A Programming Language For Finite State Transducers

Helmut Schmid

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

Abstract. This paper presents SFST-PL, a programming language for finite state transducers which is based on extended regular expressions with variables. The programming language is both simple and general and suitable for a wide range of possible applications. A compiler for the programming language is provided by the SFST tools which have successfully been used to implement a large-scale German morphology.

1 Introduction

The Stuttgart Finite State Transducer (SFST) tools have been developed for the implementation of SMOR[1], a large German computational morphology covering derivation, composition and inflection. The specification of the SMOR transducer was written in SFST-PL, the programming language of the SFST tools. SFST-PL is a general-purpose FST programming language comprising all the descriptive means for the development of finite-state transducers in a single formalism. SFST-PL is flexible and not committed to a single morphological formalism like two-level morphology [2] or ordered rewrite rules [3].

2 The Programming Language

Any regular expression is already a valid SFST program. An example is the expression:

```
Hello\ world\!
```

It defines a transducer which maps the string *Hello world!* onto itself and rejects any other input. The blank and the exclamation mark have to be quoted with a backslash because unquoted blank and tab characters are ignored and unquoted exclamation marks are interpreted as negation operators.

A less trivial mapping is performed by the following transducer for the inflection of the nouns *foot*, *house*, and *mouse*:

```
(house | foot | mouse) <N>:<> <sg>:<> |\
(house<>:s | f o:e o:e t | {mouse}:{mice}) <N>:<> <pl>:<>
```

This example uses the or-operator '`|`'. Strings which are enclosed in angle brackets (like `<N>`, `<sg>`, and `<pl>` in the example) are multi-character symbols. They are treated in the same way as single-character symbols. '`<>`' is a special symbol representing the empty string. The expression '`house<>:s`' therefore maps *house* to *houses* (in generation mode). '`fo:eo:et`' specifies a transducer which maps *foot* to *feet*. The expression `{mouse}:{mice}` is equivalent to `m:mo:iu:cs:ee:<>` and maps *mouse* to *mice*.

Regular expressions terminate at the end of a line unless the end of the line is quoted with a backslash. This was the reason for adding a backslash at the end of the first line in the preceding example.

Variables

An SFST program is essentially a regular expression. This expression may be quite complex for transducers which perform sophisticated tasks. Using variables, it is possible to break it into smaller units. The preceding program, for instance, can be reformulated as follows:

```
$Nsg$ = house | foot | mouse
$Npl$ = house<>:s | f o:e o:e t | {mouse}:{mice}
$Nsg$ <N>:<> <sg>:<> | $Npl$ <N>:<> <pl>:<>
```

The first two lines define the two variables `$Nsg$` and `$Npl$`. Variable names begin and end with a dollar sign. The right-hand side of a variable definition is any valid regular expression (see the next section).

SFST programs usually consist of a long sequence of variable definitions followed by a single expression which specifies the result transducer.

Basic Regular Expressions

The transducer expressions are defined over a set of **symbol pairs**. Symbols are specified in one of the following ways:

- single characters like '`A`' or ''
- quoted characters like '`\ `' or '`\!`'
- quoted numbers like '`\32`' which are translated to the character with the respective code ('`\32`' translates to the blank character.)
- multi-character symbols like `<N>` or `<k$ln_5>` which are enclosed in angle brackets
- the special symbol `<>` which designates the empty string

The following expressions are examples of **basic regular expressions**:

a:b	defines a transducer which maps the symbol **a** to the symbol **b**.
a	is identical to **a:a**

a:. maps the symbol **a** to any symbol allowed by the alphabet. (The alphabet is described below.)

. is identical to **.:.**. The transducer defined by this expression performs any mapping of a single symbol allowed by the alphabet.

[abc]:[de] is identical to **a:d | b:e | c:e** ("|" is the or-operator which is introduced below.)

[a-d]:[A-C] is identical to **[abcd]:[ABC]**.

