Software Construction

my_declaration_bug.pl

This shows a bug due to a missing my declaration

In this case the use of $i in is_prime without a my declarations changes $i outside the function and breaks the while loop calling the function

```
sub is_prime {
    my ($n) = @_;
    $i = 2;
    while ($i < $n) {
        return 0 if $n % $i == 0;
        $i++;
    }
    return 1;
}

$i = 0;
while ($i < 1000) {
    print "$i\n" if is_prime($i);
    $i++;
}
```

dot_product.0.pl

calculate Dot Product https://en.wikipedia.org/wiki/Dot_product of 2 lists - list are assumed to be the same length
this version of dot_product does not work
Perl functions argument are passed in a single array the functions can't determine how the list of arguments was formed

```
sub dot_product {
    my (@a, @b) = @_;  # BROKEN

    print "\@a = @a\n"; # all elements of @_ will be in @a
    print "\@b = @b\n"; # @b will be empty

    my $sum;
    foreach $i (0..$#a) {
        $sum += $a[$i] * $b[$i];
    }

    return $sum;
}

@x = (5..9);
@y = (11..15);

$dp = dot_product @x, @y;

print "(@x) . (@y) = $dp\n";
```

dot_product.1.pl

calculate Dot Product https://en.wikipedia.org/wiki/Dot_product of 2 lists - list are assumed to be the same length
this version of dot_product divides the single list of arguments into two lists this works in this case, but is not a general solution

```
sub dot_product {
    my $n = @_/2;
    my @a = @_[0 .. ($n-1)];
    my @b = @_[$n .. $#_];
    print "n=$n \@a=@a \@b=@b\n";
    my $sum;
    foreach $i (0..$#a) {
        $sum += $a[$i] * $b[$i];
    }

    return $sum;
}

@x = (5..9);
@y = (11..15);

$dp = dot_product @x, @y;

print "(@x) . (@y) = $dp\n";
```

dot_product.2.pl

calculate Dot Product https://en.wikipedia.org/wiki/Dot_product of 2 lists - list are assumed to be the same length
this version of dot_product expects two array references

```
sub dot_product {
    my ($aref, $bref) = @_;

    my $sum;
    foreach $i (0..$#$aref) {
        $sum += $$aref[$i] * $$bref[$i];
    }

    return $sum;
}

@x = (5..9);
@y = (11..15);

$dp = dot_product \@x, \@y;

print "(@x) . (@y) = $dp\n";
```

dot_product.3.pl

calculate Dot Product https://en.wikipedia.org/wiki/Dot_product of 2 lists - list are assumed to be the same length
this version of dot_product expects two array references and has a function_prototype specifying this which changes how dot_product can be called

```
sub dot_product(\@\@) {
    my ($aref, $bref) = @_;

    my $sum;
    foreach $i (0..$#$aref) {
        $sum += $$aref[$i] * $$bref[$i];
    }

    return $sum;
}

@x = (5..9);
@y = (11..15);

# due to the function prototype, implicitly a reference will be passed to @a and @b

$dp = dot_product @x, @y;

print "(@x) . (@y) = $dp\n";
```

sum_list.pl
3 different ways to sum a list - illustrating various aspects of Perl
simple for loop

```
sub sum_list0 {
    my (@list) = @_;
    my $total = 0;
    foreach $element (@list) {
        $total += $element;
    }
    return $total;
}

# recursive
sub sum_list1 {
    my (@list) = @_;
    return 0 if !@list;
    return $list[0] + sum_list1(@list[1..$#list]);
}

# join+eval - interesting but not recommended
sub sum_list2 {
    my (@list) = @_;
    return eval(join("+", @list))
}

print sum_list0(1..10), " ", sum_list1(1..10), " ", sum_list2(1..10), "\n";
```

sort_dates.pl

simple example illustrating use of sorting comparison funcrion note use of <=?>

```
sub random_date {
    return sprintf "%02d/%02d/%04d", 1 + rand 28, 1 + rand 12, 2000+rand 20
}

sub compare_date {
    my ($day1,$month1,$year1) = split /\D+/, $a;
    my ($day2,$month2,$year2) = split /\D+/, $b;
    return $year1 <=> $year2 || $month1 <=> $month2 || $day1 <=> $day2;
}

push @random_dates, random_date() foreach 1..5;
print "random_dates=@random_dates\n";
@sorted_dates = sort compare_date @random_dates;
print "sorted_dates=@sorted_dates\n";
```

