Regular expressions: Basic notions

Fritz Henglein

Department of Computer Science
University of Copenhagen
Email: henglein@diku.dk

2nd International Summer School on Advances in Programming Languages
Edinburgh, Scotland

2014-08-22

Kleene Meets Church (KMC)

Research project on type theory for

- proof-theoretic foundations of formal language theory, specifically regular languages;
- semantically well-behaved, powerful, scalable, efficient (large) stream processing tools

Web site: http://www.diku.dk/kmc

Regular expression

Definition (Regular expression)

A *regular expression (RE)* over *alphabet* (set) A is an expression of the form

$$E, F ::= 0 \mid 1 \mid a \mid E|F \mid EF \mid E*$$

where $a \in A$.

- This should be read as an *abstract syntax* specification.
- Operator precedence: $*$ $>$ juxtaposition $>|$. Use parentheses for associating subexpressions explicitly.
- A is often assumed to be finite.
- Kleene (1956) coined the term "regular expression". The above (though as a concrete grammar) is what he meant by it *and nothing else*.

What is (the meaning of) a regular expression?

What does a regular expression *denote*?

- Regular expression as a *language*: A set of strings (= classical automata theory)
- Regular expression as a *grammar* or *type*: A set of parse trees.
- Regular expression as a *Kleene algebra*: Result of interpreting in an algebraic structure satisfying certain equational properties.
- Regular expression as a *tool* (loose notion): Domain-specific language for string processing that is embedded in or accessible from general-purpose programming language

We focus on *regular expressions as types*, which is most relevant for extracting and transforming information from input streams.

Regular expression as language

Definition (Language denoted by regular expression)

A regular expression E *denotes* the language $\mathcal{L}[\![E]\!]$ (set of strings) defined by

$$\begin{array}{lcllcl} \mathcal{L}[\![0]\!] & = & \emptyset & \mathcal{L}[\![E|F]\!] & = & \mathcal{L}[\![E]\!] \cup \mathcal{L}[\![F]\!] \\ \mathcal{L}[\![1]\!] & = & \{\epsilon\} & \mathcal{L}[\![EF]\!] & = & \mathcal{L}[\![E]\!] \odot \mathcal{L}[\![F]\!] \\ \mathcal{L}[\![a]\!] & = & \{a\} & \mathcal{L}[\![E*]\!] & = & \bigcup_{i \geq 0} (\mathcal{L}[\![E]\!])^i \end{array}$$

where $S \odot T = \{s\,t \mid s \in S \wedge t \in T\}$, $E^0 = \{\epsilon\}$, $E^{i+1} = E \odot E^i$.

Regular language

Definition (Regular language)

A *regular language* is a language (set of strings) over some finite alphabet A that is accepted by some deterministic finite automaton.

- A regular language may be *infinite*, but it is recognized by a *finite* automaton, which may be *circular* (contain loops).
- Loose intuition: "Regular" = "There is something finitary and potentially circular about it"

Kleene's Theorem

Theorem (Kleene 1956)

A language is regular if and only it is denoted by a regular expression.

Many other characterizations of regular languages

A language is regular if and only if:

- it is accepted by a *nondeterministic finite state machine (NFA)*
- it is accepted by an *alternating finite automaton (AFA)*
- it is generated by a *regular grammar*
- it is generated by a *prefix grammar*
- it is accepted by a *read-only Turing machine*
- it is defined in *monadic second-order logic over strings*
- it is recognized by a *finitely generated monoid*
- it is the preimage of a subset of a *finite monoid under a homomorphism from the free monoid on its alphabet*.

Regular expression equivalence and containment

Sometimes we are interested in regular expression containment or equivalence.

Definition

- E is *contained* in F, written $\models E \leq F$, if $\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$.
- E is *equivalent* to F, written $\models E = F$, if $\mathcal{L}[\![E]\!] = \mathcal{L}[\![F]\!]$.

Regular expression equivalence and containment are easily related:

$$\models E \leq F \iff \models E|F = F$$

and

$$\models E = F \iff \models E \leq F \wedge \models F \leq E.$$

Regular expression as language: What we usually learn

- They're just a way to talk about finite state automata
- All equivalent regular expressions are interchangeable since they accept the same language.
- All equivalent automata are interchangeable since they accept the same language.
  - We might as well choose an efficient one (deterministic, minimal state): it processes its input in linear time and constant space.
- Myhill-Nerode Theorem (for proving a language regular)
- Pumping Lemma (for proving a language nonregular)
- Equivalence is decidable: PSPACE-complete.
- They are closed under complement (for finite A) and intersection (for arbitrary A).
- Star-height problem...
- Good for specifying lexical scanners.

