

[« Back to article overview.](#)

The true power of regular expressions 15. June 2012

As someone who frequents the [PHP tag on StackOverflow](#) I pretty often see questions about how to parse some particular aspect of HTML using regular expressions. A common reply to such a question is:

You cannot parse HTML with regular expressions, because HTML isn't regular. Use an XML parser instead.

This statement - in the context of the question - is somewhere between very misleading and outright wrong. What I'll try to demonstrate in this article is how powerful modern regular expressions *really* are.

What does “regular” actually mean?

In the context of [formal language theory](#), something is called “regular” when it has a grammar where all production rules have one of the following forms:

```
B -> a
B -> aC
B -> ε
```

You can read those \Rightarrow rules as “The left hand side can be replaced with the right hand side”. So the first rule would be “B can be replaced with a”, the second one “B can be replaced with aC” and the third one “B can be replaced with the empty string” (ε is the symbol for the empty string).

So what are B, C, ε and ā? By convention, uppercase characters denote so called “non-terminals” - symbols which *can* be broken down further - and lowercase characters denote “terminals” - symbols which *cannot* be broken down any further.

All that probably sounds a bit abstract, so let's look at an example: Defining the natural numbers as a grammar.

```
N -> 0
N -> 1
N -> 2
N -> 3
N -> 4
N -> 5
N -> 6
N -> 7
N -> 8
N -> 9
N -> 0N
N -> 1N
N -> 2N
N -> 3N
N -> 4N
N -> 5N
N -> 6N
N -> 7N
N -> 8N
N -> 9N
```

What this grammar says is:

```
A natural number (N) is
... one of the digits 0 to 9
or
... one of the digits 0 to 9 followed by another natural number (N)
```

In this example the digits 0 to 9 would be terminals (as they can't be broken down any further) and N would be the only non-terminal (as it can be and is broken down further).

If you have another look at the rules and compare them to the definition of a regular grammar from above, you'll see that they meet the criteria: The first ten rules are of the form $B \Rightarrow a$ and the second ten rules follow the form $B \Rightarrow aC$. Thus the grammar defining the natural numbers is *regular*.

Another thing you might notice is that even though the above grammar defines such a simple thing, it is already quite bloated. Wouldn't it be better if we could express the same concept in a more concise manner?

And that's where regular expressions come in: The above grammar is equivalent to the regex `[0-9]*` (which is a hell lot simpler). And this kind of transformation can be done with *any* regular grammar: Every regular grammar has a corresponding regular expression which defines all its valid strings.

What can regular expressions match?

Thus the question arises: Can regular expressions match only regular grammars, or can they also match more? The answer to this is both *yes* and *no*:

Regular expressions in the formal grammar sense can (pretty much by definition) only parse regular grammars and nothing more.

But when programmers talk about “regular expressions” they aren't talking about formal grammars. They are talking about the regular expression *derivative* which their language implements. And those regex implementations are only very slightly related to the original notion of regularity.

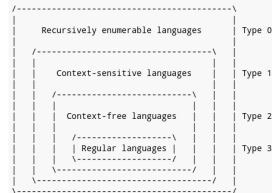
Any modern regex flavor can match a *lot* more than just regular languages. How much exactly, that's what the rest of the article is about.

To keep things simple, I'll focus on the PCRE regex implementation in the following, simply because I know it best (as it's used by PHP). Most other regex implementations are quite similar though, so most stuff should apply to them too.

The language hierarchy

In order to analyze what regular expressions can and cannot match, we first have to look at what other types of languages there are. A good starting point for this is the [Chomsky hierarchy](#):

Chomsky hierarchy:



As you can see the Chomsky hierarchy divides formal languages into four types:

Regular languages (Type 3) are the least-powerful, followed by the context-free languages (Type 2), the context-sensitive languages (Type 1) and at last the all-mighty recursively enumerable languages (Type 0).

The Chomsky hierarchy is a containment hierarchy, so the smaller boxes in the above image are fully contained in the larger boxes. For example every regular language is also a context-free language (but *not* the other way around!)

So, let's move one step up in that hierarchy: We already know that regular expressions can match any regular language. But can they also match context-free languages?

(Reminder: When I say “regular expression” here I obviously mean it in the programmer sense, not the formal language theory sense.)

matching context-free languages

The answer to this is yes, they can!

