Philip Hood
hood@panix.com

"a triptych in fragments of modern perl," no. 3
dc.pm

4 December 2012
Chapter 8.
Style and Efficiency

~.1 perlcritic

[hood@panix3 ~/perl/third 03:37:20]$ perltidy -b markov
[hood@panix3 ~/perl/third 03:37:29]$ perlcritic -l markov
Version string used at line 3, column 1. Use a real number instead.
(Severity: 3)
Code before warnings are enabled at line 6, column 1. See page 431 of PBP.
(Severity: 4)
Quotes used with a string containing no non-whitespace characters at line 12,
column 13. See page 53 of PBP. (Severity: 2)
Always unpack @_ first at line 17, column 1. See page 178 of PBP.
(Severity: 4)
[hood@panix3 ~/perl/third 03:37:39]$ cat markov
#!/net/u/11/h/hood/local/bin/perl
use v5.16.2;
use Carp;
our $VERSION = 1.0;
my @rules = ('101',
            '|0' => '0||',
            '|1' => '0|',
            '|0' => '',
            );
say markov(@rules) or croak 'cannot say';
sub markov {
    my $in = $_[0];
    my @r = @_[1 .. $#_];
    while ( defined( my $k = shift @r ) && defined( my $v = shift @r ) ) {
        $k = quotemeta $k;
        say $in or croak 'cannot say';
        $in =~ s/$k/$v/sxm && ( $_[0] = $in ) && goto &markov;
    }
    return $in;
}
[hood@panix3 ~/perl/third 03:37:58]$ ./markov
101
101
0|01
00||1
00||1
00||0|
catching exceptions

Try::Tiny

    # handle errors with a catch handler
    try {
        die "foo";
    } catch {
        warn "caught error: " . $_ ;  # not $@
    };
~.3 pragmas

perl5i (schwern, yapc 2012, “perl, the next generation”)

autobox

“autobox allows methods to be defined for and called on most unblessed variables. This means you can call methods on ordinary strings, lists and hashes! It also means perl5i can add a lot of functionality without polluting the global namespace.”

```
[hood@panix3 ~/perl/third 04:53:40]$ cat p
use perl5i::2;
my @am = (2,3,3,4,234);
say @am->pop while @am;
[hood@panix3 ~/perl/third 04:53:45]$ perl p
234
4
3
3
2
```
A pragma is a module which influences some aspect of the compile time or run time behaviour of Perl, such as `strict` or `warnings`. With Perl 5.10 you are no longer limited to the built in pragmata; you can now create user pragmata that modify the behaviour of user functions within a lexical scope.

Chapter 9.
Managing Real Programs

~.1 perl best practices

~.2 testing

~.3 warnings

    perldoc perllexwarn

    use warnings FATAL => 'all'
    => 'deprecated'

~.4 Code Generation

    metaprogramming
Manipulating stashes (Perl's symbol tables) is occasionally necessary, but incredibly messy, and easy to get wrong. This module hides all of that behind a simple API.

```perl
my $stash = Package::Stash->new('Foo');
$stash->add_symbol('%foo', {bar => 1});
# $Foo::foo{bar} == 1
$stash->has_symbol('$foo')  # false
my $namespace = $stash->namespace;
  *{ $namespace->{foo} }{HASH}  # {bar => 1}
```

### Class::MOP

(Moose) ...

#### What is a Meta Object Protocol

A meta object protocol is an API to an object system.

To be more specific, it abstracts the components of an object system (classes, object, methods, object attributes, etc.). These abstractions can then be used to inspect and manipulate the object system which they describe.

It can be said that there are two MOPs for any object system; the implicit MOP and the explicit MOP. The implicit MOP handles things like method dispatch or inheritance, which happen automatically as part of how the object system works. The explicit MOP typically handles the introspection/reflection features of the object system.
Chapter 10.
Perl Beyond Syntax

~.1 idiomatic perl
  (effective perl programming, joseph hall)

File Slurping

```
[hood@panix3 ~/perl/third 03:53:33]$ perltidy -b slurp
[hood@panix3 ~/perl/third 03:53:44]$ perlcritic -l slurp
Version string used at line 3, column 1. Use a real number instead. (Severity: 3)
Code before warnings are enabled at line 6, column 1. See page 431 of PBP. (Severity: 4)
Magic punctuation variable $/ used at line 10, column 23. See page 79 of PBP. (Severity: 2)
[hood@panix3 ~/perl/third 03:53:50]$ cat slurp
#!/net/u/11/hood/local/bin/perl
use v5.16.2;
use Carp;
our $VERSION = 1.0;
open my $fh, '<', 'test.txt' or croak 'cannot open';
my $file = do { local $/ = <$fh> };
close $fh or croak 'cannot close';
say $file or croak 'cannot say';
[hood@panix3 ~/perl/third 03:54:15]$ ./slurp
hello, i love you, won't you tell me your name ...
jump in the game
blah blah blah ...
light my fire
pyre
heart of darkness things ...
[hood@panix3 ~/perl/third 03:54:21]$ 
```

Schwartzzian Transform ...
Chapter 11.
What to Avoid

~.1 Barewords

~.2 Indirect Objects
~.3 Prototypes

Few uses of prototypes are compelling enough to overcome their drawbacks, but they exist. First, they can allow you to override builtins. First check that you can override the builtin by examining its prototype in a small test program. Then use the subs pragma to tell Perl that you plan to override a builtin, and finally declare your override with the correct prototype:

```perl
use subs 'push';
sub push (@@) { ... }
```

Beware that the subs pragma is in effect for the remainder of the file, regardless of any lexical scoping.

The second reason to use prototypes is to define compile-time constants. When Perl encounters a function declared with an empty prototype (as opposed to no prototype) and this function evaluates to a single constant expression, the optimizer will turn all calls to that function into constants instead of function calls:

```perl
sub PI () { 4 * atan2(1, 1) }
```
~.4 Method-Function Equivalence

Because a class is a package, Perl does not distinguish between a function and a method stored in a package. The same builtin, sub, declares both. Documentation can clarify your intent, but Perl will happily dispatch to a function called as a method. Likewise, you can invoke a method as if it were a function—fully-qualified, exported, or as a reference—if you pass in your own invocant manually

```perl
my $price = $o->calculate_price();
# broken; do not use
my $price = Order::calculate_price( $o );
```

~.5 Tie:

“Tied variables seem like fun opportunities for cleverness, but they can produce confusing interfaces. Unless you have a very good reason for making objects behave as if they were builtin data types, avoid creating your own ties. tie is also much slower than using the builtin types due to various reasons of implementation.

Good reasons include to ease debugging (use the logged scalar to help you understand where a value changes) and to make certain impossible operations possible (accessing large files in a memory-efficient way)”
Chapter 12.
What’s Missing

~.1 non-core CPAN
Many of the best Perl 5 modules exist on the CPAN and not in the core. The Task::Kensho metadistribution includes several other distributions which represent the best the CPAN has to offer. When you need to solve a problem, look there first.

~.2 warnings
Combine use warnings with use diagnostics to receive expanded diagnostic messages for each warning present in your programs. These expanded diagnostics come from perldoc perldiag.

