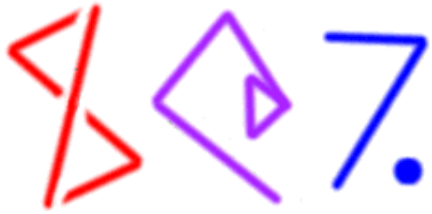
Regular Expression Mastery

M. J. Dominus

Plover Systems Co.

mjd-tpc-regex-@plover.com



v1.09 (September, 2003)

Regular Expressions

Regexes (not *Regexps*)

- Also called *patterns*
- Very useful in Perl

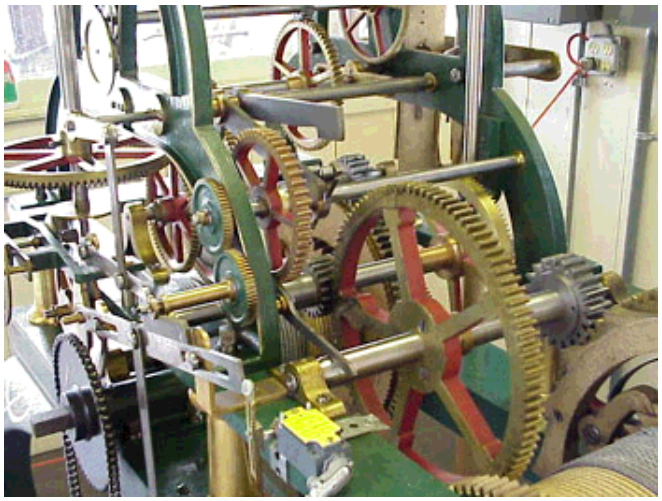
```
m/REGEX/
s/REGEX/STRING/       (left part only!)
split /REGEX/, STRING
grep /REGEX/, LIST
```

- Powerful, dangerous, risky
- Almost everyone has been unpleasantly surprised at one time or another

What We'll Do

- How regexes work on the inside
- Typical pitfalls
- How to avoid pitfalls and make regexes faster

Big Secret

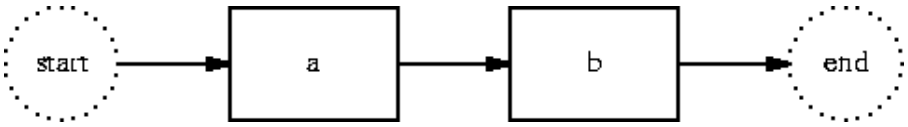


- Regex matching is like a machine running a program
- The machine is very simple, and always does the same thing
- The regex is the program, and varies the machine's behavior a little
- To understand regexes, you need to understand the machine
- The machine is called the *Regex Engine*

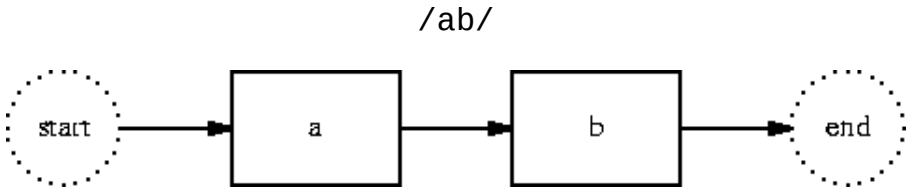
Regex Programs

- Made of *nodes*
- Each has a pointer to the next node
- Node says what to match
- For example:

/ab/



Regex Program Example

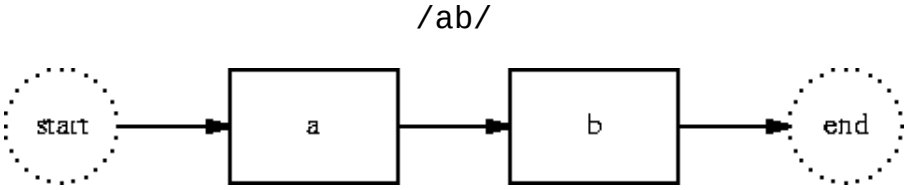


- What does this mean?
- How is the target string `ab` matched by this regex?

```
START          |a b
  a            |a b                Yes!
  b            <a|b                Yes!
  END          <a b>               Yes!
```

- We reached `END`, so the match succeeds; it found the `ab`

Regex Program Example

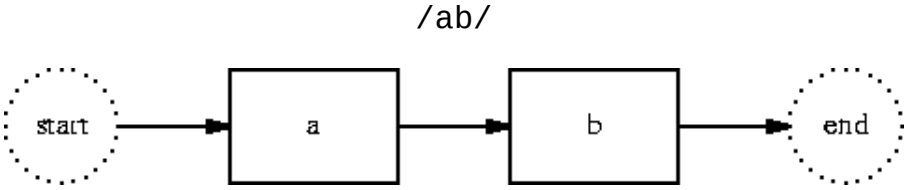


- How about `squab`?

```
START              |s q u a b
 a                 |s q u a b          Nope
START               s|q u a b
 a                  s|q u a b          Nope
START               s q|u a b
 a                  s q|u a b          Nope
START               s q u|a b
 a                  s q u|a b          Yes!
 b                  s q u<a|b          Yes!
 END                s q u<a b>         Yes!
```

- We reached `END`, so the match succeeds; it found the `ab` part of `squab`

Regex Program Example

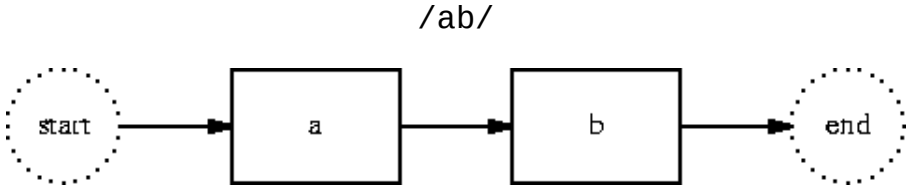


- What about dog?

```
START          |d o g
  a            |d o g          Nope
START           d|o g
  a             d|o g          Nope
START           d o|g
  a             d o|g          Nope
START           d o g|
  a             d o g|         Nope
```

- The engine ran out of characters without reaching `END`, so the match fails.

Regex Program Example

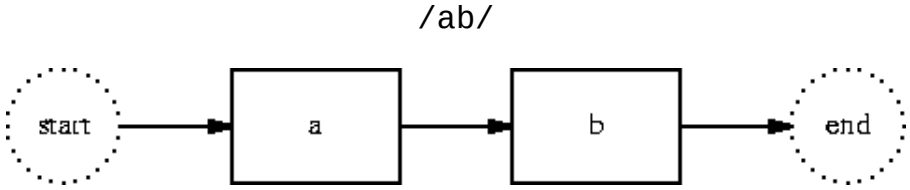


- What about aha?

```
START          |a h a
  a            |a h a          Yes!
  b            <a|h a          Nope
START           a|h a
  a             a|h a          Nope
START           a h|a
  a             a h|a          Yes!
  b             a h<a|         Nope
START           a h a|
  a             a h a|         Nope
```

- The engine ran out of characters without reaching `END`, so the match fails.

Regex Program Example



- What about `ahab`?

```
START          |a h a b
 a             |a h a b          Yes!
 b             <a|h a b          Nope
START           a|h a b
 a              a|h a b          Nope
START           a h|a b
 a              a h|a b          Yes!
 b              a h<a|b          Yes!
 END            a h<a b>         Yes!
```

- We reached `END`, so the match succeeds; it found the `ab` part of `ahab`

Regex Metacharacters

- That was simple enough...
- But the real power of regexes comes from *metacharacters*
- There are lots and lots of metacharacters:

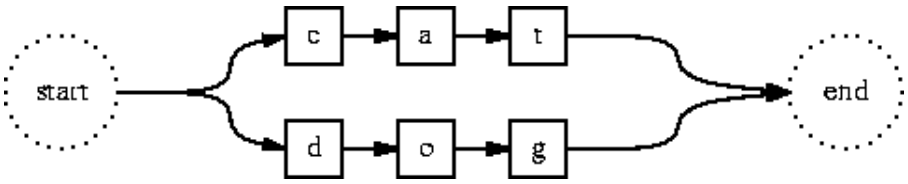
```
.       [...]           [^...]
+   *   ?   {...}
+?  *?  ??  {...}?
^       $               |
\d  \w  \s  \D  \W  \S  \b  \B
```

- We'll see all these at length later.

Regex Metacharacters

- The first metacharacter we'll see is `|`

```
/cat|dog/
```



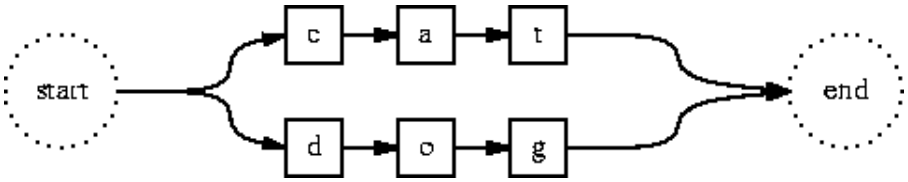
- How does this match `cat`?

```
START           |c a t
  |             |c a t
   c            |c a t          Yes!
   a            <c|a t          Yes!
   t            <c a|t          Yes!
   END          <c a t>         Yes!
```

- We reached `END`, so the match succeeds

Regex Metacharacters

/cat|dog/



- How does this match dog?

```
START            |d o g
  |              |d o g
  c              |d o g          Nope.
  d              |d o g          Yes!
  o              <d|o g          Yes!
  g              <d o|g          Yes!
  END            <d o g>         Yes!
```

- c didn't work, so it went back to try d
- **Backtracking**

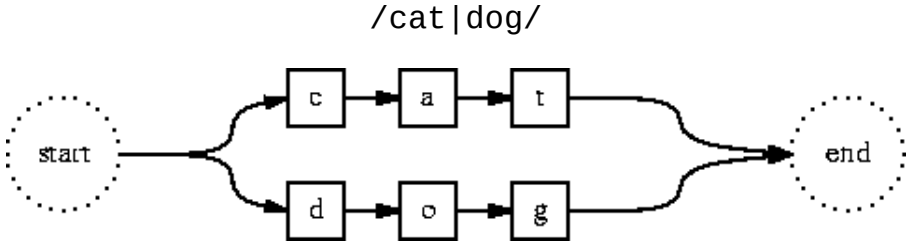
Backtracking

- Backtracking is centrally important to the regex engine
- At a choice point, the regex engine *saves its state*
- If the match fails, it returns to the last saved point
- Then it tries making the choice differently

The Big Secret

- That was it.
- You can go home now
- Or stay for some examples and details

Backtracking



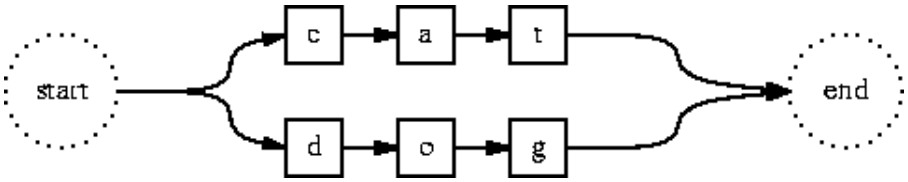
- How does this match `fish`?

```
START            |f i s h
   |             |f i s h
    c            |f i s h              Nope.
    d            |f i s h              Nope.
START             f|i s h
   |              f|i s h
    c             f|i s h              Nope.
    d             f|i s h              Nope.
START             f i|s h
   |              f i|s h
    c             f i|s h              Nope.
    d             f i|s h              Nope.
START             f i s|h
   |              f i s|h
    c             f i s|h              Nope.
    d             f i s|h              Nope.
START             f i s h|
   |              f i s h|
    c             f i s h|             Nope.
    d             f i s h|             Nope.
```

- That's all the alternatives, so the engine gives up.
- The match fails.