Let's take the classical example of a context-free language, namely $\{a^n b^n, n \geq 0\}$, which means "A number of *a* characters followed by the *same* number of *b* characters". The (PCRE) regex for this language is:

```
 /^(a(?!))?b$/
```

The regular expression is very simple: $(?!)$ is a reference to the first subpattern, namely $(a(?!))b$. So basically you could replace the $(?!)$ by that subpattern, thus forming a recursive dependency:

```
 /^(a(?!))?b$/
 /^(a(a(?!))b)?b$/
 /^(a(a(a(?!))b)?b)?b$/
 /^(a(a(a(a(?!))b)?b)?b)?b$/
 # and so on
```

From the above expansions it should be clear that this expression can match any string with the same number of *a*s and *b*s.

Thus regular expressions can match at least some non-regular, context-free grammars. But can they match all? To answer that, we first have to look at how context-free grammars are defined.

In a context-free grammar all production rules take the following form:

$A \rightarrow \beta$

Here *A* once again is a non-terminal symbol and β is an arbitrary string of terminals and non-terminals. Thus every production rule of a context-free grammar has a non-terminal on the left hand side and an arbitrary symbol string on the right hand side.

As an example, have a look at the following grammar:

```
function_declaration -> T_FUNCTION is_ref T_STRING '(' parameter_list ')' ' ' inner_statement_list ')'
is_ref -> 'g'
is_ref -> ε

parameter_list -> non_empty_parameter_list
parameter_list -> ε

non_empty_parameter_list -> parameter
non_empty_parameter_list -> non_empty_parameter_list ',' parameter

// ... ....
```

What you see there is an excerpt from the PHP grammar (just a few sample rules). The syntax is slightly different from what we used before, but should be easy to understand. One aspect worth mentioning is that the uppercase *T_SOMETHING* names here also are terminal symbols. These symbols which are usually called *tokens* encode more abstract concepts. E.g. *T_FUNCTION* represents the *function* keyword and *T_STRING* is a label token (like *getUserById* or *some_other_name*).

I'm using this example to show one thing: Context-free grammars are already powerful enough to encode quite complex languages. That's why pretty much all programming languages have a context-free grammar. In particular this also includes well-formed HTML.

Now, back to the actual question: Can regular expressions match all context-free grammars? Once again, the answer is yes!

This is pretty easy to prove as regular expressions (at least PCRE and similar) provide a syntax very similar to the above for constructing grammars:

```
/(
  (?DEFINE)
    (?<addr_spec> (?<local_part> @ (?<domain> )
      (?<local_part> (?<dot_atom> | (?<quoted_string> | (?<obs_local_part> )
        (?<domain> (?<dot_atom> | (?<domain_literal> | (?<obs_domain> )
          (?<domain_literal> (?<CFWS>? \[ (?<?> (?<FWS>? (?<dtxt>)* (?<FWS>? \] (?<CFWS>? )
            (?<quoted_pair> \[ (?<?> (?<VCCHAR> | (?<GSP> ) | (?<obs_gp> )
              (?<dot_atom_text> (?<dot_atom_text> (?<CFWS>? )
                (?<dot_atom_text> (?<atext> (?<?> \. (?<atext>)* )
                (?<atext> [a-zA-Z0-9!#$%&'*+,-./:;=?"@_{}~] (?<?>)*
              (?<atom> (?<CFWS>? (?<atext> (?<CFWS>? )
                (?<word> (?<atom> | (?<quoted_string> )
                (?<quoted_string> (?<CFWS>? " (?<?> (?<FWS>? (?<qcontent>)* (?<FWS>? " (?<CFWS>? )
                (?<qcontent> (?<dtxt> | (?<quoted_pair> )
                (?<qtext> \x21 | [\x23-\x5b] | [\x5d-\x7e] | (?<obs_qtext> )
            )
          )
        )
      )
    )
    # comments and whitespace
    (?<FWS> (?<?> (?<GSP>)* \r\n )? (?<GSP>+ | (?<obs_FWS> )
      (?<CFWS> (?<?> (?<FWS>? (?<comment>)* (?<FWS>? | (?<FWS> )
      (?<comment> \[ (?<?> (?<FWS>? (?<cccontent>)* (?<FWS>? \] )
      (?<cccontent> (?<ctext> | (?<quoted_pair> | (?<comment> )
      (?<ctext> [\x21-\x2f] | [\x2a-\x5b] | [\x5d-\x7e] | (?<obs_ctext> )
    )
    # obsolete tokens
    (?<obs_domain> (?<atom> (?<?> \. (?<atom>)* )
    (?<obs_local_part> (?<word> (?<?> \. (?<word>)* )
    (?<obs_dtext> (?<obs_NO_WS_CTL> | (?<quoted_pair> )
    (?<obs_gp> \[ (?<?> \x00 | (?<obs_NO_WS_CTL> | \n | \r ) )
    (?<obs_FWS> (?<GSP>+ (?<?> \r\n (?<GSP>)* )
    (?<obs_ctext> (?<obs_NO_WS_CTL> )
    (?<obs_NO_WS_CTL> [\x01-\x08] | \x0b | \x0c | [\x0e-\x1f] | \x7f )
    # character class definitions
    (?<VCHAR> [\x21-\x7e] )
    (?<GSP> [ \t ] )
  )
  (?<addr_spec>)
)/x
```

