Thesis for the Degree of Licentiate of Engineering

Applications of Functional Programming in Processing Formal and Natural Languages

Markus Forsberg

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

Göteborg, December 2004

Applications of Functional Programming
in Processing Natural and Formal Languages
Markus Forsberg

© Markus Forsberg, 2004

Technical Report no. 40L
ISSN 1651-4963
School of Computer Science and Engineering

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

Printed at Chalmers, Göteborg, Sweden, 2004

Abstrakt (Swedish)

Avhandlingen beskriver hur funktionell programmering kan användas för att behandla formella och naturliga språk. Teknikerna som beskrivs är nära kopplade till kompilatorkonstruktion, vilket är tydligast i arbetet om BNF Converter.

Den första delen av avhandlingen beskriver BNFC (BNF Converter), ett flerspråkligt kompilatorverktyg. BNFC tar som indata en grammatik skriven i 'Labelled BNF'-notation (LBNF), och genererar komponenter för de första faserna i en kompilator (en abstrakt syntax, en lexer och en parser). Vidare genereras fallanalys av och utskrifthanterare för den abstrakta syntaxen, ett testprogram och dokumentation för det beskrivna språket. Programkomponenterna kan genereras i Haskell, Java, C och C++ och deras parser- och lexerverktyg. BNFC är skrivet i Haskell.

Den metodik som används för att generera komponenterna är baserad på Appels böcker om kompilatorkonstruktion. BNFC har använts som ett undervisningsvertyg i kompilatorkonstruktionskurser på Chalmers. Den har även använts i forskningsrelaterad programspråksutveckling och i en industriell applikation där den har använts för att producera en kompilator för ett protokollbeskrivningsspråk för telekommunikation.

Den andra delen av avhandlingen beskriver Funktionell Morfologi, ett verktyg för att implementera morfologier för naturliga språk i det funktionella språket Haskell. Den grundläggande metoden är enkel: istället för att använda otypade, reguljära uttryck, som är det främsta verktyget för morfologibeskrivning idag, så använder vi oss av ändliga funktioner över ändliga algebraiska datatyper. Definitionen av dessa datatyper och funktioner är den språkberoende delen av morfologin. Den språkoberoende delen består av ett otypat lexikonformat som används till översättning till andra morfologiformat, syntes av ordformer och till att generera en datastruktur för analys.

Funktionell Morfologi bygger på Huets språkteknologiska verktyg Zen, som han har använt för att implementera en morfologi för Sanskrit. Målet har varit att göra det enkelt för lingvister som inte är erfarna funktionella programmarerare att applicera ideerna på nya språk. Ett bevis på metodens produktivitet är att morfologier för Svenska, Italienska, Ryska, Spanska och Latin har implementerats.

Abstract (English)

This thesis describes two applications of functional programming to process formal and natural languages. The techniques described in this thesis are closely connected to compiler construction, which is obvious in the work on BNF Converter.

The first part of the thesis describes the BNFC (the BNF Converter) application, a multi-lingual compiler tool. BNFC takes 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 and C++, and their standard parser and lexer tools. BNFC itself was written in Haskell.

The methodology used for the generated front-end is based on Appel's books on compiler construction. BNFC has been used as a teaching tool in compiler construction courses at Chalmers. It has also been applied to research-related programming language development, and in an industrial application producing a compiler for a telecommunications protocol description language.

The second part of the thesis describes Functional Morphology, a toolkit 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 translation to other morphology formats and synthesis of word forms, and to generate 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.

The four papers included in this thesis have been published previously as follows:

- **Paper I**: *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
- **Paper II (Technical Report)**: *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
- **Paper III**: *Tool Demonstration: BNF Converter*, Markus Forsberg & Aarne Ranta, Proceedings of the ACM SIGPLAN 2004 Haskell Workshop, Snowbird, Utah, USA, pages 94–95
- **Paper IV**: *Functional Morphology*, Markus Forsberg & Aarne Ranta, Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, September 19-21, 2004, Snowbird, Utah, USA, pages 213–223

Acknowledgements

First, I would like to direct my sincerest thanks to my supervisor, Aarne Ranta, who helped me immensely in getting to this point in my graduate studies.