~.3 strict
strict forbids symbolic references, requires variable declarations (Lexical Scope, pp. 79), and prohibits the use of undeclared barewords

~.4 autodie
Perl 5 leaves error handling (or error ignoring) up to you.
Chapter 6.
Regular Expressions and Matching
Regular Expressions and Matching

Perl's text processing power comes from its use of regular expressions. A regular expression (regex or regexp) is a pattern which describes characteristics of a piece of text. A regular expression engine interprets patterns and applies them to match or modify pieces of text.

Perl's core regex documentation includes a tutorial (perldoc perlretut), a reference guide (perldoc perlreref), and full documentation (perldoc perlre). Jeffrey Friedl's book Mastering Regular Expressions explains the theory and the mechanics of how regular expressions work. While mastering regular expressions is a daunting pursuit, a little knowledge will give you great power.

Literals

Regexes can be as simple as substring patterns:

```perl
my $name = 'Chatfield';
say 'Found a hat!' if $name =~ /hat/;
```

The match operator (/m/, abbreviated //) identifies a regular expression—in this example, hat. This pattern is not a word. Instead it means “the h character, followed by the a character, followed by the t character.” Each character in the pattern is an indivisible element, or atom. It matches or it doesn't.

The regex binding operator (=~) is an infix operator (Fixity, pp. 66) which applies the regex of its second operand to a string provided by its first operand. When evaluated in scalar context, a match evaluates to a true value if it succeeds. The negated form of the binding operator (!~) evaluates to a true value unless the match succeeds.

The index builtin can also search for a literal substring within a string. Using a regex engine for that is like flying your autonomous combat helicopter to the corner store to buy cheese—but Perl allows you to decide what you find most maintainable.

```
Remember index!
```

The substitution operator, s///, is in one sense a circumfix operator (Fixity, pp. 66) with two operands. Its first operand is a regular expression to match when used with the regex binding operator. The second operand is a substring used to replace the matched portion of the first operand used with the regex binding operator. For example, to cure pesky summer allergies:

```perl
my $status = 'I feel ill.';
$status =~ s/ill/well/;
say $status;
```

The qr// Operator and Regex Combinations

The qr// operator creates first-class regexes. Interpolate them into the match operator to use them:
my $hat = qr/hat/;
say 'Found a hat!' if $name =~ /$hat/;

... or combine multiple regex objects into complex patterns:

my $hat = qr/hat/;
my $field = qr/field/;

say 'Found a hat in a field!' if $name =~ /$hat$field/;

like( $name, qr/$hat$field/,
     'Found a hat in a field!' );

Test::More's like function tests that the first argument matches the regex provided as the second argument.

Quantifiers

Regular expressions get more powerful through the use of regex quantifiers, which allow you to specify how often a regex component may appear in a matching string. The simplest quantifier is the zero or one quantifier, or ?:

my $cat_or_ct = qr/ca?t/;

like( 'cat', $cat_or_ct, "'cat' matches /ca?t/ ");
like( 'ct', $cat_or_ct, "'ct' matches /ca?t/ ");

Any atom in a regular expression followed by the ? character means “match zero or one of this atom.” This regular expression matches if zero or one a characters immediately follow a c character and immediately precede a t character, either the literal substring cat or ct.

The one or more quantifier, or +, matches only if there is at least one of the quantified atom:

my $some_a = qr/ca+t/;

like( 'cat', $some_a, "'cat' matches /ca+t/ ");
like( 'caat', $some_a, "'caat' matches ");
like( 'caaat', $some_a, "'caaat' matches ");
like( 'caaaaat', $some_a, "'caaaaat' matches ");
unlike( 'ct', $some_a, "'ct' does not match" );

There is no theoretical limit to the maximum number of quantified atoms which can match.

The zero or more quantifier, *, matches zero or more instances of the quantified atom:

my $any_a = qr/ca*?/;

like( 'cat', $any_a, "'cat' matches /ca*?/ ");
like( 'caat', $any_a, "'caat' matches" );
like( 'caaat', $any_a, "'caaat' matches" );
like( 'caaaaat', $any_a, "'caaaaat' matches" );
like( 'ct', $any_a, "'ct' matches" );
As silly as this seems, it allows you to specify optional components of a regex. Use it sparingly, though: it’s a blunt and expensive tool. Most regular expressions benefit from using the `?` and `+` quantifiers far more than `*`. Precision of intent often improves clarity.

**Numeric quantifiers** express specific numbers of times an atom may match. `{n}` means that a match must occur exactly \( n \) times.

\[
\begin{align*}
\{n\} & \quad \text{matches an atom at least } n \text{ times:} \\
\{n,m\} & \quad \text{means that a match must occur at least } n \text{ times and cannot occur more than } m \text{ times:}
\end{align*}
\]

You may express the symbolic quantifiers in terms of the numeric quantifiers, but most programs use the former far more often than the latter.

### Greediness

The `+` and `*` quantifiers are greedy, as they try to match as much of the input string as possible. This is particularly pernicious. Consider a naïve use of the “zero or more non-newline characters” pattern of `.*`:

\[
\begin{align*}
\# \text{ a poor regex} \\
\text{my } \$\text{hot_meal} & = qr/hot.*\text{meal}/; \\
\text{say 'Found a hot meal!' } \\
& \quad \text{if 'I have a hot meal' =~ $\text{hot_meal};} \\
\text{say 'Found a hot meal!' } \\
& \quad \text{if 'one-shot, piecemeal work!' =~ $\text{hot_meal;}}
\end{align*}
\]

Greedy quantifiers start by matching *everything* at first, and back off a character at a time only when it’s obvious that the match will not succeed.

The `?` quantifier modifier turns a greedy-quantifier parsimonious:

\[
\begin{align*}
\text{my } \$\text{minimal_greedy} & = qr/hot.*?\text{meal}/;
\end{align*}
\]
When given a non-greedy quantifier, the regular expression engine will prefer the shortest possible potential match and will increase the number of characters identified by the \.*? token combination only if the current number fails to match. Because \* matches zero or more times, the minimal potential match for this token combination is zero characters:

    say 'Found a hot meal'
    if 'ilikeahotmeal' =~ /$minimal_greedy/;

Use +? to match one or more items non-greedily:

    my $minimal_greedy_plus = qr/hot.+?meal/;
    unlike( 'ilikeahotmeal', $minimal_greedy_plus );
    like( 'i like a hot meal', $minimal_greedy_plus );

The ? quantifier modifier also applies to the ? (zero or one matches) quantifier as well as the range quantifiers. In every case, it causes the regex to match as little of the input as possible.

The greedy patterns \.+ and \.* are tempting but dangerous. A cruciverbalist\(^1\) who needs to fill in four boxes of 7 Down ("Rich soil") will find too many invalid candidates with the pattern:

    my $seven_down = qr/l$letters_only *m/;

She'll have to discard Alabama, Belgium, and Bethlehem long before the program suggests loam. Not only are those words too long, but the matches start in the middle of the words. A working understanding of greediness helps, but there is no substitute for the copious testing with real, working data.

