositional logic formula. The formula is referred to as a *search template*. A formula is built with three connectives with their conventional meanings: *conjunction* (`&`), *disjunction* (`|`) and *negation* (`~`). The syntax is given in Fig. 7.2, without precedence and associativity information. For a complete reference, see Sec. 7.10.

Note that negation in a search template is true when no counter-example is found, i.e. it is *negation as failure*. We try to prove p, and when we fail, we conclude $\sim p$. The use of negation in lexicon extraction makes it *non-monotonic* — the addition of more data may lead to a smaller result set.

The rule in Sec. 7.2 can be rewritten with propositional logic to reflect that it is sufficient to find a word form either in the nominative or genitive case by the use of the disjunctive operator.

```
rule decl1 =
      x+"a"
      {( x+"a"  | x+"an"  | x+"or"  | x+"orna") &
       ( x+"as" | x+"ans" | x+"ors" | x+"ornas")} ;
```

The word forms in the head are not necessary in the input data, they may be constructed from the instantiated variables and constant strings.

7.2.2 Regular Expressions

In Sec. 7.2 we mentioned that it is necessary to increase the control over what substrings a rule's variables may be associated with — allowing a variable to be associated to any substring may seriously degrade the performance of the tool. For example, in Swedish it is necessary to require that the variable

corresponding to the stem contains, at least, one vowel to avoid many false positives.

The solution Extract provides is to enable variables to be associated to regular expressions describing which strings the variable can match. An unannotated variable matches any string, i.e. its regular expression is Kleene star over all characters.

Let us define the stems of German nouns, exemplified with the rule `noun`. A German noun begins with an uppercase letter, which we express easily with a regular expression. In this example, it is necessary that the word forms in the rule are capitalized to avoid capturing verbs.

```
regexp GermanUpper = "Ä" | "Ü" | "Ö" | "ß" | upper ;
regexp GermanLower = "ä" | "ü" | "ö" | lower ;
regexp UpperWord   = GermanUpper GermanLower*;

rule noun [x:UpperWord] =
          { x+"e" & x+"en" } ;
```

The syntax of the tool's regular expression is given in Fig. 7.3, with the normal connectives: union, concatenation, set minus, Kleene's star, Kleene's plus and optionality. *eps* refers to the empty string, *digit* to $0-9$, *letter* to an alphabetic character (a-z, A-Z), *lower* and *upper* to lowercase (a-z) and uppercase (A-Z), and *char* to any character. A regular expression can also contain a double quoted string, which is interpreted as the concatenation of the characters in the string.

7.2.3 Multiple Variables

Not all paradigms are as neat as the initial example — phenomena like *umlaut* require increased control over the variable part. The solution the tool provides is to allow multiple variables, i.e. that a pattern can have more than one variable. This is best explained with an example, here with the rules of two German nouns.

```
regexp GermanUpper = "Ä" | "Ü" | "Ö" | "ß" | upper ;
regexp GermanLower = "ä" | "ü" | "ö" | lower ;
regexp Pre         = GermanUpper GermanLower ;
regexp Whatever    = GermanLower* ;

rule n2 [F:Pre, ll:Whatever] =
     F+"a"+ll
    {F+"a"+ll & F+"ä"+ll+"e"} ;
```

⟨*Reg* ⟩ ::= ⟨*Reg* ⟩ | ⟨*Reg* ⟩
| ⟨*Reg* ⟩ − ⟨*Reg* ⟩
| ⟨*Reg* ⟩ ⟨*Reg* ⟩
| ⟨*Reg* ⟩ `*`
| ⟨*Reg* ⟩ +
| ⟨*Reg* ⟩ `?`
| `eps`
| ⟨*Char* ⟩
| `digit`
| `letter`
| `upper`
| `lower`
| `char`
| ⟨*String* ⟩
| `(` ⟨*Reg* ⟩ `)`

Figure 7.3: Regular expression syntax

```
rule n3 [W:Pre, rt:Whatever] =
     W+"o"+rt
    {W+"o"+rt & W+"ö"+rt+"er"} ;
```

The use of multiple variables did previously reduce the performance of the tool. This is no longer true in Extract v2.0, as long as no back reference is used, due to a new regular expression back end.

It is not required that all variables occur in every pattern, but the tool performs an initial match only on patterns that contains all variables. The reason for this is efficiency — the tool only considers one word at a time, and if the word matches one of the patterns, it searches for all other patterns with the variables instantiated by the initial match. For obvious reasons, an initial match is never performed under a negation (this would imply that the tool searches for something it does not want to find!).

