

Perl 5.10



Yves Orton / Paul Fenwick

唐鳳



perlhist

perlhist

 **5.0: 1994**


perlhist

 **5.0: 1994**

 **5.1: 1995**

perlhist

 **5.0: 1994**


 **5.1: 1995**

 **5.2: 1996**

perlhist

 **5.0: 1994**

 **5.1: 1995**


 **5.2: 1996**


 **5.3: 1996**

perlhist

 **5.0: 1994**

 **5.4: 1997**

 **5.1: 1995**


 **5.2: 1996**


 **5.3: 1996**


perlhist

 **5.0: 1994**

 **5.4: 1997**

 **5.1: 1995**

 **5.5: 1998**


 **5.2: 1996**


 **5.3: 1996**


perlhist


 **5.0: 1994**

 **5.4: 1997**

 **5.1: 1995**

 **5.5: 1998**

 **5.2: 1996**


 **5.6: 2000**


 **5.3: 1996**


perlhist

 **5.0: 1994**

 **5.4: 1997**

 **5.1: 1995**

 **5.5: 1998**

 **5.2: 1996**

 **5.6: 2000**

 **5.3: 1996**


 **5.8: 2002**

5.8.0: 2002

5.8.0: 2002


 **5.8.1: 2003**

5.8.0: 2002

 **5.8.1: 2003**

 **5.8.2: 2003**

5.8.0: 2002

 **5.8.1: 2003**


 **5.8.2: 2003**


 **5.8.3: 2004**

5.8.0: 2002

 **5.8.1: 2003**


 **5.8.2: 2003**

 **5.8.3: 2004**

 **5.8.4: 2004**


5.8.0: 2002

 **5.8.1: 2003**

 **5.8.5: 2004**


 **5.8.2: 2003**

 **5.8.3: 2004**

 **5.8.4: 2004**

5.8.0: 2002

 **5.8.1: 2003**

 **5.8.5: 2004**


 **5.8.2: 2003**


 **5.8.6: 2004**


 **5.8.3: 2004**

 **5.8.4: 2004**

5.8.0: 2002

 **5.8.1: 2003**


 **5.8.5: 2004**

 **5.8.2: 2003**

 **5.8.6: 2004**


 **5.8.3: 2004**

 **5.8.7: 2005**

 **5.8.4: 2004**

5.8.0: 2002

 **5.8.1: 2003**


 **5.8.5: 2004**

 **5.8.2: 2003**

 **5.8.6: 2004**

 **5.8.3: 2004**

 **5.8.7: 2005**

 **5.8.4: 2004**

 **5.8.8: 2006**

2007

2007

5.8.9



2007

5.8.9



6.0.0



2007

5.8.9



Perl 5.10



6.0.0



5.10: 繼往開來

5.10: 繼往開來




 承襲 5.8 系列的穩定性

5.10: 繼往開來

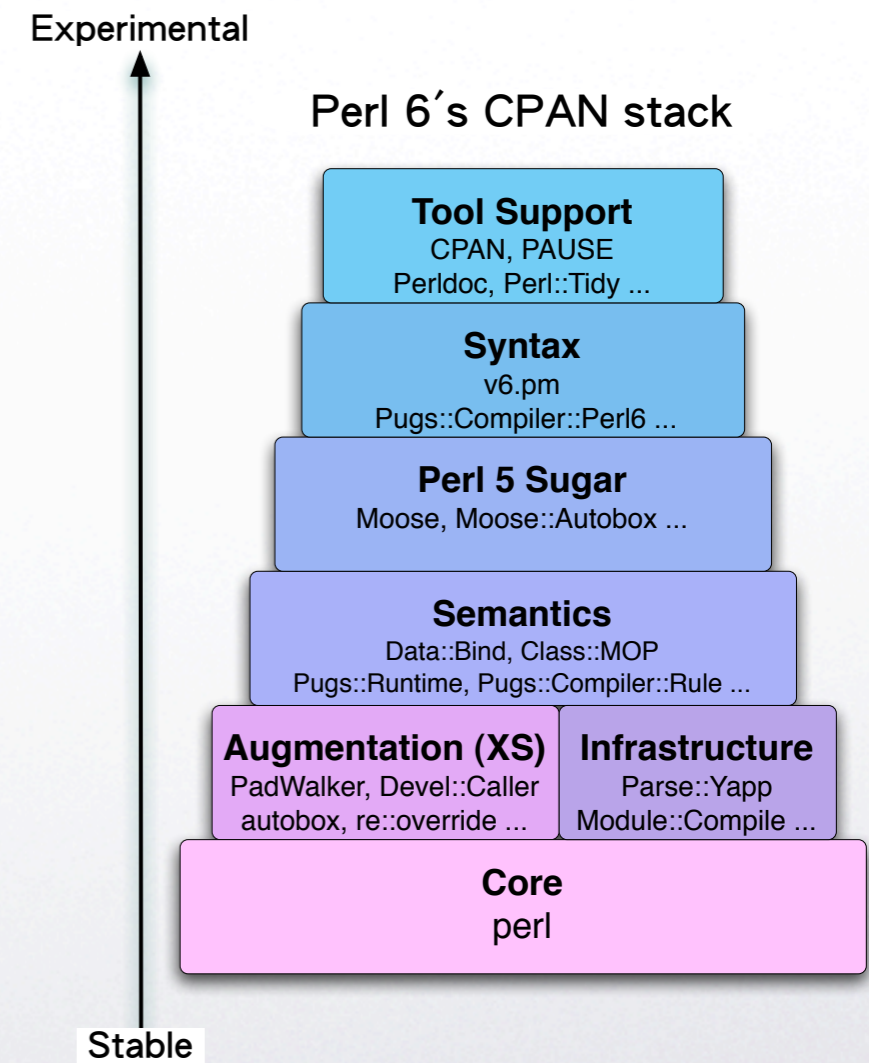
 承襲 5.8 系列的穩定性

 加入 Perl 6 的新功能

5.10: 繼往開來

-  承襲 5.8 系列的穩定性
-  加入 Perl 6 的新功能
-  大幅提昇執行效率

6-on-5 計劃



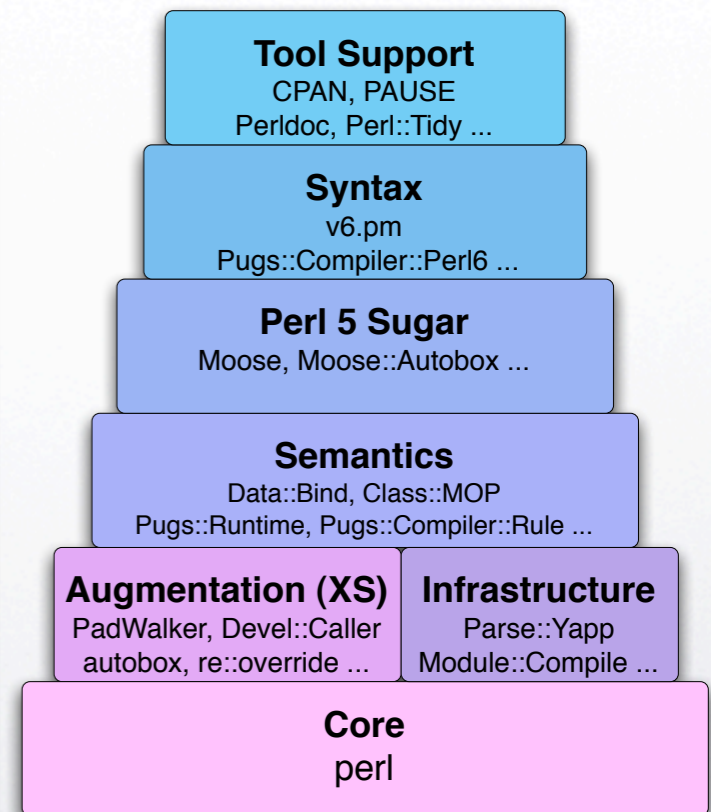
6-on-5 計劃



擴充 P5VM 的功能

Experimental

Perl 6's CPAN stack



Stable

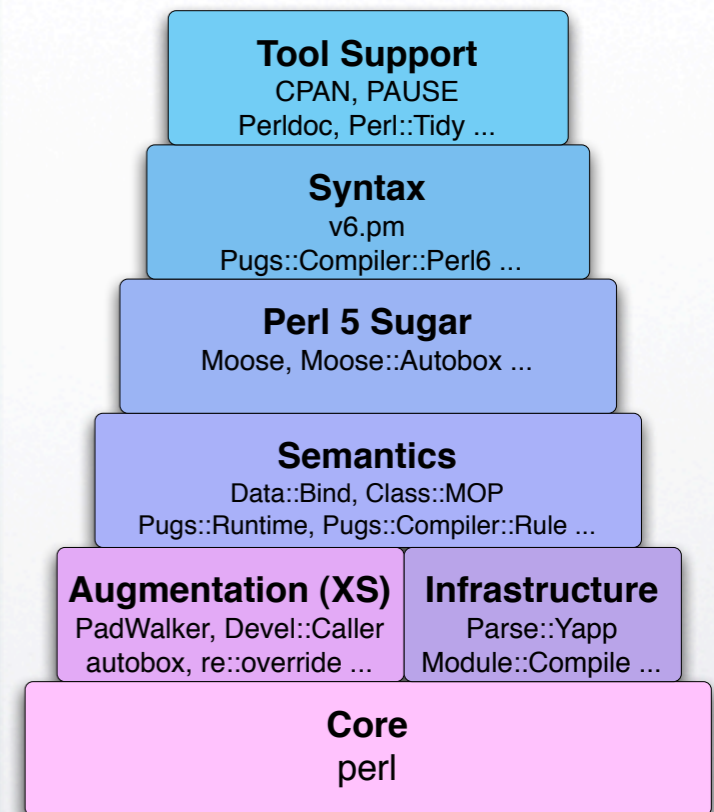
6-on-5 計劃

⌚ 擴充 P5VM 的功能

⌚ 將 Perl6 編譯成 Perl5

Experimental

Perl 6's CPAN stack



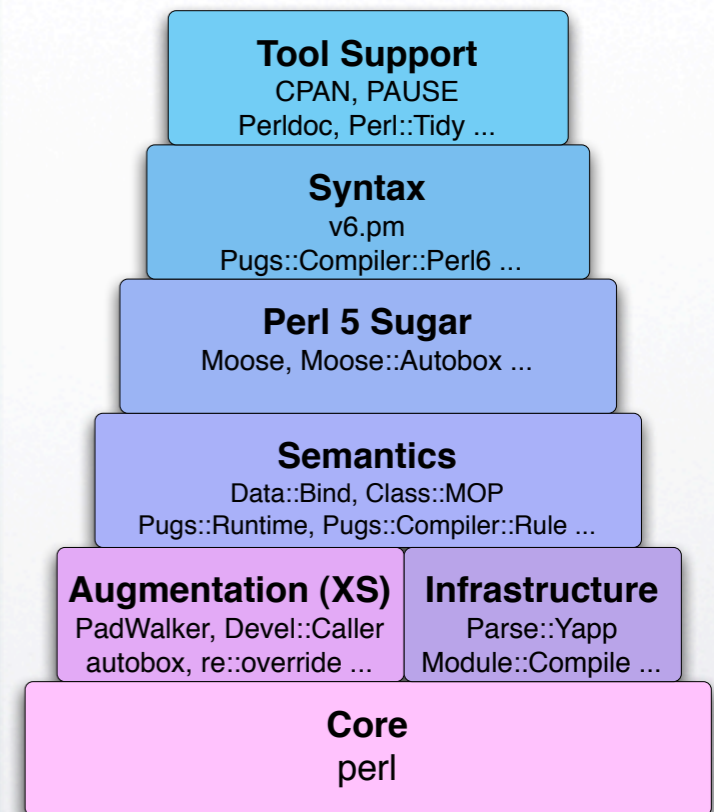
Stable

6-on-5 計劃

- ⌚ 擴充 P5VM 的功能
- ⌚ 將 Perl6 編譯成 Perl5
- ⌚ 物件模型: `Moose.pm`

Experimental

Perl 6's CPAN stack



Stable

6-on-5 計劃

Wednesday April 04, 2007

04:24 AM **Moose is turning 1yr old soon** [[4 Comments](#) | [#32891](#)]

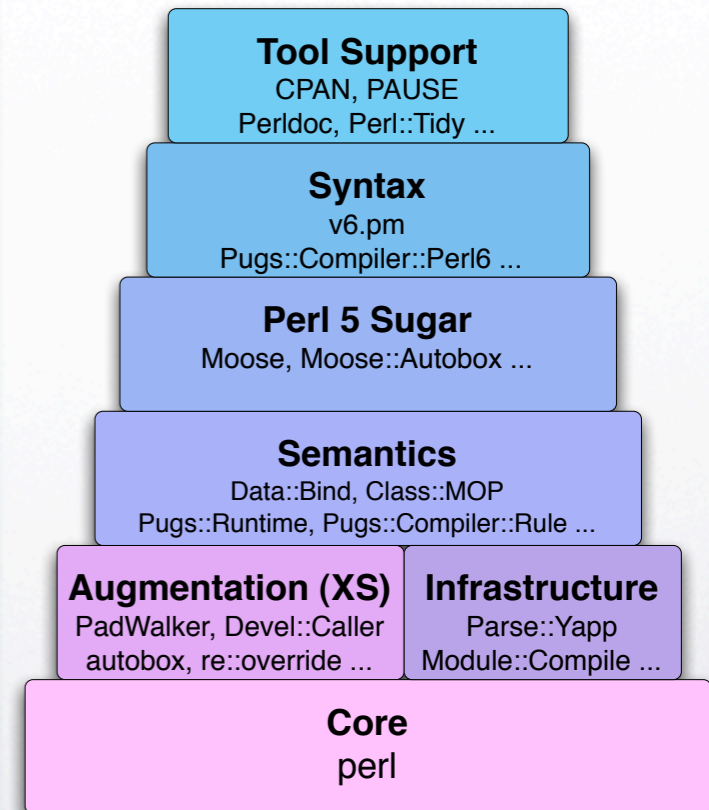