Backtracking

/cat|dog/

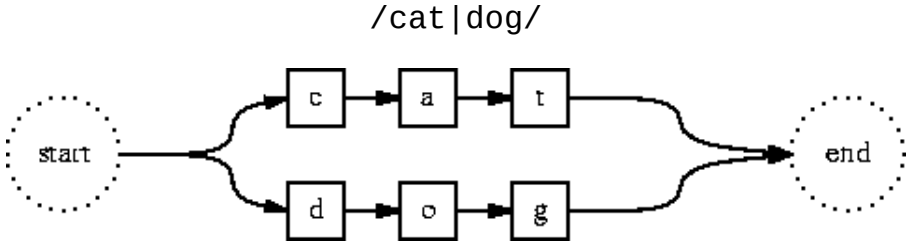


- What about `scat`?

```
START              |s c a t
   |               |s c a t
    c              |s c a t                Nope.
    d              |s c a t                Nope.
START               s|c a t
   |                s|c a t
    c               s|c a t                Yes!
    a               s<c|a t                Yes!
    t               s<c a|t                Yes!
    END             s<c a t>               Yes!
```

- We reached `END`, so the match succeeds; it found the `cat` part of `scat`

Backtracking



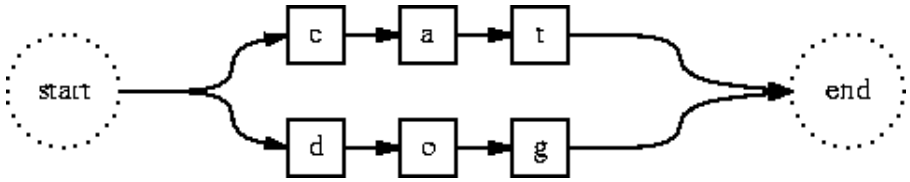
- `caricature`

```
START            |c a r i c a t u r e
   |             |c a r i c a t u r e
    c            |c a r i c a t u r e        Yes!
    a           <c|a r i c a t u r e        Yes!
    t           <c a|r i c a t u r e        Nope.
    d           |c a r i c a t u r e        Nope.
START            c|a r i c a t u r e
   |             c|a r i c a t u r e
    c            c|a r i c a t u r e        Nope.
    d            c|a r i c a t u r e        Nope.
START            c a|r i c a t u r e
   |             c a|r i c a t u r e
    c            c a|r i c a t u r e        Nope.
    d            c a|r i c a t u r e        Nope.
START            c a r|i c a t u r e
   |             c a r|i c a t u r e
    c            c a r|i c a t u r e        Nope.
    d            c a r|i c a t u r e        Nope.
START            c a r i|c a t u r e
   |             c a r i|c a t u r e
    c            c a r i|c a t u r e        Yes!
    a            c a r i<c|a t u r e        Yes!
    t            c a r i<c a|t u r e        Yes!
    END          c a r i<c a t>u r e        Yes!
```

- We reached `END`, so the match succeeds; it found the `cat` part of `caricature`

Backtracking

/cat|dog/



- `domesticate`

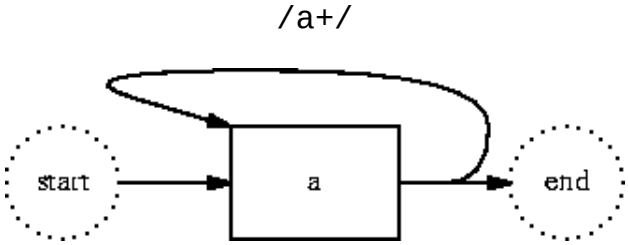
```
START              |d o m e s t i c a t e
   |               |d o m e s t i c a t e
    c              |d o m e s t i c a t e       Nope
    d              |d o m e s t i c a t e       Yes!
    o              <d|o m e s t i c a t e       Yes!
    g              <d o|m e s t i c a t e       Nope
START               d|o m e s t i c a t e
   |                d|o m e s t i c a t e
    c               d|o m e s t i c a t e       Nope
    d               d|o m e s t i c a t e       Nope
START               d o|m e s t i c a t e
   |                d o|m e s t i c a t e
    c               d o|m e s t i c a t e       Nope
    d               d o|m e s t i c a t e       Nope
```

...

```
START               d o m e s t i|c a t e
   |                d o m e s t i|c a t e
    c               d o m e s t i|c a t e       Yes!
    a               d o m e s t i<c|a t e       Yes!
    t               d o m e s t i<c a|t e       Yes!
    END             d o m e s t i<c a t>e       Yes!
```

- We reached `END`, so the match succeeds; it found the `cat` part of `domesticate`

Quantifiers

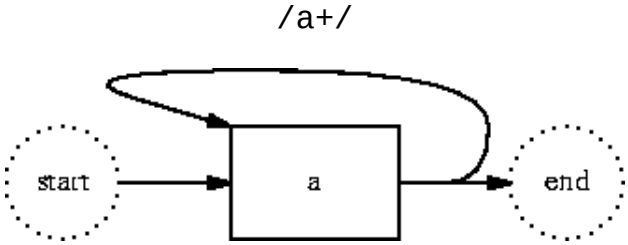


- The branch point:
  - Go on to the next thing, or
  - Go back and try another a
- `Tom`

```
START              |T o m
a                  |T o m          Nope
START               T|o m
a                   T|o m          Nope
START               T o|m
a                   T o|m          Nope
START               T o m|
a                   T o m|         Nope
```

- Out of alternatives---match fails.

Quantifiers

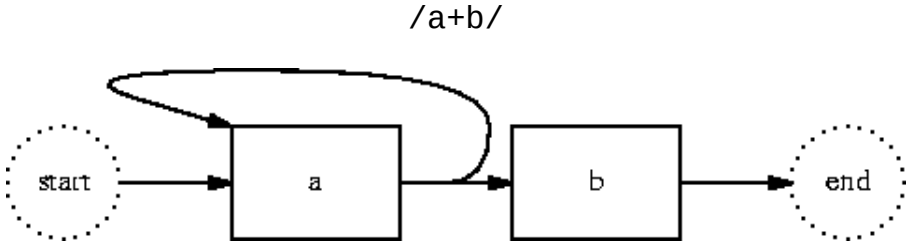


- `Nat`

```
START              |N a t
a                  |N a t          Nope
START               N|a t
a                   N|a t          Yes!
+                   N<a|t
 a                  N<a|t          Nope
 END                N<a>t          Yes!
```

- We reached `END`, so the match succeeds; it found the a part of `Nat`
- Note! It tries to get another a *before* it goes to `END`.

Greed



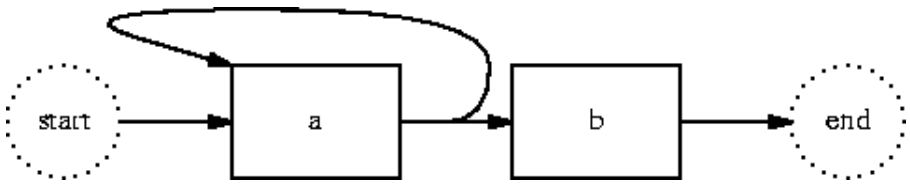
- `aaab`

```
START              |a a a b
a                  |a a a b          Yes!
+                  <a|a a b
 a                 <a|a a b          Yes!
 +                 <a a|a b
  a                <a a|a b          Yes!
  +                <a a a|b
   a               <a a a|b          Nope.
   b               <a a a|b          Yes!
   END             <a a a b>         Yes!
```

- We reached `END`, so the match succeeds; it found the `aaab` part of `aaab`
- Note! The `a+` part gobbles *all* the a's.
- We say that `+` is *greedy*.

‘Greed’ is Often Misunderstood

/a+b/



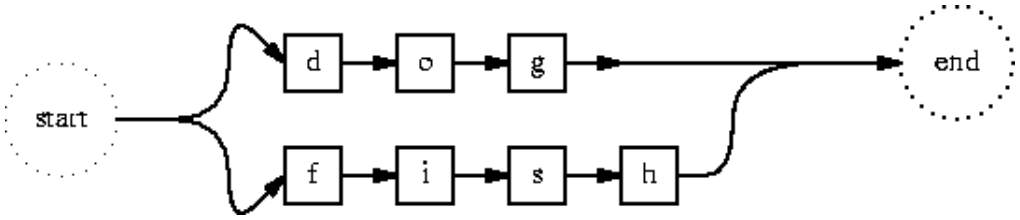
- aabaaaaaaaab

```
START              |a a b a a a a a a a a b
a                  |a a b a a a a a a a a b    Yes!
+                  <a|a b a a a a a a a a b
 a                 <a|a b a a a a a a a a b    Yes!
 +                 <a a|b a a a a a a a a b
  a                <a a|b a a a a a a a a b    Nope.
  b                <a a|b a a a a a a a a b    Yes!
  END              <a a b>a a a a a a a a b    Yes!
```

- We reached END, so the match succeeds; it found the aab part of aabaaaaaaaab
- Note! It didn’t get the *most*
  - It got the *leftest*