**Regex Anchors**

*Regex anchors* force the regex engine to start or end a match at an absolute position. The *start of string anchor* (\A) dictates that any match must start at the beginning of the string:

    # also matches "lamed", "lawmaker", and "layman"
    my $seven_down = qr/\A$letters_only\{2\}m/;

The *end of line string anchor* (\Z) requires that a match end at the end of a line within the string.

    # also matches "loom", but an obvious improvement
    my $seven_down = qr/\A$letters_only\{2\}m\Z/;

The *word boundary anchor* (\b) matches only at the boundary between a word character (\w) and a non-word character (\W).

Use an anchored regex to find loam while prohibiting Belgium:

    my $seven_down = qr/\b$letters_only\{2\}m\b/;

\(^1\) A crossword puzzle aficionado.
Metacharacters

Perl interprets several characters in regular expressions as metacharacters, characters represent something other than their literal interpretation. Metacharacters give regex wielders power far beyond mere substring matches. The regex engine treats all metacharacters as atoms.

The . metacharacter means “match any character except a newline”. Remember that caveat; many novices forget it. A simple regex search—ignoring the obvious improvement of using anchors—for 7 Down might be /l\.m/. Of course, there’s always more than one way to get the right answer:

```perl
for my $word (@words)
{
    next unless length( $word ) == 4;
    next unless $word =~ /l..m/;
    say "Possibility: $word";
}
```

If the potential matches in @words are more than the simplest English words, you will get false positives. . also matches punctuation characters, whitespace, and numbers. Be specific! The \w metacharacter represents all alphanumeric characters (Unicode and Strings, pp. 18) and the underscore:

```perl
next unless $word =~ /l\w\w\m/;
```

The \d metacharacter matches digits (also in the Unicode sense):

```perl
# not a robust phone number matcher
next unless $number =~ /\d{3}-\d{3}-\d{4}/;
say "I have your number: $number";
```

Use the \s metacharacter to match whitespace, whether a literal space, a tab character, a carriage return, a form-feed, or a newline:

```perl
my $two_three_letter_words = qr/\w{3}\s\w{3}/;
```

Negated Metacharacters

These metacharacters have negated forms. Use \W to match any character except a word character. Use \D to match a non-digit character. Use \S to match anything but whitespace. Use \B to match anywhere except a word boundary.

Character Classes

When none of those metacharacters is specific enough, specify your own character class by enclosing them in square brackets:

```perl
my $ascii_vowels = qr/[aeiou]/;
my $maybe_cat = qr/c${ascii_vowels}t/;
```

Without those curly braces, Perl's parser would interpret the variable name as $ascii_vowelst, which either causes a compile-time error about an unknown variable or interpolates the contents of an existing $ascii_vowelst into the regex.
The hyphen character (-) allows you to specify a contiguous range of characters in a class, such as this \$ascii_letters_only regex:

```
my \$ascii_letters_only = qr/[a-zA-Z]/;
```

To include the hyphen as a member of the class, move it to the start or end:

```
my \$interesting_punctuation = qr/[-!?]/;
```

... or escape it:

```
my \$line_characters = qr/\[|=\-\]/;
```

Use the caret (^) as the first element of the character class to mean “anything except these characters”:

```
my \$not_an_ascii_vowel = qr/[^aeiou]/;
```

### Metacharacters in Character Classes

Use a caret anywhere but the first position to make it a member of the character class. To include a hyphen in a negated character class, place it after the caret or at the end of the class, or escape it.

---

**Capturing**

Regular expressions allow you to group and capture portions of the match for later use. To extract an American telephone number of the form (202) 456-1111 from a string:

```
my \$area_code = qr/\(\d{3}\)/;
my \$local_number = qr/\d{3}-?\d{4}/;
my \$phone_number = qr/$area_code\s?$local_number/;
```

Note especially the escaping of the parentheses within \$area_code. Parentheses are special in Perl 5 regular expressions. They group atoms into larger units and also capture portions of matching strings. To match literal parentheses, escape them with backslashes as seen in \$area_code.

**Named Captures**

Perl 5.10 added named captures, which allow you to capture portions of matches from applying a regular expression and access them later, such as finding a phone number in a string of contact information:

```
if ($contact_info =~ /(?<phone>$phone_number)/) {
    say "Found a number $\+(phone)";
}
```

Regexes tend to look like punctuation soup until you can group various portions together as chunks. Named capture syntax has the form:

```
(?<capture name> ... )
```

Parentheses enclose the capture. The ?< name > construct names this particular capture and must immediately follow the left parenthesis. The remainder of the capture is a regular expression.

When a match against the enclosing pattern succeeds, Perl stores the portion of the string which matches the enclosed pattern in the magic variable %+. In this hash, the key is the name of the capture and the value is the appropriate portion of the matched string.
Chapter 6. Regular Expressions and Matching

**Numbered Captures**

Perl has supported *numbered captures* for ages:

```perl
if ($contact_info =~ /($phone_number)/)
{
    say "Found a number $1";
}
```

This form of capture provides no identifying name and does not store in `%+`. Instead, Perl stores the captured substring in a series of magic variables. The *first* matching capture that Perl finds goes into `$1`, the second into `$2`, and so on. Capture counts start at the *opening* parenthesis of the capture; thus the first left parenthesis begins the capture into `$1`, the second into `$2`, and so on.

While the syntax for named captures is longer than for numbered captures, it provides additional clarity. Counting left parentheses is tedious work, and combining regexes which each contain numbered captures is far too difficult. Named captures improve regex maintainability—though name collisions are possible, they’re relatively infrequent. Minimize the risk by using named captures only in top-level regexes.

In list context, a regex match returns a list of captured substrings:

```perl
if (my ($number) = $contact_info =~ /($phone_number)/)
{
    say "Found a number $number";
}
```

Numbered captures are also useful in simple substitutions, where named captures may be more verbose:

```perl
my $order = 'Vegan brownies!';

$order =~ s/Vegan \w+/Vegetarian $1/;
# or
$order =~ s/Vegan (?<food>\w+)/Vegetarian $+{food}/;
```

**Grouping and Alternation**

Previous examples have all applied quantifiers to simple atoms. You may apply them to any regex element:

```perl
my $pork = qr/pork/;
my $beans = qr/beans/;
like( 'pork and beans', qr/\A$pork?.*?$beans/, 'maybe pork, definitely beans' );
```

If you expand the regex manually, the results may surprise you:

```perl
my $pork_and_beans = qr/\Apork?.*beans/;
like( 'pork and beans', qr/$pork_and_beans/, 'maybe pork, definitely beans' );
like( 'por and beans', qr/$pork_and_beans/, 'wait... no phylloquinone here!' );
```

Sometimes specificity helps pattern accuracy:
my $pork = qr/pork/;
my $and = qr/and/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\$pork? \$and? \$beans/, 'maybe pork, maybe and, definitely beans' );

Some regexes need to match either one thing or another. The * alternation* metacharacter (|) expresses this intent:

my $rice = qr/rice/;
my $beans = qr/beans/;

like( 'rice', qr/$rice|$beans/, 'Found rice' );
like( 'beans', qr/$rice|$beans/, 'Found beans' );

The alternation metacharacter indicates that either preceding fragment may match. Keep in mind that alternation has a lower precedence (Precedence, pp. 65) than even atoms:

like( 'rice', qr/rice|beans/, 'Found rice' );
like( 'beans', qr/rice|beans/, 'Found beans' );

unlike( 'ricb', qr/rice|beans/, 'Found hybrid' );

While it's easy to interpret rice|beans as meaning ric, followed by either e or b, followed by eans, alternations always include the *entire* fragment to the nearest regex delimiter, whether the start or end of the pattern, an enclosing parenthesis, another alternation character, or a square bracket.