What you see above is a regular expression for matching email addresses as per REC 5322. It was constructed simply by transforming the BNF rules from the RFC into a notation that PCRE understands.

The syntax is quite simple:

All rule definitions are wrapped into a *DEFINE* assertion, which basically means that all those rules should not be directly matched against, they should just be defined. Only the $\wedge(?<addr_spec>)\$$ part at the end specifies what should be matched.

The rule definitions are actually not really "rules" but rather named subpatterns. In the previous $(a(?!))b$ example the *!* referenced the first subpattern. With many subpatterns this obviously is impractical, thus they can be named. So $(?<xyz> \dots)$ defines a pattern with name *xyz*. $(?&xyz)$ then references it.

Also, pay attention to another fact: The regular expression above uses the *x* modifier. This instructs the engine to ignore whitespace and to allow *#*-style comments. This way you can nicely format the regex, so that other people can actually understand it. (Much unlike this RFC 822 email address regex...)

The above syntax thus allows simple mappings from grammars to regular expressions:

```
A -> B C
A -> C D
// becomes
(?<A> (?<B> (?<C>
  | (?<C> (?<D>
)
```

The only catch is: Regular expressions don't support left recursion. E.g. taking the above definition of a parameter list:

```
non_empty_parameter_list -> parameter
non_empty_parameter_list -> non_empty_parameter_list ',' parameter
```

You *can't* directly convert it into a grammar based regex. The following will not work:

```
(?<non_empty_parameter_list>
  (?<parameter>
    | (?<non_empty_parameter_list> ) , (?<parameter>
  )
```

The reason is that here *non_empty_parameter_list* appears as the leftmost part of its own rule definition. This is called left-recursion and is very common in grammar definitions. The reason is that the LALR(1) parsers which are usually used to parse them handle left-recursion much better than right-recursion.

But, no fear, this does not affect the power of regular expressions at all. Every left-recursive grammar can be transformed to a right-recursive one. In the above example it's as simple as swapping the two parts:

```
non_empty_parameter_list -> parameter
non_empty_parameter_list -> parameter ',' non_empty_parameter_list
```

So now it should be clear that regular expressions can match any context-free language (and thus pretty much all languages which programmers are confronted with). Only problem is: Even though regular expressions can *match* context-free languages nicely, they can't usually *parse* them. Parsing means converting some string into an abstract syntax tree. This is not possible using regular expressions, at least not with PCRE (sure, in Perl where you can embed arbitrary code into a regex you can do pretty much everything...).

Still, the above `DEFINE` based regex definition has proven to be very useful to me. Usually you don't need full parsing support, but want to just match (e.g. email addresses) or extract small pieces of data (not the whole parse tree). Most complex string processing problems can be made much simpler using grammar based regexes :)

At this point, let me point out again what I already quickly mentioned earlier: Well-formed HTML is context-free. So you can *match* it using regular expressions, contrary to popular opinion. But don't forget two things: Firstly, most HTML you see in the wild is *not* well-formed (usually not even close to it). And secondly, just because you *can*, doesn't mean that you *should*. You could write your software in Brainfuck, still for some reason you don't.

