Regular Expressions Using Haskell

Juan Pedro Villa Isaza

Logic and Computation Research Group
EAFIT University, Medellín, Colombia

October 8, 2007 / Logic and Computation Seminar

Outline

Outline

Outline

Abstract

Abstract

Regular expressions (known as regexps or regexes) are a way to describe text through pattern matching. You might want to use regular expressions to validate data, to pull pieces of text out of larger blocks, or to substitute new text for old text. Today, regular expressions are included in most programming languages as well as many scripting languages, editors, applications, databases, and command-line tools. In Regular Expressions Using Haskell, we show how to use regular expressions with the Haskell programming language, and we give an introduction to the standard library that implements regexps in Haskell, the Text.Regex.Posix library.

Regular Expressions and Pattern Matching.

Regular expressions, pattern matching and regular expression engines.

- A regular expression is a string containing a combination of normal characters and special metacharacters or metasequences. The normal characters match themselves. Metacharacters and metasequences are characters or sequences of characters that represent ideas such as quantity, locations, or types of characters.
- Pattern matching consists of finding a section of text that is described (matched) by a regular expression. The underlying code that searchs the text is the regular expression engine.

Regular Expressions and Pattern Matching.

Regular expressions, pattern matching and regular expression engines.

- A regular expression is a string containing a combination of normal characters and special metacharacters or metasequences. The normal characters match themselves. Metacharacters and metasequences are characters or sequences of characters that represent ideas such as quantity, locations, or types of characters.
- Pattern matching consists of finding a section of text that is described (matched) by a regular expression. The underlying code that searches the text is the regular expression engine.

Regular Expressions and Pattern Matching.

You can guess the results of most matches by keeping two rules in mind.

1. The earliest (leftmost) match wins. Regular expressions are applied to the input starting at the first character and proceeding toward the last. As soon as the regular expression engine finds a match, it returns.
2. Standard quantifiers are greedy. Quantifiers specify how many times something can be repeated. The standard quantifiers attempt to match as many times as possible. They settle for less than the maximum only if this is necessary for the success of the match. The process of giving up characters and trying less-greedy matches is called backtracking.

Regular Expressions and Pattern Matching.

You can guess the results of most matches by keeping two rules in mind.

1. The earliest (leftmost) match wins. Regular expressions are applied to the input starting at the first character and proceeding toward the last. As soon as the regular expression engine finds a match, it returns.
2. Standard quantifiers are greedy. Quantifiers specify how many times something can be repeated. The standard quantifiers attempt to match as many times as possible. They settle for less than the maximum only if this is necessary for the success of the match. The process of giving up characters and trying less-greedy matches is called backtracking.

Regular Expression Engines.

Regular expression engines have subtle differences based on their type. There are two classes of engines: Deterministic Finite Automaton (DFA) and Nondeterministic Finite Automaton (NFA). DFAs are faster but lack many of the features of an NFA, such as capturing, lookaround, and non-greedy quantifiers. In the NFA world, there two types: Traditional and POSIX.

Regular Expression Engines.
DFA engines.

DFAs compare each character of the input string to the regular expression, keeping track of all matches in progress. Since each character is examined at most once, the DFA engine is the fastest. One additional rule to remember with DFAs is that the alternation metasequence is greedy. When more than one option in an alternation (`foo|foobar`) matches, the longest one is selected. So, rule number 1 can be amended to read “the longest leftmost match wins”.

Regular Expression Engines.

Traditional NFA engines.

Traditional NFA engines compare each element of the regex to the input string, keeping track of positions where it chose between two options in the regex. If an option fails, the engine backtracks to the most recently stored position. For standard quantifiers, the engine chooses the greedy option of matching more text; however, if that options leads to the failure of the match, the engine returns to a saved position and tries a less greedy path. The traditional NFA engine uses ordered alternation, where each option in the alternation is tried sequentally. A longer match may be ignored if an earlier match leads to a succesful match. So, rule number 1 can be amended to read “the first leftmost match after greedy quantifiers have had their fill.”

Regular Expression Engines.
POSIX NFA engines.

POSIX NFA engines work similarly to traditional NFAs with one exception: a POSIX engine always picks the longest of the leftmost matches. For example, the alternation `cat | category` would match the full word “category” whenever possible, even if the first alternative (“cat”) matched and appeared earlier in the alternation.