To reduce confusion, use named fragments in variables ($rice|$beans) or group alternation candidates in *non-capturing groups*:

my $starches = qr/(?:pasta|potatoes|rice)/;

The (?) sequence groups a series of atoms without making a capture.

Other *Escape Sequences*

To match a *literal* instance of a metacharacter, *escape* it with a backslash (\). You've seen this before, where \( refers to a single left parenthesis and \] refers to a single right square bracket. \. refers to a literal period character instead of the “match anything but an explicit newline character” atom.

You will likely need to escape the alternation metacharacter (|) as well as the end of line metacharacter ($) and the quantifiers (+, *, ?).

The *metacharacter disabling characters* (\Q and \E) disable metacharacter interpretation within their boundaries. This is especially useful when taking match text from a source you don't control when writing the program:

my ($text, $literal_text) = @_;

return $text =~ /\Q$literal_text\E/;
The $literal_text argument can contain anything—the string ** ALERT **, for example. Within the fragment bounded by \Q and \E, Perl interpret the regex as \*\* ALERT \*\* and attempt to match literal asterisk characters, rather than greedy quantifiers.

**Regex Security**

Be cautious when processing regular expressions from untrusted user input. A malicious regex master can craft a denial-of-service attack against your program.

### Assertions

Regex anchors such as \A, \b, \B, and \Z are a form of regex assertion, which requires that the string meet some condition. These assertions do not match individual characters within the string. No matter what the string contains, the regex qr/\A/ will always match.

Zero-width assertions match a pattern. Most importantly, they do not consume the portion of the pattern that they match. For example, to find a cat on its own, you might use a word boundary assertion:

```perl
my $just_a_cat = qr/cat\b/;
```

...but if you want to find a non-disastrous feline, you might use a zero-width negative look-ahead assertion:

```perl
my $safe_feline = qr/cat(?!astrophe)/;
```

The construct (?!...) matches the phrase cat only if the phrase astrophe does not immediately follow. The zero-width positive look-ahead assertion:

```perl
my $disastrous_feline = qr/cat(?=astrophe)/;
```

...matches the phrase cat only if the phrase astrophe immediately follows. While a normal regular expression can accomplish the same thing, consider a regex to find all non-catastrophic words in the dictionary which start with cat:

```perl
my $disastrous_feline = qr/cat(?!astrophe)/;
while (<$words>)
{
    chomp;
    next unless /\A(?<cat>$disastrous_feline.*)\Z/;
    say "Found a non-catastrophe '$+{cat}'";
}
```

The zero-width assertion consumes none of the source string, leaving the anchored fragment <.*\Z> to match. Otherwise, the capture would only capture the cat portion of the source string.

To assert that your feline never occurs at the start of a line, you might use a zero-width negative look-behind assertion. These assertions must have fixed sizes; you may not use quantifiers:

```perl
my $middle_cat = qr/(?<!\A)cat/;
```

The construct (?<!...) contains the fixed-width pattern. You could also express that the cat must always occur immediately after a space character with a zero-width positive look-behind assertion:
my $space_cat = qr/(?<=\s)cat/;

The construct (?<=...) contains the fixed-width pattern. This approach can be useful when combining a global regex match with the \G modifier, but it's an advanced feature you likely won't use often.

A newer feature of Perl 5 regexes is the keep assertion \K. This zero-width positive look-behind assertion can have a variable length:

my $spacey_cat = qr/\s+\Kcat/;

like( 'my cat has been to space', $spacey_cat );
like( 'my cat has been to doublespace', $spacey_cat );

\K is surprisingly useful for certain substitutions which remove the end of a pattern:

my $exclamation = 'This is a catastrophe!';
$exclamation =~ s/cat\K\w+!/./;

like( $exclamation, qr/\bcat\./, "That wasn't so bad!" );

### Regex Modifiers

Several modifiers change the behavior of the regular expression operators. These modifiers appear at the end of the match, substitution, and qr// operators. For example, to enable case-insensitive matching:

my $pet = 'CaMeLiA';

like( $pet, qr/Camelia/, 'Nice butterfly!' );
like( $pet, qr/Camelia/i, 'shift key br0ken' );

The first like() will fail, because the strings contain different letters. The second like() will pass, because the /i modifier causes the regex to ignore case distinctions. M and m are equivalent in the second regex due to the modifier.

You may also embed regex modifiers within a pattern:

my $find_a_cat = qr/(?<feline>(?i)cat)/;

The (?i) syntax enables case-insensitive matching only for its enclosing group: in this case, the named capture. You may use multiple modifiers with this form. Disable specific modifiers by preceding them with the minus character (-):

my $find_a_rational = qr/(?<number>(?-i)Rat)/;

The multiline operator, /m, allows the \A and \Z anchors to match at any newline embedded within the string.

The /s modifier treats the source string as a single line such that the . metacharacter matches the newline character. Damian Conway suggests the mnemonic that /m modifies the behavior of multiple regex metacharacters, while /s modifies the behavior of a single regex metacharacter.

The /r modifier causes a substitution operation to return the result of the substitution, leaving the original string as-is. If the substitution succeeds, the result is a modified copy of the original. If the substitution fails (because the pattern does not match), the result is an unmodified copy of the original: 106
my $status = 'I am hungry for pie.';
my $newstatus = $status =~ s/pie/cake/r;
my $statuscopy = $status
    =~ s/liver and onions/bratwurst/r;

is( $status, 'I am hungry for pie.',
    'original string should be unmodified' );

like( $newstatus, qr/cake/, 'cake wanted' );
unlike( $statuscopy, qr/bratwurst/, 'wurst not' );

The /x modifier allows you to embed additional whitespace and comments within patterns. With this modifier in effect, the regex engine ignores whitespace and comments. The results are often much more readable:

my $attr_re = qr{
\A # start of line
(?:
    [;\n\s]* # spaces and semicolons
    (?:/\*.*?\*/)? # C comments
)*
ATTR
\s+
    ( U?INTVAL
    | FLOATVAL
    | STRING\s+\*
  )
}x;

This regex isn’t simple, but comments and whitespace improve its readability. Even if you compose regexes together from compiled fragments, the /x modifier can still improve your code.