Secondly, I would like to thank all the members of the Language Technology group and my PhD committee. Thanks for all inspiring discussions and for your help with improving the quality of my work. In particular, I want to thank Kristofer Johannisson, Graham Kemp and Bengt Nordström for reading this thesis and for giving suggestions for improvement. And thanks all for being such a nice group of people.

Thirdly, thanks all co-workers at the Computing Science department at Chalmers — you provide a great working environment.

I would also like thank my family and friends for making my life outside the office meaningful and happy.

Finally, but not least, I would like to give special thanks to my girlfriend Merja, who makes my world a better place.

Contents

Chapter 1

Introduction

The work described in this thesis is in the area of Language Technology. A possible definition of Language Technology, abbreviated LT, is *the research field that studies computer-aided processing of languages*. LT can be further divided into (Natural) Language Processing and Speech Technology, where Language Processing studies symbolic languages (written text, transcribed speech etc), and Speech Technology studies speech in the shape of an audio signal. The difference in these two subfields is in reality not as clearly distinct as the above definition implies; there is a great deal of overlap. The term "Language Technology" will be used in this thesis without further distinction. A more in-depth introduction to the Language Technology field can be found in Jurafsky's and Martin's book "Speech and Language Processing" [9].

Language technologists work in the area in between of Computing Science and Linguistics, which gives rise to an interesting cross-fertilization of ideas. This thesis provides a demonstration of this. The work on the BNF Converter, a tool for defining formal languages, can be viewed as ideas brought from Linguistics into Computing Science. Many compiler tools developed within Computing Science aim at strong expressiveness, often at the expense of declarativity. In Linguistics, on the other hand, there exists a tradition to hold declarativity before expressiveness. These ideas have been brought into the BNF Converter tool, where some requirements on the target language enable a declarative source format. The work on Functional Morphology, a toolkit for defining natural language morphologies, has influences from the other direction — ideas from the functional programming community are brought into Linguistics.

The LT group at Chalmers is working at the interface of natural and formal languages. The group has a strong connection to the area of Compiler Construction, which differs from many other LT groups that have a more traditional affinity to Artificial Intelligence. The connection to the Compiler Construction field is obvious in this thesis, in particular in the work on BNF

Converter.

The main tool of Chalmer's LT group is Grammatical Framework (GF) [17, 15], a multi-lingual grammar formalism based on type theory [11, 10]. Due to this formal approach, instead of a more statistical approach, the initial idea was that only fragments of natural languages could be machine translated, and the aim was exactness rather than full coverage.

This idea is still not completely incorrect, but the view has slightly changed due to Functional Morphology, discussed in this thesis, and the GF resource grammars [16], a collection of syntactic libraries, and the goal is now more ambitious. In particular, Functional Morphology has shown to be very productive where the coverage of a major part of a language's inflectional morphology is tractable. However, it is important to stress that both morphologies in Functional Morphology and the resource grammars are syntactic resources, not semantic resources, so complete machine translation is still not considered feasible.

1.1 Formal and Natural Languages

The main theme of this thesis is computer-aided processing of languages powered by functional programming. The languages considered are both *formal* and *natural*. Natural languages are what most people think of when they hear the word "language", i.e. languages such as English, Spanish, French, Chinese and Sanskrit.

Formal languages are a much simpler class of languages — artificial languages created for a specific purpose, such as programming languages or mathematical languages.

The word *processing* is a vague term, so more precision may be in order. Languages are represented in a computer as sequences of symbols or, more technically, as *strings*, illustrated in figure 1.1. This string is *analyzed* into some internal representation that reflects the information extracted from the string. The reverse process may also be interesting — to *synthesize* the internal representation into a string.

A more concrete example is *machine translation* between two languages, where the source language, the language translated from, would be analyzed to produce the internal representation. Depending on the level of sophistication of the machine translation, the internal representation may undergo a sequence of transformations. Finally, the internal representation is synthesized into the target language.

1.2 LT and Functional Programming

High-level programming languages, i.e. *declarative* languages, can be divided into *functional languages* such as Lisp, ML and Haskell, and *logic languages*

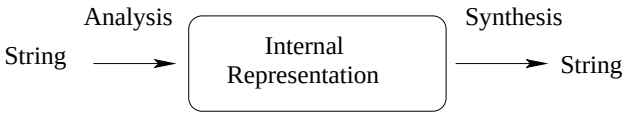

Figure 1.1: Language Processing Overview

such as Prolog. Declarative languages have been used by language technologists since they came to existence.