Outline

Preliminaries.

Let Σ be a finite set of symbols and let L, L_1, and L_2 be sets of strings from Σ^*. The concatenation of L_1 and L_2, denoted L_1L_2, is the set $\{xy|x$ is in L_1 and y is in $L_2\}$. That is, the strings in L_1L_2 are formed by choosing a string L_1 and following it by a string L_2, in all possible combinations.

Define $L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$ for $i \geq 1$.

Preliminaries.

The Kleene closure (or just closure) of L, denoted L^*, is the set

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

and the positive closure of L, denoted L^+, is the set

$$L^+ = \bigcup_{i=1}^{\infty} L^i \,.$$

That is, L^* denotes words constructed by concatenating any number of words from L. L^+ is the same, but the case of zero words, whose “concatenation” is defined to be ϵ, is excluded. Not that L^+ contains ϵ if and only if L does.

Regular Expressions.

Definition (Regular expressions)

Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote are defined recursively as follows.

1. $\emptyset$ is a regular expression and denotes the empty set.
2. ϵ is a regular expression and denotes the set $\{\epsilon\}$.
3. For each a in Σ, a is a regular expression and denotes the set $\{a\}$.
4. If r and s are regular expressions denoting the languages R and S, respectively, then $(r + s)$, (rs), and (r^*) are regular expressions that denote the sets $R \cup S$, RS, and R^*, respectively.

Regular Expressions.

Definition (Regular expressions)

Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote are defined recursively as follows.

1. $\emptyset$ is a regular expression and denotes the empty set.
2. ϵ is a regular expression and denotes the set $\{\epsilon\}$.
3. For each a in Σ, a is a regular expression and denotes the set $\{a\}$.
4. If r and s are regular expressions denoting the languages R and S, respectively, then $(r + s)$, (rs), and (r^*) are regular expressions that denote the sets $R \cup S$, RS, and R^*, respectively.

Regular Expressions.

Definition (Regular expressions)

Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote are defined recursively as follows.

1. $\emptyset$ is a regular expression and denotes the empty set.
2. ϵ is a regular expression and denotes the set $\{\epsilon\}$.
3. For each a in Σ, a is a regular expression and denotes the set $\{a\}$.
4. If r and s are regular expressions denoting the languages R and S, respectively, then $(r+s)$, (rs), and (r^*) are regular expressions that denote the sets $R \cup S$, RS, and R^*, respectively.

Regular Expressions.

Definition (Regular expressions)

Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote are defined recursively as follows.

1. $\emptyset$ is a regular expression and denotes the empty set.
2. ϵ is a regular expression and denotes the set $\{\epsilon\}$.
3. For each a in Σ, a is a regular expression and denotes the set $\{a\}$.
4. If r and s are regular expressions denoting the languages R and S, respectively, then $(r + s)$, (rs), and (r^*) are regular expressions that denote the sets $R \cup S$, RS, and R^*, respectively.

Regular Expressions.

Definition (Regular expressions)

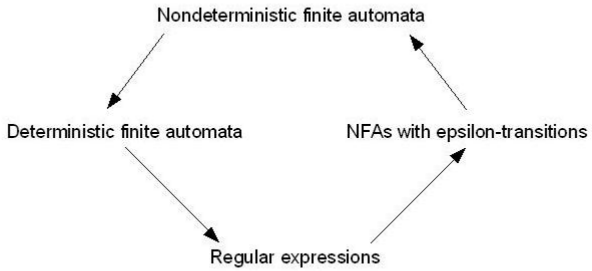
Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote are defined recursively as follows.

1. $\emptyset$ is a regular expression and denotes the empty set.
2. ϵ is a regular expression and denotes the set $\{\epsilon\}$.
3. For each a in Σ, a is a regular expression and denotes the set $\{a\}$.
4. If r and s are regular expressions denoting the languages R and S, respectively, then $(r + s)$, (rs), and (r^*) are regular expressions that denote the sets $R \cup S$, RS, and R^*, respectively.

Regular Expressions.

In writing regular expressions we can omit many parentheses if we assume that $*$ has higher precedence than $+$. When necessary to distinguish between a regular expression r and the language denoted by r, we use $L(r)$ for the latter. When no confusion is possible we use r for both the regular expression and the language denoted by the regular expression.