sort_days.pl

Simple example of sorting a list based on the values in a hash.
This is very common pattern in Perl.

```
%days = (Sunday => 0, Monday => 1, Tuesday => 2, Wednesday => 3,
         Thursday => 4, Friday => 5, Saturday => 6);

sub random_day {
    my @days = keys %days;
    return $days[rand @days];
}

sub compare_day {
    return $days{$a} <=> $days{$b};
}

push @random_days, random_day() foreach 1..5;
print "random days = @random_days\n";
@sorted_days = sort compare_day @random_days;
print "sorted days = @sorted_days\n";
```

sort_days.1.pl

Simple example of sorting a list based on the values in a hash.
This is very common pattern in Perl. modified version ilustration Perl quote word operator and a hash slice

Perl's quote appropriate is a convenient way to create a list of words

```
@days = qw/Sunday Monday Tuesday Wednesday Thursday Friday Saturday/;

# Perl allows you to assign to multiple values in a hash simultaneously
@days{@days} = (0..6);

sub random_day {
    my @days = keys %days;
    return $days[rand @days];
}

sub compare_day {
    return $days{$a} <=> $days{$b};
}

push @random_days, random_day() foreach 1..5;
print "random days = @random_days\n";
@sorted_days = sort compare_day @random_days;
print "sorted days = @sorted_days\n";
```

split_join.pl

implementations of Perl's split & join

```
sub my_join {
    my ($separator, @list) = @_;

    return "" if !@list;

    my $string = shift @list;
    foreach $thing (@list) {
        $string .= $separator . $thing;
    }

    return $string;
}

sub my_split1 {
    my ($regexp, $string) = @_;
    my @list = ();

    while ($string =~ /(.*)$regexp(.*)/) {
        unshift @list, $2;
        $string = $1;
    }
    unshift @list, $string if $string ne "";

    return @list;
}

sub my_split2 {
    my ($regexp, $string) = @_;

    my @list = ();
    while ($string =~ s/(.*?)$regexp//) {
        push @list, $1;
    }
    push @list, $string if $string ne "";

    return @list;
}

$s = my_join("+", 1..5);

# prints 1+2+3+4+5 = 15
print "$s = ", eval $s, "\n";

# prints 2 4 8 16
@a = my_split1(",", "2,4,8,16");
print "@a\n";

# prints 2 4 8 16
@a = my_split2(",", "2,4,8,16");
print "@a\n";
```

push.pl

implementations of Perl's push

```
sub my_push1 {
    my ($array_ref,@elements) = @_;

    @$array_ref = (@$array_ref, @elements);

    return $#$array_ref + 1;
}


# same but with prototype
sub my_push2(\@@) {
    my ($array_ref,@elements) = @_;

    @$array_ref = (@$array_ref, @elements);

    return $#$array_ref + 1;
}

sub mypush2 {
    my ($array_ref,@elements) = @_;
    if (@elements) {
        @$array_ref = (@$array_ref, @elements);
    } else {
        @$array_ref = (@$array_ref, $_);
    }
}

@a = (1..5);

# note explicitly passing an array reference \@a
my_push1 \@a, 10..15;

# note prototype allows caused reference to array to be passed
my_push2 @a, 20..25;

# prints 1 2 3 4 5 10 11 12 13 14 15 20 21 22 23 24 25
print "@a\n";
```