The reasons for using declarative languages are many — quick prototyping; closeness to mathematical description and hence to linguistic models influenced by mathematics; simple construction and manipulation of structured data.

Logic languages, in particular the programming language Prolog, have been the prominent tool within LT. Some of the reasons for their popularity are the fact that logic languages are founded on predicate logic, a popular formalism to describe the semantics of natural languages, and because many implementations have the built-in grammar description language DCG (Definite Clause Grammar) [13].

The foundation of functional programming is the Lambda Calculus [1], a minimal, Turing-complete [1] language based solely on functions. The focus of this thesis is on typed functional languages, and the programming language under consideration is Haskell [14]. One of the main themes in this thesis is compiler construction and it is well known in the functional programming community that one of the strong applications of functional programming is compiler construction. Some of the features of Haskell that are essential in this work are described in the following sections. A more in-depth motivation of functional programming is given in Hughes' article *Why Functional Programming Matters* [8].

1.2.1 A strong, polymorphic type system

There is a tradition in LT of using untyped or weakly typed languages. Functional Morphology and Grammatical Framework can be considered as evidence that the use of types in a LT application can increase productivity and reduce the amount of errors. The types in Functional Morphology ensure, for example, that the inflectional parameter type of a noun is not by accident constructed with parameters from another word class and that all cases of a noun's inflection table are actually defined.

An example of the use of types, here with *algebraic data types*, is the description of the inflectional parameters of Swedish nouns. There are three parameters: number, case and species. Three new types are introduced — `Number`, `Case` and `Species` — and the elements of the types are enumerated.

[1] All computable functions can be expressed and calculated in Lambda Calculus

The type `NounForm` is just the composition of the three parameters to form one inflection type.

```
data Number   = Singular   | Plural
data Case     = Nominative | Genitive
data Species  = Indefinite | Definite
data NounForm = NF Number Case Species
```

As a next step, the inflection table of nouns in the first declension is defined as a function `decl1`. The function takes as the first argument a lemma, as the second argument the parameters type, and as the result it gives the inflected form. The function is essentially case analysis over the parameters, and the result is constructed with the concatenation operator `(++)` that concatenates two strings. The function `mkCase` describes the case inflection.

```
decl1 :: String -> NounForm -> String
decl1 apa (NF number case' species) =
      mkCase case' $
       case number of
        Singular -> case species of
                     Definite    -> apa ++ "n"
                     Indefinite  -> apa
        Plural   -> case species of
                     Definite    -> (tk 1 apa) ++ "orna"
                     Indefinite  -> (tk 1 apa) ++ "or"

mkCase :: Case -> String -> String
mkCase case' word = case case' of
                      Nominative -> word
                      Genitive   ->
                          if (last word == 's') then word
                            else word ++ "s"
```

The inflection table of the words *flicka* and *apa* can be defined with the following definitions.

```
flicka = decl1 "flicka"
apa    = decl1 "apa"
```

`flicka` and `apa` are now functions that, given a parameters type, return a string, e.g. `flicka (NF Plural Genitive Definite) = "flickornas"`.

1.2.2 Higher-order functions

Higher-order functions can take functions as arguments and return a function as a result of a computation, that is, functions are *first-class values*.

A classic example of a higher-order function is the `map` function, that takes any function from `a` to `b`, and a list of `a`:s (`[a]`), and applies that function to all elements in the list of `a`:s, to receive a list of `b`:s (`[b]`). The variables `a` and `b` are *type variables* — `a` and `b` can be replaced by any type — and because of this, `map` is said to be a *polymorphic function*.

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

The ability to program on more than one abstraction layer gives perfect support to linguistic abstractions. An example of this is the exception handling in Functional Morphology, described in chapter 5, where higher-order functions are used to handle exceptional linguistic phenomena.

