Kleene, a Free and Open-Source Language for Finite-State Programming

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

Abstract

Kleene is a high-level programming language, based on the OpenFst library, for constructing and manipulating finite-state acceptors and transducers. Users can program using regular expressions, alternation-rule syntax and right-linear phrase-structure grammars; and Kleene provides variables, lists, functions and familiar program-control syntax. Kleene has been approved by SAP AG for release as free, open-source code under the Apache License, Version 2.0, and will be available by August 2012 for downloading from http://www.kleene-lang.org. The design, implementation, development status and future plans for the language are discussed.

1 Introduction

Kleene[1] is a finite-state programming language in the tradition of the AT&T Lextools (Roark and Sproat, 2007),[2] the SFST-PL language (Schmid, 2005),[3] the Xerox/PARC finite-state toolkit (Beesley and Karttunen, 2003)[4] and FOMA (Huldén, 2009b),[5] all of which provide higher-level programming formalisms built on top of low-level finite-state libraries. Kleene itself is built on the OpenFst library (Allauzen et al., 2007),[6] developed by Google Labs and NYU's Courant Institute.

The design and implementation of the language were motivated by three main principles, summarized as Syntax Matters, Licensing Matters and Open Source Matters. As for the syntax, Kleene allows programmers to specify weighted or unweighted finite-state machines (FSMs)—including acceptors that encode regular languages and two-projection transducers that encode regular relations—using regular expressions, alternation-rule syntax and right-linear phrase-structure grammars. The regular-expression operators are borrowed, as far as possible, from familiar Perl-like and academic regular expressions, and the alternation rules are based on the "rewrite rules" made popular by Chomsky and Halle (Chomsky and Halle, 1968). Borrowing from general-purpose programming languages, Kleene also provides variables, lists and functions, plus nested code blocks and familiar control structures such as `if-else` statements and `while` loops.

As for the licensing, Kleene, like the OpenFst library, is released under the Apache License, Version 2.0, and its other dependencies are also released under this and similar permissive licenses that allow commercial usage. In contrast, many notable finite-state implementations, released under the GPL and similar licenses, are restricted to academic and other non-commercial use. The Kleene code is also open-source, allowing users to examine, correct, augment and even adopt the code if the project should ever be abandoned by its original maintainer(s).

[1] 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.

[2] http://www.research.att.com/~alb/lextools/

[3] http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html

[4] http://www.fsmbook.com

[5] http://code.google.com/p/foma/

[6] http://www.openfst.org

Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing, pages 50–54,
Donostia–San Sebastián, July 23–25, 2012.

It is hoped that Kleene will provide an attractive development environment for experts and students. Pre-edited Kleene scripts can be run from the command line, but a graphical user interface is also provided for interactive learning, programming, testing and drawing of FSMs.

Like comparable implementations of finite-state machines, Kleene can be used to implement a variety of useful applications, including spell-checking and -correction, phonetic modeling, morphological analysis and generation, and various kinds of pattern matching. The paper continues with a brief description the Kleene language, the current state of development, and plans for the future.

2 Implementation

The Java-language Kleene parser, implemented with JavaCC and JJTree (Copeland, 2007),[7] is Unicode-capable and portable. Successfully parsed statements are reduced to abstract syntax trees (ASTs), which are interpreted by calling C++ functions in the OpenFst library via the Java Native Interface (JNI).

3 Kleene Syntax

3.1 Regular Expressions

Basic assignment statements have a regular expression on the right-hand side, as shown in Table 1. As in Perl regular expressions, simple alphabetic characters are literal, and concatenation is indicated by juxtaposition, with no overt operator. Parentheses can be used to group expressions. The postfixed $\star$ (the "Kleene star"), + (the "Kleene plus"), and `?` denote zero-or-more, one-or-more, and optionality, respectively. Square-bracketed expressions have their own internal syntax to denote character sets, including character ranges such as `[A-Z]`. The union operator is `|`. Basic regular operations missing from Perl regular expressions include composition (`o` or `_o_`), crossproduct (`:`), language intersection (`&`), language negation ($\sim$) and language subtraction ($-$). Weights are indicated inside angle brackets, e.g. `<0.1>`.