My opinion on the topic is: Whenever you need generic HTML processing, use a DOM library of your choice. It'll gracefully handle malformed HTML and take the burden of parsing from you. On the other hand if you are dealing with specific situations a quick regular expression is often the way to go. And I have to admit: Even though I often tell people to not parse HTML with regular expressions I do it myself notoriously often. Simply because in most cases I deal with specific and contained situations in which using regex is just simpler.

Context-sensitive grammars

Now that we covered context-free languages extensively, let's move up one step in the Chomsky hierarchy: Context-sensitive languages.

In a context-sensitive language all production rules have the following form:

```
αβ → αγβ
```

This mix of characters might start to look more complicated, but it is actually quite simple. At its core you still have the pattern $A \rightarrow \gamma$, which was how we defined context-free grammars. The new thing now is that you additionally have α and β on both sides. Those two form the *context* (which also gives this grammar class the name). So basically A can now only be replaced with γ if it has α to its left and β to its right.

To make this more clear, try to interpret the following rules:

```
a b A -> a b c
a B C -> a Q H C
H B -> H C
```

The English translations would be:

```
Replace 'A' with 'c', but only if it has 'a b' to its left.
Replace 'B' with 'Q H', but only if it has 'a' to its left and 'c' to its right.
Replace 'B' with 'C', but only if it has 'H' to its left.
```

Context-sensitive languages are something that you will rarely encounter during “normal” programming. They are mostly important in the context of natural language processing (as natural languages are clearly not context-free. Words have different meaning depending on context). But even in natural language processing people usually work with so called “mildly context-sensitive languages”, as they are sufficient for modeling the language but can be parsed much faster.

To understand just how powerful context-sensitive grammars are let's look at another grammar class, which has the exact same expressive power as the context-sensitive ones: Non-contracting grammars.

With non-contracting grammars every production rule has the form $\alpha \rightarrow \beta$ where both α and β are arbitrary symbol strings with just one restriction: The number of symbols on the right hand side is not less than on the left hand side. Formally this is expressed in the formula $|\alpha| \leq |\beta|$ where $|x|$ denotes the length of the symbol string.

So non-contracting grammars allow rules of any form as long as they don't shorten the input. E.g. $A B C \rightarrow H Q$ would be an invalid rule as the left hand side has three symbols and the right hand side only two. Thus this rule would be shortening (or “contracting”). The reverse rule $H Q \rightarrow A B C$ on the other hand would be valid, as the right side has more symbols than the left, thus being lengthening.

This equivalence relationship of context-sensitive grammars and non-contracting grammars should make pretty clear that you can match near-everything with a context-sensitive grammar. Just don't shorten :)

To get an impression of why both grammar kinds have the same expressive power look at the following transformation example:

```
// the non-contracting grammar
A B -> C D
// can be transformed to the following context-sensitive grammar
A B -> A X
A X -> Y X
Y X -> Y D
Y D -> C D
```

Anyways, back to regular expressions. Can they match context-sensitive languages too?

This time I can't give you definite answer. They certainly can match *some* context-sensitive languages, but I don't know whether they can match *all* of them.

An example of a context-sensitive language that can be easily matched using regex is a modification of the context-free language $\{a^n b^n, n>0\}$ mentioned above. When you change it into $\{a^n b^n c^n, n>0\}$, i.e. some number of a s followed by the same number of b s and c s, it becomes context-sensitive.

The PCRE regex for this language is this:

```
/?^
    (?=a(?-1)?b)c)
a+(b(?-1)?c)
$/x
```

If you ignore the $(?=...)$ assertion for now you're left with $a+(b(?-1)?c)$. This checks that there is an arbitrary number of a s, followed by the same number of b s and c s. The $(?=-1)$ is a relative subpattern reference and means “the last defined subpattern”, which is $(b(?-1)?c)$ in this case.

The new thing now is the $(?=...)$ which is a so called zero-width lookahead assertion. It checks that the following text matches the pattern, but it does not actually consume the text. Thus the text is basically checked against both patterns at the same time. The $a+(b(?=-1)?c)$ part verifies that the number of b s and c s is the same and the $(a(?-1)?b)c$ part checks that the number of a s and b s is the same. Both pattern together thus ensure that the number of all three characters is the same.