The exception handling of Functional Morphology can be demonstrated through the Latin word *vis* (Eng. violence, force), that inflects in the same way as the word *hostis* (Eng. enemy), with the exception of the vocative, genitive, dative case in singular, which are missing. This would be described with the following function, where the higher-order function is `missing`, written in infix notation:

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

1.2.3 Class system

The class system of Haskell provides overloading. Overloading means that the definition of a function symbol is defined by its type. This is static typing — everything is decided at compile-time, not runtime. The class system provides a good abstraction mechanism, and is the driving machinery of Functional Morphology.

An example from Functional Morphology is the `Param` class, that contains the overloaded constant `values`, which every instance of that class must define. The `values` constant is an enumeration of all objects in a particular type `a`.

```
class Param a where
  values :: [a]
```

A fundamental function in Functional Morphology is `table`, which transforms a function into an (inflection) table. The function can be read as: if a function is from `a` to `b`, where `a` is an instance of the class `Param`, then a list of pairs that constitute the table can be generated by picking out objects from `values`, one by one, and applying the input function to every object.

```
table :: Param a => (a -> b) -> [(a,b)]
table f = [(a, f a) | a <- values]
```

If the `NounForm` type is made an instance of the `Param` class, then an inflection table for nouns can be generated with the function `nounTable` below, where the function `table` is used.

```
instance NounForm where
  values =  -- details omitted

nounTable :: (NounForm -> String) -> [(NounForm, String)]
nounTable noun = table noun
```

`nounTable` only specializes the types of the more general function `table`. Every occurence of `nounTable` can be replaced by `table`.

1.3 BNF Converter

The three papers in chapter 2, 3 and 4 describe 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.

BNF Converter, abbreviated BNFC, started as an experimental study into what extent the GF tool could be used as a compiler front-end generator. Though it was possible, it was soon realized that some extra notation 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 performed 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 that 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++, Java 1.4 and Java 1.5.

1.3.1 Technical 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 a LBNF grammar are: an abstract syntax, a lexer, a parser, a pretty-printer, a case-skeleton that traverses the abstract syntax, a test bench that puts everything together in an executable and language documentation.

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 parser 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.

Requirements of BNF Converter

Some requirements are put on the languages implemented in BNFC, to be able to generate all the mentioned modules. All requirements follow the guidelines of any modern compiler construction handbook, 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 because the engine in a lexer is a finite state automaton, which is equivalent with the input regular expression. It is, however, possible to execute arbitrary code in the semantic action of a lexer rule, so that non-regular phenomena such as nested comments can be handled by the lexer.

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, possibly, as layout information. That is, white spaces can safely be removed in the tokenization process.

The grammar must, in most cases, be LALR(1) parsable. This requirement is actually not something inherent in BNFC, but rather in the tools that are used to produce the parser. For example, the Happy parser gener-

ator has been generalized with the Tomita algorithm [18, 12] so that it will produce a forest of parse trees, instead of a single tree.

1.4 Functional Morphology

A toolkit for defining natural language morphologies, called Functional Morphology (FM), is described in chapter 5. The goal of Functional Morphology is to supply a convenient way of defining full-scale morphologies. The approach is based on Hockett's word-and-paradigm model [5], which essentially is the idea that a morphology is defined as a list of dictionary forms, or *lemmas*, and each lemma has a pointer to its corresponding inflection table.

A morphology implementation is a digital representation of a particular linguistic resource. Bird and Simon [3] lists a number of problems with digital representation of linguistic resources. In particular, they point out that:

> Funded documentation projects are usually tied to software versions, file formats, and system configurations having a lifespan of three to five years.

FM provides a couple of solutions to this problem. The source code is open source, so the internal behavior of the system will, at least in theory, be accessible long after a compiler for the language is no longer maintained. But a more important point is that FM can generate a set of formats, some which aim for proprietary systems such as XFST [2], and some of which are of a more general format, such as full-form lexicon. These properties of FM ensure that the lifetime of a resource is extended significantly.

One of the main goals of FM was to develop a methodology which is powerful enough to enable the description of a multiple of languages' morphologies, but simple enough to enable a unified back-end to the methodology, so that morphology implementations can share translators, analyzer and synthesizer.

The standard practice today when implementing a morphology is to use finite state technology. The task of convincing linguists to use FM instead of, for example, XFST involved two considerations. First, to minimize the learning curve in the transition to FM, the aim was that minimal knowledge of functional programming would be required to use FM. Secondly, source code for finite state tools had to be generated, so that no work would be lost if a linguist decided to switch back to finite state technology.