The /g modifier matches a regex globally throughout a string. This makes sense when used with a substitution:

# appease the Mitchell estate
my $contents = slurp( $file );
$contents =~ s/Scarlett O'Hara/Mauve Midway/g;

When used with a match—not a substitution—the \G metacharacter allows you to process a string within a loop one chunk at a time. \G matches at the position where the most recent match ended. To process a poorly-encoded file full of American telephone numbers in logical chunks, you might write:

while ($contents =~ /\G(\w{3})(\w{3})(\w{4})/g)
{
    push @numbers, "($1) $2-$3";
}

Be aware that the \G anchor will take up at the last point in the string where the previous iteration of the match occurred. If the previous match ended with a greedy match such as .*, the next match will have less available string to match. Lookahead assertions can also help.

The /e modifier allows you to write arbitrary Perl 5 code on the right side of a substitution operation. If the match succeeds, the regex engine will run the code, using its return value as the substitution value. The earlier global substitution example could be simpler with code like:
Modern Perl

# appease the Mitchell estate
$sequel =~ s{Scarlett( O'Hara)?}{
    'Mauve' . defined $1 ? ' Midway' : ''
}ge;

Each additional occurrence of the /e modifier will cause another evaluation of the result of the expression, though only Perl golfers use anything beyond /ee.

Smart Matching

The smart match operator, ~~, compares two operands and returns a true value if they match. The fuzziness of the definition demonstrates the smartness of the operator: the type of comparison depends on the type of both operands. *given* (Given/When, pp. 36) performs an implicit smart match.

The smart match operator is an infix operator:

    say 'They match (somehow)' if $loperand ~~ $roperand;

The type of comparison generally depends first on the type of the right operand and then on the left operand. For example, if the right operand is a scalar with a numeric component, the comparison will use numeric equality. If the right operand is a regex, the comparison will use a grep or a pattern match. If the right operand is an array, the comparison will perform a grep or a recursive smart match. If the right operand is a hash, the comparison will check the existence of one or more keys. A large and intimidating chart in perldoc perlsyn gives far more details about all the comparisons smart match can perform.

A serious proposal for 5.16 suggests simplifying smart match substantially. The more complex your operands, the more likely you are to receive confusing results. Avoid comparing objects and stick to simple operations between two scalars or one scalar and one aggregate for the best results.

With that said, smart match can be useful:

    my ($x, $y) = (10, 20);
    say 'Not equal numerically' unless $x ~~ $y;

    my $z = '10 little endians';
    say 'Equal numeric-ishally' if $x ~~ $z;

    # regular expression match
    my $needle = qr/needle/;
    say 'Pattern match' if 'needle' ~~ $needle;
    say 'Grep through array' if @haystack ~~ $needle;
    say 'Grep through hash keys' if %hayhash ~~ $needle;
    say 'Grep through array' if $needle ~~ @haystack;
    say 'Array elements exist as hash keys'
        if %hayhash ~~ @haystack;
    say 'Smart match elements' if @straw ~~ @haystack;

108
say 'Grepl through hash keys' if $needle ~~ %hayhash;

say 'Array elements exist as hash keys'
    if @haystack ~~ %hayhash;

say 'Hash keys identical' if %hayhash ~~ %haymap;

Smart match works even if one operand is a *reference* to the given data type:

say 'Hash keys identical' if %hayhash ~~ \%hayhash;
Term Rewriting Systems

Terese
We start the book by presenting some basic examples of rewriting, in order to set the stage.

At school, many of us have been drilled to simplify arithmetical expressions, for instance:

\[(3 + 5) \cdot (1 + 2) \rightarrow 8 \cdot (1 + 2) \rightarrow 8 \cdot 3 \rightarrow 24\]

This simplification process has several remarkable properties.

First, one can perceive a direction, at least in the drill exercises, from complicated expressions to simpler ones. For this reason we use $\rightarrow$ rather than $=$, even though the expressions are equal in the sense that they all denote the same number (24). The relation $\rightarrow$ is called a reduction relation.

Second, in most drill exercises the simplification process yields a result in the form of an expression that cannot be simplified any further, which we call a normal form. In the above example the result 24 is such a normal form.

Third, the simplification process is non-deterministic, often different simplifications are possible. It is clearly desirable that all simplifications lead to the same result. Indeed the above outcome 24 can be obtained in different ways, e.g. also by

\[(3 + 5) \cdot (1 + 2) \rightarrow (3 + 5) \cdot 3 \rightarrow 3 \cdot 3 + 5 \cdot 3 \rightarrow 9 + 5 \cdot 3 \rightarrow 9 + 15 \rightarrow 24\]

The property that simplifications can have at most one final result is called uniqueness of normal form.

The process of simplifying arithmetical expressions built up from numbers with operations like + and $\cdot$ can be analysed (and taught) as performing elementary steps in contexts. The elementary steps are based on the tables of addition and multiplication. The contexts are arithmetical expressions with a hole, denoted by $\square$, indicating the place where the elementary step is to take place. In the first example above, the elementary step $3 + 5 \rightarrow 8$ is performed in the context $\square \cdot (1 + 2)$. This means that $\square$ is $3 + 5$ before, and 8 after, the elementary step, yielding the simplification $(3 + 5) \cdot (1 + 2) \rightarrow 8 \cdot (1 + 2)$. (Brackets are not part of the term, but are auxiliary symbols preventing false readings of the term.) The next elementary step is $1 + 2 \rightarrow 3$, performed in the context $8 \cdot \square$, yielding the simplification $8 \cdot (1 + 2) \rightarrow 8 \cdot 3$. Finally, the elementary step $8 \cdot 3 \rightarrow 24$ is performed in the context $\square$, as this elementary step involves the whole term.
Among our early experiences are, besides arithmetical simplification, also drill exercises that do not yield a final result, such as counting:

\[ 1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots \]

Another example is the cyclic elapsing of the hours on a clock:

\[ 1 \rightarrow 2 \rightarrow \cdots \rightarrow 11 \rightarrow 12 \rightarrow 1 \rightarrow \cdots \]

The absence of a normal form is called \textit{non-termination}. In many applications termination is a desirable property. The following example shows that termination can be a non-trivial question.

Define a reduction relation on positive integer numbers

\[ n \rightarrow n' \]

by putting \( n' = n/2 \) if \( n > 1 \) is even, and \( n' = 3n + 1 \) if \( n > 1 \) is odd. Thus 1 is the only normal form. We have reductions such as

\[
\begin{align*}
7 & \rightarrow 22 \rightarrow 11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \rightarrow 26 \rightarrow 13 \\
& \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1
\end{align*}
\]

Note that we have both decrease (in the case \( n/2 \)) and increase (in the case \( 3n + 1 \)). It is an open question\textsuperscript{1} whether or not every positive integer can be reduced to 1 in this way.

This book is about the theory of stepwise, or discrete, transformations of objects, as opposed to continuous transformations of objects. Many computations, constructions, processes, translations, mappings and so on, can be modelled as stepwise transformations of objects. Clearly, this yields a large spectrum of applications, depending on what are the objects of interest and what transformations, or \textit{rewriting}, one wishes to do. One has string rewriting, term rewriting, graph rewriting, to name some of the principal subjects that will be treated. In turn, these main subjects lead to several specialized theories, such as conditional term rewriting, infinitary term rewriting, term graph rewriting, and many more. In all these different branches of rewriting the basic concepts are the same, and known as termination (guaranteeing the existence of normal forms) and confluence (securing the uniqueness of normal forms), and many variations of them.