7.2.4 Multiple Output Patterns

The head of a rule may have multiple output patterns, which support more abstract rules. An example is Swedish nouns, where many nouns can be correctly classified by just analyzing the word forms in nominative singular and nominative plural. The first and second declensions are handled with the same paradigm function, and the head consists of two output patterns. The constraints are omitted.

```
rule regNoun =
        flick+"a" flick+"or"
        {...} ;

rule regNoun =
        pojk+"e" pojk+"ar"
        {...} ;
```

7.3 Structured Input Data

The input data of Extract v2.0 can either be raw text data or structured data. Structured data consists of a list of sequences of tokens, *chunks*, where chunks correspond to a meaningful context. A token is a word form associated with a list of *terms*, referred to as the *ambiguity class*. A term is a non-atomic structured information unit, consisting of labels and applications of labels. Here is an example of two chunks: the first one consisting of two unambiguous tokens, and the second one of four tokens, two unambiguous tokens, one ambiguous (`sticka`), and one token without analysis (`fingret`).

```
{ ("hej",in)
  ("!",spec) }
{ ("sticka",nn sg indef nom u|vb inf aktiv|vb imper)
  ("i",pr)
  ("fingret",)
  (".",spec) }
```

A chunk may be a sentence, a phrase or some other appropriate unit. Raw text data is also divided into chunks, where the dividers are major punctuations. Although a naive approach, it works quite well, since the garbage generated by, e.g. abbreviations, does not normally give rise to any spurious analyses.

A chunk is used to generate the context of a token, which can be referred to with Constraint Grammar (CG) constructs, explained Sec. 7.4. If no CG constructs appear in any rule, then the chunks are left unused, and the tool acts in the same manner as before.

A morphological lexicon typically includes more information about the word forms than just part of speech, e.g. inflectional parameters, such as number or case, and inherent parameters, such as gender. Because of this, it is natural to allow non-atomic class labels that can be partially specified. This is done by a simple term language. When we refer to non-atomic class labels, we can use a *don't care* symbol _ to state that a part of a class label is irrelevant.

If a lexical resource in FM is available, then structured input data can be produced with the following command. We use the FM implementation of Latin in this example to produce our structured data.

```
$ cat raw_text_data.txt | ./morpho_lat -pos > structured_data.txt
```

This produces a file `structured_data.txt`, consisting of a sequence of chunks, where the tokens have been annotated with the analyses provided by the FM implementation.

7.4 Constraint Grammar

7.4.1 Introduction to Constraint Grammar

Constraint grammar (CG) [3] [4] was first introduced by F. Karlsson, presented as a facility for performing disambiguation and light parsing. Karlsson's Constraint Grammar framework assumes *total knowledge*, i.e. that all possible analyses of a word is already known, and the objective of a Constraint Grammar is to reduce ambiguity. Yet another assumption, related to total knowledge, is the *Sherlock Holmes assumption*, stating that if we remove all possible analyses except one, then that one must be the correct one, no matter how improbable.

The input word forms of a CG have an *ambiguity class* associated to them. The ambiguity classes consist of lists of readings, and the goal of the rules of a CG is to reduce these ambiguity classes.

A CG rule consists of a *domain*, a *target*, an *operator* and a *context*.

The *domain* provides a typing of the rule, declaring which of the tokens that are affected. For example, `@w` declares that all tokens are affected. The *target* declares a result of the rule, its reading, and together with the *operator* of the rule, defines the outcome of the rule if the input word form satisfies the rule's *context*.

Two examples are taken from F. Karlsson [3]. The first rule states that for any word (the domain `@w`) preceded by a word with the reading `TO` (the context), then the target `VFIN` is discarded (stated by the operator `=0`). The second rule contains a operator `=!`, stating that the target reading `"<REL>"` is unambiguous, if the input word form `that` (the domain), if it is preceded by `NOMHEAD` and followed by `VFIN`.

```
1. (@w=0 VFIN (-1 TO))

2. ("that" =! "<REL>" (-1 NOMHEAD) (1 VFIN))
```

7.4.2 Constraint Grammar in Extract Rules

The total knowledge assumption is a reasonable assumption with a large dictionary, but in a lexicon extraction setting it is invalid, since it is the unknown word forms that are interesting. Theoretically, we could fix this by associating all words with all possible analyses, but this would destroy all information provided by our dictionary.

