The Kleene Language for Weighted Finite-State Programming:
User Documentation, Version 0.9.4.1

This Document is Work in Progress
Corrections and Suggestions Are Welcome

Kenneth R. Beesley
SAP Labs, LLC
P.O. Box 540475
North Salt Lake
Utah 84054, USA
ken.beesley@sap.com

7 July 2014
Preface

What is Kleene?

Kleene is a programming language that can be used to create many useful and efficient linguistic applications based on finite-state machines (FSMs). These applications include tokenizers, spelling checkers, spelling correctors, morphological analyzer/generators and shallow parsers. FSMs are also widely used in speech synthesis and speech recognition.

Kleene allows programmers to define, build, manipulate and test finite-state machines using regular expressions and right-linear phrase-structure grammars; and Kleene supports variables, rule-like expressions, user-defined functions and familiar programming-language control syntax. The FSMs can include acceptors and two-projection transducers, either weighted under the Tropical Semiring or unweighted. If this makes no sense, the purpose of the book is to explain it.

Pre-edited Kleene scripts can be run from the command line, and a graphical user interface is provided for interactive learning, programming and testing.

Operating Systems

Kleene runs on OS X and Linux, requiring Java version 1.6 or higher.

Prerequisites

This book assumes only superficial familiarity with regular languages, regular relations and finite-state machines.

While readers need not have any experience with finite-state programming, those who have no programming experience at all, e.g. in a language like Java, C++, Perl, Python, Ruby, etc., will find it difficult.
The Kleene Language

Implementation

The Kleene parser is implemented in JavaCC/JJTree* and Java,† and the interpreter calls functions in the OpenFst library‡ via the Java Native Interface (JNI).§ Kleene has its own syntax, based on familiar regular expressions, and it borrows much of its semantics from the Xerox/PARC finite-state toolkit.¶

Language-restriction expressions and alternation rules are implemented using algorithms supplied by Måns Huldén.‖

Acknowledgments

Thanks are due to the following people for advice and encouragement over the years. I have not always followed the advice I received, so I take full responsibility for any problems, failures or infelicities.

Lauri Karttunen, Måns Huldén, Paola Nieddu, Phil Sours, Cyril Allauzen, André Kempe, Mike Wilkens, Kemal Oflazer, Natasha Lloyd

Kleene is named after American mathematician Stephen Cole Kleene (1909–1994), who investigated the properties of regular sets and invented the metalanguage of regular expressions.

---

*http://java.net/projects/javacc
†http://www.java.com/
§http://www.ibm.com/developerworks/java/tutorials/j-jni/
¶http://www.fsmbook.com
License

Kleene is free and open-source, released 4 May 2012 by SAP AG†† under the Apache License, Version 2.** The OpenFst library is available under the same license.

††http://www.sap.com/
**http://www.apache.org/licenses/LICENSE-2.0
# Contents

1 Introduction ......................................................... 1  
   1.1 What is Kleene? ........................................... 1  
   1.2 The Kleene Name ........................................... 3  
   1.3 Possible Applications ................................... 3  
   1.4 Design Criteria .......................................... 3  
   1.5 Terminology ............................................... 5  

2 Getting Started with Kleene ....................................... 7  
   2.1 Installation and Prerequisites ............................ 7  
      2.1.1 Pre-compiled Binaries .............................. 7  
      2.1.2 Compiling Kleene from Source Code ............... 8  
   2.2 Launching the Kleene GUI ................................. 9  
   2.3 OK, What can you do in Kleene? ........................ 11  
      2.3.1 First, is it working? ............................... 11  
      2.3.2 Visualizing Finite-State Machines ................. 11  
      2.3.3 Compile regular expressions into FSMs .......... 13  
      2.3.4 Optimization of FSMs .............................. 23  
      2.3.5 Using Defined Variables ............................ 26  
      2.3.6 Characters ........................................... 27  
      2.3.7 Regular Languages and Regular Relations ........ 31  
   2.4 Looking at FSMs ............................................ 41  
   2.5 Applications ............................................... 42
3 Regular Expression Syntax

3.1 Regular Expressions ........................................ 45
  3.1.1 Primary Regular Expressions ......................... 46
  3.1.2 Inherently Delimited Regular Expressions ......... 47
  3.1.3 Regular-Expression Operators ....................... 49
  3.1.4 Precedence Issues .................................... 49
  3.1.5 Weights .................................................. 52
  3.1.6 Whitespace in Regular Expressions ................ 54
  3.1.7 Denoting the Empty String ......................... 55
  3.1.8 Denoting Any Symbol ................................ 56

3.2 Abstraction Mechanisms .................................. 57
  3.2.1 Variables ................................................. 57
  3.2.2 Pre-defined Functions ................................ 58
  3.2.3 User-defined Function Syntax ....................... 60
  3.2.4 Function Call Semantics .............................. 64
  3.2.5 Function Parameters with Default Values .......... 66

3.3 Right-linear Phrase-structure Grammars ................. 68
  3.3.1 Right-linear Syntax .................................. 68
  3.3.2 Right-linear Semantics ............................... 70

3.4 Scope ......................................................... 72
  3.4.1 Assignments, Declarations and Local Scope ....... 72
  3.4.2 Stand-alone Code Blocks ............................. 73
  3.4.3 Function Blocks ........................................ 73
  3.4.4 External Variables ..................................... 74
  3.4.5 Free Variables ......................................... 77
  3.4.6 Combining Free and Local Usage .................... 78
  3.4.7 Export Statements ..................................... 79
  3.4.8 Practical Use of Code Blocks ....................... 80

3.5 Language Restriction Expressions ....................... 83

4 Alternation Rules ............................................ 87
  4.1 What are Alternation Rules? ............................. 87
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Mindtuning for Alternations</td>
<td>88</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Underlying and Surface, Upper and Lower</td>
<td>88</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Writing and Testing Alternation Rules</td>
<td>92</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Derivations or Cascades of Rules</td>
<td>94</td>
</tr>
<tr>
<td>4.2.4</td>
<td>The Richness of Alternation-Rule Types</td>
<td>99</td>
</tr>
<tr>
<td>4.3</td>
<td>Basic Mapping Rules</td>
<td>100</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Right-Arrow Rules</td>
<td>100</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Left-arrow Rules</td>
<td>105</td>
</tr>
<tr>
<td>4.4</td>
<td>Optional Alternation Rules</td>
<td>106</td>
</tr>
<tr>
<td>4.5</td>
<td>Maximum and Minimum Matching</td>
<td>107</td>
</tr>
<tr>
<td>4.6</td>
<td>Deletion Rules</td>
<td>108</td>
</tr>
<tr>
<td>4.7</td>
<td>Epenthesis Rules</td>
<td>108</td>
</tr>
<tr>
<td>4.8</td>
<td>Markup Rules</td>
<td>109</td>
</tr>
<tr>
<td>4.9</td>
<td>Two-Level Rule Contexts</td>
<td>110</td>
</tr>
<tr>
<td>4.10</td>
<td>Parallel Rules</td>
<td>112</td>
</tr>
<tr>
<td>4.10.1</td>
<td>Vertical Derivations vs. Parallel Rules</td>
<td>112</td>
</tr>
<tr>
<td>4.10.2</td>
<td>The Built-In Parallel Rule-Compilation Function</td>
<td>115</td>
</tr>
<tr>
<td>4.10.3</td>
<td>Rules with Where-Clauses</td>
<td>119</td>
</tr>
<tr>
<td>4.10.4</td>
<td>Rules where the Input can Match the Empty String</td>
<td>124</td>
</tr>
<tr>
<td>4.11</td>
<td>Transducer-Style Rules</td>
<td>125</td>
</tr>
<tr>
<td>4.11.1</td>
<td>Traditional Rules vs. Transducer-Style Rules</td>
<td>125</td>
</tr>
<tr>
<td>4.11.2</td>
<td>Simple Mapping Rules in the Transducer Style</td>
<td>127</td>
</tr>
<tr>
<td>4.11.3</td>
<td>Transducer-Style Rules vs. Where-Clause Rules</td>
<td>133</td>
</tr>
<tr>
<td>4.11.4</td>
<td>Transducer-Style Rules vs. Traditional Markup Rules</td>
<td>134</td>
</tr>
<tr>
<td>4.11.5</td>
<td>Transducer-Style Rules Not Expressible as Traditional Rules</td>
<td>135</td>
</tr>
<tr>
<td>5</td>
<td>Examples without Weights</td>
<td>137</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>137</td>
</tr>
<tr>
<td>5.2</td>
<td>Spanish Morphology</td>
<td>137</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Spanish Verbs</td>
<td>137</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Spanish Nouns</td>
<td>150</td>
</tr>
</tbody>
</table>
5.3 Latin Morphology ........................................ 150
5.4 Aymara Morphology .................................... 150

6 Examples with Weights ...................................... 151
   6.1 Introduction ........................................... 151
   6.2 Weighted Edit Distance for Spelling Correction .... 151

7 Challenges in Morphology ................................ 163
   7.1 Overview .............................................. 163
   7.2 Infixation ............................................. 164
   7.3 Reduplication ......................................... 164
      7.3.1 Range of Reduplicative Phenomena ............ 164
      7.3.2 Contiguous Reduplication ...................... 165
      7.3.3 Discontiguous Reduplication .................. 174
   7.4 Semitic Interdigitation ............................... 174
   7.5 Constraining Long-distance Dependencies .......... 174

8 Arithmetic Expressions ................................... 175
   8.1 Mindtuning for Arithmetic Expressions .............. 175
   8.2 Primary Arithmetic Expressions ..................... 176
   8.3 Arithmetic Expression Operators ................... 177
   8.4 Arithmetic Functions ................................ 177
   8.5 Boolean Functions ................................... 179
      8.5.1 Special Case of Arithmetic Functions ........ 179
      8.5.2 Assert and Require Statements ................. 181

9 Lists .......................................................... 183
   9.1 Lists of FSMs and Numbers ......................... 183
   9.2 List Literals, Identifiers, and Assignment .......... 184
   9.3 Pre-Defined Functions Operating on Lists ......... 185
      9.3.1 Functions Joining the Elements of a List .... 188
      9.3.2 Functions Returning the Map of a List ....... 190
      9.3.3 Functions Returning the Alphabet (Sigma) of an FSM 191
9.3.4 Functions Returning the Size of a List . . . . . . . . 192
9.4 Iteration through Members of a List . . . . . . . . . . . 192
9.5 User-Defined Functions and Lists . . . . . . . . . . . . . 193
9.6 Traditional Array-indexing Syntax . . . . . . . . . . . . 194

10 Other Syntax 197
10.1 Void Functions with Names . . . . . . . . . . . . . . . . 197
10.2 Anonymous Void Functions . . . . . . . . . . . . . . . . . 197
10.3 Control Syntax . . . . . . . . . . . . . . . . . . . . . . . . 198
10.4 Input/Output . . . . . . . . . . . . . . . . . . . . . . . . . . 198
  10.4.1 Scripts . . . . . . . . . . . . . . . . . . . . . . . . . 198
  10.4.2 XML Input/Output . . . . . . . . . . . . . . . . . . 201
  10.4.3 DOT Output . . . . . . . . . . . . . . . . . . . . . 202
  10.4.4 Interactive Testing in the GUI . . . . . . . . . . . . 203
10.5 Memory Management . . . . . . . . . . . . . . . . . . . . 205
  10.5.1 Garbage Collection . . . . . . . . . . . . . . . . . . 205
  10.5.2 Java Memory Usage . . . . . . . . . . . . . . . . . 206
  10.5.3 Java Fst Objects . . . . . . . . . . . . . . . . . . . 206
  10.5.4 Contents of the Main Symbol Table . . . . . . . . 207
  10.5.5 C++ Memory Space . . . . . . . . . . . . . . . . . 207

11 Code Generation and Runtime Code 209
11.1 Using Kleene FSMs in Real Projects . . . . . . . . . . . . 209
11.2 The fst2java Experiment . . . . . . . . . . . . . . . . . . . 210
  11.2.1 Stand-Alone Java Code Generation . . . . . . . . . 210
  11.2.2 How it Works . . . . . . . . . . . . . . . . . . . . . 211
  11.2.3 How to Create the Java Class Files for an FSM . . . 213
  11.2.4 FSM to Java Examples . . . . . . . . . . . . . . . 221
  11.2.5 Generated Java Code API . . . . . . . . . . . . . 231
11.3 Runtime Code . . . . . . . . . . . . . . . . . . . . . . . . . 234
  11.3.1 Status of Runtime Code . . . . . . . . . . . . . . . 234
  11.3.2 Basic Functionality of Runtime Code . . . . . . . . 235
  11.3.3 Kinds of Runtime Code . . . . . . . . . . . . . . . 236
C.2 Illegal Characters Inside Multichar-symbol Names and Double-Quoted Strings ........................................ 266
C.3 Control Characters and Writing Networks to XML 1.0 .... 267

D  GUI FSM Testing and OTHER 269

E  Pre-defined FSM-valued Functions 271
   E.1 Commonly Used Functions ................................... 272
   E.2 Less Used Functions ........................................... 273
   E.3 Functions for Use by Experts ................................. 273

F  Pre-defined List Functions 275
   F.1 Pre-defined Functions for FSM Lists ...................... 275
   F.2 Pre-defined Functions for Number Lists .................. 276

G  Pre-defined Case Functions 277

H  Pre-defined Diacritic Functions 279

I  Pre-defined Arithmetic-valued Functions 281
   I.1 General ....................................................... 281
   I.2 Boolean ...................................................... 282
Chapter 1

Introduction

1.1. What is Kleene?

Kleene is a programming language for building and manipulating finite-state machines (FSMs), and these finite-state machines can be used as tokenizers, spelling checkers, spelling correctors, morphological analyzer/generators, shallow parsers, speech generators and speech recognizers. Applications implemented using finite-state machines are mathematically beautiful, unusually modifiable and re-usable, and typically more efficient than the alternatives. This book will concentrate mostly on showing you how to define FSMs that implement morphological analyzer/generators.

The advantages of computing with finite-state machines are well known, and numerous computer implementations have been developed. Dr. Anssi Yli-Jyrä maintains a formidable list of finite-state projects at https://kitwiki.csc.fi/twiki/bin/view/KitWiki/FsmReg. Kleene is firmly in the tradition of the AT&T Lextools (Roark and Sproat, 2007),\(^1\) the SFST-PL language (Schmid, 2005),\(^2\) The HFST software,\(^3\) the Xerox/PARC finite-

\(^1\)http://serrano.ai.uiuc.edu/catms/
\(^2\)http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html
\(^3\)http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/
state toolkit (Beesley and Karttunen, 2003);\(^4\) the foma language\(^5\) and OpenGrm-Thrax,\(^6\) all of which provide higher-level programming formalisms built on top of low-level finite-state libraries.

Kleene allows programmers to specify finite-state machines, including acceptors that encode regular languages and two-projection transducers that encode regular relations, using both regular expressions and right-linear phrase-structure grammars. The FSMs can be weighted, under the Tropical Semiring, or unweighted. The language supports variables, user-defined functions and a rich set of alternation-rule notations, plus familiar control structures such as if-elsif-else statements and while loops.\(^7\)

The Java-language Kleene parser, implemented with JavaCC and JJTree (Copeland, 2007),\(^8\) is Unicode-capable and portable.\(^9\) Kleene currently has pre-compiled binaries available for OS X and Linux.

The Kleene graphical user interface (GUI), implemented with the Java Swing library, allows interactive creation, testing and graphic display of finite-state machines. The overall application window encloses a Unicode-capable terminal-like window, into which Kleene statements can be typed, and a symbol-table window that displays an icon for each defined machine. Right-clicking on an icon triggers a pop-up menu with commands to test, draw, invert, determinize, draw, delete, etc. the associated machine.

Kleene is copyrighted by SAP AG and released under the Apache License, Version 2, which is a very liberal license that allows even commercial usage without payment of license fees or royalties. The OpenFst Li-

\(^4\)http://www.fsmbook.com
\(^5\)https://code.google.com/p/foma/
\(^6\)http://openfst.cs.nyu.edu/twiki/bin/view/GRM/Thrax
\(^7\)Kleene was first reported at the 2008 conference on Finite-State Methods in Natural-language Processing (Beesley, 2009).
\(^8\)https://javacc.dev.java.net
\(^9\)Successfully parsed Kleene statements are reduced to abstract syntax trees (ASTs); and the Java-language interpreter, implemented using the visitor design pattern,\(^10\) interprets the ASTs by calling C++ functions in the OpenFst finite-state library (Allauzen et al., 2007) via the Java Native Interface (JNI) (Gordon, 1998; Liang, 1999).
brary on which Kleene is based is also released under the same Apache License, Version 2. See the Apache site at http://www.apache.org/licenses/ for the full details.

1.2. The Kleene Name

Kleene is named in honor of American mathematician Stephen Cole Kleene (1909–1994), who, among many accomplishments, investigated the properties of regular sets, including regular languages, and invented the metalanguage of regular expressions, which is the foundation of Kleene-language syntax. In the USA, the name Kleene is commonly pronounced /kliːn/, like the English word clean, or /ˈkliːni/, but Kleene himself reportedly pronounced it /ˈkleni/.11

1.3. Possible Applications

Kleene can build acceptors to be used in spell-checking and similar applications. Kleene transducers can be used in spelling correction, tokenization, phonological modeling, morphological analysis and generation, speech synthesis and generation, and shallow or “robust” parsing. [KRB: expand this section]

1.4. Design Criteria

The following requirements and desiderata have guided the design and implementation of Kleene. You will have to judge if the project has succeeded.

1. The Kleene language must be compelling, easy to learn and well documented. The syntax and semantics should always aim for maximum

familiarity and “least astonishment.”

2. Programmers must be able to run pre-edited scripts or type statements interactively into a GUI.

3. Kleene must allow finite-state machines to be defined using both regular expressions and right-linear phrase-structure grammars.

4. The syntax should follow, as far as possible and appropriate, the familiar syntax of Perl-like regular expressions. Non-regular features of Perl regular expressions, such as back-references, will be excluded; and operators will be added to denote weights, two-projection relations, subtraction, complementation and intersection, which are lacking in Perl-like regular expressions.

5. The abstraction mechanisms must include variables, built-in functions and user-defined functions.

6. The syntax will include rule-like expressions similar in semantics and function to the Xerox/PARC Replace Rules (Karttunen, 1995; Karttunen and Kempe, 1996; Kempe and Karttunen, 1996; Mohri and Sproat, 1996) that denote regular relations and compile into finite-state transducers.

7. Unicode must be supported from the beginning, not only in data strings, but also in Kleene identifiers and operators.

8. The implementation should be maximally portable.

9. The implementation should be maximally modular, allowing the writing of interpreters based on various finite-state libraries that might become available.

12: This was a fundamental criterion from the very beginning, and Mike Wilkens correctly identified it as the major reason that Kleene looks the way it does.
1.5. Terminology

A *regular language* is a set of strings that can be encoded as a finite-state acceptor. A *regular relation* is a set of ordered string pairs that can be encoded as a two-projection (or “two-tape”) finite-state transducer. Theoretically, transducers can have any number of projections, but most practical implementations—including the Xerox/PARC, AT&T, SFST and OpenFst implementations—have been limited to two. Kleene can be used to build finite-state acceptors and two-projection finite-state transducers.

In some traditions, the terms *finite-state machine* and *finite-state automaton* (plural: *automata*) are used to denote acceptors, in contrast to finite-state *transducers*. Herein, however, I avoid the term automaton and use the term *finite-state machine* to encompass both acceptors and transducers.\(^{13}\)

I dislike acronyms and initialisms, especially as they have proliferated and become increasingly ambiguous; but some are well established and almost unavoidable in the field. I will often use *FST* to refer to finite-state transducers, and *FSM* to refer to finite-state machines (encompassing both finite-state acceptors and transducers). Where it is important to distinguish a finite-state machine as an acceptor, it will be called an acceptor, without abbreviation.

Complicating the terminological issue is the fact that OpenFst finite-state machines are always two-projection transducers in their structure; each arc in an OpenFst finite-state machine has an “input label” and an associated “output label.” In the OpenFst implementation of finite-state machines, an acceptor is just a special case of a transducer, which can also be considered an *identity transducer*, wherein each input label is equal to its associated output label. These structural issues, and the terminology, will be discussed again in more detail below in a section on how finite-state machines are implemented and visualized in various traditions.

\(^{13}\)In the Xerox/PARC tradition, the term *network* is used as a cover term to include both acceptors and transducers, but this usage has not become general.
Chapter 2

Getting Started with Kleene

2.1. Installation and Prerequisites

2.1.1. Pre-compiled Binaries

This chapter, which is definitely work in progress, is designed to get you started in Kleene and to give you a quick overview of its capabilities. The features presented here informally will be treated again with more rigor later in the book.

As you read this chapter, it is recommended that you follow along, testing the examples. This will, of course, require that you have Kleene and its prerequisites installed and running on your own computer. By far the easiest way to install Kleene is by downloading one of the pre-compiled binary versions from http://www.kleene-lang.org. At the time of writing, such pre-compiled binaries are available for

- Apple OS X, and
- Linux, compiled on Mint

Check the web site for the latest offerings. It is hoped that Kleene will someday be available on Windows.
To install Kleene, you will need to have some basic computer skills, including knowing how to download files from the Internet, launch a terminal application, move around the directory structure, copy files, edit files, etc. If this means nothing to you, seek the help of an expert.

The precompiled binaries have names like `kleene-mac-0.9.4.0.tar.gz` and `kleene-linux-0.9.4.0.tar.gz`; these are sometimes called *tarballs*. More recent versions will have higher numbers. Once downloaded, the tarball should be moved to a convenient location of your choice, e.g. into directory `~/kleene/`, and then “unpacked” with the command

```
$ gunzip kleene-mac-0.9.4.0.tar.gz
```

This should produce a file named `kleene-mac-0.9.4.0.tar`, and then “untar” that file with

```
$ tar xvf kleene-mac-0.9.4.0.tar
```

which should produce a directory named `kleene-mac-0.9.4.0`. Inside that directory, find a file named `README.install` and read it carefully, following the instructions. Again, get help as necessary from friends and associates who have experience in installing computer software.

You will also need to have Java 1.6, or a newer version, installed, and there are various others settings and edits required. They are all described in detail in the `README.install` file. Corrections, and suggestions for making the installation instructions clearer for all potential users, would be much appreciated.

### 2.1.2. Compiling Kleene from Source Code

If a pre-compiled binary works for you, just skip this section and proceed to the next. You do not need to, and probably do not want to, compile Kleene from the source code. “Building” Kleene from the source files is

---

1 Sometimes your web browser will `gunzip` the file for you.
currently rather difficult, an exercise for experts, and this is definitely a target for simplification in the future.

If a pre-compiled binary does not work for you, and you don't mind a little challenge, read the files README.git and README.build, which will direct you to download the Kleene source code from a Github repository. Follow the instructions in README.build and then those in README.install. If you succeed in compiling and running Kleene in a new operating system, you are urged to contribute the tarball file for use by others.

### 2.2. Launching the Kleene GUI

Once you have followed all the instructions in README.install, you should be able to cd to the directory where the Kleene.jar file is installed, e.g. `~/kleene/`, and then invoke

```bash
$ java -jar Kleene.jar
```

or one of the aliases described in README.install. This should bring up the Kleene GUI (Graphical User Interface), which looks like this:
The look-and-feel will be a little different on OS X and Linux. The application window, which almost fills the screen, can be minimized using the buttons at the top, and all the windows can be adjusted in size by mouse-clicking on the lower-right corner. Inside the application window, there is a pseudo-terminal window\textsuperscript{2} on the left, and a symbol-table window, initially empty, on the right. At the very bottom of the pseudo-terminal window, there is an active one-line text-entry field where you can enter commands to program (and learn) Kleene interactively.

\textsuperscript{2}It is called a \textit{pseudo} terminal because it allows you to type commands, and view responses, in a terminal-like window, but it lacks a history memory and other useful features of a traditional command-line terminal application. It is hoped that this pseudo-terminal can be improved in the future.
2.3. **OK, What can you do in Kleene?**

2.3.1. **First, is it working?**

Just to make sure that Kleene is working, enter the following statement carefully in the bottom line of the pseudo-terminal window:

```
$fsm = dog | cat | bird ;
```

Be sure to include the semicolon at the end, and press the Enter key on your keyboard to initiate interpretation. If everything is working, you should see an icon named \$fsm appear in the symbol-table window on the right. Your statement has been successfully interpreted to build a finite-state machine. In this case, it is an FSM that encodes the language that contains just three words: “dog,” “cat” and “bird.”

2.3.2. **Visualizing Finite-State Machines**

To view the finite-state machine that you just created, you can right-click on the \$fsm icon and select the drop-down-menu item named draw. Equivalently, you can enter

```
draw $fsm ;
```

manually in the pseudo-terminal. In fact, selecting the draw menu item causes the statement draw $fsm ; to be written in the pseudo-terminal, and then interpreted, exactly as if you had typed it yourself. This behavior is designed to help you learn the scripting language.

If everything is working correctly, you should see the following FSM diagram appear on your screen.
Note that the finite state machine (FSM) has a start state, represented by a bold circle labeled “Start,” a single final state represented with a double circle, and other states represented with plain circles. Where the FSM has \( n \) states, they are numbered in a dense range from 0 to \( n - 1 \). The start state is often, but not always, number 0. This particular FSM has three paths leading from the start state to the final state, each path encoding one word.

Each transition or arc leading from a state to a state has a label like \( d: d \). The first \( d \) is termed the input symbol, and the second the output symbol, in the OpenFst visualization of FSMs. Thus the compound labels are always interpreted as InputLabel:OutputLabel.

In other traditions, FSMs are visualized a bit differently, and the terminologies vary, sometimes confusingly. A label like \( d: d \) could be thought of as LeftSymbol:RightSymbol. In the Two-Level-Morphology tradition, FSMs are usually visualized vertically, with upper-side symbols called lexical symbols, because they matched symbols in a lexicon, and with lower-side symbols called surface symbols, because they matched symbols in surface strings: Lexical:Surface.

The Xerox/PARC tradition also visualizes FSMs vertically but emphasizes the bi-directionality of transducers: either side could be used as the input side, and so the opposite of the side currently being used as the input side is the output side. Abstracting away from how a particular FSM is being used, Xerox researchers often refer to the labels as upper and lower: Upper:Lower.
Kleene is based on the OpenFst library, but with some powerful semantic features borrowed from the Xerox tradition. In this book, it is sometimes important to understand an FSM in OpenFst terms, as having Input:Output labels. At other times, where the bi-directionality of FSMs is relevant, I will talk about Upper:Lower labels.

The “sides” or “levels” of an FSM are technically known as projections, and we will refer to the input and output projections when thinking in OpenFst terms, and we will refer to the upper and lower projections when thinking in Xerox terms. Transducers can theoretically have any number of projections, but most implementations, including OpenFst, are limited to two.

2.3.3. Compile regular expressions into FSMs

Assignment Statements

The example that we used previously for testing

```plaintext
$fsm = dog|cat|bird ;
```

is an assignment statement that has a variable `$fsm` on the left-hand side and a regular expression on the right-hand side, all terminated with a semicolon. Such a statement can continue over multiple lines.

Many programmers will already have experience with programming languages, such as Perl, Python and Java, that support a kind of regular expressions to perform various useful tasks, including pattern matching and tokenization. While these Perl-like regular expressions are often very useful, they are not always regular\(^3\) in the mathematical sense, and so they don’t have all the attractive properties of truly regular expressions.\(^4\)

---

\(^3\)Regular is a technical term, and sometimes the equivalent term rational is also used.

\(^4\)For a thorough presentation of “regular” expressions as they appear in common programming languages and operating-system utilities, see the book *Mastering Regular Expressions* by Friedl (2006).
In contrast to programming-language “regular” expressions, the regular expressions of Kleene are true regular expressions in the sense that they always encode regular languages or regular relations, and they compile into finite-state machines (FSMs). Such regular languages/relations, and the corresponding FSMs, have mathematically clean and powerful closure properties, meaning that they can be combined together using various operations—including concatenation, union and composition—and the results are still regular. These terms will be explained as we progress.

Regular Expressions by Example

We will look in detail at Kleene regular expressions in the next chapter. For now, let’s look at some simple examples to get an intuitive feel for regular expressions and the FSMs corresponding to them. One of the simplest regular expressions consists of just a single letter, e.g. d. Try entering the following in the Kleene GUI.

```
$fsm = d ;
draw $fsm ;
```

The resulting FSM has two states, one the start state and the other a final state, with one arc labeled d: d leading from the start state to the final state.

```
Start -- 0 --> 1
  d:d
```

Alphabet: d

The regular expression d describes the regular language consisting of just the one string “d.” The FSM encodes this language, which means that if you apply it to the string “d,” it will match and accept it, and it will reject all other strings, including “a,” “z,” “dog,” ”mmmmmm,” “hwiughuiegw,”
etc. An FSM can thus be used as an acceptor that accepts all and only the strings in the language that it encodes.

This same FSM can be viewed as an identity transducer that maps the string “d” to “d,” i.e. maps “d” to itself. In OpenFst, the labels on the arcs are always two-level, input:output, or, in the Xerox visualization, upper:lower. An FSM like that shown above is interpreted either as an acceptor or as an identity transducer, depending on what is needed by an operation.

In the Kleene GUI, you can actually test an FSM, to see what it accepts and rejects, by entering

test $fsm ;

This will bring up a test window into which you can type an input string.

The FST being tested is represented by the bar labeled “FST” in the center. The test window has two string-input fields, one at the top and one at the bottom, both labeled “String>>.” For testing an acceptor, it doesn’t matter which field you use. If you enter d in the top field, and then press the Enter key, the answer
The Kleene Language

d: 0.0

will appear in the history part of the pseudo-terminal window. This indicates that the input string was accepted by the FSM. (Don’t worry now about the “0:0,” which is a weight and will be explained later.) You get the same result if you enter d in the lower-side input window. But if you enter any other string, such as “b” or “dd” or “ant” or “elephant,” a message indicates that the output is empty, i.e. that the input string was not accepted.

Now let’s look at a slightly more complicated regular expression

\$fsm = \text{dog} \ ; \\
\text{draw } \$fsm \ ;

that involves the concatenation of three characters: d followed by o followed by g. In Kleene regular expressions, there is no explicit operator for concatenation; you just type one symbol after another—or more generally, one operand after another—and they will be concatenated. The argument to the draw command can be an arbitrarily complex regular expression, so one could alternatively just enter

\text{draw dog} ;

The FSM that encodes this language, consisting of the one string “dog,” has four states and three arcs.

\[
\begin{array}{c}
\text{Start} \\
0 \quad \text{d:d} \rightarrow 1 \quad \text{o:o} \rightarrow 2 \quad \text{g:g} \rightarrow 3
\end{array}
\]

Alphabet: g, d, o

The testing procedure that applies this FSM to the string “dog,” involves first setting a match pointer to the first character d in the input string, initializing the machine in its start state, and then looking for an arc labeled
d leading out of the start state. There is such an arc in this example, so the application consumes the input symbol d, resets the match pointer to the next symbol o, and resets the machine to state 1, which is the destination state of the arc labeled d. The current state of the machine is now state 1. From the current state, the process continues, matching and consuming the o, and putting the machine in state two; then matching and consuming the g, leaving the machine in state 3. Because state 3 is a final state (indicated by the double circle), and because the input string is fully consumed (no input symbols are left over), the application is a success, and the finite-state machine matches and accepts the input string “dog.” If this machine were applied to any other input string, it would fail to match, and the string would be rejected.

Note that a finite-state machine (FSM) has a finite number of states—though there might be hundreds of thousands or even millions of them—and there is no other memory mechanism, such as a stack, used during application. At any given time during application, a finite-state machine is in exactly one of its finite number of states. And when a machine is in a particular state, the only thing that affects its progression to the next state is the next input symbol; it has no stack or other memory to influence where it goes next.

Now let’s try a slightly more complicated regular expression, involving both concatenation and union, for which the operator is the vertical bar |.

$fsm = \text{dog} | \text{cat} | \text{bird} | \text{horse} ;$

draw $fsm ;$

For improved readability, you can insert spaces and newlines as desired, e.g.

$fsm = \text{dog} \mid \text{cat} \mid \text{bird} \mid \text{horse} ;$

$\mid \text{cat} \mid \text{bird}$
The resulting FSM encodes the language of four words—"dog," "cat," "bird" and "horse"—and clearly has four paths leading from the start state to the final state.

If you test this FSM,

test $fsm ;

you will find that it accept the four strings, and no others.

When the FSM is an acceptor, the print command will list the strings with their weights. If you don't want to see the weight, use the pr command.

print $fsm ;
dog : 0.0
cat : 0.0
bird : 0.0
horse : 0.0

print Hello ;
Hello : 0.0
pr apple | orange | banana | fig | cherry;
apple
orange
banana
fig
cherry

Note that a language is a set of strings, and in a set, the order of its elements undefined. FSMs may be constructed, and words encoded by the FSM may be printed, in orders that you didn’t expect.

In general, spaces and other whitespace characters are ignored in Kleene regular expressions unless they are explicitly literalized. One way to literalize spaces, and other special characters, is to precede them with the backslash character \.

// This is a comment:
// Printing a string with a literal space
pr Hello\ world;
// outputs: Hello world (as a single string)

Another equivalent way to literalize a space is to put it inside double quotes:

pr Hello " " world;
// or
pr "Hello world";
draw "Hello world";

Kleene regular expressions, like Perl regular expressions, can also use parentheses for grouping. A postfixed asterisk * meaning zero or more, a postfixed plus-sign +, meaning one or more, and a postfixed question mark ? indicating that the preceding expression is optional, are also as in Perl regular expressions. The following example encodes the words “dog,” “cat” and “horse,” each with an optional s on the end, for a total of six strings.
$\textsf{fsm} = ( \text{ dog } | \text{ cat } | \text{ horse } ) s? ; \\
\text{pr } \textsf{fsm} ; \\
// \text{ outputs: dog dogs cat cats horse horses}

And this next example encodes the language of all strings that start with zero or more a letters, followed by one or more b letters, followed by a c, d, e or f, and ending with an optional g.

$\textsf{fsm} = a^*b+(c|d|e|f)g? ;$

This language described by this expression is infinite in size but still regular, and it is encoded in a compact FSM.

![Alphabet: f, g, d, e, b, c, a
Start 0
a:a
1b:b
b:b
2
c:c
d:d
e:e
f:f
3g:g]

When it becomes tedious to type long unions of symbols, e.g. (a|b|c|d|...|x|y|z) one can use the Perl-like square-bracketed notation for character unions: for example, [a-z] encodes the union of all characters starting at a and ending with z. Similarly, [aeiou] denotes the union of a, e, i, o and u.

// Three equivalent regular expressions
$\textsf{fsm1} = a^*b+(c|d|e|f)g? ; \\
$\textsf{fsm2} = a^*b+[cdef]g? ; \\
$\textsf{fsm3} = a^*b+[c-f]g? ;$

The following example encodes the language of all strings that start with an English alphabet letter, uppercase or lowercase, and then continues with zero or more alphabetic letters or digits:

$\textsf{fsm} = [A-Za-z][A-Za-z0-9]* ;$
The resulting FSM will accept strings like “Apple,” “a27,” “dOg,” “z3b4n6” and “d2T456m7” while rejecting strings such as “4abc” and “26.”

Regular languages can be subtracted from each other:

\[
\text{lang} \equiv (\text{dog} \mid \text{cat} \mid \text{elephant} \mid \text{apple} \mid \text{orange}) - \\
(\text{banana} \mid \text{apple} \mid \text{orange})
\]

pr \ $\text{lang}$ ;
// outputs: dog cat elephant
assert #^equivalent($\text{lang}$, dog \mid cat \mid elephant) ;

Here, the resulting FSM encodes the language consisting of the strings “dog,” “cat” and “elephant.” It is not necessary to put spaces around the vertical-bar operator |. As already stated, white space is ignored in Kleene regular expressions unless it is explicitly literalized.

The assert command, in this case, tests the equivalence of the FSM $\text{lang}$ and the expected result. The assert command and the #^equivalent() function will be explained below in more detail. For now, it is sufficient to understand that the assert command will throw an exception if the boolean condition is false, and will do nothing if the boolean condition is true.

Regular languages can also be intersected, resulting in a new language containing all and only the strings they have in common. The operator for intersection is &.

\[
\text{commonAnimals} = \\
(\text{dog} \mid \text{cat} \mid \text{elephant} \mid \text{whale} \mid \text{ant} \mid \text{bird} \mid \text{reindeer}) \& \\
(\text{cat} \mid \text{horse} \mid \text{reindeer} \mid \text{snail} \mid \text{whale})
\]

pr \ $\text{animals}$ ;
// outputs: cat whale reindeer
assert #^equivalent($\text{animals}$, cat \mid whale \mid reindeer) ;

The . (dot) represents any symbol, so the FSM resulting from the assignment statement
\$fsm = p . t ;

accepts words including “pat,” “pet,” “pit,” “pot”, “put,” “pbt,” “ppt,” etc. And because . really represents any symbol, the expression . * represents the Universal Language, the language that contains all possible strings of any character, of any length, including the empty (zero-length) string.

\$UnivLang = .* ;

Note that in Kleene, as in the Xerox Finite State Toolkit, is it not necessary for the programmer to declare the alphabet being used. In Kleene regular expressions, the . really represents any possible symbol.

The Empty Language is the language that contains no strings at all, not even the empty string. There are an infinite number of ways to denote the Empty Language, including

\$EmptyLang = .* - .* ; // Universal Language minus itself
\$EmptyLang = dog - dog ; // one-string language minus itself

and

\$EmptyLang = ~.* ;

where ~ is the complement operator, returning the language of all possible strings, i.e. the Universal Language, except (i.e. minus) the strings in the language it applies to. Thus ~ . * is equivalent to . * - . *, the Universal Language minus itself.

The empty string is the string of zero length, containing no symbols at all. It can be notated in Kleene as "". The Empty String Language, not to be confused with the Empty Language, is the language that contains exactly one string, the empty string.

\$EmptyStringLang = "" ;
2.3.4. Optimization of FSMs

When FSMs are built in Kleene, they are (where possible) automatically optimized, which involves a combination of determinization, minimization and epsilon-removal. The epsilon, which can be denoted as "" in regular expressions, is the empty (zero-length) string, and epsilon-removal involves the elimination of all arcs that have [eps]:[eps] labels. Thus the assignment

\$fsm = a "" c ;

creates a network that initially looks like

![Initial Network Diagram]

including the epsilon indicated by the regular expression, plus two other epsilons introduced by the concatenation algorithm. By default, however, the FSM is automatically epsilon-removed to produce the equivalent but more compact result

![Compact Network Diagram]

Consider also the following example, which has multiple words that start with the same symbol(s) and multiple words that end with the same symbol(s).

\$foo = rat | bat | rabbit | bird | cod ;

The raw, unoptimized FSM looks like this, including a number of epsilon arcs:
This machine accurately encodes the five-word language, but in a less than optimal way. Just by running epsilon-removal, the FSM is considerably simplified, resulting in the following machine:

At this point, note that if you apply the machine to the input string “rat,” its operation is going to be non-deterministic. Starting in the machine’s start state, and placing the match pointer at the first input symbol r, there are in fact two arcs labeled r exiting the start state, and both of these arcs will need to be explored. Of course, only one of them will result in a successful match, but time will be wasted exploring the wrong r-labeled arc.

The solution to the non-determinism problem is to determinize the FSM, resulting in the following modified machine
Note that the paths for the words “rat” and “rabbit,” which both start with the prefix “ra,” now share the states and arcs for recognizing that prefix. When the determinized machine is applied to “rat” or “rabbit,” the application algorithm no longer wastes time exploring the wrong path. Note also that the b prefix shared by “bat” and “bird” is also represented in the machine by shared states and arcs.

At this point, we have a deterministic machine that can be applied to input strings with maximum efficiency, but note that the machine itself is bigger than it really needs to be. In particular, note that “rat,” “bat” and “rabbit” all end with the same t suffix, but the FSM represents them with separate states and arcs. The solution to this problem is to minimize the machine, resulting in this final optimal machine:
As you will recall, we started with the superficially simple regular expression

$foo = rat| bat| rabbit| bird| cod ;$

At each stage shown, the FSM encoded the same language of five words, but as we epsilon-removed, determinized and minimized the machine, it became both more efficient to apply, and smaller in its storage requirements. In real-life applications, where FSMs can encode languages of millions of words, and the machines can easily contain tens of thousands of states and arcs, and runtime performance is critical, the processes of epsilon-removal, determinization and minimization—known collectively as optimization—are vital. Luckily, Kleene automatically optimizes all FSMs by default, and the average Kleene programmer never has to worry about it.\(^5\)

The optimization—especially the determinization—of weighted FSMs and of FSMs with cycles is a challenge still under study.

**2.3.5. Using Defined Variables**

Once a variable has been bound to an FSM value, it can be used in subsequent regular expressions, e.g.

---

\(^5\)Expert users may occasionally want to turn off one or more of the optimization processes, and Kleene provides a way to do this. See Appendix B.
$\text{fsm1} = \text{dog} \mid \text{cat} \mid \text{elephant} ;$

$\text{fsm2} = \text{horse} \mid \text{pig} \mid \text{sheep} ;$

$\text{fsm} = \text{fsm1} \mid \text{fsm2} ;$

pr $\text{fsm} ;$

// outputs: dog cat elephant horse pig sheep

2.3.6. Characters

Unicode

Kleene supports the Unicode character set, which currently contains over 100,000 defined characters and has a capacity to encode over 1 million characters. Programmers are encouraged to edit their Kleene scripts directly in Unicode using their favorite Unicode-capable text editors, though Kleene can read and process scripts written in all common character sets.

The Kleene GUI is written using the Java Swing library; its text widgets are automatically Unicode-capable, and you can use standard Java input methods to facilitate typing in exotic characters. In some cases, it may be convenient to designate Unicode characters by their code point value. Unicode characters in the Basic Multilingual Plane can all be encoded using four hexadecimal digits, and in Kleene regular expressions they can be designated as the ASCII sequence \uHHHH, where H is in the set [0-9a-fA-F].

The following two examples are equivalent,

$\text{fsm1} = a \ b \ c \ \alpha \ \beta \ \gamma ;$

$\text{fsm2} = a \ b \ c \ \u03b1 \ \u03b2 \ \u03b3 ;$

because 03b1 (hexadecimal) is the code point value of the Greek alpha, 03b2 is the beta, and 03b3 is the gamma. Both expressions result in an FSM that looks like

---

\(^6\text{That is, each H can be a digit from 0 to 9 or a letter from A to F, in either uppercase or lowercase.}\)
In OpenFst (and Kleene) FSMs, the labels on arcs are in fact integers, and in Kleene these integers are Unicode code point values. Where possible, the code point values are displayed by the `draw` facility as letters. The Graphviz `dot` facility, which is used to display the FSMs, has limitations in displaying characters like α, β and γ, so they are displayed in hexadecimal.