print_odd.pl

8 different ways to print the odd numbers in a list - illustrating various aspects of Perl
simple for loop

```perl
sub print_odd0 {
    my (@list) = @_;

    foreach $element (@list) {
        print "$element\n" if $element % 2;
    }
}

# simple for loop using index
sub print_odd1 {
    my (@list) = @_;

    foreach $i (0..$#list) {
        print "$list[$i]\n" if $list[$i] % 2;
    }
}

# set $_ in turn to each item in list
# evaluate supplied expression
# print item if the expression evaluates to true

sub print_list0 {
    my ($select_expression, @list) = @_;
    foreach $_ (@list) {
        print "$_\n" if &$select_expression;
    }
}

# more concise version of print_list0
sub print_list1 {
   &{$_[0]} && print "$_\n" foreach @_[1..$#_];
}


# set $_ in turn to each item in list
# evaluate supplied expression
# return a list of items for which the expression evaluated to true
sub my_grep0 {
    my $select_expression = $_[0];
    my @matching_elements;
    foreach $_ (@_[1..$#_]) {
        push @matching_elements, $_ if &$select_expression;
    }
    return @matching_elements;
}


# more concise version of my_grep0
sub my_grep1 {
    my $select_expression = shift;
    my @matching_elements;
    &$select_expression && push @matching_elements, $_ foreach @_;
    return @matching_elements;
}

# calling helper function which returns
# list items selected by an expression
sub print_odd4 {
    my @odd = my_grep0 sub {$_ % 2}, @_;
    foreach $x (@odd) {
        print "$x\n";
    }
}


@numbers = (1..10);

# all 8 statements print the numbers 1,3,5,7,9 one per line

print_odd0(@numbers);
```

```
print_odd1(@numbers);

print_list0(sub {$_ % 2}, @numbers);

print_list1(sub {$_ % 2}, @numbers);

print_odd4(@numbers);

my_grep1 sub {odd $_ && print "$_\n"}, @_;

# using built-in grep and combining print
grep {$_ % 2 && print "$_\n"} @numbers;

# using built-in grep and join
print join("\n", grep {$_ % 2} @numbers), "\n";
```

rename.pl
rename specified files using specified Perl code
For each file the Perl code is executed with $_ set to the filename and the file is renamed to the value of $_ after the execution.
/usr/bin/rename provides this functionality

```
die "Usage: $0 <perl> [files]\n" if !@ARGV;
$perl_code = shift @ARGV;
foreach $filename (@ARGV) {

    $_ = $filename;
    eval $perl_code;
    die "$0: $?" if $?; # eval leaves any error message in $?
    $new_filename = $_;

    next if $filename eq $new_filename;

    die "$0: $new_filename exists already\n" if -e $new_filename;

    rename $filename, $new_filename or
        die "$0: rename '$filename' -> '$new_filename' failed: $!\n";
}
```

html_times_table0.pl
print a HTML times table

Note html_times_table has 6 parameters calls to the function are hard to read and its easy to introduce errors

```
sub html_times_table {
    my ($min_x, $max_x, $min_y, $max_y, $bgcolor, $border) = @_;

    my $html = "<table border=$border bgcolor=$bgcolor>\n";

    foreach $y ($min_y..$max_y) {
        $html .= "<tr>";
        foreach $x ($min_x..$max_x) {
            $html .= sprintf "<td align=right>%s</td>", $x * $y;
        }
        $html .=  "</tr>\n";
    }

    $html .=  "</table>\n";

    return $html;
}

# what do each of these parameters do?
print html_times_table(1, 12, 1, 12, "pink", 1);
```

html_times_table1.pl
print a HTML times table

Note use of a hash to pass named parameters

```
sub html_times_table {
    my %parameters = @_;

    my $html = "<table border=$parameters{border} bgcolor=$parameters{bgcolor}>\n";

    foreach $y ($parameters{min_y}..$parameters{max_y}) {
        $html .= "<tr>";
        foreach $x ($parameters{min_x}..$parameters{max_y}) {
            $html .= sprintf "<td align=right>%s</td>", $x * $y;
        }
        $html .=  "</tr>\n";
    }

    $html .=  "</table>\n";

    return $html;
}

# easy to understand what each paramater does
print html_times_table(bgcolor=>'pink', min_y=>1, max_y=>12, border=>1, min_x=>1, max_x=>12);
```