Regular expression as grammar (type): What we actually use[1]

- Full matching, `s =~ /^E$/`:
  Does E match s; that is, $s \in \mathcal{L}[\![E]\!]$?
- Partial matching, `s =~ /E/`:
  Does E match a substring of s; that is, $s \in \mathcal{L}[\![A* \; E \; A*]\!]$?
- Partial matching with recording, `s =~ /(E)/`:
  Does E match a substring of s, and, if so, bind a matching substring to a variable.
- Partial matching with nested recording,
  `s =~ /(...(E)...(F)...)/`:
  Does the RE match a substring in s, and, if so, bind some matching substring in s to each of the parenthesized *groups* (regular subexpressions).

[1] in Perl and such

Regular expression as grammar (type): What we actually use[3]

- Substitution, `s =~ s/E/F/m` where F may reference group variables and m are modifiers affecting the matching semantics.
- Extensions: Backreferences, look-ahead, look-behind,...
- Disambiguation: Manually annotating RE to guide which particular substrings in the input are to be matched.
  - lazy and greedy matching
  - possessive quantifiers
  - atomic grouping
- Optimization: Manually transforming RE such that it is efficiently processed by pattern matching engine[2]

[2] Friedl, Mastering Regular Expressions, chapter 6: *Crafting an efficient expression*

[3] in Perl and such

Disambiguation and optimization?

- Why the need for disambiguation and optimization?
- There is no mention of that in theory of regular expression as regular *languages*:
  - Corresponds to full matching problem *without recording* (membership testing)
  - No ambiguity possible: we only return yes or no!
  - Any finite automaton can be used: same automaton can be constructed and used for all equivalent regular expressions—no point in "optimizing".
- Returning substring matches makes regular language and automata theory a priori *inapplicable*:
  - Substring matching is often ambiguous: Multiple candidates for substrings that match
  - Cannot use optimized automata: Grammatical information of original regular expression is lost, can't easily find substring matches.

Ambiguity

Example

`((a|ab)(bc|(c)))*.`

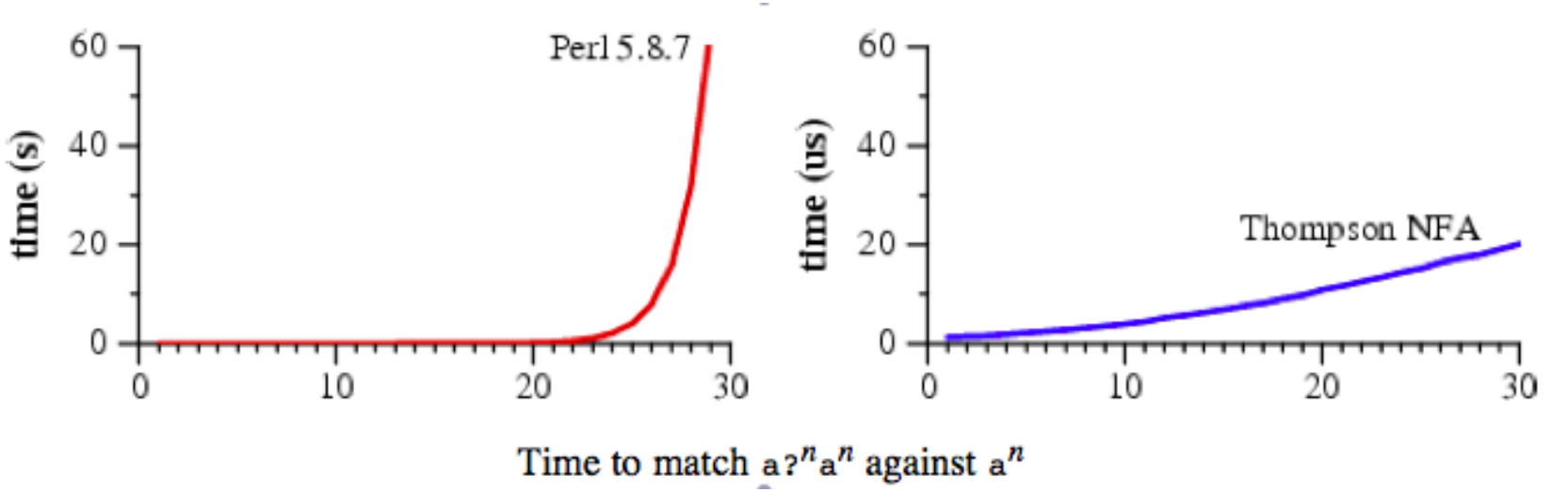
Match against `acabc`.
For each parenthesized *group* one substring occurrence is returned.