Yet another goal was to embed the methodology in a full programming language, instead of developing a new language, i.e. to create a domain-specific embedded language [6, 7] in Haskell. This gives a lot for free, such as all normal programming constructs, and that the maintenance of the language is mainly done by Haskell's compiler development team.

There are also some drawbacks of using an embedded domain-specific language, such the inability to perform task-specific optimizations.

The power of a full programming language enables computations over complex data structures. This is important when a language with a complicated morphology is implemented, such as Sanskrit or Arabic.

Languages can be said to be typed — they are usually analyzed into exact categorizations, such as part of speech and grammatical features — and in FM this is directly reflected in the description.

1.4.1 Technical overview of FM

As mentioned previously, FM describes a morphology with the lemmas of the target language augmented with a pointer to its corresponding inflection table, or with the technical term, its *paradigm*.

The paradigms in FM are described with finite functions, which is later translated into tables. The lexicon consists of a listing of the lemmas applied to their corresponding paradigm function. The pointer is the function symbol, and the inflection table is the evaluation of the application.

A morphology implementer using FM has to provide three language-specific components:

- a **type system** that defines all word classes and the parameters belonging to them;
- an **inflection machinery** that defines all paradigms for all word classes;
- a **lexicon** that lists all words in the target language with their paradigms.

The translation of a morphology into the language-independent part of FM is done through type classes and is described in more detail in chapter 5.

1.5 Current States of the Work

BNF Converter is now coming of age and is being widely used. It is has been used as a teaching tool and for development of full-scale languages. It is also now part of the *testing* distribution of Debian Linux. The restrictions posed on the languages are, by many, considered to be small compared with the gains that the tool provides — shorter development time, simplified maintenance and reduced code size.

Functional morphology has been developed under a period of several years, and has now reached a stable situation. The goal of creating a tool accessible to linguists with no previous experience of functional programming has been demonstrated by the master students that successfully used

FM and developed two substantial morphologies with no or limited experience of Haskell.

When a morphology has already been defined, and the task is only lexicographic, i.e. to extend the lexicon, FM can be used without any knowledge of functional programming. The only requirement is that the user knows the target language well enough. FM has been used in a first-year course, where the students were supposed to extend the lexicon and, without any knowledge of Haskell, they successfully extended the lexicon with hundreds of lemmas.

1.6 Future Work

BNF Converter and FM have evolved into mature systems and will be subject of continued development and maintenance. However, in the next stages of the project the focus will on other areas.

The particular way of representing the morphology in FM, lemmas paired with their paradigms, opens up doors for *lexicon extraction*; if a lemma and its paradigm are successfully identified in a corpus, the FM machinery can generate the full inflection table, with word forms that possibly do not exist in the corpus.

An simple example is the first declension paradigm in Swedish.

```
decl1:  ap/a (apor|apors|aporna|apornas) 3
```

To identify a lemma *blurga* in a corpus as a noun in the first declension, we require that at least one of the forms *blurgor*, *blurgors*, *blurgorna* and *blurgornas* are present. The number at the end of the paradigm description imposes a minimal length requirement on the lemmas — this to avoid spurious findings.

Lexicon extraction is a difficult problem, e.g. paradigms can be overlapping or undistinguishable, and knowledge about syntax or even semantics may be necessary to identify a particular paradigm. However, our approach still saves a lot of work: one who knows Latin can quite quickly decide if a verb is in, for example, the first conjugation, by seeing the first few word forms. This could be contrasted with word form based extraction, where the, probably incomplete, inflection table of a lemma would be expanded and mixed with other word forms. The verification of a word form based extraction is a much more time-consuming task.

An experimental tool for extraction has been developed, is available at the FM homepage [4], and some successful experiments have been performed. In the near future we will analyze and document this approach to lexicon extraction. One interesting direction is to use existing morphology implementations to extract automatically a lemma-based bilingual dictionary from a text that exists in both languages. The idea is simple: to align words

according to their part of speech and common grammatical features, and to pair their corresponding lemmas.