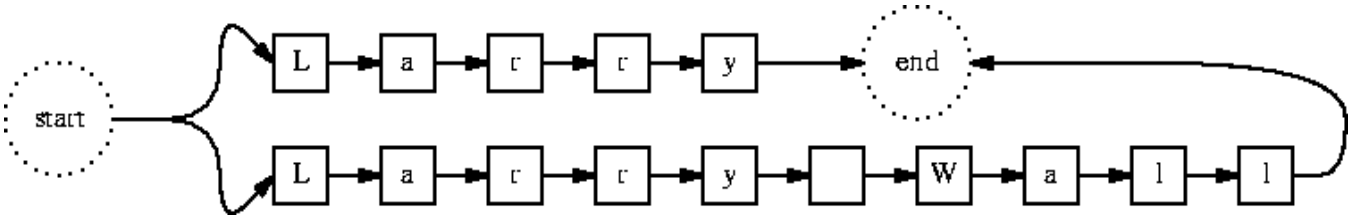
‘Greed’ is Often Misunderstood

`/dog|fish/`



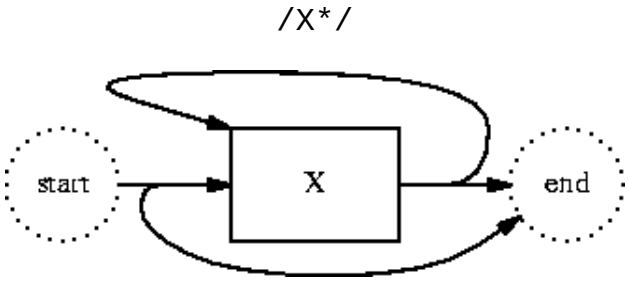
- With `dogfish` it matches `dog`, not `fish`, even though `fish` is longer
- Because `dog` is *further to the left*
- Similarly:

`/Larry|Larry Wall/`

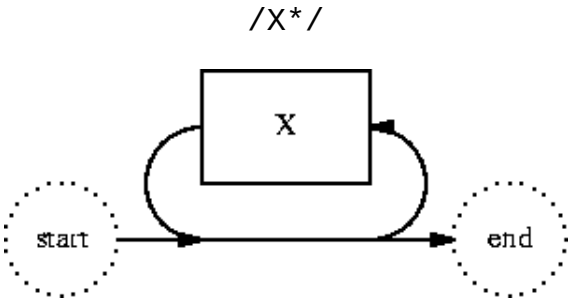


- `Good Morning Larry Wall`
- It gets `Larry`, not `Larry Wall`
  - Even though `Larry Wall` is *longer*
  - Because Perl tries the alternatives *in order*
- We’ll see later that this is useful

Digression on *

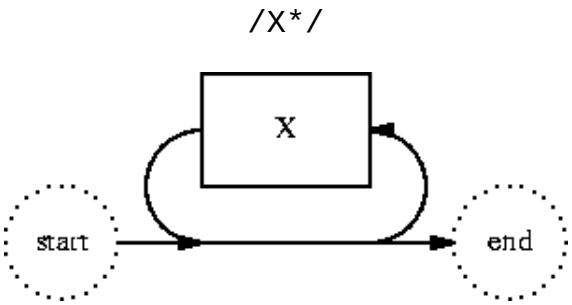


- Just like + but with an option to skip X entirely.
- Simpler diagram:



‘Greed’ is Often Misunderstood

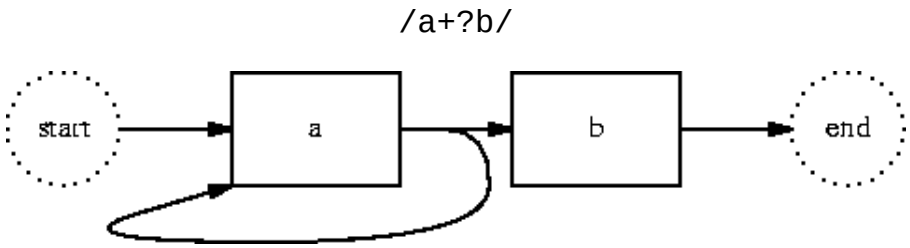
- Consider `"Hot XXX Action!" =~ s/X*//`



- It gets the empty string, not `XXX`
  - Even though `XXX` is longer
  - Because Perl starts at the leftmost position first
  - `X*` will match *zero* `X`es.
  - At the leftmost position, there *are* zero `X`es.
- Solution: Use `X+` instead
- Maxim: ‘‘Say what you mean!’’
- People over-use `*`
- Many `*`’s should be `+` instead

Anti-Greed

- What's the opposite of 'greedy'? ('Monastic'?)
- `a+?b` is just like `a+b`
- Except it tries the arrows in *the other order*



- `aaab`

```
START              |a a a b
a                  |a a a b          Yes!
+?                 <a|a a b
 b                 <a|a a b          Nope
 a                 <a|a a b          Yes!
 +?                <a a|a b
  b                <a a|a b          Nope
  a                <a a|a b          Yes!
  +?               <a a a|b
   b               <a a a|b          Yes!
   END             <a a a b>         Yes!
```

- Notice *more backtracking*
- Usually *less efficient*
- That's why the 'normal' one is greedy

Why the Greedy Ones are the Defaults

- Typical case:

```
# $s contains a line of code:
$s = '($label =~ tr/.//) < 3; # do not attach these';

# Let's strip out comments
$s =~ s/#.*//;
```

- $s is now:

```
'($label =~ tr/.//) < 3; '
```

- If it weren't greedy, $s would be:

```
'($label =~ tr/.//) < 3;  do not attach these';
```

- Suppose * were nongreedy by default....
- To get the expected behavior, you'd have to say

```
# In the parallel universe where * is nongreedy
$s =~ s/#.*$//;
```

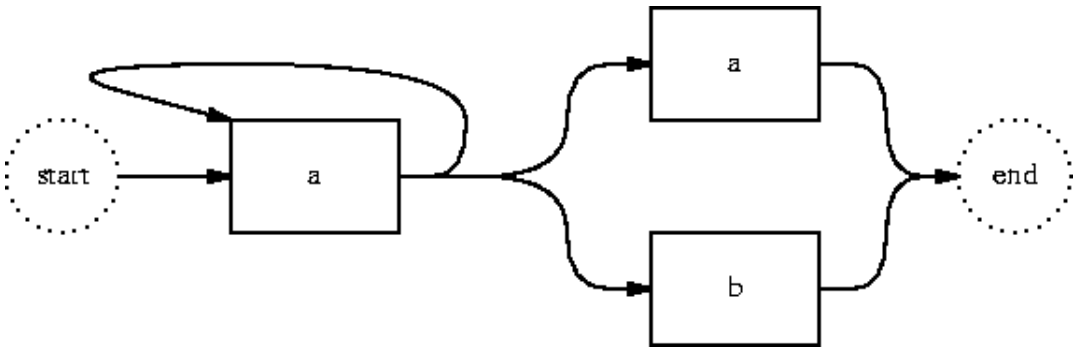
- But that would be *inefficient* because it would backtrack on every character!

Anti-Greed

- Here's an example where the greedy one is less efficient.

Greedy Version

`/a+(a|b)/`



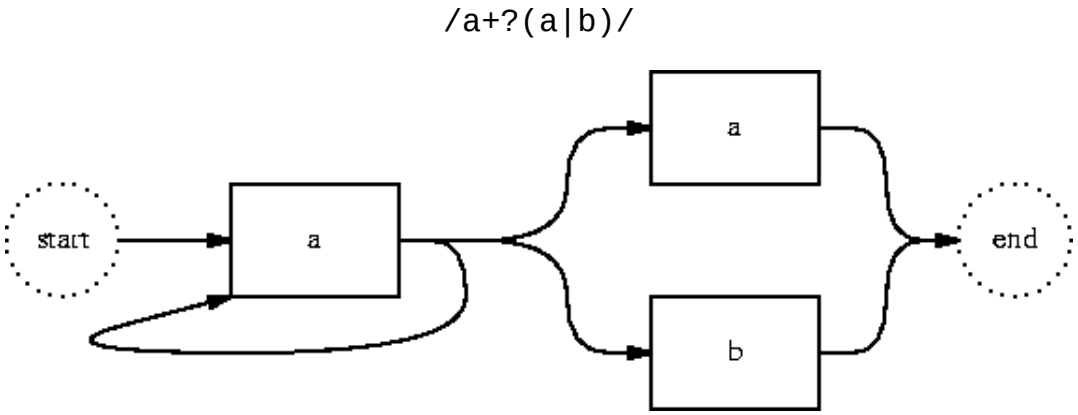
- `aaab`

```
START              |a a a b
a                  |a a a b          Yes!
+                  <a|a a b
 a                 <a|a a b          Yes!
 +                 <a a|a b
  a                <a a|a b          Yes!
  +                <a a a|b
   |               <a a a|b
    a              <a a a|b          Nope
    b              <a a a|b          Yes!
   END             <a a a b>         Yes!
```

- We reached `END`, so the match succeeds; it found the `aaab` part of `aaab`

Anti-Greed

Ungreedy Version



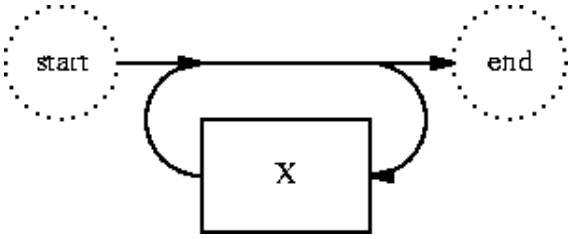
- `aaab`

```
START           |a a a b
a               |a a a b        Yes!
+               <a|a a b
 |              <a|a a b        Nope
  a             <a|a a b        Yes!
  END           <a a>a b        Yes!
```

- We reached `END`, so the match succeeds; it found the `aa` part of `aaab`
- This time the non-greedy match was more efficient
- But that's because it was *lucky* -- it happened to find a *shorter* match
- When shorter matches exist, non-greedy may find them quickly
- But if not, they are slower than their greedy counterparts.

Non-Greedy *

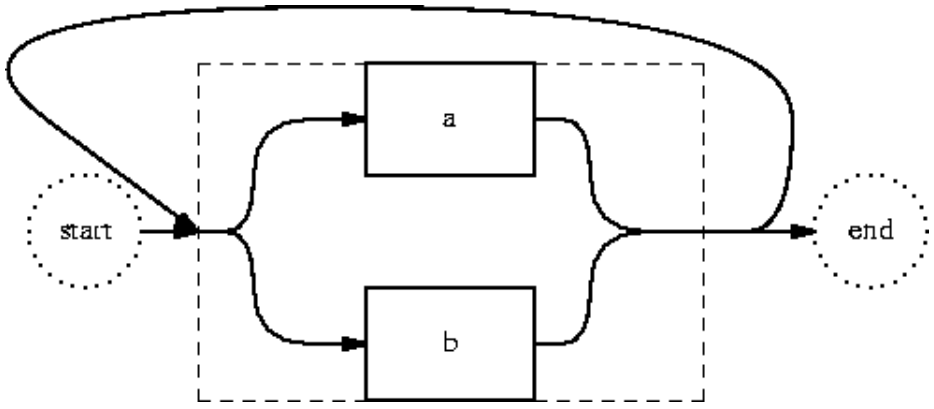
/X*?/



Nested Operations

- Pretty much as you would expect.