Constraint Grammar in Extract differs from the traditional definition, since the task is no longer to reduce ambiguity, but to classify unknown words.

There is only one, implicit, operator used in the rules, which is not quite the same as any of the operators in CG, since it selects a reading that, possibly, is not part of the ambiguity class. This is natural since the words that we are aiming for are outside the lexicon.

The constraints in the rules are defined with a propositional formula, which already existed in Karlsson's framework in an indirect fashion. Conjunction is implicit in the rules, what he refers to as 'polarity' corresponds to negation, and disjunction can be defined by translating it to a list of rules.

A *context element* is a triple, consisting of a position, a regular expression and a term, as illustrated in Fig. 7.4. Karlsson's CG lacked the possibility to refer to words in the context with regular expressions, which is natural since the assumption is that the ambiguity class contains the correct class, so we normally only need to refer to the classes. Here, we need to be able to refer to the actual words, since we cannot assume that the correct class is included.

An example is given below where we are interested in identifying nouns. The constraint is in square brackets and states that the word in question must be preceded by an article and followed by a verb in present tense. We use the don't care symbol _ for positions that we are not interested in. We also use the uniqueness operator `!` to express that we are only interested in words where the preceding and following words is unambiguous.

```
rule noun x [x:Word] =
  x
  {
   x [(-1,_, ! article _ _) &
      (1, _, ! verb present _ _)]
  }
```

Wild cards (_) can be used anywhere except for positions. A wild card can be matched with anything.

(⟨*Position* ⟩ , ⟨*Reg* ⟩ , ⟨*Unique* ⟩ ⟨*Patt* ⟩)

Figure 7.4: An atom of a constraint

7.4.3 Positions in CG

Positions in a context element of CG are either *absolute*, *unbounded*, *relative* or *unbounded relative*. A position identifies a token in the context of the current token.

Positions are referred to with integers, where the current token is at position 0, the ones to the left have increasingly negative numbers, and the ones to the right have increasingly positive numbers. An absolute position is an integer referring to a token, much in the same way as indexing an array.

Unbounded positions, marked by a star '*', refer to the first hit starting at an absolute position within the current chunk. An unbounded position may furthermore be labelled so the resulting position can be referred to by a relative position. As an example, consider the following two context elements. The first element contains an unbounded position, where the label `t` will be assigned the value of the position of the first occurrence of `the`, if any, starting to the left of the current token. The second element contains a relative position, stating that the word to the right of the first `the` should be `black`.

```
(t@-1*,"the",_) & (t+1,"black",_)
```

Relative positions may also be unbounded. We could change our example of a relative position to `t+1*` to express that the word `black` should be somewhere on the right of `the`, and moreover, we could label it so its position becomes referable: `p@t+1*`.

7.4.4 Changes in the Algorithm

Here is a modified version of the algorithm (without CG) given in Forsberg et al. [1], in pseudo-code notation:

```
let L be the empty lexicon.
let R be the set of extraction rules.
let W be all word types in the corpus.
let S be the list of chunks.
for each w : W
 for each r : R
  for each constraint r(C) with which w matches r(C)
   if W,S satisfies instantiated r(C) with the result H,
    add H to L
   endif
  end
 end
end
```

The algorithm is initialized by reading the word types of the corpus into an array W. A word w **matches** a constraint of a rule $r(C)$, if it can match any of the patterns in the rule's constraint that contains all variables occurring in the constraint. The result of a successful match is an **instantiated constraint**, i.e. a logical formula with words as atomic propositions. The corpus W **satisfies** an instantiated constraint $r(C)$ if the formula is true, where the truth of an atomic proposition a means that the word a occurs in W. And finally, the store S, containing the contextual information, **satisfies** an instantiated constraint $r(C)$ if, for a Constraint Grammar cg_i associated to an instantiated pattern $a \in r(C)$, there exists at least one chunk $ch \in S$, in which a appears, which **satisfies** cg_i.

7.5 The Implementation

An important consideration with the implementation is to be able to deal with as much data as possible. It is space, not speed, which is the critical issue (within reason, of course), since the result typically improves with larger amounts of data.

The representation of strings in Haskell is a bit wasteful, and if one is not careful, one soon runs out of memory. There are other string types with more efficient representations, but these types are not as convenient to program with. We solved this by implementing *sharing*, i.e. a word form is only fully represented once in memory, and all other occurrences are pointers

to the full representation. This solution gives the tool a reasonable space behavior.