About [a year ago](#) I released the first version of [Moose](#) to CPAN, and since then [we](#) have used it extensively in a number of projects. And now with 3 apps which have been in production for between 6 and 9 months without issue, and positive reports from several other developers who have had similar experiences, I am happy to announce that [Moose](#) is pretty much ready for general use (i.e. - not scary anymore).



物件模型： Moose.pm

Experimental

Perl 6's CPAN stack



Stable



use feature;

use feature;

★ 過去: 沒人敢加新保留字

use feature;

★ 過去: 沒人敢加新保留字

★ 八年來祇多了 **our** 和 **CHECK**

use feature;

- ★ 過去: 沒人敢加新保留字
- ★ 八年來祇多了 **our** 和 **CHECK**
- ★ 現在: 自行挑選新功能匯入

```
{  
    use feature 'say'; # 區塊範圍內生效  
    say "Hello, World!";  
}
```

```
use feature ':5.10'; # 啟用所有 5.10 版的新保留字  
                    # (當然全都是從 Perl 6 搬來的)
```

自動啟用新保留字

自動啟用新保留字

★ 命令列：`perl -E "..."`

自動啟用新保留字

★ 命令列：`perl -E "..."`

★ 程式內：`use v5.10;`

↳ `use feature ':5.10';`

use feature 'say' ;

```
use feature 'say';
```

★ `print` 再加上換列符號

```
use feature 'say' ;
```

★ `print` 再加上換列符號

★ `sub say { print @_, "\n" }`

`use feature 'say' ;`

★ `print` 再加上換列符號

★ `sub say { print @_, "\n" }`

★ 既省力又不會出錯

```
use 5.10;
```

```
print "Hello World\n";  
say "Hello World";
```

省下 4 個鍵

```
print some_function(), "\n";  
say some_function();
```

省下 8 個鍵

```
say foo();  
print foo(), "\n";  
print foo()."\n";
```

串列語境

串列語境

不小心用了純量語境!

```
print 'Hello World\n';  
print "Hello World/n";  
say 'Hello World';
```

引號打錯了

斜線打錯了

不可能出錯 ♡

use feature 'state';

use feature 'state';

★ state \$x; # 宣告靜態變數

```
use feature 'state';
```

★ `state $x;` # 宣告靜態變數

★ 下次進入區塊時，值不會清空

use feature 'state';

★ **state** \$x; # 宣告靜態變數

★ 下次進入區塊時，值不會清空

★ 相當於 C 語言裡的 **static**

舊的寫法

```
{  
    my $i = 0;  
    sub increment {  
        return ++$i;  
    }  
}
```

舊的寫法

```
{  
    my $i = 0;  
    sub increment {  
        return ++$i;  
    }  
}
```

新的寫法

```
use 5.10;  
sub increment {  
    state $i = 0;  
    return ++$i;  
}
```

舊的寫法

```
{  
  my $i = 0;  
  sub increment {  
    return ++$i;  
  }  
}
```

新的寫法

```
use 5.10;  
sub increment {  
  state $i = 0;  
  return ++$i;  
}
```

```
for my $x (...) {  
  for my $y (...) {  
    state %seen; # 不必提到迴圈最外層了!  
    next if $seen{$x}{$y}++;  
    ...;  
  }  
}
```

use feature 'switch' ;

use feature 'switch' ;

★ 啟用 **given/when** 保留字

use feature 'switch' ;

★ 啟用 **given/when** 保留字

★ 相當於 C 語言的 **switch/case**

use feature 'switch';

★ 啟用 **given/when** 保留字

★ 相當於 C 語言的 **switch/case**

★ 支援 **break/continue/default**

use feature 'switch' ;