Equivalence of Finite Automata and Regular Expressions.


Nondeterministic finite automata
Deterministic finite automata
NFAs with epsilon-transitions
Regular expressions

Equivalence of Finite Automata and Regular Expressions.

Theorem

Let r be a regular expression. Then there exists an NFA with ϵ-transitions that accepts $L(r)$.

Theorem

If L is accepted by a DFA, then L is denoted by a regular expression.

Equivalence of Finite Automata and Regular Expressions.

Theorem

Let r be a regular expression. Then there exists an NFA with ϵ-transitions that accepts $L(r)$.

Theorem

If L is accepted by a DFA, then L is denoted by a regular expression.

Outline

Overview

- Chris Kuklewicz has developed a regular expression library for Haskell that has been implemented with a variety of backends. Some of these backends are native Haskell implementations, others are not and rely on external C libraries such as libpcre.
- Speed is something that should be benchmarked by the actual user, since the story changes so much with the task, new GHC, compiler flags, etc. The algorithm used may be a useful thing (backtracking vs NFA/DFA).
- All backends support String, (Seq Char), ByteString, and ByteString.Lazy.

Overview

- Chris Kuklewicz has developed a regular expression library for Haskell that has been implemented with a variety of backends. Some of these backends are native Haskell implementations, others are not and rely on external C libraries such as libpcre.
- Speed is something that should be benchmarked by the actual user, since the story changes so much with the task, new GHC, compiler flags, etc. The algorithm used may be a useful thing (backtracking vs NFA/DFA).
- All backends support String, (Seq Char), ByteString, and ByteString.Lazy.

Overview

- Chris Kuklewicz has developed a regular expression library for Haskell that has been implemented with a variety of backends. Some of these backends are native Haskell implementations, others are not and rely on external C libraries such as libpcre.
- Speed is something that should be benchmarked by the actual user, since the story changes so much with the task, new GHC, compiler flags, etc. The algorithm used may be a useful thing (backtracking vs NFA/DFA).
- All backends support String, (Seq Char), ByteString, and ByteString.Lazy.

Overview

Backend	POSIX/Perl	Speed	Native impl?
regex-posix	POSIX	Very slow	No
regex-parsec	Both	Slow	Yes
regex-tre	POSIX	Fast	No
regex-tdfa	Perl	Fast	Yes
regex-pcre	Perl	Fast	No
regex-dfa	POSIX	Fast	Yes

Table: Feature matrix of backends

Outline

- Module that provides the Regex backend that wraps the c posix regex api. This is the backend being used by the regex-compat package to replace Text.Regex.
- The version that comes with GHC 6.6 is 0.71.
- Haskell Hierarchical Libraries: Text.Regex.Posix.

- Module that provides the Regex backend that wraps the c posix regex api. This is the backend being used by the regex-compat package to replace Text.Regex.
- The version that comes with GHC 6.6 is 0.71.
- Haskell Hierarchical Libraries: Text.Regex.Posix.

- Module that provides the Regex backend that wraps the c posix regex api. This is the backend being used by the regex-compat package to replace Text.Regex.
- The version that comes with GHC 6.6 is 0.71.
- Haskell Hierarchical Libraries: Text.Regex.Posix.

Outline

Introduction.

- While Haskell’s regexp libraries provide the same functionality that you’ll find in Perl, Python, and Java, they provide a rich and fairly abstract interface that can be daunting to newcomers.
- The standard library that implements regexps is Text.Regex.Posix. As the name suggests, this wraps the system’s native POSIX extended regexp library.

Introduction.

- While Haskell's regexp libraries provide the same functionality that you'll find in Perl, Python, and Java, they provide a rich and fairly abstract interface that can be daunting to newcomers.
- The standard library that implements regexps is Text.Regex.Posix. As the name suggests, this wraps the system's native POSIX extended regexp library.

Introduction.

- Perl-style regexps are much more expressive, and hence far more widely used. POSIX regexps look superficially similar to Perl-style regexps, but they´re not as expressive, and they have different matching behaviour.
- The only advantage of the POSIX regexp library is that it's bundled with GHC; you don't have to fetch any extra bits to get it to work.

Introduction.