Special characters can be literalized with a preceding backslash or inside double quotes, e.g. `\*` or `"*"` denotes a literal asterisk rather than the Kleene plus. To improve the readability of expressions, spaces are not significant, unless they appear inside square brackets or are explicitly literalized inside double quotes or with a preceding backslash.

In a language like Kleene where alphabetic symbols are literal, and the expression `dog` denotes three literal symbols, *d*, *o* and *g*, concatenated together, there must be a way to distinguish variable names from simple concatenations. The Kleene solution is to prefix variable names that are bound to FSM values with a dollar-sign sigil, e.g. `$myvar`. Once defined, a variable name can be used inside subsequent regular expressions, as in the following example, which models a fragment of Esperanto verb morphology.

```
$vroot = don | dir | pens | ir ;
// "give", "say", "think", "go"
$aspect = ad ;
// optional repeated aspect
$vend = as | is | os | us | u | i ;
// pres, past, fut, cond, subj, inf
$verbs = $vroot $aspect? $vend ;
// use of pre-defined variables
```

Similarly, names of functions that return FSMs are distinguished with the `$^` sigil. To denote less common operations, rather than inventing and proliferating new and arguably cryptic regular-expression operators, Kleene provides a set of predefined functions including

`$^reverse`(*regexp*)
`$^invert`(*regexp*)
`$^inputProj`(*regexp*)
`$^outputProj`(*regexp*)
`$^contains`(*regexp*)
`$^ignore`(*regexp*, *regexp*)
`$^copy`(*regexp*)

Users can also define their own functions, and function calls are regular expressions that can appear as operands inside larger regular expressions.

3.2 Alternation-Rule Syntax

Kleene provides a variety of alternation-rule types, comparable to Xerox/PARC Replace Rules (Beesley and Karttunen, 2003, pp. 130–82), but implemented using algorithms by Måns Huldén (Huldén, 2009a).

[7] `https://javacc.dev.java.net`

```
$var = dog ;
$var = d o g ;          // equivalent to dog
$var = ~( a+ b* c? ) ;
$var = \~ \+ \* \? ;  // literalized special characters
$var = "~+*?";          // literalized characters inside double quotes
$var = "dog" ;          // unnecessary literalization, equivalent to dog
$myvar = (dog | cat | horse) s? ;
$yourvar = [A-Za-z] [A-Za-z0-9]* ;
$hisvar = ([A-Za-z]-[aeiouAEIOU])+ ;
$hervar = (bird|cow|elephant|pig) & (pig|ant|bird) ;
$ourvar = (dog):(chien) ∘ (chien):(Hund) ;
$theirvar = [a-z]+ ( a <0.91629> | b <0.1> ) ;  // weights in brackets
```

Table 1: Kleene Regular-Expression Assignment Examples.

input-expression −> *output-expression* / *left-context* _ *right-context*

Table 2: The Simplest Kleene Alternation-Rule Template.

The simplest rules have the template shown in Table 2, and are interpreted into transducers that map the input to the output in the specified context. Such rules, which cannot be reviewed in detail here, are commonly used to model phonetic and orthographical alternations.

3.3 Right-Linear Phrase Structure Grammars

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 describe 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, pp. 203–78).

A Kleene phrase-structure grammar is defined as a set of productions, each assigned to a variable with a `$>` sigil. Productions may include right-linear references to themselves or to other productions, which might not yet be defined. The productions are parsed immediately but are not evaluated until the entire grammar is built into an FSM via a call to the built-in function `$^start()`, which takes one production variable as its argument and treats it as the starting production of the whole grammar. The following example models a fragment of Esperanto noun morphotactics, including noun-root compounding.