html_times_table2.pl

print a HTML times table

Note use of a hash to pass named parameters combined with a hash to provide default values for parameters

```
sub html_times_table {
    my %arguments = @_;

    my %defaults = (min_x=>1, max_x=>10, min_y=>1, max_y=>10, bgcolor=>'white', border=>0);

    my %parameters = (%defaults,%arguments);

    my $html = "<table border=$parameters{border} bgcolor=$parameters{bgcolor}>\n";

    foreach $y ($parameters{min_y}..$parameters{max_y}) {
        $html .= "<tr>";
        foreach $x ($parameters{min_x}..$parameters{max_y}) {
            $html .= sprintf "<td align=right>%s</td>", $x * $y;
        }
        $html .=  "</tr>\n";
    }

    $html .=  "</table>\n";

    return $html;
}

# even more readable because we don't have to supply default values for parameters

print html_times_table(max_y=>12, max_x=>12, bgcolor=>'pink');
```

quicksort0.pl

```
@list = randomize_list(1..20);
print "@list\n";
@sorted_list0 = sort {$a <=> $b} @list;
print "@sorted_list0\n";
@sorted_list1 = quicksort0(@list);
print "@sorted_list1\n";
@sorted_list2 = quicksort1(sub {$a <=> $b}, @list);
print "@sorted_list2\n";

sub quicksort0 {
    return @_ if @_ < 2;
    my ($pivot,@numbers) = @_;
    my @less = grep {$_ < $pivot} @numbers;
    my @more = grep {$_ >= $pivot} @numbers;
    my @sorted_less = quicksort0(@less);
    my @sorted_more = quicksort0(@more);
    return (@sorted_less, $pivot, @sorted_more);
}


sub quicksort1 {
    my ($compare) = shift @_;
    return @_ if @_ < 2;
    my ($pivot, @input) = @_;
    my (@less, @more);
    partition1($compare, $pivot, \@input, \@less, \@more);
    my @sorted_less = quicksort1($compare, @less);
    my @sorted_more = quicksort1($compare, @more);
    my @r = (@sorted_less, $pivot, @sorted_more);
    return (@sorted_less, $pivot, @sorted_more);
}

sub partition1 {
    my ($compare, $pivot, $input, $smaller, $larger) = @_;
    foreach $x (@$input) {
        our $a = $x;
        our $b = $pivot;
        if (&$compare < 0) {
            push @$smaller, $x;
        } else {
            push @$larger, $x;
        }
    }
}

sub randomize_list {
    my @newlist;
    while (@_) {
        my $random_index = rand @_;
        my $r = splice @_, $random_index, 1;
        push @newlist, $r;
    }
    return @newlist;
}
```

quicksort1.pl

```perl
sub quicksort0(@);
sub quicksort1(&@);
sub partition1(&$\@\@\@);
sub randomize_list(@);

@list = randomize_list 1..20;
print "@list\n";
@sorted_list0 = sort {$a <=> $b} @list;
print "@sorted_list0\n";
@sorted_list1 = quicksort0 @list;
print "@sorted_list1\n";
@sorted_list2 = quicksort1 {$a <=> $b} @list;
print "@sorted_list2\n";

sub quicksort0(@) {
    return @_ if @_ < 2;
    my ($pivot,@numbers) = @_;
    my @less = grep {$_ < $pivot} @numbers;
    my @more = grep {$_ >= $pivot} @numbers;
    my @sorted_less = quicksort0 @less;
    my @sorted_more = quicksort0 @more;
    return (@sorted_less, $pivot, @sorted_more);
}


sub quicksort1(&@) {
    my ($compare) = shift @_;
    return @_ if @_ < 2;
    my ($pivot, @input) = @_;
    my (@less, @more);
    partition1 \&$compare, $pivot, @input, @less, @more;
    my @sorted_less = quicksort1 \&$compare, @less;
    my @sorted_more = quicksort1 \&$compare, @more;
    my @r = (@sorted_less, $pivot, @sorted_more);
    return (@sorted_less, $pivot, @sorted_more);
}

sub partition1(&$\@\@\@) {
    my ($compare, $pivot, $input, $smaller, $larger) = @_;
    foreach $x (@$input) {
        our $a = $x;
        our $b = $pivot;
        if (&$compare < 0) {
            push @$smaller, $x;
        } else {
            push @$larger, $x;
        }
    }
}

sub randomize_list(@) {
    my @newlist;
    while (@_) {
        my $random_index = rand @_;
        my $r = splice @_, $random_index, 1;
        push @newlist, $r;
    }
    return @newlist;
}
```