In order to appreciate the variety of applications and to further introduce the main concepts, let us view a few examples from different fields. We will return to some of these examples in future chapters.

\textsuperscript{1}Collatz's problem, also known as the Syracuse problem.
0.1. An example from functional programming

Rewriting techniques can be used to specify operations on abstract data types. The first introductory example, taken from Arts [1997], describes in a concise way an algorithm for dividing natural numbers, giving for all pairs \((m, n)\) with \(n \geq 1\) and \(m = n \cdot q \geq 0\) the correct result \(q\). The abstract data type in question is that of the natural numbers built up with \(0 : \mathbb{N}\) and \(S : \mathbb{N} \to \mathbb{N}\). We write \(0\) for \(0\), \(1\) for \(S(0)\), \(2\) for \(S(S(0))\), and so on. There are four rewrite rules:

\[
\begin{align*}
\text{minus}(x, 0) & \rightarrow x \\
\text{minus}(S(x), S(y)) & \rightarrow \text{minus}(x, y) \\
\text{quot}(0, S(y)) & \rightarrow 0 \\
\text{quot}(S(x), S(y)) & \rightarrow S(\text{quot}(\text{minus}(x, y), S(y)))
\end{align*}
\]

As an example we exhibit the following reduction sequence.

\[
\begin{align*}
\text{quot}(4, 2) & \rightarrow S(\text{quot}(\text{minus}(3, 1), 2)) \\
& \rightarrow S(\text{quot}(\text{minus}(2, 0), 2)) \\
& \rightarrow S(\text{quot}(2, 2)) \\
& \rightarrow S(S(\text{quot}(\text{minus}(1, 1), 2))) \\
& \rightarrow S(S(\text{quot}(\text{minus}(0, 0), 2))) \\
& \rightarrow S(S(\text{quot}(0, 2))) \\
& \rightarrow S(S(0)) = 2
\end{align*}
\]

Normal forms are \(0, 1, 2, \ldots\), but also \(\text{minus}(0, 1)\) and \(\text{quot}(2, 0)\). For the correctness of such algorithms with respect to the operations on the abstract data type, it is obviously desirable that the algorithms have unique results. This is guaranteed if every reduction sequence eventually terminates, and moreover the resulting normal form is independent of the particular choice of reduction sequence. Later on in this book we will develop methods to facilitate the proofs of such properties.

0.2. An example from topology: Reidemeister moves

In this example the objects are knots. For our purposes it suffices to state that a knot is a piece of (flexible) wire whose ends coincide. We assume knots to be two-dimensional, lying on a flat surface such as a kitchen table, with crossings only in one single point. In case of a crossing, it is always clear which of the two parts of the wire involved is the upper and which is the lower (in the pictures, the lower wire is interrupted just before and after the crossing).

Knots are considered to be equivalent when they can be transformed into one another in a continuous way, that is, without breaking the wire. As a
consequence, knots could be taken as equivalence classes, but here we are interested in the equivalence relation itself. There are some well-known elementary transformations, known as the ‘Reidemeister moves’, such that two knots are equivalent if and only if they can be transformed into one another using only Reidemeister moves.

Figure 1 depicts the Reidemeister moves, Figure 2 gives a transformation between a certain knot and the trivial knot, also called the ‘un-knot’. Although the un-knot is the simplest knot, it should not be considered as a normal form: the Reidemeister moves can be used in both directions. There is no use for the concept of normal form in this example. Not every knot can be transformed into the un-knot; there are in fact infinitely many different knots.

0.3. An example from logic: tautology checking

This example exploits the well-known equivalence between boolean algebras and boolean rings (rings with $x^2 = x$). In the setting of boolean algebras $\Rightarrow, \lor, \neg$ are the usual propositional connectives; $+$ is exclusive disjunction and $\cdot$ is conjunction. In the setting of rings $+$ is the ring addition, $\cdot$ the ring multiplication. Furthermore, 0 stands for the boolean false as well as for the additive unit, and 1 stands for the boolean true as well as for the multiplicative unit. The operators $+$ and $\cdot$ are supposed to be commutative and associative.

Now we have the remarkable fact that every propositional formula is a tautology if and only if it can be rewritten to 1 using the rewrite rules be-
low, *modulo associativity and commutativity*. This means that rewrite steps may be preceded by rearranging the order and the bracket structure of sums and products. (In fact this constitutes an example of rewriting modulo an equivalence relation, made precise in Section 14.3.)

The above equivalence and the rewrite rules already appear in Herbrand’s PhD thesis of 1929. The symbol $\rightarrow$ is used for a rewrite step, and should not be confused with $\Rightarrow$ which stands for implication.

\[
\begin{align*}
x \Rightarrow y & \rightarrow x \cdot y + x + 1 \\
x \lor y & \rightarrow x \cdot y + x + y \\
\neg x & \rightarrow x + 1 \\
x + 0 & \rightarrow x \\
x + x & \rightarrow 0 \\
x \cdot 0 & \rightarrow 0 \\
x \cdot 1 & \rightarrow x \\
x \cdot x & \rightarrow x \\
x \cdot (y + z) & \rightarrow x \cdot y + x \cdot z
\end{align*}
\]

As an example we exhibit the following reduction of the tautology $p \Rightarrow (q \Rightarrow p)$ to 1, where the most relevant associative(commutative) steps are also stipulated. As usual in algebra, we omit the multiplication sign $\cdot$.

\[
\begin{align*}
p \Rightarrow (q \Rightarrow p) & \rightarrow p(q \Rightarrow p) + p + 1 \\
& \rightarrow p(qp + q + 1) + p + 1 = p((qp + q) + 1) + p + 1 \\
& \rightarrow p(qp + q) + p1 + p + 1 \\
& \rightarrow pqp + pq + p1 + p + 1
\end{align*}
\]
It is not our goal to give a correctness proof of this algorithm for tautology checking. Note, once more, that rewriting is non-deterministic: we could have started in the example above with the rightmost implication in \( p \Rightarrow (q \Rightarrow p) \). It is necessary for the correctness of the algorithm that any reduction sequence starting with \( p \Rightarrow (q \Rightarrow p) \) yields the outcome 1. This will be guaranteed by a notion called *confluence*, formulated for arbitrary abstract reduction systems in the chapter that follows now.
We’ve seen that Scheme is, as languages go, pretty small. There are just a few keywords, and most of the utility of the language is inherent in its minimal, unornamented structure, unlike, say, “public static void main” Java. We can come up with a short list of fundamental Scheme concepts:

```
if  lambda  cons  define  car
    cdr    numbers arithmetic symbols true false
```

Without motivating it too deeply, let’s consider how we could reduce this list further, by implementing some of these features in terms of others.