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

$net = $^start($>Root) ;
```

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

4 Kleene FSMs

Each Kleene finite-state machine consists of a standard OpenFst FSM, under the default Tropical Semiring, wrapped with a Java object[8] that stores the private alphabet[9] of each machine.

In Kleene, it is not necessary or possible to declare the characters being used; characters appearing in regular expressions, alternation rules and right-linear phrase-structure grammars are stored automatically as FSM arc labels using their Unicode

[8] Each Java object of the class Fst contains a long integer field that stores a pointer to the OpenFst machine, which actually resides in OpenFst's C++ memory space.

[9] The alphabet, sometimes known as the *sigma*, contains just the symbols that appear explicitly in the labels of the FSM.

code point value, and this includes Unicode supplementary characters. Programmer-defined multi-character symbols, represented in the syntax with surrounding single quotes, e.g. `'+Noun'` and `'+Verb'`, or, using another common convention, `'[Noun]'` and `'[Verb]'`, also need no declaration and are automatically stored using code point values taken from a Unicode Private Use Area.

The dot (`.`) denotes *any* character, and it translates non-trivially into reserved arc labels that represent OTHER (i.e. unknown) characters.[10]

5 Status

5.1 Currently Working

As of the date of writing, Kleene is an advanced beta project offering the following:

- Compilation of regular expressions, right-linear phrase-structure grammars, and several alternation-rule variations into FSMs.
- Robust handling of Unicode, including supplementary characters, plus support for user-defined multi-character symbols.
- Variables and maintenance of symbol tables in a frame-based environment.
- Pre-defined and user-defined functions.
- Handling of lists of FSMs, iteration over lists, and functions that handle and return lists.
- A graphical user interface, including tools to draw FSMs and test them manually.
- File I/O of FSMs in an XML format.
- Interpretation of arithmetic expressions, arithmetic variables and functions, including boolean functions; and `if-then` statements and `while` loops that use boolean operators and functions.

[10]The treatment of FSM-specific alphabets and the handling of OTHER characters is modeled on the Xerox/PARC implementation (Beesley and Karttunen, 2003, pp. 56–60).

5.2 Future Work

The work remaining to be done includes:

- Completion of the implementation of alternation-rule variations.
- Writing of runtime code and APIs to apply FSMs to input and return output.
- Conversion of FSMs into stand-alone executable code, initially in Java and C++.
- Expansion to handle semirings other than the default Tropical Semiring of OpenFst.
- Testing in non-trivial applications to determine memory usage and performance.

6 History and Licensing

Kleene was begun informally in late 2006, became part of a company project in 2008, and was under development until early 2011, when the project was canceled. On 4 May 2012, SAP AG released Kleene as free, open-source code under the Apache License, Version 2.0.[11]

The Kleene source code will be repackaged according to Apache standards and made available for download by August of 2012 at `http://www.kleene-lang.org`. A user manual, currently over 100 pages, and an engineering manual will also be released. Precompiled versions will be provided for Linux, OS X and, if possible, Windows.

Acknowledgments

Sincere thanks are due to the OpenFst team and all who made that library available. A special personal thanks goes to Måns Huldén, who graciously released his algorithms for interpreting alternation rules and language-restriction expressions, and who went to great lengths to help me understand and re-implement them. I also acknowledge my SAP Labs colleagues Paola Nieddu and Phil Sours, who contributed to the design and implementation of Kleene, and my supervisor Michael Wiesner, who supported the open-source release. Finally, I thank Lauri Karttunen, who introduced me to finite-state linguistics and has always been a good friend and mentor.

[11]`http://www.apache.org/licenses/LICENSE-2.0.html`

References

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings of the Ninth International Conference on Implementation and Application of Automata (CIAA 2007)*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer.

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite State Morphology*. CSLI Publications, Palo Alto, CA.

Noam Chomsky and Morris Halle. 1968. *The Sound Pattern of English*. Harper and Row, New York.

Tom Copeland. 2007. *Generating Parsers with JavaCC*. Centennial Books, Alexandria, VA.

Måns Huldén. 2009a. *Finite-State Machine Construction Methods and Algorithms for Phonology and Morphology*. Ph.D. thesis, The University of Arizona, Tucson, AZ.

Måns Huldén. 2009b. Foma: a finite-state compiler and library. In *Proceedings of the EACL 2009 Demonstrations Session*, pages 29–32, Athens, Greece.

Brian Roark and Richard Sproat. 2007. *Computational Approaches to Morphology and Syntax*. Oxford Surveys in Syntax & Morphology. Oxford University Press, Oxford.

Helmut Schmid. 2005. A programming language for finite state transducers. In *FSMNLP'05*, Helskinki.