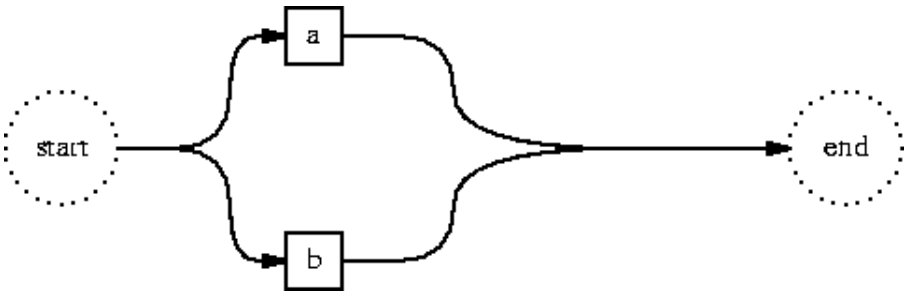
`/(a|b)+/`



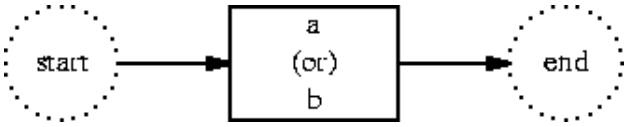
Character Classes

- `[ab]` is *not* the same as `a|b`

`/a|b/`



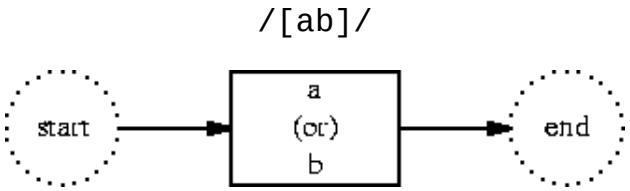
`/[ab]/`



- `[ab]` is a *single node*

Character Classes

`[ab]` vs `a|b`

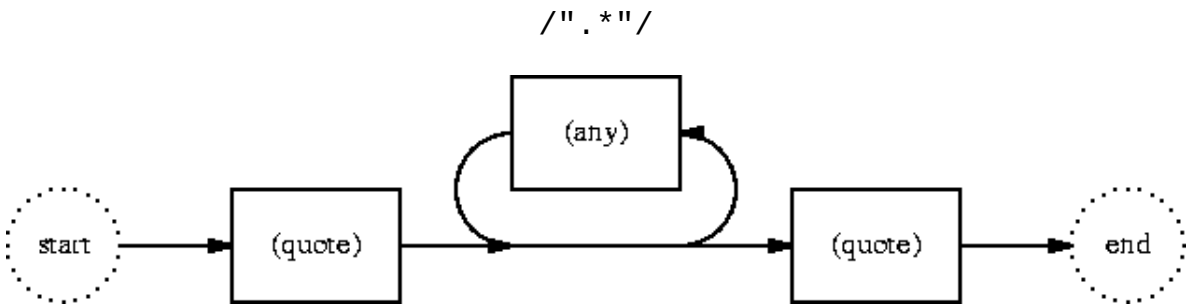


- No backtracking
- Much more efficient (5x or so)
- Use when appropriate

Greed is Good

- ‘‘How do I match a double-quoted string?’’

Wrong



- Why?

```
"Betty","White",143.12,"Hartford","CT",06117
open F, "< $file" or die "Ouchie";
"If I were your husband," he replied, "I should drink it."
```

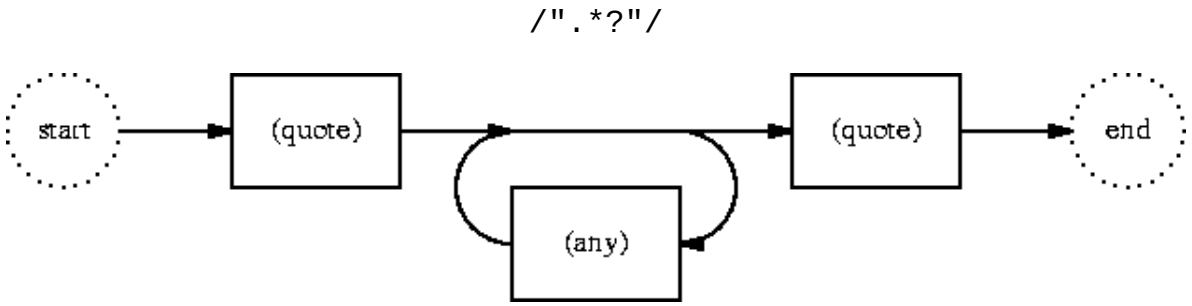
- Probably what was wanted was

```
"Betty","White",143.12,"Hartford","CT",06117
open F, "< $file" or die "Ouchie";
"If I were your husband," he replied, "I should drink it."
```

Greed is Good

- ‘‘How do I match a double-quoted string?’’

The ‘Little Knowledge’ solution

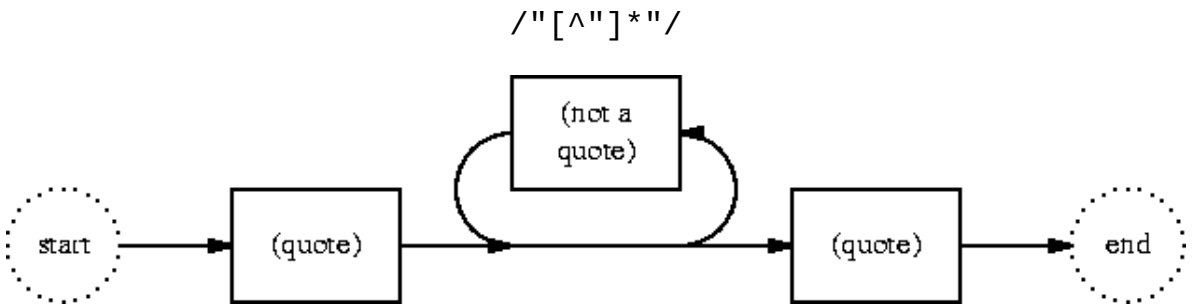


- It works, but in older versions of Perl it was slow
- Why?
- **`"If I were your husband,"`** he replied, "I should drink it."

Greed is Good

- ‘‘How do I match a double-quoted string?’’

The Best Solution

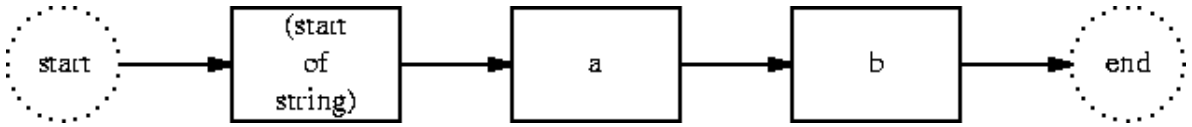


- **`"If I were your husband,"`** he replied, "I should drink it."
- Starting in 5.6.0, `.*` and `.*?` got an optimization
  - As a result, there is no longer much difference between these examples
  - However, the difference still holds for more complicated cases

Anchors

- Beginning anchor ^

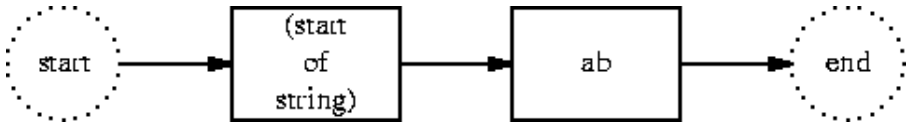
/^ab/



- Attempt to match ^ fails unless cursor is before the first character of the string
- absinthe

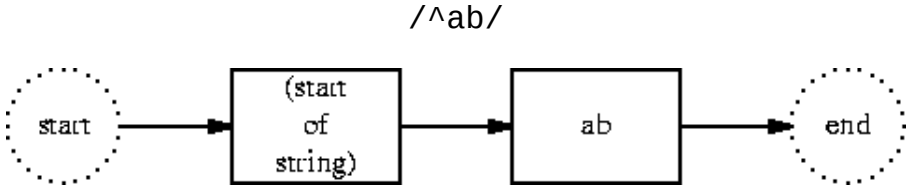
```
START              |a b s i n t h e
^                  |a b s i n t h e       Yes!
a                  |a b s i n t h e       Yes!
b                  <a|b s i n t h e       Yes!
END                <a b>s i n t h e       Yes!
```

- By the way, I've been telling you a little fib up to now
- It really looks like this:




```
START              |a b s i n t h e
^                  |a b s i n t h e       Yes!
ab                 |a b s i n t h e       Yes!
END                <a b>s i n t h e       Yes!
```

Anchors



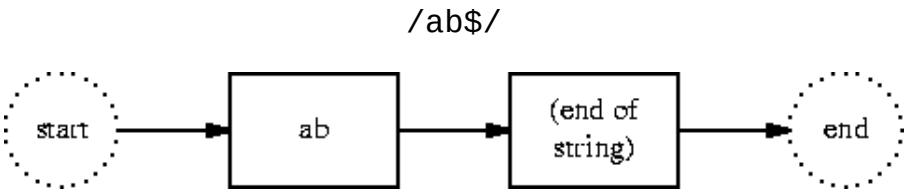
- Attempt to match ^ fails unless cursor is before the first character of the string
- But also, the start node is altered so that the engine can only start at the beginning of the string
- `grab`

```
START          |g r a b
^              |g r a b          Yes!
ab             |g r a b          Nope
```

- Match fails.
- More about optimizations later

Anchors

- Ending anchor `$`



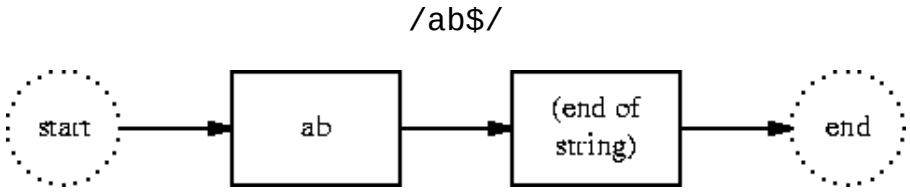
- Attempt to match `$` fails unless cursor is after the last character of the string
- `absinthe`

```
START              |a b s i n t h e
ab                 |a b s i n t h e       Yes!
$                  <a b|s i n t h e       Nope
START               a|b s i n t h e
ab                  a|b s i n t h e       Nope
START               a b|s i n t h e
ab                  a b|s i n t h e       Nope
START               a b s|i n t h e
ab                  a b s|i n t h e       Nope
```

...

