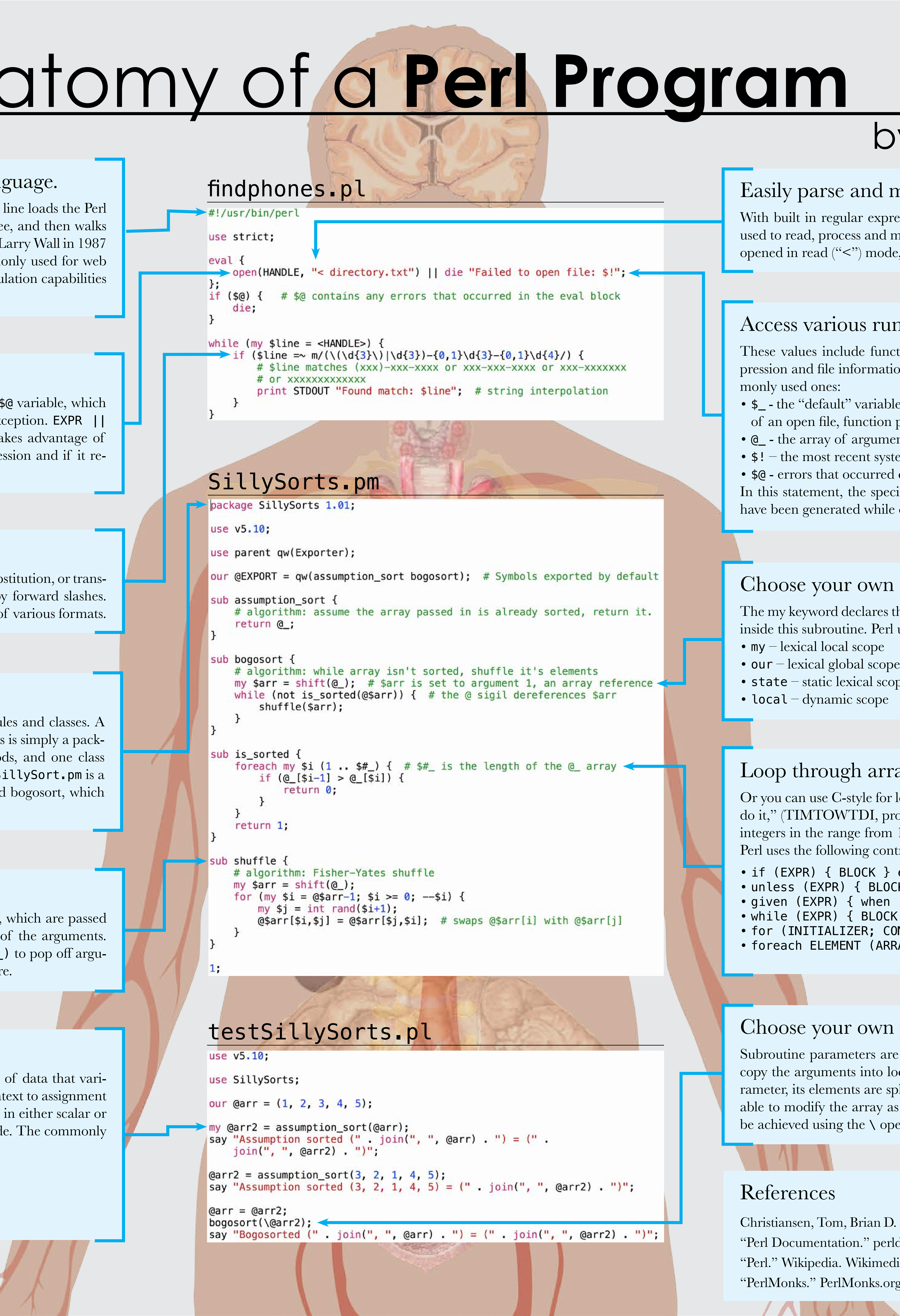
The Anatomy of a **Perl Program**

by Chip Jackson



Used as a general purpose scripting language.

Perl is high-level, interpreted, and dynamically typed. This line loads the Perl interpreter, which first compiles the script into a syntax tree, and then walks the tree to execute the program. Perl was first developed by Larry Wall in 1987 to make data processing easier, but today it is most commonly used for web scripting. It's greatest strengths are its powerful text manipulation capabilities and it's extensive set of modules available through CPAN.

Catch exceptions using `eval`.

Exceptions that occur within an eval block set the special `$@` variable, which can then be tested at the end of the block to catch the exception. `EXPR || die("error")` is a commonly used Perl statement that takes advantage of Perl's short-circuit Boolean evaluation to execute an expression and if it returns false throw an exception.

Regular expressions are built in.

The =~ operator is used for regular expression matching, substitution, or transliteration. Regular expressions are commonly delimited by forward slashes. This regular expression matches a 10-digit phone number of various formats.

Create your own modules and classes.

Perl uses the same package system for creating both modules and classes. A module is simply a file containing one package, while a class is simply a package. The subroutines in a package define a class's methods, and one class can inherit from another using the `use parent` directive. `SillySort.pm` is a module that exports two sort routines, assumption sort and bogosort, which can be used to sort arrays of scalars.

Write your own subroutines.

Subroutines can be called with any number of arguments, which are passed in through the `@_` special variable that contains an array of the arguments. A common way to access the arguments is using `shift(@_)` to pop off arguments from the front of the array one at a time, as seen here.