In the above regex you can already see how the concept of “context” is realized in regular expressions: Using assertions. If we get back to the definition of a context-sensitive grammar, you could now say that a production rule of type

```
αβ → αγβ
```

can be converted into the following regex `DEFINE` rule:

```
(?<A> (?<= α ) γ (?= β ) )
```

This would then say that A is γ , but only if it has α to its left and β to its right.

Now, the above might look as if you can easily convert a context-sensitive grammar into a regular expression, but it's not actually true. The reason is that lookahead assertions $(?=...)$

Regular expressions, like most formally used, are subject to some constraints. One of them is that they have one very significant limitation: They have to be fixed-width. This means that the length of the text matched by the assertion has to be known in advance. E.g. you can write `(?<= a(bc|cd))`, but you can't write `(?<= ab*)`. In the first case the assertion matches exactly three characters in any case, thus being fixed-width. In the second case on the other hand the assertion could match `ab`, `abb`, `abbb` etc. All of those have different lengths. Thus the engine can't know when it should start to match them and as such they are simply disallowed.

This pretty much blows the easy conversion of context-sensitive grammars to regex. Pretty much all such grammars require variable-width lookbehind assertions.

But the fact that there is no direct context-sensitive grammar to regex conversion doesn't by itself mean that regular expressions can't match all of them. E.g. the above `{anbncn | n>0}` language also has a grammar that would require variable-width lookbehind assertions. But we can still avoid using them as regex isn't bound to specifying rules in a grammar. Maybe the same is possible for all other context-sensitive grammars too. I honestly don't know.

So, what can we say here? Regex can match at least *some* context-sensitive languages, but it's unknown whether it can match *all* of them.

Unrestricted grammars

The next grammar class in the Chomsky hierarchy are the unrestricted grammars. The language set which one can form using them is the set of all recursively enumerable languages.

There is little to say about unrestricted grammars as they are, well, unrestricted. Production rules for unrestricted grammars have the form `α -> β`, where `α` and `β` are symbol strings with no restrictions whatsoever.

So basically unrestricted grammars remove the "non-contracting" part of the non-contracting grammars. Thus for them `A B C -> H Q` would be a valid rule, even though previously it wasn't.

How powerful are unrestricted grammars exactly? They are as powerful as it gets: They are Turing-complete. There even is a "programming language" which is based on unrestricted grammars: [Thue](#). As it is Turing-complete it can do everything that other languages can do.

One implication of being Turing-complete is that checking whether a certain string adheres to some grammar is undecidable for the general case.

Sadly I can't say anything whatsoever about how regular expressions and unrestricted grammars relate. Heck, I couldn't even find an example of a meaningful unrestricted grammar (that wasn't non-contracting).

But now that we started talking about Turing-completeness we get to another point:

Regular expressions with backreferences are NP-complete

There is another very powerful regular expression feature that I did not mentioned previously: backreferences.

E.g. consider this very simple regex:

```
/^(.+)\1$/
```

`{.+}` matches some arbitrary text and `\1` matches the *same* text. In general `\n` means "whatever the *n*th subpattern matched". E.g. if `{.+}` matched `foo`, then `\1` will also match only `foo` and nothing else. Thus the expression `{.+}\1` means "Some text followed by a copy of itself".

What this simple regex matches is called the "copy language" and is another typical example of a context-sensitive language.

Similarly you can match the other example grammars from above using backreferences:

```
# {anbn | n>0} (context-free)
/^ (?= a* (\1?+ b) ) >+ \1 $/x
# {anbncn | n>0} (context-sensitive)
/^ (?= a* (\1?+ b) b* (\2?+ c) ) >+ \1 \2 $/x
```

Explaining how these work is outside the scope of this article, but you can read an excellent explanation on [StackOverflow](#).

As you can see, the mere addition of backreference (without subpattern recursion support) already adds a lot of power to regular expressions. The addition is actually so powerful that it makes matching of regular expressions an NP-complete problem.

What does NP-complete mean? NP-complete is a computational complexity class for decision problems in which many "hard" problems fall. Some examples of NP-complete problems are the traveling salesman problem (TSP), the boolean satisfiability problem (SAT) and the knapsack problem (BKP).