- Match fails.
- This simple case is of course optimized
- In general, it really does do it this way

Anchors

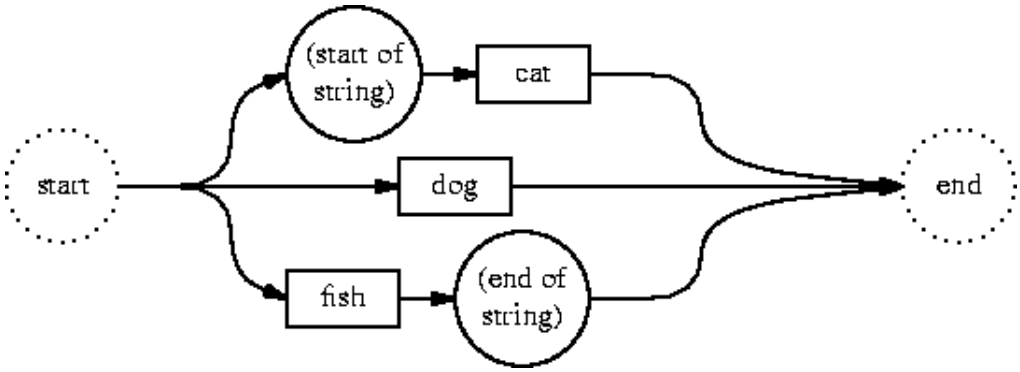


- Attempt to match `$` fails unless cursor is after the last character of the string
- `grab`

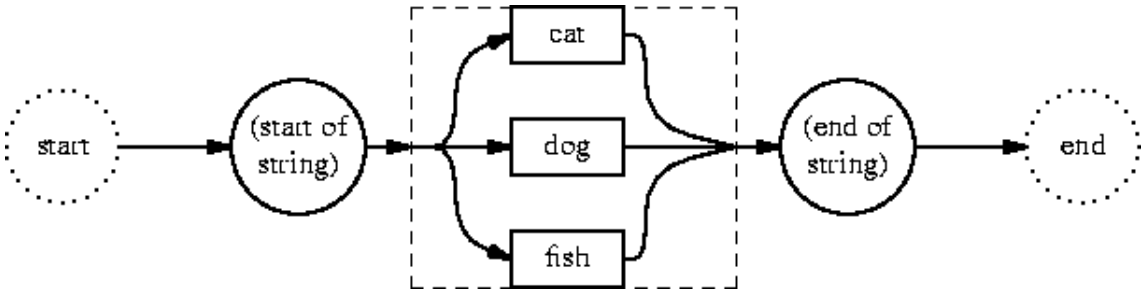
```
START          |g r a b
ab             |g r a b          Nope
START           g|r a b
ab              g|r a b          Nope
START           g r|a b
ab              g r|a b          Yes!
$               g r<a b|         Yes!
END             g r<a b>         Yes!
```

Common Anchor Error

```
/^cat|dog|fish$/
```




```
/^(cat|dog|fish)$/
```



The Rest of the Metacharacters

dot

- . matches any character....
- *except* newline!
- Why not?

```
$time = <STDIN>;                # "11:29\n"
($minutes) = ($time =~ /:(.*)$/);
```

- So that `$minutes` gets `"29"` and not `"29\n"`

The Rest of the Metacharacters

dot

- This brings up a subtlety:

```
$time = <STDIN>;                # "11:29\n"
($minutes) = ($time =~ /:(.*)$/);
```

- If `.` doesn't match `\n`, why does this pattern match succeed?
- The string ends with `\n`, and `.` won't match `\n`.
- Answer: `$` doesn't have to be exactly at the end. It will match at a `\n` that is at the end.

The Rest of the Metacharacters

dot

- To make `.` match anything at all, even `\n`, use the `/s` modifier.

```
$time = <STDIN>;                # "11:29\n"
($minutes) = ($time =~ /:(.*)$/s);
```

- `$minutes` is now `"29\n"` rather than `"29"`
- This might be useful in HTML matching, for example:

```
<p align=center><table align=center border=1><font size="+2">
<tr><td>\d</td><td>[0-9]</td></tr>
<tr><td>\D</td><td>[^0-9]</td></tr>
<tr><td>\w</td><td>[A-Za-z0-9_]</td></tr>
<tr><td>\W</td><td>[^A-Za-z0-9_]</td></tr>
<tr><td>\s</td><td>[ \t\n\f\r]</td></tr>
<tr><td>\S</td><td>[^ \t\n\f\r]</td></tr>
</font></table></p>
```

- `<table[^>]*?>.*</table>` won't match this unless you use `/s`

The Rest of the Metacharacters

- `\d \D \w \W \s \S`
  - These are just character classes.

`\d`	`[0-9]`
`\D`	`[^0-9]`
`\w`	`[A-Za-z0-9_]`
`\W`	`[^A-Za-z0-9_]`
`\s`	`[ \t\n\f\r]`
`\S`	`[^ \t\n\f\r]`

- Actually they depend on the locale, so they’re not only shorter, they’re also safer
- Example: In France, `\w` will match É and ï.
- But `[A-Za-z0-9_]` only includes `E` and `i`.

The Rest of the Metacharacters

```
|D|o|n|'|t| |t|o|u|c|h| |t|h|a|t|!|
```

- `\b`: ('word boundary')
  - It succeeds when the previous character is a `\w` and the next is not (or vice versa)

```
|D o n|'|t| |t o u c h| |t h a t|!
```

- `\B` is the opposite:
  - It succeeds when the previous and next characters are both `\w`, or neither is `\w`

```
D|o|n ' t   t|o|u|c|h   t|h|a|t !|
```

- Neither one will advance the cursor: They are *assertions*.
- Both pretend that string is bounded by `\W` characters.

Lookahead Assertions

- `(?=...)` and `(?!...)` are similar to `\b` and `\B`.
- They look ahead in the string to see if what follows matches . . .
  - If so, they succeed, but don't advance the cursor
- Example: Split an email header into fields:

```
Received: from ni-s.u-net.com ([193.119.182.90] helo=bactrian.ni-s.u-net.com)
        by he101war.uk.vianw.net with esmtp (Exim 3.22 #5)
        id 17H8J0-0005po-00; Sun, 09 Jun 2002 20:24:51 +0100
Content-Disposition: inline
Content-Transfer-Encoding: binary
MIME-Version: 1.0
X-Mailer: Id: //depot/mail/tkmail#119 /Perl5.008 Mail::Internet v1.46
Subject: Re: Standard layers, documentation
In-Reply-To: <20020609191647.GE31617@ool-18b93024.dyn.optonline.net> from
    Michael G Schwern on Sun, 09 Jun 2002 15:16:47 -0400
Content-Type: text/plain; charset="UTF-8"
To: schwern@pobox.com
```

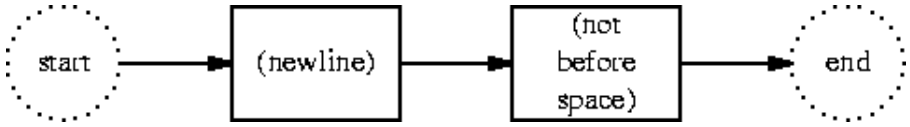
- Wrong: `split /\n/`
  - (Consider the `Received` line for example)
- Also wrong: `split /\n\S/`

Lookahead Assertions

```
Received: from ni-s.u-net.com ([193.119.182.90] helo=bactrian.ni-s.u-net.com)
        by he101war.uk.vianw.net with esmtp (Exim 3.22 #5)
        id 17H8J0-0005po-00; Sun, 09 Jun 2002 20:24:51 +0100
Content-Disposition: inline
```

- Solution:

```
split /\n(?!\s)/
```



- Here's a trick: Make a pattern that never matches:

```
/(?!)/
```

The Rest of the Metacharacters

- `{m,n}` is straightforward now
- It's like `*` but keeps track of the number of matches
- `P{n}` is the same as `P{n,n}`
- Because it keeps track of the number in a small integer, m and n are restricted to be between 0 and 32767.
- There's a non-greedy version `{m,n}?` which is rarely used
- Actually `X*` is implemented with `{m,n}` for nontrivial `X`.
- This means that `^(foo|bar)*$` wouldn't match `"foo" x 35000`.
  - Actually the regex engine would run out of stack and dumps core before that
- Sometime after 5.004_04 and at or before 5.005_02, this was fixed
  - n=32767 now has a special meaning; it is used internally to mean infinity
  - You are no longer allowed to specify 32767 explicitly

Regex Target Variables

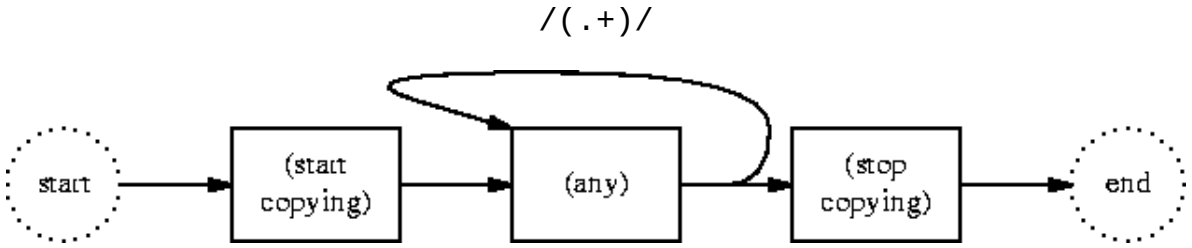
- `$‘`
  - Characters skipped before matching begins
  - (Always empty when `^` is used)
- `$&`
  - Matched string
- `$’`
  - Characters not used after end of match
  - (Always empty when `$` is used)

Regex Target Variables

- `$‘ $& $’`
- If your program never uses these, Perl doesn’t bother to maintain them at run time
- Result: All regexes get faster
- If you use them anywhere, you lose this speed benefit
- Avoid them
- **Never** use them in a module
- Don’t `use English`

Backreferences

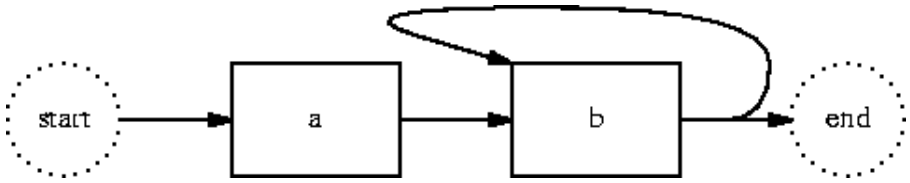
- (and)
- These also cause copying
- They’re slow for the same reason as `$&` etc.
- But they only slow the regexes that use them.
- How do they work?