Kleene can also handle Unicode supplementary characters, and they can be entered as the ASCII sequence \UHHHHHHHH, that is, \U followed by exactly eight hexadecimal digits. The following example contains the first three characters in the Deseret Alphabet, which is encoded in the supplementary area.

```plaintext
$fsm = \U00010400 \U00010401 \U00010402 ;
draw $fsm ;
```

This results in the following FSM

```plaintext
Alphabet: \u10401, \u10400, \u10402
```

which has a single supplementary code point value on each arc.\(^7\)

Kleene makes a fundamental distinction between symbols, such as alphabetic characters, and numbers. Symbols appear in regular expressions, while numbers appear in arithmetic expressions (see chapter 8). Although alphabetic symbols are stored in FSMs using their Unicode Code Point value, which is an integer, you should not confuse alphabetic symbols

\(^7\)However, if the characters are entered in a Swing text widget using the Java Code-Point Input Method, each character is somehow divided into the two BMP surrogate characters used to encode the single supplementary character in the UTF-16 encoding. This behavior needs to be reviewed in the context of Kleene.
and numbers in Kleene expressions. For example, the Kleene expression \\u0065 denotes the symbol e, which has the Unicode code point value 0x65 (i.e. hexadecimal 65). You can include \\u0065 in a regular expression just like any letter

```cpp
$fsm1 = b e t ; 
$fsm2 = b \u0065 t ; // equivalent
assert(#^ equivalent($fsm1, $fsm2) ;
```

But in these regular expressions you cannot replace \u0065 with the integer expression 0x65 or its decimal equivalent 101. The following two expressions are illegal and cause exceptions to be thrown.

```cpp
// illegal regular expressions
$fsm = b 0x65 t ; // ILLEGAL
$fsm = b 101 t ; // ILLEGAL
```

Multi-character Symbols

In addition to Unicode characters, it is often useful to have user-defined symbols that have multi-character names, often called “multi-character symbols.” For example, you can define [Noun], [Verb], [Adj], [Adv], [Sing], [Plur], etc. as symbols that suggest linguistic categories or features. In another notational tradition, these might be spelled +Noun, +Verb, +Adj, etc. In Kleene syntax, you simply surround a string of characters in single quotes to indicate that they are to be treated as a single multi-character symbol.

```cpp
$fsm = a b '[Noun]' ;
```

The single quotes delimit the name of the symbol but are not actually part of the symbol name. Each multi-character symbol is also stored as an integer label on an arc in the result FSM, and the integer is taken from a Unicode Private Use Area (PUA).
In general it is recommended that multi-character symbol names include at least one punctuation character to help human beings distinguish them from sequences of alphabetical characters. There is nothing to prevent you from using a multi-character symbol like noun, by putting 'noun' in a regular expression,

```plaintext
// a multi-character symbol without punctuation,
// NOT recommended
$myfsm = 'noun' ;
print $myfsm ;
// outputs: noun

// a concatenation of four symbols
$yourfsm = noun ;
// outputs: noun
```

but the printed output noun, being a single multi-character symbol, is humanly indistinguishable from the printed word “noun” that is a concatenation of four separate symbols. In practice, the use of multi-characters symbols like noun leads to much confusion and is strongly discouraged.

**Characters in Variable Names**

FSM variable names in Kleene start with a dollar sign, a letter, and then any number of letters and digits.

```plaintext
$fsm = a b c ;
$fsm1 = x y z ;
$q7 = r s t ;
```

The letters after the dollar sign can include any letters in the Unicode BMP (Basic Multilingual Plane). The following two assignments are equivalent

---

8Because of limitations in JavaCC and Java itself, variable names cannot contain supplementary characters.
Kleene supports Unicode to the extent that Java does, which is pretty well but not perfectly. Users can write their scripts using the International Phonetic Alphabet, Greek, Cyrillic, Arabic, etc. without ever needing to resort to clumsy transliterations. Whether the Kleene GUI can actually display a character depends on the fonts installed in your Java installation.

2.3.7. Regular Languages and Regular Relations

Regular Languages and Acceptors

In formal language theory, a language is a set of strings. A regular language, where regular is a technical term,\(^9\) is formally defined as a language that can be defined using only concatenation, union and the Kleene-closure\(^10\) operator *, which indicates zero or more repetitions. Any finite (i.e. not infinite) language is a regular language. Some infinite languages, such as b*, which is the language of all strings that contain zero or more b letters, are also regular. Some infinite languages are not regular and so cannot be encoded in an FSM; we cannot handle such languages in Kleene.

A regular language can be encoded as an FSM (Finite-State Machine) called an acceptor. An FSM has a finite number of states, one of which is designated as the start state, and zero or more of which are final. An FSM can also contain zero or more labeled directed arcs that lead from a state to a state.

When an acceptor is applied to an input string, it will either accept it or reject it. It will accept all and only the strings in the regular language that it encodes.

In classic visualizations of acceptors, they have only a single label on each arc. In OpenFst, and therefore in Kleene, all arc labels have two

---

\(^9\)In some traditions, the term rational is used instead of regular.

\(^10\)The Kleene-closure is named after mathematician Steven Cole Kleene, who invented it. The Kleene programming language is named after the same man.
labels, an input label and an output label: $i: o$. By convention in OpenFst, an FSM is an acceptor, or can be interpreted as an acceptor, if and only if each input label is equal to its associated output label.

Regular languages, and the acceptors that encode them, have some very interesting and useful mathematical properties, known as closure properties. For example, regular languages are closed under the union operation because if you union any two regular languages together, the result is also a regular language. Regular languages and the acceptors that encode them are closed under the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>$A \cdot B$</td>
</tr>
<tr>
<td>Union</td>
<td>$A \mid B$</td>
</tr>
<tr>
<td>Iteration</td>
<td>$A^*$</td>
</tr>
<tr>
<td>Subtraction</td>
<td>$A - B$</td>
</tr>
<tr>
<td>Intersection</td>
<td>$A &amp; B$</td>
</tr>
<tr>
<td>Complementation</td>
<td>$\overline{A}$</td>
</tr>
</tbody>
</table>

**Regular Relations and Transducers**

Whereas a regular language is a set of strings, a regular relation is a set of ordered pairs of strings. For example, (dog, dogs), (cat, cats), (elephant, elephants), (woman, women) and (deer, deer) are ordered pairs of words that, in English, have a singular noun as the first word, and its plural as the second word. A set of such pairs, denoted in set theory (not Kleene syntax) as \{(dog, dogs), (cat, cats), (elephant, elephants), (woman, women), (deer, deer)\}, is a regular relation. All finite relations are regular relations. Some infinite relations are also regular.

A regular relation can be encoded as an FSM called a transducer. Theoretically, regular relations and finite-state transducers can have any number of levels, known as projections, but OpenFst and most practical computer implementations have been limited to two. A finite-state transducer (FST) has a finite number of states, one of which is designated as the start state, and zero or more of which are final. An FST can also contain zero
of more labeled directed arcs that lead from a state to a state. Each label consists of two parts, an input label and an output label, notated $i: o$. In a finite-state transducer, the input and output labels of an arc can be different, or the same.

In Kleene syntax, a label with $a$ on the input side and $b$ on the output side is denoted $a: b$. The statement

$\text{fsm} = a: b ;$

results in the FST

![Finite State Transducer Diagram]

Alphabet: b, a

More generally, $:$ is the cross-product operator that takes two operands that must both denote regular languages (not regular relations). It creates a regular relation with the left operand as the input projection, and the right operand as the output projection. Often the cross-product operator is used to map individual strings, as in this Kleene example.

$\text{fsm} = (\text{dog}):(\text{dogs}) \mid (\text{cat}):(\text{cats}) \mid (\text{elephant}):(\text{elephants}) \mid (\text{woman}):(\text{women}) \mid (\text{deer}):(\text{deer}) \mid (\text{formula}):(\text{formulas}) \mid (\text{formula}):(\text{formulae}) ;$

More generally, the operands of the cross-product operator $:$ can denote arbitrarily complex regular languages, and every string in the input language is related to every string of the output language, and vice-versa. Consider, for example,

$\text{foo} = (\text{dog} \mid \text{cat} \mid \text{rat}):(\text{animal} \mid \text{mammal}) ;$
where the input (upper-side) language consists of the strings “dog,” “cat” and “rat,” and the output (lower-side) language consists of the strings “animal” and “mammal.” The cross-product is an FST that relates “dog” in the input language to both “animal” and ”mammal” in the output language. And similarly for “cat” and “rat.” The resulting relation contains the six ordered pairs (dog, animal), (dog, mammal), (cat, animal), (cat, mammal), (rat, animal) and (rat, mammal). Note that a relation is a set of ordered string pairs, and the order of the string pairs is not significant.

Note that the cross-product operator : has higher precedence than concatenation, so the example

```latex
\$\text{foo} = abcd: ef ;
```

is equivalent to

```latex
\$\text{foo} = abc(d: e)f ;
```

Parentheses can be used to force whatever groupings are desired.

**Applying Transducers**

In the OpenFst visualization, an FST is applied to an input string by matching the symbols of the input string against a sequence of input symbols. If the input string matches successfully and completely against a sequence of input symbols, on a path leading from the start state, arc-by-arc to a final state, the output is a string consisting of the sequence of output symbols from the same path in the FST.

The output string can be different from the input string, and thus the FST can transduce or map from one kind of string to another kind of string. To return to the previous noun example, if an ordered pair (dog, dogs) is interpreted as (inputString, outputString), then the FST that encodes the regular relation \{(dog, dogs), (cat, cats), (elephant, elephants), (woman, women), (deer, deer)\} would provide a transduction or mapping from singular nouns like “dog,” “cat” and “woman” to their related plural forms.
“dogs,” “cats” and “women,” respectively. Relations relate words (strings of symbols) to other words (strings of symbols). Looked at in the other direction, the same FST provides a mapping from plural forms to their related singular forms.

Try building and testing the following example in the Kleene GUI:

```plaintext
$numberMapper = (dog):(dogs) | (cat):(cats) |
              (elephant):(elephants) | (woman):(women) | (deer):(deer) |
              (formula):(formulas) | (formula):(formulae) |
              (cherub):(cherubim) | (man):(men) | (knife):(knives) ;
test $numberMapper ;
```

Enter a singular noun like “dog” in the upper-side input field, and the output will be the plural form “dogs.” Enter “formula” in the upper-side input field, and the output will be the two plural forms: “formulas” and “formulae.”

An FSM encoding an acceptor can also be interpreted and used as an identity transducer that maps each of the words in the language to itself.

An FST that encodes a relation is functional if each input string has exactly one output string. Very commonly when modeling natural languages, an FST can map one input string to multiple output strings, reflecting ambiguity.

A regular relation is a relation between two regular languages, one of which is, in the OpenFst tradition, termed the input projection and the other the output projection. In all cases, every string in the input language/projection is related to at least one string in the output language/projection, and each string in the output language/projection is related to at least one string in the input language/projection.

In the Xerox visualization of FSTs, which emphasizes the bi-directionality of FSTs, the projections are called upper and lower rather than input and output. What OpenFst calls normal application, matching the symbols of an input string against “input” symbols on a path, outputting the corresponding “output” symbols, Xerox calls generation or “application in a downward
direction,” matching the symbols of the input string against upper-side labels, and outputting strings of lower-side labels. Xerox researchers also talk about analysis or “application in an upward direction,” matching the symbols of the input string against lower-side symbols, and outputting strings of upper-side symbols. The test facility in the Kleene GUI reflects the Xerox/PARC visualization of bidirectional FSTs.

The following transducer, which includes multi-character symbols, models the formation of regular English plurals more elegantly, and reflects the Xerox visualization.

```plaintext
// a set of noun roots
$nroot = dog|cat|bird|horse|worm ;
$ncat = '[Noun]':"" ; // [Noun] tag on the upper side,
     // the empty string on the lower side
$num = '[Sg]':"" | '[Pl]':s ; // [Sg] on the upper side,
    // and empty string on the lower.
    // [Pl] on the upper side,
    // and s on the lower.

// now just concatenate the three FSMs
$nouns = $nroot $ncat $num ;

test $nouns ; // and launch the test window
```

The resulting FST looks like this:

Alphabet: g, d, e, b, c, a, o, m, h, i, w, t, s, r, [Pl], [Noun], [Sg]
showing that strings like “dog[Noun][Sg]” and “cat[Noun][Pl]” are in the upper-side language, while their related strings like “dog” and “cats” are in the lower-side language. The [eps] character in the diagram represents the empty-string language, called the epsilon by convention. When you test the network and enter cat[Noun][Pl] in the upper-side entry field, the output is

\texttt{cats: 0.0}

Again, just ignore the weight 0.0, which will be explained later. If you enter horse[Noun][Sg] in the upper entry field, the output will be horse. Xerox researchers working on morphology generally referred to this kind of application as generation, taking an abstract upper-side string and generating a surface lower-side string (or strings). Now, applying the same FST in the opposite direction, try entering horse in the lower input field; the result will be horse[Noun][Sg]. And if you enter horses in the lower input field, the output will be horse[Noun][Pl]. Xerox researchers referred to this mode of application as analysis, mapping from surface strings to abstract analysis strings. FSTs are bidirectional, and the Xerox visualization of FSTs emphasizes that bidirectionality. Such FSTs, which perform morphological analysis and generation, can be written to model word analysis in natural languages like English, French, German, Arabic, Zulu, etc.

As we shall see, there is value to both the Xerox and the OpenFst visualizations of FSTs. Although, as the Xerox tradition emphasizes, an FST can always be applied validly in either direction, downward or upward, a determinization algorithm cannot generally determinize both sides of an FST—it has to determinize one side or the other.\footnote{In some traditions, the term \textit{determinization} is used for acceptors, and the determinization of one side of a transducer is called \textit{sequentialization}.} The OpenFst determinization algorithm optimizes what the OpenFst tradition calls the input side, and because the input side is determinized, and the output side is not necessarily determinized at the same time, it is generally more efficient to match input strings against the input side than against the output side.
In Kleene, one is free to build FSTs in any orientation that the programmer finds intuitive. But one should be aware that it is always the upper side—what OpenFst calls the input side—that is determinized for the most efficient runtime application. If you intend to apply your FST in a downward direction, matching input against the upper side, and reading the results off the lower side, then it is already in the proper orientation for the most efficient operation.

If, however, you build an FST that is properly applied in an upward direction, matching the input against the lower side, to get the results you need, all you have to do, as a final step, is to invert the FST. Inversion exchanges the upper side and the lower side, such that the new upper side (the old lower side) will be determinized for the most efficient application. Then use the inverted FST in the OpenFst way, matching input strings against the new “input” side.

In Kleene, some operations like inversion are implemented as pre-defined functions rather than using operators. Here is how you take a defined FST and compute its inversion.

\[ \text{invertedFst} = \text{invert}(\text{fst}) ; \]

That is, Kleene provides a function named invert that takes one argument, an FST, and returns a new FST that is the inversion of the argument. The argument itself is not modified.

**Closure Properties of Regular Relations/Transducers**

The closure properties of regular relations, and of the FSTs that encode them, are not the same as the closure properties of regular languages. In particular, FSTs are closed under concatenation, union and iteration, but not generally under subtraction, intersection or complementation.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>A B</td>
<td></td>
</tr>
<tr>
<td>Union</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Iteration</td>
<td>A*</td>
<td></td>
</tr>
</tbody>
</table>
If you try to subtract, intersect or complement transducers, Kleene will throw an exception and you will get an error message. The illegal operations for transducers all reduce to subtraction.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>A - B</td>
<td>A - B</td>
</tr>
<tr>
<td>A &amp; B</td>
<td>A - (B - A)</td>
</tr>
<tr>
<td>~A</td>
<td>.* - A</td>
</tr>
</tbody>
</table>

**Closure Properties of Weighted FSMs**

Weights in FSMs complicate the closure properties, and each operation may have its own restrictions on which of its operands can be weighted. Whenever the Kleene interpreter tries to combine two FSM operands, it first consults a LibraryChecker class that “knows” the mathematical requirements and the limitations of the algorithms supplied in the OpenFst library. If any attempted operation is illegal, the interpreter will throw an exception and print out a (hopefully) useful message.

**Playing with FSTs**

A regular relation, encoded as an FST, is a relation between two regular languages, called the input projection and the output projection, and we can extract those projections using the pre-defined Kleene functions $^\text{inputProj()}$ and $^\text{outputProj()}$.

```
$fst = (dog|cat|rat):(animal|mammal) ;
$inputLang = $^\text{inputProj}($fst) ;
pr $inputLang ;
// outputs: dog cat rat

$outputLang = $^\text{outputProj}($fst) ;
pr $outputLang ;
// outputs: animal mammal
```
We can compute the inversion of an FST with the \$invert() function, and the reverse of an FSM (which reverses all the paths right-to-left) with the \$reverse() function.

\$fstinv = \$invert($fst) ;
\$fstrev = \$reverse($fst) ;

There are many more pre-defined functions in Kleene, and it is also possible to define your own functions.

Although you cannot, as we have already stated, subtract, intersect or complement transducers, you can combine them using composition, a powerful operation that we will be exploring throughout the book. In particular, composition is central to the application of alternation rules, of which Kleene offers a rich variety. As a simple example, consider the rule

\$rule = s -> z / (a|e|i|o|u) _ (a|e|i|o|u) ;

or the equivalent

\$rule = s -> z / [aeiou] _ [aeiou] ;

which states that an s on the input side is mapped to a z on the output side when it appears between two vowels. The rule compiles into a transducer, which you can draw and test in the usual ways:

draw $rule ;
test $rule ;

When you test $rule and enter “casa” in the upper field, the output will be “caza.” If you input “susisesos,” the output will be “suzizezos,” mapping each input s into z only when it appears between vowels.
2.4. Looking at FSMs

As you build FSMs, you will often want to see what they contain and what they look like. For finite (very finite) acceptors, you can print out the words in the regular language.

$acc = (dog|cat|rat) s? ;
print $acc ;
pr $acc ;

But printing out an infinite language is obviously out of the question, and Kleene will print a warning if you try. Even if a language is infinite, the FSM encoding it can still be small, as in

$infLang = a^* b^* ;
draw $infLang ;

and it can still be drawn.

In any non-trivial project, however, the FSMs can soon become huge, consisting of hundreds of thousands of states and arcs, and drawing is obviously out of the question. In such cases, one can still need to ask “What do the input (or output) strings look like?”, and Kleene offers some useful commands to show you random samples of the languages. In particular, the randInput command will print out a random list of the input (upper-side) strings:

randInput $fsm ;

Currently, the commands randUpper, rinput and rupper are synonyms of randInput.

Similarly, the randOutput will print out a random list of the output (lower-side) strings:

randOutput $fsm ;

Currently, the commands randLower, routput and rlower are synonyms of randOutput.
2.5. Applications

While it may be hard to imagine now, FSMs can be used to implement a number of useful applications for natural-language processing, including

- Tokenizers
- Spelling checkers
- Spelling correctors
- Phonological models
- Morphological analyzer/generators
- Taggers (part-of-speech disambiguators)
- Shallow parsers
- Speech generator/recognizers

We have seen that FSTs can map an input string to a very different output string. If the input string is a normal English sentence, then the output string might be the same string with token delimiters added—that would be a tokenizer. We have seen that an acceptor will accept all and only the strings in the language that it encodes. If we then had a large acceptor that encoded millions of strings that looked like properly spelled Spanish words, then it would be the basis for a Spanish spelling checker, accepting Spanish words and rejecting everything else. Properly constructed FSTs could also map misspelled words into properly spelled words, orthographical words into strings of International Phonetic Alphabet symbols representing their pronunciation, perhaps to be fed to a speech synthesizer, and morphological analyzer FSTs would map orthographical words into analysis strings showing the baseform, part of speech and other relevant information. Such morphological analyzers could be used to aid in dictionary lookup and to provide output to syntactic parsers.
Grammars consisting of multiple sub-components, performing tokenization, morphological analysis, disambiguation and shallow-robust parsing can be written as FSTs and then composed together into a single transducer that performs all the operations cleanly and efficiently. Speech synthesis and speech recognition are often approached this way.

This book will focus mostly on morphological analysis, which is often the first step in computational linguistics. After traditional fieldwork has defined the phonology, morphology and orthography of a language, and after reasonable lexicography has been done, a finite-state implementation like Kleene can help the linguist take that information, often paper-bound, and animate it on computers, creating applications that analyze and look up words, providing a foundation for language documentation, teaching, promotion and further computational linguistics. The resulting morphological analyzer can easily and repeated be tested, using millions of words, to test its coverage and accuracy. The writing of a finite-state morphological analyzer for a new language is a suitable and popular project for a Master’s thesis.
Chapter 3

Regular Expression Syntax

3.1. Regular Expressions

In Kleene, regular expressions are the primary way to specify finite-state machines. The basic Kleene assignment statements have an FSM variable on the left-hand side and a regular expression on the right-hand side, e.g.

$\text{var} = d ;$
$\text{var2} = \text{dog} ;$
$\text{myvar} = (\text{dog}|\text{cat}|\text{horse}) s? ;$
$\text{your var} = [A-Za-z] [A-Za-z0-9]* ;$
$\text{hisvar} = ([A-Za-z]-[aeiouAEIOU])^+ ;$
$\text{her var} = (\text{bird}|\text{cow}|\text{elephant}|\text{pig}) \& (\text{pig}|\text{ant}|\text{bird}) ;$
$\text{our var} = (\text{dog}):\text{chien} \circ (\text{chien}):\text{Hund} ;$

These regular expressions should already be reasonably familiar to those with experience in mathematical or programming-language regular expressions. Don’t worry if they are not yet familiar to you; this chapter will present and explain each of the regular-expression operators available in Kleene.
3.1.1. Primary Regular Expressions

Primary regular expressions are recognized directly by the tokenizer and so are effectively of highest precedence.

| a b c | simple alphabetic symbols |
| \uHHHH | BMP symbol specified by code point value |
| \uHHHHHHH | supplementary symbol specified by code point value |
| . | (dot) matches any symbol |
| \* \+ \? \. \' | literalized special characters |
| \n \r \t \b \f | conventional control characters |
| '[Noun]' '+Noun' | single symbols with multi-character names |
| $myvar $foo | names of variables denoting a finite-state machine |

Symbols with multi-character names (also known as “multi-character symbols”) can contain any Unicode character from the BMP\(^1\) (Basic Multilingual Plane) except for newline and carriage return.\(^2\) The single quotes are delimiters of the multi-character name in Kleene syntax and are not part of the name. If a multi-character symbol name contains a straight single quote (‘), it must be literalized in the syntax with a preceding backslash, e.g. ‘[o’clock]’.

Symbols with multi-character names are most often used as tags, e.g. ‘[Noun]’, ‘[Verb]’, ‘[Adj]’, ‘[Sg]’, ‘[PI]’, ‘[1P]’, ‘[2P]’, ‘[3P]’, ‘[Masc]’ and ‘[Fem]’, that convey categorial, featural or other grammatical information to a human reader. In general, any sequence of BMP characters can be delimited in single quotes and used as a single multi-character symbol. However, it is highly recommended that the names contain punctuation symbols; that is, it is almost always a mistake to define multi-character symbols with plain alphabetic names like 'Noun', 'Verb', 'sh' or 'ing' that, minus the single quotes, could be visually confused

---

\(^1\) [https://en.wikipedia.org/wiki/Plane_(Unicode)]

\(^2\) The newline or line feed character has the code point value \u000A; the carriage return is \u000D.
with a simple sequence of separate alphabetic symbols.  
Multi-character names starting with __ (two underscores) or ** (two asterisks) are special and are reserved for internal system use. Any attempt to define such a multi-character symbol directly, e.g. '__foo', in your code will cause an exception to be thrown.

3.1.2. Inherently Delimited Regular Expressions

The following regular expressions are syntactically complex but inherently delimited, making them also of highest precedence.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[aeiou] [a-z] [A-Za-z0-9]</td>
<td>character sets (unions)</td>
</tr>
<tr>
<td>[^aeiou] [^a-z] [^A-Za-z0-9]</td>
<td>complemented character sets</td>
</tr>
<tr>
<td>&quot;dog&quot; &quot;+&quot; &quot;AT&amp;T&quot;</td>
<td>double-quoted literalized concatenations of symbols</td>
</tr>
<tr>
<td>&lt;0.5&gt; &lt;0.01&gt; &lt;0.36&gt; &lt;1.0&gt;</td>
<td>weights</td>
</tr>
<tr>
<td>$^myfunction(args ...)</td>
<td>call to a function returning an FSM</td>
</tr>
</tbody>
</table>

Kleene employs a system of sigils\(^4\) to distinguish identifiers like $abc from simple concatenations of symbols like abc. A prefixed $ marks a variable name with a finite-state-machine value; a prefixed $^ marks the name of a function that returns a finite-state-machine value; and a prefixed $@ marks the name of a list of machines (see chapter 9).

---

\(^3\)In rare instances, the orthography of a language may contain digraphs, trigraphs, etc. that are always treated as indivisible units, and these might be encoded safely and usefully as multi-character symbols in a finite-state machine that models the phonology or orthography of that language.

\(^4\)http://en.wikipedia.org/wiki/Sigil_(computer_programming)
The Kleene Language

<table>
<thead>
<tr>
<th>abc</th>
<th>the concatenation of the three separate symbols ( a, b ) and ( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>$abc$</td>
<td>a variable named $abc$</td>
</tr>
<tr>
<td>$^abc(args...)$</td>
<td>a function named $^abc$ that returns a finite-state-machine value</td>
</tr>
</tbody>
</table>

It is important to note the distinction between a single-quoted multi-character symbol like '([Noun])', which denotes a single symbol with the multi-character print name [Noun], versus a double-quoted string like "dog", which denotes the concatenation of the individual symbols between the quotes: d followed by o followed by g. A double-quoted string can contain any Unicode BMP symbol except for newline and carriage-return.\(^5\)

Double quoting is not needed for normal alphabetic symbols—the regular expression "dog" is equivalent to dog, d o g, etc.—but is useful for literalizing special characters, e.g. "+", and for surrounding strings that include a special character, e.g. "AT&T" and "myfilename.txt".

A closed set of control characters can appear inside double-quoted strings, and in normal regular-expression text, represented using backslash conventions that will be familiar to many programmers.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Code Point Value</th>
<th>Character Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>0xA</td>
<td>LINE FEED (LF) or “newline”</td>
</tr>
<tr>
<td>\r</td>
<td>0xD</td>
<td>CARRIAGE RETURN (CR)</td>
</tr>
<tr>
<td>\t</td>
<td>0x9</td>
<td>CHARACTER TABULATION or “tab”</td>
</tr>
<tr>
<td>\b</td>
<td>0x8</td>
<td>BACKSPACE</td>
</tr>
<tr>
<td>\f</td>
<td>0xC</td>
<td>FORM FEED (FF)</td>
</tr>
</tbody>
</table>

For examples of the use of such special characters, and limitations on

\(^5\) The newline or LINE FEED character is \u000A; the CARRIAGE RETURN is \u000D.
writing finite-state machines containing such characters to XML, see Appendix C.

### 3.1.3. Regular-Expression Operators

The following regular expression operators are available, listed from high to low precedence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>parenthetical grouping</td>
<td>circumfix</td>
</tr>
<tr>
<td>:</td>
<td>crossproduct</td>
<td>infix</td>
</tr>
<tr>
<td>* + ? {2} {2,4} {2,}</td>
<td>iteration</td>
<td>postfix</td>
</tr>
<tr>
<td>~</td>
<td>complement/negation</td>
<td>prefix</td>
</tr>
<tr>
<td>(no overt operator)</td>
<td>concatenation</td>
<td>juxtapose</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>infix</td>
</tr>
<tr>
<td>&amp;</td>
<td>intersection</td>
<td>infix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>union</td>
</tr>
<tr>
<td>(various rule operators)</td>
<td>composition</td>
<td>infix</td>
</tr>
</tbody>
</table>

The : is the cross-product operator, used in examples like a: b, (book):(books) and $fst1:$fst2. The operands must denote regular languages (not regular relations), and the result is a relation that relates each string in one language to all the strings in the other language. Because : has high precedence, note that a regular expression like ab: cd is equivalent to a(b: c)d.

### 3.1.4. Precedence Issues

The relative precedence of : , ~, and the various postfix iteration operators could still be debated, though there are precedents to follow.\(^6\) Currently,

---

\(^6\)The precedence shown is that currently favored by Helmut Schmid (SFST), and it is the relative precedence that Lauri Karttunen has chosen for the PARC xfst code to be released with the second printing of the book *Finite State Morphology* (private communications).
\(\sim\). * is equivalent to \(\sim(.\,*\) and so denotes the empty language; \(a: b^*\) is equivalent to \((a: b)^*\), and \(\sim a: b\) is equivalent to \(\sim(a: b)\), which is semantically illegal.\(^7\)

By long tradition, concatenation has higher precedence than union and intersection, so one can write

\[
\$\text{foo} = \text{dog}\mid\text{cat}\mid\text{mouse} ;
\]

to mean

\[
\$\text{foo} = (\text{dog})\mid (\text{cat})\mid (\text{mouse}) ;
\]

Following other programming languages, \& has slightly higher precedence than |.\(^8\) The precedence of subtraction (-) relative to intersection (\&) is still debatable, but it should probably be higher than union (|). Rules are typically composed together, so composition is given lower precedence than the various operators used to construct rules. The use of parentheses, even when formally unnecessary, to show groupings can often improve the readability of your source code.

Unicode is embraced from the beginning in the Java/Swing GUI, and users are encouraged, though not required, to edit their script files using a Unicode-capable text editor.\(^9\) Unicode characters can also be indicated in the syntax using the familiar Java-like \(\text{\textbackslash uHHHH}\) escape sequence, i.e. \(\text{\textbackslash u}\) followed by exactly four hex digits: 0-9, a-f or A-F. For supplementary Unicode characters, Kleene recognizes the Python-like \(\text{\textbackslash UHHHHHHHH}\) escape sequence, i.e. uppercase \(\text{\textbackslash U}\) followed by exactly eight hex digits; and

---

\(^7\)In general, transducers are not closed under complementation, so \(\sim T\), where \(T\) is a transducer, causes a runtime exception in Kleene, much like division by zero in arithmetic expressions.

\(^8\)Similarly in most other languages, &&, the Boolean \textit{and}, has slightly higher precedence than ||, the Boolean \textit{or}.

\(^9\)Kleene has a Java-language parser, and so is able, using normal Java features, to read a file in almost any standard encoding and convert it to Unicode. Unless told otherwise, Java assumes that a file being read is in the default encoding of the host operating system and will convert it to Unicode accordingly.
for any Unicode character, Kleene also recognizes the \U{H...} notation, which contains one or more hex digits between the curly braces.

Kleene source files are typically Unicode files, and may contain code or comments containing arbitrary Unicode characters; so in Unicode files the \u sequence must be followed by exactly four hex digits, even inside comments.\(^\text{10}\)

Kleene also provides a function $^\text{charForCpv(#num)}$, aliased as $^\text{cpv2char (#num)}$, which takes an arbitrarily complex arithmetic expression (see chapter 8) as its argument and returns a character symbol. There is also a function $^\text{getIntCpv($char$)}$, aliased as $^\text{char2cpv($char$)}$, that takes a one-symbol regular expression and returns the code point value as a number. The following expressions are equivalent:

\begin{verbatim}
 fsm = a b c ;
fsm = a b \u0063 ;
fsm = a b $^\text{charForCpv(0x63)} ;
fsm = a b $^\text{charForCpv(99)} ;  // decimal 99 = hexadecimal 0x63
fsm = a b $^\text{cpv2char(0x63)} ;
fsm = a b $^\text{cpv2char(99)} ;

fsm = a b $^\text{cpv2char(0x62 + 1)} ;
fsm = a b $^\text{cpv2char(98 + 1)} ;
fsm = a b $^\text{cpv2char($^\text{char2cpv(a)} + 1$)} ;
\end{verbatim}

Note that the argument to $^\text{cpv2char (#num)}$ is an arbitrarily complex arithmetic expression, not a regular expression; and in an arithmetic expression, the plus sign + denotes simple arithmetic addition, not one or more iterations like the Kleene-plus in a regular expression.

\(^{10}\)Whatever the encoding of a Kleene source file might be, when it is read in, tokenized and parsed by Kleene, it is converted, one way or another, into Unicode. This is standard behavior for Java programs, which handle text internally as String objects, which are always Unicode strings.
3.1.5. Weights

The Tropical Semiring

Kleene can be used to build finite-state machines that are weighted or unweighted. The current implementation of weights is limited to the Tropical Semiring, which is the default semiring of the OpenFst library, and no doubt the most useful semiring for linguistic applications. In the Tropical Semiring

- The weights are floating-point logarithmic costs, to be explained below.

- The extension operation that accumulates weights along a single path, to calculate the weight of the whole path, is simple addition.\(^\text{11}\)

- The collection operation for combining the weights of multiple paths is \(\min\).\(^\text{12}\) Thus, among multiple solutions, the one with the minimum cost is preferred.

- As the extension operation is simple addition, the neutral weight for extension is 0.0.

- As the collection operation is \(\min\), the neutral weight for collection is infinity (\(\infty\)).

Where \(p\) is a probability ranging from 0.0 (impossible) to 1.0 (certain), the cost is calculated as \(-\log(p)\), such that 0.0 probability corresponds to infinite cost, and 1.0 probability corresponds to a cost of zero. In Kleene regular-expression syntax, weights are denoted within angle brackets, e.g. \(<0.1>\) and \(<0.9>\).

Each arc and each final state in a finite-state machine has a weight. An unweighted machine in Kleene is one in which all the weights are 0.0.

\(^{11}\)In OpenFst, the extension operation of each semiring is abstractly called \(\text{Times}(\).

\(^{12}\)In OpenFst, the collection operation of each semiring is abstractly called \(\text{Plus}(\).
Usually the weights in the Tropical Semiring are non-negative, and if the user tries to denote negative weights straightforwardly as, for example, <-0.1>, this creates a problem for the Kleene tokenizer, which recognizes the sequence <- as a left arrow, used in alternation rules, which will be presented in chapter 4. The workarounds for this problem, in the rare cases where negative weights might be required, are to

1. Put whitespace between the < and -, i.e. < -0.1>, or

2. Put parentheses around the negative value, i.e. <(-0.1)>

If the user types <-0.1>, Kleene generates a ParseException and, in the GUI, prints a message showing how to fix the syntax.

The use of the Tropical Semiring, with cost weights, has practical computational advantages, but programmers are much more likely to think in terms of probabilities (0.0 to 1.0) or percentages (0 to 100). Kleene supports functions that return arithmetic values (see chapter 8), including #^prob2c(#num), which takes a probability argument (0.0 to 1.0) and returns the cost, and #^pct2c(#num), which takes a percentage argument (0 to 100) and returns the cost. These functions can be used inside the angle brackets, e.g.

\[
\texttt{\$fst = c ( a <#^\text{prob2c}(.5)>}
\]
\[
\texttt{\hspace{1cm} | u <#^\text{prob2c}(.4)>}
\]
\[
\texttt{\hspace{1cm} | o <#^\text{prob2c}(.1)> ) t ;}
\]

or

\[13\]In the classic probability semiring, the weights are probabilities, and the extension operation, which combines the weights along a single path, is multiplication. Probabilities tend to be very small numbers, and when they are multiplied together, precision is lost. In the Tropical Semiring, where the weights are converted to costs, and the extension operation is addition, the precision problem is avoided. Also, computers traditionally performed addition more efficiently than multiplication, but this may not be true in modern computers.
\$fst = c
    ( a ^{#\text{pct}2c(50)} | u ^{#\text{pct}2c(40)} | o ^{#\text{pct}2c(10)} )
    t ;

to reflect that intuitive notion that “cat” occurs 50% of the time, “cut” 40% of the time, and “cot” 10% of the time.

**3.1.6. Whitespace in Regular Expressions**

Whitespace is ignored in Kleene regular expressions unless it is literalized. The following statements are equivalent, each denoting the simple concatenation of the d, o and g characters because the spaces in the regular expressions are simply ignored:

\$foo = dog ;
\$foo = do g ;
\$foo = d og ;
\$foo = d o g ;

Spaces and other whitespace characters can be inserted anywhere between operators and operands in Kleene regular expressions, and such whitespace often improves human readability. In the following examples, literal spaces are displayed as \( \_ \) for clarity.

A space in a regular expression can be literalized in three ways:

1. Putting the literalizing backslash directly before the space, i.e. \( \_ \)

2. Putting the space inside square-bracketed symbol unions [ ...] or [ ^...], e.g. [ \_\_abc] matches a, b, c or a literal space, or

3. Putting the space inside double quotes, e.g. " \_" and "John\_Smith"

For example, the following assignments are all equivalent, each containing two literal spaces:
$fsm = \text{to and fro} ;$
\text{\textbackslash $fsm = \text{to and fro} ;$
\text{\textbackslash $fsm = \text{to and fro} ;$
\text{\textbackslash $fsm = \text{to and fro} ;$
\text{\textbackslash $fsm = \text{to} } \text{and} \text{ fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
\text{\textbackslash $fsm = \text{to} \text{and fro} ;$
$\text{A language of strings that start with a lowercase letter a-to-z and continue with any number of lowercase letters or spaces can be defined as}$
$fsm = [a-z \ [a-z] \ * ;$
\text{Note that the second square-bracketed character set expression, [a-z], contains a literal space.}$
\text{The Kleene treatment of whitespace in regular expressions is similar to the way that whitespace is ignored in arithmetic expressions, and it is like Perl regular expressions marked with the /x suffix.}$
\text{3.1.7. Denoting the Empty String}$
\text{The empty (zero-length) string can be represented in various equivalent ways, including the Unicode U+03F5 GREEK Lunate epsilon symbol \epsilon, the Unicode escape sequence \textbackslash \text{U03F5}, the ASCII sequence \_e_, an empty double-quoted string \"", a? - a, etc. The global start-up script also defines the variables \textt{se and $\text{eps as the empty string. The following examples are all equivalent, denoting a relation with a on the upper side related to the empty string on the lower side:}$
\text{There are similar options in Python and Java to allow you to insert whitespace inside regular expressions to make them more readable for human beings.}$
\text{This Unicode character can be typed into the Kleene GUI, using standard Java Input Methods, including the CodePoint Input Method, and into any Kleene script prepared with a Unicode-capable text editor. The Unicode Standard specifies that U+03F5 GREEK LUNATE EPSILON SYMBOL is for use in mathematical formulas, such as regular expressions, and is not to be used in normal Greek text, where U+03B5 GREEK SMALL LETTER EPSILON is appropriate.}$
$\text{foo} = a: \epsilon$

$\text{foo} = a: \backslash u03F5$

$\text{foo} = a: _e_$

$\text{foo} = a: "$$

$\text{foo} = a: $e$

$\text{foo} = a: $\text{eps}$

3.1.8. Denoting Any Symbol

In Kleene regular expressions, the . (dot) syntax by itself is a wildcard that denotes any symbol in an acceptor or, in a transducer, the mapping of any possible symbol to itself. The ::. syntax denotes the mapping of any possible symbol to any possible symbol, including itself. The . therefore covers $a:a$, $b:b$, $c:c$, etc., but not $a:b$ or $b:a$; while ::. covers $a:a$, $b:b$, $c:c$, etc., plus $a:b$, $b:a$, etc.

$v = . ; // \text{any symbol, or map any symbol to itself}$

$w = ::. ; // \text{map any symbol to any symbol, including itself}$

As in the Xerox finite-state toolkit, the . really represents any possible character—including simple and multi-character symbols—or the mapping of any character to itself, and is not limited to some finite alphabet pre-defined by the programmer.

The semantics of the special . (dot or period) is quite complex, being interpreted into machines that match OTHER, also known as unknown, symbols, and this subject is treated in more detail in Appendix A. Luckily, the Kleene interpreter takes care of this, and the programmer doesn’t have to worry about the underlying complexity.

To denote a literal dot (a period) in a regular expression, use the backslashed \ or the double-quoted "$", or put the dot inside a square-bracketed symbol-union expression. Once again, literalized spaces are shown in the following examples as $\text{␣}$. 

$fsm = \text{T} h e \ \_\text{end} \ \_\text{.} \ ;$

$\text{fsm} = "\text{The}_{}\text{end."} \ ;$

$\text{fsm} = \text{The } "_\text{end} \text{."} \ ;$

$\text{fsm} = \text{T} h e \ [_\text{end}] \ e n d \ [_{.] \ ;}$

### 3.2. Abstraction Mechanisms

#### 3.2.1. Variables

As previously explained above, variables having a finite-state machine value are distinguished syntactically with a $ sigil, and they can appear on the left-hand side of an assignment statement.

```plaintext
$\text{variableName} = \text{RegularExpression} ;$
```

The regular expression can continue over any number of lines, and the assignment statement is terminated with a semicolon. Once variables such as $\text{foo}$ and $\text{bar}$ have been bound to FSM values, they can appear as operands in subsequent regular expressions.

```plaintext
$\text{foo} = \text{dog} | \text{cat} | \text{elephant} | \text{zebra} ; \ // \text{bind} \ \text{foo}
$\text{bar} = \text{bat} | \text{dog} | \text{octopus} | \text{frog} ; \ // \text{bind} \ \text{bar}$
```

// refer to and use the values of $\text{foo}$ and $\text{bar}$
// in a subsequent regular expression

```plaintext
$\text{result} = (\text{foo} | \text{bar}) - (\text{elephant} | \text{bat}) ;$
```

In this example, the resulting FSM would encode the language consisting of the strings *dog, cat, zebra, octopus* and *frog*. (The FSM could also be viewed and used as an identity transducer that maps each of these words to itself.) A reference to an unbound variable inside a regular expression raises a runtime exception, from which interactive Kleene can recover.

Because finite-state machines can get very large, copying is avoided. The following sequence of assignment statements results in $\text{var2}$ being an alias, bound to the same FSM object as $\text{var1}$. 

```plaintext
$\text{var1} = \text{dog} | \text{cat} ;$
$\text{var2} = \text{dog} | \text{cat} ;$
```
$\texttt{var1} = \texttt{a*b+[A-Za-z0-9][3]} ;
$\texttt{var2} = \texttt{var1} ;  // \texttt{var2} and \texttt{var1} are now bound
  // to the same FST

3.2.2. Pre-defined Functions

Regular Operations

Rather than inventing and proliferating new regular-expression operators, the Kleene philosophy is to give access to some operations via pre-defined functions, including

$^\texttt{invert}(\texttt{regexp})$

$^\texttt{reverse}(\texttt{regexp})$

$^\texttt{inputProj}(\texttt{regexp})$ or $^\texttt{inputside}(\texttt{regexp})$ or $^\texttt{upperside}(\texttt{regexp})$

$^\texttt{outputProj}(\texttt{regexp})$ or $^\texttt{outputside}(\texttt{regexp})$ or $^\texttt{lowerside}(\texttt{regexp})$

$^\texttt{rmWeight}(\texttt{regexp})$

$^\texttt{copy}(\texttt{regexp})$

Functions that return an FSM value are preceded with the $^\texttt{sigil}$. Note that $^\texttt{inputProj}()$, $^\texttt{inputside}()$ and $^\texttt{upperside}()$ are equivalent, where the “input” terminology reflects the OpenFst visualization of an FST, and the “upper” terminology reflects the Xerox visualization. The same holds for “output” (OpenFst) and “lower” (Xerox).

A function call that returns an FSM value is a Kleene regular expression and can, just like a variable having an FSM value, appear as an operand inside a larger regular expression. Note that while

$\texttt{var2} = \texttt{var1} ;$

simply makes $\texttt{var2}$ an alias for $\texttt{var1}$, binding $\texttt{var2}$ to the same FSM as $\texttt{var1}$,
$\text{var2} = \text{\textasciicircum copy}($\text{var1});

creates a deep copy of the FSM referenced by $\text{var1}$ and binds $\text{var2}$ to that deep copy.

The functions presented above do not destroy or modify their arguments, thus

$\text{orig} = (\text{dog}): (\text{chien});$
$\text{new} = \text{\textasciicircum lowerside}($\text{orig});$

leaves $\text{orig}$ intact while setting $\text{new}$ to a new FSM that encodes the language consisting only of the string \textit{chien}.

The following \textit{destructive} functions, which operate on an FSM in place, have also been defined:

$\text{\textasciicircum invert}!(\text{regexp})$
$\text{\textasciicircum inputside}!(\text{regexp})$ or $\text{\textasciicircum upperside}!(\text{regexp})$
$\text{\textasciicircum outputside}!(\text{regexp})$ or $\text{\textasciicircum lowerside}!(\text{regexp})$
$\text{\textasciicircum rmWeight}!(\text{regexp})$

Note that the names of destructive functions end with an exclamation mark to mark them as dangerous.\footnote{There is no magic to the exclamation mark, and simply adding one to the end of a function name does not make the function destructive.} After executing the following example

$\text{orig} = (\text{dog}): (\text{chien});$
$\text{new} = \text{\textasciicircum invert}!(\text{orig});$

both $\text{orig}$ and $\text{new}$ would be bound to the same modified FSM, with \textit{chien} now on the input (upper) side, and \textit{dog} on the output (lower) side. Such behavior is dangerous, not generally recommended, and the use of these destructive functions is recommended only for experts working at the limits of memory.
While Unicode code point values are integers, and Kleene does handle basic arithmetic expressions involving integers and floating-point numbers (see chapter 8), it is important to understand that Kleene makes a clear distinction between regular expressions and arithmetic expressions. A Unicode character like b, though it has an integer code point value of hexadecimal 62, is not, for Kleene, the same as the integer value. We have already presented the \u0062 notation, which is a regular expression and denotes the Unicode character b. In contrast, in Kleene arithmetic expressions, hexadecimal 62 is denoted as 0x62.