1.7 Contributions

The research contributions of the author of the thesis are the following:

- development of the major part of the Haskell generation, the first target language of BNF Converter. The Java, C and C++ generation was developed by Michael Pellauer and Java 1.5 by Björn Bringert;
- the further development and implementation in Haskell of the morphological analysis algorithm described by Huet;
- development of the Functional Morphology library to achieve a language independent framework;
- development of the Latin morphology and parts of the Swedish morphology in Functional Morphology;
- the main author of the papers in chapters 2 and 5, where Aarne Ranta co-authored;
- A co-author of the papers in chapters 3, where Michael Pellauer was the main author, and 4, where Aarne Ranta was the main author.

Bibliography

[1] H. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, 1981.

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

[3] S. Bird and G. Simons. Seven dimensions of portability for language documentation and description. *Language*, 79:557–582, 2003.

[4] M. Forsberg and A. Ranta. Functional morphology. `http://www.cs.chalmers.se/~markus/FM`, 2004.

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

[6] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), 1996.

[7] 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.

[8] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[9] D. Jurafsky and J. H. Martin. *Speech and Language Processing, An introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, Upper Saddle River, New Jersey 07458, 2000.

[10] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[11] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Lf's Type Theory. An Introduction*. Oxford University Press, 1990.

[12] Paul Callaghan and students. Project Description, 2004. `http://www.dur.ac.uk/computer.science/ug/mods/y3proj/proj01/pcc2.htm`.

[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 from `http://www.haskell.org`, February 1999.

[15] A. Ranta. Grammatical Framework Homepage, 2000–2004. `www.cs.chalmers.se/~aarne/GF/`.

[16] A. Ranta. GF Resource grammar site. Program and documentation, `http://www.cs.chalmers.se/~aarne/GF/lib/resource`, 2004.

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

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

Chapter 2

Paper I: 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 Gothenburg
SE-412 96 Gothenburg, Sweden
{markus,aarne}@cs.chalmers.se

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 [13]. 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 a 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*[5]. 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[15], and its target languages are presently Haskell, the associated compiler tools Happy[11] and Alex[3], and LaTeX. Happy is a parser generator tool, similar to YACC[8], which from a BNF-like description builds an LALR(1) parser. Alex is a lexer generator tool, similar to Lex[10], 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 con-

struction, 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 [7], and JLex [4][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.

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 datatype 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 [2]). 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):

[1] Work by Michael Pellauer at Chalmers

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 datatype 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 ;
```

2.4 BNF Converter code generation

2.4.1 The files

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

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

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

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

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

```
_.  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 datatypes, 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 conctructor. 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[5] 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 [9] 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

[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.

that is expressible in the language. When designing new languages—and 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 [8] 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 [12], which uses an EBNF-like notation to generate a Happy parser, an Alex lexer, and a datatype 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 [6]. 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 [2] 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) [14]. 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 [16]. 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 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] A. W. A. J. L. K. Daniel C. Wang and C. S. Serra. The zephyr abstract syntax description language. *Proceedings of the Conference on Domain-Specific Languages*, 1997.

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

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

[5] M. Forsberg and A. Ranta. Labelled BNF: a highlevel 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.

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

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

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

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

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

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

[12] N. Martinsson. Cactus (Concrete- to Abstract-syntax Conversion Tool with Userfriendly Syntax) . Master's Thesis in Computer Science, 2001. `http://www.mdstud.chalmers.se/~md`nm/cactus/6.

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

[14] 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.

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

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

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

Chapter 3

Paper II: BNF Converter

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 Gothenburg, Sweden
pellauer, markus, aarne@cs.chalmers.se

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.

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

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.

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.