		PCRE	*POSIX*
$0	=	*acabc*	*acabc*
$1	=	*abc*	*abc*
$2	=	*a*	*ab*
$3	=	*bc*	*c*
$4	=	ϵ[a]	*c*

[a] Or special *no match*-value to distinguish from match with empty string.

Note: Only *last* match under Kleene-star $*$ is returned.

Optimization



Cox (2007)

- Perl-compliant regular expressions (what you get in Perl, Python, Ruby, Java) use *backtracking parsing*.
- Requires worst-case exponential time or may even crash (stack overflow) if E contains ϵ..

Why discrepancy between theory and practice?

- Theory is *extensional*: About regular *languages*.
  - Does this string match the regular expression? Yes or no?
- Practice is *intensional*: About regular expressions as *grammars*.
  - Does this string match the regular expression and if so *how*—which parts of the string match which parts of the RE?
- Ideally (conceptually): Regular expression processing = Guaranteed efficient parsing + *catamorphic* postprocessing[4]
  - KMC Project: Turn ideal into reality.
- Reality: Regular expression processing =
  - Ad-hoc backtracking matching with numerous extensions (Perl); or
  - Finite automaton + opportunistic instrumentation to get *some* parsing information (RE2).
- Regular expressions as grammars is semantically and computationally much more difficult than regular expressions as languages!
  - Ambiguity
  - Time and space complexity of parsing and processing

Type interpretation

Definition (Type interpretation)

The *type interpretation* $\mathcal{T}[\![.]\!]$ compositionally maps a regular expression E to the corresponding simple type:

$$
\begin{array}{rcll}
\mathcal{T}[\![0]\!] & = & \emptyset & \text{empty type} \\
\mathcal{T}[\![1]\!] & = & \{()\} & \text{unit type} \\
\mathcal{T}[\![a]\!] & = & \{a\} & \text{singleton type} \\
\mathcal{T}[\![E|F]\!] & = & \mathcal{T}[\![E]\!] + \mathcal{T}[\![F]\!] & \text{sum type} \\
\mathcal{L}[\![E\ F]\!] & = & \mathcal{T}[\![E]\!] \times \mathcal{T}[\![F]\!] & \text{product type} \\
\mathcal{T}[\![E^*]\!] & = & \{[v_1, \ldots, v_n] \mid v_i \in \mathcal{T}[\![E]\!]\} & \text{list type}
\end{array}
$$

$$
\begin{array}{rcl}
T + U & = & \{\mathrm{inl}\, v \mid v \in T\} \cup \{\mathrm{inr}\, w \mid w \in U\} \\
T \times U & = & \{(v, w) \mid v \in T \wedge w \in U\} \\
[v1, \ldots, v_n] & = & v1 :: \ldots :: v_n :: \mathrm{nil} \\
v :: vs & = & \mathrm{inl}\,(v, vs) \\
\mathrm{nil} & = & \mathrm{inr}\,()
\end{array}
$$

Flattening (Unparsing)

Definition

The *flattening* function $\text{flat}(.) : \text{Val}(\mathcal{A}) \to \text{Seq}(\mathcal{A})$ is defined as follows:

$$\begin{array}{rclrcl} \text{flat}(()) & = & \epsilon & \text{flat}(a) & = & a \\ \text{flat}(\text{inl}\, v) & = & \text{flat}(v) & \text{flat}(\text{inr}\, w) & = & \text{flat}(w) \\ \multicolumn{6}{c}{\text{flat}((v, w)) \;=\; \text{flat}(v)\ \text{flat}(w)} \end{array}$$

Note: $\text{flat}([v_1, \ldots, v_n]) = \text{flat}(v_1) \ldots \text{flat}(v_n)$

Example

$$\begin{array}{rcl} \text{flat}([\text{inl}\,((a, b), \text{inr}\, d), \text{inr}\,(a, (b, c))]) & = & abdabc \\ \text{flat}([\text{inl}\,((a, b), \text{inr}\, d), \text{inl}\,((a, b), \text{inl}\, c)]) & = & abdabc \end{array}$$

Parse tree = element of type

Example

Parse `acabc` according to `((a|ab)(bc|(c)))*`.