3.2.3. User-defined Function Syntax

Simple Examples

Users can also declare and call their own functions. As a minimal and admittedly silly example, consider the function definition

```plaintext
$^myunion($a, $b) {
    return $a | $b ;
}
```

which defines $^myunion as a function that takes two FSM arguments, represented here by the formal parameters $a$ and $b$, unions them together, and returns the FSM result. Note that the formal parameters are marked as being of type FSM by their $ sigils. Function names (of all types) in Kleene are always prefixed with the ^ prefix, and the $ prefix, as in $^myunion, marks this as a function that returns an FSM value. Once defined, our new function can be called just like a pre-defined Kleene function:

```plaintext
$foo = dog| cat| rat ;
$bar = elephant| horse| bird ;

$newfsm = $^myunion($foo, $bar) ;
// equivalent to $newfsm = $foo | $ bar ;
```
When a function is called, the arguments can be arbitrarily complex expressions, as long as they evaluate to a value of the type required by the function.

```perl
// call $^myunion with complex regular-expression arguments
$newfsm = $^myunion(worm | dog | rabbit | fox,
                    skunk | cat | hippopotamus);
```

In addition, for a function like $^myunion() that returns an FSM, a call to $^myunion() is itself a regular expression and can appear anywhere a regular expression is legal.

```perl
$newfsm = $^myunion(worm | dog | rabbit | fox,
                    $^myunion(skunk, $^myunion(cat, hippopotamus)));
```

**Practical Example**

As a much more practical function-definition example, consider the operation of *priority union*, which is defined as follows:

Let Q and R be transducers. The *priority union* of Q and R, giving *input-side* priority to Q, returns the union of Q and R with the added restriction that if both Q and R share an input string $i$, then the result transducer contains only the paths from Q that have $i$ on the input side.

Priority union with *output-side* priority is also potentially useful. In Kleene these functions can be defined as

```perl
$^priority_union_input($q, $r) {
    return $q | (~$^inputside($q) _o_ $r);
}
```

```perl
$^priority_union_output($q, $r) {
    return $q | ($r _o_ ~$^outputside($q));
}
```
Such function \textit{definitions} are equivalent to the following function \textit{assignments}, which have just a function variable on the left-hand side, followed by an equal sign and an \textit{anonymous function} on the right-hand side, which most users will find less friendly.\footnote{Lisp and Python programmers will recognize anonymous functions as “lambda” functions. Anonymous functions are important in the Scala and Clojure languages, and have recently been added to Java 8.}

\begin{verbatim}
$^\text{priority_union_input} = $^($q, $r) {
    return $q | (~$^\text{inside($q)} \_o_ $r) ;
} ;

$^\text{priority_union_output} = $^($q, $r) {
    return $q | ($r \_o_ ~$^\text{outside($q)}$) ;
} ;
\end{verbatim}

The anonymous-function expressions
\begin{verbatim}
$^($q, $r) { return $q | (~$^\text{inside($q)} \_o_ $r) ; }
\end{verbatim}

and
\begin{verbatim}
$^($q, $r) { return $q | ($r \_o_ ~$^\text{outside($q)}$) ; }
\end{verbatim}

look just like function definitions, but they begin with the $^\text{sigil}$ followed directly by the parameter list. So an anonymous function in Kleene is one that has a function sigil but no name. The $^\text{sigil}$, as always in Kleene, indicates that these functions return an FSM, and the parameter list, here ($q, r$), indicates that the functions take two FSM arguments. We will find uses for anonymous functions later, but for now we will restrict ourselves to the arguably friendlier function definitions.

Priority union can be useful in morphology to override regular but incorrect forms with their correct irregular forms. For example, assume that an FSM named $\text{productive_english}$ has been productively generated to
contain input ↔ output string pairs like the following (where [Verb] and [Past] are multi-character symbols, and ↔ is not a regular-expression operator but is used here just to indicate that two whole strings are related):

walk[Verb][Past] ↔ walked

kick[Verb][Past] ↔ kicked

think[Verb][Past] ↔ thinked

go[Verb][Past] ↔ goed

Incorrect forms like *thinked and *goed can be overridden by defining a smaller FSM encoding the correct mappings and simply priority-unioning it with the FSM $productive_english$.

$corrections = (\ (dig):(dug) \ |\ (go):(went) \ |\ (say):(said) \ |\ (think):(thought) ) ('[Verb] ' '[Past]'):"" ;

$english = $^priority_union_input($corrections, $productive_english) ;

Once defined, functions can be called directly in regular expressions and used in the definition of yet other functions. For example, the normal composition of Q and R is Q o R (also typeable in Unicode as Q ◦ R, using the Unicode RING OPERATOR character, U+2218); and if the input-side language of Q is I, then the input-side language of Q ◦ R may be a proper subset of I. That is, one or more of the original input strings of Q may not be accepted by the composition. The Lenient Composition of
transducers Q and R accepts exactly the same input language as Q. The following definition of $^\text{lenient\_composition\_input()}$ is appropriate for the examples in Karttunen's regular formalization of Optimality Theory (Karttunen, 1998), where the $^\text{base}$ transducer encodes a lexicon, and the $^\text{filter}$ transducer encodes an optimality rule or filter being composed “underneath” the lexicon.

$^\text{lenient\_composition\_input($^\text{base}, ^\text{filter}$) { return $^\text{priority\_union\_input($^\text{base\_o\_filter}, ^\text{base}$); }

When, conversely, the rule or filter is being composed “on top of” the lexicon, and the desire is to preserve the output language of the lexicon, then the following function $^\text{lenient\_composition\_output}$ is appropriate.

$^\text{lenient\_composition\_output($^\text{filter}, ^\text{base}$) { return $^\text{priority\_union\_output($^\text{filter\_o\_base}, ^\text{base}$); }

3.2.4. Function Call Semantics

Kleene maintains its environment as a directed graph of frames, where each frame contains a symbol table and both a dynamic link and a static link to other frames (or to null at the root of the environment). When a function is called, a new frame is allocated for its execution; the dynamic link of the new frame points back to the frame from which the function was called, and the static link points back to the frame where the function was defined.

The formal parameters of the function are bound, in the new frame’s local symbol table, to the passed-in argument values\textsuperscript{18} and any variables

\textsuperscript{18}When an FSM is passed as an argument, no copy is performed, and the local parameter becomes an alias for the original FSM. If it is necessary to pass a copy, the explicit $^\text{copy()}$ function can be used in the argument list.
introduced in the body of the function are also stored in the local symbol table. References to free (non-local) variables are resolved through the static link, thus implementing lexical scope. When the function terminates, it pushes the return value on the interpreter stack; then the calling frame, pointed to by the dynamic pointer, is once again made the current frame.

This fairly standard environment design supports functions that call other functions, functions that call themselves recursively, functions that themselves contain local definitions of functions, etc.

**Higher-order Functions**

Kleene also supports higher-order functions that return functions, as in the following example:

```plaintext
// a function that returns a function that returns an FSM
$^^append_suffix($suff) {
    // return a function, denoted here as an
    // anonymous function
    return $^($a) { return $a $suff ; } ;
}

$^append_ing = $^^append_suffix(ing) ;
$^append_espVend = $^^append_suffix(as| is| os| us| u| i) ;

$net1 = $^append_ing(walk| talk) ;
$net2 = $^append_espVend(pens| dir) ;

Recall that the sigil $^ marks a function that returns an FSM. Similarly, the sigil $^^, as in $^^append_suffix, marks a function that returns a function that returns an FSM. Note that the return statement in this function returns a function, notated as an anonymous function.

// return an anonymous function
return $^($a) { return $a $suff ; } ;
```
The function $^\text{append\_ing}$ just concatenates $\text{ing}$ to its FSM argument, so $\text{net1}$ will be set to an FSM that encodes the language containing $\text{walking}$ and $\text{talking}$. The function $^\text{append\_espVend}$ is designed to model the suffixation of Esperanto verb endings to verb roots, and $\text{net2}$ is set to an FSM that encodes the language containing “$\text{pensas}$”, “$\text{pensis}$”, “$\text{pensos}$”, “$\text{pensus}$”, “$\text{pensu}$”, “$\text{pensi}$”, “$\text{diras}$”, “$\text{diris}$”, etc.

### 3.2.5. Function Parameters with Default Values

In what follows, the term *parameter* is used for the local parameters in a function definition, and the term *argument* is used for the values passed in a function call. The parameters are bound to the passed-in function values when a function is called.

Kleene functions can be defined to have *required* and/or *optional* parameters, where optional parameters have explicit default values. In the following example, $a$ and $b$ are required parameters, and $c$ and $d$ are optional parameters.

```plaintext
$^\text{myfunc}(a, b, c = abc, d = xyz) \rightarrow 
\text{return } a \ b \ c \ d ;
```

In the parameter list, any required parameters must precede any optional parameters. A call to $^\text{myfunc()}$ must include at least two arguments to bind to the first two parameters—the *required* parameters. If a call to $^\text{myfunc()}$ does not supply arguments for the *optional* parameters, here $c$ and $d$, they are bound to the default values indicated in the function definition.

---

19The new Kleene argument-passing and parameter-binding scheme is modeled on that of Python, minus the Python parameters denoted with initial single and double asterisks. In Python, a parameter with a name like *foo* is bound to a tuple containing any extra positional arguments in the call; and a parameter with a name like **bar** is bound to a dictionary (hash table) containing any extra name = value pairs in the call.
A function call may contain *positional* and/or *named* arguments, where any positional arguments must precede any named arguments. A named argument is of the form `paramName = value`. The following function call contains two positional arguments and one named argument:

```bash
$net = $^myfunc(a*b+c, [a-z]{3,6}, $d = ing) ;
```

It is important to understand that there is not always a straightforward mapping between positional arguments and required parameters, nor between named arguments and optional parameters. As in Python, positional arguments are treated first; if there are \( n \) positional arguments in the call, their values are bound to the first \( n \) parameters, in syntactic order, whether those parameters are required (having no default value) or optional (having a default value). Any named arguments in the call are then used to bind the same-named parameters, whether those parameters are required or optional. The following runtime errors are detected:

- Passing more arguments than there are parameters
- Attempting to set a parameter twice, first by a positional argument and then by a named argument
- Failure to set a required parameter

The use of optional parameters and named arguments makes function calling more flexible and often more readable. Consider the following definition of a function with three required parameters:

```bash
$^func($prefix, $root, $suffix) {
    return $prefix $root $suffix ;
}
```

This function can be called with three positional arguments,

```bash
$net = $^func(re, work, ing) ;
```
or with named arguments in any order:

\[
\text{
$\text{net1} = \func{\text{think}, \text{"}, \text{ing}}$ ;
$\text{net2} = \func{\text{un}, \text{ing}, \text{do}}$ ;
}
\]

Function calls with positional arguments can, of course, easily become opaque to human readers, especially when the function contains an unusually large number of parameters.

If parameters are optional—having default values in the function definition—then the call may contain only the subset of arguments needed to override selected defaults. The function could, for example, expand the arguments, plus default values, into larger FSMs representing complex feature structures.

**Note to myself**: The AT&T Lextools contain a built-in syntactic feature that handles feature structures, linearizing the component features in a predefined canonical order. Show how this can be done with Kleene functions with optional arguments.

### 3.3. Right-linear Phrase-structure Grammars

#### 3.3.1. Right-linear Syntax

While regular expressions are formally capable of describing any regular language or regular relation, some linguistic phenomena—especially productive morphological compounding and derivation—can be awkward to model this way. Kleene therefore provides right-linear phrase-structure grammars that are similar in semantics, if not in syntax, to the Xerox/PARC lexc language (Beesley and Karttunen, 2003). While general phrase-structure grammars are context-free, requiring a push-down stack to parse, and so go beyond regular power, a right-linear (or left-linear) grammar is regular and so can be compiled into a finite-state machine.
A Kleene phrase-structure grammar is a set of productions, each assigned to a variable with a $>$ sigil. Productions may include right-linear references to themselves or to other productions, which might not yet be defined. The key to understanding productions is delayed evaluation. The productions are parsed immediately—and the parsed productions are stored as ASTs in the symbol table—but they are not evaluated until the entire grammar is built into an FSM via a call to the built-in function

$$\texttt{start(StartProduction)}$$

which takes one production name as its argument and treats it as the starting production of the whole grammar. The following example models a fragment of Esperanto noun morphotactics:

```
$\rightarrow$Root = ( kat | hund | elefant | dom ) ( $\rightarrow$Root | $\rightarrow$AugDim ) ;
$\rightarrow$AugDim = ( eg | et )? $\rightarrow$Noun ;
$\rightarrow$Noun = o $\rightarrow$Plur ;
$\rightarrow$Plur = j? $\rightarrow$Case ;
$\rightarrow$Case = n? ;

$\text{net} = \texttt{start(}$\rightarrow$Root) ;
```

The syntax on the right-hand-side of productions is identical to regular-expression syntax, but allowing right-linear references to productions of the form $\rightarrow$Name.

---

20 If anyone has a better suggestion for the syntax, please contact me.

21 Because it requires a special type of argument, $\texttt{start()}$ is a wired-in function that cannot be aliased. It is thus similar to what is called a special form in Lisp.
3.3.2. Right-linear Semantics

Enforcing the Right-linear Limitation

After a production is parsed into an AST, and before it is evaluated and stored in the symbol table, the AST is sent a message to accept a visitor object\textsuperscript{22} that ensures that all references to productions are genuinely right-linear. In the simplest cases, a right-linear reference is visually on the far right-hand side of the expression, as with \texttt{$>$Foo} in the following example:

\begin{verbatim}
$>$Production = a b c $>$Foo ;
\end{verbatim}

The following production is illegal because the reference to \texttt{$>$Foo} is not right-linear.

\begin{verbatim}
$>$Production = a b c $>$Foo d ;
\end{verbatim}

In more complicated examples, multiple legal right-linear references can be unioned at the end of the expression:

\begin{verbatim}
$>$Production = a b c ( $>$A | $>$B | $>$C ) ;
\end{verbatim}

Even more complicated examples are possible, as long as the references remain right-linear.

\begin{verbatim}
$>$Production = a b c ( d e f $>$X | g h i j $>$Y | k l m $>$Z ) ;
\end{verbatim}

Violations of the right-linear restriction are found and reported at production parse-time.

\textsuperscript{22}This visitor is of type RrKleeneVisitor.
The Implied Grammar

The built-in $^\text{start}(\cdot)$ function takes the identifier of a single production as its argument, and treats the start state of this production as the start state of the resulting FSM. The overall implied grammar is the set of productions including the starting production, any productions referred to by the starting production, and, recursively, any productions referred to by the referred-to productions. The productions of the implied grammar can also refer to normal FSM-valued variables and normal FSM-valued function calls that must be defined at the time when $^\text{start}(\cdot)$ is called. The call to $^\text{start}(\cdot)$ fails if any referred-to production, variable or function in the implied grammar is not yet defined.

If the call to $^\text{start}(\cdot)$ succeeds, it returns an FSM value. The productions are not consumed during compilation, but remain available for potential reuse. While productions can be defined at any time and in any place, it is often convenient to encapsulate an entire right-linear grammar in a function, which can be called and then deleted when no longer needed.

```perl
$^\text{esperanto_noun_function}() { 
    $_[Root] = (kat|hund|elefant|dom) ( $_[Root] | $_[AugDim] ) ;
    $_[AugDim] = ( eg | et )? $_[Noun] ;
    $_[Noun] = o $_[Plur] ;
    $_[Plur] = j? $_[Case] ;
    $_[Case] = n? ;
    return $^\text{start}($[Root]) ;
}

$\text{esp_nouns} = $^\text{esperanto_noun_function}() ;
delete $^\text{esperanto_noun_function} ;
```

Deletion of the function severs the reference link between the function identifier $^\text{esperanto_noun_function}$ and the underlying object that
represents the function, allowing the memory tied up by that object to be reclaimed.\textsuperscript{23} Another way to encapsulate right-linear grammars, in stand-alone code blocks, is described in section 3.4.8 on page 80.

**Uses of Right-linear Grammars**

\textit{Note to myself:} Make it clear that the productions can be defined in any order, as long as they are all available when $\texttt{start()}$ is called. Similarly, productions can refer to functions that have not yet been defined, as long as those functions are available when $\texttt{start()}$ is called. The delayed evaluation allows grammars to be written in a top-down fashion. Contrast regular-expression assignments, which must be fully evaluable at the moment they are parsed, with right-linear grammars, where evaluation is delayed. Try to reconstruct the Aymara-derivation example and include it here.

### 3.4. Scope

#### 3.4.1. Assignments, Declarations and Local Scope

Kleene, like Python, and unlike Java and C++, does not require variables to be declared. When a value is first assigned to a variable, as in

\[
\texttt{foo} = a*b+c-g;
\]

the statement automatically creates the variable \texttt{foo} and then binds the FSM value to the variable in the current local symbol table. A subsequent statement like

\[
\texttt{foo} = (\text{dog} \mid \text{cat} \mid \text{rat}) s?;
\]

\textsuperscript{23}The symbol tables and their objects are implemented in Java, and objects are automatically garbage collected when there are no more references to them.
causes the existing $foo to be re-bound to the new FSM value, again in the local symbol table.

3.4.2. Stand-alone Code Blocks

Kleene supports stand-alone code blocks, grouping a set of statements that are to be evaluated in a new frame, which means in a new scope. In the following example, a new frame/scope is allocated for the stand-alone block, and the $foo at point B is local to the block and distinct from the $foo at point A. Inside the block, the local $foo shadows the $foo defined at point A. At the end of the code block, the new frame/scope is released, and all memory used inside the block is freed and made available for garbage collection. At point C, the value of $foo is the language consisting of the word *cat*, and the $foo defined at point B is out-of-scope and unavailable.

```bash
$foo = cat ;       // point A
{
    $foo = dog ;   // point B
}
print $foo ;      // point C
```

3.4.3. Function Blocks

Function blocks are very similar to the stand-alone code blocks, being evaluated in a new frame/scope. The formal parameters of the function are bound as local variables in the new frame, and any variables created in the block are also local.

```bash
^func($a, $b) {
    // parameters $a and $b are local
    $foo = $a $b ;   // $foo is local
    return $foo ;
}
```
When the function terminates, it returns a value, unless it is a void function, and the new frame is released, freeing any locally used memory for garbage collection. While locally used memory is released when a function returns, the function definition itself is stored as an Abstract Syntax Tree (AST) that could potentially tie up significant memory, and that memory is retained until the function definition itself is deleted or goes out of scope. In contrast, the AST representing a stand-alone code block is released as soon as its execution is completed.

Note that the blocks of code in if-elsif-else statements and while loops are not evaluated inside a new frame/scope.

```plaintext
if (#^numStates($fst) < 30) {
   // this block is not executed in a new frame
   draw $fst ;
} else {
   // this block is not executed in a new frame
   print "the fst is too big to draw" ;
}
```

### 3.4.4. External Variables

Normally, from inside function blocks and stand-alone code blocks, it is not possible to change the value of non-local variables, i.e. variables that are defined in “higher” frames; nor is it possible to delete such variables. Exceptionally, it may be useful or necessary for code within a stand-alone code block or function block to change the value of a variable or variables outside that scope. To allow this, the variable or variables to be changed must be overtly declared external at the top of the stand-alone code block or function block. In the following example, $foo$ is declared external inside a stand-alone block, and so the $foo$ referred to at points B and C is the same $foo$ referred to at point A, and the value of $foo$ at point C

---

24The Python global declaration has a very similar function.
is the language containing the single string *dog*.

```plaintext
$foo = cat ; // point A
{
    external $foo ;
    $foo = dog ; // point B
}
print $foo ; // point C
```

Similarly in function blocks, external declarations allow a function to change the value of variables outside the function’s own block/scope. The following function could be called repeatedly to union a number of arguments
into $result.

$result = ~.\* ; // point A, $result set to the empty language
    // the empty language could also be denoted
    // as a-a, etc.
\^func($fst) {
    external $result;
    $result = $result | $fst ; // point B:
        // changes $result at point A
} // changes $result at point A

Similarly, a non-local variable can be deleted, from inside a function block
or stand-alone code block, only if it has been declared external.

$foo = foo ;
{
    delete $foo ; // this is illegal
}

$foo = foo ; // point A
{
    external $foo;
    delete $foo ; // deletes the variable at point A
}

Note that Kleene, consistent with its lexical scope, searches for the ex-
ternal binding of a variable $var declared external in a function block by
searching up the static environment links. If multiple $var variables are in
use in the program, you can see which $var is referred to by looking at the
source code and identifying the $var that is in scope where the function
is defined.\textsuperscript{25}

\textsuperscript{25}In the alternative, and rather old-fashioned, dynamic scope, the variable referred to
by the external statement would be the one active at the point where the function is
called, and this could of course change for each call.
An external statement can appear only in a stand-alone code block or in a function block, and it must precede any other kind of statement. There can be multiple external statements at the beginning of a block, and each statement can include multiple identifiers, optionally separated by commas.

```
{
    external $foo ;
    external $foo, $bar, $result ;  // optional commas
    external $sum $avg $collection ;

    // other statements here
}
```

### 3.4.5. Free Variables

In general, expressions inside a stand-alone code block, or inside a function block, can always refer to variables outside the local scope, to retrieve their values, but they cannot change the value of such variables unless they are expressly declared to be external. In the following example, at point B, $foo is referred to, and its value retrieved, and this is legal and normal. Because there is no local binding of $foo, Kleene searches up the static environment links to find the first higher frame that contains a binding for $foo, and returns the value. In such a case, $foo is known as a free variable.

```
$foo = dog | cat | rat ;  // point A
{
    $bar = $foo $? ;       // point B, $foo is a free variable
    print $bar ;
}
```

In an assignment statement, free variables occur only on the right-hand side. If a variable, such as $bar at point B, appears on the left-hand side
of an assignment, then its value is being changed, and it is assumed to be local—and is created locally if it doesn’t already exist—unless it was overtly declared to be external.

3.4.6. Combining Free and Local Usage

It is possible to write a stand-alone code block or function block that first retrieves the value of a variable \$var\ as a free variable, and then subsequently tries to create and set \$var\ as a local variable.

    // a bad, confusing example
    {
        $a = a b c \$var ; // point A: \$var\ is a free variable
        // ...
        $var = a*b+[c-g] ; // point B: attempt to create/set
        // $var as a local variable
        // ...
    }

The statement at point B creates and sets \$var\ as a local variable because it was not declared external. The same problem can arise in a single assignment statement, where a variable name appears both on the right-hand and left-hand sides of an assignment.

    // another bad, confusing example
    $var = xyz ;
    {
        $var = a b c $var ;
        // ...
    }

In the assignment statement, the right-hand side is evaluated first, including the retrieval of the value of \$var\ as a free variable. Then, because
$\text{var}$ was not declared external, it would attempt to create a local $\text{var}$ and bind it to the value of the right-hand side.

While such code makes sense theoretically, and could be allowed, it is dangerously confusing and probably represents a programming error. For this reason, Kleene detects such cases and generates an exception at execution time.\footnote{Python also disallows such usage, and it generates an error at function-definition time.}

### 3.4.7. Export Statements

As an experiment, Kleene supports export statements that can appear only in a stand-alone code block, and which cause a local variable, and its bound value, to be “exported” up from the current frame to the mother frame.\footnote{For stand-alone code blocks, the static mother frame and the dynamic mother frame are the same.} Consider first the following example, which creates a local #\text{var} inside a stand-alone block and then tries to reference it after the block has terminated.

```plaintext
{
    $\text{var} = \text{abc} ; \quad // \text{point A}
}
\text{print} \; $\text{var} ; \quad // \text{point B: error, } $\text{var} \text{ is out of scope}
```

This example generates an exception because $\text{var}$ was created at point A but is out of scope and unavailable at point B. However, the example can
be made to work by exporting $var.

{
    $var = abc ;       // point A
    export $var ;     // export $var to the mother frame
}
print $var ;       // point B: no error, $var has the value "abc"

So far, exporting a value from a stand-alone code block looks a lot like calling a function that returns a value. A stand-alone code block can also contain multiple export statements, which can appear anywhere (but after external declarations, if any). A single export statement can list multiple variables, optionally separated by commas.

{
    $foo = abc ;
    $bar = xyz ;
    export $foo, $bar ;

    $a = a ;
    $b = b ;
    $c = c ;
    export $a $b $c ;
}

3.4.8. Practical Use of Code Blocks

Finite-state machines can often get alarmingly big, sometimes too big to process; and FSMS are normally persistent, taking up memory until they are overtly deleted or go out of scope. Stand-alone code blocks and function blocks can be used to help minimize the use of memory when programming.

For example, in practical finite-state programming it is very common to define a set of intermediate FSMs and then combine them somehow into
the final desired FSM. The intermediate FSMs can then be deleted to allow their memory to be garbage collected. Take the following example that models simple Esperanto nouns.

```plaintext
$nroot = elefant | kat | hund | bird ;
// elephant cat dog bird
// $nroot could be expanded to thousands of roots
$augdim = eg | et ;
$num = j ;
$case = n ;

$nouns = $nroot $augdim* $num? $case? ;
// now delete intermediate FSMs no longer needed
delete $nroot $augdim $num $case ;
```

Of course, it requires some attention and discipline to recognize and delete no-longer-needed intermediate FSMs, and it is far too easy to leave them lying around, taking up valuable memory.

Kleene programmers writing such grammars are encouraged to group them into stand-alone code blocks that either export the final value or
declare an external variable and set it to the final value, e.g.

```{  // intermediate FSMs
        $nroot = elefant | kat | hund | bird ;
        $augdim = eg | et ;
        $num = j ;
        $case = n ;
        // the final FSM
        $nouns = $nroot $augdim* $num? $case? ;

        export $nouns ;
    }

or

$nouns = "" ;

```  

Either way, at the end of the code block, its frame will be released and all of its local memory will be freed for garbage collection. Only the variables and values explicitly exported, or declared external, will remain.

Stand-alone code blocks can also be nested, e.g. to model Esperanto nouns and verbs, one might write the following:
After the evaluation of this nested block, only the final $esp variable would be available, and all the intermediate code and FSMs would be released, as if they had been explicitly deleted. Again, export is implemented as an experiment, and it remains to be seen if they will prove truly useful.

### 3.5. Language Restriction Expressions

Kleene language-restriction expressions are regular-expression abbreviations that denote regular languages (not regular relations) and compile into FSMs that are finite-state acceptors. For example, the expression
The Kleene Language

\[ b \Rightarrow a \_ c ; \]

denotes the Universal Language minus all strings that violate the restriction that any symbol b must be preceded immediately by a and followed immediately by c. Thus the denoted language includes dog and elephant, and all other strings that do not contain b; and it contains strings like abc, aaabcm, qaaabcmnaber wherein every occurrence of b is preceded by a and followed by c. The language excludes all strings, including bac, abm and abcqqabr, that contain any b that is not surrounded with the specified left and right contexts.

In general, the language-restriction syntax is

\[ \text{content} \Rightarrow \text{leftContext} \_ \text{rightContext} \]

where content, leftContext and rightContext can be arbitrarily complex regular expressions with the semantic restriction that each must denote a regular language (not a relation). The leftContext and/or rightContext can be omitted

\begin{align*}
  b & \Rightarrow a \_ \\
  b & \Rightarrow \_ c
\end{align*}

and word boundaries can be indicated with #. The expression

\[ b \Rightarrow \# \_ \]

allows b to appear only at the beginning of a word, and, similarly,

\[ b \Rightarrow \_ \# \]

allows b only at the end of a word. The following expressions allows abc to appear only when followed by quam and the end of the string.

\[ abc \Rightarrow \_ quam \# \]
To include a literal pound sign (also known as the hash mark or hash sign) in any regular expression, it should be literalized in the usual Kleene ways: either as `\#` or "#" or `[#].

Restriction expressions can also include multiple contexts, separated by `||`, a double vertical bar. The following expression denotes the Universal Language, minus any strings that violate the restriction that each occurrence of abc must appear either at the beginning of a string, at the end of a string, or in the context between `left` and `right`.

```
abc => # || _ # || left _ right
```

Where multiple contexts are provided, each example of the content must appear in at least one of the contexts indicated.

Because restriction expressions are regular expressions that denote regular languages and compile into acceptors, they can be assigned to variables just like any other regular expressions.

```
$var = abc => left _ right ;
```
Chapter 4

Alternation Rules

4.1. What are Alternation Rules?

Alternation rules are extensions of the regular-expression language, and each alternation rule compiles into a finite-state transducer. As always, finite-state transducers encode regular relations, and the “action” of a rule, which appears superficially to be algorithmic, changing an input string into one or more output strings, is in reality just the matching of a string of input on one side of the relation and the return of the related strings on the other side of the relation.

Kleene has its own rule syntax, designed to be as familiar and intuitive as possible to trained linguists, but the rules are interpreted using algorithms invented by Dr. Måns Huldén, who has made them freely available for general use.\footnote{Earlier versions of Hullén’s algorithms are laid out in his dissertation (Huldén, 2009), and used in his Foma language (http://foma.googlecode.com/); but Huldén generously supplied his latest algorithms and allowed them to be used in Kleene.} In addition, he kindly supplied documentation, examples and patient consultation during the development of Kleene alternation rules.

Kleene offers an unusually broad range of alternation-rule types, inspired by the Replace Rules of the xfst language (Beesley and Karttunen,
The rule types include right-arrow (downward-oriented) and left-arrow (upward-oriented) rules, rules with multiple contexts, optional rules, rules constrained to maximal or minimal matches, epenthesis and markup rules. Rules can be composed in a vertical derivation (a cascade), as in xfst Replace Rules. Thanks to the new Huldén algorithms, Kleene rules can also be compiled in parallel, with unprecedented freedom; and rule contexts can denote two-level transducers as well as one-level acceptors.

In the hands of experts, the various Kleene alternation-rule types will be powerful tools for modeling a wide variety of linguistic phenomena, but the mastery of the rich syntax and semantics of alternation rules will be a challenge for all learners. The parsing and interpretation of alternation rules were two of the most difficult challenges in the implementation of Kleene.

4.2. Mindtuning for Alternations

4.2.1. Underlying and Surface, Upper and Lower

In many kinds of linguistic theory and natural-language processing, and especially in phonology and morphology, there is often postulated an abstract, deep or underlying level of analysis that gets transformed, mapped or related, perhaps via one or more intermediate levels, to a final or “surface” level. Common examples include the following:

- Tokenization: The abstract level is a string of characters representing a sentence written in a natural language, such as French or German, and the surface level is the same string of symbols, but with token-boundary symbols inserted. If the token-boundary symbol is the newline character, and the surface-level string is printed, then one token will appear on each line.

- Syllabification: The abstract level is a string of symbols representing a word written in a traditional orthography, or in a phonemic orthog-
raphy, and the surface level is the same word with syllable-boundary characters inserted.

- Transliteration: The abstract level is a word written in traditional orthography, e.g. a Russian word written in Cyrillic characters, or a Greek word written with Greek characters, and the surface level is a representation of the pronunciation of the word written in some romanic alphabet such as the International Phonetic Alphabet (IPA).

- Morphology: The abstract level consists of a baseform or a traditional dictionary citation form, plus morphological tag symbols representing part-of-speech, tense, voice, mood, person, number and gender—e.g. *cantar*[Verb][PresIndic][1P][Pl], indicating the Spanish verb *cantar* (‘to sing’), marked as a verb, present indicative, first person and plural—and the surface form is a string that is the corresponding inflected form of the verb: *cantamos*.

The changes or differences between the abstract level and the surface level are technically known as *alternations*, and they can be described using alternation rules, which are the subject of this chapter. In Kleene and in some other implementations of regular relations, such rules can be compiled into finite-state transducers, which can be used to compute the mappings between the levels.

Before getting too formal, let’s look at some simple, intuitive examples involving the mapping of standard orthographical words into strings representing pronunciations, a task that often faces students trying to learn how to pronounce words written in the standard orthography of a foreign language. This is also a critical step in implementing text-to-speech systems. In Classical Latin, for example, the orthographical *c* always represented the /k/ phoneme, so the word written *canis* was pronounced /ˈkaniːs/, and *pacem* was pronounced /ˈpaːkem/. In this example, there is an alternation

\[ \text{The pronunciation was in fact } /ˈpɑːkem/, \text{ with a long } /əː/, \text{ but vowel length was not marked in Latin orthography and we will ignore it here.} \]
between \(c\) at the orthographical level and \(k\) at the pronunciation level. We can visualize the related strings vertically as

Orthographical level: pacem  
Pronunciation level: pakem

More generally, we will visualize the levels as upper vs. lower, as we have for all regular relations.

Upper level: pacem  
Lower level: pakem

Upper level: canis  
Lower level: kanis

We say that an upper string \(pacem\) is related to a lower string \(pakem\), with an alternation from \(c\) to \(k\). Using a rule formalism already familiar to most trained linguists, and we could describe this alternation using the alternation rule

\[c \rightarrow k\]

which is read (in various traditions) as “\(c\) becomes \(k\),” or “\(c\) is rewritten as \(k\),” or “\(c\) is realized as \(k\),” or “\(c\) maps to \(k\).” Here we will favor the “realize” and “map” terminologies, and more precisely we will talk about an upper-level \(c\) being realized as, or mapping to, a lower-level \(k\). The syntax \(c \rightarrow k\) is a valid Kleene alternation rule, and the arrow can be typed either as a hyphen - followed by a right angle-bracket \(>\) or as \(\rightarrow\), the Unicode RIGHTWARDS ARROW, which has the code point value 0x2192.

\[c \rightarrow k\]

In Italian, one of the many modern descendants of Latin, the \(c\), originally always representing /\(k\)/ in Latin, has become palatalized in the context before the vowels /\(i\)/ and /\(e\)/, and is now pronounced /\(ʧ\)/ in those
environments, like the English <ch> in chin /ʧɪn/. Elsewhere, e.g. before the vowels /a/, /o/ and /u/, the c is still pronounced /k/, as it was in Latin. Ignoring Italian accented vowels for the moment, we could describe these alternations as

\[ c \rightarrow \text{ʧ} / \_ (e \mid i) \]

read as “upper-level c maps to lower-level tf in the context before e or i” and

\[ c \rightarrow k / \_ (a \mid o \mid u) \]

read as “upper-level c maps to lower-level k in the context before a, o or u.” Note that the rules have a left-hand-side and a right-hand-side, separated by a forward slash. The underscore marks the location in the context where the alternation occurs. For now we will type the rule arrow as a hyphen followed by a right angle bracket.\(^3\)

In Latin-American Spanish, another descendant of Latin, the orthographical c maps to s before /e/ and /i/,

\[ c \rightarrow s / \_ (e \mid i) \]

and the \[ c \rightarrow k / \_ (a \mid o \mid u) \] rule is the same as in Italian. We’ll refine these rules below.

---

\(^3\)The Kleene parser will accept either a hyphen followed by a right angle bracket, or the Unicode RIGHTWARDS ARROW character, →, which has the code point value U+2192. If you install the Kleene.kmap file (a Java input method), and select it via the KMAPime express input method in the Kleene GUI, then when you type a hyphen followed immediately by a right angle bracket, the two typed characters will be automatically intercepted and substituted with a RIGHTWARDS ARROW (→). A kleene_utf-8.vim keymap file is also available for the gvim editor.
4.2.2. Writing and Testing Alternation Rules

As previously stated, Kleene rules are extensions of regular expressions, and they compile into finite-state transducers that encode the alternation as a mapping between the upper-level and the lower-level languages. To manually test the Latin $c \rightarrow k$ rule in the Kleene GUI, simply enter in the terminal window

```
$rule = c -> k ;
test $rule ;
```

or just

```
test c -> k ;
```

and a testing window will appear.

![Testing Window](image)

The testing window has an editable upper input field labeled String » at the top and an editable lower input field (also labeled String » at the bottom. The bar in the middle labeled “FST” represents the FST being tested.
Type *pacem* in the *upper* field, press the ENTER key, and the following output will appear in the GUI terminal window:

```
pakem : 0.0
```

Congratulations. You have just written and tested your first Kleene alternation rule. For now don’t worry about the “0.0,” which represents a weight of zero, being the neutral weight or the absence of weight. Try testing this rule with a few more Latin words with *c* such as *vici, pecunia, agricola, victoria*.

Then experiment with the following Italian and Spanish rules. Kleene can handle Unicode characters like ʧ, and the GUI will display them if your Java installation has a font that includes International Phonetic Alphabet glyphs, but for now you can substitute some other character or characters, e.g.

```
// Italian,
// using tS to represent the affricate represented by
// the ch in chin
test  c -> tS / _ (e| i) ;
```

```
// Spanish
test  c -> s / _ (e| i) ;
```

Note also that if you enter

```
$myrule = c -> tS / _ (e| i) ;
```

Kleene will display in the symbol-table window an icon, clearly labeled “$myrule,” that represents the resulting FST. If you right-click on that icon, a menu will appear, and you can select the test menu item to cause the test window to be displayed.

[KRB: add a graphic showing the pull-down menu.]
4.2.3. Derivations or Cascades of Rules

In the Italian and Spanish examples, there are actually two rules working together to describe the realizations of $c$ in different contexts. We’ll look at the Spanish examples and refine them to be more correct and robust. One way to combine the rules together is via composition, an operation for which the syntactic operator is the Unicode RING OPERATOR, $°$, which has the code point value 0x2218. Few fonts supply a glyph for this character, so for now, and at any time, you can use the ASCII equivalent _o_, consisting of an underscore, followed by a lowercase o letter, followed by another underscore.\footnote{If you use the Kleene.kmap input method in the GUI, or the kleene_utf-8.vim keymap in gvim, the typed _o_ sequence will be intercepted and replaced with the RING OPERATOR $°$.} Try entering the following derivation or “cascade” of rules and testing them using words like casa, cosa, poco, curioso, ciudad, cace and cimento.

\[
\text{// Two Spanish rules composed together}
\]
\[
\text{
rule = c -> s / _ (e|i) }_o_ \text{ c -> k / _ (a|o|u) ;
}
\]
\[
\text{test $rule ;}
\]

To better visualize the cascade of the two rules, they can equivalently be typed as

\[
\text{// Two Spanish rules composed together}
\]
\[
\text{
rule = c -> s / _ (e|i)
_0_
\text{c -> k / _ (a|o|u) ;
}
\]
\[
\text{test $rule ;}
\]
and we will often talk about one rule, here the $c \rightarrow k$ rule, being composed “below” or “underneath” the $c \rightarrow s$ rule. Recall that regular expressions, including those with alternation-rule notations, can extend over multiple lines and are always terminated with a semicolon. When this cascade of rules is applied in a downward direction to an input string, the output of the first (top) rule $c \rightarrow s / \_ (e| i)$ becomes the input to the second $c \rightarrow k / \_ (a| o| u)$ rule underneath it, and the output of the application is the output of the final (here the second) rule. However, when the rules are composed together, as in this example, there is no longer any intermediate level, the result is a single transducer, and the input is literally mapped to the output in a single step.

Spanish speakers will note that the $c \rightarrow s$ rule should also apply in the context before the accented letters é and í. In addition, $c$ can also occur in environments not followed by a vowel, and $c$ is realized as $k$ everywhere except before e, é, i or í. To better capture the facts of Spanish pronunciation, we can simply add the accented variants to the context of the $c \rightarrow s$ rule, and remove the context altogether from the $c \rightarrow k$ rule. Intuitively, after the $c \rightarrow s$ rule has fired, any $cs$ still remaining will be mapped to $k$.

// Improved Spanish rules for the realization of $c$
$rule = c \rightarrow s / \_ (e| é| i| í)
_\_\_\_
\_\_\_\_\_
c \rightarrow k ;

test $rule ;

The new rule cascade will now handle cases like *pacto* and *accidente*, outputting *pakto* and *aksidente*, respectively. Note that if a rule does not “match and fire,” then it simply maps the input string to itself.

[KRB: add graphics showing the rule compiled into finite-state machines]
As another example, consider the case of an imaginary language that has a morpheme $kaN$, terminating in an unspecified nasal consonant that we will represent as uppercase $N$, that can concatenate onto another morpheme $pat$ to form the abstract string $kaNpat$. Let us assume that this language has two alternation rules:

1. The unspecified nasal $N$ is realized as $m$ in the context immediately before $p$ or $b$, and

2. $p$ is realized as $m$ in the context immediately after $m$, which might be an abstract $m$ or an $m$ that was originally an $N$.

These rules can be written, composed and tested in the GUI by entering

```latex
$r = N \to m / _ (p|b) \\
_p -> m / m _ ; \\
test \ r ;$
```

and then entering $kaNpat$ as the upper-side input. The output is the string $kammat$. Intuitively, the downward application of the first $N \to m$ rule will map $kaNpat$ to the intermediate string $kampat$, and then the second $p \to m$ rule will map from $kampat$ to $kammat$. However, when the two rules are compiled and composed, the intermediate level simply disappears, and the resulting transducer maps $kaNpat$ to $kammat$ in a single step.

Using the same transducer, if you input the string $kampat$, then intuitively the first rule will not match, and will simply map $kampat$ to itself (an identity mapping), and then the second rule will map from $kampat$ to $kammat$. But once again, the composed rules result in a single two-level transducer that performs the mapping in a single step. Note also that if you input $kammat$, then the result is again $kammat$.

At this point it is instructive to use the same transducer and apply it in an upward direction to the input $kammat$. To do this, simply enter $kammat$ in the lower input field of the testing window. The output will consist of three strings:
We get these three results because the FST compiled from the two rules is a two-level transducer that encodes a regular relation. That relation maps an upper-side *kaNpat, kampat* or *kammat* to a lower-side *kammat*; so, looking at the relation in the opposite, upward direction, it maps the lower-side *kammat* to the three upper-side strings *kaNpat, kampat* and *kammat*.

So far we have experimented with individual rules, and simple cascades of two rules. More generally, in the realm of regular languages and regular relations, the upper string is “mapped” into one or more lower strings via a set of ordered rules, which could number in the dozens or hundreds. Between each pair of rules, there will be a conceptual intermediate level.
... 
Rule n
Lower Level

Formally speaking, the rule cascade represents a regular relation that has the universal language as the upper-level language, and all the intermediate languages, and the surface language, are all regular languages as well. The various rules, each one representing a regular relation, can be combined together into a single two-level transducer via the composition operation

Upper Language
  Rule 1
    _o_
  Rule 2
    _o_
  Rule 3
    _o_
  ...
  Rule n
Lower Language

and the result of the composition is a single two-level transducer that directly relates the upper language and the lower language. In the process of composition, all of the intermediate languages disappear.

4.2.4. The Richness of Alternation-Rule Types

Kleene provides a rich set, one of the richest sets anywhere, of alternation-rule types that can, in expert hands, be used to model a wide variety of linguistic phenomena. Unavoidably, this richness is a challenge to the beginner, and I will try to show how each rule type is typically used and which types are most commonly used. The semantics of alternation rules are elegant but occasionally hard to grasp, and their computational implementation is one of the biggest challenges in any language like Kleene.

Like traditional Chomsky-Halle Rewrite Rules, and like Xerox/PARC Replace Rules, multiple Kleene alternation rules can be written and ordered in a derivation or cascade. Because each rule in the cascade compiles into a finite-state transducer, the closure properties of such transducers allow us to compose the cascade of transducers together into a single finite-state transducer that has the exact same behavior as the original cascade, but which maps directly from input to output in a single step. This was one of key theoretical discoveries of Johnson.

In addition, there are various ways that Kleene alternation rules can be notated to compile and apply in parallel,\(^5\) which is a key characteristic of the “two-level” rules proposed by Koskenniemi’s Two Level Morphology and implemented in the now little-used Xerox/PARC twolc compiler.\(^6\) However, in Kleene parallel rules, the rules are not limited, as in traditional Two Level Morphology, to constraining only pairs of single characters.