- ★ 啟用 **given/when** 保留字
- ★ 相當於 C 語言的 **switch/case**
- ★ 支援 **break/continue/default**
- ★ 強大的「智慧型比對」功能

```
# 猜數字遊戲
```

```
use 5.10;
```

```
my $num = int(rand 100);
```

```
# 謎底
```

```
my @guessed;
```

```
# 猜過的數字
```

```
while (my $guess = <STDIN>) {
```

```
  chomp $guess;
```

```
  given ($guess) {
```

```
    when (/^\d/) { say "請輸入正整數" }
```

```
    when (@guessed) { say "您已經猜過這個數字了" }
```

```
    when ($num) { say "猜中了!"; last }
```

```
    when ($_ < $num) { say "再高一點"; continue }
```

```
    when ($_ > $num) { say "再低一點"; continue }
```

```
    push @guessed, $_;
```

```
  }
```

```
}
```

智慧型比對

智慧型比對

★ Perl 5.10 內建 `~~` 算符

智慧型比對

- ★ Perl 5.10 內建 `~~` 算符
- ★ 沿用 Perl 6 裡的定義

智慧型比對

- ★ Perl 5.10 內建 `~~` 算符
- ★ 沿用 Perl 6 裡的定義
- ★ 毋需 `use feature` 即可使用

```
use 5.10;
if ($x ~~ @array)      { say "$x 在陣列裡"      }
if ($x ~~ /match/)    { say "字串符合樣式"      }
if (@x ~~ /match/)    { say "陣列符合樣式"      }
if ($key ~~ %hash)    { say "$key 是雜湊鍵"      }
if (&func ~~ $arg)    { say 'func($arg) 為真'    }
```

```
use 5.10;
if (@array ~~ $x)      { say "$x 在陣列裡"      }
if (/match/  ~~ $x)    { say "字串符合樣式"      }
if (/match/  ~~ @x)    { say "陣列符合樣式"      }
if (%hash  ~~ $key)    { say "$key 是雜湊鍵"      }
if ($arg  ~~ \&func)  { say 'func($arg) 為真'      }
```

use feature 'err';

```
use feature 'err';
```

```
★ $file = param('file')  
    or die "請輸入檔名";
```

use feature 'err';

★ `$file = param('file')`
`or die "請輸入檔名";`

★ 檔名是 `"0"` 怎麼辦?

use feature 'err';

★ `$file = param('file')`
`or die "請輸入檔名";`

★ 檔名是 "0" 怎麼辦?

★ `$file = param('file')`
`err die "請輸入檔名";`

|| 陷阱

|| 陷阱

★ ($x \parallel y$)

→ ($x ? x : y$)

|| 陷阱

★ $(\$x \ || \ \$y)$

↳ $(\$x \ ? \ \$x \ : \ \$y)$

★ 空字符串、 \emptyset 、**undef** 都是假值

|| 陷阱

★ $(\$x \ || \ \$y)$

↳ $(\$x \ ? \ \$x \ : \ \$y)$

★ 空字串、 \emptyset 、**undef** 都是假值

★ 一不小心就會出錯

defined-or

defined-or

★ Perl 5.10 內建 // 算符

defined-or

- ★ Perl 5.10 內建 // 算符
- ★ 毋需 use feature 即可使用

defined-or

- ★ Perl 5.10 內建 // 算符
- ★ 毋需 use feature 即可使用
- ★ $(\$x // \$y)$
↳ **defined**(\$x) ? \$x : \$y

舊的寫法（好孩子不要學）

```
my %bugs_in;
```

```
while (my $module = <>) {  
    $bugs_in{$module} ||= count_bugs($module);  
}
```

新的寫法 (用 `//=` 就對了)

```
my %bugs_in;
```

```
while (my $module = <>) {
```

```
    $bugs_in{$module} //= count_bugs($module);
```

```
}
```

(來不及譯成中文)



WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!

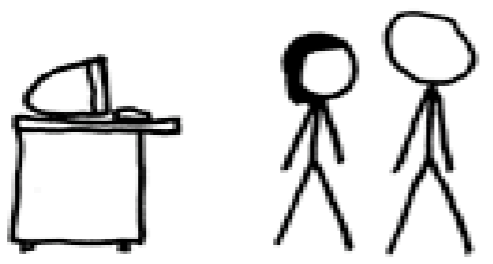


BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!

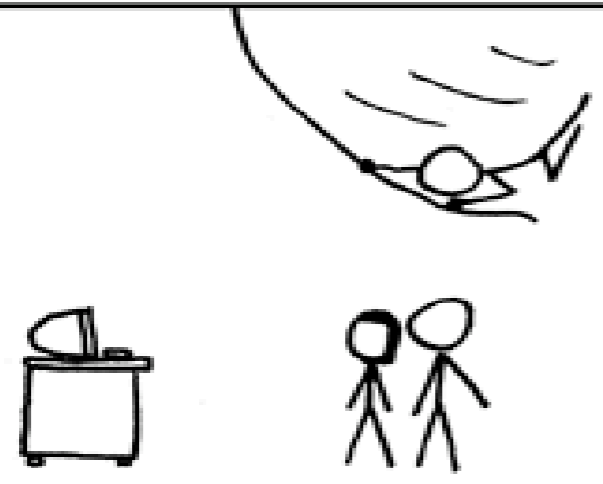


IT'S HOPELESS!

EVERYBODY STAND BACK.



I KNOW REGULAR EXPRESSIONS.



The Great Regex Engine Rewrite

The Great Regex Engine Rewrite

- The regular expression engine has been re-engineered and many new features have been added!

The Great Regex Engine Rewrite

- The regular expression engine has been re-engineered and many new features have been added!
- This should excite you. If it doesn't then you should drink more coffee!

Engine Restructured

Engine Restructured

- Recursion Eliminated

Engine Restructured

- Recursion Eliminated
 - Engine is no longer recursive. Iterative engine does not suffer from stack overflow errors in the C code for perl.

Engine Restructured

□ Recursion Eliminated

- Engine is no longer recursive. Iterative engine does not suffer from stack overflow errors in the C code for perl.
- Patterns that previously caused the engine to crash will run to finish in Perl 5.10, albeit very... very.... slowly....

Engine Restructured

□ Recursion Eliminated

- Engine is no longer recursive. Iterative engine does not suffer from stack overflow errors in the C code for perl.
- Patterns that previously caused the engine to crash will run to finish in Perl 5.10, albeit very... very.... slowly....
- It is much easier to override backtracking type behavior in an iterative engine.

Engine Restructured

□ Recursion Eliminated

- Engine is no longer recursive. Iterative engine does not suffer from stack overflow errors in the C code for perl.
- Patterns that previously caused the engine to crash will run to finish in Perl 5.10, albeit very... very.... slowly....
- It is much easier to override backtracking type behavior in an iterative engine.
- This may have a modest penalty for normal patterns, but the boys in the lab are working on it!

Engine Restructured

Engine Restructured

- Pluggable interface

Engine Restructured

- Pluggable interface
 - engine is now abstracted with a usable interface.
We can now plug in other engines....

Engine Restructured

- Pluggable interface

- engine is now abstracted with a usable interface.
We can now plug in other engines....

```
use re::engine::PCRE;
```

Engine Restructured

- Pluggable interface

- engine is now abstracted with a usable interface.
We can now plug in other engines....

use re::engine::PCRE;

- **use re 'debug' ;** is now lexically scoped, as is the use of any other engine.

Engine Restructured

- Pluggable interface

- engine is now abstracted with a usable interface. We can now plug in other engines....

use re::engine::PCRE;

- **use re 'debug' ;** is now lexically scoped, as is the use of any other engine.
- default engine can be extended or instrumented post release without requiring a full build.

Engine Restructured

- Pluggable interface

- engine is now abstracted with a usable interface. We can now plug in other engines....

use re::engine::PCRE;

- **use re 'debug' ;** is now lexically scoped, as is the use of any other engine.
- default engine can be extended or instrumented post release without requiring a full build.
- All your regex engines are belong to us!

Quantifier Combinatorial Explosion!

```
qr/ "(?: [^"\\]+ | (?:\\.)+ )* " /x
```

- This pattern matches quoted strings and supports Perl / C style escapes
- Except.... It's evil....
- The combination of * and + leads to combinatorial explosion

Call the bomb squad!

Call the bomb squad!

- How to deal with combinatorial explosion?

Call the bomb squad!

- How to deal with combinatorial explosion?
- One solution is the (λ -calculus) construct

Call the bomb squad!

- How to deal with combinatorial explosion?
- One solution is the (?>.....) construct
- This matches its contents, and then refuses to give any back should what follows not match

Call the bomb squad!

- How to deal with combinatorial explosion?
- One solution is the (?>.....) construct
- This matches its contents, and then refuses to give any back should what follows not match
- So we can rewrite the pattern to use this construct

Call the bomb squad!

- How to deal with combinatorial explosion?
- One solution is the `(?>.....)` construct
- This matches its contents, and then refuses to give any back should what follows not match
- So we can rewrite the pattern to use this construct
- Except its pretty nasty to read....

The Good, the Bad, and the Ugly!

```
qr/ "(?> (? : (?> [^"\\]+) | (?> (? : \\.)+ ) ) * ) " /x
```

- (?>....) prevents combinatorial explosion
- But there must be a nicer way to do this
- After all not *everything* in perl needs to be incomprehensible.
- Enter possessive quantifiers....

