# A compiler for phonological rules

**Hans Jansson**
Department of Computer Science, Lund University
Box 118
221 00 Lund,
Sweden,
hans.jansson.667@student.lu.se

## Abstract

This is a report of an implementation of a compiler for phonological rules. The implementation processes files written in lexc (Karttunen, 1993) annotation and produces data suitable for processing with SWI-Prolog (Wielemaker, 2005). The output is a transducer (a Mealy machine, to be more precise), a finite-state machine which not only accepts but also translates its input. Such machines can be used to perform spell checking, morphological analysis etc. The project resembles the lexicon compiler from Xerox[1].

## 1 Introduction

Implementation of a compiler for phonological rules is nothing new. Xerox has already a set of adequate tools for working with such rules, these are, however, not open source and the free of charge copy is limited and for non-commercial use only. Yet, they have gained popularity and thusly, their formalisms have too. Other formalisms and tools for phonological facts and rules (e.g., XML-annotated corpora) exists alongside those from Xerox, even some GPL-licensed ones such as SFST (Schmid, 2001), but many suffer from a common mistake: everyone likes standards, and thus a new such is borned. This project aims to be a prototype for a generic tool, able to generate descriptions of transducers provided phonological rules and easy to attach new front- and backends to.

## 2 Design

To achieve a modular, easily extendable compiler, I chose to build front- and backends around a abstract model of a transducer consisting of start node marker, nodes, labeled arcs and end node markers. Unfortunately, I have already found a design flaw in the representation of the nodes; they are objects, and as such exist uniquely in memory and are therefore already easily distinguishable, but I added a unique non-negative integer to each of them, usable only in the final output. Such details are backend-specific and should of course be implemented in the appropriate backends. The needed correction is however quite small and has no impact on the design as a whole.

## 3 Tools

The JastAdd (Ekman et al., 2005) package was used in the implementation, simply because I was already familiar with it. It was used on top of JavaCC (Viswanadha, 2006), a Java compiler compiler, and it added advanced features of which I only used the parts providing aspect orientation and abstract grammar. By using aspect oriented code, no Visitor pattern was needed to traverse the AST. The choice of JavaCC turned up to be sinister; when I was about to implement the lexc frontend I discovered that JavaCC simply couldn't tokenize certain parts correctly.

## 4 Frontend

There is only one frontend in the prototype application, for lexc source files. The lexc formalism is fairly easy to parse, there is a catch though: sometimes a string of characters should be interpreted as one token and sometimes each character alone should be considered a token. This can be solved in two ways, by means of scanner/parser co-operation, where the scanner asks the parser for the correct interpretation, or using a stateful scanner, where the interpretation depends on the current state of the scanner. The first alternative gives no reliable result within JavaCC, because the scanner may be several tokens ahead of the parser, and thus a stateful scanner was made.

There are more quirks though. A normal lexc file (see listing 1) declares multicharacter symbols before the lexicon part begins, i.e.,

[1] lexc

strings of characters intended to be interpreted together as a single token within the context where these characters normally should be considered as several one-character tokens. Since there are no characters reserved for marking the start and end of such multicharacter symbols, the only solution is to rely on the scanner's feature of choosing the longest possible match when tokenizing the character stream. All you have to do is to modify the symbol table of the scanner by adding the new symbols as soon as they are defined at the top of the source file, and the scanner will do the rest as usual. But because of efficiency issues, JavaCC constructs new scanners as finite-state automata and no symbol table is ever used. There is no way to add new keywords to the scanner at runtime. The suggested solutions to this problem is to run JavaCC and compile a new compiler at runtime (Viswanadha, 2004) (or simply implement that part of the scanner by hand). I decided to omit multicharacter symbol support in the frontend.

The lexc notation includes regular expressions, though they are not as frequently used as the multicharacter symbols. The regular expressions forms a sublanguage by themselves and while they play an important part in compactly expressing complex parts of finite automata, I found the benefit of including support for them lower than the cost, since they wouldn't contribute significantly to the usability of the prototype and yet require work corresponding to that of implementing a whole new frontend.

**Listing 1** A simple lexc snippet

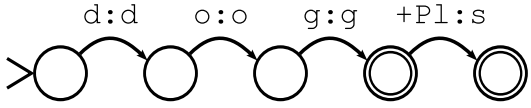
```
Multichar_Symbols +Pl

LEXICON Root
dog Noun;

LEXICON Noun
+Pl:s #;
#;
```

Some pitfalls exists regarding the generation of arcs and nodes from the lexc notation. As listing 1 shows, an entry in a lexicon may have no data and an end-of-word-token ("#"). The absence of data in an entry means that no arcs will be generated. No arcs means no nodes to mark as end nodes, which is a problem if the end-of-word-token is encountered. The solution is to always track the last node generated, even if it originated in an other lexicon, and to avoid generating nodes in advance, before it is known whether they will be used. By storing the first node each lexicon became connected to, the problem of finding nodes from preceding lexicon is solved along with another problem, the detection of cycles in the transducer. The first thing done when a lexicon is entered is to store the previous node encountered, unless that reference already points to a node, in which case a cycle has been detected and the processing of that particular lexicon should return (the lexicon has obviously been entered earlier).