\(^5\)Xerox/PARC Replace Rules offer a limited capability for parallel composition, but the rule types (rule arrows) must be exactly the same, which is not a restriction in Kleene.

\(^6\)Traditional two-level rules continue to be popular at the University of Helsinki, which has re-implemented twolc (https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstTwolC), and generally in Finland and Scandinavia.
4.3. Basic Mapping Rules

4.3.1. Right-Arrow Rules

We will now start looking at alternation rules with a bit more rigor. The most common and useful alternation rules are obligatory right-arrow mapping rules, written using the \( \rightarrow \) or \( \Rightarrow \) arrow, which are superficially similar to the classic “Rewrite Rules” of Chomsky & Halle.

The simplest alternation-rule syntax in Kleene is

\[
\text{upper} \rightarrow \text{lower}
\]

where \( \text{upper} \) and \( \text{lower} \) are arbitrarily complex regular expressions, though they are semantically limited to denoting regular languages (that is, \( \text{upper} \) and \( \text{lower} \) cannot denote regular relations). Thus the following rules are valid

\[
a \rightarrow b
\]

\[
dog \rightarrow \text{Hund}
\]

\[
ab*c+(d|e|f) \rightarrow x\{2\}y+z
\]

while the following, which have \( \text{upper} \) or \( \text{lower} \) expressions that denote regular relations (transducers) are illegal.

\[
// \text{Semantically illegal}
a: b \rightarrow c
\]

\[
// \text{Semantically illegal}
n \rightarrow n: m
\]

\[
// \text{Semantically illegal}
e: i \rightarrow o: u
\]
More precisely, the *upper* and *lower* expressions must be interpretable as denoting languages. Recall that OpenFst finite-state machines are always two-level, with each arc having an upper symbol and a lower symbol. So each finite-state machine is implemented as a transducer, and acceptors are special cases of such machines where, on every arc, the upper and lower symbols are the same. Thus the regular expression $b$ and the regular expression $b: b$ are structurally equivalent—they both get interpreted to produce an FSM with one arc that has a $b$ symbol on the upper side and a $b$ symbol on the lower side. By convention, this machine is interpretable as an acceptor.

Consistent with that convention, the following rules are valid because both the *upper* and *lower* expressions are interpretable as denoting regular languages (not relations).

// Syntactically and semantically legal,
// equivalent to a -> b
a: a -> b: b

// Syntactically and semantically legal,
// equivalent to dog -> chien
d: d o: o g: g -> c: c h: h i: i e: e n: n

(dog):(dog) -> (chien):(chien)

But if either the *upper* expression or the *lower* expression cannot be interpreted as denoting a regular language, an exception will be thrown at runtime.

Alternation rules can also be constrained to match and “fire” only in a specified context, based on the following template, where *upper*, *lower*, *left* and *right* can all be arbitrarily complex regular expressions, but all are semantically limited to be interpretable as regular languages. The left-hand-side of the rule is separated from the context (which is the right-hand-side) by a forward slash /.

The underscore character _ marks the
position in the context where the upper expression appears.

// left-hand side / right-hand side
upper -> lower / left _ right

When such a rule is applied “in a downward direction” to an input string, the input string must match the left context, the upper expression and the right context in order for the rule to fire; if the rule does match the input, then the upper content on the upper side is mapped to the lower content in the output(s). If the input is not matched by the rule, then the rule simply maps the input to itself, the identity mapping.

The use of the underscore character to separate the left context from the right context signals the Kleene interpreter that the rule writer intends the context to be interpreted as one-level, i.e. the left and right context expressions denote regular languages and are to be matched on the upper side of the relation. As we shall see below, Kleene alternation rules based on Huldén’s algorithms can also have two-level contexts, denoting transducers; but there is such a long tradition of rules with single-level contexts that Kleene retains this traditional syntax for the traditional semantics. If the programmer genuinely intends to designate two-level contexts, then that will be signaled by a slightly different syntax to be presented below. For now we will concentrate on basic mapping rules wherein all the rule parts—upper, lower, left context and right context—are intended by the programmer to denote regular languages, and where the accidental definition of a two-level context will be caught and signaled as an error.

The following rule examples are illegal because the contexts are two-level:

// Semantically bad rules
a -> b / q:r _ s:t

a -> b / q:r _

a -> b / _ s:t
The contexts can consist of or contain variables previously defined, and they must denote regular languages.

// Semantically good
$left = qrst ;
$right = xyz ;

$r = a \rightarrow b / \$left _ \$right ;

If either context part cannot be interpreted as a regular language (an acceptor), then the rule is semantically bad and a runtime exception will be thrown. As the following example illustrates, illegal two-sided contexts can in general be detected only at runtime, when the context expressions are evaluated. This is similar to the way that illegal division by zero must be detected in general-purpose programming languages.

// Semantically bad if used as rule contexts
$left = (qr):(st) ;
$right = x: y ;

a \rightarrow b / \$left _ \$right ; // throws a runtime exception

Although an overall alternation rule compiles into a two-level transducer that could be applied either in an upward or a downward direction, right-arrow rules have a clear downward bias to them; we normally think of the upper level of a two-level rule as the input side of the relation. (This notion of input side corresponds with the OpenFst visualization of a machine's “input” side.) In such a basic right-arrow rule, the input expression and both the left and right context expressions must match on the upper side of the relation.

As a concrete example, the following rule, intended to model a phenomenon in Italian and Portuguese pronunciation of written words, maps $s$ to $z$ in the context between two vowels.
s -> z / (a|e|i|o|u) _ (a|e|i|o|u)

This particular rule could be written equivalently as

s -> z / [aeiou] _ [aeiou]

Either way, this rule would map, in a downward direction, from casa to caza, mesa to meza, isose to izose, etc.

The syntactic contexts in alternation rules are optional, allowing the following variations:

upper -> lower / left _ right // both contexts present
upper -> lower / _ right // right context only
upper -> lower / left _ // left context only
upper -> lower // no contexts

A missing context is semantically equivalent to .*, the Universal Language, e.g.

upper -> lower / _ right // right context only
upper -> lower / .* _ right // equivalent

Unless a left or right rule context explicitly uses # to reference the beginning or end of a word, the context is implicitly extended with .*, the Universal Language.

s -> z / [aeiou] _ [aeiou]

// is interpreted as
s -> z / .* [aeiou] _ [aeiou] .*

Alternation rules can have multiple contexts, separated by ||, a double vertical bar.

input -> output / L1 _ R1 || L2 _ R2

input -> output / L1 _ R1 || L2 _ R2 ... || Ln _ Rn
4.3.2. Left-arrow Rules

Left-arrow rules, using the `<` arrow, are similar to right-arrow rules but have a default upward orientation or reading.

$$z \leftarrow s / \text{[aeiou]} _ / \text{[aeiou]}$$

In a left-arrow rule, it is natural to think of the lower side as being the input side, and in this example the $s$ and the two contexts would need to match the lower-side input for the rule to fire. If this rule is applied “in an upward direction” to the lower-side input string $casa$, the upper-side output is $caza$. You can test this in the GUI in the usual way:

```bash
$leftrule = z \leftarrow s / \text{[aeiou]} _ / \text{[aeiou]};
test $leftrule;
```

and then enter a string like “casa” in the lower input field of the test window.

Experts in finite-state programming are often quick to point out that left-arrow rules are technically unnecessary. The composition

$$\begin{array}{c}
\text{result} = z \leftarrow s / \text{[aeiou]} _ / \text{[aeiou]} \\
\_ / \text{[aeiou]} \\
\_ / \text{[aeiou]} \\
\_ / \text{[aeiou]} \\
\_ / \text{[aeiou]} \\
\_ / \text{[aeiou]} \\
\end{array}$$

which composes a left-arrow rule “on top of” an existing machine bound to $fst$, could be accomplished equivalently, without left-arrow rules, by

- Writing the alternation as a right-arrow rule:
  $$s \rightarrow z / \text{[aeiou]} _ / \text{[aeiou]}$$

- Inverting the $fst$: $^\text{invert}(fst)$

---

7Recall, however, that in the OpenFst visualization of transducers, the upper side is always termed the “input side” and the lower side is always termed the “output side.”
• Composing the right-arrow rule on the lower side of the inverted $fst$:
  $^\text{invert}(fst) \circ s \rightarrow z / [aeiou]_-[aeiou]$

• And then inverting the result:
  $^\text{invert}(^\text{invert}(fst) \circ s \rightarrow z / [aeiou]_-[aeiou])$

While all that inversion makes perfect sense in theory, and works perfectly well in practice, many find it less than intuitive to write grammars that way. Users will have to judge for themselves whether to use the formally unnecessary left-arrow rules.

### 4.4. Optional Alternation Rules

A right-arrow rule using the $\rightarrow?$ or $\rightarrow?$ arrow (a normal right arrow followed by a question mark\(^8\)) applies optionally. Thus the rule

\[
s \rightarrow? z / [aeiou]_-[aeiou]
\]

maps the upper-side input string \textit{casa} to two outputs, \textit{casa} (representing the output when the rule matches and fires) and \textit{casa} (representing the output when the rule doesn’t fire). It maps the string \textit{isose} to four outputs: \textit{izoze}, \textit{izoze}, \textit{isose} and \textit{isose}.

Similarly, a left-arrow rule using the $\leftarrow?$ or $\leftarrow?$ arrow (a normal left arrow followed by a question mark) also applies optionally, but with a upward-biased orientation.

\[
\textit{z} \leftarrow? \textit{s} / [aeiou]_-[aeiou]
\]

\(^8\)In regular expressions, the question mark is a postfix operator indicating the optionality of the expression that precedes it. In a similar spirit, adding a question mark after a rule arrow signals that the mapping indicated by the rule is optional.
4.5. Maximum and Minimum Matching

Straightforward mapping rules will match and apply in all possible ways, with multiple outputs whenever there are multiple ways to match the input. The rule

\[ a^* b \rightarrow c \]

will map the input string \( aaaab \) downward to five outputs:

- \( aaaac \)
- \( aaac \)
- \( aac \)
- \( ac \)
- \( c \)

In practice, it is often useful to write rules that act as if a matching algorithm were moving left-to-right through the input string, always matching the maximum numbers of symbols whenever two or more matches are possible. The syntax for such rules adds \( \{\text{max}\} \) to the arrow:

\[
\begin{align*}
\text{upper } \{\text{max}\} & \rightarrow \text{lower} / \text{left } _ \text{right} \\
\text{upper } \leq \{\text{max}\} & \rightarrow \text{lower} / \text{left } _ \text{right}
\end{align*}
\]

Note that the \( \{\text{max}\} \) appears next to the input expression, which is the upper side in a right-arrow rule, and the lower side in a left-arrow rule.

Returning to the \( a^* b \rightarrow c \) example, adding \( \text{max} \) before the arrow causes the rule to map \( aaaab \) downward to a single output, \( c \).

\[ a^* b \{\text{max}\} \rightarrow c \]

// maps \( aaaab \) to \( c \)
It is important to understand that although the behavior of these rules appears to move left-to-right, and looks like some kind of algorithm is being performed at runtime, these rules, like all the other alternation rules, are compiled into finite-state transducers that encode a regular relation.

In some cases, it is also useful to write rules that again act as if a matching algorithm were moving left-to-right through the input string, but which always match the minimum rather than the maximum if multiple matches are possible. In such rules a {min} is added to the arrow:

\[
\begin{align*}
\text{upper } \{\text{min}\} & \rightarrow \text{lower} / \text{left } \_ \text{ right} \\
\text{upper } & \rightarrow \{\text{min}\} \text{ lower} / \text{left } \_ \text{ right}
\end{align*}
\]

[KRB: consider implementing similar rules with apparent right-to-left behavior, similar to the Xerox ->@ and >@ rules. 2013. Direct implementation of such rules would appear to require modifications of Hulden’s algorithms.]

### 4.6. Deletion Rules

[KRB: fill out this section]

A deletion rule maps something into nothing, i.e. into the empty string. The following rule maps p, t and k to the empty string, in the context at the end of a string.

\[
[pk] \rightarrow " / \_ \#"
\]

### 4.7. Epenthesis Rules

An epenthesis rule inserts symbols where no symbols were before. That it, it matches the empty string and maps it to something not empty. Recall that empty double quotes, "", denotes the empty string. The following epenthesis rule
maps the upper string *apto* to the lower string *apito*. Technically speaking, there are an infinite number of empty strings between any two adjacent symbols like *p* and *t*, and at the beginning and end of a string, and such rules might have been interpreted to force an infinite number of outputs. However, the intent of a programmer who writes "" -> i / p _ t is almost always to interpret the input as having only one empty string between symbols, and at the boundaries of a string, and the Kleene rule is automatically interpreted in this way.

### 4.8. Markup Rules

Markup rules match an input expression, map that input expression to itself in the output, and then “mark up” the output by surrounding the copied expression with symbols specified in the rule, e.g.

```
[aeiou] -> '<v>' ~~~ '</v>'
```

where the intent is to surround each vowel in the output with the XML-like symbols `<v>` and `</v>`. The `~~~` operator on the right side of the arrow is intended to represent the copy of whatever was matched in the input. Applied downward, this rule will map *unique* to

```
<v>u</v><v>i</v><v>n</v><v>q</v><v>u</v><v>e</v>
```

Markup rules can have contexts.

```
[aeiou] -> '<v>' ~~~ '</v>' / leftContext _ rightContext
```

Such markup rules are very similar to those in the Xerox xfst language. As we shall see, Kleene also offers transducer-style rules that perform markup in even more flexible and powerful ways.

---

9The Xerox replace rule intuitively written θ -> i || p _ t, where θ represents the empty string, is indeed interpreted this way and results in an error. Xerox epenthesis rules must be notated with special “dotted brackets,” e.g. [..] -> i || p _ t to avoid the error.
4.9. Two-Level Rule Contexts

Using Huldén's algorithms, it became possible to implement alternation rules with two-level contexts.\textsuperscript{10} Compare the following examples:

\[
\text{a} \rightarrow \text{b} / \text{b } _-
\]

is a rule with a one-level context, and it maps baaa downward to bbba, while

\[
\text{a} \rightarrow \text{b} / (\ast): \text{b } _2-
\]

is a rule with a two-level context that maps baaa to bbba. The _2_ syntax, which can also be typed as \_{\_}, the Unicode DOUBLE LOW LINE, indicates that the context or contexts are to be interpreted as having two levels. Here the left context is \( \ast \) (the Universal Language) on the upper side and \( \text{b} \) on the lower side.

[KRB: explain, step by step, how the only possible output is bbba]

This kind of rule, where the left-context matches on the lower side of the relation, appears to generate its own left context as it moves along, resulting in a kind of left-to-right domino effect. However, it is important to realize that there is no algorithm or process being performed; the rule, like all alternation rules, is compiled into a static finite-state transducer, a data structure that encodes a relation between two regular languages. The output only appears to reflect a kind of left-to-right domino effect that generates its own lower-side left contexts as it moves along.

In natural-language phonetics and morphology, such rules are often used to model a phenomenon known as vowel harmony. [Give examples.]

In a similar way, note that

\[
\text{a} \rightarrow \text{b} / _- \text{b}
\]

maps aaab to aabb, while

\textsuperscript{10}Two-level contexts were not possible in Xerox Replace Rules.
maps aaab to bbbb. Here again, the \_2\_ operator indicates that the context or contexts are to be interpreted as having two levels, and here the right context is .\* on the upper side and b on the lower side. The rule with a two-level right context appears to operate right-to-left, generating its own lower-side right context as it moves along, outputting bbbb in a kind of right-to-left domino effect. Again, the rule really just compiles into a finite-state transducer that encodes a regular relation, and there is no runtime algorithm or process involved. Such rules have been found useful to model a phenomenon known as umlaut.

The following example of maximal matching involves the insertion of a hyphen to mark a syllable-boundary marker, where a syllable starts with a consonant, continues with one or more vowels and ends with zero or more consonants, provided that the right context starts with one consonant and a vowel.\textsuperscript{11}

\begin{verbatim}
// Define a set of consonants for your language
// (modify as necessary)
$C = [bcdfghjklmnprstvyz] ;

// Define a set of vowels for your language
$V = [aeiou] ;

// Define a syllable as the maximal match of
// one consonant, followed by
// one or more vowels, followed by
// zero or more consonants, provided that
// it is followed by one consonant and a vowel.
// Insert a hyphen after such a matched syllable.
$Syll = $C $V+ $C* {max}-> ~~~ "-" / _ $C $V ;
\end{verbatim}

\textsuperscript{11}Thanks to Lauri Karttunen for this example.
// Define a convenient function that
// takes an input string,
// apples $Syll to the string in a downward direction, and
// returns the lower side of the resulting relation.
$^Syllabify($str) {
    return $^lowsider($str _o_ $Syll) ;
}

// Syllabify an input string. Use double quotes
// to literalize the space.
$words = $^Syllabify("lauri karttunen") ;
print $words ;
// => lau-ri kart-tu-nen : 0.0

// Syllabify another word, bypassing the $words variable.
// No double quotes are needed here.
print $^Syllabify(continental) ;
// => con-ti-nen-tal

This modeling of syllabification won’t work for all languages, but it is
often a useful starting point.

### 4.10. Parallel Rules

#### 4.10.1. Vertical Derivations vs. Parallel Rules

So far we have combined multiple rules in vertical derivations (cascades),
and this approach is often natural and intuitive. In a derivation, the
rules are combined using the composition operation. When constructing
a derivation, the relative orders of the rules can be critical, and there are
classic cases where rule ordering causes challenges that can be overcome
only with ugly kludges. For example, consider the case where we want a
rule (or set of rules) to take the input string \textit{abba} and return the string \textit{baab}; and for the input \textit{baab}, we want the output \textit{abba}. Similarly, \textit{aaaaa} should map to \textit{bbbb}, and \textit{ababaa} should map to \textit{bababb}. In short, we want every \textit{a} to be mapped to \textit{b}, and we want every \textit{b} to be mapped to \textit{a}. Instinctively, we first imagine two trivial rules:

\begin{verbatim}
\texttt{a -> b}
\end{verbatim}

and

\begin{verbatim}
\texttt{b -> a}
\end{verbatim}

As a first attempt, we try to compose the two rules with the \texttt{a -> b} rule above the \texttt{b -> a} rule, and we define a trivial function $^\text{swap}$ to facilitate testing.

\begin{verbatim}
$deriv = a -> b
     _o_
    b -> a ;
$^\text{swap} ($word) {
    return $^\text{lowerside}($word _o_ $deriv) ;
}
\end{verbatim}

\begin{verbatim}
print $^\text{swap}(abba) ;
// => aaaa
\end{verbatim}

To our disappointment, when we apply $^\text{swap}$ to \textit{abba}, the output is not the desired \textit{baab} but rather the string \textit{aaaa}. We then try the opposite ordering of the two rules

\begin{verbatim}
$deriv = b -> a
     _o_
    a -> b ;
$^\text{swap} ($word) {
\end{verbatim}
\[
\text{return } \text{\textsuperscript{lowerside}}(\text{\textsuperscript{word o deriv}}) ;
\]

print \text{\textsuperscript{swap}}(abba) ;
// => bbbb

and the output this time is bbbb, again not the desired baab. The problem can be traced by imagining the two rules applying separately. In the first case, with the ordering

\[
\text{deriv} = a \rightarrow b \\
\text{o} \\
b \rightarrow a ;
\]

the first rule will map abba to bbbb, and then the second rule will map bbbb to aaaa. In the second case, with the ordering

\[
\text{deriv} = b \rightarrow a \\
\text{o} \\
a \rightarrow b ;
\]

the first rule will map abba to aaaa, and then the second rule will map aaaa into bbbb. Obviously, one rule is undoing the work of the other, and simple rule re-ordering will not solve this problem.

In ordered derivations, such problems can be solved by the use of a temporary symbol—we will use the multicharacter symbol [ TEMP]—in the following way

\[
\text{deriv} = a \rightarrow '[TEMP]' \\
\text{o} \\
b \rightarrow a \\
\text{o} \\
'[TEMP]' \rightarrow b ;
\]
This works as desired, but it is far from beautiful or intuitive. Instinctively, we just want to write our two simple rules, $a \rightarrow b$ and $b \rightarrow a$, and specify somehow that they are to be applied in parallel, simultaneously, rather than one ordered after the other.

4.10.2. The Built-In Parallel Rule-Compilation Function

The abba Example

Huldén's algorithms allow alternation rules to be compiled in parallel with complete freedom [KRB: test and confirm this]. Unlike xfst Replace Rules, which can be compiled in parallel only if they have the exact same arrow operator, Kleene allows rules of any arrow type to be compiled in parallel. Unlike traditional Two-Level alternation rules, which are compiled in parallel but are limited to controlling the mapping of one symbol to one symbol (the so-called “concrete pairs”), Kleene allows rules with arbitrary-length mappings to be compiled in parallel.

The overt way to indicate that a set of rules is to be compiled in parallel is to enclose them in the $^\parallel()$ built-in function, separated by commas. Our problematic $abba$ example shown just above can be formulated straightforwardly as

$r = ^\parallel(a \rightarrow b, b \rightarrow a) ;$

\[
\text{\#^swap($word$) { return ^\lowerside($word$ _o_ $r$) ; }}
\]
print $\texttt{^swap(abba)}
// => baab

print $\texttt{^swap(baab)}
// => abba

print $\texttt{^swap(ababaa)}
// => bababb

By compiling the a -> b and b -> a rules in parallel, they apply simultaneously and give us the desired results. The resulting machine will map abba to baab, baab to abba, etc.

Proponents of parallel rules, notably Professor Kimmo Koskenniemi of the University of Helsinki, argue that parallel rule compilation is inherently simpler than ordered derivations. They emphasize the difficulties of rule ordering and point out that a developer must understand and somehow keep track of all the intermediate levels in a derivation. However, non-trivial sets of rules applied in parallel can conflict in various ways that are not at all easy for most developers to understand and fix, especially where the rule set includes deletions or epentheses. In addition, writing rules in parallel also requires the careful use of two-level contexts, often specifying that one side of the context must match while the other side can be anything (the universal language).

Consider the case of the kaNpat example, previously encoded as a cascade

\$$r = N -> m / _ (p| b) \$$
\$_o_
\$$p -> m / m _ ;$$
test \$r ;$

Using parallel rules, this example can also be encoded equivalently as
\[ r = \text{\texttt{\textasciitilde parallel}}( \text{\texttt{N \to m / \_ (p|b)}}, \text{\texttt{p \to m / (\_):m \_2_})} \];

test $r$;

where the \texttt{p \to m} rule now has a left context \texttt{(\_):m} that matches anything (the universal language) on the upper side but \texttt{m} on the lower side. This is effectively the way of saying that the \texttt{m} of the left context could be an original input \texttt{m}, or it could be something (a symbol, a sequence of symbols, or perhaps even the empty string) that is being mapped to \texttt{m} by any one of the other rules being compiled in parallel.

Consider also the mapping of Brazilian-Portuguese \textit{time} to \textit{tSimi} and \textit{parte} to \textit{partSi}, done earlier using a cascade of rules.

\[ r = \text{\texttt{e \to i / \_ \#}} \]
\[ \text{\texttt{t \to tS / \_ i}} \];

The same mappings can be performed by two rules compiled in parallel.

\[ r = \text{\texttt{\textasciitilde parallel}}( \text{\texttt{e \to i / \_ \#}}, \text{\texttt{t \to tS / \_2_ (\_):i})} \); 

In this case, the right context of the \texttt{t \to tS} rule must be written as the two-level \texttt{(\_):i}, indicating that the \texttt{i} could be an original input \texttt{i} or something mapped to \texttt{i} by another parallel rule. Writing such two-level contexts correctly (neither too loose nor too specific) is not always easy for developers.

Consider also the following example, which (in a rule with one-level contexts) maps \texttt{a} to \texttt{b} in the context between \texttt{c} and \texttt{d}, and also, in a rule with two-level contexts, maps \texttt{d} to \texttt{e} after an \texttt{a} that has been simultaneously mapped to \texttt{b} by the first rule. These rules compile in parallel to a machine that maps \texttt{cad} to \texttt{cbe}.

\[ r = \text{\texttt{\textasciitilde parallel}}( \text{\texttt{a \to b / c \_ d}}, \text{\texttt{d \to e / a:b \_2_}}) \];
In general, every rule compiled in parallel must “be aware” of what all the other rules in the parallel-rule set are doing. More precisely, the writer of each rule must be aware of what all the other rules in the parallel-rule set are doing simultaneously. Where there are only two or a few rules, as in these examples, the challenge may seem trivial; but in real-life systems, there may easily be dozens of rules, and conceiving the effect of all of them operating simultaneously is challenging, to say the least.

There is no mathematical basis for choosing to write rules in a vertical cascade vs. in parallel—the result is always regular in power, encodable as a finite-state transducer. Some examples will seem easier to encode one way or the other, and different developers will have differing conceptual capabilities and tastes. Kleene allows rule sets to be encoded as derivations, in parallel, or in a mixture of the two approaches.

Note that $\parallel$ is a special built-in function\(^{12}\) that requires that its arguments be in full rulesyntax. The following example, which tries to compile the rules separately into finite-state machines, and then pass the machines as arguments to $\parallel$ is syntactically and semantically illegal. In order to compile rules in parallel, the Huldén algorithms need to access and compile all of the original rule parts, combining them together in non-trivial ways. If a rule has been pre-compiled into a finite-state machine, the original parts of the component rules are no longer identifiable or extractable.

```
$\text{rule1} = e \rightarrow i / _ # ; \\
$\text{rule2} = t \rightarrow tS / _2_ (.*) : i ;
```

```
// this is ILLEGAL
$r = \parallel ( \text{rule1} , \text{rule2} ) ;$
```

```
// while this is legal
$r = \parallel ( e \rightarrow i / _ # , t \rightarrow tS / _2_ (.*):i ) ;$
```

\(^{12}\)\$\text{parallel()}$ processes its arguments in a particular way and cannot be aliased. It is thus similar to a Lisp special form.
4.10.3. Rules with Where-Clauses

The Devoicing Example

Kleene alternation rules can also have where-clause directives that define the settings of local variables inside the rules. The following rule

\[ r = \text{\$voiced} \rightarrow \text{\$unvoiced} / _ # \\
\text{where \$voiced \_E_ \$(@(b, d, g)}, \\
\text{\$unvoiced \_E_ \$(@(p, t, k}) \}; \]

models the phenomenon where voiced consonants become devoiced at the end of a word. (The rule is written on multiple lines to be more readable, but this is not required.) In German, as a real-world example, the word written Tag is pronounced Tak, the word written Hund is pronounced Hunt, etc. The rule above is effectively expanded to three rules that are compiled in parallel, equivalent to

\[ r = \text{\$}^\parallel( b -> p / _ #, \\
d -> t / _ #, \\
g -> k / _ # ) ; \]

In this case, the where ... clause defines two local variables, \$voiced and \$unvoiced, each of which is bound to a value, one at a time, from a supplied list of finite-state machines. The finite-state machines can be arbitrarily complex, though they must conform to any semantic limitations inherent to rules; in this example, the local variables \$voiced and \$unvoiced appear on the left-hand-side of the rule, where the expressions must denote regular languages (not relations). By default, as in this case, the local variables are bound, during interpretation, to the list values in a left-to-right “matched” fashion such that when \$voiced is bound to its first value b, \$unvoiced is bound to its first value p, and when \$voiced is bound to d, \$unvoiced is bound to t, etc. This matched assignment behavior requires that the two list expressions have exactly the same length, and Kleene will throw an exception if they are not.
Where-clauses can define one, two or more variables that are interpreted as being local to the rule. Each local variable in the `where`-clause is followed by the “element of” operator, which can be the \( \in \) character (Unicode `ELEMENT OF`, 0x2208) or the ASCII equivalent `_E_`. The syntax `@(...)` is the normal Kleene syntax for a literal anonymous list of finite-state-machine values (explained in chapter 9). After the \( \in \) or `_E_` can appear any Kleene expression whose value is a list of finite-state machines, including variables and even function calls that return such a list.

```
$@voicedstops = @b, d, g ;
$@unvoicedstops = @p, t, k ;

$r = $voiced -> $unvoiced / _#
   { where $voiced _E_ $@voicedstops,
     $unvoiced _E_ $@unvoicedstops } ;
```

The following rule, which doubles all vowels, e.g. mapping `unido` to `uuniidoo`, calls `@^getSigma()`, which is a pre-defined function that calculates the alphabet (the “sigma”) of its FSM argument and returns it as a list of finite-state machines.

```
$r = $vowel -> $vowel $vowel
   { where $vowel _E_ $@^getSigma(aeiouAEIOU) } ;
```

**The Caesar Cipher Example**

Julius Caesar employed a simple-substitution cipher wherein each Latin letter was replaced by the letter three steps along in the alphabet, e.g. A was replaced by D, B was replaced by E, C was replaced by F, etc. At the end of the alphabet, the mapping “wrapped around” to the beginning so that X was replaced by A, Y by B and Z by C. Like the `abba` example

---

13The Kleene interpreter literally pushes to a new frame/scope while compiling a rule with `where`-clauses, so that resetting the values of the local rule variables does not effect any variables outside the rule that happen to have the same names.
above, we intuitively just want all the letter substitutions to happen simultaneously, without mutual interference, and this is another classic example where parallel compilation is desirable. The following script uses the common convention of treating the “plaintext” as consisting only of lowercase letters, and the “ciphertext” as consisting only of uppercase letters. Arbitrarily, the script is written to generate (apply downwards) from a plaintext string to the corresponding ciphertext strings.

```plaintext
$@plainletter = $@(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z) ;
$en = $pl -> $cl {where $pl _E_ $@plainletter,
    $cl _E_ $@cipherletter} ;

$^encipher($plain) {
    return $^lowerside($plain _o_ $en) ;
}

pr $^encipher("attack at dawn") ;
// => DWWDFN DW GDZQ
```

The script begins with the definition of $@plainletter as a list that contains all the plaintext letters in normal alphabetical order. $@cipherletter is then defined as a list of ciphertext letters starting with D, reflecting Caesar’s original offset. $en is a rule that maps each plaintext letter to its corresponding ciphertext letter, and $^encipher is a little function that composes the $en rule “under” the input string and then extracts and returns the lower side (the output) of the result. The result of $^encipher("attack at dawn") is the ciphertext DWWDFN DW GDZQ.

There are two ways to implement decipherment of the same cipher. Following the same pattern, we can define $de as a rule that maps each
ciphertext letter to its plaintext equivalent, and the $^\text{decipher}$ function facilitates the testing to map $DWWDFN DW GDZQ$ back into the original attack at dawn message.

\[
\text{de} = \text{cl} \rightarrow \text{pl} \{\text{where cl}_E \text{cipherletter}, \pl_E \text{plainletter}\};
\]

\[
\text{^decipher}(\text{cipher}) \{
    \text{return } \text{^lowerside}(\text{cipher}_o \text{de});
\}
\]

\[
\text{pr } \text{^decipher("DWWDFN DW GDZQ")};
// => attack at dawn
\]

However, because the original $\text{en}$, defined as

\[
\text{en} = \text{pl} \rightarrow \text{cl} \{\text{where pl}_E \text{plainletter}, \cl_E \text{cipherletter}\};
\]

maps downward from plaintext to ciphertext, and because we remember that regular transducers are in fact bidirectional, it is tempting to avoid the definition of $\text{de}$ and just use $\text{en}$ to implement $\text{^decipher}$ thus

\[
\text{^decipher}(\text{cipher}) \{
    \text{return } \text{^upperside}(\text{en}_o \text{cipher});
\}
\]

that is, to compose the cipher string on the lower side of $\text{en}$ and return the upper side of the result. However, if we try to do that, each letter on the lower side, such as $D$, will map upward to itself or to its plaintext equivalent, here $a$, and even for this short text there will be over a hundred outputs, reflecting all the possible upward mappings. Recall that such a right-arrow rule has a downward orientation: it states that an $a$ on the upper side maps to a $D$ on the lower side, but it doesn’t say that a $D$ on
the lower side has to map to an a on the upper side. Because we defined the problem in a way that the intended upper alphabet is all lowercase, and the intended lower alphabet is all lowercase, we can fix this particular example by simply constraining $en$ not to contain any ciphertext (uppercase) letters on the upper side.

\[
\text{$en = \neg \text{^contains([A-Z])} \_o_ \text{$pl \rightarrow $cl \{where $pl \_E_ \text{@plainletter,} \text{$cl \_E_ \text{@cipherletter}\}} ;}
\]

\[
\text{^decipher($cipher) \{ return \text{^upperside($en \_o_ \text{$cipher}) ;}
\}
\]

pr $^\text{decipher("DWWDFN DW GDZQ") ;} // => attack at dawn

This is a good reminder that while alternation rules compile into transducers, and transducers are bi-directional, a right-arrow rule constrains the mapping only from the upper side to the lower side, not vice versa. Conversely, a left-arrow rule constrains the mapping only from the lower side to the upper side, not vice versa.

**Where Mixed Clauses**

Kleene also supports the \texttt{where mixed} keyword, which causes the interpreter to compute the Cartesian Product of all the list values and assign the local variables to each of the results in turn. For \texttt{where mixed}, the lists of values do not have to have the same length. For example, in the rule

\[
\text{$r = $foo -> $bar \_o_ \{ where mixed $foo \_E_ \text{@(a, b, c),} \text{$bar \_E_ \text{@(y, z) } \}} ;}
\]
the Cartesian product of the values contains the six ordered pairs (a, y),
(a, z), (b, y), (b, z), (c, y) and (c, z), and so the single rule is equivalent to
the following six rules compiled in parallel:

\[
r = \parallel(a -> y,
            a -> z,
            b -> y,
            b -> z,
            c -> y,
            c -> z);
\]

and either rule will map upper-side input cab to yyy, yyz, yzy, yzz, zyy, zyz,
zyy and zzz.

Kleene also supports the where matched keyword, equivalent to where,
if the programmer desires to make the matched-assignment behavior more explicit.

\[
r = \text{voiced} -> \text{unvoiced} / _ #
   \{ \text{where matched} \text{voiced} _E_ @(b, d, g),
       \text{unvoiced} _E_ @(p, t, k) \};
\]

In addition, one can write rules with multiple where-clauses, potentially
mixing where (or where matched) and where mixed clauses, and the
interpreter will compute the Cartesian product of values among the multi-
ple where clauses. As the semantics quickly become opaque to the human
programmer, multiple where clauses are best avoided.

4.10.4. Rules where the Input can Match the Empty String

We have already examined epenthesis rules such as

"" -> i / p _ t

where the upper-side input expression is the empty string. A similar case
is where the input expression denotes a language that includes the empty
string, e.g.
a* \rightarrow x / l r

Note that a* denotes a language that includes the empty string. Because rules match however they can, such a rule will match \textit{laaar} and map it to \textit{bxr}; and it will also match \textit{lr} and map it to \textit{bxr}, mapping nothing to something, just like an epenthesis rule. To handle such cases Kleene automatically interprets

a* \rightarrow x / l r

as two rules compiled in parallel,

\texttt{\$^\parallel( (a* - "") \rightarrow x / l r, "" \rightarrow x / l r)}

but this is all hidden from the programmer.\textsuperscript{14}

\section*{4.11. Transducer-Style Rules}

\subsection*{4.11.1. Traditional Rules vs. Transducer-Style Rules}

Transducer-style rules were an innovation in the Foma language\textsuperscript{15} of Dr. Måns Huldén, inspired by a suggestion by Gerdemann and van Noord

\textsuperscript{14}Xerox Replace Rules for epenthesis require the “dotted bracket” notation as the input expression, e.g. \texttt{[..] \rightarrow \textit{i}}, or surrounding the input expression, e.g. \texttt{[. a*. ] \rightarrow \textit{x}}, in cases where the input language includes the empty string. This notation causes the Xerox xfst interpreter to treat the input as having only one empty string where, mathematically, an infinite number of empty strings occur. The dotted-bracket notation is easily forgotten by programmers, and the failure to use it, in cases where the input expression is or includes the empty string, results in annoying errors. Kleene has no need for a similar notation because it automatically recognizes such rules and interprets them as if there were only one empty string before and after each input symbol.

\textsuperscript{15}https://code.google.com/p/foma/
With help from Huldén, they were added to Kleene in January 2014.

In the traditional alternation rules that we have seen so far, the left-hand side of the rule always contains upper and lower expressions that must denote regular languages (not relations) and which are separated in the rule syntax with one of the many rule-arrow operators. A right-arrow rule has a default downward bias, meaning that we typically think of the input matching on the upper side of the relation, and the output being read off the lower side. Conversely, a left-arrow rule has a default upward bias, meaning that we typically think of the input matching on the lower side of the relation, and the output being read off the upper side.\(^\text{17}\)

// a traditional right-arrow rule, with a default
// downward bias
// left-hand side / right-hand side
upper -> lower / left _ right

// a traditional left-arrow rule, with a default upward bias
// left-hand side / right-hand side
upper <- lower / left _ right

In some cases, it can be more intuitive or convenient to express the mapping of a rule not as separate upper and lower expressions, but rather as a single combined transducer expression, e.g. \((upper):(lower)\). Such rules


\(^{17}\)In Kleene, as in the Xerox Finite State Tools, we typically visualize transducers as having an upper and a lower side, and we emphasize their bidirectionality: they can be applied to input either in an downward direction (“generation”) or in an upward direction (“analysis”). Recall that in the OpenFst tradition, what Kleene and Xerox call the upper side is always termed the “input” side, and the lower side is always termed the “output” side.
are known in Kleene as left-hand-side transducer rules or just transducer-style rules. As we shall see, some rules with where clauses can be expressed more readably as transducer-style rules, and transducer-style rules provide a more general way to express multiple insertions than traditional markup rules. As we shall see, there are even some cases where a potentially useful mapping that is easily expressible in a transducer-style rule cannot be re-expressed as a rule with separate upper and lower expressions.

4.11.2. Simple Mapping Rules in the Transducer Style

Right-Arrow Transducer-Style Rules

We start with some simple examples involving right-arrow mapping rules that can be expressed either with separate upper and lower expressions, or equivalently in the transducer style. After that we will look at left-arrow rules and then progress to more complicated examples that better illustrate the power and generality of the transducer-style rules.

The simplest kind of traditional right-arrow mapping rule maps an upper expression, e.g. a, downward to a lower expression, e.g. b.

```
// a traditional right-arrow rule
// upper -> lower
$downward = a -> b ;
```

This trivial example can be re-expressed as an equivalent transducer-style rule by moving the default output expression, here b, over to the default input side of the arrow, and combining it with the default input expression, here a, with the crossproduct operator to form the transducer expression a:b. The key point here is that there is a single expression, denoting a transducer, on the left side of the arrow.

```
// an equivalent transducer-style rule
$downward = a:b -> ;
```
More generally, if you have any right-arrow rule with a left-hand side that fits this template

\[ \text{upper} \rightarrow \text{lower} \]

it can be converted into an equivalent transducer-style rule by combining the \textit{upper} and \textit{lower} expressions into the combined transducer expression \((\text{upper}): (\text{lower})\).

\((\text{upper}): (\text{lower}) \rightarrow \)

Recall that the cross-product operator, \(:\), is of high precedence, so the parentheses may be necessary to ensure that the \textit{upper} and \textit{lower} expressions are interpreted correctly.

These are trivial examples, and at this point it is impossible to see the point of the exercise. Rest assured that transducer-style rules will soon be shown to be more interesting. For now, note the following points, which hold for both left-arrow and right-arrow rules:

- In a transducer-style rule, the traditionally separated upper and lower expressions are combined into a single expression that denotes a transducer.

- The transducer expression appears syntactically where the default/intuitive \textit{input} expression appears in traditional mapping rules.

- The rule arrow, as always, indicates the direction of the default/intuitive mapping: a right-arrow indicates a default downward bias, and a left arrow indicates a default upward bias.

- The rule arrow, in a transducer-style rule, always points at nothing.

Transducer-style rules can have contexts and \texttt{where} clauses like all the traditional rules; we leave them out here to simplify the examples.

Note the following equivalences for right-arrow rules:
// epenthesis

// traditional
$epenthesis = "" -&gt; b ;

// equivalent transducer-style
$epenthesis = "":b -&gt; ;

// optional mapping

// traditional
$opt = a -&gt;? b ;

// equivalent transducer-style
$opt = a:b -&gt;? ;

// max

// traditional
$max = ab* {max} -&gt; c ;

// equivalent transducer-style
$max = (ab*):c {max} -&gt; ;

// max optional

// traditional
$maxopt = ab* {max} -&gt;? c ;

// equivalent transducer-style
$maxopt = (ab*):c {max} -&gt;? ;
// min

// traditional
$min = a*b* {min} -> c ;

// equivalent transducer-style
$min = (a*b*):c {min} -> ;

// min optional

// traditional
$min = a*b* {min} ->? c ;

// equivalent transducer-style
$min = (a*b*):c {min} ->? ;

Note again that for right-arrow transducer-style rules, the arrows all point at nothing. If the rule has contexts or where clauses, they appear immediately after the right arrow, e.g.

// a transducer-style rule with a context
$rule1 = a*b -> / left _ right ;

// a transducer-style rule with a where clause
$rule2 = $up:$low -> { where $up _E_ $@(a, b, c),
    $low _E_ $@(x, y z) } ;

// a transducer-style rule with a context and a where clause
$rule3 = $up:$low -> / _ # { where $up _E_ $@(b, d, g),
    $low _E_ $@(p, t, k) } ;
Left-Arrow Transducer-Style Rules

For left-arrow rules, recall that they have a default upward bias; so we usually think of the transducer specified by a left-arrow rule as being applied in an upward direction, with the input matched on the lower side, and the output being read off the upper side. To turn a trivial left-arrow rule such as

```
// traditional left-arrow rule
$upward = a <- b ;
```

into an equivalent transducer-style rule, we move the output expression, here \( a \), over to the input side expression, here \( b \), and combine them with the crossproduct operation to create the transducer expression \( a:b \).

```
// equivalent transducer-style left-arrow rule
$upward = <- a:b ;
```

Note that in the resulting left-arrow transducer-style rule, the left arrow points at nothing. (Recall that in right-arrow transducer-style rules, shown above, the right arrows also point at nothing.) As always, the left arrow indicates the upward bias of the rule, here mapping up from lower-side \( b \)s upward to upper-side \( a \)s.

Note the following equivalences between traditional left-arrow rules and their transducer-style left-arrow equivalents:

```
// epenthesis

// traditional
$epenthesis = a <- "" ;

// equivalent transducer-style
$epenthesis = <- a:"" ;
```
// optional mapping

// traditional
$opt = a <-? b ;

// equivalent transducer-style
$opt = <-? a:b ;

// max

// traditional
$max = c <- {max} ab* ;

// equivalent transducer-style
$max = <- {max} c:(ab*) ;

// max optional

// traditional
$maxopt = c <-? {max} ab* ;

// equivalent transducer-style
$maxopt = <-? {max} c:(ab*) ;

// min

// traditional
$min = c <- {min} a*b* ;

// equivalent transducer-style
$min = <- {min} c:(a*b*) ;
// min optional

// traditional
$min = c <-? {min} a*b* ;

// equivalent transducer-style
$min = <-? {min} c:(a*b*) ;

4.11.3. Transducer-Style Rules vs. Where-Clause Rules

In cases where a set of parallel rules differ only in the input and output expressions, transducer-style rules can provide the most readable syntax. Consider the following three equivalent rule definitions for devoicing consonants at the end of a word, involving explicitly parallel rules (using the $\parallel()$ function), a rule with a where clause, and a transducer-style rule.