Possessive Quantifiers

```
qr/ "(?: [^"\\]+ | (?:\\.)+ )*+ " /x
```

Possessive Quantifiers

```
qr/ " (?: [^"\\]+ | (?:\\.\\.)+ )*+ " /x
```

- Now we can write common forms of this in an easier way by using possessive quantifiers.

Possessive Quantifiers

```
qr/ " (?: [^"\\]+ | (?:\\.)+ )*+ " /x
```

- Now we can write common forms of this in an easier way by using possessive quantifiers.
- A possessive quantifier matches as much as it can and never gives any back.

Possessive Quantifiers

```
qr/ " (?: [^"\\]+ | (?:\\.)+ )*+ " /x
```

- Now we can write common forms of this in an easier way by using possessive quantifiers.
- A possessive quantifier matches as much as it can and never gives any back.
- The notation is to put a plus immediately after the main quantifier.

Possessive Quantifiers

```
qr/ "(?: [^"\\]+ | (?:\\.)+ )*+ " /x
```

- Now we can write common forms of this in an easier way by using possessive quantifiers.
- A possessive quantifier matches as much as it can and never gives any back.
- The notation is to put a plus immediately after the main quantifier.
- Thus `/(?>X+)/` can be rewritten as `/X++/`

```
'aaaa' =~ m/ a+ a /x;           # Will match by backtracking
```

```
Compiling REx " a+ a "  
Final program:  
  1: PLUS (4)  
  2:   EXACT <a> (0)  
  4: EXACT <a> (6)  
  6: END (0)  
anchored "a" at 0 floating "aa" at 0..2147483647 (checking floating) plus minlen 2  
Guessing start of match in sv for REx " a+ a " against "aaaa"  
Found floating substr "aa" at offset 0...  
Found anchored substr "a" at offset 0...  
Guessed: match at offset 0  
Matching REx " a+ a " against "aaaa"  
  0 <> <aaaa>           |  1: PLUS (4)  
                        |    EXACT <a> can match 4 times out of 2147483647...  
  3 <aaa> <a>           |  4:   EXACT <a> (6)  
  4 <aaaa> <>           |  6:   END (0)  
Match successful!  
Freeing REx: " a+ a "
```

```
# Will not match because a++ wont ever give anything back!  
'aaaa' =~ m/ a++ a /x;
```

```
Compiling REX " a++ a "
```

```
Final program:
```

```
1: SUSPEND (8)  
3: PLUS (6)  
4: EXACT <a> (0)  
6: SUCCEED (0)  
7: TAIL (8)  
8: EXACT <a> (10)  
10: END (0)
```

```
[ ..... ]
```

```
Matching REX " a++ a " against "aaaa"
```

```
0 <> <aaaa> | 1:SUSPEND(8)  
0 <> <aaaa> | 3: PLUS(6)  
EXACT <a> can match 4 times out of 2147483647...  
4 <aaaa> <> | 6: SUCCEED(0)  
subpattern success...  
4 <aaaa> <> | 8:EXACT <a>(10)  
failed...
```

```
[ ..... ]
```

```
3 <aaa> <a> | 1:SUSPEND(8)  
3 <aaa> <a> | 3: PLUS(6)  
EXACT <a> can match 1 times out of 2147483647...  
4 <aaaa> <> | 6: SUCCEED(0)  
subpattern success...  
4 <aaaa> <> | 8:EXACT <a>(10)  
failed...
```

```
Match failed
```

```
Freeing REX: " a++ a "
```

Capture Buffers

```
/ (foo) $user_qr (what-number-am-i) /x
```

Capture Buffers

- Before Perl 5.10 capture buffers were numbered only.

```
/ (foo) $user_qr (what-number-am-i) /x
```

Capture Buffers

- Before Perl 5.10 capture buffers were numbered only.
- Adding a new buffer means that the numbering of the buffers following it changes, often requiring changes to the code using the pattern.

```
/ (foo) $user_qr (what-number-am-i) /x
```

Capture Buffers

- Before Perl 5.10 capture buffers were numbered only.
- Adding a new buffer means that the numbering of the buffers following it changes, often requiring changes to the code using the pattern.
- How do we know what number the last buffer in the below pattern will have?

```
/ (foo) $user_qr (what-number-am-i) /x
```

Named Capture Buffers

Named Capture Buffers

- So in Perl 5.10 we added named capture buffers.

Named Capture Buffers

- So in Perl 5.10 we added named capture buffers.
- We used the .Net syntax as most people think its nicer than Pythons.

Named Capture Buffers

- So in Perl 5.10 we added named capture buffers.
- We used the .Net syntax as most people think its nicer than Pythons.
- We didn't use their numbering scheme tho.
(Note to .Net developers: Crack Kills.)

Named Capture



Named Capture

- Declaration:



Named Capture

- Declaration:

(?<name>pat) or (? 'name' pat)



Named Capture

- Declaration:
 (?<name>pat) or (? 'name' pat)
- Backreference:



Named Capture

- Declaration:
`(?<name>pat)` or `('name'pat)`
- Backreference:
`\k<name>` or `\k'name'`



Getting results from named captures

Getting results from named captures

- `%+` hash contains the contents of the leftmost capture of a given name that was involved in the match.

Getting results from named captures

- `%+` hash contains the contents of the leftmost capture of a given name that was involved in the match.

For example: `$+{foo}`

Getting results from named captures

- `%+` hash contains the contents of the leftmost capture of a given name that was involved in the match.

For example: `$+{foo}`

- `%-` hash contains an array with the contents of all the buffers of a given name.

Getting results from named captures

- `%+` hash contains the contents of the leftmost capture of a given name that was involved in the match.

For example: `$+{foo}`

- `%-` hash contains an array with the contents of all the buffers of a given name.

For example: `$- {foo}[0]`

Getting results from named captures

- `%+` hash contains the contents of the leftmost capture of a given name that was involved in the match.

For example: `$+{foo}`

- `%-` hash contains an array with the contents of all the buffers of a given name.

For example: `$- {foo}[0]`

- `exists()` can be used to check if a buffer has content just like with any other hash.

Original Back-reference Syntax

```
/(.)(.)(.)(.)(.)(.)(.)(.)(.)(.)\11/
```

Original Back-reference Syntax

- It is tricky to use back-references in embeddable `qr//` constructs

```
/ (.) (.) (.) (.) (.) (.) (.) (.) (.) (.) \11/
```

Original Back-reference Syntax

- It is tricky to use back-references in embeddable `qr//` constructs
- Original back-reference syntax is open to ambiguity. (Is it octal or not?)

```
/ (.) (.) (.) (.) (.) (.) (.) (.) (.) (.) \11/
```

Original Back-reference Syntax

- It is tricky to use back-references in embeddable `qr//` constructs
- Original back-reference syntax is open to ambiguity. (Is it octal or not?)
- What does `\11` in the below pattern mean?

```
/ (.) (.) (.) (.) (.) (.) (.) (.) (.) (.) \11/
```

Original Back-reference Syntax

- It is tricky to use back-references in embeddable `qr//` constructs
- Original back-reference syntax is open to ambiguity. (Is it octal or not?)
- What does `\11` in the below pattern mean?
- Is that octal or a back-reference?

```
/ (.) (.) (.) (.) (.) (.) (.) (.) (.) (.) \11/
```

More Problems with Numeric Backreferences

More Problems with Numeric Backreferences

- Concatenation is unsafe (think of “\1” and “1”)

More Problems with Numeric Backreferences

- Concatenation is unsafe (think of “\1” and “1”)
- This could cause problems for code generators

More Problems with Numeric Backreferences

- Concatenation is unsafe (think of “\1” and “1”)
- This could cause problems for code generators
- Traditional styles of escaping don't help. About the best solution is to use (?:\1) or /x and spaces to separate the components.

More Problems with Numeric Backreferences

- Concatenation is unsafe (think of “\1” and “1”)
- This could cause problems for code generators
- Traditional styles of escaping don't help. About the best solution is to use (?:\1) or /x and spaces to separate the components.
- It would be nice to have a syntax that would let us avoid these problems

More Problems with Numeric Backreferences

- Concatenation is unsafe (think of “\1” and “1”)
- This could cause problems for code generators
- Traditional styles of escaping don't help. About the best solution is to use (?:\1) or /x and spaces to separate the components.
- It would be nice to have a syntax that would let us avoid these problems
- So we invented one....

New Back-Reference Syntax

New Back-Reference Syntax

- New syntax for \1

New Back-Reference Syntax

- New syntax for `\1`
`\g{1}` or `\g1`

New Back-Reference Syntax

- New syntax for `\1`
`\g{1}` or `\g1`
- Relative back-references

New Back-Reference Syntax

- New syntax for \1
 - `\g{1}` or `\g1`
- Relative back-references
 - `\g{-1}` or `\g-1`

New Back-Reference Syntax

- New syntax for $\backslash 1$

$\backslash g\{1\}$ or $\backslash g1$

- Relative back-references

$\backslash g\{-1\}$ or $\backslash g-1$

refers to the previous Nth capture buffer

New Back-Reference Syntax

- New syntax for `\1`
`\g{1}` or `\g1`
- Relative back-references
`\g{-1}` or `\g-1`
refers to the previous Nth capture buffer
- For instance a generic embeddable dupe word matcher:

New Back-Reference Syntax

- New syntax for `\1`

`\g{1}` or `\g1`

- Relative back-references

`\g{-1}` or `\g-1`

refers to the previous Nth capture buffer

- For instance a generic embeddable dupe word matcher:

```
my $dupew = qr/(\w+)\s+\g{-1}/;
```

Matching balanced constructs....