- $p_1 = [(\mathrm{inl}\, a, \mathrm{inr}\, c), (\mathrm{inl}\, a, \mathrm{inl}\, (b, c))]$
- $p_2 = [(\mathrm{inl}\, a, \mathrm{inr}\, c), (\mathrm{inr}\, (a, b), \mathrm{inr}\, c)]$

- p_1, p_2 have *type* $\boxed{(\{\mathtt{a}\} + \{\mathtt{a}\} \times \{\mathtt{b}\}) \times (\{\mathtt{b}\} \times \{\mathtt{c}\} + \{\mathtt{c}\})\,\mathrm{list}}$.
- Compare with *regular expression* $\boxed{\mathtt{((ab)(c|d)|(abc))*}}$.
- The *elements* of *type* E correspond to the *syntax trees* for strings parsed according to *regular expression* E!

Regular expression as type and as language

Theorem

$$\mathcal{L}[\![E]\!] = \{\mathrm{flat}(v) \mid v \in \mathcal{T}[\![E]\!]\}$$

The flattened *values* of *type* E = the *strings* in *regular language* E.

Membership testing versus parsing

Example

E = `((a|ab)(bc|(c)))*` E_d = `(a(bb|b|1)c)*`

- E_d is *unambiguous*: If $v, w \in \mathcal{T}[\![E_d]\!]$ and $\mathrm{flat}(v) = \mathrm{flat}(w)$ then $v = w$. (Each string in E_d has exactly one syntax tree.)
- E is *ambiguous*. (Recall p_1 and p_2.)
- E and E_d are *equivalent*: $\mathcal{L}[\![E]\!] = \mathcal{L}[\![E_d]\!]$
- Matching (membership testing): Easy—construct DFA from either E or E_d.
- But: How to *parse* according to E using E_d or (same) DFA for either?

Regular expression as algebra

Definition

A *Kleene algebra* is any algebraic structure with operations $0, 1, +, \cdot, *$ over carrier set V such that $t_1 = t_2$ for terms built from the operations and constants drawn from V whenever they denote the same regular language under the standard interpretation of the operations as in the definition of "language denoted by regular expression".

- A Kleene algebra may satisfy additional equalities.
- The extension of Kleene algebras with *test* operations is useful in program verification and other applications.

Example: Relational algebra

Example

Consider the following algebraic structure:

- $V = 2^{U \times U}$, the set of binary relations over some given *universe* of elements.
- $0 = \{\}$, the empty relation.
- $1 = \{(x, x) \mid x \in U\}$, the relation that relates all elements of U only to themselves.
- $R + S = R \cup S$, the union of R and S.
- $R \cdot S = \{(x, z) \mid \exists y \in U. (x, y) \in R \wedge (y, z) \in S\}$
- $R^* = \{(x_1, x_n) \mid \exists x_2, \ldots, x_{n-1}. \forall i \in \{2, \ldots, n\}. (x_{i-1}, x_i) \in R\}$.

This is a Kleene algebra.

Regular expression as tool

- Some of most used *domain specific languages (DSLs)* for programming:
  - SQL
  - *Regexes*
- Regexes used for substring matching and substitution in input strings
- **Warning:** "Regular expression" or "regex" in programming (e.g. Perl) are *proper extensions* of regular expressionsas defined by Kleene.
  - Backreferences make "regex"es drastically more powerful and difficult to process than regular expressions – they are not regular (!)

Exercises I

1. Implement a backtracking full regular expression matcher in Haskell, which takes a regular expression and a string as inputs and returns a boolean: true if the string matches (fully), false if it does not. Does it terminate for all inputs? If not, for which inputs does it not terminate? Can you change it to make it terminating for all inputs?
2. Draw a DFA for E_d (see slide "Membership testing versus parsing"). Is it minimal?
3. Argue how one can see that E_d is unambiguous.
4. Consider $(a \mid 1)\,(a \mid 1)\,a\,a$. What is the *type* of its parse trees (in Haskell or Standard ML)? Match it against *aaa*. Give the set of all parse trees of *aaa* (as elements of the above type). Is the regular expression ambiguous or unambiguous? What is its minimal DFA? Execute the DFA on *aaa*. Can you construct the parse trees for $(a \mid 1)\,(a \mid 1)\,a\,a$ from it?

Exercises II

1. Can you come up with sound rules that ensure that a regular expression is unambiguous? Are they complete?
2. Prove that $\models a\,a* = a*\,a$. Generalize your proof to $\models E\,E* = E*E$ for all E.
3. Alan claims that $\models E = E\,F \mid G$ implies $\models E = G\,F*$. Is this implication
   1. true for all choices of E, F, G?
   2. false for all choices of E, F, G?
   3. true for some choices of E, F, G and false for some choices of E, F, G?
4. Dan claims that $\models E\,F \leq F$ implies $\models E*\,F \leq F$. Always true, always false or sometimes true/sometimes false?
5. The regular languages and binary relations are Kleene algebras. Can you come up with another one?