// explicit parallel rules
$explicitPar = $\parallel( b -> p / _ #,
  d -> t / _ #,
  g -> k / _ # ) ;

// equivalent rule with a where clause
$wherePar = $voiced -> $unvoiced / _ #
{ where $voiced _E_ $@(b, d, g),
  $unvoiced _E_ $@(p, t, k) } ;

// equivalent transducer-style rule
$transducerPar = b: p | d: t | g: k -> / _ # ;

In this case, most people would prefer to write and read the transducer-style rule. If, however, a set of parallel rules involved variations in the contexts, a where clause rule would be the best option, and it could not be re-expressed as a transducer-style rule.
4.11.4. Transducer-Style Rules vs. Traditional Markup Rules

Transducer-style rules provide a more general and powerful alternative to the markup rules previously presented. First recall that an epenthesis rule maps the empty string into some non-empty string, e.g. to insert an i "out of nowhere" between a p and an immediately following t:

\[ \text{ep} = "" \rightarrow i / p _ t; \]

// maps "apto" downward to "apito"

// equivalent transducer-style rule
\[ \text{ep} = ":i -> / p _ t; \]

Markup rules, presented in section 4.8, page 109, are essentially double-epenthesis rules, matching an input expression, copying it to the output, and epenthetically inserting symbols before and output the copied output. Recall that the operator ~~~ denotes the copy of the input expression. The following rule puts XML-like <v> and </v> symbols around each vowel.

\[ \text{markup} = (a|e|i|o|u) \rightarrow '<<v>' ~~~ '</v>'; \]

// equivalent
\[ \text{markup} = [aeiou] \rightarrow '<v>' ~~~ '</v>'; \]

// maps "rake" to "r<v>a</v>k<v>e</v>" where <v> and </v>
// are multi-character symbols

The transducer-style equivalent rule does not use the ~~~ operator, and it is arguably more readable.

// equivalent transducer-style rule to surround vowels
// with <v> and </v>
Note that these rules will match a vowel, copy it to the output, and insert “from nothing” a $<v>$ on the left and a $</v>$ on the right.

### 4.11.5. Transducer-Style Rules Not Expressible as Traditional Rules

**Generalized Epenthesis**

Because the transducer expression on the left-hand side of a transducer-style rule can denote any two-projection relation, i.e. any relation expressible in Kleene and OpenFst, it should be obvious that we are no longer limited to the single insertions of tradition epenthesis rules or the double epenthesis of traditional markup rules. We can now write a transducer-style rule to insert three or more epenthetical sequences.

```plaintext
$vow = [aeiou] ;
$con = [ptkbdgszlrwy] ;

// wrap each vowel with $<v>$ and $</v>$, and wrap each
// consonant(s)-vowel(s) sequence in $<cv>$ and $</cv>$

$generalMarkup =
    "":"<$cv>" $con+ ("": '<v>' $vow "": '</v>')+ "": '</cv>' {max}-> ;

// maps "stabile" to
//    "<cv>st<v>a</v></cv><cv>b<v>i</v></cv><cv>l<v>e</v></cv>"
// maps "wuuti" to
//    "<cv>u<v>u</v></cv><v>u</v></cv><cv>t<v>i</v></cv>"
```
Such a rule cannot be re-expressed as a traditional epenthesi

**Markup with Mapping**

Whereas a traditional Xerox-inspired markup rule like

```
$markup = cat -> l ~~~ r ;
```

matches the input expression `cat`, copies it to the output, and surrounds it on the output side with `l` and `r`, a transducer-style rule can match an input expression, map it to something different, and surround the output with arbitrary symbols.

```
$generalMapAndMarkup = "":l (cat):(feline) ":r -> ;
```

// maps "cat" to "lfeliner"
// maps "scatter" to "slfelinerter"

Again, such a transducer-style rule cannot be re-expressed as a traditional markup rule.

**More Transducer-Style Rules that Cannot be Expressed as Traditional Rules**

The following example, from Måns Huldén, copies a core expression, $C^*$, from the input to the output, but also matches expressions on the left and right and maps those bracketing expressions to something different.

```
$rule = a: e C* a: e -> / # _ ;
```

This rule maps `aa` to `ee`, `aCa` to `eCe`, `aCCa` to `eCCE`, etc. and cannot be re-expressed as a traditional mapping rule.

In conclusion, the new transducer-style rules offer a clean notation that is often more readable, general and powerful than traditional mapping rules that separate the *input* and *output* expressions.
Chapter 5

Examples without Weights

5.1. Introduction

The purpose of this chapter is to present some non-trivial examples that do not involve weights. This is definitely work in progress, and examples will be added, corrected and expanded in future releases.

5.2. Spanish Morphology

5.2.1. Spanish Verbs

Mind-Tuning

I chose Spanish as an initial example because Spanish is widely studied and extensively documented, and I have some basic familiarity with it. I will start with modeling Spanish verbs—in particular, the highly productive regular first conjugation.

The various Spanish verb classes, and their conjugations, are documented in numerous published books\(^1\) and are even available on the Internet. See books such as *501 Spanish Verbs* and especially the verb-conjugation books published by Larousse and Bescherelle.
I hope eventually to offer a Kleene script, downloadable from www.kleene-lang.org, that handles all Spanish verbs. I emphasize that my approach in the following script is only one of many.

By long lexicographic convention, Spanish verbs are listed in dictionaries under the infinitive form, e.g. the infinitive *amar*, meaning “to love,” is the conventional dictionary citation form. In Spanish and other languages, the traditional dictionary citation forms should not be confused with base-forms or roots. If anything, the root of the “love” verb is just *am*, to which various suffixes are attached. The infinitive citation form, *amar*, is really just *am* with an *ar* suffix; it is just one of the many conjugated forms of the verb. However, in a bow to tradition, we will initially list verbs in their infinitive forms, and analyses of verbs will show the infinitive form to facilitate lookup in traditional dictionaries. The script will use alternation rules to strip the infinitive suffixes from the lower side of the FSMs before adding other suffixes to implement the various conjugated forms.

**Modeling Spanish First-conjugation Verbs**

A verb class traditionally called the *first conjugation* contains thousands of regular verbs whose infinitive forms end in *-ar*, including *amar* “to love,” *cantar* “to sing,” and *hablar* “to speak.” My Larousse *Conjugación* book calls this verb class number 3, and I will use the Larousse numbers (with some modification) to distinguish the various verb-conjugation classes. We can start by collecting a sampling of this class of verbs in a simple union of the infinitive forms.

```plaintext
// "First Conjugation" class 3
$V3CitationForms = amar | cantar |
```

---

2See, for example, http://www.conjugacion.es for the conjugations of Spanish verbs.
cortar |
hablar |
similar ; // and continue to add hundreds more

See http://www.www.conjugacion.es/del/verbo/amar.php, the Larousse or Bescherelle books, or any of the other sources for a table showing all the conjugated forms. We will limit ourselves for now to the single-word conjugations, ignoring composite multi-word conjugations. Clitic pronouns will also be ignored for the time being.

These charts contain conjugation groups, almost always of six forms, showing the

1st person singular
2nd person singular
3rd person singular
1st person plural
2nd person plural
3rd person plural

forms for present indicative, preterite imperfective, preterite perfect, future indicative, conditional, etc. For example, the six present indicative forms of amar are

amo
amas
ama
amamos
amáis
aman

showing that amo (“I love”) is the first-person singular present indicative form of amar, amas (“thou lovest”) is the second-person singular, ama (“he/she loves”) is the third-person singular, amamos (“we love”) is the first-person plural, amáis (“you love”) is the second-person plural, and
aman ("they love") is the third-person plural. The root is am, and the six suffixes for this group are pretty obviously o, as, a, amos, áis and an. We want to build a morphological analyzer that will eventually accept any orthographical verb string, e.g. amo, and return a string that includes the infinitive amar, the information that it is a verb, and the information that it is the first-person singular present indicative form. We will employ a number of multi-character symbol tags to convey part-of-speech, person, number, tense, aspect and mood information. Our FSM will include paths like the following, ignoring alignment and epsilons,

Upper: amar[VERB][V3][PresIndic][1P][Sg]
Lower: amo

where [VERB], [V3], [PresIndic], [1P] and [Sg] are multi-character symbols. Conversely, we want to be able to apply the same FSM in a downward direction to the upper-side string, and see the output amo.

While the spellings of the lower-side words in the FSM are determined by the rules of Spanish orthography, the design of the upper-side analysis strings is in our hands, and some care and study should go into that design. This is just one possible design. Another possibility is to include XML-like symbols to mark up the analysis strings.

3 Don’t worry too much about the spelling of the multi-character symbols; we will show later that they can be changed trivially at a later time, using alternation rules. Multi-characters symbols like [V3], identifying the Larousse conjugation class, can also be changed trivially to other tags (e.g. to reflect some other numbering system) or simply deleted. Here is the list that we will use in the script:
Starting from the dictionary citation forms, such as *amar*, we want to leave the full *amar* on the upper side but effectively strip off the -ar infinitive ending on the lower side, leaving just the root. This can be accomplished with the following first-draft function, to be expanded and generalized later:

```
$^\text{stripRegInfinEnding}($fst, $classTag) { 
    return ( $fst
        _o_
        ar -> "" / _ # ) ('[VERB]' $classTag):"" ;
}
```

If we call this function on the word *amar*, with the $classTag [V3],

```
$\text{stem} = $^\text{stripRegInfinEnding}(amar, '[V3]') ;
```

the result is an FSM with the following path, ignoring alignment and epsilons.
Because the conjugations are in groups of six, we can facilitate modeling each conjugation group by defining the following \( ^\text{conj6}() \) function, which takes seven arguments, the last six being the six suffixes for a conjugation group.

\[
\begin{align*}
^\text{conj6}(\text{tags}, \text{OnePerSg}, \text{TwoPerSg}, \text{ThreePerSg}, \\
\text{OnePerPl}, \text{TwoPerPl}, \text{ThreePerPl}) &= \{
\text{return } \text{tags}: "" \text{ ( ( '[1P]' '[SG]' ): OnePerSg} \\
&\quad | \text{ ( '[2P]' '[SG]' ): TwoPerSg} \\
&\quad | \text{ ( '[3P]' '[SG]' ): ThreePerSg} \\
&\quad | \text{ ( '[1P]' '[PL]' ): OnePerPl} \\
&\quad | \text{ ( '[2P]' '[PL]' ): TwoPerPl} \\
&\quad | \text{ ( '[3P]' '[PL]' ): ThreePerPl} 
\});
\end{align*}
\]

And we can then use \( ^\text{conj6}() \) to define the regular suffixes for the first-conjugation verbs including \textit{amar}: 

\[
\begin{align*}
\text{regArVerbSuffs} &= \{ \\
\quad ^\text{conj6}( '[\text{PresIndic}]' , o, as, a, amos, áis, an ) \\
\quad | ^\text{conj6}( '[\text{PretImperf}]' , aba, abas, aba, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
\end{align*}
\]
áramos, arais, aran)
| $^\text{conj6}( '[\text{ImperfSubj}]' '[\text{Var2}]', ase, ases, ase,
   ásemos, aseis, asen )
| $^\text{conj6}( '[\text{FutSubj}]', are, ares, are,
   áremos, areis, aren )
| $^\text{conj6}( '[\text{Imptv}]', '[\text{Defective}]', a, e, emos, ad, en )
| '[\text{Infin}]':(ar)
| '[\text{PresPart}]':(ando)
| '[\text{PastPart}]':(ado)
);

The first-person singular present indicative suffix path will look like this:

Upper: [PresIndic][1P][Sg]
Lower: o

When concatenated with the truncated infinitive, the full path (ignoring alignment and epsilons) looks like

Upper: amar[VERB][V3][PresIndic][1P][Sg]
Lower: amo

We can then build our first verb-conjugating FSM, for first-conjugation verbs, with

\$V3 = $^\text{stripRegInfinEnding}(\$V3CitationForms, '[V3]')
    $\text{regArVerbSuffs};$

test $\$V3 ;$

If we apply this FSM in an upward direction to the string amo, meaning “I love,” the output is the string amar[VERB][V3][PresIndic][1P][Sg], which we can read as the citation form amar, which is a verb of class 3,
in the present indicative first-person singular conjugation. And we can do the inverse, applying the same FSM in a downward direction to the input amar[VERB][V3][PresIndic][1P][Sg], and the result will be amo. The FSM thus transduces from strings representing conjugated verbs to strings representing analyses of those verbs, and vice versa.

Finally, we can define a function for testing that prints out all the conjugated forms for a verb, in a canonical order that can be compared to the usual published and online charts.

```
^conj($fst, $infin) {
    pr("\nPresent Indicative") ;
    pr($\^lowerside( ( $infin ']'VERB]' .
        '[PresIndic]' ']'[1P]' ']'[SG]' ) _o_ $fst)) ;
    pr($\^lowerside( ( $infin ']'VERB]' .
        '[PresIndic]' ']'[2P]' ']'[SG]' ) _o_ $fst)) ;
    pr($\^lowerside( ( $infin ']'VERB]' .
        '[PresIndic]' ']'[3P]' ']'[SG]' ) _o_ $fst)) ;
    pr($\^lowerside( ( $infin ']'VERB]' .
        '[PresIndic]' ']'[1P]' ']'[PL]' ) _o_ $fst)) ;
    pr($\^lowerside( ( $infin ']'VERB]' .
        '[PresIndic]' ']'[2P]' ']'[PL]' ) _o_ $fst)) ;
    pr($\^lowerside( ( $infin ']'VERB]' .
        '[PresIndic]' ']'[3P]' ']'[PL]' ) _o_ $fst)) ;
}
```
```
pr("\nPreterit Imperfect") ;
pr($\^lowerside( ( $infin ']'VERB]' .
    '[PretImperf]' ']'[1P]' ']'[SG]' ) _o_ $fst)) ;
pr($\^lowerside( ( $infin ']'VERB]' .
    '[PretImperf]' ']'[2P]' ']'[SG]' ) _o_ $fst)) ;
pr($\^lowerside( ( $infin ']'VERB]' .
    '[PretImperf]' ']'[3P]' ']'[SG]' ) _o_ $fst)) ;
pr($\^lowerside( ( $infin ']'VERB]' .
    '[PretImperf]' ']'[1P]' ']'[PL]' ) _o_ $fst)) ;
```
pr(\$^lowerside( ( $infin [\VERB]' .
\[PretImperf\]' [2P] ' [PL]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[PretImperf\]' [3P] ' [PL]' ) _o_ $fst));
pr(\n\nPreterit Perfect\n);
pr(\$^lowerside( ( $infin [\VERB]' .
\[PretPerf\]' [1P] ' [SG]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[PretPerf\]' [2P] ' [SG]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[PretPerf\]' [3P] ' [SG]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[PretPerf\]' [1P] ' [PL]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[PretPerf\]' [2P] ' [PL]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[PretPerf\]' [3P] ' [PL]' ) _o_ $fst));
pr(\n\nFuture Indictive\n);
pr(\$^lowerside( ( $infin [\VERB]' .
\[FutIndic\]' [1P] ' [SG]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[FutIndic\]' [2P] ' [SG]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[FutIndic\]' [3P] ' [SG]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[FutIndic\]' [1P] ' [PL]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[FutIndic\]' [2P] ' [PL]' ) _o_ $fst));
pr(\$^lowerside( ( $infin [\VERB]' .
\[FutIndic\]' [3P] ' [PL]' ) _o_ $fst));
pr("\nConditional")
pr($^lowerside( ( $infin '[ VERB]' .
    '[Cond]' '[1P]' '[SG]' ) _o_ $fst))
pr($^lowerside( ( $infin '[ VERB]' .
    '[Cond]' '[2P]' '[SG]' ) _o_ $fst))
pr($^lowerside( ( $infin '[ VERB]' .
    '[Cond]' '[3P]' '[SG]' ) _o_ $fst))
pr($^lowerside( ( $infin '[ VERB]' .
    '[Cond]' '[1P]' '[PL]' ) _o_ $fst))
pr($^lowerside( ( $infin '[ VERB]' .
    '[Cond]' '[2P]' '[PL]' ) _o_ $fst))
pr($^lowerside( ( $infin '[ VERB]' .
    '[Cond]' '[3P]' '[PL]' ) _o_ $fst))
pr("\nPresent Subjunctive")
pr($^lowerside( ( $infin '[ VERB]' .
    '[PresSubj]' '[1P]' '[SG]' ) _o_ $fst))
pr($^lowerside( ( $infin '[ VERB]' .
    '[PresSubj]' '[2P]' '[SG]' ) _o_ $fst))
pr($^lowerside( ( $infin '[ VERB]' .
    '[PresSubj]' '[3P]' '[SG]' ) _o_ $fst))
pr($^lowerside( ( $infin '[ VERB]' .
    '[PresSubj]' '[1P]' '[PL]' ) _o_ $fst))
pr($^lowerside( ( $infin '[ VERB]' .
    '[PresSubj]' '[2P]' '[PL]' ) _o_ $fst))
pr($^lowerside( ( $infin '[ VERB]' .
    '[PresSubj]' '[3P]' '[PL]' ) _o_ $fst))
pr("\nImperfect Subjunctive, Var 1")
pr($^lowerside( ( $infin '[ VERB]' .
    '[ImperfSubj]' '[Var1]' '[1P]' '[SG]' ) _o_ $fst))
pr(\text{"\textbackslash nImperfect Subjunctive, Var 2"})
pr($^\text{lowerside}( ( \text{infin} '[[\text{VERB}'].
  '[\text{ImperfSubj}]' '[\text{Var}2]' '[\text{1P}]' '[\text{SG}]' ) _o_ $fst))
pr($^\text{lowerside}( ( \text{infin} '[[\text{VERB}'].
  '[\text{ImperfSubj}]' '[\text{Var}2]' '[\text{2P}]' '[\text{SG}]' ) _o_ $fst))
pr($^\text{lowerside}( ( \text{infin} '[[\text{VERB}'].
  '[\text{ImperfSubj}]' '[\text{Var}2]' '[\text{3P}]' '[\text{SG}]' ) _o_ $fst))
pr($^\text{lowerside}( ( \text{infin} '[[\text{VERB}'].
  '[\text{ImperfSubj}]' '[\text{Var}2]' '[\text{1P}]' '[\text{PL}]' ) _o_ $fst))
pr($^\text{lowerside}( ( \text{infin} '[[\text{VERB}'].
  '[\text{ImperfSubj}]' '[\text{Var}2]' '[\text{2P}]' '[\text{PL}]' ) _o_ $fst))
pr($^\text{lowerside}( ( \text{infin} '[[\text{VERB}'].
  '[\text{ImperfSubj}]' '[\text{Var}2]' '[\text{3P}]' '[\text{PL}]' ) _o_ $fst))

pr("\text{\textbackslash nFuture Subjunctive"})
pr($^\text{lowerside}( ( \text{infin} '[[\text{VERB}'].
  '[\text{FutSubj}]' '[\text{1P}]' '[\text{SG}]' ) _o_ $fst))
pr($^\text{lowerside}( ( \text{infin} '[[\text{VERB}'].
  '[\text{FutSubj}]' '[\text{2P}]' '[\text{SG}]' ) _o_ $fst))
pr($^\text{lowerside}( ( \text{infin} '[[\text{VERB}'].
  '[\text{FutSubj}]' '[\text{3P}]' '[\text{SG}]' ) _o_ $fst))
pr("\nImperative")
pr($^lowerside( ( $infin '[VERB]' .
    '[FutSubj]' '[1P]' '[PL]' ) _o_ $fst))
pr($^lowerside( ( $infin '[VERB]' .
    '[FutSubj]' '[2P]' '[PL]' ) _o_ $fst))
pr($^lowerside( ( $infin '[VERB]' .
    '[FutSubj]' '[3P]' '[PL]' ) _o_ $fst))

pr($^lowerside( ( $infin '[VERB]' .
    '[Imptv]' '[2P]' '[SG]' ) _o_ $fst))
pr($^lowerside( ( $infin '[VERB]' .
    '[Imptv]' '[3P]' '[SG]' ) _o_ $fst))
pr($^lowerside( ( $infin '[VERB]' .
    '[Imptv]' '[1P]' '[PL]' ) _o_ $fst))
pr($^lowerside( ( $infin '[VERB]' .
    '[Imptv]' '[2P]' '[PL]' ) _o_ $fst))
pr($^lowerside( ( $infin '[VERB]' .
    '[Imptv]' '[3P]' '[PL]' ) _o_ $fst))

pr("\nInfinitive")
pr($^lowerside( ( $infin '[VERB]' .
    '[Infin]' ) _o_ $fst))

pr("\nPresent Participle (Gerund)")
pr($^lowerside( ( $infin '[VERB]' .
    '[PresPart]' ) _o_ $fst))

pr("\nPast Participle")
pr($^lowerside( ( $infin '[VERB]' .
    '[PastPart]' ) _o_ $fst))
}
One can then simply call \texttt{\textasciitilde conj($fst,$inf\in}) to test various verbs, printing out the entire conjugation paradigm.

\texttt{\textasciitilde conj($V3,$amar) ;}
\texttt{\textasciitilde conj($V3,$cortar) ;}

The coverage can be expanded to other verbs, in the same first-conjugation group, by simply adding more citation forms to the definition of \texttt{$V3CitationForms}.

**Expanding the Spanish Verb Morphological Analyzer**

In addition to the first conjugation, a large group consisting of regular -\textit{ar} verbs, the second conjugation consists of regular -\textit{er} verbs, including \textit{beber} ("to drink"), and the third conjugation consists of regular -\textit{ir} verbs, including \textit{vivir} ("to live"). Both my Larousse and Bescherelle conjugation guides identify a total of 90 verb-conjugation classes for Spanish, though their numbering systems differ, and even a single publisher may not use consistent numbering from one edition to another. The 90 classes include a large number of irregular verbs with idiosyncratic conjugations.

The \texttt{spanverbs-0.9.4.0.kl} script is offered on the downloads page from \texttt{www.kleene-lang.org}. It is work in progress and incomplete in several ways:

1. It covers only about 2/3 of the 90 verb-conjugation classes.

2. The definitions of the various "CitationForms" for each class are incomplete, but could be expanded easily.

3. The grammar does not yet handle clitic pronouns.

The script is in UTF-8 and can be loaded from the Kleene GUI by invoking

\texttt{source "path/to/spanverbs-0.9.4.0.kl", "UTF-8" ;}
from the pseudo-terminal, replacing “path/to” with the path to where you have stored the script on your file system. The final result of the compilation is $spanverbs, and it can be tested using the $conj($spanverbs, infinitiveform) function, e.g.

$conj($spanverbs, hablar);  

You can also test $spanverbs in the usual way, invoking

test $spanverbs;

and then entering forms like canto, cantas and cantamos in the lower-side field of the test window.

5.2.2. Spanish Nouns
This section is currently empty.

5.3. Latin Morphology
This section is currently empty.

5.4. Aymara Morphology
This section is currently empty.
Chapter 6

Examples with Weights

6.1. Introduction

This chapter, definitely work in progress, presents examples that use weights. New examples will be added as they become available.

6.2. Weighted Edit Distance for Spelling Correction

The following script, based on an example by Måns Huldén, illustrates how weighted FSTs can be used to make a spelling corrector that returns a set of possible corrections ranked by likelihood, or even just the best (most likely) correction, for a misspelled word.

The first phenomenon that we need to model is the degree of dissimilarity between an observed misspelled word and a correctly spelled possible correction. For example, the misspelled word *accomodation* is intuitively very similar to the correct word *accommodation*, differing in only one missing letter. Similarly, *panic* is very similar to *panic*, differing in having one extra letter. And *advertize* is similar to the correctly spelled *advertise*, differing only in one changed letter. Increasing in dissimilarity, *camoflage*
differs from camouflage in two letters, garentee also differs from guarantee in two letters, and bizness differs from business in three letters.

We can quantify the amount of dissimilarity between two words as the edit distance, a count of the number of editing changes, or, perhaps a more sophisticated measure of the amount of work, required to change one word into the other. The changes could include adding a letter, deleting a letter, or changing one letter into another letter.

Weighted Kleene FSTs have weights that are “costs,” under the Tropical Semiring, and the edit distance can be formalized in Kleene as the cost of transducing one word into another, assigning a cost to deleting a letter, a cost to adding a letter and a cost to changing one letter into another letter. In the Tropical Semiring, the costs associated with multiple changes are simply added together to yield the total cost. Mapping a letter to itself is free, involving no cost, i.e. a neutral cost of 0.0. Of course, with enough editing changes, any word can be changed to any other word, but intuitively we want to concentrate on the possible corrections that involve the lowest cost, the fewest edits. These intuitions will be encoded in what is known as a error model or edit model.

Another phenomenon that needs to be modeled is the fact that some words are simply more common than others in any reasonably sized corpus. Looking at a misspelled word in isolation, if a set of possible corrections having the same edit distance includes a more common word and some less common words, the more common word is more likely to be the best correction. For example, the misspelled frm differs in only one letter from the possible corrections farm, firm and from, but from is far more common than the alternatives and is—considered in isolation—more likely to be the intended word. These intuitions will be encoded in a unigram language model.

We will build our weighted edit-modeling transducer from four main parts. The first part is the identity mapping:

```
$ident = . <0.0> ; // map any symbol to itself
```
Recall that . (the dot, period or full stop) denotes any symbol or the transducer that maps any symbol to itself. Such identity mappings are free, having no cost in our spell-checking example, so we assign the identity mapping a weight of 0.0.

The second part is

$\text{changecost} = \langle 7.0 \rangle ;$
$\text{delete} = .:"$ $\text{changecost} \ ; \ // \ map \ a \ symbol$

$\quad // \ to \ the \ empty \ string$

which denotes the transducer that maps any symbol, denoted as . (dot), downward to the empty string, denoted "". We arbitrarily assign the weight of 7.0 to such a deletion, a value that we can adjust later.

The third part is

$\text{insert} = "":.$ $\text{changecost} \ ; \ // \ map \ the \ empty \ string$

$\quad // \ to \ a \ symbol$

which denotes the transducer that maps the empty string downward to any symbol, inserting a symbol where there was none before. Such mappings are technically known as epentheses. We assign such insertions the weight of 7.0.

Finally, the fourth part is

$\text{swap} = ( .:. - . ) $ $\text{changecost} \ ;$

Recall that .:. denotes the transducer that maps any symbol to any symbol, including itself. Then ( .:. - . ) denotes the transducer that maps any symbol to any other symbol, not including identity mappings. We assign such letter swaps the cost of 7.0.

Armed with these definitions, our first draft of the model for transducing between a misspelled word and a properly spelled word, the edit model, is a sequence of zero or more identity mappings, deletions, insertions or swaps, in Kleene terms:
We also need a unigram language model that encodes the language (i.e. the set) of properly spelled words that will serve as possible corrections. Initially, such a model can be constructed as a simple unweighted union of words,

\[
\text{$\text{langModel} = a \mid \text{back} \mid \text{bake} \mid \text{bee} \mid \text{carrot} \mid \text{deer} \mid \text{eye} \; ;$
\]

// and so on for hundreds, thousands, millions of words extending it eventually to hundreds of thousands or even millions of words. For the purposes of this example, we can simply search the web for posted wordlists, such as the “Simpsons Frequency Dictionary,”\(^1\) which contains the 5000 most frequent words found in open subtitles from The Simpsons television series. If we download this list, we can, with a little editing,\(^2\) convert it into a useful test model.

\[
\text{$\text{langModel} = \text{the} \mid \text{you} \mid \text{i} \mid \text{a} \mid \text{to} \mid \text{and} \mid \text{of} \mid \text{it} \; ;$
\]

// the full list has 5000 words, some needing editing // to compile correctly in Kleene

Then if the misspelled word is \textit{frm}, the language of possible corrections, \textit{$\text{corr}$}, is computed as

\[
\text{$\text{corr} = \text{frm} \_o_ \text{$\text{editModel}$} \_o_ \text{$\text{langModel}$} ;$
\]

\[\text{FstType: vector, Semiring: standard, 12992 states, 56693 arcs, 2330246 paths, Transducer, Weighted, Closed Sigma}\]

When I run this experiment, using the bare Simpsons wordlist, with my edits, Kleene informs me that the result \textit{$\text{corr}$} contains 2330246 paths,

\(^1\)http://pastebin.com/anKcMdvk

\(^2\)There are some apparently extraneous punctuation letters in this list, and for Kleene regular expressions, some vertical bars (“pipes”), digits and punctuation letters need to be removed or literalized using double quotes or backslashes.
representing the fact that, with enough edits, *frm* can be transduced into any of the 5000 words in the language model, in multiple ways.

At this point, we can start to use the weights (costs) to focus on the more likely corrections, given our edit model and language model. In the OpenFst library, the function that prunes an FSM to contain only the best paths (i.e. lowest cost, in the Tropical Semiring) is called ShortestPath. That operation and terminology are exposed in the Kleene function $^\text{shortestPath}(\text{fst}, \#\text{num}=1)$. For example, we can limit $\text{corr}$ to the best five corrections using

\[
\text{corr} = \text{^shortestPath}(\text{^lowerside}(\text{frm} \\
\quad _0_ \\
\quad \text{$\text{editModel}$} \\
\quad _0_ \\
\quad \text{$\text{langModel}$}), \\
\quad 5);
\]

print $\text{corr}$ ;
fry : 7.0
from : 7.0
firm : 7.0
farm : 7.0
arm : 7.0

Each of the five possible corrections has a weight of 7.0, indicating (in our current model that assigns a weight of 7.0 to each change) that just one edit was needed to change *frm* into the correctly spelled word.

If we ask ShortestPath to return just one result,

\[
\text{corr} = \text{^shortestPath}(\text{^lowerside}(\text{frm} \\
\quad _0_ \\
\quad \text{$\text{editModel}$} \\
\quad _0_ \\
\quad \text{$\text{langModel}$}),
\]

If we ask ShortestPath to return just one result,
print $corr ;
arm : 7.0

it randomly gives us arm, which is as close to frm as the other possibilities, given our edit model, but intuitively is not the mostly likely correction.

What’s missing in this experiment so far is a modeling of the fact that some words, in the language model, occur more frequently than others. And when a misspelled word is considered in isolation, possible corrections that are more likely should have precedence over those that are less likely. We know intuitively that from, a common English function word, occurs more often than the alternatives farm, firm, fry and arm, and we can confirm this by looking at actual frequency counts. Again, we will model the likelihood of individual words in the language model using cost weights in the Tropical Semiring.

The Simpsons word list, in fact, gives us some additional information that allows us to at least approximate the probability (and therefore the cost) of each correct word. The top of the list looks like this:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Word</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>the</td>
<td>(107946)</td>
</tr>
<tr>
<td>2</td>
<td>you</td>
<td>(98068)</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>(91502)</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>(79241)</td>
</tr>
<tr>
<td>5</td>
<td>to</td>
<td>(70362)</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>(47916)</td>
</tr>
<tr>
<td>7</td>
<td>of</td>
<td>(42175)</td>
</tr>
<tr>
<td>8</td>
<td>it</td>
<td>(36497)</td>
</tr>
<tr>
<td>9</td>
<td>in</td>
<td>(32503)</td>
</tr>
<tr>
<td>10</td>
<td>my</td>
<td>(32254)</td>
</tr>
<tr>
<td>11</td>
<td>that</td>
<td>(32083)</td>
</tr>
<tr>
<td>12</td>
<td>is</td>
<td>(31533)</td>
</tr>
<tr>
<td>13</td>
<td>this</td>
<td>(29902)</td>
</tr>
</tbody>
</table>
The “Frequency” column shows the count of occurrences of each word from the corpus. For example, $P(\text{the})$, the probability of a word being $\text{the}$, i.e. the probability of selecting a word from the corpus at random, and having it be $\text{the}$, is computed as the count of occurrences of $\text{the}$, here 107946, divided by the total number of tokens in the corpus, i.e.

$$P(\text{the}) = \frac{107946}{\text{TotalTokens}}$$

Disappointingly, the Simpsons wordlist doesn’t tell us the TotalTokens, so we are left to do some informed guessing. In the Brown Corpus, the most common word in English, $\text{the}$, is said to account for almost 7% of the tokens. For now, let’s accept this number as being valid for the Simpsons corpus as well. If 107946 instances of $\text{the}$ represented 7% of the corpus, then the total number of tokens would be a little over one and half million.

$$\frac{107946}{\text{TotalTokens}} = .07$$

TotalTokens = 1542085

For our example, let’s use the round number of 1,500,000. The probability of $\text{the}$, and some other common words, can then be estimated as

$$P(\text{the}) = \frac{107946}{1,500,000} = 0.07196$$

$$P(\text{you}) = \frac{98068}{1,500000} = 0.06538$$

$$P(\text{i}) = \frac{91502}{1,500,000} = 0.06100$$

$$P(\text{a}) = \frac{79241}{1,500,000} = 0.05300$$
Note that as the frequency of occurrence decreases, the probability value also decreases. As usual, if the probability of an event, such as the appearance of the word the in running English text, is p, then the cost, $C(\text{the})$, is computed as $-\log(p)$.

$$C(\text{the}) = -\log\left(\frac{107946}{1,500,000}\right) = 2.63159$$

$$C(\text{you}) = -\log\left(\frac{98068}{1,500,000}\right) = 2.72756$$

$$C(\text{i}) = -\log\left(\frac{91502}{1,500,000}\right) = 2.79686$$

$$C(\text{a}) = -\log\left(\frac{79241}{1,500,000}\right) = 2.94073$$

Note that as the frequencies/probabilities decrease, the costs increase. High probabilities correspond to low costs, and low probabilities correspond to high costs.

These costs can now be included in an improved weighted language model. If we tediously precompute the costs, we can simply list the cost for each word, using the usual angle-bracket syntax. Here is the Kleene syntax:

```plaintext
$\text{langModel} =$
  the <2.63159>
| you <2.72756>
| i <2.79686>
| a <2.940731>
... ;
```

Alternatively, we can let Kleene convert the probabilities into costs using the `\texttt{prob2c()}` function, “probability to cost,” which is pre-defined as

```plaintext
\texttt{prob2c(prob)} {
    \texttt{return -log(prob) ;}
}
```
The definition of the improved, weighted language model might then look like this, with \#TotalTokens defined as 1500000.0, a floating-point number, so that when it is used in division, the result is also a float.

\[
\#\text{TotalTokens} = 1500000.0 ;
\]

\[
\text{\$langModel} =
\text{the <\#^prob2c(107946 / \#TotalTokens)>}
| \text{you <\#^prob2c(98068 / \#TotalTokens)>}
| \text{i <\#^prob2c(91502 / \#TotalTokens)>}
| \text{a <\#^prob2c(79241 / \#TotalTokens)>}
\ldots ;
\]

// and so on for all 5000 words in the model.

We could easily adjust the value of \#TotalTokens if we ever get more precise information about the size of the corpus.

With the improved language model, now weighted, the best correction for \text{frm} selected by \$^\text{shortestPath()} is now \text{from}, which intuitively seems right.

\[
\text{\$corr} = \text{\$^shortestPath(\$^\text{lowerside}(\text{frm}
\ldots
\_o_\$editModel
\ldots
\_o_\$langModel))} ;
\]

print \text{\$corr} ;
from : 12.295898

And we can see the \text{n} best possible corrections by passing an optional numerical argument to \$^\text{shortestPath()}, e.g. 10:

\[
\text{\$corr} = \text{\$^shortestPath(\$^\text{lowerside}(\text{frm}
\ldots
\_o_\$editModel
\ldots
))} ;
\]
The Kleene Language

```plaintext
print $corr ;
from : 12.295898
from : 15.744141
arm : 15.818359
farm : 16.557617
fry : 17.483398
firm : 17.483398
for : 18.069336
i'm : 18.161133
are : 18.52832
him : 19.44336
```

In the Tropical Semiring, as in golf, the lower scores are the better scores, and `shortestPath()` chooses the path(s) with the lowest cost(s).

After the weighted language model is defined, the rest of the final script is shown below, including the definition of the convenience functions `correct()`, which returns an FSM, and `correctp()`, which simply prints out the results. Both functions have an optional numerical second argument, default 1, which controls how many paths are retained in the shortest-path result, and an optional numerical third argument, default 7.0, which assigns the weight for each editing change.

```plaintext
$ident = . ; // identity mapping, no change

// Three kinds of change
$delete = .:"" ; // map any symbol to the empty string
$insert = "":. ; // map the empty string to any symbol
```
$\text{swap} = \ldots - \ldots$; // map one symbol to another symbol
   // (not to itself)

// Higher change weights (costs) make changes more costly,
// favoring corrections that look as much as possible
// like the misspelled word. Lower change weights will
// tend to favor "corrections" that are very high-frequency
// words, no matter how high the edit distance between
// the observed misspelled word and the proposed corrections.

^editModel(#changecost) {
   return ( $\text{ident} <0.0> | 
           $\text{delete} <$\text{changecost}> | 
           $\text{insert} <$\text{changecost}> | 
           $\text{swap} <$\text{changecost>
              )* ;
}

^correct($\text{word}$, #num=1, #changecost = 7.0) {
   return ^shortestPath(^lowerside($\text{word}
                                  _o_
                                  ^editModel(#changecost)
                                  _o_
                                  $\text{langModel}$),
                                  #num) ;
}

^correctp($\text{word}$, #num=1, #changecost=7.0) {
   print "\n";
   print $^\text{toString}(\text{num})" correction(s) for " $\text{word}$;
   print $^\text{shortestPath}(^\text{lowerside}(\text{word
                                  _o_
                                  ^editModel(#changecost)
                                  _o_
                                  $\text{langModel}$),
                                  #num) ;
}
Once the script is loaded, one can simply call

\[
\text{print } ^\text{correct}(\text{misspelledword}) ; \\
\text{print } ^\text{correct}(\text{misspelledword}, 5) ; \\
\text{print } ^\text{correct}(\text{misspelledword}, 5, 8.0) ;
\]

or just

\[
^\text{correctp}(\text{misspelledword}) ; \\
^\text{correctp}(\text{misspelledword}, 5) ; \\
^\text{correctp}(\text{misspelledword}, 5, 8.0) ;
\]

The full script, \text{weightededitdistance.kl}, is available from the download page at www.kleene-lang.org.
Chapter 7

Challenges in Morphology

7.1. Overview

As we have seen, building FSMs for natural language morphology requires modeling both the morphotactics and the phonological and/or orthographical alternations. In morphotactics, the most common word-building processes are prefixation and suffixation, which translate straightforwardly into the finite-state operation of concatenation and seldom cause much trouble in Kleene and similar finite-state frameworks.

Of course, as soon as concatenative morphotactics was shown to be tractable, even for richly agglutinative languages such as Finnish, Basque, Turkish and Aymara, academic attention quickly turned to a range of more challenging phenomena, including infixation, reduplication and Semitic interdigitation. Another general morphotactic challenge, even in concatenative languages, is the constraint of separated ("long-distance") dependencies within a word, e.g. when a particular prefix requires a particular suffix, or when a particular prefix is incompatible with a particular suffix. This chapter, which is definitely work in progress, will address such challenges in modeling natural-language morphology in Kleene.
7.2. **Infixation**

This section is currently empty.

7.3. **Reduplication**

7.3.1. **Range of Reduplicative Phenomena**

Whereas prefixation, suffixation and infixation involve concatenating or inserting some relatively static morpheme onto/into a root or stem, reduplication is a morphotactic process in which the root or stem, or part of it, is *copied* into another part of the resulting word. The reduplicated parts, sometimes called *reduplicants*, may be prosodic feet, syllables, fairly arbitrary sequences of consonants and vowels, or single consonants or vowels, depending on the language. It is even possible to have *triplication*, where two copies are made in the same word.

It is beyond the scope of this book to go into all the semantics of reduplication. Very commonly, a reduplicated noun marks it as plural or genuine/authentic; a reduplicated verb can be emphatic or convey some aspectual meaning labeled imperfective, repetitive, frequentative or habitual.

Kleene currently offers two predefined functions, both based on algorithms kindly provided by Dr. Måns Huldén, for modeling reduplication: $^\text{redup}()$, which serves to model the majority of reduplicative phenomena, and $^\text{eq}()$, which can be used to model more difficult cases. $^\text{redup}()$ in fact calls $^\text{eq}()$, and other convenience functions could easily be defined on top of $^\text{eq}()$ as necessary.

These functions, and the syntactic idioms that use them, are best understood in concrete examples.

---

1Personal communications and Hulden and Bischoff (2009).
7.3.2. **Contiguous Reduplication**

The $^\text{redup()}$ Function

In the majority of languages that exhibit reduplication, there is a single reduplicant that appears contiguous with (next to) the substring being copied, sometimes separated in the orthography by a special symbol, which is often the hyphen.

To model such contiguous-reduplication phenomena Kleene offers the $^\text{redup}($lang, $^\text{sep}="")$ function, in which $\text{lang}$ is a finite language of substrings to be reduplicated, and $^\text{sep}$ denotes the separator symbol, which by default is the empty string. Both $\text{lang}$ and $^\text{sep}$ must denote acceptors.

Before we look at the full idioms needed to model reduplication, let’s see what $^\text{redup}()$ itself does:

```plaintext
$\text{lang} = \text{foo} | \text{bar} | \text{bas} ;
$r = ^\text{redup}($\text{lang}, \text{-}) ;
pr $r ;
```

The output of this little script is

```
foo-foo
bar-bar
bas-bas
```

Similarly, the output of the following script

```plaintext
$\text{lang} = \text{foo} | \text{bar} | \text{bas} ;
$r = ^\text{redup}($\text{lang}, "") ; \text{ // or just } ^\text{redup}($\text{lang})
print $r ;
```

is

```
foofoo
barbar
basbas
```
**Full (Contiguous) Reduplication**

Indonesian illustrates productive full-root and full-stem reduplication, and the standard orthography now dictates that a hyphen be used as the separator. For example, where *buku* is the word for book, the overt plural is written *buku-buku*;\(^2\) and where *rumah* is the word for house, the overt plural is *rumah-rumah*. Though this was once a very difficult form of reduplication to handle in finite-state systems, it is quite easy using \(^*\)redup().

Let's assume that we want to create a transducer $\text{indonesianNouns}$ that relates “buku” on the lower side to “buku[NOUN]” on the upper side, and “buku-buku” on the lower side to “buku[NOUN][REDUP]” on the upper side. This requires surrounding the call to $^*\text{redup}()$ with some idiomatic code, as in this example:

\[
\text{$\text{nouns} = \text{buku} \mid \text{rumah} \mid \text{anjing} \; \text{// book, house, dog}$}
\]

\[
\text{$\text{reduplicatedNouns} =$
\begin{align*}
^0 \cdot^ * \text{nouns} \\
^*\text{redup(nouns, \text{\-})} & ^* \text{\[^\text{\-}\]* } ^*\text{ignore(nouns, \text{\-})} ; \end{align*}
\]

\[
\text{$\text{indonesianNouns} = \text{nouns} '([NOUN]:"' | }$
\]

\[
\text{$\text{reduplicatedNouns} (['\text{NOUN}'] ['\text{REDUP}']):"' ;}$
\]

Let's take this line by line. The definition of $\text{nouns}$

\[
\text{$\text{nouns} = \text{buku} \mid \text{rumah} \mid \text{anjing} \; \text{// book, house, dog}$}
\]

is a language consisting of strings that, when overtly plural, need to be reduplicated. These words will be considered the baseforms. The following line

\(^2\)Reduplication is not used if the word appears in a context that makes the plurality obvious, e.g. when *buku* is preceded by the words for the plural cardinal numbers two, three, four, etc.
$reduplicatedNouns =
  "":.* $nouns
  _0_
  $^redup($nouns, \-) & [^-]* $^ignore($nouns, \-) ;

is a composition of two FSMs, the first being the FST

"":.* $nouns

which has the un-reduplicated nouns on the upper side (the baseforms),
related to lower-side strings that start with any number of any symbols,
ending with the un-reduplicated nouns. The second line

$^redup($nouns, \-) & [^-]* $^ignore($nouns, \-)

denotes an acceptor that is the intersection of $^redup($nouns, \-),
which is the language consisting of “buku-buku,” “rumah-rumah” and
“anjing-anging,” with [^-]* $^ignore($nouns, \-), the language of
strings starting with any number of non-hyphen symbols, and ending with
our baseform-noun strings, but ignoring any hyphens. In this case, the
second line could be simplified a bit to

$reduplicatedNouns = "":.* $nouns
  _0_
  $^redup($nouns, \-) & [^-]* \- $nouns ;

but the call to $^ignore() is part of the more general idiom, as will be
seen in subsequent examples.