**Warming up: booleans and conditionals**

In Scheme, what is the basic way of making conditional statements?

```
(if  C  T  F)
```

There is a form that begins with the keyword `if`, and has three arguments, a test expression `C`, and things to do depending on the value of `C`. `C` is a boolean, and as such can have two possible values, true and false. How might we represent these? Well, these values are making a choice between two values, so why not pass in the values and let them choose?

```
true  ≡  (λ(T  F)  T)
false ≡  (λ(T  F)  F)
```
Thus true and false are functions with an arity\(^1\) of \(2^2\) that select one or the other of their arguments. \(\text{if}\) \(\text{then}\) is simply

\[
(\text{if} \ C \ T \ F) \equiv (C \ T \ F)
\]

which applies its first argument to the other two. Note that for the usual ‘short-circuiting’ behaviour of \(\text{if}\) to work (and it is necessary, in order to properly terminate recursion, avoid dividing by zero, etc.), evaluation must be lazy, that is, we don’t actually evaluate an expression until we need its value, e.g. to perform arithmetic on it. Since we never need a value from the unused half of the conditional, we never try to evaluate it and the short-circuiting behaviour is preserved.

We’ve thus eliminated \(\text{if}\), \(\text{true}\), and \(\text{false}\) from our list. Thus emboldened, we move on to

Lists

What, really, is a Scheme list? It’s a pair, with the first value of the pair representing the first item of the list, and the second half of the pair containing the rest of the list. To use standard LISPish terminology,

\[
\begin{align*}
(\text{car} \ (\text{cons} \ A \ B)) & \equiv A \\
(\text{cdr} \ (\text{cons} \ A \ B)) & \equiv B
\end{align*}
\]

So how can we represent a cons? If we think in an object-oriented way for a moment\(^5\), we realise that a list, or a pair, is something that we want to feed two values into, and then be able to pass messages to in order to get those values back out. Well, we can feed two values in as arguments—

\[
(\lambda(a \ b))
\]

—and then return something that reads messages and uses them to act on the data—

\[
(\lambda(m)(m \ a \ b)).
\]

---

1The “arity” of a function is simply a fancy word for the number of arguments it takes. Derived from ‘binary’, ‘ternary’, ‘n-ary’.
2We can simplify further and make \(\lambda\) only create functions of one argument by currying\(^3\) the arguments; for legibility and convenience we will not be doing this here.
3To ‘curry’ a function of \(n\) arguments, one makes a function of 1 argument that returns a function of \(n - 1\) arguments, repeating until \(n = 1\) at which point you perform the calculation in the original \(n\)-ary function. Named after Haskell Curry\(^4\), who did much of the early work with the process, though it was actually invented by a guy named Schönfinkel; presumably “schönfinkeling” was considered too unwieldy a term.
4A local—his parents were the founders of Curry College of Boston.
5Just make sure to wash your hands afterwards.
Our messages \( m \) that we pass in obviously have to be functions, and since they’re fed the two halves of the pair as arguments, they just pick one and return it:

\[
\begin{align*}
car \quad &\equiv \quad (\lambda(a \ b) \ a) \\
cdr \quad &\equiv \quad (\lambda(a \ b) \ b)
\end{align*}
\]

Just like true and false! This isn’t too surprising, though, since the structure is fairly similar: like conditionals, which select between two expressions to evaluate, the list operators are also selecting between two values (the halves of the pair).

That removes \texttt{cons}, \texttt{car}, \texttt{cdr} from our list. Let’s move on to something a little more challenging.

**Something a little more challenging: Numbers**

First, a simplifying assumption: we will only deal with the set of natural numbers. That is, all integers not less than zero, or to put it more usefully, a natural number is either zero or one plus a natural number.

What is a number? A number is a count of stuff. It’s a number of things, it’s a number of sheep, it’s a number of whatever you want it to be. Such as... function applications? Consider

\[
N = \begin{cases} 
0 & \text{0 function applications} \\
1 + N & \text{1 more function application than } N
\end{cases}
\]

What function should we use? It doesn’t really matter. Pick something relevant to the task at hand.

\[
\begin{align*}
0 &\equiv (\lambda(f) \ (\lambda(x) \ x)) \\
1 &\equiv (\lambda(f) \ (\lambda(x) \ (f \ x))) \\
2 &\equiv (\lambda(f) \ (\lambda(x) \ (f \ (f \ x)))) \\
N &\equiv (\lambda(f) \ (\lambda(x) \ (f^N \ x)))
\end{align*}
\]

This numeric representation is known as a “Church numeral”\(^6\). To get a Scheme number out of a Church numeral, just use \texttt{add1} for \( f \) and 0 for \( x \).

Of course, numbers aren’t all that useful in and of themselves, until we can do things with them. Like the “succ” function, i.e. \texttt{add1}. It takes a Church numeral. What does it return? A Church numeral, which is starts with \( \lambda f \), then \( \lambda x \), and the actual result is \( f \) applied one more time:

\[
succ \equiv (\lambda(n) \ (\lambda(f) \ (\lambda(x) \ (f \ ((n \ f) \ x)))))
\]

\(^6\)Not an ecclesiastical appellation, but an honorary one—they are named after their discoverer, Alonzo Church, who was a contemporary of Curry, Schönfinkel, Turing, and that whole crowd.
What about addition? Well, we can be a little creative here and use our Church numerals—they apply a function \( n \) times, and we have a “plus 1” function \( (\text{succ}) \), so we take one of the addends \( a \), and use the other addend \( b \) to apply the \( \text{succ} \) function \( b \) times:

\[
\text{add} \equiv (\lambda(a) \ (\lambda(b) \ ((a \ \text{succ}) \ b)))
\]

Similar reasoning can be used to get multiplication—\( a \times b \) is just adding \( b \), \( a \) times, to zero.

\[
\text{mult} \equiv (\lambda(a) \ (\lambda(b) \ ((a \ (\text{add}) \ b)) \ \text{zero}))
\]

But we ain’t seen nothing yet. Increasing a number is easy (“easy”), because you can just wrap a few more function applications around it. But once you’ve done that, you have a closure, and since you can’t see into a closure, how can you reduce the number of function applications? How in the world can we do the ‘pred’ function (subtract one)?

A brilliant intuition by Stephen Kleene\(^7\) shows us that it is possible\(^8\). This fundamental insight goes as follows: ultimately, we want a mapping from numbers to their predecessors; we could view this as a list of pairs. Each pair contains a number and its predecessor (ignoring the base case of zero, for now)

\[
\langle 0, 0 \rangle \quad \langle 1, 0 \rangle \quad \langle 2, 1 \rangle \quad \langle 3, 2 \rangle
\]

Well, it’s pretty easy to construct one pair if you already have the previous one, so what we need is something that will take the base case of \( \langle 0, 0 \rangle \) and find the \( n \)th pair from there. As it happens, the Church numeral itself will do this! Once we have the \( n \)th pair, we merely look at the second half to find the value of \( n - 1 \). Thus we have

\[
\text{pred} = (\lambda(n) \ (\text{cdr} \ ((n \ (\lambda(p) \ ((\text{cons} \ (\text{succ} \ (\text{car} \ p)) \ (\text{car} \ p)))) \ (\text{cons} \ 0 \ 0)))))
\]

And finally, to wrap up the numbers section, we need some numeric test in order to bottom out our recursion. How do we test if something is zero? Well the

\(^7\)Another of those swinging theoreticians of the 1930s. This stuff is way cooler than regular expressions, though that was the field where Kleene’s name became more famous.

