A Programming Language for Finite State Transducers

Helmut Schmid

Institute for Natural Language Processing (IMS)
University of Stuttgart, Germany
schmid@ims.uni-stuttgart.de

SFST-PL is a programming language for finite-state transducers which is based on extended regular expressions with variables. SFST-PL is used by the Stuttgart Finite-State-Transducer (SFST) tools which are available under the GNU public license. SFST-PL was designed as a general programming language for the development of tokenizers, pattern recognizers, computational morphologies and other FST applications. The first SFST application was the SMOR morphology [1], a large-scale German morphology which covers composition, derivation and inflection. An SFST program consists of a list of variable and alphabet assignments followed by a single regular expression which defines the resulting transducer. The following basic transducer expressions are available:

a:b	defines a transducer which maps the symbol **a** to **b**
a	abbreviation of **a:a**
a:.	maps the symbol **a** to any symbol that it occurs with in the alphabet (see below).
.	abbreviation of **.:.**, the union of all symbol-pairs in the alphabet.
[abc]:[de]	identical to **a:d | b:e | c:e** ("|" is the union operator.)
[a-c]:[A-C]	same as **[abc]:[ABC]**.
{abc}:{de}	identical to **a:d b:e c:<>** This expression maps the string **abc** to **de**.
var	the transducer stored in variable *var*.
"lex"	a transducer consisting of the union of the lines in the file *lex* (Apart from ":" and previously seen multi-character symbols, all symbols in the argument file are interpreted literally.)
"<file>"	is a pre-compiled transducer which is read from *file*

SFST-PL supports multi-character symbols (which are enclosed in angle brackets like `<Sg>`) and a wide range of operators including concatenation, union '|', intersection '&', composition '||', complement '!', optionality '?', Kleene star '*' and Kleene plus '+', range '^', domain '_', inversion '^_', and two-level rules (`<=`, `=>`, `<=>`). The special symbol `<>` represents the empty string.

Variables are surrounded by dollar signs. They are defined with a command `$var$ = expression` (where `expression` is some transducer expression). The alphabet is defined with the command `ALPHABET = expression`. The definition

A. Yli-Jyrä, L. Karttunen, and J. Karhumäki (Eds.): FSMNLP 2005, LNAI 4002, pp. 308–309, 2006.

of an alphabet is required for the interpretation of the wild-card symbol '.' and for the complement and replacement operators.

Comments start with a percent sign and extend up to the end of the line. Whitespace is ignored unless it is quoted by a backslash. Programs can be partitioned into several files which are combined with include commands (like `#include "file"`) which insert the contents of the argument file at the current position. It is also possible to pre-compile component transducers in order to speed up the compilation.

A compiler translates SFST programs into minimized finite-state transducers. The compiler was implemented using a high-level C++ library and the YACC compiler generator, which makes it easy to change or extend the syntax of the programming language. The compiler generates three different transducer formats which are optimized for flexibility, speed or memory and startup efficiency, respectively. The SFST tools also include programs for analysis, printing, and comparison of transducers. The following simple SFST-PL program will correctly inflect adjectives like "easy" (easier, easiest) and late (later, latest).

```
% the set of valid character pairs
ALPHABET = [A-Za-z]:[A-Za-z] y:i [#e]:<>

% Read a list of adjectives from a lexicon file
$WORDS$ = "adj"

% rule replacing y with i if followed by # and e
$Rule1$ = y <=> i (#:<> e)

% rule eliminating e if followed by # and e
$Rule2$ = e <=> <> (#:<> e)

$Rules$ = $Rule1$ & $Rule2$

% add inflection to the words
$S$ = $WORDS$ <ADJ>:# ({<pos>}:{} | {<comp>}:{er} | {<sup>}:{est})

% apply the phonological rules to obtain the resulting transducer
$S$ || $Rules$
```

A more comprehensive morphology including mechanisms for dealing with derivation, compounding and inflection is available with the SFST tools. It is adaptable to other languages by changing the lexicon, the inflectional classes, and the phonological rules.

References

1. H. Schmid, A. Fitschen, and U. Heid. SMOR: A German computational morphology covering derivation, composition and inflection. In *Proceedings of the 4th International Conference on Language Resources and Evaluation*, volume 4, pages 1263–1266, Lisbon, Portugal, 2004.