## 5 Internal representation

The internal representation should as far as possible reflect the transducer as mathematical idea. In purifying it, the modularity of the compiler is preserved and the transducer becomes available for optimizing algorithms. No such algorithms have yet been implemented, but the possibility to combine transducers for re-writing rules with those handling lexicon is essential for any practical use of a phonological compiler, and a natural extension of the prototype presented would consist of algorithms for combining and optimizing transducers, together with a frontend for re-write rules.

**Figure 1** Abstract representation of a transducer




## 6 Backend

One backend has been implemented, producing Prolog predicates suitable for SWI-Prolog. The intention is to load the predicates together with a Prolog interface program bundled with the compiler, but the external program could have been included in the file with the predicates without any problems. The predicates consists of start node marker, arcs and end node markers, more or less a pretty-printing of the internal representation. The node representation has changed though, the interesting parts in the transducers are the labeled arcs and very little information is connected to the nodes, thus there is no need to build predicates around the nodes. These are instead represented by unique non-negative integers.

When building a Prolog program from several lexc files, it is convenient to allow several different transducers in the same program, one for each lexc file. To avoid mixing up the enumeration of the nodes, a Singleton pattern is used, producing one single sequence of unique integers during the run of the compiler. Alas, this enumeration had been put in the internal representation as mentioned above.

The backend simplifies the syntax of the output to make it more readable. A general arc is encoded as `arc/4`, with start node, end node, lexical form and surface form being its arguments. If an arc contains the same symbol in both the lexical form and the surface form, it is encoded as `arc/3`, where the two forms are represented by one argument.

Note that multicharacter symbols is no issue for the backend, if they exist in the internal representation, they will show up in the Prolog predicates. On the other hand, the interface program must be able to recognize these symbols in the user input. A solution in this particular combination of frontend and backend would be to store a list of the multicharacter symbols in the internal representation or passing them immediately to the backend, then encoding them as predicates; `multicharsymbol/1`. This would contaminate the internal abstract model and break the modularity of the compiler. A better solution would be to scan through all arcs in the backend and construct the list of all multicharacter symbols before encoding them as before. This strains the Prolog interface; it must find the longest match when tokenizing user input. It could be extended but that would require unnecessary effort. Implementing a new interface has its advantages, I am free to choose whatever formalism I prefer, thus, I force the user to surround each multicharacter symbol with percentage signs, following the principle of KISS[2]. No need to encode any symbols as predicates or juggle tokenization in Prolog, the (relatively small) burden is laid upon the user instead.

The user interface in Prolog provides three predicates:

**down(*+Atom*)** finds all possible translations of *Atom* from lexical (upper) form to surface (lower) form and prints them.

**up(*+Atom*)** finds all possible translations of *Atom* from surface (lower) form to lexical (upper) form and prints them.

**listall** lists all possible translations. It does not work for transducers containing cycles, for obvious reasons.

[2]Keep It Simple, Stupid!

**Listing 2** Prolog representation of a transducer

```
startnode( 0 ).
arc( 0, 1, 'd' ).
arc( 1, 2, 'o' ).
arc( 2, 3, 'g' ).
arc( 3, 4, '+Pl', 's' ).
endnode( 3 ).
endnode( 4 ).
```

## 7 Conclusions

Starting with an optimistic idea about implementing frontends for both lexc and twolc (Karttunen and Beesley, 1992), I found out I had to cut down on my ambition a bit.

The implementation of the backend and the associated user interface turned out to be easy to accomplish, mainly because I was able to rely on Prolog's backtracking instead of implementing an engine on my own. The cost of using backtracking is that speed is lost, even if the transducer could be treated like a deterministic machine, it will be treated like a nondeterministic dito (which is slower).

The lexc formalism seemed straight-forward to me, but as I soon discovered, it would have been easier to implement a scanner of my own rather than twisting and bending JavaCC for a task it was not designed to handle. Needless to say, I did not realize in the beginning that regular expressions were a whole language on their own.

It would have been interesting to test the efficiency of the compiler's output by compiling a huge database but that would require a new frontend, I have not seen such a database in lexc notation.

Some features which should be implemented if the prototype is extended:

- Multicharacter symbol support in scanner.
- Regular expression support.
- Algorithms for combining transducers.
- Algorithms for optimizing transducers.

## References


Torbjörn Ekman, Görel Hedin, and Eva Magnusson. 2005. Jastadd. http://jastadd.cs.lth.se.

Lauri Karttunen and Kenneth R. Beesley. 1992. Two-level rule compiler. Technical Report ISTL-92-2, Xerox Palo Alto Research Center, October. available at http://www.xrce.xerox.com/competencies/content-analysis/fssoft/docs/twolc-92/twolc92.html.

Lauri Karttunen. 1993. Finite-state lexicon compiler. Technical Report ISTL-NLTT2993-04-02, Xerox Palo Alto Research Center, April. available at http://www.xrce.xerox.com/competencies/content-analysis/fssoft/docs/lexc-93/lexc93.html.

Helmut Schmid. 2001. SFST. http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html.

Sreenivas Viswanadha. 2004. [javacc] changing keywords at runtime. answer in a support forum https://javacc.dev.java.net/servlets/ReadMsg?list=users&msgNo=408.

Sreenivas Viswanadha. 2006. Javacc. https://javacc.dev.java.net/.

Jan Wielemaker. 2005. SWI-Prolog. http://www.swi-prolog.org/.