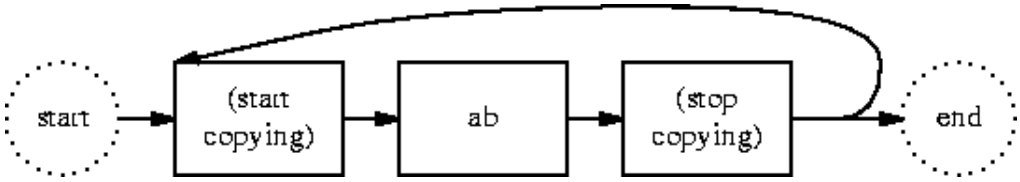
Backreferences

- Occasionally you want the grouping effect of `(...)` without the capturing effect

`/ab+/`

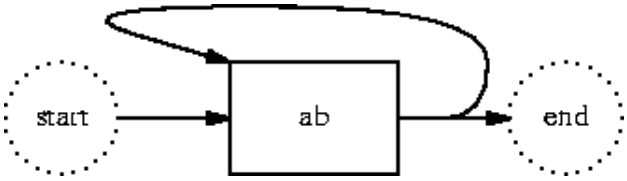


`/(ab)+/`



- Use `(?:...)` instead

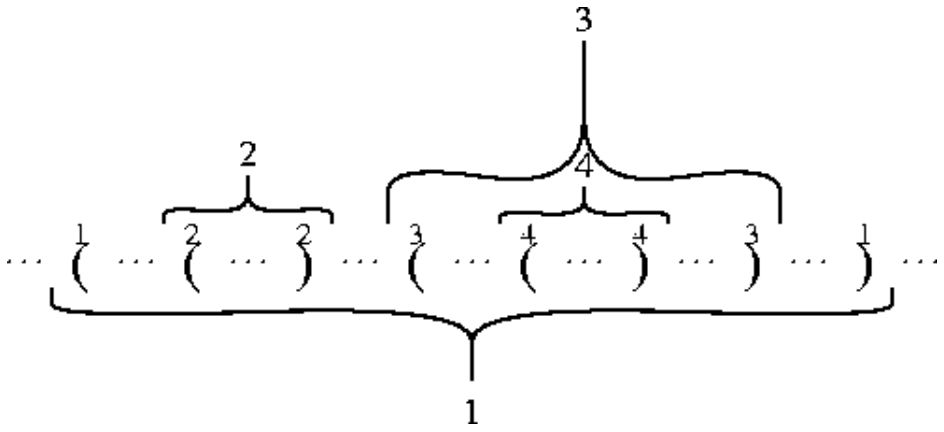
`/(?:ab)+/`



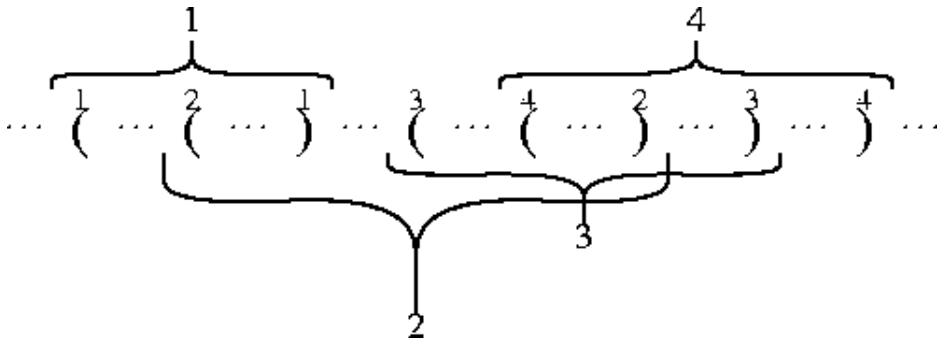
- In Perl 6, this skanky notation will be replaced with `[...]`

Backreferences

Like this:



Not like this:



Backreference Numbering is Lexical

- Consider:

```
# $file is "report.pl" or "/usr/local/bin/report.pl"
($path, $name, $suff) = $file =~ m{(.*/)?(.*)\.(.*)};
```

- When `$file` is `/usr/local/bin/report.pl`

```
/usr/local/bin/  report  .   pl
-----path------  -name-     suff
      $1           $2        $3
```

- But what about when `$file` is `report.pl` and has no path?
- Since the `(.*/)?` is 'skipped', will `(.*)\.(.*)` be `$1` and `$2`?
- No. The parentheses are numbered at *compile time*
  - The value of `$file` cannot affect that

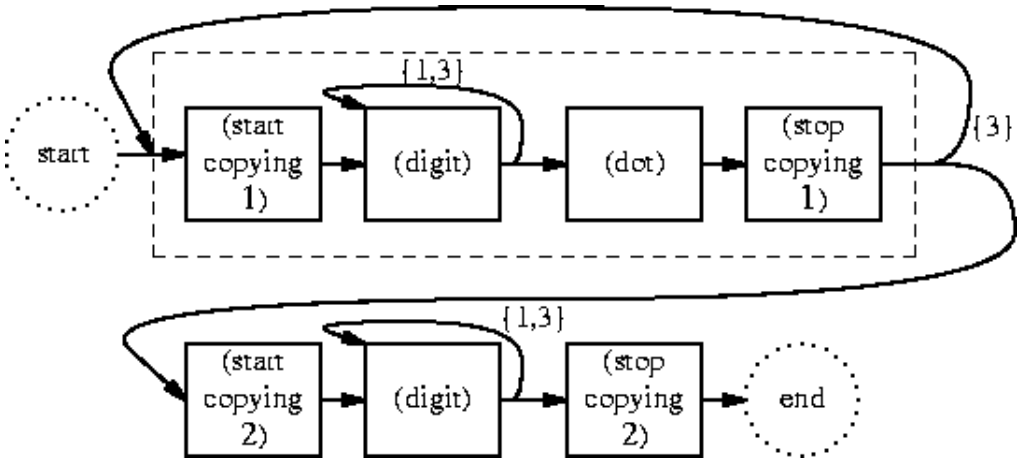
```
                 report  .   pl
      path       -name-     suff
       $1          $2        $3
```

- `$path` here is undefined
- Similarly `/(a+)|(b+)/`
  - If there are any `a`'s, they will be in `$1`
  - If there are `b`'s but no `a`'s, the `b`'s will be in `$2`, and `$1` will be undefined
  - `$1` *always* contains the `a`'s; `$2` *always* contains the `b`'s

Backreferences

```
/(\d{1,3}\.){3}(\d{1,3})/
```

- People sometimes expect this to capture into `$1`, `$2`, `$3`, `$4`, but that's wrong
- It has only two pairs of parentheses, so it captures only `$1` and `$2`
- Why? Isn't the `{3}` supposed to 'repeat three times'?



- What does it do with `130.91.6.1`?

  Start copying, copy the `130` into `$1`, stop copying, repeat
  Start copying, copy the `91` into `$1`, stop copying, repeat
  Start copying, copy the `6` into `$1`, stop copying, go on
  Start copying, copy the `1` into `$2`, stop copying, end of string

- End result: Only `6` is in `$1`.
- Solution: Use `m//g` (coming up) or `split`

Backreferences

- Instead of `$&` etc., use `/^(.*?)(PATTERN)(.*)$/`
  - Then `$1`, `$2`, `$3` instead of `$‘`, `$&`, `$’`
  - Just as slow as `$&` etc., but doesn’t affect *other* regexes
  - Why `(.*?)` here?

Where Do Machines Come From?

- Usually constructed at compile time
- Same machine used repeatedly to match any string
- When regex varies at run time, construction deferred
- `/$PAT/` is *very* slow

Run-Time Construction Disaster

- Beginners like to do this:

```
my @pats = ('fo*', 'ba.', 'w+3');

while (<>) {
  foreach $pat (@pats) {
    print if /$pat/;
  }
}
```

- Regex machine is constructed *each* time through the loop, then discarded
- 1 million lines of input---3 million constructions

Avoiding This Disaster

```
push @pats, qr/$_/ for 'fo*', 'ba.', 'w+3';

while (<>) {
  foreach $pat (@pats) {
    print if /$pat/;
  }
}
```

- Since 5.005, regexes are first-class objects
- `$regex = qr/REGEX/` yields a regex object
- `$string =~ /$regex/` does *not* perform another compilation
- `$string =~ $regex` works also
- This is about 80% faster

Minor Disaster

`grep`

```
my $pat = shift;

while (<>) {
  print if /$pat/;
}
```

- Here the pattern *does not* vary at runtime
- Perl still checks each time to see if it has changed

```
my $pat = shift;

while (<>) {
  print if /$pat/o;
}
```

- `/o` modifier promises that the pattern will *never* change
- Perl no longer needs to check

Another Disaster

```
/^(\w+|::)*$/
```

- Matches Perl identifiers like `Foo` and `Getopt::Std`.
- What does it do with `abcd!`?

```
\w+                 <a  b  c  d> !      No good
\w+ \w+             <a  b  c><d> !      Also no good
\w+ \w+             <a  b><c  d> !      Also no good
\w+ \w+ \w+         <a  b><c><d> !      Still no good
\w+ \w+             <a><b  c  d> !      Also no good
\w+ \w+ \w+         <a><b  c><d> !      Still no good
\w+ \w+ \w+         <a><b><c  d> !      Still no good
\w+ \w+ \w+ \w+     <a><b><c><d> !      Guess what?
Gives up.
```

- This doesn't include all the times it tried to match `::` against one of the letters, or the times it tried making `*` match no times, or...

Disaster Continues

```
/^(\w+|::)*$/
```

- Try

```
perl -Mre=debug -e '"abcd!" =~ /^(?:\w+|::)*$/'
```

- It generates 279 lines of diagnostic output about the backtracking that it tried before it gave up.

```
perl -Mre=debug -e '"abcde!" =~ /^(?:\w+|::)*$/'
```

takes twice as long and generates twice as much: 535 lines. We would expect

```
perl -Mre=debug -e '"what_an_incredible_disaster!" =~ /^(?:\w+|::)*$/'
```

to take about 8,388,608 times as long and to generate 2,147,483,671 lines of output.

- It doesn't take forever, but it's hard to tell the difference.

Avoiding This Disaster

```
/(\w+|::)*/
```

- Nested quantifiers are always risky
- Whenever you write one, make sure you really need it
- To fix this one is easy:

```
/(\w|::)*/
```

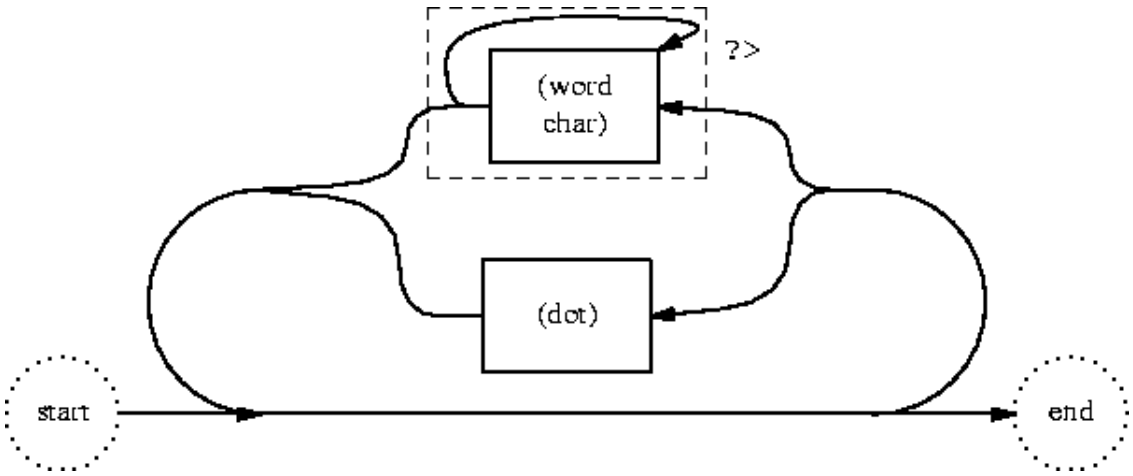
- This is much more efficient --- there aren’t so many things to try.