One of the main conditions for a problem being NP-complete is that every other NP problem is reducible to it. Thus all NP-complete problems are basically interchangeable. If you find a fast solution to one of them, you got a fast solution to all of them.

So if somebody found a fast solution to a NP-complete problem, pretty much all of the computationally hard problems of humanity would be solved all in one strike. This would mean the end to civilisation as we know.

To prove that regular expressions with backreferences are indeed NP-complete one can simply take one of the known NP-complete problems and prove that it can be solved using regular expressions. As an example I choose the 3-CNF SAT problem:

3-CNF SAT stands for "3-conjunctive normal form boolean satisfiability problem" and is quite easy to understand. You get a boolean formula of the following form:

```
(1$a || $b || $d)
&& ( $a || !$c || $d)
&& ( $a || !$b || !$d)
&& ( $b || !$c || !$d)
&& (1$a || $c || !$d)
&& ( $a || $b || $c)
&& (1$a || !$b || !$c)
```

Thus the boolean formula is made up of a number of clauses separated by ANDs. Each of those clauses consists of three variables (or their negations) separated by ORs. The 3-CNF SAT problem now asks whether there exists a solution to the given boolean formula (such that it is true).

The above boolean formula can be converted to the following regular expression:

```
$regex = '/^
(x?)(x?)(x?)(x?) .+ ;
(?: x\1 | \2 | \4 |
(?: \1 | x\3 | \4 |
(?: \1 | x\2 | x\4 |
(?: \2 | x\3 | x\4 |
(?: x\1 | \3 | x\4 |
(?: \1 | \2 | \3 |
(?: x\1 | x\2 | x\3 |
$/x';
$string = 'xxxx;x,x,x,x,x,x,x,x';
var_dump(preg_match($regex, $string, $matches));
var_dump($matches);
```

If you run this code you'll get the following `$matches` result:

```
array(5) {
  [0] => string(19) "xxxx;x,x,x,x,x,x,x,x"
  [1] => string(1) "x"
  [2] => string(1) "x"
  [3] => string(0) ""
  [4] => string(0) ""
}
```

This means that the above formula is satisfied if `$a = true`, `$b = true`, `$c = false` and `$d = false`.

The regular expression works with a very simple trick: For ever 3-clause the string contains a `x` which has to be matched. So if you have something like `(?<= \1)(?<= \2)(?<= \3)`

contains a `x`, which has to be matched, so if you have something like `\d{x}\d{1}|\d{2}|\d{4}`, in the regex, then the string can be only matched if either `\d` is `x` (true), `\d{3}` is the empty string (false) or `\d{4}` is `x` (true).

The rest is left up to the engine. It'll try out different ways of matching the string until it either finds a solution or has to give up.

Wrapping up

As the article was quite long, here a summary of the main points:

- The “regular expressions” used by programmers have very little in common with the original notion of regularity in the context of formal language theory.
- Regular expressions (at least PCRE) can match all context-free languages. As such they can also match well-formed HTML and pretty much all other programming languages.
- Regular expressions can match at least some context-sensitive languages.
- Matching of regular expressions is NP-complete. As such you can solve any other NP problem using regular expressions.


But don't forget: just because you *can*, doesn't mean that you *should*. Processing HTML with regular expressions is a really bad idea in some cases. In other cases it's probably the best thing to do.

Just check what the easiest solution to your particular problem is and use it. If you choose to solve a problem using regular expressions, don't forget about the `x` modifier, which allows you to nicely format your regex. For complex regular expressions also don't forget to make use of `DEFINE` assertions and named subpatterns to keep your code clean and readable.

That's it.

If you liked this article, you may want to [browse my other articles](#) or [follow me on Twitter](#).


ALSO ON NIKIE'S BLOG



Internal value representation in PHP

6 years ago • 10 comments

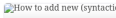
My last article described the improvements to the hashtable ...



PHP 7 Virtual Machine

4 years ago • 6 comments


This article aims to provide an overview of the Zend Virtual ...



How to add new (syntactic) features

8 years ago • 26 comments

Several people have recently asked me where you should start if you ...




Methods on primitive types in PHP

7 years ago • 20 comments

A few days ago Anthony Ferrara wrote down s thoughts on the futur...

59 Comments nikie's Blog Disqus' Privacy Policy Login





Recommend 35 Tweet Share Sort by Best



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS



Name