\(^8\)And indeed, put his advisor, Alonzo Church, back on his track of research in this area.
number 0 is represented by zero function applications, so we need to pass our
number a base case that starts out true, and a function that returns false if ever
applied. Simply,

\[
\text{zero?} = (\lambda(n) \quad ((n \ (\lambda(dummy) \ false) \ true))
\]

Good, that knocks numbers and arithmetic off our list; there’s not much left:

\[
\text{lambda define symbols ( )}
\]

The parentheses aren’t going away, and symbols will be left as an exercise to the
reader. That leaves \text{define} as the only thing not defined in terms of lambda;
that is,

\[\text{What’s left? \textit{Recursion}.}\]

Our first instinct is simply to define factorial as

\[
\text{fact} = (\lambda (n) \quad (\text{if (zero? n) 1 (* n (fact (sub1 n))))})
\]

, which (except for the symbols) we’ve now shown can be entirely reduced to
lambdas. Problem is, it doesn’t work: \text{fact} hasn’t yet been bound when we use
it in the last line. Well, we could do

\[
\text{fact} = (\lambda (n) \quad (\text{if (zero? n) 1 (* n (lambda (n) \quad (\text{if (zero? n) 1 (* n (fact (sub1 n))))}) (sub1 n))))})
\]

ad nauseam, but of course that’s just postponing the problem. However, there is
a pattern here, and we know what to do with patterns: lambda-abstract them.

\[
((\lambda (mk\text{-fact}) \quad ...) \quad (\lambda (fact) \quad (\lambda (n) \quad (\text{if (zero? n) 1 (* n (fact (sub1 n))))}))})
\]
So far, now, we’ve just assigned a name: we’ll call those last three lines `mk-fact`. What goes in the ...? Let’s first try something like

```
(mk-fact ⚡)
```

, which is to say: ⚡ is something we don’t want to touch, if we do it’ll blow up. Well, if we can’t touch the bomb that the symbol `fact` is bound to, we can’t do the recursive case—but we can do the base case. So `(mk-fact ⚡)` returns a function that correctly calculates the factorial of zero, with unspecified behaviour if you pass it other stuff. Not very useful, but now consider

```
(mk-fact (mk-fact ⚡))
```

First time through, `fact` is now bound to `(mk-fact ⚡)`, so if it tries to call `fact` the recursion actually works. But not very well, because that second time through, touching `fact` hits the ⚡. All the same, we now have something that correctly calculates the factorial of both zero and one. Ultimately, this isn’t any different than the version where we kept rewriting the entire function, but at least it’s more succinct and compact.

However. Consider what happens when we try

```
(mk-fact mk-fact)
```

That is, the entire thing looks like this:

```
((lambda (mk-fact)
    (mk-fact mk-fact))
 (lambda (mk-fact)
    (lambda (n)
      (if (zero? n) 1 (* n ((mk-fact mk-fact) (sub1 n)))))))
```

The `mk-fact` in line 1 feeds its value to the two `mk-facts` in line 2; the second of these becomes the value of `fact` in the function of lines 3–5, and this function is itself bound to `mk-fact`. Nifty, we seem to have set up recursion. Of course, that last line won’t work, since `fact` (whose value is `mk-fact`) is expecting a function, not a value like `(sub1 n)`, so let’s feed it a function, like say `fact`. Meanwhile, we should rename `fact` since it is no longer a number-to-number function, but a function-to-function function that makes a factorial function; how about a name like “`mk-fact`”?  

```
((lambda (mk-fact)
    (mk-fact mk-fact))
 (lambda (mk-fact)
    (lambda (n)
      (if (zero? n) 1 (* n ((mk-fact mk-fact) (sub1 n)))))))
```
Mind-boggling. Why/how does it work? By making sure that anytime we are about to “run out of function”, we make another one. Earlier, we noted that if we’re taking the factorial of zero, it doesn’t matter what the value of mk-fact is— Millennials or anything else. And if we follow the recursive case, we run mk-fact to generate a function that will do one iteration of fact. Inductively, it then works for all cases.

Those last two lines are almost, but not quite, the pretty factorial function we all know and love, but that (mk-fact mk-fact) is ugly and doesn’t have much to do with the actual factorial. We can abstract that out... and what to name it? Why, fact, of course:

```
(((lambda (mk-fact)
    (mk-fact mk-fact))
  (lambda (mk-fact)
    ((lambda (fact)
        (lambda (n)
          (if (zero? n)
              1
              (* n (fact (- n 1)))))
      (mk-fact mk-fact))))
```

That is, (mk-fact mk-fact) is bound to fact, giving us exactly the previous version. But a further modification is desirable: lines 3–5 here are really the meat of the factorial calculation, while lines 1, 2, and 6 are all part of the instrumentation of making the recursion work, and furthermore, those lines could be reused for other functions we want to be recursive. Thus we have the function

```
(lambda (f)
  ((lambda (x) (x x))
    (lambda (x) (f (x x)))))
```

This function is known as $Y$, or the $Y$-combinator, and it enables one to perform recursion in the lambda calculus. It obeys the equality

$$(Y f) \equiv (f (Y f))$$

that is, it is a fixed point.

Note that this crucially only works under lazy evaluation—otherwise the (mk-fact mk-fact) would get evaluated once, get its value bound to fact, and that’d be the end of it, but we need fact to generate a new function each time through. In fact, the lambda calculus always requires lazy evaluation; there is a parallel theory known as the lambda-value calculus in which eager evaluation is used. It is left as an exercise to the reader how to modify the example so that it would work in an eager regime.
A job well done: Epilogue

We have now reduced our language to the following three-case grammar:

\[ \Lambda ::= \langle \text{var} \rangle \]
\[ | (\Lambda \ \Lambda) \]
\[ | (\lambda \ (\text{var}) \ \Lambda) \]

The language \( \Lambda \) has, simply, variables, function application, and abstraction; nothing else. It is known as the lambda calculus, and its development in the 1930s made theoreticians deliriously happy, because A) it strongly lends itself to inductive proofs, and B) the proofs need only deal with three cases!

Back in the 1930s, a lot of people were asking the question, “what can we compute?” This was not a trivial question, as it had important bearing on whether one language (or method of computation) was ‘more powerful’ than another (is C more powerful than Scheme? and so on). Turing tried and succeeded in reducing a huge variety of computation to a few simple operations involving a tape with a read/write head; meanwhile Church, Curry, et al were attacking the same problem from a completely different direction, and reduced a huge variety of computation to the lambda calculus. Happily, it was proven that Turing’s tape machine and Church’s lambda calculus are isomorphic—an algorithm in one can be reduced to an algorithm for the other—so that theoreticians can prove things under one system or the other, as convenient. Obviously, though, the ones that prove using the lambda calculus are much cooler.