Avoiding This Disaster

```
/(\w|::)*/
```

- Perhaps a more general solution involves the new `(?>...)` operator:

```
/((?>\w+)|::)*/
```



- State is saved as usual inside the fence
  - But this state is discarded when the node pointer exits the fence
  - State can backtrack *past* the fenced area
  - But not *into* the fenced area
- `\w+` might match many different strings
  - `(?>\w+)` says that only the *first choice* can be correct
  - If the first choice doesn't work, don't try any other choice

Perl 6

- Perl has a (deserved) reputation for having too much punctuation
  - A lot of that reputation is based on Perl's regex syntax
  - But a lot of the regex syntax was inherited from Ken Thompson's original design
  - He used up all the brackets for things like quantification
  - All that was left were things like `(?:...)`
- Perl 6 will completely overhaul its regex syntax
  - Patterns will become much more like BNF grammars
  - They will efficiently incorporate other patterns as sub-parts:

```
rule octet { \d <1,3> }
rule ip_address { <octet> [ \. <octet> ]<3> }
```

- Traditional-style constructions will continue to be supported:

```
$ip_address = /\d<1,3>[\.\d<1,3>]<3>/;
```

- As will the old notation:

```
$ip_address = m:p5/\d{1,3}(?:\.\d{1,3}){3}/;
```

- See `http://www.perl.com/lpt/a/2002/06/04/apo5.html` for the fascinating details

Strings that contain newlines: `/s` and `/m`

- `/s`: Make `.` match newline (normally it doesn't)
- `/m`: Make `^` match at beginning of line (after a newline) rather than beginning of string, and similarly `$`
  - Example: Suppose `$message` contains an entire mail message.

```
($subject) = ($message =~ /^(Subject:\s+(.|\n\s)*)$/m);
```

- Extracts `Subject` field.
- If you use `/m`, use `\A` and `\Z` to get the old meanings of `^` and `$`: Match at beginning or end of string only
- Recall that `$` normally matches before a newline at the end of the string. `\Z` does that too.
- If you *really* want to match *only* at the end of the string, use `\z`
- Perl 6 will fix this mess; `/s` and `/m` are going away:

```
^       beginning of string
$       end of string
^^      beginning of line
$$      end of line
.       match any single character
\n      match a newline
\N      match any single character except a newline
```

Repeated Matching: `/g`

- `/g` means to do the match repeatedly
  - with `s///g`, replace all occurrences (non-overlapping)
  - with `m//g`, find all matches, starting each where the previous one finished
- `m//g` in list context returns a list of all matching strings:

```
"Madagascar" =~ m/a./g;     # returns ('ad', 'ag', 'as', 'ar')
```

- Extract all the numerals from a string:

```
"12-345:6 78" =~ m/\d+/g;   # returns ('12', '345', '6', '78')
```

- Note that this does *not* return `2` or `34` or `45`
  - Each `m//g` picks up where the previous match *ended*
- Split a string into fixed-length substrings:

```
@substrings = "abcdefghijklmnopqrstuvwxyz" =~ /.{1,5}/g;
# Yields ('abcde', 'fghij, 'klmno', 'pqrst', 'uvwxy', 'z')
```

- Notice importance of greed here - what if we had used `.{1,5}?` ?
- To omit `z`, use `.{5}` instead of `.{1,5}`

Randal's Rule

- Randal Schwartz (author of *Learning Perl*) says:

Use capturing or `m//g` when you know what you want to **keep**.

Use `split` when you know what you want to **throw away**.

Repeated Matching: `/g`

- In scalar context, `/g` turns the matcher into an iterator

```
while ("I like pie" =~ /\w+/g) {
  print "<$&>\n";
}

<I>
<like>
<pie>
```

- Each scalar has a *current position*
- `/g` starts from the current position and sets it afterwards
- You can get and set the current position with the `pos` function:

```
my $s = "I like pie";
for ($i = 0; $i < length($s); $i += 2) {
  pos($s) = $i;
  $s =~ /\w*/g;
  print "<$&>\n";
}

<I>
<like>
<ke>
<>
<ie>
```

- A failed match on a string resets its `pos`

Extended Format: `/x`

- `/x` lets you write regexes more readably.
  - White space is ignored. (Use `\s`)
  - #-style comments are allowed
- Extended and very practical example coming up later...

- `Caution`: Unescaped `/` will still terminate the regex, even if it's in a comment!

```
$x =~ /\d+      # numerator
       $FRAC    # FRAC matches either a / or a : symbol
       \d+      # denominator
      /x;
```

- Perl sees the `/` in the 'comment' before it sees the `/x`
  - It thinks that the `/` ends the regex
  - Confusion ensues
  - Perl 6 will fix this: modifiers precede the pattern instead of following it

Tokenizing

- *Tokens* are the basic syntactically meaningful portions of an input.
- For example, in

print 12+3;

- The tokens are `print`, `12`, `+`, `3`, and `;`
- Individual characters are not generally meaningful.
- *Tokenizing* is the act of converting a character stream into a token stream.
- Also called *lexing*

Tokenizing

- In C, you use programs like `lex` to convert a description of the legal tokens into a tokenizer program.
- Or you write a program to read the input character-by-character and run a state machine
- That is not very Perl-like.
- It is also not very efficient.

Tokenizing

- A regex is *already* a program for reading data character-by-character and running a state machine
- Let's write a lexer for a calculator. It has the following tokens:
  - `+`, `-`, `*`, `/`, `^`, `**`, `(`, `)`, `=`
  - `:=`
  - Variable names: `Value2`, for example
  - Numbers with optional decimal points and scientific notation
  - Whitespace will be ignored except where it separates tokens

Tokenizing

- Our trick:

```
split /(a+)/, $string
```

- This breaks `$string` into pieces which alternate between
  - Strings of `a`'s
  - The other stuff that was between the `a`'s
- Note special `split` meaning of `(capturing parentheses)`.

Tokenizing

- The tokenizer:

```
sub tokens {
  my @tokens =
    split m{(
             \*\* | :=  # ** or := operator
             |
             [-+*/^()=]  # some other operator
             |
             [A-Za-z]\w* # Identifier
             |
             \d*\.\d+(?:[Ee]\d+)?  # Decimal number
             |
             \d+         # Integer
            )}x, shift();
  grep /\S/, @tokens;
}
```

- Easy to understand and to change, efficient, predictable.
- Behaves very much like similar `lex`-generated parsers
- This is why we need `/x`:

```
split
  m{(\*\*|:=|[-+*/^()=]|[A-Za-z]\w*|\d*\.\d+(?:[Ee]\d+)?|\d+)},
  shift();
```

- Note that the order of the `|` alternatives is important
  - Is `**` one token or two? What about `12.23`?

Tokenizing

A different version of the same thing:

```
my $s;
sub set_string {
    $s = shift;
}

sub next_token {
  return POWER        if $s =~ /\G\*\*/gc;
  return ASSIGNMENT   if $s =~ /\G:=/gc;
  return "OP $1"      if $s =~ /\G([-+*/^()=])/gc;
  return IDENT        if $s =~ /\G[A-Za-z]\w*/gc;
  return FLOAT        if $s =~ /\G\d*\.\d+(?:[Ee]\d+)?/gc;
  return INT          if $s =~ /\G\d+/gc;
  return next_token() if $s =~ /\G\s+/gc;
  return BAD_CHAR     if $s =~ /\G./gc;
}
```

- This uses the `/gc` modifier with `\G`
- `\G` anchors the match to occur *at* the current `pos()`
  - Rather than somewhere to the right of it as usual
- Normally, the `pos()` is discarded if the match fails
  - `/c` disables this misfeature

Optimizations

- Common cases are heavily optimized
- `/literal/` doesn't use the regex engine
  - Instead, it does a Boyer-Moore search
- `/^PAT/` never advances the cursor
- `/PAT$/` starts at the correct place if the length of the result is known
- If the target string is too short, the regex engine is never invoked
  - `/(fish|dog){7,12}\s+/` cannot match any string shorter than 22 characters
- When in doubt, benchmark!
- `-Mre=debug` is helpful here also
- The `/i` modifier makes the match case-insensitive
  - It tends to **disable** optimizations
  - Use it sparingly

Optimizations

- Since 5.6, Perl has had a very clever *floating-anchored* search
- It tries to locate two long strings which *must* be in the target
- It searches for these first, then works inward
- For example, in

```
"----B----A---" =~ /A-*B/
```

- Perl looks first for `A`, then for `B`
- It figures out that there's only one `A`
- There's no consistent choice for `B`, so it fails immediately
- No backtracking search

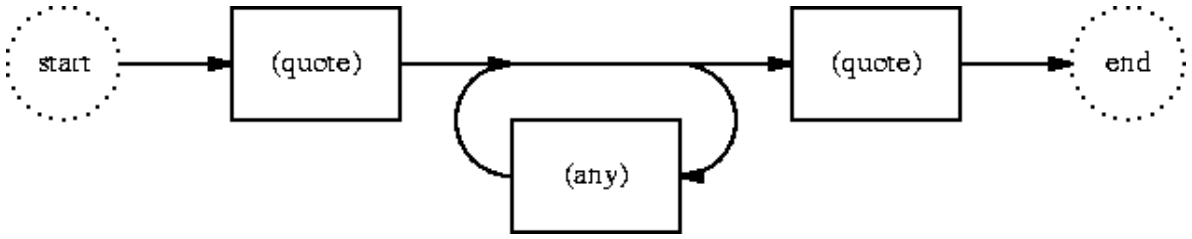
```
"----A----B---" =~ /A-*B/
```

- Here Perl locates the `A` immediately, and skips the preceding characters
- For fullest details, see `perldebguts` and `perl -Mre=debug` output
- `/i` disables this --- avoid it

Optimizations

- Since 5.6, Perl has treated `.*`, `.+`, `.*?`, and `.+?` specially
  - When they are followed by some literal string...
  - ...the engine is smarter about how many repetitions might work
  - As a result, this example is no longer slow:

`/".*?"/`



More New Metacharacters

- These all appeared in 5.005
- `(?{CODE})` embeds arbitrary code into a regex
- The code is executed when the node pointer passes through it
- It matches the empty string and always succeeds
- `(?(CONDITION)YES|NO)` evaluates the condition
  - If true, try to match `YES`, else `NO`
  - omit `NO`, it defaults to nothing
  - `CONDITION` can be a `(?{CODE})` expression
- Example: Match strings where `(...)` are balanced
  - (The holy grail of regular expressions.)