7.6 Experiments

An experiment with the French Hansard corpus has been conducted and furthermore described in Extract's FinTAL article [1].

Some initial experiments have been tried using CG and structured data while extracting Swedish words, where the starting point is a morphology implemented in Functional Morphology (FM) [2]. FM supports the analysis of a text into Extract's data format, where the chunks are divided, in a naive fashion, at major punctuations. No disambiguation is performed, i.e. FM augments all word forms with all possible interpretations.

In these experiments we realized that without any form of disambiguation, the information provided by the existing lexicon is of little use, at least for Swedish. It is not necessary to perform full disambiguation: it is sufficient to focus on classes mentioned in the constraints. For example, consider the word *att* in Swedish, which is a function word that is either an infinitive marker or a subjunction (*that*). If we knew that *att* is an infinitive marker, it would be a good indicator for identifying verbs, but with no disambiguation, we are not helped.

It is our experience that it is more difficult than we previously thought to use the context consisting of either unannotated word forms or word forms annotated with ambiguous information for improving the extraction in a substantial way. Our hypothesis is that to be able to use CG in Extract efficiently, we need to augment the word forms with unambiguous information. Ambiguous information is useful for filtering out spurious outputs, but difficult to use in a positive sense, i.e. to locate and extract new words. However, it is too early to draw any conclusions.

7.7 Compiling Extract

The source code is downloadable at Extract's homepage[1].

Extract requires the GHC compiler[2] to be built. Since Extract is a command-line program, it should work on all platforms supported by the GHC compiler.

[1] `http://www.cs.chalmers.se/~markus/Extract`

[2] `http://www.haskell.org/ghc`

1. `tar xvfz Extract_2.0.tgz`
2. `cd Extract_2.0`
3. `make`
4. This produces a binary: `extract`

7.8 Running Extract

Extract is command-line program, which is run with a *rule file*, a *text file*, and possibly, some flags.

```
$ extract [Flag(s)] rule_file  text_file
```

The text file `text_file` may either be structured, in the sense described in Sec. 7.3, or unformatted. Extract is able to automatically detect which of the different types it is.

If a lexical resource in Functional Morphology is available, then it can be used to generate structured data. See the technical report of Functional Morphology for details.

7.9 Command-line Options

All command-line flags available in Extract are printed by calling Extract with the flag `-h`. The output is given in Fig. 7.5. Most of the flags are self-explanatory, but we will still give some clarifying comments for all of them.

7.9.1 Character Encodings

Extract v1.0 supported the character encoding ISO-8859, which restricts the use of the tool to a particular set of language, or forces the use of some ad-hoc encoding. Extract v2.0 acknowledges this problem by introducing UTF-8 encoding, activated with the flag `-utf8`. If the UTF-8 mode is activated, then both the input data file and the rule file must be in that encoding.

7.9.2 Data Preprocessing

There are two commands for data preprocessing: (`-uncap`), which transforms the word forms to lowercase and `-nocap` which removes all capitalized words.

```
$ extract -h

*****************************************
*           Lexicon Extraction          *
*                 with                  *
*           Constraint Grammar          *
*****************************************
* (c) Markus Forsberg & Aarne Ranta 2007 *
* under GNU General Public License.      *
*****************************************

 Help message:
 extract [Option(s)] rule_file  corpus_file
 Options:
  -h       Display this message
  -utf8    Use UTF-8 encoding
  -nobad   Keep only the analysed words
  -uncap   Transform uppercase to lowercase
  -nocap   Remove all words in uppercase
  -e       Print evidence as comment
  -u       Print no duplicates
  -id      Print identifier
  -r       Reverse words
```

Figure 7.5: Help command

7.9.3 Output Control

The word forms that do not give rise to any output are by default printed with dashes (`--`) in front of them. To avoid printing these, use the flag `-nobad`.

Extract traverses all word forms and tries to instantiate the rule's constraint. This, in turn, typically means duplicated output, since every word form gives rise to its own instantiations. This is avoided by giving the uniqueness flag, `-u`, which means that a history of previous results is kept and no duplicates are output. Extract is typically run with the uniqueness flag, or, if preferred, with the Unix command `sort -u`.

A constraint of a rule is fulfilled by a set of word forms, but since a constraint may contain disjunctive patterns, it is not always clear which word forms were used. Since this information may be useful, it is possible to obtain it with the evidence flag `-e`. Every output will then be marked with the word forms used.