{abc}:{de} is identical to **a:d b:e c:<>** This expression maps **abc** to **de**.

var is a variable (see below)

Operators

More complex expressions are built by combining transducer expressions with operators. SFST-PL supports the following operators:

rs concatenation
If **r** maps the string α to β and **s** maps γ to δ, then **rs** maps the string $\alpha\gamma$ to $\beta\delta$.

r|s union
r|s maps α to β iff **r** or **s** maps α to β.

r||s composition
r||s maps α to γ iff **r** maps α to some β and **s** maps β to γ.

r&s intersection
r&s maps the string $a_1a_2...a_n$ to $b_1b_2...b_n$ iff both **r** and **s** map $a_1a_2...a_n$ to $b_1b_2...b_n$ by aligning a_i with b_i for all $1 \leq i \leq n$. Either a_i or b_i is allowed to be the empty symbol.
Note: Although **a:b** and **a:<><>:b** both map the string **a** to **b**, the result of **a:b & (a:<><>:b)** is nevertheless empty because the alignment is different.

!r complement
!r maps the string $a_1a_2...a_n$ to $b_1b_2...b_n$ iff the alphabet contains a_i:b_i for $1 \leq i \leq n$, and $r\&(a_1{:}b_1a_2{:}b_2...a_n{:}b_n)$ is the empty transducer. Either a_i or b_i is allowed to be the empty symbol.

r? optionality
identical to **<>|r**

r* Kleene star
if **r** maps α to β, then $r*$ maps n repetitions of α to n repetitions of β, with $0 \leq n$.

r+ Kleene plus
identical to **r r***

^ r range
maps β to β iff **r** maps some string α to β.

_r domain
maps α to α iff **r** maps α to some string β.

^ _r inversion
maps α to β iff **r** maps β to α.

SFST also supports single-symbol replacement operators [4]:

l a <= b r obligatorily maps the symbol **a** to **b** if **l** precedes and **r** follows. (Elsewhere, the mapping of **a** to **b** is optional. **l** and **r** are arbitrary regular expressions.)
This expression is identical to **!(.* l (a:. & !a:b) r .*)**

l a => b r allows the mapping of symbol **a** to **b** only if **l** precedes and **r** follows. (The mapping of **a** to **b** is optional in this context.)
The expression is equivalent to **!(!(.* l) a:b .* | .* a:b !(r .*))**

l a <=> b r maps the symbol **a** to **b** if and only if **l** precedes and **r** follows.
The expression is identical to **(l a => b r) & (l a <= b r)**

Finally, there are two commands which create transducers from files.

"lex" lexicon reader
reads a text file named *lex* and returns the disjunction of the lines of the file (see below)

"<inc>" transducer reader
reads a precompiled transducer from a binary file named *inc* and returns it (see below).

The Alphabet

The alphabet contains a set of symbol pairs which is required for the interpretation of the wildcard symbol '.'. The following SFST program e.g. defines a transducer which maps a sequence of letters to the same sequence of letters, but with lower-case letters replaced by upper-case characters (in generation mode):

```
ALPHABET = [A-Z] [a-z]:[A-Z]
.*
```

The first line defines the alphabet. The right-hand side of this assignment is a transducer expression. The set of symbol pairs is obtained by (i) compiling the expression to a transducer and (ii) extracting the symbol pairs from all state transitions of the transducer.

The definition of an alphabet is also required by the negation operator and the replacement operators.

Comments

Comments in SFST programs start with a percent character (`%`) and extend up to the end of the line. The following lines of code return the expression `abc`.

```
% This is a comment
abc % This is another comment
% This comment also extends up to the end of the line % cde
```

Lexica

The lexicon entries of morphological analyzers are usually stored in a separate file, one entry per line. SFST-PL provides the operator `"lexicon"` to read lexicon entries from a file called *lexicon*. The result of the operator is a disjunction of all the lines from the file. If the file *lexicon* contains the following entries

```
house
mouse
foot
```

then the result of the operator `"lexicon"` is the expression `house|mouse|foot`. The file reader treats all characters literally except for :, `<>`, \, and multi-character symbols which have been introduced before.