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.

```
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

The Lexer and Parser.

The BNF Converter generates lexer and parser specifications for the Alex [6] and Happy [16] 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. 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

[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.

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

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 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 subclass 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. 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.

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

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.

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

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 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. [4]

Another real-world example is the object-oriented specification language OCL [20].[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, [19]).

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.

[4] BSc thesis of Ulf Persson at Chalmers.

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

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 [17], 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 [5] 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) [18]. 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.

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

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

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

[17] N. Martinsson. Cactus (Concrete- to Abstract-syntax Conversion Tool with Userfriendly Syntax) . Master's Thesis in Computer Science, 2001. `http://www.mdstud.chalmers.se/~md`nm/cactus/6.

[18] 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.

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

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

Chapter 4

Paper III: Demonstration Abstract: BNF Converter

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

Abstract

We will demonstrate BNFC (the BNF Converter) [8, 7], a multi-lingual compiler tool. BNFC takes as its input a grammar written in LBNF (Labelled BNF) 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 language specification.

The program components can be generated in Haskell, Java, C and C++ and their standard parser and lexer tools. BNFC itself was written in Haskell.

The methodology used for the generated front-end is based on Appel's books on compiler construction [3, 1, 2]. BNFC has been used as a teaching tool in compiler construction courses at Chalmers. It has also been applied to research-related programming language development, and in an industrial application producing a compiler for a telecommunications protocol description language [4].

BNFC is freely available under the GPL license at its website and in the testing distribution of Debian Linux.

4.1 Demo overview

The demo will consists of a brief explanation of the LBNF source format, followed by instructions on how to compile LBNF source format into a front-end in Haskell, Java, C, and C++. The rest of the demonstration will consist of explaining the generated code for Haskell.

4.2 Goals and limits

The central goals of BNFC are

- to minimize the effort needed for compiler front-end construction
- to encourage clean and simple language design
- to make front-end definitions independent of implementation language and thus portable

The LBNF grammar formalism can be learnt in a few minutes by anyone who knows ordinary BNF. The main addition is that each grammar rule has a label, which is used as a constructor of a syntax tree. No semantic actions other than tree construction are allowed. Therefore the formalism is declarative and portable, and a pretty-printer can be derived from the same grammar as the parser. In addition to syntactic rules, LBNF provides a regular expression notation for defining lexical structure, and some pragmatic declarations defining features such as comments.

Since semantic actions are banned, BNFC can only describe languages that are context-free. The lexer must be finite-state and neatly separated from the parser. Even though these requirements are widely propagated in compiler text books, many real-world languages have features that do not quite conform to them. However, practice has shown that such problems can often be overcome by preprocessing. For example, layout syntax can be handled in BNFC by adding a processing level between the lexer and the parser.

4.3 An example grammar

We will now give a short example to give a taste of what the language implementer has to supply, and what BNFC generates. The example grammar is a subset of the Prolog, known as *pure Prolog*.

```
Db       . Database ::= [Clause] ;
Fact     . Clause ::= Predicate ;
Rule     . Clause ::= Predicate ":-" [Predicate] ;
APred    . Predicate ::= Atom ;
CPred    . Predicate ::= Atom "(" [Term] ")" ;
TAtom    . Term ::= Atom ;
VarT     . Term ::= Var ;
Complex  . Term ::= Atom "(" [Term] ")" ;

terminator Clause "." ;
separator nonempty Predicate "," ;
separator nonempty Term "," ;

token Var  ((upper | '_') (letter | digit | '_')*) ;
token Atom (lower (letter | digit | '_')*) ;

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

The grammar shows a couple of things that go beyond the basic idea of labelled BNF rules and regular expressions: a special syntax `[C]` for polymorphic lists, to avoid cluttering the AST:s with monomorphic lists,

as well as shorthands for defining the concrete syntax of a list in terms of its terminator or separator. In addition, LBNF has a notion of precedence levels expressed by integer indices attached to nonterminals. A complete reference of the LBNF language can be found on the BNFC website [7].

4.4 Compiling a grammar

Assuming that the grammar of the previous section is contained in a file named `Prolog.cf`, a Haskell front-end is compiled by issuing the following command:

```
bnfc -m -haskell Prolog.cf
```

This command generates the following file:

- `AbsProlog.hs`: Algebraic datatypes for the AST:s
- `LexProlog.x`: Alex [6] lexer (v1.1 and v2.0)
- `ParProlog.y`: Happy [9] parser
- `PrintProlog.hs`: pretty printer
- `SkelProlog.hs`: AST traversal skeleton
- `TestProlog.hs`: test bench (a program that parses a file and displays the AST and the pretty-printed program)
- `Makefile`: an easy way to compile the test bench
- `DocProlog.tex`: language documentation in LaTeX

For C and C++, similar files are generated but with slightly different names. For Java, many more files are generated, because the abstract syntax definition consists of separate classes for each nonterminal and constructor, following the methodology of Appel [2].