Multiple rules may have the same identifier, and it is sometimes useful to know exactly which of the rules were used: if for nothing else, so for rule debugging purposes. The identify flag `-id` annotates the output with an integer corresponding to a rule in the rule file.

7.9.4 Dictionary

One of the data structures of Extract is a trie, which is a deterministic automaton, i.e. it is prefix-minimal but not suffix-minimal. If a language is suffix-heavy, it may save memory to reverse the string, which can be done with the flag `-r`.

7.10 BNFC Documentation of Extract

7.10.1 The Language Extract

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of Extract

Identifiers

Identifiers ⟨*Ident* ⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

String literals ⟨*String* ⟩ have the form `"`x`"`, where x is any sequence of any characters except `"` unless preceded by `\`.

Integer literals ⟨*Int* ⟩ are nonempty sequences of digits.

Character literals ⟨*Char* ⟩ have the form `'`c`'`, where c is any single character.

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 Extract are the following:

```
char      context  digit
eps       letter   lower
paradigm  regexp   rule
upper
```

The symbols used in Extract are the following:

```
;  =  {
}  [  ]
:  ,  _
(  )  +
&  |  ~
!  @  −
*  ?
```

Comments

Single-line comments begin with $--$.
Multiple-line comments are enclosed with $\{-$ and $-\}$.

The syntactic structure of Extract

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.

⟨*Grammar* ⟩ ::= ⟨*ListDef* ⟩

⟨*ListDef* ⟩ ::= ϵ
| ⟨*Def* ⟩ `;` ⟨*ListDef* ⟩

⟨*Def* ⟩ ::= `paradigm` ⟨*Ident* ⟩ ⟨*Env* ⟩ = ⟨*Head* ⟩ { ⟨*Logic* ⟩ }
| `rule` ⟨*Ident* ⟩ ⟨*Env* ⟩ = ⟨*Head* ⟩ { ⟨*Logic* ⟩ }
| `regexp` ⟨*Ident* ⟩ = ⟨*Reg* ⟩
| `context` ⟨*Ident* ⟩ = ⟨*CLogic* ⟩

⟨*Env* ⟩ ::= `[` ⟨*ListBinding* ⟩ `]`
| ϵ

⟨*Binding* ⟩ ::= ⟨*Ident* ⟩ `:` ⟨*Reg* ⟩

⟨*ListBinding* ⟩ ::= ϵ
| ⟨*Binding* ⟩
| ⟨*Binding* ⟩ `,` ⟨*ListBinding* ⟩

⟨*Pattern* ⟩ ::= ⟨*ListItem* ⟩ ⟨*Constraint* ⟩

⟨*Item* ⟩ ::= ⟨*String* ⟩
| ⟨*Ident* ⟩

⟨*Patt1* ⟩ ::= `_`
| ⟨*Ident* ⟩
| `(` ⟨*Patt* ⟩ `)`

⟨*Patt* ⟩ ::= ⟨*Ident* ⟩ ⟨*ListPatt1* ⟩
| ⟨*Patt1* ⟩

⟨*ListPatt1* ⟩ ::= ⟨*Patt1* ⟩
| ⟨*Patt1* ⟩ ⟨*ListPatt1* ⟩

⟨*Head* ⟩ ::= ⟨*ListPattern* ⟩

⟨*ListPattern* ⟩ ::= ⟨*Pattern* ⟩
| ⟨*Pattern* ⟩ ⟨*ListPattern* ⟩

⟨*ListItem* ⟩ ::= ⟨*Item* ⟩
| ⟨*Item* ⟩ + ⟨*ListItem* ⟩

⟨*Constraint* ⟩ ::= `[` ⟨*CLogic* ⟩ `]`
| `[` ⟨*Ident* ⟩ `]`
| ϵ

⟨*CLogic* ⟩ ::= ⟨*CLogic* ⟩ `&` ⟨*CLogic1* ⟩
| ⟨*CLogic* ⟩ | ⟨*CLogic1* ⟩
| ⟨*CLogic1* ⟩

⟨*CLogic1* ⟩ ::= `~` ⟨*CLogic1* ⟩
| `_`
| `(` ⟨*Position* ⟩ `,` ⟨*Reg* ⟩ `,` ⟨*Unique* ⟩ ⟨*Patt* ⟩ `)`
| `(` ⟨*CLogic* ⟩ `)`