Matching balanced constructs....

- Is not a problem that traditional regular expression engines can solve

Matching balanced constructs....

- Is not a problem that traditional regular expression engines can solve
- They can only handle an arbitrary nesting depth

Matching balanced constructs....

- Is not a problem that traditional regular expression engines can solve
- They can only handle an arbitrary nesting depth
- Except Perl doesn't use a traditional regular expression engine

Matching balanced constructs....

- Is not a problem that traditional regular expression engines can solve
- They can only handle an arbitrary nesting depth
- Except Perl doesn't use a traditional regular expression engine
- So we can write patterns that will match any level of nesting if we wish

Matching balanced constructs....

- Is not a problem that traditional regular expression engines can solve
- They can only handle an arbitrary nesting depth
- Except Perl doesn't use a traditional regular expression engine
- So we can write patterns that will match any level of nesting if we wish
- But it's not exactly easy.....

Recursive Patterns Using Eval

Recursive Patterns Using Eval

- Old way - dynamic patterns

```
our $pat;
```

```
$pat = qr/\((?>(?!>[^\(\)]+)|(?{ $pat }))*\)/x;
```

```
if ('(x(x)y(x)x)' =~ /^($pat)$/) { ... }
```

- Inherent problems

1. Slow - requires the interpreter to resolve what \$pat holds
2. Fat - requires two patterns (inner and outer)
3. Clumsy - requires use of global vars
4. Ugly - pattern is not self contained. Pattern does not in of itself explain what it does.

Old way in more detail

```
qr/  
  \(          # Open paren  
  (?>       # Possessive subgroup  
    (?> [^()]+ ) # Grab all the non parens we can  
  |          # or  
    (??{$pat}) # Recurse, grab a balanced paren  
  )*        # Zero or more times  
  \)        # Close paren  
/x;
```

Recursive Patterns

Recursive Patterns

- With the (?1) notation things are a little easier.

Recursive Patterns

- With the (?1) notation things are a little easier.

```
if ('(x(x)y(x)x)' =~ m/^( \ ( (?> [ ^ ( ) ] ++ | (?1) ) * \ ) ) $ / x )
```

Recursive Patterns

- With the (?1) notation things are a little easier.

```
if ('(x(x)y(x)x)' =~ m/^( \ ( (?> [ ^ ( ) ] ++ | (?1) ) * \ ) ) $ / x )  
{ ... }
```

Recursive Patterns

- With the (?1) notation things are a little easier.

```
if ( ' (x(x)y(x)x) '=~m/^ (\ ( (?> [ ^ ( ) ] ++ | (?1) ) * \ ) ) $ / x )  
{ ... }
```

- Advantages:

Recursive Patterns

- With the (?1) notation things are a little easier.

```
if ( ' (x(x)y(x)x) '=~m/^ (\ ( (?> [ ^ ( ) ] ++ | (?1) ) * \ ) ) $ / x )  
{ ... }
```

- Advantages:

1. **Faster** – Perl interpreter is not involved. Since pattern may not change the engine can optimize the pattern.

Recursive Patterns

- With the (?1) notation things are a little easier.

```
if ( ' (x(x)y(x)x) '=~m/^\ ( (?> [ ^ ( ) ] ++ | (?1) ) * \ ) $ / x )  
{ ... }
```

- Advantages:

1. **Faster** – Perl interpreter is not involved. Since pattern may not change the engine can optimize the pattern.
2. **Smaller** – pattern need not be duplicated

Recursive Patterns

- With the (?1) notation things are a little easier.

```
if ( ' (x(x)y(x)x) '=~m/^( \ ( (?> [ ^ ( ) ] ++ | (?1) ) * \ ) ) $ / x )  
{ ... }
```

- Advantages:

1. **Faster** – Perl interpreter is not involved. Since pattern may not change the engine can optimize the pattern.
2. **Smaller** – pattern need not be duplicated
3. **Self contained** –no global vars

Recursive Patterns

- With the (?1) notation things are a little easier.

```
if ( ' (x(x)y(x)x) '=~m/^( \ ( (?> [ ^ ( ) ] ++ | (?1) ) * \ ) ) $ / x )  
{ ... }
```

- Advantages:

1. **Faster** – Perl interpreter is not involved. Since pattern may not change the engine can optimize the pattern.
2. **Smaller** – pattern need not be duplicated
3. **Self contained** –no global vars
4. **Self describing** – pattern is self contained. Context is not required to see what the pattern matches.

Pattern Recursion in more detail

Pattern Recursion in more detail

- Pattern recursion allows us to treat the contents of a particular pattern buffer as an independent subexpression

Pattern Recursion in more detail

- Pattern recursion allows us to treat the contents of a particular pattern buffer as an independent subexpression
- (?1) recurses into the first capture buffer in the pattern

Pattern Recursion in more detail

- Pattern recursion allows us to treat the contents of a particular pattern buffer as an independent subexpression
- (?1) recurses into the first capture buffer in the pattern
- (?R) and its alias (?0) allows us to treat the entire pattern as an independent subexpression

Pattern Recursion in more detail

- Pattern recursion allows us to treat the contents of a particular pattern buffer as an independent subexpression
- (?1) recurses into the first capture buffer in the pattern
- (?R) and its alias (?0) allows us to treat the entire pattern as an independent subexpression
- If we have named a buffer we can also recurse into it by name by using (?&NAME)

Pattern Recursion in more detail

- Pattern recursion allows us to treat the contents of a particular pattern buffer as an independent subexpression
- (?1) recurses into the first capture buffer in the pattern
- (?R) and its alias (?0) allows us to treat the entire pattern as an independent subexpression
- If we have named a buffer we can also recurse into it by name by using (?&NAME)
- This is useful for writing grammars...

New way in more detail

qr/

```
^          # Start of string
(         # Start capture group 1
 \ (      # Open paren
 (?>     # Possessive capture subgroup
    [^ () ]++ # Grab all the non parens we can
 |        # or
    (?1)   # Recurse into group 1
 )*       # Zero more times
 \ )      # Close Paren
 )        # End capture group 1
 $        # End of string
```

/x;

Recursive Patterns

Recursive Patterns

- Need not be recursive:

Recursive Patterns

- Need not be recursive:

```
if ("AABB"=~/(A)(?1)(?2)(B)/) { . . . }
```

Recursive Patterns

- Need not be recursive:

```
if ("AABB"=~/(A)(?1)(?2)(B)/) { ... }
```

- Think “regex subroutine” or Perl6 Rule (sorta)

Recursive Patterns

- Need not be recursive:

```
if ("AABB"=~/(A)(?1)(?2)(B)/) { . . . }
```

- Think “regex subroutine” or Perl6 Rule (sorta)
- Relative referencing is possible:

Recursive Patterns

- Need not be recursive:

```
if ("AABB" =~ / (A) (?1) (?2) (B) /) { . . . }
```

- Think “regex subroutine” or Perl6 Rule (sorta)

- Relative referencing is possible:

```
/ (A) (?-1) (?+1) (B) /
```

Recursive Patterns

- Need not be recursive:

```
if ("AABB" =~ / (A) (?1) (?2) (B) /) { ... }
```

- Think “regex subroutine” or Perl6 Rule (sorta)