Depending on target language, the generated code is 10–100 times the size of the LBNF source. Yet it isn't hopelessly ugly or low-level, but looks rather similar to hand-written code that follows the chosen compiler writing discipline.

4.5 Related work

Cactus [10], uses an EBNF-like notation to generate front ends in Haskell and C. Cactus is more powerful than BNFC, which makes its notation more complex. Cactus does not generate pretty printers and language documents.

The Zephyr language [5] is portable format for abstract syntax translatable into SML, Haskell, C, C++, Java, and SGML, together with functions for displaying syntax trees. Zephyr does not support the definition of concrete syntax.

4.6 When to use BNFC

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 on an existing language.

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. 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.

4.7 Bio section

Markus Forsberg is an PhD student at the Swedish Graduate School of Language Technology (GSLT) positioned at the department of Computing Science at Chalmers University of Technology and the University of Gothenburg. Aarne Ranta is an associative professor at the same department.

Forsberg and Ranta started the development of the BNF Converter in 2002, as a tool generating Haskell. It was retargeted to C, C++, an Java in 2003 by Michael Pellauer (at Chalmers). Later contributors are Björn Bringert, Peter Gammie and Antti-Juhani Kaijanaho.

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

[5] A. W. A. J. L. K. Daniel C. Wang and C. S. Serra. The zephyr abstract syntax description language. *Proceedings of the Conference on Domain-Specific Languages*, 1997.

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

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

[8] M. Forsberg and A. Ranta. Labelled BNF: a highlevel 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.

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

[10] N. Martinsson. Cactus (Concrete- to Abstract-syntax Conversion Tool with Userfriendly Syntax) . Master's Thesis in Computer Science, 2001. `http://www.mdstud.chalmers.se/~md`nm/cactus/6.

Chapter 5

Paper IV: Functional Morphology

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

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 datatypes. The definitions of these datatypes 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.

5.1 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 5.2, and then the contemporary approaches are discussed in section 5.3. The main part of the paper, section 5.4, focuses on describing the *Functional Morphology* library. We conclude in sections 5.5 and 5.6 with some results and a discussion.

5.2 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 agree-

ment 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.

5.3 Implementations of Morphology

5.3.1 Finite State Technology

The main contemporary approach within computational morphology is finite state technology - the morphology is described with a regular expression [17, 26, 18, 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.

5.3.2 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.

5.3.3 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.

5.4 Functional morphology

5.4.1 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, 16, 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 represent 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.

5.4.2 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.

5.4.3 System overview

A Functional Morphology system consists of two parts, one language dependent part, and one language independent part, illustrated in figure 5.1.

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.

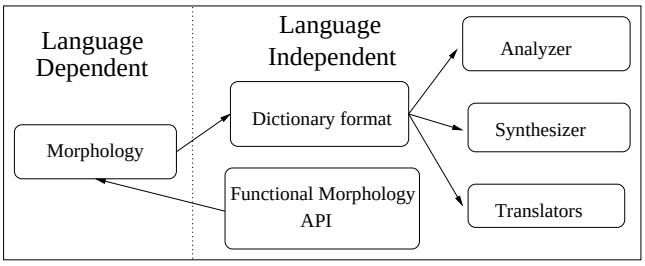

Figure 5.1: Functional Morphology system overview

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.

5.4.4 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 5.4.4 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:

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

```
data VerbForm =
    Indicative Person Number Tense Voice   |
    Infinitive TenseI Voice                |
    ParticiplesFuture  Voice               |
    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 5.4.4, 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 case, 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.

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

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 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 5.4.4 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.

```
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 5.4.4, 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 5.4.6.

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 loose 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 5.4.4, 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]

5.4.5 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).

5.4.6 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

[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.

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 -]
```

5.5 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.

5.6 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`, 2004.

[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://pauillac.inria.fr/~huet/SKT/`, 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] G. Klyve. *Latin Grammar*. Hodder & Stoughton Ltd., London, 2002.

[17] K. Koskenniemi. *Two-level morphology: a general computational model for word-form recognition and production*. PhD thesis, University of Helsinki, 1983.

[18] G. G. L. Karttunen, J-P Chanond and A. Schille. Regular expressions for language engineering. *Natural Language Engineering*, 2:305–328, 1996.

[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 from 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/