Use sigils to indicate variable type.

Sigils make it easy to tell what is a variable and what type of data that variable contains just by looking at its name. They also give context to assignment statements, making the right-hand expression be evaluated in either scalar or list context, depending on the sigil used on the left-hand side. The commonly used sigils are as follows:

- `$` - an individual number or string value.
- `@` - an array of values, indexed starting at 0.
- `%` - a hash table storing values keyed by strings.
- `&` - a callable subroutine.

findphones.pl

```perl
#!/usr/bin/perl

use strict;

eval {
    open(HANDLE, "< directory.txt") || die "Failed to open file: $!";
};
if ($@) {   # $@ contains any errors that occurred in the eval block
    die;
}

while (my $line = <HANDLE>) {
    if ($line =~ m/(\(\d{3}\)|\d{3})-{0,1}\d{3}-{0,1}\d{4}/) {
        # $line matches (xxx)-xxx-xxxx or xxx-xxx-xxxx or xxx-xxxxxxx
        # or xxxxxxxxxxxxx
        print STDOUT "Found match: $line";  # string interpolation
    }
}
```

SillySorts.pm

```perl
package SillySorts 1.01;

use v5.10;

use parent qw(Exporter);

our @EXPORT = qw(assumption_sort bogosort);  # Symbols exported by default

sub assumption_sort {
    # algorithm: assume the array passed in is already sorted, return it.
    return @_;
}

sub bogosort {
    # algorithm: while array isn't sorted, shuffle it's elements
    my $arr = shift(@_);  # $arr is set to argument 1, an array reference
    while (not is_sorted(@$arr)) {  # the @ sigil dereferences $arr
        shuffle($arr);
    }
}

sub is_sorted {
    foreach my $i (1 .. $#_) {  # $#_ is the length of the @_ array
        if (@_[$i-1] > @_[$i]) {
            return 0;
        }
    }
    return 1;
}

sub shuffle {
    # algorithm: Fisher-Yates shuffle
    my $arr = shift(@_);
    for (my $i = @$arr-1; $i >= 0; --$i) {
        my $j = int rand($i+1);
        @$arr[$i,$j] = @$arr[$j,$i];  # swaps @$arr[i] with @$arr[j]
    }
}

1;
```

testSillySorts.pl

```perl
use v5.10;

use SillySorts;

our @arr = (1, 2, 3, 4, 5);

my @arr2 = assumption_sort(@arr);
say "Assumption sorted (" . join(", ", @arr) . ") = (" .
    join(", ", @arr2) . ")";

@arr2 = assumption_sort(3, 2, 1, 4, 5);
say "Assumption sorted (3, 2, 1, 4, 5) = (" . join(", ", @arr2) . ")";

@arr = @arr2;
bogosort(\@arr2);
say "Bogosorted (" . join(", ", @arr) . ") = (" . join(", ", @arr2) . ")";
```

Easily parse and manipulate text files.

With built in regular expressions and a rich set of parsing abilities, Perl is commonly used to read, process and manipulate textual data. Here, the file "directory.txt" is being opened in read ("<") mode, and it's file handle is being stored in the `HANDLE` variable.

Access various runtime values using special variables.

These values include function parameters, environment variables, errors, regular expression and file information, and various other general-purpose variables. Some commonly used ones:

- `$_` - the "default" variable. Set to contain loop items, regular expression matches, lines of an open file, function parameters and more.
- `@_` - the array of arguments the currently executing function was called with.
- `$!` – the most recent system error.
- `$@` - errors that occurred during the most recently executed `eval` block.

In this statement, the special `$!` variable, which contains any error message that may have been generated while opening the file, is being inserted into the string.

Choose your own scoping rules.

The my keyword declares the `$arr` variable with local scope, so it is only accessible from inside this subroutine. Perl uses the following keywords to declare a variables scope:

- `my` – lexical local scope
- `our` – lexical global scope
- `state` – static lexical scope
- `local` – dynamic scope

Loop through arrays using `foreach`.

Or you can use C-style for loops, after all Perl's slogan is "There's more than one way to do it," (TIMTOWTDI, pronounced "Tim Toady"). Here, `$i` is being set to successive integers in the range from 1 to the size of the `@_` array with each iteration of the loop. Perl uses the following control flow structures:

- `if (EXPR) { BLOCK } elsif (EXPR) { BLOCK } else { BLOCK }`
- `unless (EXPR) { BLOCK }`
- `given (EXPR) { when (EXPR) { BLOCK } default { BLOCK } }`
- `while (EXPR) { BLOCK }`
- `for (INITIALIZER; CONDITION; MODIFIER) { BLOCK }`
- `foreach ELEMENT (ARRAY) { BLOCK }`

Choose your own parameter passing method.

Subroutine parameters are pass-by-reference, but to get pass-by-value you can simply copy the arguments into locally scoped `my` variables. When an array is passed as a parameter, its elements are split into separate arguments to the subroutine. In order to be able to modify the array as a whole, you must pass a reference to the array, which can be achieved using the \ operator, as shown here.

References

Christiansen, Tom, Brian D. Foy, and Larry Wall. Programming Perl. 4th ed. O'Reilly, 2012.

"Perl Documentation." perldoc.perl.org.

"Perl." Wikipedia. Wikimedia Foundation, 03 Oct. 2014.

"PerlMonks." PerlMonks.org. The Perl Foundation.