- Relative referencing is possible:

```
/ (A) (?-1) (?+1) (B) /
```

- Relative referencing facilitates easy embedding and reuse of recursive patterns

```

qr {
  (? (DEFINE)
    (?<address>          (?&mailbox) | (?&group))
    (?<mailbox>          (?&name_addr) | (?&addr_spec))
    (?<name_addr>        (?&display_name)? (?&angle_addr))
    (?<angle_addr>       (?&CFWS)? < (?&addr_spec) > (?&CFWS)?)
    (?<group>            (?&display_name) : (?: (?&mailbox_list) | (?&CFWS))? ;
                          (?&CFWS)?)

    (?<display_name>     (?&phrase))
    (?<mailbox_list>     (?&mailbox) (?: , (?&mailbox))* )
    (?<address_list>     (?&address) (?: , (?&address))* )
    (?<addr_spec>        (?&local_part) \@ (?&domain))
    (?<local_part>       (?&dot_atom) | (?&quoted_string))
    (?<domain>           (?&dot_atom) | (?&domain_literal))
    (?<domain_literal>  (?&CFWS)? \[ (?: (?&FWS)? dcontent)* (?&FWS)?
                          \] (?&CFWS)?)

    (?<dcontent>         (?&dtext) | (?&quoted_pair))
    (?<dtext>            (?&NO_WS_CTL) | [\x21-\x5a\x5e-\x7e])
    (?<atext>           (?&ALPHA) | (?&DIGIT) | [!#\$\%&'*+\/=?^_`{|}~])
    (?<atom>            (?&CFWS)? (?&atext)+ (?&CFWS)?)
    (?<dot_atom>        (?&CFWS)? (?&dot_atom_text) (?&CFWS)?)
    (?<dot_atom_text>   (?&atext)+ (?: \. (?&atext)+)* )
    (?<text>            [\x01-\x09\x0b\x0c\x0e-\x7f])
    (?<quoted_pair>     \\ (?&text))
    (?<qtext>           (?&NO_WS_CTL) | [\x21\x23-\x5b\x5d-\x7e])
    (?<qcontent>        (?&qtext) | (?&quoted_pair))
    (?<quoted_string>   (?&CFWS)? (?&DQUOTE) (?: (?&FWS)? (?&qcontent))*
                          (?&FWS)? (?&DQUOTE) (?&CFWS)?)

    (?<word>            (?&atom) | (?&quoted_string))
    (?<phrase>          (?&word)+)
    # Folding white space
    (?<FWS>             (?: (?&WSP)* (?&CRLF))?) (?&WSP)+)
    (?<ctext>           (?&NO_WS_CTL) | [\x21-\x27\x2a-\x5b\x5d-\x7e])
    (?<ccontent>        (?&ctext) | (?&quoted_pair) | (?&comment))
    (?<comment>        \( (?: (?&FWS)? (?&ccontent))* (?&FWS)? \) )
    (?<CFWS>           (?: (?&FWS)? (?&comment))*
                          (?: (?&FWS)? (?&comment)) | (?&FWS)))

    # No whitespace control
    (?<NO_WS_CTL>      [\x01-\x08\x0b\x0c\x0e-\x1f\x7f])
    (?<ALPHA>          [A-Za-z])
    (?<DIGIT>          [0-9])
    (?<CRLF>           \x0d \x0a)
    (?<DQUOTE>         ")
    (?<WSP>            [\x20\x09])

  )
  (?&address)
}x;

```

What's this (? (DEFINE)....) thing?

What's this (?DEFINE)....) thing?

- Recursing to a named capture buffer is pretty much like matching a rule

What's this (?DEFINE)....) thing?

- Recursing to a named capture buffer is pretty much like matching a rule
- So it would be nice to be able to define a rule out of place from where it is first used

What's this (?**DEFINE**)....) thing?

- Recursing to a named capture buffer is pretty much like matching a rule
- So it would be nice to be able to define a rule out of place from where it is first used
- So you can use the (?**DEFINE**)....) construct

What's this (?**DEFINE**)....) thing?

- Recursing to a named capture buffer is pretty much like matching a rule
- So it would be nice to be able to define a rule out of place from where it is first used
- So you can use the (?**DEFINE**)....) construct
- Whatever is in the will never be used as part of the match

What's this (?**DEFINE**)....) thing?

- Recursing to a named capture buffer is pretty much like matching a rule
- So it would be nice to be able to define a rule out of place from where it is first used
- So you can use the (?**DEFINE**)....) construct
- Whatever is in the will never be used as part of the match
- Unless it is recursed into.

The “Keep” pattern \K

The “Keep” pattern \K

- Originally implemented via `Regexp::Keep` by Jeff Pinyan (japhy)

The “Keep” pattern \K

- Originally implemented via `Regexp::Keep` by Jeff Pinyan (japhy)
- Says that everything before the `\K` should not be included in a match. Same thing as `<(` in perl6.

The “Keep” pattern \K

- Originally implemented via `Regexp::Keep` by Jeff Pinyan (japhy)
- Says that everything before the `\K` should not be included in a match. Same thing as `<(` in perl6.
- This is effectively a form of variable length positive look-behind, but much more efficient.

The “Keep” pattern \K

- Originally implemented via `Regexp::Keep` by Jeff Pinyan (japhy)
- Says that everything before the `\K` should not be included in a match. Same thing as `<(` in perl6.
- This is effectively a form of variable length positive look-behind, but much more efficient.
- Especially useful in substitution as it means you often can avoid capturing

The “Keep” pattern \K

- Originally implemented via `Regexp::Keep` by Jeff Pinyan (japhy)
- Says that everything before the `\K` should not be included in a match. Same thing as `<(` in perl6.
- This is effectively a form of variable length positive look-behind, but much more efficient.
- Especially useful in substitution as it means you often can avoid capturing
- Thanks japhy!

```

cmpthese -1, {
  keep => sub {
    my $str = $test;
    $str =~ s/fo+\Kbar/baz/;
  },
  nokeep=> sub {
    my $str = $test;
    $str =~ s/(fo+)bar/${1}baz/;
  },
}

```

	Rate	nokeep	keep
nokeep	74490/s	--	-88%
keep	595923/s	700%	--

The /p modifier!

The /p modifier!

- Due to the dynamic nature of Perl the use of \$`, \$& and \$' anywhere in a script has a global performance penalty.

The /p modifier!

- Due to the dynamic nature of Perl the use of \$`, \$& and \$' anywhere in a script has a global performance penalty.
- With /p modifier the variables \${^PREMATCH}, \${^MATCH}, and \${^POSTMATCH} maybe be used instead.

The /p modifier!

- Due to the dynamic nature of Perl the use of \$`, \$& and \$' anywhere in a script has a global performance penalty.
- With /p modifier the variables \${^PREMATCH}, \${^MATCH}, and \${^POSTMATCH} maybe be used instead.
- No penalty! Or at least the same penalty as from using capturing, that is only the regex with the /p modifier will be affected.

Branch Reset Pattern

```
while (/\G\s*
      (?:value=(\w+))
      |"(\w+)")
      /gx)
{ my $text= defined $1 ? $1 : $2 }
```

Branch Reset Pattern

```
while (/\G\s*
      (?:value=(\w+)
      |"(\w+)")
      /gx)
{ my $text= defined $1 ? $1 : $2 }
```

- Sometimes you want to match one of several possibilities, but only capture part of what you match. For instance in a tokenizer

Branch Reset Pattern

```
while (/\G\s*
      (?:value=(\w+)
      |"(\w+)")
      /gx)
{ my $text= defined $1 ? $1 : $2 }
```

- Sometimes you want to match one of several possibilities, but only capture part of what you match. For instance in a tokenizer
- But that can be a real pain. So H. Merijn Brand suggested a better way....

Branch Reset Pattern

```
while (/\G\s*
      (?:value=(\w+)
      |"(\w+)")
      /gx)
{ my $text= defined $1 ? $1 : $2 }
```

- Sometimes you want to match one of several possibilities, but only capture part of what you match. For instance in a tokenizer
- But that can be a real pain. So H. Merijn Brand suggested a better way....
- (?|.....)

Branch Reset Pattern

```
while (/\G\s*
      (?|
        value=(\w+)
      |
        "(\w+)"
      )/gx)
{ my $text= $1 }
```

Branch Reset Pattern

```
while (/\G\s*
      (?|
        value=(\w+)
      |
        "(\w+)"
      )/gx)
{ my $text= $1 }
```

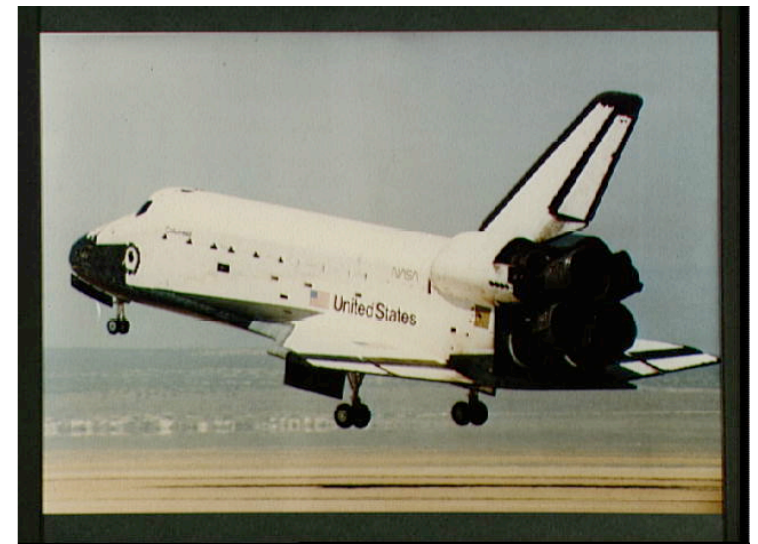
- Capture buffers in each branch share the same numbers, in other words the buffer number is reset to the same value at the start of each branch.

Branch Reset Pattern

```
while (/\G\s*
      (?|
        value=(\w+)
      |
        "(\w+)"
      )/gx)
{ my $text= $1 }
```

- Capture buffers in each branch share the same numbers, in other words the buffer number is reset to the same value at the start of each branch.
- Buffers following the construct are numbered as though there is only one branch, that with the most buffers in it.