⟨*Unique* ⟩ ::= `!`
| ϵ

⟨*Position* ⟩ ::= `_`
| ⟨*Pos* ⟩
| ⟨*Ident* ⟩ `@` ⟨*Pos* ⟩
| ⟨*Ident* ⟩ `@` ⟨*Ident* ⟩ ⟨*Pos* ⟩
| ⟨*Ident* ⟩ ⟨*Pos* ⟩

⟨*Pos* ⟩ ::= ⟨*Integer* ⟩
| + ⟨*Integer* ⟩
| − ⟨*Integer* ⟩
| + ⟨*Integer* ⟩ `*`
| − ⟨*Integer* ⟩ `*`
| `*`

⟨*Logic* ⟩ ::= ⟨*Logic* ⟩ `&` ⟨*Logic1* ⟩
| ⟨*Logic* ⟩ | ⟨*Logic1* ⟩
| ⟨*Logic1* ⟩

⟨*Logic1* ⟩ ::= `~` ⟨*Logic1* ⟩
| `_`
| ⟨*Pattern* ⟩
| `(` ⟨*Logic* ⟩ `)`

⟨*Reg* ⟩ ::= ⟨*Reg* ⟩ | ⟨*Reg1* ⟩
| ⟨*Reg1* ⟩ − ⟨*Reg1* ⟩
| ⟨*Reg1* ⟩
⟨*Reg1* ⟩ ::= ⟨*Reg1* ⟩ ⟨*Reg2* ⟩
| ⟨*Reg2* ⟩
⟨*Reg2* ⟩ ::= ⟨*Reg2* ⟩ `*`
| ⟨*Reg2* ⟩ +
| ⟨*Reg2* ⟩ `?`
| `eps`
| ⟨*Char* ⟩
| `[` ⟨*String* ⟩ `]`
| `digit`
| `letter`
| `upper`
| `lower`
| `char`
| `_`
| ⟨*String* ⟩
| ⟨*Ident* ⟩
| `(` ⟨*Reg* ⟩ `)`

7.10.2 The Language Data

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of Data

Identifiers

Identifiers ⟨*Ident* ⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

String literals ⟨*String* ⟩ have the form `"`x`"`, where x is any sequence of any characters except `"` unless preceded by `\`.

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 Data are the following:

There are no reserved words in Data.

The symbols used in Data are the following:

```
{  }  (
,  )  |
```

Comments

There are no single-line comments in the grammar.
There are no multiple-line comments in the grammar.

The syntactic structure of Data

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Input* ⟩ ::= ⟨*ListData* ⟩

⟨*Data* ⟩ ::= { ⟨*ListTokD* ⟩ }

⟨*TokD* ⟩ ::= *(* ⟨*String* ⟩ **,** ⟨*ListPattern* ⟩ *)*

⟨*Pattern* ⟩ ::= ⟨*Ident* ⟩ ⟨*ListPattern1* ⟩
| ⟨*Pattern1* ⟩

⟨*Pattern1* ⟩ ::= ⟨*Ident* ⟩
| (⟨*Pattern* ⟩)

⟨*ListData* ⟩ ::= ϵ
| ⟨*Data* ⟩ ⟨*ListData* ⟩

$$
\begin{array}{lcl}
\langle \mathit{ListTokD} \rangle & ::= & \epsilon \\
& | & \langle \mathit{TokD} \rangle \ \langle \mathit{ListTokD} \rangle \\
\langle \mathit{ListPattern1} \rangle & ::= & \langle \mathit{Pattern1} \rangle \\
& | & \langle \mathit{Pattern1} \rangle \ \langle \mathit{ListPattern1} \rangle \\
\langle \mathit{ListPattern} \rangle & ::= & \epsilon \\
& | & \langle \mathit{Pattern} \rangle \\
& | & \langle \mathit{Pattern} \rangle \mid \langle \mathit{ListPattern} \rangle
\end{array}
$$

Bibliography

[1] M. Forsberg, H. Hammarström, and A. Ranta. Morphological lexicon extraction from raw text data. *FinTAL 2006, LNAI 4139*, pages 488–499, 2006.

[2] M. Forsberg and A. Ranta. Functional morphology. `http://www.cs.chalmers.se/~markus/FM`, 2007.

[3] F. Karlsson. Constraint grammar as a framework for parsing running text. *13th International Conference of Computational Linguistics*, 3:168–173, 1990.

[4] A. Voutilainen, J. Heikkilä, and A. Anttila. Constraint grammar of english, a performance-oriented introduction. *University of Helsinki, publication no. 21*, 1992.