- Perl-style regexps are much more expressive, and hence far more widely used. POSIX regexps look superficially similar to Perl-style regexps, but they´re not as expressive, and they have different matching behaviour.
- The only advantage of the POSIX regexp library is that it's bundled with GHC; you don't have to fetch any extra bits to get it to work.

Introduction.

- The simplest way to use regexps is via the "= " function, which we normally use as an infix operator (It's exactly the same operator that Perl uses for regexp matching.)
- This function is polymorphic in both its arguments and its return type. Here's a simplified type signature: `(= ) :: string -> pattern -> result`.
- Since the result type is polymorphic, `ghci` has no way to infer what type we might actually want.

Introduction.

- The simplest way to use regexps is via the “= ” function, which we normally use as an infix operator (It’s exactly the same operator that Perl uses for regexp matching.)
- This function is polymorphic in both its arguments and its return type. Here’s a simplified type signature: `(= ) :: string -> pattern -> result`.
- Since the result type is polymorphic, `ghci` has no way to infer what type we might actually want.

Introduction.

- The simplest way to use regexps is via the “= ” function, which we normally use as an infix operator (It’s exactly the same operator that Perl uses for regexp matching.)
- This function is polymorphic in both its arguments and its return type. Here’s a simplified type signature: `(= ) :: string -> pattern -> result`.
- Since the result type is polymorphic, `ghci` has no way to infer what type we might actually want.

Introduction.

- By constraining the type of the result to Bool, we get a simple `True` or `False` answer when we ask if the match has succeeded. But we can also use String as the result type, which gives as the first string that matches, or an empty string if the match fails.
- If we use [String], we get a list of every string that matches. It's also possible to get more detail about the context in which a match occurred using tuples.

Introduction.

- By constraining the type of the result to Bool, we get a simple `True` or `False` answer when we ask if the match has succeeded. But we can also use String as the result type, which gives as the first string that matches, or an empty string if the match fails.
- If we use [String], we get a list of every string that matches. It's also possible to get more detail about the context in which a match occurred using tuples.

Examples.

```
$ ghci

Loading package base ... linking ... done.

> :mod +Text.Regex.Posix

> "bar" =~ "(foo|bar)"

> "bar" =~ "(foo|bar)" :: Bool
True
```

Examples.

```
> let pat = "(foo[a-z]*bar|quux)"

> "foodiebar, fooquuxbar" =~ pat :: String
"foodiebar"

> "nomatch" =~ pat :: String
""

> "foodiebar, fooquuxbar" =~ pat :: [String]
["foodiebar","fooquuxbar"]
```

Examples (Text.Regex.Base.)

```
> :mod +Text.Regex.Base

> "the foodiebar" =~ pat :: (MatchOffset,MatchLengt
(4,9)

> "no match" =~ pat :: [(MatchOffset,MatchLength)]
[]

> a <- "foo" =~~ "foo" :: IO Int
1

> "foo" =~~ "bar" :: Maybe String
Nothing
> "foo" =~~ "foo" :: Maybe String
Just "foo"
```

Summary

- Outlook
  - In practice, you will want to avoid the Text.Regex.Posix implementation.
  - The regexp library is experimental.

Summary

- Outlook
  - In practice, you will want to avoid the Text.Regex.Posix implementation.
  - The regexp library is experimental.

Summary

- Outlook
    - In practice, you will want to avoid the Text.Regex.Posix implementation.
    - The regexp library is experimental.

For Further Information

John E. Hopcroft and Rajeev Motwani and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2007.

Bryan O'Sullivan, *A Haskell regular expression tutorial*. Bryan O'Sullivan's blog, 2007.

Tony Stubblebine, *Regular Expression Pocket Reference*. O'Reilly Media, 2003.

For Further Information

- John E. Hopcroft and Rajeev Motwani and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2007.
- Bryan O'Sullivan, *A Haskell regular expression tutorial*. Bryan O'Sullivan's blog, 2007.
- Tony Stubblebine, *Regular Expression Pocket Reference*. O'Reilly Media, 2003.

For Further Information

- John E. Hopcroft and Rajeev Motwani and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2007.
- Bryan O'Sullivan, *A Haskell regular expression tutorial*. Bryan O'Sullivan's blog, 2007.
- Tony Stubblebine, *Regular Expression Pocket Reference*. O'Reilly Media, 2003.