If you test the resulting FST in the usual way

test $indonesianNouns ;

and enter, in the lower-side entry field, “buku,” the output will be “buku[NOUN].”
And if you enter “buku-buku,” the output will be “buku[NOUN][REDUP].”
Partial Contiguous Prefixal Reduplication

The following example, from Måns Huldén, models a reduplication phenomenon found in Warlpiri, where the first prosodic foot of a word, defined by the pattern C V C? C? V (a consonant, a vowel, one or two optional consonants, and a second vowel) is reduplicated. No separating hyphen is used in the orthography. Thus if the baseform is *pangurnu*, the reduplicated form is *pangupangurnu*. Because the reduplication is attached to the front of the word, this is known as *prefixal* reduplication.

// Hulden's example of partial prefixal reduplication
// in Warlpiri. Reduplication of the initial prosodic foot
// of the form C V C? C? V

$lexicon = pangurnu | tiirlparnkaja | wantimi | pakarni ;
$C = p | ng | rn | rl | k | j | w | n | t | m ;
$V = [aeiou] ;

// the pattern of the prosodic foot
$prosodicFoot = $C $V $C? $C? $V ;

// extract out the prosodic stems (the substrings that need to
// be reduplicated)
$prosodicStem = $^lowerside($lexicon _o_ $prosodicFoot .:"") ;

// calculate/construct the reduplicated strings
$rmorphology =
  ":.* $lexicon '[REDUP]'":"
  _o_
  $^redup($prosodicStem, \-).* &[^-]* $^ignore($lexicon, \-)
  _o_
  \- -> "" ;
If you test the result, you will find that lower-side “wantimi” is related to upper-side “wantimi[UNMARKED],” and lower-side “wantiwanti-mi” is related to upper-side “wantimi[REDUP].” Draw the network, test the strings in both directions, and walk through the example until you are comfortable with it. This illustrates the general idiom for computing prefixal reduplication, and the main change for other languages with this phenomenon will be to redefine the prosodic foot (or syllable) that is appropriate.

**Partial Contiguous Suffixal Reduplication**

(KRB: research the actual facts of Dakota reduplication. This is a pseudo example for now.) The Dakota language is reputed to have a reduplication where the final C C V of an adjective root is reduplicated to mark the plural. Because the copy attaches to the end of the word, this is known as suffixal reduplication. The idiom for suffixal reduplication is very similar to that for prefixal reduplication. If you have trouble typing the $ used in the example, simply replace it with S or '[$]' for now.

```r
$C = [ptkbdghfsʃzwz] ;
$V = [aeiou] ;
$adj = haska | waʃte | mazdi | tuftu ;
$pattern = $C $C $V ;
$syl = $^outputProj($adj _o_ .::* $pattern) ;
$redupMorphology =
```
If you test the resulting FST in the usual way, the lower-side “haska” is related to the upper-side “haska[ADJ][SG],” and lower-side “haskaska” is related to the upper-side “haska[ADJ][REDUP].” Other languages with suffixal reduplication can be handled similarly, modifying the $pattern as appropriate for the language. In another language, for example, the pattern might be $C $V $C.

**Partial Contiguous Infixal Reduplication**

[KRB: examples taken from Wikipedia article on reduplication. Check them when possible.]

Samoan displays a kind of infixal reduplication in which a penultimate syllable of the form C V is reduplicated. Thus the reduplicated form of savali is savavali, and the reduplicated form of alofa is alolofa. The final syllable can be just V, so the reduplicated form of tamaloa is tamaloloa.

```
$C = [fglnpovhr\']);
$V = [aeiou];

$lang = savali | alofa | tamaloa;

// pattern of penultimate syllable that is reduplicated
$pat = $C $V;
```
// extract the syllables to be reduplicated
$sylls = $^outputProj($lang
    .:"* $pat $C:"? $V:"" ) ;

$rmorph = $lang
    .[^-]* "": (\- [^-]* \- ) $C? $V
    .[^-]* $^redup($sylls, \-) \- $C? $V
     \- \- > "" ;

$samoan = $lang '[UNMARKED]':'' | $rmorph '[REDUP]':'' ;

Full Reduplication with Alternations

Sproat (1992, p. 57) lists the following examples of full-stem reduplication in Javanese that also involve some vowel alternations. ³

<table>
<thead>
<tr>
<th>Base</th>
<th>Habitual-Repetitive</th>
<th>Gloss</th>
</tr>
</thead>
<tbody>
<tr>
<td>adʊs</td>
<td>odas + adʊs</td>
<td>‘take a bath’</td>
</tr>
<tr>
<td>bali</td>
<td>bola + bali</td>
<td>‘return’</td>
</tr>
<tr>
<td>bosən</td>
<td>bosan + bosən</td>
<td>‘tired of’</td>
</tr>
<tr>
<td>ɛlɛq</td>
<td>elaq + ɛlɛq</td>
<td>‘bad’</td>
</tr>
<tr>
<td>ibu</td>
<td>iba + ibu</td>
<td>‘mother’</td>
</tr>
<tr>
<td>dolan</td>
<td>dolan + dolen</td>
<td>‘engage in recreation’</td>
</tr>
<tr>
<td>udan</td>
<td>udan + uden</td>
<td>‘rain’</td>
</tr>
<tr>
<td>djaron</td>
<td>djoran-djaron</td>
<td>‘horse’</td>
</tr>
<tr>
<td>djaron</td>
<td>djoran-djəren</td>
<td>‘horse’</td>
</tr>
<tr>
<td>djaron</td>
<td>djaran-djəren</td>
<td>‘horse’</td>
</tr>
</tbody>
</table>

³Sproat in turn cites an unpublished manuscript by Paul Kiparsky, “The Phonology of Reduplication,” Stanford University, that I have not yet examined.
Here is Sproat’s description of the alternations:

As Kiparsky describes it, if the second vowel of the stem is not /a/—as in the first five examples above—the algorithm is as follows: copy the stem, make the first vowel of the left copy nonlow, with the result that /a/ is changed to /o/ and /ɛ/ is changed to /e/, and change the second vowel to /a/. If the second vowel of the stem is /a/, then the left copy remains unchanged but the /a/ in the right copy changes to /ɛ/. If both vowels are /a/, as in djaran ‘horse’, there are three different possibilities, as shown [above].

Måns Huldén wrote a Foma script to handle the Javanese examples, as just described, and here, with his permission, is his script translated into Kleene syntax:

```plaintext
$Base = adʊs | bali | bosən | ɛlɛq | ibu |
       dolan | udan | djaran ;
$V = [aeiouɛʊə] ;
$C = [bfghkmnpqrstvwx] ;
$Lexicon = "" <- \-* || _ '[HabRep]' |
   _O_
   $redup(Base, \-) '[HabRep]': '"' |
   $Base '[Unmarked]': '"' ;
```

// The Lexicon transducer now contains input-output
// pairs such as:

// b o l a [HabRep] (input)
// b o l a - b o l a (output)
// or
// b o l a [Unmarked] (input)
// b o l a (output)
// This we can now compose with the rules that change
// vowels in the two copies yielding, e.g.,
// bola-bola -> bola-bali, etc.

// The main rule
$Rule1 =
    $parallel( 
        $a -> $b / # $C* _ $C* ($V - a) .* \-
        { where $a E_ $@($a, e), 
          $b E_ $@($o, e) },
        $V -> a / # $C* $V $C* _ .* \-
        a -> e / \- $C* ($V - a) $C* _ .* 
    ) ;

// To handle the "three different possibilities"
// seen in "djaran"
// KRB: Is -> || a two-level restriction expression?
// KRB: Need fixed here for Foma-to-Kleene
$Rule2 =
    $parallel( 
        a:[o|a] [C|V]* %- C* a:e C* a:e -> || .#. C* _ ?* ,
        a:(o|a) ($C|$V)* \- $C* a:e $C* a:e -> / # $C* _ .* ,

        a (->) o || .#. C* _ ?* %-
        a ->? o / # $C* _ .* \-
    ) ;

$javanese = $Lexicon _o_ $Rule1 _o_ $Rule2 ;

The resulting $javanese FST can be tested in the usual way.
7.3.3. Discontiguous Reduplication

The \(^{eq}\) Function

The \(\text{redup}(\text{lang}, \text{sep}="\) function demonstrated above calls a lower-level function \(^{eq}\) that is harder to use but permits the modeling of discontiguous reduplication.

7.4. Semitic Interdigitation

This section is currently empty.

7.5. Constraining Long-distance Dependencies

This section is currently empty.
Chapter 8

Arithmetic Expressions

8.1. Mindtuning for Arithmetic Expressions

While Kleene is designed primarily for creating and manipulating finite-state networks, it does support arithmetic expressions, variables that hold arithmetic values, functions that return arithmetic values, etc. Variables with an arithmetic value are marked with a # sigil, functions that return an arithmetic value with #^, etc. Wherever a simple integer or float can appear in Kleene syntax, including numbered iterations like a{2, 4} and weights like <0.1>, an arbitrarily complex arithmetic expression can appear, e.g. a{2, #maxlength - 1} and <#defaultweight + .01>.

The arguments passed to a function could be any mix of regular expressions and arithmetic expressions, and one of the biggest challenges during Kleene design and development was the distinguishing and proper tokenization/parsing of the two separate expression types. For example, both use the plus sign as an operator, but in arithmetic expressions it is a binary infix operator of fairly low precedence, e.g. 2+3, while in regular expressions it is a unary postfix operator of fairly high precedence, e.g. ab+c. Distinguishing the two expression types by inventing new operators—either for regular expressions or for arithmetic expressions—was

\[\text{Internally, Kleene stores arithmetic values as either a Long or a Double object.}\]
judged to be completely unacceptable; and forcing users to surround regular expressions, or arithmetic expressions, with some kind of explicit delimiters, such as the Perl slashes /.../, was deemed inelegant and undesirable.

The solution adopted was to define a systematic set of sigils starting with $ for network-value variables and functions, and # for arithmetic-value variables and functions. Parser lookahead distinguishes #a+#b as an arithmetic expression, involving addition, from $a+$b, which is a regular expression indicating the concatenation of one or more iterations of network $a$ with network $b$. Once the sigil system is mastered, users can, in almost all cases, simply type familiar regular and arithmetic expressions in appropriate places.

The remaining problematic cases are expressions like 2 and 2+3, which start with digits. Are they arithmetic expressions, having a integer or float value, or regular expressions, having a network value? The Kleene solution is to treat bare digits by default as arithmetic expressions. To be interpreted as literal characters, and therefore regular expressions, digits must be literalized in the usual Kleene ways:

- Using the prefix backslash literalizer: \2
- Surrounding them with double quotes: "2", or
- Putting them in square-bracketed expressions: [ 2 ] [ ^2 ] [ 0-9 ] [ ^0-9 ]

### 8.2. Primary Arithmetic Expressions

The primary arithmetic expressions are the following:
0 1 42  
0x1 0x1A 0x23ABC  
2.5 .234 103.  
#n #count #foo  
#^abs(#numexp)  
#^arcCount($regexp)  
#^stateCount($regexp)  
#^myfunction(args)  

decimal integers  
hex integers  
decimal floats  
names of variables denoting an arithmetic value  
names of functions returning an arithmetic value  

Note that Kleene does not support octal (base 8) representations of integers.

8.3. Arithmetic Expression Operators

The following regular expression operators are available, listed from high to low precedence.

<table>
<thead>
<tr>
<th></th>
<th>parenthetical grouping</th>
<th>circumfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ -</td>
<td>positive, negative</td>
<td>prefix</td>
</tr>
<tr>
<td>* / %</td>
<td>multiply, divide, mod</td>
<td>infix</td>
</tr>
<tr>
<td>+ -</td>
<td>add/subtract</td>
<td>infix</td>
</tr>
<tr>
<td>&lt; &lt;= &gt;= == != &gt;</td>
<td>Boolean comparisons</td>
<td>infix</td>
</tr>
<tr>
<td>!</td>
<td>Boolean NOT</td>
<td>prefix</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Boolean AND</td>
<td>infix</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The result of boolean operations is always 1 (true) or 0 (false). There is no separate boolean type in Kleene.

8.4. Arithmetic Functions

The following pre-defined arithmetic-valued functions are provided.
In Kleene there is no difference between \(^{\text{long}}(\text{num})\) and \(^{\text{int}}(\text{num})\); both are stored as long integers.\(^2\) Similarly, there is no difference between \(^{\text{double}}(\text{num})\) and \(^{\text{float}}(\text{num})\); both are stored as double values.\(^3\)

The \(^{\text{log}}(\text{num})\) function returns the natural logarithm of the argument. The \(^{\text{prob2c}}(\text{prob})\) takes an argument that represents a probability, where probabilities range from 1.0 (certain) to 0.0 (impossible), and returns a cost weight suitable for the Tropical Semiring.\(^4\) The \(^{\text{prob2c}}(\text{prob})\) function might be used in examples like the following,

\[
\text{\$fst = a b ( c <^{\text{prob2c}}(0.1)> | d <^{\text{prob2c}}(0.9)> ) ;}
\]

where the path containing c has 0.1 probability, and the path containing d has 0.9 probability. The \(^{\text{pct2c}}(\text{pct})\) is similar but allows you

\(^2\)Long integers are stored internally as Java Long objects.

\(^3\)Double values are stored internally as Java Double objects.

\(^4\)In the Tropical Semiring, the weights are costs. For any probability \(p\), the corresponding cost is \(-\log(p)\). A probability of 1.0 corresponds to a cost of 0.0, and a probability of 0.0 corresponds to an infinite cost.
to express the weights as percentages, e.g. 10% vs. 90%. The following example is equivalent to the one just above.

\[
fsm = a \ b \ ( c \ \text{<#^pct2c(10)>} \ | \ d \ \text{<#^pct2c(90)>}) ;
\]

[KRB: explain why the #$^\text{pathCount()}$ is not completely accurate. Explain the other functions.]

## 8.5. Boolean Functions

### 8.5.1. Special Case of Arithmetic Functions

Boolean functions are special cases of arithmetic functions that return 1 for true and 0 for false. They are typically used in if-else expressions or in loops for testing qualities of finite-state machines. The predefined boolean functions are

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#$^\text{isAcceptor($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{isTransducer($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{isRtn($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{isWeighted($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{isIDeterministic($fst)}$</td>
<td>#$^\text{isUDeterministic($fst)}$</td>
</tr>
<tr>
<td>#$^\text{isODeterministic($fst)}$</td>
<td>#$^\text{isLDeterministic($fst)}$</td>
</tr>
<tr>
<td>#$^\text{isCyclic($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{isIBounded($fst)}$</td>
<td>#$^\text{isUBounded($fst)}$</td>
</tr>
<tr>
<td>#$^\text{isOBounded($fst)}$</td>
<td>#$^\text{isLBounded($fst)}$</td>
</tr>
<tr>
<td>#$^\text{containsOther($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{hasClosedAlphabet($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{isEpsilonFree($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{isEmptyLanguage($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{isEmptyStringLanguage($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{isString($fst)}$</td>
<td>#$^\text{isSingleStringLanguage($fst)}$</td>
</tr>
<tr>
<td>#$^\text{containsEmptyString($fst)}$</td>
<td></td>
</tr>
<tr>
<td>#$^\text{isUniversalLanguage($fst)}$</td>
<td></td>
</tr>
</tbody>
</table>
and others will no doubt be defined as they become needed.

The \(^{\text{isAcceptor}}(\text{fst})\) function returns true if and only if the argument network is semantically an acceptor. In OpenFst, where all arc labels \(i:o\) in a finite-state machine are two-level, with an input label \(i\) and an output label \(o\), an acceptor is a special case of a machine wherein each arc-label mapping is an identity mapping.\(^5\)

The \(^{\text{isTransducer}}(\text{fst})\) function returns true if and only if the argument network is semantically a non-identity transducer. Thus if the machine contains any arc label like \(a:b\), where the upper label is different from the lower label, \(^{\text{isTransducer}}\) will return true.\(^6\)

The \(^{\text{isWeighted}}(\text{fst})\) function returns true if and only if the argument finite-state machine contains any arc with a non-neutral weight value.

The \(^{\text{isIDeterministic}}(\text{fst})\) function returns true if and only if the finite-state machine is input-side deterministic, where “input” here is taken in the OpenFst sense of the upper side. Similarly, the function \(^{\text{isODeterministic}}(\text{fst})\) returns true if and only if the network is output-side deterministic, in the OpenFst sense.\(^7\)

The \(^{\text{isCyclic}}(\text{fst})\) function returns true if and only if the finite-state machine has cycles, also known as loops.

The \(^{\text{isIBounded}}(\text{fst})\) function returns true if and only if the finite-state machine has no input side epsilon loops.

The \(^{\text{isOBounded}}(\text{fst})\) function returns true if and only if the finite-state machine has no output side epsilon loops.

---

\(^5\)If the finite-state machine contains an arc labeled OTHER NONID:OTHER NONID, then it will look like an acceptor to the OpenFst library, but in Kleene such a network is semantically and in practice a transducer; if \(^{\text{isAcceptor}}()\) is called on such a network the return value is 0 (false). Conversely, if \(^{\text{isTransducer}}()\) is called on any network that contains an arc labeled OTHER NONID:OTHER NONID, the return value is 1 (true).

\(^6\)If the argument contains an arc labeled OTHER NONID:OTHER NONID, then it will look like an identity relation to OpenFst itself, but in Kleene this is semantically a non-identity transducer and \(^{\text{isTransducer}}\) will return 0 (false).

\(^7\)The spellings IDeterministic and ODeterministic are used inside the OpenFst library.
The \(^\text{containsOther}(\text{fst})\) function returns true if and only if the network contains at least one \text{OTHER_ID} or \text{OTHER_NONID} label. Conversely, \(^\text{hasClosedAlphabet}(\text{fst})\) returns true if and only if the alphabet of the network is closed, i.e. the network does not contain any \text{OTHER_ID} or \text{OTHER_NONID} labels. Thus \(^\text{hasClosedAlphabet}(\text{fst})\) is equivalent to \(!^\text{containsOther}(\text{fst})\).

The \(^\text{isEpsilonFree}(\text{fst})\) function returns true if and only if the network contains no \text{EPSILON:EPSILON} arcs, i.e. no double-sided epsilon labels. Because Kleene routinely runs epsilon-removal on all networks, which removes such double-sided epsilon arcs, unless such removal is explicitly turned off, there is seldom any practical need to call \(^\text{isEpsilonFree}(\text{fst})\).

The \(^\text{isEmptyLanguage}(\text{fst})\) function returns true if and only if the network encodes the empty language, i.e. the language that contains no strings at all, not even the empty string.

The \(^\text{isEmptyStringLanguage}(\text{fst})\) function returns true if and only if the network encodes the empty-string language, i.e. the language that contains only the empty string.

The \(^\text{isString}(\text{fst})\) function, aliased as \(^\text{isSingleStringLanguage}(\text{fst})\), returns true if and only if the network encodes a language of exactly one string.

The \(^\text{containsEmptyString}(\text{fst})\) function returns true if and only if the network encodes a language that contains the empty string.

The \(^\text{isUniversalLanguage}(\text{fst})\) function returns true if and only if the network encodes the universal language, i.e. the language that contains all possible strings. [KRB, 2012-08-19: This function will not be fully reliable until a “compact sigma” function is implemented.]

### 8.5.2. Assert and Require Statements

The \text{assert}(#)\text{numexp, $regexp$} statement has a required arithmetic first argument, which is analyzed as a boolean, and an optional second regular-expression argument, which must denote a single-string language. If the
first argument analyzes as true, Kleene takes no further action and control simply passes to the next statement. But if the first argument analyzes as false, a runtime exception is thrown and the single-string second argument, if present, is used as the exception message. The `assert(#numexp, $regexp)` statement is typically used for testing.

```perl
$fst = a - a;
assert (#^isEmptyLanguage($fst),
       "Should denote the empty language.");

$fst2 = a*;
assert (#^containsEmptyString($fst2),
       "Should contain the empty string.");
```

In complete parallel, the `require(#numexp, $regexp)` statement also has a required arithmetic first argument, which is analyzed as a boolean, and an optional second regular-expression argument, which must denote a single-string language. If the first argument analyzes as true, Kleene takes no further action and control simply passes to the next statement. But if the first argument analyzes as false, a runtime exception is thrown and the single-string second argument, if present, is used as the exception message. The `require(#numexp, $regexp)` statement is typically used in user-defined functions to impose semantic restrictions on the arguments.  

```perl
#^myIntersect($a, $b) {
    require (#^isAcceptor($a) && #^isAcceptor($b),
             "The arguments to #^myIntersect must denote an acceptor.");

    return $a & $b;
}
```

---

8The need for separate `assert(#numexp, $regexp)` and `require(#numexp, $regexp)` arguments could be debated. One possibility would be to implement a global switch that would cause all `assert(#numexp, $regexp)` statements, but not `require(#numexp, $regexp)` statements, to be ignored at runtime.
Chapter 9

Lists

9.1. Lists of FSMs and Numbers

In addition to individual FSMs, and individual numbers, Kleene also supports collections of FSMs and collections of numbers. These collections are implemented as linked lists, and functions are provided to manipulate them as lists, stacks, queues and double-ended queues (also known as deques, pronounced decks). They will be referred to collectively herein as lists.

Kleene lists must be homogeneous; that is, a single list must contain only FSMs, or only numbers, though the numbers may be a mix of integers and floats. Kleene lists are sometimes referred to informally as arrays, but they are technically lists and are better thought of as such.

Kleene provides syntax to represent, assign, access and manipulate lists. Lists can be passed as arguments to functions, and functions can return a list as a result. A for each statement supports iteration through the members of a list.

---

1In the underlying Java code, they are implemented as LinkedList objects.
2Kleene integers are always stored internally as Java Long objects.
3Kleene floats are always stored internally as Java Double objects.
4Readers familiar with Lisp and Scheme are warned that Kleene lists cannot contain lists.
The sigil $@$ marks lists of FSMs, and #@ marks lists of numbers.

9.2. List Literals, Identifiers, and Assignment

List literals, also known as anonymous lists, are denoted by the appropriate sigil, $@$ for FSMs and #@ for numbers, followed by a parenthesized list of elements.

$@(dog, cat, a, a*b+[c-g]) // a literal list of 4 FSMs

#@(12, -45, 2.47, 0.326, 0) // a literal list of 5 numbers

An identifier of the form $@name can be bound to an FSM-list value; and, in parallel, an identifier of the form #@name can be bound to a number-list value.

$@foo = $@(a, b, $fsm1, $fsm2, (dog|cat|elephant)s?)

#@bar = #@(1, 2, 12.23, 9, -234)

When such assignment statements are executed in the GUI, the lists are represented by named icons that appear automatically in the symbol-table window.

The info and delete commands work with list identifiers just as they do with individual FSM and number identifiers.

info $@foo;
info #@bar;

delete $@foo;
delete #@bar;

In the GUI, these commands can be accessed by right-clicking on a list icon.
9.3. Pre-Defined Functions Operating on Lists

Functions Accessing the Elements of a List

Individual list elements can be accessed non-destructively from a list using the following pre-defined functions. Note that index counting starts at 0 (zero).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$^\text{head}(@list)</td>
<td>Returns the zeroth FSM element of the argument list</td>
</tr>
<tr>
<td>$^\text{getLast}(@list)</td>
<td>Returns the last FSM element of the argument list</td>
</tr>
<tr>
<td>$^\text{get}(@list, n)</td>
<td>Returns the nth FSM element of the argument list</td>
</tr>
</tbody>
</table>

Such functions return FSMs and can be called in the usual ways, e.g.

```
@@mylist = @(@a, @b, @c, @d) ;
$fsm = ^\text{head}@(@mylist) ;
```

would set $fsm to the FSM denoted by $a$.

Parallel functions are defined for number-lists:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#^\text{head}(@list)</td>
<td>Returns the zeroth number element of the argument list</td>
</tr>
<tr>
<td>#^\text{getLast}(@list)</td>
<td>Returns the last number element of the argument list</td>
</tr>
<tr>
<td>#^\text{get}(@list, n)</td>
<td>Returns the nth number element of the argument list</td>
</tr>
</tbody>
</table>

The following functions return a new list containing all or part of the elements of the argument list. These operations are non-destructive.
The statement

\[$@\text{tailList} = @^\text{tail}(@\text{a, b, c}) ;\]

sets $@\text{tailList}$ to the value $@\text{(b, c)}$. As in Scheme, Haskell and Scala, the head and tail functions throw an exception if the argument is an empty list. The call

\[$@\text{newList} = @^\text{getSlice}(@\text{list, 2, 4, 7:10}) ;\]

\begin{table}[h]
\begin{tabular}{|l|l|}
\hline
$@^\text{copy} (@@\text{list})$ & Returns a shallow copy of the argument FSM-list \\
\hline
$@^\text{tail} (@@\text{list})$ & Returns a new list containing all but the first element of the argument FSM-list \\
\hline
$@^\text{getSlice} (@@\text{list}, #n, #r:#s, ...)$ & Returns a new list containing the indicated elements of the argument FSM-list \\
\hline
$@@^\text{copy} (@@\text{list})$ & Returns a shallow copy of the argument number-list \\
\hline
$@@^\text{tail} (@@\text{list})$ & Returns a new list containing all but the first element of the argument number-list \\
\hline
$@@^\text{getSlice} (@@\text{list}, #n, #r:#s, ...)$ & Returns a new list containing the indicated elements of the argument number-list \\
\hline
\end{tabular}
\end{table}
sets $\text{@newList}$ to a list containing items 2, 4, 7, 8 and 9 of $\text{@list}$. Note that index counting starts at 0, and the notation $n:m$ is inclusive of $n$ but exclusive of $m$, so 7:10 includes 7, 8 and 9.

The following functions destructively return an individual element from the argument list, removing that element from the list. As elsewhere in Kleene, destructive function names are marked with a final !.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{^pop!(@list)}$</td>
<td>Removes and returns the zeroth element of a FSM-list</td>
</tr>
<tr>
<td>$\text{^removeLast!(@list)}$</td>
<td>Removes and returns the last element of a FSM-list</td>
</tr>
<tr>
<td>$\text{^remove!(@list, #n)}$</td>
<td>Removes and returns the nth element of a FSM-list</td>
</tr>
<tr>
<td>$\text{^pop!(#list)}$</td>
<td>Removes and returns the zeroth element of a number-list</td>
</tr>
<tr>
<td>$\text{^removeLast!(#list)}$</td>
<td>Removes and returns the last element of a number-list</td>
</tr>
<tr>
<td>$\text{^remove!(#list, #n)}$</td>
<td>Removes and returns the nth element of a number-list</td>
</tr>
</tbody>
</table>

The alias $\text{^pop_back!(@list)}$ is pre-defined for $\text{^removeLast!(@list)}$, and $\text{^pop_back!(#list)}$ is pre-defined as an alias for $\text{^removeLast!(#list)}$.

The following functions add elements destructively to a list, changing the list, and returning the changed list.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{^push!(fst, @list)}$</td>
<td>Push $fst$ on the front of the FSM-list, and return the modified list</td>
</tr>
<tr>
<td>$\text{^add!(@list, fst)}$</td>
<td>Add $fst$ on the end of the FSM-list, and return the modified list</td>
</tr>
<tr>
<td>$\text{^addAt!(@list, #n, fst)}$</td>
<td>Add/insert $fst$ at the indicated index of the FSM-list, and return the modified list</td>
</tr>
</tbody>
</table>
The alias \(^\text{push\_back!}(\text{list}, \text{fst})\) is pre-defined for the function \(^\text{add!}(\text{list}, \text{fst})\), and \(^\text{push\_back!}(\text{list}, \text{num})\) is pre-defined as an alias for \(^\text{add!}(\text{list}, \text{num})\).

The \(^\text{set\_set!}(\text{list}, \text{n}, \text{fst})\) function and the parallel function for number-lists \(^\text{set\_set!}(\text{list}, \text{n}, \text{num})\) reset the list value at index \(\text{n}\) to \(\text{fst}\) and \(\text{num}\), respectively, and return the modified list.

### 9.3.1. Functions Joining the Elements of a List

The function \(^\text{reduceLeft}(\text{bin}, \text{list})\) takes as arguments a binary function \(^\text{bin}\) that takes two FSM arguments and a list of FSMs \(\text{list}\); and it returns an FSM value. \(^\text{reduceLeft}(\text{bin}, \text{list})\) applies the passed-in binary function to combine the members of the list from left to right and returns the result. The behavior is best seen in the following example, where a list of FSMs is reduced first by concatenation and then
by composition.

```perl
$^concat($x, $y) { return $x $y; }
$^compose($x, $y) { return $x _o_ $y ; }

@list = @a:b, b:c, c:d ;

$catfsm = $^reduceLeft($^concat, @list) ;
// $catfsm encodes the relation a:b b:c c:d

$compfsm = $^reduceLeft($^compose, @list) ;
// $compfsm encodes the relation a:d

$^foldLeft($^bin, @list, $init) is like $^reduceLeft() except that it has three arguments: a binary function, an FSM list and an initial FSM value. $^foldLeft() is often preferable to $^reduceLeft() because it returns the initial value, rather than throwing an exception, when the argument list is empty.

$result = $^foldLeft($^concat, @list, "") ;
```

Similarly, the numerical function $^reduceLeft($^bin, @list) takes as arguments a binary function $^bin and a list of numbers @list, and it returns a number value. In the following example, a list of numbers is reduced first by addition and then by multiplication.

```perl
// define some binary functions
$^add($x, $y) { return $x + $y ; }
$^mult($x, $y) { return $x * $y ; }

@list = @1, 2, 3, 4 ;

#sum = $^reduceLeft($^add, @list) ;
// #sum has the value 10
```
#product = ^reduceLeft(^mult, @list) ;
// #product has the value 24

^foldLeft(^bin, @list, #init) is like ^reduceLeft() except that it has three arguments: a binary function, a number list and an initial number value. ^foldLeft() is often preferable to ^reduceLeft() because it returns the initial value, rather than throwing an exception, when the argument list is empty.

With the foldLeft and reduceLeft functions, it is sometimes convenient to pass an anonymous function as the first argument, e.g.

// calling foldLeft with anonymous functions
#sum = ^foldLeft(^(#a, #b) {return #a + #b;}, @list, 0) ;
#product = ^foldLeft(^(#a, #b) {return #a * #b;}, @list, 1) ;

9.3.2. Functions Returning the Map of a List

The ^map($^mon, @$list) applies the single-argument function $^mon to each FSM element of @$list and returns a list of the returned FSM values. The arguments can be anonymous. Note that the ^map($^mon, @$list) function has the sigil $^, indicating a function (function names are always immediately preceded by ^) that returns a list of FSMs ($@).

@$inList = @(a, b, c, d) ;

// called with an anonymous function that concatenates
// the two arguments
@$outList = ^map($^(#fst){ return #fst $fst ; }, @$inList) ;
// @$outList will be @$aa, bb, cc, dd)

// called with an anonymous function and an anonymous list
@$outList = ^map($^(#fst){ return #fst $fst ; },
        @$a, b, c, d)) ;
Similarly, the `map(^mon, list)` applies the single-argument function `^mon` to each number element of `list` and returns a list of the returned values. The arguments can be anonymous. Note that the `map(^mon, list)` function has the sigil `^`, indicating a function (function names are always immediately preceded by `^`) that returns a list of numbers (`@`).

```plaintext
@inList = @(0, 1, 2, 3);

// called with an anonymous function
@outList = map(^{ n => n + n }, @inList);
// @outList will be @(0, 2, 4, 6)

// called with an anonymous function and an anonymous list
@outList = map(^{ n => n * n }, @(1, 2, 3, 4));
// @outList will be @(1, 4, 9, 16)
```

### 9.3.3. Functions Returning the Alphabet (Sigma) of an FSM

The `getSigma(fsm)` function returns the alphabet (the sigma) of the argument FSM as a list of FSMs, each one consisting of a start state and a final state, linked by one arc labeled with a symbol from the sigma. Special symbols used only internally in Kleene are excluded from the sigma.

The parallel `getSigma(fsm)` function returns the sigma of the argument FSM as a list of integers.

```plaintext
$fsm = abc;

$@list = @getSigma($fsm);
@list = @getSigma($fsm);
```
9.3.4. Functions Returning the Size of a List

The \(^\text{size}(\text{list})\) function takes a list argument, either an FSM-list or a number-list, and returns the size as an integer.\textsuperscript{5}

\[
#n = ^\text{size}(#@(1, 2, 3, 4)) ;
\]
\[
#m = ^\text{size}($@(#(a, b, c, d, e, f)) ;
\]

The \(^\text{isEmpty}(\text{list})\) function takes a list argument and returns 1 (true) if the list is empty and 0 (false) otherwise.

9.4. Iteration through Members of a List

The \texttt{foreach} statement iterates through the elements of a list, allowing some operation or operations to be performed on each element. The body of the \texttt{foreach} statement is a block or a single statement.

\[
\text{foreach ($fsm \text{ in } @\text{list}) { info } \text{ $fsm ;}}
\]
\[
\text{foreach ($fsm \text{ in } @\text{list}) \text{ info } \text{ $fsm ;}}
\]
\[
\text{foreach (#num \text{ in } @\text{list}) { info #num ;}}
\]
\[
\text{foreach (#num \text{ in } @\text{list}) \text{ info } #num ;}
\]

\textsuperscript{5}\text{Kleene does not, in general, support function overloading, wherein multiple functions of the same name are distinguished by the type and/or number of arguments they take. The \texttt{size} pseudo function is an exception, wired into the parser and specially interpreted to accept an argument of either list type.}
As a more concrete example, consider

$fsm = abc ;$

foreach (#cpv in #@^getSigma($fsm)) info #cpv ;
// output:
// Long value: 98
// Long value: 99
// Long value: 97

For each iteration of the block or statement, the iteration identifier, $fsm$, $num$ and #cpv in these examples, is bound, in the current frame, to the next item of the list, and after the foreach statement, the iteration identifier is left bound to the last value in the list. In the following example, the value of $num$ at Point B is 3.

$num = 0 ; // Point A$
foreach (#num in #@{1, 2, 3}) {
    print #num ;
}
info #num ; // Point B

(KRB: consider the wisdom of treating the iteration identifier this way. Perhaps it should be (at least conceptually, a new identifier that is bound for each value, and perhaps it should disappear after the foreach loop is finished.)

### 9.5. User-Defined Functions and Lists

Users can define their own functions that take lists as arguments, and functions that return lists, e.g.

---

6The foreach statement does not, like a function call, trigger the allocation of a new frame.
The Kleene Language

```
$@reverse($@list) {
    $@result = $@(); // an empty list
    foreach ($fsm in $@list) {
        $@result = $@^push!($fsm, $@result) ;
    }
    return $@result ;
}

$@reversedList = $@^reverse($@(a, b, c)) ;
```

9.6. Traditional Array-indexing Syntax

[KRB: This section is work-in-progress.]

Although the current implementation of lists provides access functions like $^get($@list, #n), to retrieve a value at a specified index, some users might prefer to use postfixed indices in square brackets or parentheses, as in most implementations of arrays.

```
// NOT implemented in Kleene
$@arr[0] // array indexing as in C/C++, Java
$@arr(0) // array indexing as in Scala
```

Assuming for a second that such post-fixed indexing is desired, it is not yet clear that it could be implemented easily in Kleene. Because square brackets normally denote symbol unions in Kleene regular expressions, tokenizing and parsing them differently when they are intended to indicate an index would be a challenge; the notation with parentheses, as used in Scala, might be easier to implement and less likely to cause confusion. As is often the case, tokenization is a bigger challenge than the parsing.

If the expression denoting a list is just an identifier like $@list, as in the examples above, the challenge is trivial. But if the expression denoting a list is more complex, such as a function call returning a list, and
having arbitrary arguments to tokenize and parse, then correctly tokenizing a post-fixed index expression that uses square brackets would be rather involved. Again, using parentheses might avoid most of the problem.

Traditional arrays also allow setting of elements of the array, e.g.

```perl
$@arr[2] = a*b+[c-g]?
```

and this would also present challenges for Kleene tokenization and parsing.

At this point, we should continue to think about post-fixed indexing syntax. As Kleene lists are not really arrays, and as for each statements are provided for iterating easily through lists, I believe that we can dispense with postfix iteration altogether.
Chapter 10

Other Syntax

10.1. Void Functions with Names

Void functions return no value, and their names are prefixed with the plain ^ sigil, like all Kleene functions, but with nothing before the ^. Such void functions can be used to abbreviate and generalize a sequence of Kleene commands.

^inputSigma($fst) {
    $input = ^inputProj($fst);
    sigma $input;
    draw $input;
}

// an invocation of the defined void function
^inputSigma(abc);

10.2. Anonymous Void Functions

An anonymous void function looks like ^(...args...){...body...}, with the plain ^ sigil marking a function that has no return value.
10.3. Control Syntax

In addition to FSM-, arithmetic- and function-assignment statements, Kleene provides if-elsif-then statements, while and until loops, iteration over list elements, and a variety of “housekeeping” statements for input-output, executing pre-edited scripts, retrieving information about FSMs, drawing FSMs, etc. For example, numbered iteration could be implemented by defining the following functions (though Kleene already provides the convenient \{n\} postfix operator):

```bash
$^iterate_by_recursion($fsm, #count) {
    if (#count > 0) {
        return $fsm $^iterate_by_recursion($fsm, #count - 1) ;
    } else {
        return "" ;
    }
}

$^iterate_by_loop($fsm, #count) {
    $result = "" ;
    while (#count > 0) {
        $result = $result $fsm ;
        #count = #count - 1 ;
    }
    return $result ;
}
```

10.4. Input/Output

10.4.1. Scripts

When Kleene is invoked “bare” from the command line, using
$ java -jar Kleene.jar

the GUI is launched, allowing users to type individual statements for immediate interpretation.¹ A Kleene SCRIPT is a sequence of Kleene statements pre-edited and stored in a file. Scripts can be executed by including their filenames in the command line,

$ java -jar Kleene.jar myscript
$ java -jar Kleene.jar myscript1 myscript2 ...

and Kleene will, by default, exit after all the scripts are executed. The script names can be paths such as /Users/beesley/kleene/scripts/myscript.kl. By default, Kleene (and Java programs in general) will assume that the script file is in the default encoding of the operating system, and will convert it to Unicode accordingly.² If a script is not in the default encoding of the operating system, its encoding should be specified explicitly using the -encoding flag, which should appear before the filename(s):

$ java -jar Kleene.jar -encoding UTF-8 myscript
$ java -jar Kleene.jar -encoding Latin-1 myscript ...

To cause Kleene to execute one or more scripts—typically start-up scripts—and then launch the GUI for interactive input, add the -gui flag anywhere among the script names.

$ java -jar Kleene.jar myStartupScript -gui

To execute a Kleene script from the GUI, i.e. from the Kleene language itself, use the source command, which requires one Kleene-regular-expression argument denoting a single-string filename or path, with an

¹The dollar sign in these examples represents the command-line prompt. Kleene will initially be packaged for distribution as an executable JAR file.

²Java’s String objects are always Unicode. Text input to Java programs is always converted to Unicode, one way or another. By default, text output from Java is converted from Unicode to the default encoding of the operating system.
optional comma-separated second argument (also denoting a language of
a single string) indicating the encoding of the source file. In Kleene, a
sequence of alphabetic letters such as \texttt{myscript} is of course such a single-
string regular expression; and double-quoted strings, also regular expres-
sions, are appropriate if the name or path contains special characters like
periods, slashes and hyphens.

// invoke from the Kleene GUI
// By default, the encoding of the script is assumed to be the
default encoding of the operating system
source "myscript.kl" ;
source "/Users/beesley/kleene/scripts/testscript.kl" ;
source "myscript.kl", "UTF-8" ; // specify the encoding

By default, Kleene will attempt to read the script file using the default
encoding of your operating system, as perceived by Java. In Java code,
the default encoding of the operating system is supposed to be returned
by \texttt{System.getProperty("file.encoding")}, and on Linux this seems to
work. However, Apple OS X annoyingly reports the default encoding as
"MacRoman" even if you've reset your locale otherwise.\textsuperscript{3}

In the Kleene GUI, the terminal widget has a pull-down Source menu
that brings up a window that allows you to browse for the source file
and to indicate the encoding. The encoding will be pre-set to the default
encoding of the operating system, but it can be changed if necessary.

\textsuperscript{3}In Apple OS X, you can set your \texttt{LANG} environment variable to something like
\texttt{en\_US.UTF-8}, which causes the \texttt{locale} command to report UTF-8 encoding, but inside
Java, \texttt{System.getProperty("file.encoding")} still returns MacRoman. If you, like
me, have set up your whole OS X system for default UTF-8 encoding, and (naturally)
expect Java to perceive the default encoding as UTF-8, you have to launch Kleene with
something like \texttt{java -jar -Dfile.encoding=UTF-8 Kleene.jar}.
10.4.2. XML Input/Output

An FSM can be written to file in Kleene's XML language using the \texttt{writeXml} command, which takes as arguments a regular expression representing the FSM to be written, and a second regular expression indicating the file-path:

\begin{verbatim}
writeXml $fsm, "/Users/beesley/kleene/xml/fsm.xml" ;
writeXml (dog|cat|elephant)s? , 
        "/Users/beesley/kleene/xml/animals.xml" ;
\end{verbatim}

By default, the file is written in Unicode UTF-8, which is the official default encoding for all XML files.\footnote{The XML header of the file will be written as \texttt{<?xml version="1.0" encoding="UTF-8"?>}, though the overt specification of UTF-8 encoding is redundant.} Other encodings can be specified in an optional third argument, but the only other recommendable encoding is UTF-16. XML parsers are required to handle only UTF-8 and UTF-16, though they may handle other encodings. Most users will want to stay with the default UTF-8 encoding.

\begin{verbatim}
// default UTF-8 output
writeXml $fsm, "/Users/beesley/kleene/xml/fsm.xml" ;

// explicit UTF-8 output
writeXml $fsm, "/Users/beesley/kleene/xml/fsm.xml, "UTF-8" ;

// explicit UTF-16 output
writeXml $fsm, "/Users/beesley/kleene/xml/fsm.xml", "UTF-16" ;
\end{verbatim}

In the Kleene GUI, the \texttt{writeXml} command can be invoked by right-clicking on an FSM icon and selecting the \texttt{writeXml} item. The file selection window allows you to select the encoding, which is pre-set to the recommended UTF-8.

The built-in function \texttt{$\wedge$readXml()} reads from a Kleene XML file and returns an FSM.
// to read file fsm.xml in the current directory
$newfsm = $\textasciitilde \text{readXml}("fsm.xml") ;

// a full pathname can be specified
$newnfsm = $\textasciitilde \text{readXml}("/Users/beesley/kleene/xml/fsm.xml") ;

The encoding of an XML file is always 1) specified by an overt encoding in the XML header, if present, or 2) detectable automatically from the BOM (Byte Order Mark), if present, or 3) UTF-8 by default. The reading of XML files is now done using the ROME library’s XmlReader class, which knows how to detect the encoding of an XML file.

10.4.3. DOT Output

FSMs can be written to file in the GraphViz dot format\(^5\) using the \texttt{writeDot} command, which is similar to the \texttt{writeXml} command.

\begin{verbatim}
writeDot $fsm, "/Users/beesley/kleene/xml/fsm.dot" ;
writeDot (dog|cat|elephant)s? ,
"/Users/beesley/kleene/xml/animals.dot" ;
\end{verbatim}

By default, the file will be written to file in the UTF-8 encoding. A different encoding can be specified in an optional third argument, but the only valid alternative is ISO-8859-1, because the dot parser can handle only UTF-8 and ISO-8859-1. Most users will want to stay with the default UTF-8 encoding, which is the only alternative if the sigma of the FSM contains Unicode symbols beyond the ISO-8859-1 range.

In the Kleene GUI, the \texttt{writeDot} command can be invoked by right-clicking on an FSM icon and selecting the \texttt{writeDot} item. The file selection window allows you to select the encoding, which is pre-set to the recommended UTF-8. DOT output is used in the background when you invoke the \texttt{draw} command, which can also be generated by right-clicking

\(^5\text{http://www.graphviz.org}\)
on the FSM’s icon and selecting the draw item, or by double-clicking on the icon.