New Optimizations



New Optimizations

- Trie and Aho-Corasick matching

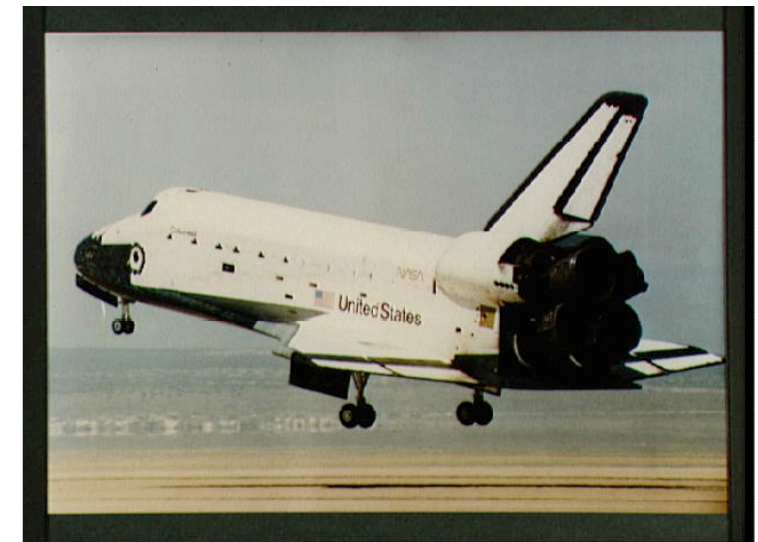


New Optimizations

- Trie and Aho-Corasick matching
 - Alternations starting with literal text will be merged into a single TRIE construct.

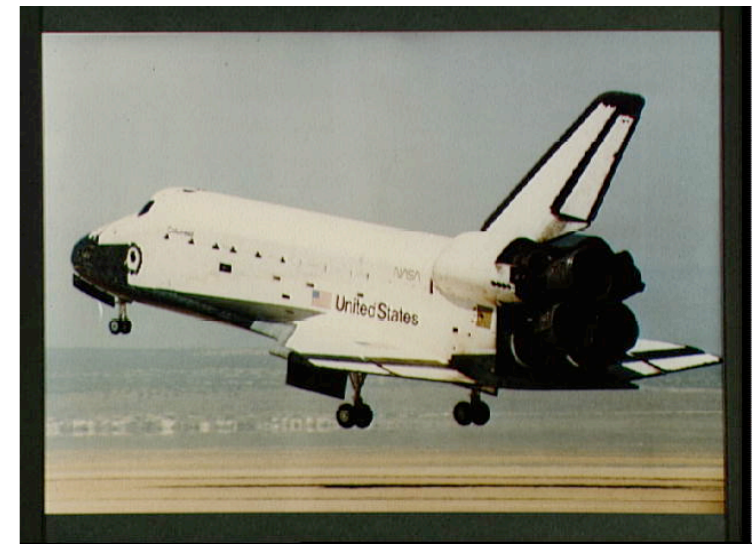


New Optimizations



- Trie and Aho-Corasick matching
 - Alternations starting with literal text will be merged into a single TRIE construct.
 - If a TRIE is the first matching regop in a regular expression the engine will create an Aho-Corasick matcher and use it for start point determination.

New Optimizations



- Trie and Aho-Corasick matching
 - Alternations starting with literal text will be merged into a single TRIE construct.
 - If a TRIE is the first matching regop in a regular expression the engine will create an Aho-Corasick matcher and use it for start point determination.
 - Much more efficient, match time is not dependent on the number of sub-patterns in the TRIE, unlike normal alternation.

New Optimizations

New Optimizations

- One cool thing about tries is that they allow us to find common prefixes....

New Optimizations

- One cool thing about tries is that they allow us to find common prefixes....
- Which we can extract from the alternation if it exists:

New Optimizations

- One cool thing about tries is that they allow us to find common prefixes....
- Which we can extract from the alternation if it exists:

/foam|foal|foad/

New Optimizations

- One cool thing about tries is that they allow us to find common prefixes....
- Which we can extract from the alternation if it exists:

/foam|foal|foad/

will be optimized to perform the same as

New Optimizations

- One cool thing about tries is that they allow us to find common prefixes....
- Which we can extract from the alternation if it exists:

/foam|foal|foad/

will be optimized to perform the same as

/foa[mld]/

Backtracking



A bit about backtracking

A bit about backtracking

- Perls engine is a backtracking engine

A bit about backtracking

- Perls engine is a backtracking engine
- Don't confuse NFA with 'backtracking', just because an engine is NFA doesn't mean it uses backtracking.

A bit about backtracking

- Perl's engine is a backtracking engine
- Don't confuse NFA with 'backtracking', just because an engine is NFA doesn't mean it uses backtracking.
- It has to use backtracking to provide some of the advanced features, especially backreferences, which strictly speaking aren't regular

A bit about backtracking

- Perls engine is a backtracking engine
- Don't confuse NFA with 'backtracking', just because an engine is NFA doesn't mean it uses backtracking.
- It has to use backtracking to provide some of the advanced features, especially backreferences, which strictly speaking aren't regular
- Leftmost-longest is also a side effect of backtracking.

Backtracking Control Verbs

Backtracking Control Verbs

- Regexes are normally declarative, meaning that their behavior, in theory, should be independent of their implementation.

Backtracking Control Verbs

- Regexes are normally declarative, meaning that their behavior, in theory, should be independent of their implementation.
- In real life implementation has a significant effect on behavior and performance.

Backtracking Control Verbs

- Regexes are normally declarative, meaning that their behavior, in theory, should be independent of their implementation.
- In real life implementation has a significant effect on behavior and performance.
- Controlling how the engine backtracks can result in significant speedups

Backtracking Control Verbs

- Regexes are normally declarative, meaning that their behavior, in theory, should be independent of their implementation.
- In real life implementation has a significant effect on behavior and performance.
- Controlling how the engine backtracks can result in significant speedups
- Thus we now have verbs to control this behavior

New Verbs: (*VERB)

- (*FAIL)
- (*ACCEPT)
- (*PRUNE)
- (*MARK)
- (*SKIP)
- (*THEN)
- (*COMMIT)
- (*PRUNE:NAME)
- (*MARK:NAME)
- (*:NAME)
- (*SKIP:NAME)
- (*THEN:NAME)
- (*COMMIT:NAME)

Simple Backtracking Control Verbs

□ (*FAIL)

- simplest verb. Syntactic sugar for (?!)
- can be used to force the engine to backtrack and therefore find all matches in a pattern, similar to how “exhaustive matching” would work in Perl6

```
'aaab' =~ /a+b?(?{print $&}) (*FAIL) /
```

- would print out every possible substring that matches the pattern `/a+b?/`

Exhaustive matching with (*FAIL)

```
'aaab' =~ /(?{ print "\n" }) a+ b? (?{ print "$& " }) (*FAIL)/x;  
print "\n";
```

```
aaab aaa aa a  
aab aa a  
ab a
```

Simple Backtracking Control Verbs

□ (*ACCEPT)

- causes a pattern to be accepted at the current match point even if there is more pattern to be matched.
- think of this as being something like a “return” statement for regexes.

'AC' =~ /A (? B | C (*ACCEPT) | D) E/x

will match

Using (*ACCEPT)

```
'AC'=~/A(?: B | C (*ACCEPT) | D ) E/x  
and print $&,"\\n";
```

```
Guessing start of match in sv for REx "A(?: B | C (*ACCEPT) | D ) E" against "AC"  
Found anchored substr "A" at offset 0...  
Gussed: match at offset 0  
Matching REx "A(?: B | C (*ACCEPT) | D ) E" against "AC"  
  0 <> <AC>          | 1:EXACT <A>(3)  
  1 <A> <C>          | 3:TRIE-EXACT[BCD](15)  
  1 <A> <C>          |   State:    1 Accepted:    0 Charid:  2 CP:  43 After State:    3  
  2 <AC> <>          |   State:    3 Accepted:    1 Charid:  2 CP:   0 After State:    0  
                        got 1 possible matches  
                        only one match left: #2 <C>  
  2 <AC> <>          | 9:ACCEPT0(15)  
Match successful!  
AC
```

Backtracking Control Verbs

- Rest of the verbs support an argument and are of the form (*NAME:arg)
- When such verbs are used the regex engine will set the package variable \$REGERROR and \$REGMARK.
- On success \$REGERROR is set to false, and \$REGMARK set to the 'arg' of the last verb involved in the match
- On failure it is the reverse.