Matching Strings with Balanced Parentheses

- How does a human decide that `((I)(like(pie))!)` is balanced?

```
( ( I ) ( l i k e ( p i e ) ) ! )
1 2   1 2         3       2 1   0
```

- That's what we'll do:

```
^
(?{ local $d=0 }) # Set depth to 0
(?:
    \(            # When you see an open parenthesis...
    (?{$d++})     #  ...increment the depth
 |
    \)            # or you could see a close parenthesis...
    (?{$d--})     #  ...in which case decrement the depth...
    (?            #  ...and check...
      (?{$d<0})   #  ...if there was no matching open paren...
      (?!)        #  ...then fail.
    )
 |
    (?> [^()]* )  # or you could see some non-parenthesis text
                  # (but don't bother backtracking into it)
)*
                  # After you match as much as possible...
(?                # ...check to see if...
  (?{$d!=0})      # ...there were unmatched open parentheses...
  (?!)            # ...if so then fail.
)
$
```

- `/x` was essential here:

```
^(?{local$d=0})(?:\((?{$d++})|\)(?{$d--})(?(?{$d<0})(?!))|(?>[^()]*))*(?(?{$d!=0})(?!))$
```

- Similarly: Recognize palindromes:

```
/^(.*).?(?>(.*))(?(?{$1 ne reverse $2})(?!))/
```


Thanks!

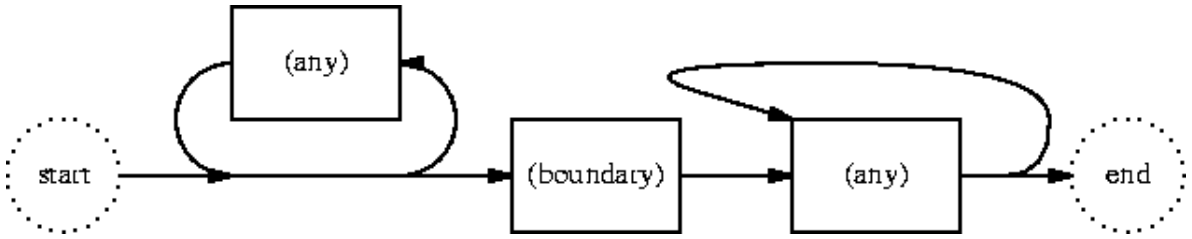
- More information:
  - *Mastering Regular Expressions* (Jeffrey E. F. Friedl; O'Reilly & Associates)
    - A new and wonderful second edition was released in July 2002
  - `perlre` manual page (reference and definitions)
    - `http://www.perldoc.com/perl5.8.0/pod/perlre.html`
  - `perlop` manual page (examples; details of `s///` and `m//` and their modifiers)
    - `http://www.perldoc.com/perl5.8.0/pod/perlop.html`
  - `perlretut` and `perlrequick` tutorials (new in 5.6.1)
    - `http://www.perldoc.com/perl5.8.0/pod/perlretut.html`
    - `http://www.perldoc.com/perl5.8.0/pod/perlrequick.html`
  - `perlfaq6` - frequently asked questions
    - `http://www.perldoc.com/perl5.8.0/pod/perlfaq6.html`
  - *Perl Cookbook* (Christiansen and Torkington; O'Reilly & Associates)
    - Chapter 6 especially
  - Apocalypse 5: Regexes in Perl 6
    - `http://www.perl.com/pub/a/2002/06/04/apo5.html`

Residue of the Regexes

- These talks evolve over time
- Old slides move out, new ones come in
- You might as well see the slides that were dropped

Word Boundary Assertion

```
/.*\b.+/
```



- What ho?

```
START               |W h a t . h o ?
.                   |W h a t . h o ?          Yes!
*                   <W|h a t . h o ?
 .                  <W|h a t . h o ?          Yes!
 *                  <W h|a t . h o ?
```

...

```
                .   <W h a t . h|o ?          Yes!
               *    <W h a t . h o|?
                .   <W h a t . h o|?          Yes!
                *   <W h a t . h o ?|
                 .  <W h a t . h o ?|         Nope
                 \b <W h a t . h o ?|         Nope
                \b  <W h a t . h o|?          Yes!
                .   <W h a t . h o|?          Yes!
                +   <W h a t . h o ?|
                 .  <W h a t . h o ?|         Nope
                 END<W h a t . h o ?>         Yes!
```

- Maybe it's a little surprising that the word boundary it found was the one in the o?

New Features: POSIX and Unicode Character Classes

- `[:space:]` matches a whitespace character
- Anything that would test true with the C `isspace` function
- `\P{IsSpace}` matches any Unicode character that possesses the `IsSpace` property
- This is new in 5.6.0.

Upcoming Enhancements?

- ‘Onion rings’
- Match occurrences of `PATTERN2` but only when it occurs inside something that also matches `PATTERN1`
- For example:

```
(?<> <[^>]*>       # Inside an HTML tag expression...
     \w+ = \w+  # Match an attribute=value pair
)x              # But otherwise attribute=value is not allowed.
```

- This might change before it actually puts in an appearance.
- Didn’t get into 5.8; maybe 5.10?

Tokenizing

- We can get rid of that `grep`:

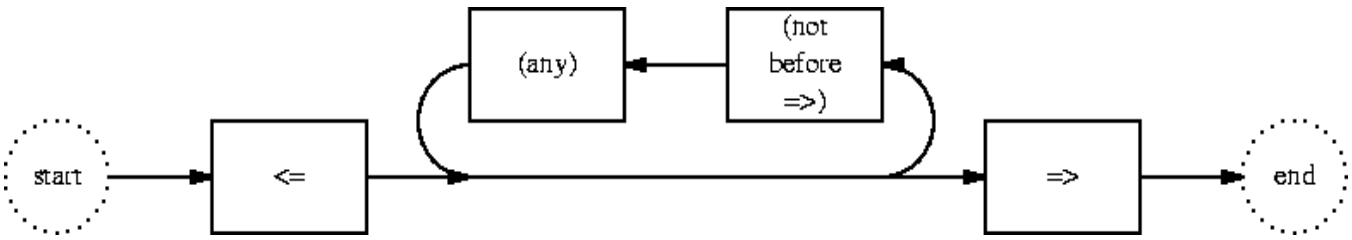
```
sub tokens {
    split m{(
              \*\* | :=  # ** or := operator
             |
              [-+*/^()=]  # some other operator
             |
              [A-Za-z]\w* # Identifier
             |
              \d*\.\d+(?:[Ee]\d+)?  # Decimal number
             |
              \d+         # Integer
             )
             |
              \s+
            }x, shift();
}
```

- (Thanks to Andy Wardley.)

Lookahead Assertions

- `(?=...)` and `(?!...)` are similar to `\b` and `\B`.
- They look ahead in the string to see if what follows matches `...`
- If so, they succeed, but don't advance the cursor
- Example: Match everything from <= up to next =>
  - Wrong: `<=.*=>`
  - (Consider `<=foo bar => baz =>`)
- Solution:

```
/<=((?!=>).)*)=>/
```



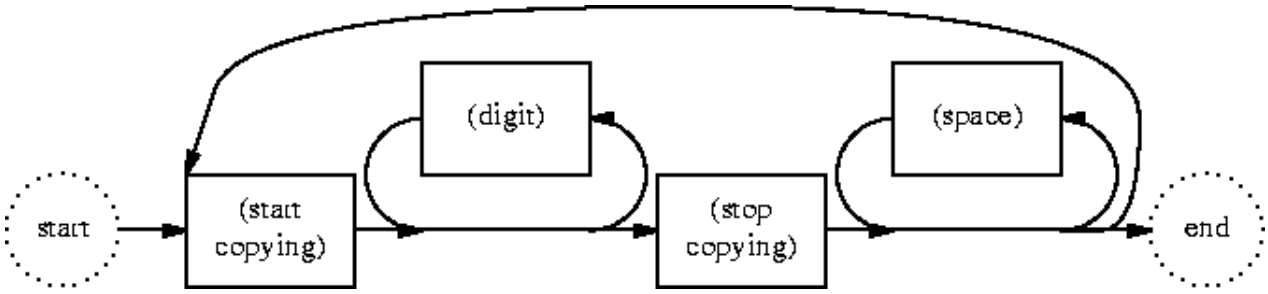
- Here's a trick: Make a pattern that never matches:

```
/(?!)/
```

Backreferences

```
/(?:(\d*)\s*)+/
```

- Here's a FAQ:
- If you try to match `12 34 56`, only the `56` goes into `$1`.
- Why? Isn't the + supposed to 'repeat'?



- What does it do?
  - Start copying, copy the `12` into `$1`, stop copying, repeat
  - Start copying, copy the `34` into `$1`, stop copying, repeat
  - Start copying, copy the `56` into `$1`, stop copying, end of string
- End result: Only `56` is in `$1`.
- Solutions:
  - Use `split`
  - Use `m//g` (coming up)

Randal's Rule

For example:

```
Newsgroups: comp.lang.perl.moderated
Subject: perl question
Date: Tue, 04 Feb 2003 21:52:02 GMT

I have a perl question, I have this as

$string = ((!TM)*A)|(TM*((((!TASEL)*TAA)|((TASEL)*TAB))) ;

I want this to be seperated as TM, A, TM, TASEL, TAA, TASEL,
TAB. How do i do it ?

Thanks in advance ?
perluser99
```

- Once I figured out what the question was, the answer was just

```
@parts = $string =~ m/[A-Z]+/g;
```

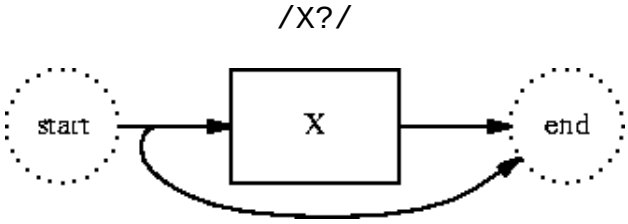

Digression for a Practical Application

- Let’s apply what we know
- Someone showed up on IRC asking this today:
- ‘‘How do I remove the characters from the last `x` to the end?’’

```
s/x.*//;                # WRONG
s/(.*)x.*/$1/;          # Right, but slow
s/x[^x]*//;             # WRONG
s/x[^x]*$//;            # Ahhhh.  (1/3 faster)
```

- End of digression

Option



- Also there's a non-greedy version `X??`
- I used to pay US$60 for a live sighting of `??` in the wild
- But one day I thought of

```
if ($option =~ /^-f(i(e(ld??)??)??)??$/) {
  ...
}
```