10.4.4. Interactive Testing in the GUI

For interactive manual testing of an FSM within the GUI, use the Kleene test command, e.g.

```
test $myfsm ;
test (dog|cat|rat) '[Noun]':"" ( '[Sg]':"" | '[Pl]':s ) ;
```

The test command causes the indicated FSM to be scanned for multi-character symbols, which are used to build string-to-symbol tokenizers, one for the input side, and one for the output side.æk A testing window is opened, with string input fields allowing the user to type in strings either for analysis or generation. An input string for analysis is tokenized into symbols, built-in a one-string FSM, and composed on the output (lower) side of the transducer; the input (upper) side of the composed FSM is then extracted as the result. Generation is the same, except that the one-string FSM is composed on the input (upper) side of the original FSM, and the output (lower) side of the composed FSM is extracted as the result. If the result language is reasonably finite, the individual strings are displayed in the terminal window.

When the FSM being tested contains OTHER (unknown) characters, the output may be unintuitive; see Appendix D for more details.

When an FSM is finite (no cycles) relatively small and an acceptor, you can print out the strings using the print command:

```
$net = dog | cat | elephant ;
print $net ;
// output:
```

---

6These tokenizers are currently built using the Transliterator object from ICU (International Components for Unicode).
// cat: 0.0
// dog: 0.0
// elephant: 0.0

[KRB: Random upper and lower strings. New functionality 2013-04. Not well tested yet.] For transducers and large FST, even those with cycles, you can print out a random selection of upper-side strings, or lower-side strings, using the randInput (alias randUpper, rinput and rupper) command:

```
randInput $fst;
randUpper $fst;
```

Similarly, you can use the randOutput (alias randLower, routput and rlower) command to print out a random selection of the output/lowerside strings.

```
randOutput $fst;
randLower $fst;
```

These commands are often useful during development to “see what the upper (or lower) strings look like.” These commands also have optional second and third arguments, the second being the number of strings to display (the default is currently 15), and the third is the maximum length of a string (default 50), which can be useful to avoid getting into infinite loops.

```
// print 20 random input strings
randInput $fst, 20;

// print 10 random input strings, max 25 symbols
randUpper $fst, 10, 25;
```

The “random” function are based on the OpenFst RandGen() function.
10.5. Memory Management

Memory management commands are intended for use by the developers and maintainers of Kleene, not for typical users.

Kleene is based on the OpenFst library, and the finite-state machines (the actual states and arcs) are built by OpenFst and are therefore C++ objects, i.e. all the states and arcs, comprising a C++ FSM object, reside in the C++ memory space.

On the Java side of Kleene, each finite-state machine is represented by a Java Fst object, and each allocated Fst object holds a pointer to the corresponding C++ finite-state machine object. The memory taken up by the Fst object itself is Java memory, and can be reclaimed by the normal Java garbage collection. But the memory required by the states and arcs is C++ memory and is invisible to the Java garbage collector.

10.5.1. Garbage Collection

The Kleene parser and interpreter are written in Java, which automatically performs garbage collection of Java objects when they no longer have any references to them. In general, Kleene users should never have to worry about garbage collection, and even lower-level Java programmers have little control over when garbage collection is done.

In case it should become necessary or useful, Kleene offers the gc command, which when evaluated suggests to Java that “now would be a good

---

7 Defined in the file Fst.java.
8 The Java Fst object also keeps track of each FSM’s private sigma, which is a concept alien to OpenFst.
time to perform garbage collection".  

```plaintext
$foo = foo ;
$bar = bar ;
delete $foo, $bar ;
gc ;
```

### 10.5.2. Java Memory Usage

Kleene also offers the `memory` command, which prints out a summary showing the maximum memory, the memory in use, and the free memory, as seen by Java.

```plaintext
memory ;
```

The format of the output, not shown here, will no doubt evolve as needs become clear. Note that Kleene currently has no way to interrogate the memory usage or availability in the C++ memory space.

### 10.5.3. Java Fst Objects

Kleene now keeps track of how many Java Fst objects have been allocated, and how many have been “finalized”, and this information can be displayed by the `fsts` command. When an Fst object has been finalized, the Java object itself should be garbage collected, and the corresponding OpenFst finite-state machine should be deleted.

---

9Java does not necessarily have to accept the hint to perform garbage collection. The `gc` command calls `System.gc()` and `System.runFinalization()`, multiple times, which is alleged to nudge Java more aggressively. In a Java Fst object, the `finalize()` method calls a native C++ function that in turn calls the normal `delete` command of C++ to delete the C++ finite-state machine (the states and arcs in C++ memory). The `finalize()` method is advertised as the way to release/delete “native”, i.e. non-Java, resources, so it’s just what Kleene needs; but there is much debate among Java experts about if and how `finalize()` should be relied on.
The command also displays the number of “live” Fsts, which is the difference between the number of allocated Fsts and the number of finalized Fsts.

10.5.4. Contents of the Main Symbol Table

The symtab command prints a list of the items defined in the main “program” symbol table, and these items should match the icons in the GUI symtab window.

Contents of the Global Symbol Table

The gsymtab command prints a list of the items defined in the global symbol table, which is the root of the Kleene environment and the mother of the “program” symbol table. The items in the list should correspond to the definitions in the system-supplied .kleene/global/predefined.kl start-up script.

10.5.5. C++ Memory Space

The Kleene system includes one custom-written non-OpenFst C++ file, named kleeneopenfst.cc, that defines the “native” (C++) functions directly callable from the Kleene interpreter and serves as the JNI bridge between Kleene and the OpenFst library. As with any program involving C or C++, there is a significant danger of memory leaks; finding and plugging them will be a high priority.

Another possible and significant source of C++ memory leaks would be the creation of native finite-state machines that are never deleted; the
The *fsts* command, described above, may provide clues to tracking down such leaks. Kleene currently has no direct way to interrogate the memory usage/availability in the C++ memory space.\(^\text{10}\)

\(^{10}\)Memory profilers may provide some help here.
Chapter 11

Code Generation and Runtime Code

11.1. Using Kleene FSMs in Real Projects

You can build FSMs using Kleene scripts and interactive GUI sessions, and you can use various commands, notably test, to test your FSMs manually inside the Kleene GUI.

```
test $MyFsm ;
```

You have learned that these FSMs can also be stored to file in DOT and XML formats; and the XML files can even be read back into Kleene, or shared with other Kleene programmers. However, an XML representation of an FSM is not executable code—it doesn’t actually do anything by itself. Even an FSM residing in memory is just a data structure consisting of nodes and arcs, and again it doesn’t do anything at all by itself. An FSM needs to be applied to input, and the output needs to be retrieved, by code that is derived from or exterior to the FSM itself.

In order for your FSMs to be incorporated and used in projects external to the Kleene development environment itself, we need to do one or both of the following:
• Write runtime code that reads in a Kleene FSM, e.g. from an XML representation, rebuilds the FSM in memory, applies it to input and retrieves the output. Such runtime code, probably implemented in a library in Java, C++, Python or whatever, could be imported and used by application programs.

• Or, write scripts that take a Kleene FSM and convert it into stand-alone executable code that has the same behavior as the runtime code.

At this time, there is no runtime code available to load and run the FSMs created in Kleene. Perhaps an interested developer would volunteer to write such code. However, I (Ken Beesley) have written some experimental XSLT scripts that take a “state-oriented” XML file saved by Kleene and convert it into executable Java code (a Java package) that can be imported and used inside a larger Java program. It is thus easy to incorporate your FSM, e.g. a transducer that performs morphological analysis, inside any program written in the Java language, or in any program written in Scala, Groovy or Clojure, which are also based on the Java Virtual Machine (JVM) and so can use Java classes.

In the future, it should be possible to convert Kleene FSMs into classes in other languages, e.g. C++ and Python.

I will first discuss the current status and usage of the “fst2java” project, which generates executable code in Java, and then give a high-level view of the requirements for runtime code.

11.2. The fst2java Experiment

11.2.1. Stand-Alone Java Code Generation

There is an experimental project called fst2java that takes a Kleene FSM, stored to file in a state-oriented XML format, and runs this XML file through a set of XSLT scripts, ultimately generating a Java package that implements
the FSM as executable Java code. This generated code performs complete matching of an input string, suitable for applications including tokenization and morphological analysis, and returns a set of result strings for each successfully (completely) matched input string.

The current fst2java project does not implement partial matching, which would be suitable for applications such as entity extraction inside large input strings.

Once an FSM has been stored to file, e.g. as MyFst.xml, and converted to a Java package, e.g. MyFstPackage, containing a class definition MyFst.java, then any Java (or Scala, Groovy or Clojure) program can simply

- Import the MyFst package/class
- Create an instance of the class, and
- Call methods of the class to pass in input strings and various options, and get back the results

The strings passed as input to the class methods are normal Java String objects, and the methods inside the class are generated to convert these String objects automatically into lists of code point values, taking into consideration any multi-character symbols in the alphabet of the original FSM. A variety of public class methods are defined to accept String input and return the results either as set (Java ArrayList) of Strings or as a marked-up XML string.

### 11.2.2. How it Works

**States Translate to Functions**

Recall that a Kleene FSM consists of a finite number of states, one of which is designated as the start state, and zero or more of which are final states; and each state has zero or more exit arcs, each with an upper label and a
lower label and a weight, leading to a destination state. In a high-altitude overview, fst2java translates each state in the FSM into a Java function that looks at the next input symbol in the input and tries to match that symbol to an upper or input label (“input” in the OpenFst visualization of FSMs) on one or more exit arcs. For each matching exit arc, it calls the function representing the arc’s destination state, passing the remainder of the input string. Functions representing final states also check, when first called, to see if the input is exhausted, and if so, record a successful full match.

The scheme is complicated by weights, arcs with epsilon labels, which have to be “explored” without consuming any input, and by arcs labeled with other symbols, which match any input symbol that is not in the alphabet of the FSM. These complications are all taken care of in the generated code, and the Kleene/Java programmer only needs to learn the API, that is, the set of public methods available in the generated class.

**Generation**

The fact that the generated Java code matches input symbols against upper-side labels (“input” labels in the OpenFst visualization), means that the FSM is effectively being applied in a downward direction, what the Xerox visualization of FSMs calls generation mode. However, if you have built a typical Xerox-style FST morphological analyzer, i.e. an FST that has analysis strings (consisting of citation forms and tag symbols) on the upper side, and orthographical strings on the lower side, and you want to generate Java code that accepts orthographical strings and returns analysis strings, then all you need to do is to invert the FST before generating the state-oriented XML file. Then the input side (in the OpenFst visualization) will contain orthographical strings, the string input will be matched on that input side, the output side will contain the analysis strings (consisting of baseforms and tags), and the Java code will work as expected.

Again, it is vitally important to understand that a Java class generated by fst2java from a Kleene FSM operates only in the downward or gener-
ation direction as just described. The input will be matched on the upper side, what the OpenFst tradition calls the input side.

If, in your Java code, you want to apply an original FST in both the downward (generation) and upward (analysis) directions, then you need to generate two separate Java classes:

1. One from the original FST, perhaps resulting in MyFstPackage/MyFst.java, and

2. Another from the inverted FST, perhaps resulting in MyFstInvertedPackage/MyFstInverted.java

Then your Java code could import both the MyFst and MyFstInverted packages/classes, calling one to correspond to generation with the original FST, and the other to correspond to analysis with the original FST.

11.2.3. How to Create the Java Class Files for an FSM

Status

The fst2java project is still experimental and is not well tested. It is not yet wired into the Kleene GUI, and it will not be until it stabilizes and is better tested. For now, users who want to generate Java classes will need to follow the steps listed below, which hopefully aren’t too onerous. The generation of Java code is performed by a cascade of XSLT scripts, and is rather slow. I (Ken Beesley) would be most grateful for any information about your problems, successes and failures in generating Java code from your FSMs.

Step One: Create a Kleene FSM

Starting in the Kleene GUI or in a Kleene script, create an FSM, for example one named $Fst, that you would like to have converted into stand-alone Java code.
The purpose of fst2java is to take such an FSM and output Java code that simulates the *downward* application of the FSM to the input, what the Xerox visualization of FSMs calls generation. That is, the input string will be matched on the upper side (the OpenFst “input” side) of the FSM, and the results will be read off of the lower side (the OpenFst “output” side). So for success in using fst2java, visualize your FSM using the OpenFst concepts of input and output.

**Step Two: Make Sure that the Input Side is the Upper Side**

Once you have an FSM, call the `randInput` function (which has as aliases `rinput`, `randUpper` and `rupper`):

```
randInput $fst ;
```

This will print out a random set of strings from the input/upper side of the FSM. If these strings look like the strings you want to use as inputs in the Java-code version, then the FSM is properly oriented.

If and only if you want the Java code to apply your FSM in the other direction, i.e. in an *upward* direction, matching input on the lower side (the OpenFst “output” side), then you simply need to invert your FSM before writing it out as XML.

```
// invert your FSM only if needed to put the desired "input"
// side on the upper side
$Fst = $^invert($Fst) ;
```

Examples below will illustrate when such inversion is required.

**Step Three: Check for Input Side Epsilon Cycles**

Your FSM should now be oriented so that the input strings you provide will be matched on the input/upper side. In typical natural-language systems,
an input string will match only one path, or a very finite number of paths, producing one output string, or a very finite number of output strings. However, it is possible that your FSM has input epsilon cycles, loops that have only epsilon on the input side, allowing a single input string to map to an infinite number of output strings. For any FSM that is to be converted into stand-alone Java code, you probably do not want input epsilon cycles.

Here is a trivial example of an FSM that has an input side epsilon cycle:

\[ \text{\$Fst = c\ a\ t\ "":s* ;} \]

The resulting FSM looks like this:

If this FSM is applied in a downward direction to \textit{cat}, i.e. if the input string \textit{cat} is matched against the upper/input side, the input side epsilon cycle on node 3 allows an infinite number of matches, with an infinite number of outputs:

\begin{verbatim}
cat
cats
catss
catsss
catssss
catssss...
\end{verbatim}

To test your FSM for cycles, use the pre-defined boolean predicate \#^isIBounded() (for “is input bounded”), which is aliased as \#^isUBounded() (for “is upper bounded”). For example, in the Kleene GUI, if your FSM is named \$Fst, just do the following:
#bool = #^isIBounded($Fst) ;

or equivalently

#bool = #^isUBounded($Fst) ;

and the system will respond indicating 1 (= true = input/upper bounded) or 0 (= false = the FSM has input/upper epsilon cycles). You could be a little fancier and write

if (#^isIBounded($Fst))
    pr "OK. It's input bounded." ;
else
    pr "DANGER: There are input epsilon cycles." ;
}

Again, you probably do not want input/upper cycles in your FSM. If you proceed with an FSM that has input epsilon cycles, the generated Java code should look for and block epsilon loops, but this feature is still experimental.

**Step Four: Output the FSM as a State-Oriented XML file**

Beware: there are currently two quite separate ways to output a Kleene FSM as XML:

1. An arc-oriented format using the command `writeXml`; this XML format can be read back into Kleene using the `readXml` command, and

2. A state-oriented XML format using the command `writeXmlStateOriented`

For use with `fst2java`, you want the second (state-oriented) output.

Output your FSM to file in the special state-oriented XML format using the command `writeXmlStateOriented`. If your FSM is named something other than `$Fst`, substitute “Fst” with your own FSM’s name in the following command:
writeXmlStateOriented $Fst, "Fst.xml", "UTF-8";

This command creates a file named $Fst.xml in the current directory. This file contains a state-oriented representation of your FSM in an XML format. Don’t worry too much about this right now—it just means that XML elements representing the arcs exiting a state are grouped inside another XML element representing that state.) The challenge is now to take this $Fst.xml file (or whatever you have called it) and generate from it executable Java code that simulates the application of the original FSM.

**Step Five: Generating the Java Package using fst2java**

Now cd to the fst2java directory (downloadable from www.kleene-lang.org), which contains four XSLT scripts that convert the FSM from one form of XML to another form of XML, plus one final XSLT script, named generate.java.xsl, that takes the final XML file and generates from it the Java code as a Java package.¹ The application of these five XSLT scripts is all controlled by the Makefile in the fst2java directory, so we’re very close to the end.

Java packages need to have a name, so you now have to choose the name. I assume here that you want to call it “Fst,” but choose any name that you like.

If you want to use the name “Fst,” then do exactly one of three things:

1. Move your just-generated $Fst.xml file to the fst2java directory, or
2. Copy your $Fst.xml file to the fstjava directory, or
3. Create a soft link named $Fst.xml, in the fst2java directory, to $Fst.xml

It doesn’t really matter which one you choose. If in doubt, just create the soft link. Here’s how to create a soft link (fixing the paths to reflect your environment as appropriate):

---

¹A Java package is a set of related Java files residing in one directory.
$ cd /path/to/fst2java
$ ln -s /path/to/Fst.xml

If you want the Java class to be named something else, such as “My-Morphology,” then do exactly one of these three things:

1. Move your Fst.xml (or whatever you called it) file to the fst2java directory and rename it MyMorphology.xml, or

2. Create a copy of Fst.xml in the fst2java directory and call that copy MyMorphology.xml, or

3. Create a soft link, named MyMorphology.xml, in the fst2java directory, linking it to Fst.xml

If in doubt, just create a soft link. Here’s how to create a soft link named MyMorphology.xml

$ cd /path/to/fst2java
$ ln -s /path/to/Fst.xml MyMorphology.xml

Step 6: Validate the XML File

From this point on, you will be using the Makefile in the fst2java directory, and that Makefile depends on having Java, the JDK (Java Development Kit),\(^ \text{2} \) and two Java JAR files, installed in your environment.

To see if you have the JDK installed, enter

$ which javac

Hopefully, the system will return a path showing where javac is installed. If the command fails, returning a message like “Command not found,” then

\(^ \text{2} \)If you are running Kleene, you have the JRE Java Runtime Environment, but you may not have the JDK (Java Development Kit), which includes the javac compiler.
you will first need to install the JDK. If this means nothing to you, consult your system administrator or a Java-savvy friend.

Now, in the fst2java directory, edit the Makefile, using a plain-text editor like emacs, vim, vi, gedit, etc. and note the two definitions at the top:

```
SAXONJAR=/Users/beesley/java/saxon/saxon.jar
# This is a comment.
# Edit the SAXONJAR path above to point to your copy of
#  saxon.jar, which may be named saxon9he.jar
# If you don't have saxon.jar, see
#  http://saxon.sourceforge.net/
#  and download the Saxon-HE (home edition),
#  which contains the jar file.

JINGJAR=/Users/beesley/java/jing/bin/jing.jar
# Edit the JINGJAR path above to point to your copy of
#  jing.jar
# If you don't have jing.jar, see
#  http://code.google.com/p/jing-trang/downloads/list
#  and download jing-20091111.zip or a newer version.
```

As of this writing, the current version of Saxon is SaxonHE9-5-0-1J and the actual JAR file in the download is named saxon9he.jar. Save this JAR file in a convenient location and edit the Makefile so that the definition of SAXONJAR points to the JAR file on your system. For example, if your name is Victoria and you store the JAR file in /home/victoria/javajars/, edit the line to

```
SAXONJAR=/home/victoria/javajars/saxon9he.jar
```

Similarly, you need jing.jar. If you don't have it already, surf to /http://code.google.com/p/jing-trang/downloads/list and download jing-20091111.zip (or a newer version, if any). If necessary, unzip
the file (sometimes this is done by your browser during the download). Inside the resulting jing-20091111-1 directory, inside the bin subdirectory, is the jing.jar file. Save this file in a convenient location and edit the definition of JINGJAR in the Makefile to show the path to it on your system. Save the Makefile.

As a first test, let's try to validate your Fst.xml file, or whatever it's called. Validation (using the Makefile and the supplied Relax NG script) requires Java and the jing.jar file. Enter the following command:

$ make name=Fst validate

If your file (or soft link) is name something else, such as BobsFirstTry.xml, then change the command to

$ make name=BobsFirstTry validate

If there is no response, simply a return to the command-line prompt, then all is well. If the command itself fails to run, then there is some problem with your Java or jing.jar installation. Review the instructions above and/or consult an expert. If the validation starts but reports errors in the XML file, then please contact me (krbeesley@gmail.com) with a detailed error report. You won't be able to progress with code generation until the problem is solved.

If the validation is successful, then enter

$ make name=Fst

changing the name as appropriate. If all goes well, it will generate a Java package in a new directory named FstPackage.

If your XML file (or link) is named something else, like BobsFirstTry.xml, then launch

$ make name=BobsFirstTry

and it will generate a Java package in a directory named BobsFirstTryPackage.

---

3The validation is performed against a Relax NG schema named fst.rnc, which is supplied in the fst2java directory.
11.2.4. FSM to Java Examples

A Simple Alternation Rule

Congratulations on making it this far. It’s time for some practical examples. Let’s start with a simple alternation rule that maps s to z when it occurs between two vowels. In the Kleene GUI, define the following rule:

$Vowel = [aeiou] ;$

$StoZrule = s -> z / $Vowel _ $Vowel ;$

or just

$StoZrule = s -> z / [aeiou] _ [aeiou] ;$

As previously noted, alternation rules compile into transducers, and a right-arrow rule has an inherent downward-oriented (or generation) bias built into it. The way the rule is written, we naturally think of applying it in a downward direction to an input string like casa and getting back caza as the output. That is, the input is matched against the upper side (the Openfst “input” side), and the output is read off of the lower side of the matched paths(s). Now let’s test the rule, still in the Kleene GUI:

test $StoZrule ;

A test window will pop up, and we enter casa in the upper-side text-entry field. When we press Enter, the answer

caza: 0.0

appears in the history window. Now enter sasusis in the upper-side entry field and the output, as expected, is

sazuzis: 0.0
Let's assume that we want to generate Java code that will have this same downward-oriented behavior, a “generation” behavior, matching input against the upper side, with the output read off of the lower side. This reflects the OpenFst notion of input side, and so everything is already in the proper orientation for code generation. We do not want to invert this FSM before Java-code generation.

Before we proceed to generate Java code, let's take a look at the FSM.

draw $StoZule ;

Note that the FSM has three states, numbered 0, 1 and 2, and we'll see this structure reflected in the generated Java code. We also test the FSM for input-side boundedness.
if (isIBounded($StoZrule))
    pr "All is OK";
else
    pr "Problem: input epsilon cycle(s)";

The system responds with “All is OK” and we can proceed.

Now we output the rule FSM to file in the state-oriented XML format:

writeXmlStateOriented $StoZrule, "StoZrule.xml", "UTF-8";

This will create file StoZrule.xml. Now cd to the fst2xml directory, create a soft link to the StoZrule.xml file just created, and validate the file just to be sure that it’s kosher.

$ cd /path/to/fst2xml
$ ln -s /path/to/StoZrule.xml
$ make name=StoZrule validate

If the validation is successful, i.e. if it returns silently, we can, finally, generate the Java code:

$ make name=StoZrule

You should now see a newly created directory named StoZrulePackage that contains the central Java source-code file StoZRule.java plus an sN.java file for each state in the original FSM, where N is 0 ... n. When we drew the FSM earlier, we saw that there were three states, and in StoZrulePackage we see s0.java, s1.java and s2.java representing those three states.

$ cd StoZrulePackage
$ ls
StoZrule.java s0.java s1.java s2.java
The new Java class \texttt{StoZrule} can be imported and used inside any Java program. Type the following minimal example into a file named \texttt{Test.java} in the same directory where \texttt{StoZrulePackage} resides.

// Test.java
// For testing fst2java for Kleene FSMs

import java.util.Scanner;
// import the new package/class
import StoZrulePackage.StoZrule;

public class Test{
    public static void main(String[] args) {
        String result;

        // create an instance of the StoZrule class
        StoZrule rule = new StoZrule();

        // create an instance of Scanner to read
        // input from the terminal
        Scanner scanner = new Scanner(System.in);
        while (true) {
            // output a prompt
            System.out.print("Enter input string: ");
            // read the input string entered by the user
            String input = scanner.nextLine().trim();

            // if the input string is 'quit' or 'exit',
            // then exit the loop and terminate
            if (input.equals("quit") || input.equals("exit"))
                break;

            // call the xapply(String) method of StoZrule,
// one of several available methods
result = rule.xapply(input) ;

System.out.println(result) ;
}
}

This particular example creates an instance of StoZrule, named rule, gets an input string entered by the user (you), and applies StoZrule's xapply() method to that input string. The method call returns an output string in an XML format and prints it to the terminal.

<results>
  <result><str>caza</str><w>0.0</w></result>
</results>

In addition to xapply(String str) there are several other application methods available; these methods and the XML format will be documented below.

To compile the Java file, use the javac command:

$ javac Test.java

This creates a file named Test.class, which you can run with the java command.

$ java Test

The system will then display the prompt, wait for you to type a string and press the Enter key, print the result, and then re-display the prompt for the next input. Try entering casa.

---

4If your system does not have javac, you need to install the JDK, the Java Development Kit. See your system administrator or a Java-savvy friend.
Enter input string: casa
caza
Enter input string:

Then try entering other words such as *elephant*, *sopa*, *sasesisosus*, etc. to make sure that the output is correct, mapping all (and only) *ss* that appear between two vowels into *z*. Note that when a word does not contain an *s* at all (such as *elephant*), and when a word contains an *s*, but not between two vowels (as in *sopa*), the input string is mapped to the output without change—the rule simply does not apply to such strings.

Enter *exit* or *quit* to break the loop and terminate the program.

**An Esperanto Verb Example**

As a second example, we will build a small morphological analyzer FST that models a fragment of Esperanto verbs. In particular, we'll model verbs that start with an optional prefix *mal*, meaning opposite, or *ne*, meaning negative; followed by a required verb root; followed by an optional aspect marker suffix *ad*, indicating repetitive/habitual/imperfect; followed by a required verb suffix indicating tense/mood. Type the following into a Kleene script file, named something like *EspVerb.kl*.

```kl
// EspVerb.kl
$vpref = '[Op]':(mal) | '[Neg]':(ne) ; // opposite, negative
$vroot = pens | don | ir | dir | est ;
// think give go say be
$vrep = '[Rep]':(ad) ; // repetitive aspect/imperfect
$vend = '[Pres]':(as) | // present tense
' [Past]':(is) | // past tense
' [Fut]':(os) | // future tense
' [Cond]':(us) | // conditional
' [Subj]':u | // subjunctive
' [Inf]':i ; // infinitive
```
$EspVerb = $vpref? $vroot '^[Verb]' : "" $vrep? $vend ;

and then, inside the Kleene GUI, invoke

source "EspVerb.kl" ;
test $EspVerb ;

This will bring up the now-familiar testing window, and let’s assume that that want to test this FSM in analysis mode, applying it in an upward direction to surface orthographical strings like *estos* and *donadis*, getting back analysis strings like *est*[Verb][Fut] and *don*[Verb][Rep][Past], respectively. To do this we enter strings like *donadis* in the lower input field of the testing window so that they are matched against the lower side of the FSM, and the results are read off of the upper side of the matched path(s). Test examples like *estas*, *estis*, *malpensos*, *neestos*, *irus*, *diri* and *donu*.

To generate Java code that performs this same upward-oriented analysis, recall that the Java code always matches input strings against what was the upper side of the FSM, what OpenFst called the “input” side. This FSM is effectively upside-down for our intended purpose; so before we generate the Java code, we need to invert this particular FSM, and we could call the result something like $EspVerbInv or $EspVerbAnalyze.

$EspVerbAnalyze = $\text{invert}(EspVerb) ;

And then we can test the result

test $EspVerbAnalyze ;

noting that now we enter strings like *malpensos* in the upper input field of the test window. This is the orientation we need before we generate the Java code.

Now we generate the state-oriented XML file in the usual way:

writeXmlStateOriented $EspVerbAnalyze, "EspVerbAnalyze.xml", "UTF-8" ;
We then exit Kleene, cd to the fst2java directory and in a command-line terminal do the following:

$ ln -s /path/to/EspVerbAnalyze.xml
$ make name=EspVerbAnalyze validate
$ make name=EspVerbAnalyze

The result will be a directory named EspVerbAnalyzePackage containing EspVerbAnalyze.java and a Java file for each state in the original FSM. We can test this in Java using the following code, which now uses EspVerbAnalyze where we previously had StoZrule.

```java
import java.util.Scanner;
import EspVerbAnalyzePackage.EspVerbAnalyze;

public class Test {
    public static void main(String[] args) {
        String result;

        EspVerbAnalyze esp = new EspVerbAnalyze();

        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.print("Enter input string: ");
            String input = scanner.nextLine().trim();
            if (input.equals("quit") || input.equals("exit"))
                break;

            // call the xapply(String) method of EspVerbAnalyze
            result = esp.xapply(input);

            System.out.println(result);
        }
    }
```
Compile and run the program as shown for the StoZrule example above, and you should be able to enter strings like *donadis* and get back strings like *don[Verb][Rep][Past]*. [KRB: test and show output in XML]

**Multiple Output Example**

In some languages, certainly including English, many orthographical words are ambiguous, requiring a morphological analyzer to return multiple results. Consider the following example that models a small fragment of English morphotactics with some ambiguities:

```plaintext
$vroot = \text{think} | \text{look} | \text{file} ;
$nroot = \text{desk} | \text{look} | \text{file} ;
$nouns = $nroot '[\text{Noun}]:"" ( '[\text{Sg}]:"" | '[\text{Pl}]:s ) ;
$verbs = $vroot '[\text{Verb}]:""
 ( '[\text{Bare}]:"" | '[\text{3PS}]:s | '[\text{PresP}]:(:ing) ) ;

$Eng = $nouns | $verbs ;
```

This grammar captures the fact that words like *look* and *files* are ambiguous, being either a noun or a verb. If we compile this little grammar in Kleene, creating $Eng, and then test it manually:

```plaintext
test $Eng ;
```

we can enter *files* in the *lower* input field and get back two solutions:

```plaintext
file[Noun][Pl]
file[Verb][3PS]
```
Note that this FSM, as we have built it, has analysis strings on the upper side, and orthographical strings on the lower side. Assuming that we want to generate Java code that performs analysis, mapping from orthographical strings to analysis strings, we will need to invert this FSM before generating the state-oriented XML output file.

\[$\text{EngAnalysis} = \$^\text{invert}(\text{Eng}) ;\]
\[
\text{writeXmlStateOriented} \ \text{EngAnalysis}, \ "\text{EngAnalysis.xml}" , \ "\text{UTF-8}" ;
\]

If you follow the steps of the previous examples to generate the Java code and modify the Java test script to run it, you should be able to enter files and get back

\[
\begin{aligned}
&\langle \text{results} \rangle \\
&\langle \text{result} \rangle <\text{str}>\text{file}[\text{Noun}][\text{Pl}]</\text{str}> <\text{w}>0.0</\text{w}> </\text{result}> \\
&\langle \text{result} \rangle <\text{str}>\text{file}[\text{Verb}][3PS]</\text{str}> <\text{w}>0.0</\text{w}> </\text{result}>
\end{aligned}
\]

showing that the orthographical word files can be analyzed either as a plural noun or a third-person-singular verb.

The generated Java code contains other methods that return the results not as XML but as an ArrayList of String, as in the following example:

```java
import java.util.Scanner ;
import EspVerbAnalyzePackage.EspVerbAnalyze ;

public class Test {
    public static void main(String[] args) {
        ArrayList<String> result ; // note the ArrayList here

       EspVerbAnalyze esp = new EspVerbAnalyze() ;

        Scanner scanner = new Scanner(System.in) ;
        while (true) {
```
System.out.print("Enter input string: ");
String input = scanner.nextLine().trim();
if (input.equals("quit") || input.equals("exit")) break;

// call the apply(String) method of EspVerbAnalyze
// Note that examples above used the xapply(String str)
// method.
result = esp.apply(input);

// iterate through the results in the ArrayList, if any,
// and print them out
if (result.isEmpty())
    System.out.println("***No Results***");
else
    for (String item: results)
        System.out.println(item);

The methods that return ArrayList, as well as those that return an XML String, are all documented below.

11.2.5. Generated Java Code API

Methods that Return an ArrayList of String

The generated Java class file contains the following public methods, all of which match the input string on the “input” side of the original FSM, but they differ in the way that the results are returned.

    public ArrayList<String> apply(String str)
This method tokenizes the input string (looking for multi-character symbols), applies the FSM to the String argument,\(^5\) and returns the result(s) as an ArrayList of String. An output label OTHER_NONID in the original FSM is represented as ? (the normal question mark) and the string output is separated from the weight by :: (two colons).

```java
public ArrayList<String> apply(String str,
                                Boolean tokenizeInputString)
```

As for `apply(String str)` except that the second argument controls whether tokenization (to look for multi-character symbols) is performed. If the input strings are surface, orthographical strings, then they typically contain no multi-character symbols, and the tokenization step can be skipped, improving performance.

```java
public ArrayList<String> apply(String str,
                                String other_nonid)
```

As for `apply(String str)` except that the second argument indicates how an output label OTHER_NONID should be displayed in output strings. (The default, when calling `apply(String str)` is ?.

```java
public ArrayList<String> apply(String str,
                                String other_nonid,
                                Boolean tokenizeInputString)
```

As for `apply(String str)` except that the second argument indicates how OTHER_NONID should be displayed, and the third argument controls whether tokenization (to look for multi-character symbols) is performed.

\(^5\)More precisely, the string is converted to a list of integer code point values, and the FSM is effectively applied to that list of code point values.
As in the examples above, plus additional arguments to specify custom markup: mcs_start and mcs_end indicate strings to surround multi-character symbols; str_start and str_end indicate strings to surround string outputs, and weight_start and weight_end indicate strings to surround the weights.

### Methods that Return an XML String

The following methods return the results as a single XML String.

```java
public String xapply(String str)
```

This method returns a String in an XML format. When the output has exactly one string, the following structure is returned (whitespace added here for human readability):

```xml
/results>
  <result><str>string</str><w>weight</w></result>
</results>
```

If the output contains two results, then the structure is (again with added whitespace for readability)

```xml
/results>
  <result><str>string</str><w>weight</w></result>
  <result><str>string</str><w>weight</w></result>
</results>
```

and similarly for three or more results. A multi-character symbol is returned inside `<mcs>`...`</mcs>` tags, and any OTHER_NONID output symbol is represented as `<other/>`. 
public String xapply(String str,  
    Boolean tokenizeInputString)

As for xapply(String str) except that tokenizeInputString controls whether the input string is tokenized (to look for multi-character symbols). If the input strings are orthographical strings, which typically do not contain multicharacter symbols, then tokenization is not required and tokenizeInputString can be set to false for better performance.

public String xapply(String str,  
    String other,  
    String mcs_tag,  
    String str_tag,  
    String weight_tag,  
    String result_tag,  
    String results_tag,  
    Boolean tokenizeInputString)

As above except that additional argument allow the user to control the spelling of the XML tags. For example, if the other value is “unknown”, then any OTHER_NONID output symbol will be represented in the XML as <unknown/> . If mcs_tag is “mult”, then any multi-character symbols in the output will be surrounded with <mult>...</mult>. Similarly, str_tag specifies the spelling of the tags surrounding string outputs, weight_tag specifies the spelling of the tags surrounding weights, result_tag specifies the spelling of the tags surrounding each result, and results_tag specifies the spelling of the top-level XML tags surrounding all the results.

11.3. Runtime Code

11.3.1. Status of Runtime Code

There is currently no traditional runtime code available for running Kleene FSMs. Perhaps some interested readers would be interested in writing such
code. This section give a high-level overview of what runtime code would do.

See the section above entitled “The fst2java Experiment” for a way to convert a Kleene FSM into a Java class that can be imported and used by any Java program or any other language (Scala, Groovy, Clojure) that is based on the Java Virtual Machine (JVM).  

11.3.2. Basic Functionality of Runtime Code

If we have built an FSM in Kleene and saved it to file as XML (or some similar markup language), runtime code is code, external to Kleene itself, that would perform the following tasks:

1. Read the XML file representing a Kleene FSM and build a representation of that FSM in memory
2. Accept input in the form of a string
3. Convert the input string into a list of Unicode code point values, taking account of any multi-character symbols in the alphabet of the FSM
4. Apply the FSM to the list of code point values (representing the input string)
5. Collect the output string or strings, and
6. Return them in an appropriate form to the caller

The application capabilities might include downward application and/or upward application. Downward application involves matching the list of code point values (representing the input string) against paths of labels

---

6Programs in Java, Scala, Groovy and Clojure are compiled into byte-code that is then interpreted by the JVM (Java Virtual Machine) at runtime. JVM programs run on all popular platforms and operating systems.
on the upper (OpenFst “input”) side of the FSM, and returning the lists of labels on the lower (OpenFst “output”) side of the matched paths. Conversely, upward application involves matching the list of code point values against paths of labels on the lower (OpenFst “output”) side of the FSM, and returning the lists of labels on the upper (OpenFst “input”) side of the matched paths.

The runtime code would typically be implemented as a library, for example a C++ library that could be loaded by a C++ program. The library would have an API listing the callable functions of the library that implement the tasks just listed. Any larger program that imported the runtime-code library would be able to integrate FSM processing.

11.3.3. Kinds of Runtime Code

Complete Match

FSMs can be used in several modes, and different modes may require different kinds of runtime code. The most straightforward way to use FSMs involves taking an input string (or a list of code point values computed from that string) and looking for complete matches of those code point values against complete paths of arc labels in the FSM. A complete or full match starts at the FSM’s start state and ends at one of the FSM’s final states, with none of the input string’s symbols (code point values) remaining unmatched. Full matching behavior is typical of applications including

- Tokenization
- Phonological modeling
- Morphological analysis and generation

In morphological analysis, for example, the input is typically a string representing an orthographical word, and the entire string should completely match a path in the FSM.
Partial Match

Another desirable modality in many applications is to view the FSM as a union of patterns to be matched anywhere inside the input string, which might be a long file. This could be called *partial matching* because any given pattern need match only part of the input string. Many patterns/paths in the FSM might match somewhere inside the input string, and any single pattern/path in the FSM might match multiple times inside the input string. Parts of the input string—which, again, might be long file—may not be matched by any pattern. Partial matching behavior is typical of applications such as

- Entity extraction, e.g. finding names of people, companies, countries, etc. in an input text
- Sentiment analysis
- Shallow/robust parsing

One possible strategy for partial-matching is to implement runtime code that begins by setting a *match pointer* at the beginning of the input string and trying to find a pattern from the FSM that matches at least part of the input string, starting at the match pointer. If a single pattern matches, then the start and end of the match is stored as a result/success, the match pointer is advanced to the end of the matched input segment, and the matching procedure is restarted from that new point. If no patterns match, then the match pointer is advanced to the next symbol or word, as appropriate for the application. If multiple patterns match starting at the same match point in the input, then the runtime code might be written to record all matches, or to prefer the longest match. Many variations in matching behavior are possible, depending on the demands of the larger application.

Eventually, it would be useful to have runtime code to implement complete-matching and partial-matching strategies for Kleene FSMs, perhaps in several semantic variations and in several computer languages,
to allow FSM-based functionality to be integrated into arbitrary software applications. The difficulty of writing correct and robust runtime code should not be underestimated, and runtime code must be highly tuned to achieve maximum performance.
Chapter 12

RTN Support

12.1. OpenFst Recursive Transition Networks (RTNs)

12.1.1. Status

[KRB 2013-04] WORK IN PROGRESS: There is currently no runtime-code support for RTNs. This chapter has not be reviewed recently.

Kleene supports the building of Recursive Transition Networks (RTNs) that are compatible with the OpenFst Replace() and ReplaceFst() operations. An RTN contains arc labels that are to be interpreted as “references” to subnetworks; and compatible RTN-savvy runtime code, when applying a network to data, will recognize such a reference and “push” to the referenced subnetwork to continue the matching, and then “pop” back to the calling network when the subnetwork has successfully matched. The references to subnetworks can also be thought of as non-terminal labels.

RTNs, containing references to subnetworks, are often smaller than full-sized networks that must contain a full copy of each subnetwork wherever it is needed. However, RTNs can denote context-free languages, and so can go beyond regular power; a special subclass of RTNs remain regular. The recognition/parsing of context-free languages requires memory,
in particular a push-down stack, and so any runtime code to apply RTNs must include such a stack.

At the time of writing (13 October 2010) there are hints that the RTN conventions currently required in OpenFst might change, and there is new library support for mathematically equivalent Pushdown Automata (PDAs) that we have not yet tested. Kleene support for OpenFst RTNs and PDAs will necessarily evolve along with the library.

12.1.2. Syntax for Creating an OpenFst RTN

For the programmer, there needs to be a Kleene syntax to denote a reference to a subnetwork, and it needs to be distinct from the syntax that causes a copy of a network to be inserted. For example, in this example

```plaintext
$vowel = [aeiou];
$net = k $vowel t $vowel b $vowel;
```

the $vowel network will be copied three times into $net. In real-life applications, a subnetwork might encode something much larger, such as nouns or even noun phrases in a natural language, and multiple copies could easily cause the final network to become very large, or even too large to compute.

Kleene currently supports a wired-in function $^\text{sub}(s)$ that programmers can use to denote a reference to a subnetwork $s$.

To continue with our trivial example,

```plaintext
$vowel = [aeiou];
$rtn = k ^\text{sub}($vowel$) t ^\text{sub}($vowel$) b ^\text{sub}($vowel$);
```

would result in an $rtn$ network that contains three compact one-label references to $vowel$ rather than three full copies of it. In real-life applications, the savings in memory can be very significant.

---

1I'm not tied to $^\text{sub}(s)$ and would be comfortable with alternatives such as $^\text{ref}(s)$ or $^\text{push}(s)$, or even some special syntax like $\gg s$ or $\ggg s$. 
12.1.3. What does a Reference Look Like in an OpenFst RTN?

OpenFst networks consist of states and arcs, and each arc has two labels, an input label, and an output label.\(^2\) In the actual network, the labels are really integers, and all Unicode characters, such as a, b, c, etc. are represented using their standard Unicode code point values. Multichar symbols are stored using code point values selected at random from a Unicode Private Use Area.

To maximize compatibility with the off-the-shelf OpenFst Replace() and ReplaceFst() operations, the integer representing a reference to a subnetwork must currently appear on the output side of an arc. In Kleene, multichar symbol names representing a call to a subnetwork $s$ are spelled __$s$, with two initial underscores, and the syntax $^\text{sub}(s)$ currently yields the following network.

![Diagram](image)

Currenty in OpenFst, the symbol __$s$ could in fact appear also on the input side. Of course, in the real network, the labels are really just integers, 0 for epsilon, and some arbitrary value from the Private Use Area for the multichar symbol.

Kleene programmers should use the wired-in $^\text{sub}()$ function to denote references to subnetworks and should not try to specify special symbols like __$s$ directly. For example, the following statement is illegal and will cause an exception to be thrown.

```plaintext
$rtn = "":'__$foo' ;   // raises an exception
```

\(^2\)The terminology of input and output labels is that of the OpenFst tradition. In the Xerox tradition, they are called upper and lower, or sometimes lexical and surface.
The use of $^\text{sub()}$ at the programming level will also make it easy for Kleene to adapt to any changes to RTN representations that might be made in the underlying OpenFst library.

### 12.1.4. Embedding Subnetworks in an OpenFst RTN

The $^\text{embedRtnSubnets()}$ function takes an OpenFst RTN argument, i.e. a network that contains OpenFst-format references to subnetworks, and returns a network that consists of the original network unioned with a prefixed copy of each referred-to subnetwork. For each referred-to network $s$, the prefix consists of the special symbol __SUBNETWORKS followed by the special symbol __$s$. Thus from the following code

```plaintext
$p = p ;$
$q = q ;$
$rtn = a ^\text{sub}(p) b ^\text{sub}(q) ;$
```

the result $rtn$ is this network containing two references to subnetworks

![Diagram 1](image1.png)

SIGMA: b, a, __$q, __$p

but the three referred-to networks remain separate. After the following call to $^\text{embedRtnSubnets()}$,

```plaintext
$embedded = ^\text{embedRtnSubnets}(rtn) ;$
```

the resulting $embedded$ network looks like this

![Diagram 2](image2.png)

SIGMA: b, q, p, a, __SUBNETWORKS, __$q, __$p
The embedded network can be saved to file as a single network and could be applied by RTN-savvy runtime code (not yet written) that knows the prefix convention.