Backtracking Control Verbs

□ (*PRUNE)

- when backtracked into, the match fails at the current starting position.
- similar to (?>...) except that prune is unary and not a bracketing construct. Thus you can write:

```
/A (? B | C (*PRUNE) ) D/
```

- Can be used with (*FAIL) to find the longest match possible for every start position in the string.

```
/a+b? (? {print $&} ) (*PRUNE) (*FAIL) /
```

Using (*PRUNE)

```
'aaab' =~ / a+ b? (?{print "PRUNE: $&" }) (*PRUNE) (*FAIL) /x;
```

```
PRUNE: aaab  
PRUNE: aab  
PRUNE: ab
```

Backtracking Control Verbs

- (*MARK:name)

- when executed “marks” the current position in the string and gives it a name.
- Provides a way to see what “path” the engine has taken through the pattern.

```
/foo (*:A) | bar (*:B) | baz (*:C) /
```

```
and print $REGMARK;
```

- (*:name) is a short form of (*MARK:name)
- Invented for Spam-Assassin

Using (*MARK)

```
'baz' =~ /foo (*:A) | bar (*:B) | baz (*:C) /  
and print "REGMARK: $REGMARK";
```

```
REGMARK: C
```

Backtracking Control Verbs

- (*SKIP:name)
 - when backtracked into the engine will reject all matches up to the point where the cursor was when the (*SKIP) was entered.
 - can be coupled with (*MARK:name) to skip the text up to the point where the (*MARK) was executed.
 - Can be used with (*FAIL) to find all of the non overlapping matches in a string:
`'aaabaaab' =~ /a+b? (*SKIP) (?{print $&}) (*FAIL) /`

Using (*SKIP)

```
'aaabaaab' =~ / a+ b? (*SKIP) (?{print "SKIP: $&"}) (*FAIL) /x;
```

```
SKIP: aaab  
SKIP: aaab
```

Backtracking Control Verbs

□ (*THEN)

- When backtracked into causes the pattern to backtrack directly to the next alternation in the pattern.
- Can be thought of as an if/then statement for regular expressions.

`/COND1 (*THEN) REST1 | COND2 (*THEN) REST2/`

- When not used in an alternation acts just like (*PRUNE) would.
- In Perl6 this is called the “cut group” operator.

Backtracking Control Verbs

□ (*COMMIT)

- When backtracked into causes the pattern to fail outright. The engine will not try to match the pattern at any other start point.

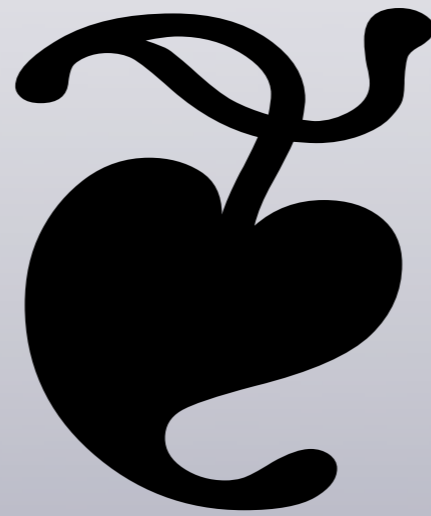
- In Perl6 this is called... <commit>

'foofoofoobar' =~ / (fo+)* (*COMMIT) [BC]ar /x;

will fail, and will do so quickly.

'foofoofoobar' =~ / (fo+)* [BC]ar /x;

won't, although it could be worse as the super-linear cache kicks in. (see commit.pl)



斷言函式 (Assertion)

斷言函式 (Assertion)

🍎 **sub** assert :assertion {...}

斷言函式 (Assertion)

🍎 `sub assert :assertion {...}`

🍎 `perl -A` 啟用所有斷言

斷言函式 (Assertion)

🍎 **sub** assert :assertion {...}

🍎 perl -A 啟用所有斷言

🍎 perl -A=Foo 啟用特定斷言

斷言函式 (Assertion)

🍎 `sub assert :assertion {...}`

🍎 `perl -A` 啟用所有斷言

🍎 `perl -A=Foo` 啟用特定斷言

🍎 自動忽略未啟用的斷言呼叫

自行定義 pragma

自行定義 `pragma`

🍎 `use strict;` # 區塊內生效

自行定義 pragma

🍎 `use strict;` # 區塊內生效

🍎 編譯時將資訊寫入 `%^H` 變數

自行定義 pragma

- 🍎 `use strict;` # 區塊內生效
- 🍎 編譯時將資訊寫入 `%^H` 變數
- 🍎 執行時用 `caller` 取回編譯資訊

```
package yes_means_no;  
sub import    { $^H{yes_means_no} = 1 }  
sub unimport  { $^H{yes_means_no} = 0 }  
1;
```

```
package yes_means_no;
sub import    { $^H{yes_means_no} = 1 }
sub unimport  { $^H{yes_means_no} = 0 }
1;
```

```
package main;
sub yes {
    my $hints = (caller(0))[10];
    if ($hints->{yes_means_no}) {
        return "No";
    }
    else {
        return "Yes";
    }
}
```

```
package yes_means_no;
sub import    { $^H{yes_means_no} = 1 }
sub unimport  { $^H{yes_means_no} = 0 }
1;
```

```
package main;
sub yes {
    my $hints = (caller(0))[10];
    if ($hints->{yes_means_no}) {
        return "No";
    }
    else {
        return "Yes";
    }
}
```

```
{ use yes_means_no; print yes(); } # "No"
print yes();                       # "Yes"
```

UNIVERSAL :: isa

UNIVERSAL :: isa

🍷 \$obj->isa('Logger');

UNIVERSAL :: isa

🍏 `$obj ->isa('Logger');`

🍏 `$obj` 必須繼承 `Logger` 類型

UNIVERSAL :: isa

🍎 `$obj ->isa('Logger');`

🍎 `$obj` 必須繼承 `Logger` 類型

🍎 不支援集成、委派、擬倣等關係

UNIVERSAL :: DOES

UNIVERSAL :: DOES

🍷 \$obj->DOES('Logger');

UNIVERSAL :: DOES

🍎 `$obj ->DOES('Logger');`

🍎 `$obj` 必須實作 `Logger` 角色

UNIVERSAL :: DOES

🍎 `$obj ->DOES('Logger');`

🍎 `$obj` 必須實作 `Logger` 角色

🍎 類型可自行定義 `DOES` 方法

Hash :: Util :: FieldHash

Hash::Util::FieldHash

🍎 用物件當雜湊鍵：`$hash{$obj}`

Hash::Util::FieldHash

- 🍎 用物件當雜湊鍵：`$hash{$obj}`
- 🍎 物件消滅時自動刪除雜湊鍵

Hash::Util::FieldHash

- 🍎 用物件當雜湊鍵：`$hash{$obj}`
- 🍎 物件消滅時自動刪除雜湊鍵
- 🍎 大幅提昇 `Class::InsideOut` 等物件管理模組的執行效能

更多核心模組

更多核心模組

🍎 `encoding::warnings`

更多核心模組

🍎 `encoding::warnings`

🍎 `Math::BigInt::FastCalc`

更多核心模組

- 🍎 `encoding::warnings`
- 🍎 `Math::BigInt::FastCalc`
- 🍎 `Time::Piece`

更多核心模組

- 🍷 `encoding::warnings`
- 🍷 `Math::BigInt::FastCalc`
- 🍷 `Time::Piece`
- 🍷 `Win32API::File`

更多核心模組

- 🍷 `encoding::warnings`
- 🍷 `Math::BigInt::FastCalc`
- 🍷 `Time::Piece`
- 🍷 `Win32API::File`
- 🍷 `CPANPLUS!`

CPANPLUS . pm

CPANPLUS . pm

 CPAN . pm 的接班者

CPANPLUS . pm

- 🍏 CPAN.pm 的接班者
- 🍏 便捷可靠的 API 和互動界面

CPANPLUS . pm

- 🍎 CPAN.pm 的接班者
- 🍎 便捷可靠的 API 和互動界面
- 🍎 五年過去終於成為核心模組

CPANPLUS . pm

- 🍎 CPAN . pm 的接班者
- 🍎 便捷可靠的 API 和互動界面
- 🍎 五年過去終於成為核心模組
- 🍎 帶了一大串模組進來!

CPANPLUS.pm

Archive::Extract	Module::Build
Archive::Tar	Module::CoreList
Compress::Zlib	Module::Load
Digest::SHA	Module::Load::Conditional
ExtUtils::CBuilder	Module::Loaded
ExtUtils::XSBuilder	Module::Pluggable
File::Fetch	Object::Accessor
IO::Zlib	Package::Constants
IPC::Cmd	Params::Check
Locale::Maketext::Simple	Term::UI
Log::Message	...族繁不及備載



帶了一大串模組進來!

效能提昇

效能提昇

 加速萬國碼字串處理

效能提昇

- 🍎 加速萬國碼字串處理
- 🍎 減少常數函式記憶體用量

效能提昇

- 🍎 加速萬國碼字串處理
- 🍎 減少常數函式記憶體用量
- 🍎 更有效率的執行緒管理

還有還有...

還有還有...

- 🍎 更詳盡的污染檢查 (taint check)

還有還有...

- 🍎 更詳盡的污染檢查 (taint check)
- 🍎 穩定的 .pmc 支援 (Module::Compile)

還有還有...

- 🍎 更詳盡的污染檢查 (taint check)
- 🍎 穩定的 .pmc 支援 (Module::Compile)
- 🍎 新的核心文件 (perlunitut 等)

還有還有...

- 🍎 更詳盡的污染檢查 (taint check)
- 🍎 穩定的 .pmc 支援 (Module::Compile)
- 🍎 新的核心文件 (perlunitut 等)
- 🍎 ...其他請自行參閱 perldelta 😊

Fin.