Include Command

Complex computer programs are usually stored in a set of files rather than a single file, and the compiler combines these files to a single program. The same can be done with SFST programs. The command `#include "file.fst"` instructs the compiler to insert the contents of the file *file.fst* at the current position.

SFST programs create complex transducers by combining simpler transducers. If the compilation of some component transducer is expensive and the respective source code is seldom modified, it is useful to *pre-compile* this transducer. To this end, a separate SFST program has to be written which implements the component transducer. This program is compiled and the resulting transducer is stored e.g. in a file named *inc.a*. The main program reads the precompiled transducer with the command `"<inc.a>"`.

3 Compilation

The SFST compiler translates the extended regular expressions of the source code file to transducers. The result transducer is minimized and stored in the output file. Transducers which are assigned to variables are minimized, as well. The compiler is quite efficient and was successfully used to compile SMOR, a large German computational morphology.

The SFST tools support three different transducer formats which are optimized for flexibility, processing speed and start-up time/memory efficiency, respectively. The implementation of the SFST tools is based on a C++ class library which facilitates the development of new analysis tools. A sample C++ program is shown in the appendix.

4 Summary

We presented SFST-PL, a programming language for the development of finite-state-transducers. SFST-PL is based on extended regular expressions with variables and supports most operations on finite-state transducers including single character replacement. (String replacement operators have not been implemented, yet.) SFST-PL is simple and easy to learn. It is suitable for a wide range of applications (except for weigthed transducers).

The SFST tools provide a compiler which translates SFST programs into transducers and tools for analysis, printing etc. The SFST tools have been implemented based on a C++ library which is easy to use and facilitates the development of new tools. The SFST tools are freely available under the GNU public license.

References

1. Schmid, H., Fitschen, A., Heid, U.: SMOR: A German computational morphology covering derivation, composition and inflection. In: Proceedings of the 4th International Conference on Language Resources and Evaluation. Volume 4., Lisbon, Portugal (2004) 1263–1266
2. Koskenniemi, K.: Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production. PhD thesis, University of Helsinki (1983)
3. Kaplan, R., Kay, M.: Regular models of phonological rule systems. Computational Linguistics **20** (1994) 331–379
4. Karttunen, L., Koskenniemi, K., Kaplan, R.M.: A compiler for two-level phonological rules. In Dalrymple, M., Kaplan, R.M., Karttunen, L., Koskenniemi, K., Shaio, S., Wescoat, M., eds.: Tools for Morphological Analysis. Volume 87-108 of CSLI Reports. Center for the Study of Language and Information, Stanford University, Palo Alto, CA (1987) 1–61

A SFST Example Program

The following sample code implements an analyzer for English adjectives which maps `happier` to `happy<JJR>` and `latest` to `late<JJS>`.

```
% define the alphabet for the following replacement rules
% The morphological markers <JJ>, <JJR> and <JJS> are deleted
% on the surface
ALPHABET = [A-Za-z] [#<JJ><JJR><JJS>]:<> y:i e:<>

% rule replacing y with i
$R1$ = y <=> i (#:<> e)

% rule eliminating e
$R2$ = e <=> <> (#:<> e)

% the conjunction of these rules
$R$ = $R1$ & $R2$

% create a recognizer for the words in the file "adj"
$WORDS$ = "adj"

% append the different endings to the words
$S$ = $WORDS$ <>:# (<JJ> | {<>}:{er}<JJR> | {<>}:{est}<JJS>)

% apply the phonological rules to obtain the result transducer
$S$ || $R$
```

B A C++ Example Program

The following code shows the implementation of a program which converts the input string into a transducer, composes it with two other transducers which are read from files, minimizes the result and extracts and prints the analysis strings.

```
#include <stdio.h>
#include "fst.h"

int main(int argc, char **argv) {
  FILE *file1=fopen("file1","rb");
  FILE *file2=fopen("file2","rb");
  char *s="Hello";
  Automaton a1(file1);
  Automaton a2(file2);
  Automaton a(s);

  (a1 || a2 || a).minimise().lower_level().print_strings(stdout);
}
```