The unioning of the base network with the subnetworks also guarantees that the meaning of OTHER, if present, is standardized throughout all the networks.

12.1.5. Expanding an OpenFst RTN into a Full Normal Network

In some cases, it may also be useful to take an OpenFst RTN and replace each reference to a subnetwork with an actual copy of that subnetwork, expanding the RTN into a full normal network. This is mathematically and computationally possible only if the RTN is regular. Using the same $r tn example, a call to $^expandRtn()

$expanded = $^expandRtn($r tn) ;

produces the following $expanded network

In real-life RTNs, with multiple references to the same subnetwork, the expanded network will be significantly bigger than the original.

If the RTN contains cyclic references to subnetworks, it is context-free in power (i.e. no longer regular in power), and its expansion would result in an infinite network. The $^expandRtn($r tn) function checks for cyclic references and throws an exception if the $r tn is not regular.\(^{3}\)

\(^{3}\)In the OpenFst library, the ReplaceFst() operation provides a convenient method that checks for cyclic references, and $^expandRtn() invokes this method.
Chapter 13

Unicode Support

13.1. Kleene, Java and Unicode

The Kleene parser is a Java program, and Kleene supports Unicode to the extent (and in the same way) that any Java program supports Unicode, which is pretty well—but not perfectly.\(^1\) The Kleene GUI is written using the Java Swing library, and Swing text widgets—including JTextField and JTextArea—are automatically Unicode-friendly.

13.2. Kleene Scripts and Unicode

13.2.1. The Default Encoding of the Operating System

Pre-edited Kleene scripts can be run from the command line, or from the GUI. It is not required that Kleene scripts be stored as Unicode—Kleene can read and execute scripts written in a huge number of standard encodings, converting the text to Unicode for internal processing. If Kleene is told to

\(^1\)The Unicode Standard (http://www.unicode.org) is mature and very comprehensive, but the implementation of Unicode in programming languages, text editors, GUI libraries, typesetting packages, and other text-handling software varies considerably in completeness and reliability.
run a script, and if the encoding of that script is not explicitly specified, then by default it will assume that the script is in the default encoding of the operating system, whatever that might be.

$ java -jar Kleene.jar myscript

In this example, myscript would be assumed to be stored in the default encoding of the operating system and would be opened and read as such, with Java performing an automatic conversion of the text from that default encoding to Java's internal Unicode encoding, which happens to be UTF-16.

Note that Kleene does not attempt to analyze the input file and detect what its encoding might be from internal evidence. Rather it interrogates the operating system to find the default encoding, and it attempts to open and read the file accordingly. If, for some reason, the input file is not in the default encoding of the operating system, the input may result in a warning message or garbling.

In a Unix-like system, the default encoding of the operating system can be seen by entering locale at the command line:

$ locale

On any system, the default encoding, as seen by a Java program, can also be revealed by compiling and running the following trivial Java script, which should be in a file named FindDefaultEncoding.java.

```java
public class FindDefaultEncoding {
    public static void main(String[] args) {
        String s = System.getProperty("file.encoding") ;
        System.out.println(s) ;
    }
}
```

To compile and run this script, do the following:
$ javac FindDefaultEncoding.java
$ java FindDefaultEncoding

On OS X even if the locale has been set to some specified encoding, such as UTF-8, the call to System.getProperty("file.encoding") will still, unfortunately, return MacRoman. This can be overcome by calling any Java program with a -D option specifying the desired default encoding.

$ javac FindDefaultEncoding.java
$ java -Dfile.encoding=UTF-8 FindDefaultEncoding

On my own Mac, where I’ve set the locale to UTF-8 and edit all my Kleene scripts in UTF-8, I launch Kleene with the following command:

$ java -Dfile.encoding=UTF-8 -jar Kleene.jar

### 13.2.2. Running Scripts in Any Standard Encoding

Regardless of the default encoding of the operating system, Kleene can run scripts in any standard encoding as long as that encoding is explicitly specified using the -encoding flag. So if the script is, for some reason, stored in an encoding that is not the default encoding of the operating system, it is necessary to specify that encoding as in the following examples:

$ java -jar Kleene.jar -encoding UTF-8 myscript
$ java -jar Kleene.jar -encoding UTF-16 myscript
$ java -jar Kleene.jar -encoding Latin-1 myscript
$ java -jar Kleene.jar -encoding ISO-8859-1 myscript
$ java -jar Kleene.jar -encoding ISO-8859-6 myscript
$ java -jar Kleene.jar -encoding EUC-JP myscript

2Curiously, in the Groovy and Scala languages, which are, like Java, based on the Java Virtual Machine (JVM), this problem appears to have been fixed. On a system where the locale is set to UTF-8, executing println(System.getProperty("file.encoding")) in Scala 2.8.1 or Groovy 1.7.5 returns "UTF-8".

3This -encoding flag, and its semantics, are copied from the same flag used for the javac compiler; see http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javac.html.
13.2.3. A Plug for Unicode

All things being equal, users working on languages with orthographies that cannot be represented in ASCII are highly encouraged to use Unicode rather than resorting to obsolete 8-bit encodings or, especially, to Roman transliterations that may have been required in pre-Unicodesoftware.\footnote{The E-MELD School of Best Practices recommends the use of Unicode for all textual archiving: http://emeld.org/school/bpnutshell.html; http://emeld.org/school/classroom/unicode/index.html.}

13.3. Typing Unicode Characters into Kleene GUI Text Widgets

13.3.1. Unicode-capable Text Widgets

Because the Kleene GUI is written in Java/Swing, the text widgets are automatically Unicode-capable and sensitive to standard Java Input Methods. At any time when typing text into a Swing text widget, you can select (or “activate”) a particular Java Input Method of your choice (as long as it’s installed in your Java environment) to facilitate typing in Unicode characters for 1) European scripts with various accents, 2) IPA, 2) Greek, 3) Russian, 4) Arabic, 5) Chinese or whatever. You can switch from one input method to another at any time. In their simplest form, Java Input Methods define a straightforward remapping of the keyboard; they can also support code-point (hex) input of characters, dead-key sequences, transliteration-based input methods, and more challenging dialog-based input methods for the Chinese/Japanese/Korean scripts.

13.3.2. Java Input Methods

Java Input Methods are standard, well-documented, cross-platform, and often freely available; some useful Java Input Methods will be distributed
with Kleene. You can install and use your own favorite Java Input Methods in your own Java installation. The key documentation on installing and selecting Java Input Methods, for users, is “Using Input Methods on the Java Platform”, by Naoto Sato.⁵

### 13.3.3. Installing Java Input Methods

To run any Java program, including Kleene, you need to have a Java installation on your system. If you can run Kleene at all, you have such a Java installation. To see which version of Java you have installed, enter

```
$ java -version
```

You should be running Java 1.6 or higher.

The root of your Java installation should (on most platforms) be pointed at by the environment variable `JAVA_HOME`.

```
$ echo $JAVA_HOME
```

On OS X, `/Library/Java/Home` should be a link to the root of the Java installation, within the rather complicated hierarchy of OS X frameworks. So on OS X if you need to define `JAVA_HOME`, set it to `/Library/Java/Home`.

Your Java installation has an Extension Directory where you can install Extensions, including Java Input Methods.⁶ On OS X, user-installed Java extensions are most safely put in `/Library/Java/Extensions/`; this directory stays stable when you upgrade to new versions of Java. On any system, compile and run the following Java program to find the extension directory or directories.

---

⁵[http://javadesktop.org/articles/InputMethod/index.html](http://javadesktop.org/articles/InputMethod/index.html)

⁶If you work on a network, with a shared Java installation, you might not have permission to copy extensions to the extension directory, and then you would need to contact your administrator for help.
public class FindExtensionDirectory {
    public static void main(String[] args) {
        System.out.println(System.getProperty("java.ext.dirs"));
    }
}

To compile and run this script, which should be stored in a file named FindExtensionDirectory.java:

$ javac FindExtensionDirectory.java
$ java FindExtensionDirectory

The output shows the directory, or set of directories, where you can install new extensions. Once an input method is installed in the extension directory of the JVM, it is automatically visible to all Java programs running that JVM, without recompilation, without resetting your CLASSPATH, and without use of the -D command-line flag.

Java usually comes complete with some built-in input methods, such as CodePointIM.jar, which is found in \$JAVA_HOME/demo/plugin/jfc/CodePointIM/ or \$JAVA_HOME/demo/jfc/CodePointIM/. This should make CodePointIM.jar available without you having to install anything.

Some pure Java Input Methods are readily downloadable:

<table>
<thead>
<tr>
<th>CodePointIM.jar</th>
<th>enter Unicode characters by Unicode code-point value</th>
</tr>
</thead>
<tbody>
<tr>
<td>zh_pinyin.jar</td>
<td>Chinese</td>
</tr>
<tr>
<td>vietIM.jar</td>
<td>Vietnamese</td>
</tr>
<tr>
<td>BrahmaKannadaIM.jar</td>
<td>Brahmî</td>
</tr>
<tr>
<td>BibleIM.jar</td>
<td>Hebrew and Greek</td>
</tr>
</tbody>
</table>

Running this program on my Linux system produced the output /usr/java/jdk1.6.0_16/jre/lib/ext/usr/java/packages/lib/ext, showing two extension directories. In this case, putting Java Input Methods in /usr/java/packages/lib/ext is safer because they will be unaffected by future Java updates.
Reasonably skilled Java programmers can even write their own custom Java Input Methods, but it's not painless.\(^\text{13}\) Luckily, as an easy alternative, there is a very useful meta-Java Input Method, installable as kmap_ime.jar and kmap_ime_gui.jar, that allows you to use simple Yudit kmap files\(^\text{14}\) as if they were Java Input Methods. Yudit kmap files are much easier to write than pure Java Input methods, and many are already included with kmap_ime. These kmap meta-Java Input Methods are treated in detail below.

### 13.3.4. The CodePointIM.jar

When you need to enter a single exotic character, or even just a word or two, it is often easiest to simply enter each character by its code point value. When the CodePointIM.jar is selected, you can continue to type ASCII-range characters as normal, but the sequence

\[
\backslash uHHHH
\]

where HHHH is exactly four hexadecimal digits, is intercepted, and the single Unicode character with the code point value HHHH is entered into the buffer. Similarly, the sequence

\[
\backslash uHHHHHHH
\]

\(^\text{8}\)http://www.grogy.com/local_doc/www/apache22/data/local_doc/jdk1.6.0/demo/jfc/CodePointIM/, http://courses.cs.tau.ac.il/databases/workshop/jdk1.6.0_01/jdk1.6.0_01/demo/jfc/CodePointIM/

\(^\text{9}\)http://www.chinesecomputing.com/programming/java.html


\(^\text{11}\)http://sourceforge.net/projects/brahmi

\(^\text{12}\)http://code.google.com/p/bibleunicodepad/

\(^\text{13}\)A Java Input Method is a Java class that implements the InputMethod interface. For information on writing new input methods, and packaging them properly as extensions, see the book *Java Internationalization* (Deitsch and Czarnecki, 2001).

\(^\text{14}\)They also allow you to use Simredo kmp files, which I haven't used myself.
with an uppercase U followed by exactly six hexadecimal digits, is intercepted and the single supplementary Unicode character with the code point value \HHHHHH is entered into the buffer.

Note that Kleene, like Python, accepts the \UHHHHHHHH notation, which requires eight hex digits rather than the six required by the CodePointIM. Kleene also accepts the \U{H...} notation, which allows one or more hex digits between the curly braces.)

13.3.5. The kmap Meta-Java Input Method

There's a very useful meta-Java Input Method called kmap_ime, or kmap for short; the extension files kmap_ime.jar and kmap_ime_gui.jar can be downloaded from http://sourceforge.net/projects/jgim. Install these extensions in a Java extension directory as usual. The kmap extension allows you to use Yudit kmap input methods as if they were pure Java Input Methods. It is much easier to write a Yudit kmap file than it is to write a new Java Input Method. Many kmap files are freely available, and instructions for writing new Yudit kmap files can be found at http://www.yudit.org/en/howto/keymap/.

To be visible to this meta-Java Input Method, your Yudit kmap files should be placed in the directory ~/kmap/. The kmap download comes complete with a kmap/ directory containing over 200 kmap files, defining input methods for everything from Albanian to Yoruba. Of course, any single user will probably need only a small subset of those input methods, and perhaps a different subset from time to time. It also appears that the input-method-selection dialog does not allow scrolling through the 200 possibilities. One reasonable approach is to keep the full set of input methods available in a repository directory, named something like ~/kmap.all/, and copy a desired subset of the kmap files to ~/kmap/. I.e., starting with the kmap/ directory as downloaded:

```
$ cp -r kmap ~/kmap.all
$ cd ~
```
$ mkdir kmap

and then copy some desired subset of the kmap files from `~/kmap.all/` to `~/kmap/`. Put any new kmap files that you write or modify yourself into `~/kmap/`. Again, only the kmap files in `~/kmap/` will be visible to the kmap meta-Java Input Method.

The kmap download also includes a `simredo/` directory containing over 100 input methods in the Simredo kmp format. Kmap can recognize and use these kmp files as well. To make these Simredo files visible to kmap, create a `~/simredo/` directory on the model of the `~/kmap/` directory just described.

### 13.4. Selecting a Java Input Method

How you select a Java Input Method depends on your platform, and this subject is treated in detail in the Sato paper cited above. When running Java on Solaris and Windows, the System pull-down menu automatically contains a menu item allowing you to select one of the installed Java Input Methods.

On Linux and OS X it's not quite so easy; you need to define a “hot key” that triggers a pop-up menu that allows you to select one of the installed Java Input Methods. The JAR file `InputMethodHotKey.jar`, used to set the hot key, can be downloaded from the Sato paper cited, and it is run as

```
$ java -jar InputMethodHotKey.jar
```

You can re-run `InputMethodHotKey.jar` as often as desired or necessary to reset the hot key. Again, the setting of the hot key is necessary only on Linux and OS X.\textsuperscript{15}

\textsuperscript{15}For example, I use Ctrl-i as my hot key on Linux and OS X. When `InputMethodHotKey.jar` is launched, a little dialog box is displayed showing the current setting, if any. To set or reset the hot key, wait for the box to be displayed and then press Ctrl-i, or whatever, directly on your keyboard. Then close the dialog box and restart Java.
When you select the kmap input method (via the System menu, or via a hot key), the menu lets you make a secondary selection of one of the Yudit kmap files installed in your `~/kmap/` directory, or one of the Simredo kmp files installed in your `~/simredo/` directory.

### 13.5. Rendering of Unicode Characters in the GUI

Entering Unicode characters into a Swing text widget is one thing—seeing them rendered properly on the screen is another. Whether your Unicode characters render properly will depend on the fonts installed in your Java installation, and on the sophistication of the rendering engines in the Swing text widgets themselves.

Font installation varies by platform and by Java version; see your documentation. Rendering engines are an area where implementations, including the allegedly Unicode-savvy Swing text widgets, can be disappointing. The more “exotic” the script, the less likely the text widgets will render them properly. If rendering becomes an issue with your applications, edit Kleene source files outside the GUI using your favorite Unicode-savvy text editor, and run them as scripts.

**KRB Note to myself:** Try to be more helpful here. In my own experience with Swing Text Widgets, the rendering of letters with Combining Diacritical Marks is disappointing, sometimes unacceptable. Of course, Swing Text Widgets *should* render Combining Diacritical Marks acceptably, so it is up to me and others to lobby for improvements.
13.6. Java Input Method Conclusions

Preferences about input methods, as about text editors, are intensely personal and often almost religious. My own needs for entering Unicode include the International Phonetic Alphabet (IPA), Roman letters with unusual Combining Diacritical Marks, and very occasionally Greek, Cyrillic and Georgian letters. Some non-ASCII mathematical symbols can be used in Kleene syntax itself. And I insist on being able to write my own input methods to match my needs and taste.

With this background, my favorite approach to entering exotic Unicode characters in Kleene, and in any Java GUI, is the use of the kmap_ime.jar and kmap_ime_gui.jar meta-input methods, along with easily defined and edited Yudit kmap files. I will eventually deliver with Kleene a kmap file that facilitates typing in some of the Unicode characters used in Kleene syntax, such as the epsilon $\epsilon$ and the $\circ$ symbol that can be used to indicate composition. You can also type in the ASCII _e_ and _o_ for these (and Kleene will parse the ASCII forms without trouble); but the Kleene kmap (when active) will simply intercept the typed sequences _e_ and _o_ and turn them into the real epsilon and composition-operator characters, respectively.
Appendix A

Alphabet (Sigma) and OTHER/UNKNOWN Characters

Kleene automatically keeps track of the set of symbols that are “known” to each network, and this set is known traditionally as the alphabet or sigma. Each Kleene network carries its own private sigma.\(^1\) In the simple case
\[ v = abc; \]
the sigma of network $v$ will contain the symbols $a$, $b$, $c$ and no other symbols. The Kleene programmer never has to declare sigmas manually.

In Kleene regular expressions, the . (dot) special syntactic symbol semantically represents any symbol, and has a non-trivial semantics. Intuitively, the notion of any symbol properly includes all known symbols in the sigma, plus the infinite set of OTHER, also known as UNKNOWN, symbols. We will use the term OTHER herein.

Kleene transducers must also distinguish between the identity mapping of OTHER symbols vs. the non-identity mapping of OTHER symbols. The identity mapping maps any OTHER symbol to itself, while the non-identity mapping maps any OTHER symbol to any OTHER symbol except itself. The difference is best seen in concrete examples.

\(^1\)The OpenFst library does not maintain a private sigma for each network; this functionality is added at the Java level of Kleene.
When . (dot) appears in a regular expression like

$w = . ;$

it is interpreted to produce the arc label OTHER_ID:OTHER_ID, which represents the identity mapping of OTHER symbols, i.e. the mapping of any OTHER symbol to itself. As the sigma of “known” symbols in $w$ is empty, the resulting network maps $a$ to $a$, $b$ to $b$, $c$ to $c$, and similarly for all possible symbols—but not $a$ to $b$ because this would be a non-identity mapping.

To denote the mapping of any symbol to any symbol, including itself, the Kleene syntax $:. is used in regular expressions, e.g.

$y = .:. ;$

The resulting network for $y$ contains a start state and a final state, linked by two arcs labeled OTHER_ID:OTHER_ID and OTHER_NONID:OTHER_NONID, respectively. As the name implies, OTHER_NONID:OTHER_NONID represents the non-identity mapping of OTHER symbols. The two arcs therefore handle the cases of identity mapping and non-identity mapping.

In Kleene networks, the symbol coverage of OTHER_ID and OTHER_NONID are identical; it can be thought of as OTHER, i.e. the set of all possible symbols not in the sigma of the network; the labels OTHER_ID:OTHER_ID and OTHER_NONID:OTHER_NONID differ only in their mapping behavior: identity mapping vs. non-identity mapping.

When two networks are combined, via operations like union, concatenation and composition, the result is a new network with its own sigma. Where one or both networks to be combined “contain OTHER”, the operation, and the calculation of the new sigma, can be quite complicated, but the programmer never needs to worry about it.

The notion of any symbol, denoted . (for “map any other symbol to itself”) or $:. (“map any other symbol to any other symbol, including itself”) are syntactic notions that appear in regular expressions. The notions of identity mapping, non-identity mapping and the labels OTHER_ID:OTHER_ID and OTHER_NONID:OTHER_NONID belong to underlying networks.
The arc labels OTHER_ID and OTHER_NONID are special, and these two special symbols should not appear in regular expressions.
Appendix B

Optimization of Networks

B.1. Automatic Optimization

The default behavior of Kleene is to optimize each network by calling the OpenFst Determinize(), Minimize() and RmEpsilon() functions wherever these operations are mathematically safe. Note that an operation that is mathematically safe can still overwhelm the available memory or take a very long time to complete.

In the Kleene interpreter, the Java function optimizeInPlace() is automatically called to optimize each new network, and it in turn calls a native C++ function called optimizeInPlaceNative(), which calls the OpenFst functions Determinize(), Minimize() and RmEpsilon(), where mathematically safe.

B.2. Suppressing Automatic Optimization

In some cases it may be desirable or necessary to suppress all or part of the automatic optimization that Kleene performs by default. For example, developers and advanced users might want to see what a network looks like before and after optimization. More critically, in real applications the networks can easily get very large, and it may be necessary to sup-

261
press some optimization, especially the determinization step, to keep the networks from blowing up in size and/or taking forever to finish.

Kleene defines (in the system-supplied start-up file ~/.kleene/global/predefined.kl) the following three special numeric variables which are interpreted as booleans (true or false):

- #KLEENEdeterminize
- #KLEENEminimize
- #KLEENErmepsilon

These variables display a naming convention that puts KLEENE in front of system variables that control the behavior of Kleene itself.

These variables can be set and interrogated in the usual ways, e.g.

```c
#KLEENEdeterminize = 0; // to turn off just determinization
#KLEENEdeterminize = 1;
```

or equivalently

```c
#KLEENEdeterminize = #false; // given that #false is set to 0
#KLEENEdeterminize = #true;
```

```c
if (#KLEENEdeterminize) {
    // then do something
} else {
    // do something else
}
```

These variables affect the behavior of the optimizeInPlace() algorithm, which is called routinely on newly created networks.

Also defined in predefined.kl are the following convenience functions:

```c
^setOptimize(#bool)
```

which sets #KLEENEdeterminize, #KLEENEminimize and #KLEENErmepsilon to the passed in #bool value,
e. g.
\`setOptimize(#false) ;

also
\`setDeterminize(#bool) ;
\`setMinimize(#bool) ;
\`setRmepsilon(#bool) ;

The global variables \#KLEENEdeterminize, \#KLEENEminimize and \#KLEENErmepsilon, defined in the global start-up file predefined.kl, can be shadowed by user-defined variables of the same name that are local to function code blocks and stand-alone code blocks. So programmers can play the scope game when turning these variables on and off.

**B.3. User-callable Optimization Functions**

Kleene provides built-in net-valued functions that take a network argument and return a new optimized network as the result. These functions are for use when the default optimization has been turned off. The non-destructive functions are

\$\text{\`optimize}(regexp) \\
\$\text{\`rmEpsilon}(regexp) \\
\$\text{\`determinize}(regexp) \\
\$\text{\`minimize}(regexp) \\
\$\text{\`synchronize}(regexp)

and the destructive versions are

\$\text{\`optimize!}(regexp) \\
\$\text{\`rmEpsilon!}(regexp) \\
\$\text{\`determinize!}(regexp) \\
\$\text{\`minimize!}(regexp)
The \texttt{synchronize!}(regexp) function causes—where mathematically safe—determinization, minimization and epsilon-removal to be invoked. More specific functions such as \texttt{determinize()} and \texttt{determinize!()} are also performed only when the operation is mathematically safe.

Kleene also supplies the following statements, which take one or more network-ID arguments, optionally separated by commas, and optimize them in place (destructively).

\begin{verbatim}
optimize $a, $b, ... ;
determinize $a, $b, ... ;
minimize $a, $b, ... ;
rmEpsilon $a, $b, ... ;
synchronize $a, $b, ... ;
\end{verbatim}

Just as the \texttt{rmEpsilon} command and the \texttt{rmEpsilon()} function remove two-sided $\eps$-$\eps$ arcs, the \texttt{synchronize} command and the \texttt{synchronize()} function return networks with a minimal number of one-sided epsilon arcs. For example, if the input network contains a four-arc path with the four labels \texttt{a:$\eps$, \texttt{b:$\eps$, \texttt{c:$\eps$, \texttt{d}}, in that sequence, the synchronized result path has only two labels, \texttt{a:b} and \texttt{c:d}. Note that the synchronized network is equivalent to the input network—i.e. it encodes the exact same relation; so synchronization is a lossless way to minimize the arcs in a network. As currently implemented, only acyclic networks can be synchronized. Cyclic networks can be synchronized only if the cycles (loops) themselves have equal input and output lengths, i.e. only if the cycles do not contain one-sided epsilons. KRB: I'm not sure yet how to test a cyclic network to make sure that the loops are consistent with this restriction.
Appendix C

Control Characters in Kleene

C.1. Characters Representing Whitespace

In Kleene source code, as in Java, Unicode characters specified by their code point value, such as \u000D (the CARRIAGE RETURN) or \u000A (the LINE FEED), are converted to the real Unicode character even before the tokenizer starts its work. So a statement typed as

```
$foo = \u000D ;
// or
$foo = \u000A ;
```

would look to the Kleene tokenizer/parser like

```
$foo =
;
```

and is not a valid statement because the right-hand side of the assignment is empty. But a statement typed as

```
$foo = a b c \u000D e f ;
```

would look to the tokenizer/parser like
The Kleene Language

```plaintext
$foo = a b c
e f ;
```

and would compile successfully and match the string "abcef". The carriage returns and line feeds (newlines) in these examples are just whitespace and are ignored by the tokenizer and the parser.

### C.2. Illegal Characters Inside Multichar-symbol Names and Double-Quoted Strings

Anything typed as

```plaintext
$foo = a b '\u000Dxyz' e f ;
```

would look to the tokenizer/parser like

```plaintext
$foo = a b '
xyz' e f ;
```

which is illegal (an attempt to put a newline inside single quotes) and will raise an exception. Similarly, anything typed as

```plaintext
$foo = a b "\u000Dxyz" e f ;
```

would look to the tokenizer/parser like

```plaintext
$foo = a b "
xyz" e f ;
```

and is similarly illegal (an attempt to put a carriage return or newline inside double quotes).

You can legally type the special sequences \r (carriage return) and \n (newline, line feed) by themselves, and in double-quoted strings.
// legal statements
$foo = \n ;
$foo = a b \r c ;
$foo = a b "cd\nef" g ;

In these cases, the \n and \r get translated by the Kleene tokenizer into their code point values, and these values duly appear on arcs in the resulting networks. The actual values can be seen by drawing the network or just invoking the sigma command, e.g.

$foo = \n ;
sigma $foo ;

You cannot type \n and \r inside single quotes to represent newline and carriage return. Currently they can be typed, e.g.

$foo = 'a\nb' ;

but the backslash gets interpreted as a literal backslash.
KRB: This behavior will be reviewed.

C.3. Control Characters and Writing Networks to XML 1.0

Although Kleene per se can handle the special characters \b (the BACKSPACE) and \f (the FORM FEED), any attempt to write the resulting network to XML will cause problems because XML 1.0 simply does not allow such control characters to appear in XML text. For example

$foo = \b ;
writeXml $foo, "fooFile.xml" ;
will result in an XML-like file that cannot be read back in because it is technically invalid, containing the BACKSPACE symbol that XML 1.0 simply does not allow. XML 1.0 does allow the \n (\u000A), \r (\u000D) and \t (\u0009) characters to appear in text, and so they do not cause the same problem.¹

¹The XML 1.1 standard, first published in 2004, also accepts \b (the BACKSPACE) and \f (the FORM FEED) in XML text, but XML 1.1 has never been well accepted or supported.
Appendix D

GUI FSM Testing and OTHER

In the Kleene GUI, using the test window, you can type in individual strings for generation or analysis, and see the output string or strings displayed in the terminal window. When testing FSMs containing OTHER (i.e. unknown) labels, the output may at first be unintuitive.

The first step is to understand that the test function implements the formal algorithm for applying an FSM to an input string. For generation using $testFst$

1. The input string is built into a one-path FSM; think of it as $input$.

2. The input FSM is composed “on top of” the $testFst$, i.e. ($input \circ testFst$)

3. Then the output side (also known as the lower side) of the composition is extracted, i.e. $^\text{lowerside}(input \circ testFst)$. The result is an acceptor.

4. Then, if the result acceptor is not empty, and the number of paths is finite, the strings are listed in the terminal window.

Similarly for analysis, the result acceptor is computed as $^\text{upperside}(testFst \circ input)$.

The second important point is that if the FSM being tested contains OTHER_NONID, then any output strings containing unknown characters
will display as OTHER_ID rather than OTHER_NONID. This is because when a projection (an acceptor) is extracted, the OTHER_NONID label cannot appear in an acceptor and is automatically converted to OTHER_ID.

\$testFst = .:. ;  
draw $testFst ;  
$generationresult = $^\text{lowerside}(z \_o\_ $testFst) ;  
draw $result ;

test $testFst ; // and enter z on top for generation

In this example, the $testFst FSM will contain both OTHER_ID and OTHER_NONID labels, but the $generationresult FSM, and the printed output of testing $testFst using the test facility, and generating z, will contain OTHER_ID and not OTHER_NONID. Again, the reason is that the application algorithm extracts a projection of the composition, and a projection is an acceptor and cannot contain OTHER_NONID.

The final important point in interpreting the output is to understand that the meaning/coverage of OTHER_ID in the result should be interpreted relative to the $generationresult or $analysisresult FSM, which is automatically created and displayed when using test.

Other kinds of testing, using external runtime code, cannot reasonably implement the formal composition-plus-projection-extraction method of application, and the results will be different. In the future, the Kleene test facility may offer a choice between the current formal method and alternative methods used in runtime code.
Appendix E

Pre-defined FSM-valued Functions

E.1. Commonly Used Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$^\text{copy}($fst)</td>
<td>Copy FSM</td>
</tr>
<tr>
<td>$^\text{complement}($fst)</td>
<td>Complement FSM</td>
</tr>
<tr>
<td>$^\text{reverse}($fst)</td>
<td>Reverse FSM</td>
</tr>
<tr>
<td>$^\text{invert}($fst)</td>
<td>Invert FSM</td>
</tr>
<tr>
<td>$^\text{invert!}($fst)</td>
<td>Invert FSM (another form)</td>
</tr>
<tr>
<td>$^\text{inputside}($fst) or $^\text{inputProject}($fst) or $^\text{upperside}($fst)</td>
<td>Input side of FSM</td>
</tr>
<tr>
<td>$^\text{inputside!}($fst) or $^\text{inputProject!}($fst) or $^\text{upperside!}($fst)</td>
<td>Input side of FSM (another form)</td>
</tr>
<tr>
<td>$^\text{outputside}($fst) or $^\text{outputProject}($fst) or $^\text{lowerside}($fst)</td>
<td>Output side of FSM</td>
</tr>
<tr>
<td>$^\text{outputside!}($fst) or $^\text{outputProject!}($fst) or $^\text{lowerside!}($fst)</td>
<td>Output side of FSM (another form)</td>
</tr>
<tr>
<td>$^\text{shortestPath}($fst, #num=1)</td>
<td>Shortest path in FSM</td>
</tr>
<tr>
<td>$^\text{toString}(#num)</td>
<td>Convert to string</td>
</tr>
<tr>
<td>$^\text{implode}($fst)</td>
<td>Implode FSM</td>
</tr>
<tr>
<td>$^\text{explode}($fst)</td>
<td>Explode FSM</td>
</tr>
<tr>
<td>$^\text{readXml}($filepath)</td>
<td>Read XML</td>
</tr>
<tr>
<td>$^\text{ignore}($base, $fluff)</td>
<td>Ignore elements</td>
</tr>
<tr>
<td>$^\text{contains}($fst)</td>
<td>Contains</td>
</tr>
<tr>
<td>$^\text{substSymbol}($fst, $old, $new)</td>
<td>Substitute symbol in FSM</td>
</tr>
<tr>
<td>$^\text{substSymbol!}($fst, $old, $new)</td>
<td>Substitute symbol in FSM (another form)</td>
</tr>
</tbody>
</table>
### E.2. Less Used Functions

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$^randGen($fst, #npath=15, #max_length=50)</code></td>
</tr>
<tr>
<td><code>$^rmWeight($fst)</code></td>
</tr>
<tr>
<td><code>$^rmWeight!($fst)</code></td>
</tr>
<tr>
<td><code>$^priority_union_input($net1, $net2)</code></td>
</tr>
<tr>
<td><code>$^priority_union_output($net1, $net2)</code></td>
</tr>
<tr>
<td><code>$^lenient_composition_input($base, $filter)</code></td>
</tr>
<tr>
<td><code>$^lenient_composition_output($filter, $base)</code></td>
</tr>
</tbody>
</table>

### E.3. Functions for Use by Experts

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$^charForCpv(#num)</code> or <code>$^cpv2char(#num)</code></td>
</tr>
<tr>
<td><code>$^optimize($fst)</code></td>
</tr>
<tr>
<td><code>$^optimize!($fst)</code></td>
</tr>
<tr>
<td><code>$^determinize($fst)</code></td>
</tr>
<tr>
<td><code>$^determinize!($fst)</code></td>
</tr>
<tr>
<td><code>$^minimize($fst)</code></td>
</tr>
<tr>
<td><code>$^minimize!($fst)</code></td>
</tr>
<tr>
<td><code>$^rmEpsilon($fst)</code></td>
</tr>
<tr>
<td><code>$^rmEpsilon!($fst)</code></td>
</tr>
<tr>
<td><code>$^synchronize($fst)</code></td>
</tr>
<tr>
<td><code>$^synchronize!($fst)</code></td>
</tr>
<tr>
<td><code>$^closeSigma($fst, $base=&quot;&quot;)</code></td>
</tr>
<tr>
<td><code>$^closeSigma!($fst, $base=&quot;&quot;)</code></td>
</tr>
<tr>
<td><code>$^flatten($fst)</code></td>
</tr>
<tr>
<td><code>$^flatten!($fst)</code></td>
</tr>
</tbody>
</table>
Appendix F

Pre-defined List Functions

F.1. Pre-defined Functions for FSM Lists

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$@^\text{copy}(@$list)</td>
<td>Copy list</td>
</tr>
<tr>
<td>$^\text{head}(@$list)</td>
<td>Get first element</td>
</tr>
<tr>
<td>$^\text{getLast}(@$list)</td>
<td>Get last element</td>
</tr>
<tr>
<td>$^\text{get}(@$list, #n)</td>
<td>Get element at index #n</td>
</tr>
<tr>
<td>$@^\text{getSlice}(@$list, #n, #r:#s, ...)</td>
<td>Get slice at indices #r to #s, ...</td>
</tr>
<tr>
<td>$@^\text{tail}(@$list)</td>
<td>Get tail list</td>
</tr>
<tr>
<td>$^\text{pop!}(@$list)</td>
<td>Remove and return first element</td>
</tr>
<tr>
<td>$^\text{removeLast!}(@$list)</td>
<td>Remove last element and return list</td>
</tr>
<tr>
<td>$^\text{remove!}(@$list, #n)</td>
<td>Remove element at index #n</td>
</tr>
<tr>
<td>$@^\text{push!}(fst, @$list)</td>
<td>Add first to list</td>
</tr>
<tr>
<td>$^\text{add!}(@$list, $fst)</td>
<td>Add element to list</td>
</tr>
<tr>
<td>$@^\text{addAt!}(@$list, #n, $fst)</td>
<td>Add element at index #n to list</td>
</tr>
<tr>
<td>$@^\text{set!}(@$list, #n, $fst)</td>
<td>Replace element at index #n with $fst</td>
</tr>
<tr>
<td>$^\text{size}(@$list)</td>
<td>Get size</td>
</tr>
<tr>
<td>$@^\text{getSigma}(fst)</td>
<td>Get sigma</td>
</tr>
<tr>
<td>$^\text{reduceLeft}(^\text{bin}, @$list)</td>
<td>Reduce left with function ^\text{bin}</td>
</tr>
<tr>
<td>$^\text{foldLeft}(^\text{bin}, @$list, $init)</td>
<td>Fold left with function ^\text{bin} and initial value $init</td>
</tr>
<tr>
<td>$@^\text{map}(^\text{mon}, @$list)</td>
<td>Map list with function ^\text{mon}</td>
</tr>
</tbody>
</table>

275
F.2. Pre-defined Functions for Number Lists

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>^copy</code>(@list)</td>
</tr>
<tr>
<td><code>^head</code>(@list)</td>
</tr>
<tr>
<td><code>^getLast</code>(@list)</td>
</tr>
<tr>
<td><code>^get</code>(@list, n)</td>
</tr>
<tr>
<td><code>^getSlice</code>(@list, n, r:s, ...)</td>
</tr>
<tr>
<td><code>^tail</code>(@list)</td>
</tr>
<tr>
<td><code>^pop!</code>(@list)</td>
</tr>
<tr>
<td><code>^removeLast!</code>(@list)</td>
</tr>
<tr>
<td><code>^remove!</code>(@list, n)</td>
</tr>
<tr>
<td><code>^push!</code>(num, @list)</td>
</tr>
<tr>
<td><code>^add!</code>(@list, num)</td>
</tr>
<tr>
<td><code>^addAt!</code>(@list, n, num)</td>
</tr>
<tr>
<td><code>^set!</code>(@list, n, num)</td>
</tr>
<tr>
<td><code>^size</code>(@list)</td>
</tr>
<tr>
<td><code>^getSigma</code>($fst)</td>
</tr>
<tr>
<td><code>^reduceLeft</code>^bin, @list)</td>
</tr>
<tr>
<td><code>^foldLeft</code>^bin, @list, init)</td>
</tr>
<tr>
<td><code>^map</code>^mon, @list)</td>
</tr>
</tbody>
</table>
Appendix G

Pre-defined Case Functions

The following functions change or augment networks to handle case variants. The $proj argument must have the value input (or upper), output (or lower), or both, which is the default, e.g.

```
$foo = (abc):(def) ;
$bar = $\^uc($foo, output) ; // or $\^uc($foo, "output")
```

sets $bar to a version of $foo that has “DEF” rather than “def” on the output/lower side.
The \textit{ci} in the final four names stands for “case insensitive” (i.e. accept uppercase or lowercase), and the following functions with abbreviated names are also provided for convenience.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{$^uc($fst, $proj=&quot;both&quot;)}</td>
<td>convert to uppercase</td>
</tr>
<tr>
<td>\texttt{$^uc!($fst, $proj=&quot;both&quot;)}</td>
<td>convert to uppercase</td>
</tr>
<tr>
<td>\texttt{$^lc($fst, $proj=&quot;both&quot;)}</td>
<td>convert to lowercase</td>
</tr>
<tr>
<td>\texttt{$^lc!($fst, $proj=&quot;both&quot;)}</td>
<td>convert to lowercase</td>
</tr>
<tr>
<td>\texttt{$^init_uc($fst, $proj=&quot;both&quot;)}</td>
<td>convert initial char to uppercase</td>
</tr>
<tr>
<td>\texttt{$^init_uc!($fst, $proj=&quot;both&quot;)}</td>
<td>convert initial char to uppercase</td>
</tr>
<tr>
<td>\texttt{$^init_lc($fst, $proj=&quot;both&quot;)}</td>
<td>convert initial char to lowercase</td>
</tr>
<tr>
<td>\texttt{$^init_lc!($fst, $proj=&quot;both&quot;)}</td>
<td>convert initial char to lowercase</td>
</tr>
<tr>
<td>\texttt{$^opt_uc($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add uppercase</td>
</tr>
<tr>
<td>\texttt{$^opt_uc!($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add uppercase</td>
</tr>
<tr>
<td>\texttt{$^opt_lc($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add lowercase</td>
</tr>
<tr>
<td>\texttt{$^opt_lc!($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add lowercase</td>
</tr>
<tr>
<td>\texttt{$^opt_init_uc($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add initial uppercase</td>
</tr>
<tr>
<td>\texttt{$^opt_init_uc!($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add initial uppercase</td>
</tr>
<tr>
<td>\texttt{$^opt_init_lc($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add initial lowercase</td>
</tr>
<tr>
<td>\texttt{$^opt_init_lc!($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add initial lowercase</td>
</tr>
<tr>
<td>\texttt{$^opt_ci($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add uppercase &amp; lowercase</td>
</tr>
<tr>
<td>\texttt{$^opt_ci!($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add uppercase &amp; lowercase</td>
</tr>
<tr>
<td>\texttt{$^opt_init_ci($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add initial uppercase &amp; lowercase</td>
</tr>
<tr>
<td>\texttt{$^opt_init_ci!($fst, $proj=&quot;both&quot;)}</td>
<td>allow/add initial uppercase &amp; lowercase</td>
</tr>
</tbody>
</table>

It is anticipated that the behavior of the “ci” will be re-examined and perhaps changed, and that new functions will be added in future releases to handle “title”-casing and “camel”-casing.
Appendix H

Pre-defined Diacritic Functions

The following functions are provided to make a network diacritic-insensitive, which is significantly different from the case-insensitivity previously described. In particular, diacritic insensitivity is unidirectional, modifying a network that already accepts accented strings, e.g. école, to also accept partially or completely unaccented versions of the same string, here ecole. If the original network accepts only élève, then a diacritic-insensitive version of the network would also accept eleve, éleve and elève. Note that if a network accepts only the string resume, applying diacritic insensitivity will not cause it to accept strings like résumé, résume, resumé, rèsümè or any other strings with added or changed diacritics.¹

\[
\text{allow/add unaccented paths}
\]
\[
\text{allow/add unaccented paths}
\]

For convenience, the following shorter aliases are pre-defined.

¹Modifying a network to accept added or changed diacritics would, even if limited to pre-composed Unicode characters, cause a network to explode in size. If the open-ended system of Combining Diacritical Marks were also handled, the size of the network would be infinite.
Diacritic insensitivity works for all accented—also known as pre-composed—Unicode letter-characters, and it also includes handling of Unicode Combining Diacritical Marks.² Thus if a network originally accepts just école, the diacritic-insensitive version will accept école and ecole whether the original é was encoded as the single Unicode character U+00E9 (LATIN SMALL LETTER E WITH ACUTE) or as the sequence of characters U+0065 (LATIN SMALL LETTER E) followed by U+0301 (COMBINING ACUTE ACCENT).³

²If a network contains an arc labeled é, diacritic insensitivity adds a parallel arc labeled e. The unaccented version of the accented or “pre-composed” character is found via Unicode normalization to the canonical decomposed NFD form. For an arc labeled with a Combining Diacritical Mark, such as the COMBINING ACUTE ACCENT, diacritic insensitivity adds a parallel epsilon arc.

³OpenFst networks created by Kleene always store one Unicode character (using its standard code point value) in one arc label, and this includes supplementary Unicode characters and Combining Diacritical Marks.
Appendix I

Pre-defined Arithmetic-valued Functions

I.1. General

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathCount($fst)</td>
</tr>
<tr>
<td>stateCount($fst)</td>
</tr>
<tr>
<td>arcCount($fst)</td>
</tr>
<tr>
<td>arity($fst)</td>
</tr>
<tr>
<td>abs(#num)</td>
</tr>
<tr>
<td>ceil(#num)</td>
</tr>
<tr>
<td>floor(#num)</td>
</tr>
<tr>
<td>round(#num)</td>
</tr>
<tr>
<td>long(#num) or int(#num)</td>
</tr>
<tr>
<td>double(#num) or float(#num)</td>
</tr>
<tr>
<td>rint(#num)</td>
</tr>
<tr>
<td>getIntCpv($fst) or char2cpv($fst)</td>
</tr>
<tr>
<td>log(#num)</td>
</tr>
<tr>
<td>prob2c(#prob)</td>
</tr>
<tr>
<td>pct2c(#pct)</td>
</tr>
</tbody>
</table>
I.2. Boolean

```plaintext
#^isAcceptor($fst)
#^isTransducer($fst)
#^isCyclic($fst)
#^isEpsilonFree($fst)
#^isIBounded($fst) or #^isUBounded($fst)
#^isOBounded($fst) or #^isLBounded($fst)
#^isIDeterministic($fst) or #^isUDeterministic($fst)
#^isODeterministic($fst) or #^isLDeterministic($fst)

#^isEmptyLanguage($fst)
#^isEmptyStringLanguage($fst)
#^containsEmptyString($fst)
#^isString($fst) or #^isSingleStringLanguage($fst)
#^isUniversalLanguage($fst) (not reliable yet)

#^containsOther($fst)
#^isWeighted($fst)
#^isRtn($fst) or #^isRTN($fst)

#^equivalent($one, @two, #delta=#kDeltaFromOpenFst)
#^randEquivalent($one, $two, #delta=#kDeltaFromOpenFst, #seed=60, #path_length=60)
```


