SUBROUTINES

Brandon Seah

Perl Course 2017

LANGUAGE IS BUILT ON ABSTRACTION

- “Wood fibers flattened, stacked, attached”
- “Paper stack, attached”
- “Book”

MATH IS BUILT ON ABSTRACTION

$\frac{n_1 + n_2}{2}$

$\frac{n_1 + n_2 + n_3}{3}$

etc.

“Sum”, “Mean”

CODE IS BUILT ON ABSTRACTION

```
my $number = 3;
my $result = $number*$number*$number;
my $result2 = $number**3; # Built-in operator
```

PERL THE LANGUAGE VS. PERL THE INTERPRETER

Perl is written in C

```
1134  /* Ordinary operators. */
1135
1136  PP(pp_pow)
1137  {
1138      dSP; dATARGET; SV *svl, *svr;
1139  #ifdef PERL_PRESERVE_IVUV
1140      bool is_int = 0;
1141  #endif
1142      tryAMAGICbin_MG(pow_amg, AMGf_assign|AMGf_numeric);
1143      svr = TOPs;
1144      svl = TOPm1s;
1145  #ifdef PERL_PRESERVE_IVUV
1146      /* For integer to integer power, we do the calculation by hand wherever
1147         we're sure it is safe; otherwise we call pow() and try to convert to
1148         integer afterwards. */
1149      if (SvIV_please_nomg(svr) && SvIV_please_nomg(svl)) {
```

```
1150                    UV power;
1151                    bool baseuok;
1152                    UV baseuv;
1153
1154                    if (SvUOK(svr)) {
1155                        power = SvUVX(svr);
1156                    } else {
1157                        const IV iv = SvIVX(svr);
1158                        if (iv >= 0) {
1159                            power = iv;
1160                        } else {
1161                            goto float_it; /* Can't do negative powers this way.  */
1162                        }
1163                    }
```

LEVELS OF ABSTRACTION

- Perl is written in C
- C compiler is written in ... C (?!)
- Interpreter translates human-readable code to machine code
- Machine code is specific to hardware type

MACHINES SIMULATING MACHINES

Minecraft computer (Trailer)

Virtual machines

MAKE YOUR OWN ABSTRACTIONS

```
my $number = 3;
my $result = power($number,2);
sub power {
  my ($num, $exp) = @_; # Inputs
  my $output = 1;
  for (my $i=1; $i <= $exp; $i++) {
    $output = $output * $num;
  }
  return $output; # Output
}
```

What happens if `$exp <= 0`?

BENEFITS OF ABSTRACTION

- Avoid repetition
- Code is cleaner
- Changes only made once
- Reduce human error
- Build abstractions
- Easier to read code
- Easier to write code

SUB BLOCKS CAN GO ANYWHERE

```
sub say_hello { print "Hello!\n"; }
say_hello();
```

Equivalent to:

```
say_hello();
sub say_hello { print "Hello!\n"; }
```

SCOPING OF VARIABLES IN A `SUB` BLOCK

```
sub cube_it {
    my ($input) = @_;
    my $out = $input***3;
    return $out;
}
print $out; # Doesn't exist outside block
```

INPUT TO A FUNCTION

```
sub cube_it {
    my ($input) = @_;
    my $out = $input***3;
    return $out;
}
```

- Inputs are passed to a special array called @_
- Even if there is only one input, still goes to an array

CALLING A SUBROUTINE WITH INPUT AND RETURN VALUE

```
sub cube_it {
    my ($input) = @_;
    my $out = $input***3;
    return $out;
}
my $answer = cube_it(3);
print $answer."\n";
```

EXERCISE: MATCHING FASTA HEADERS

Convert the following script to use a subroutine

```
while (<>) {
  if ($_ =~ m/^>/) {
    print "Header found\n";
  } else {
    print "Not a header\n";
  }
}
```

File: XXX

EXERCISE: MATCHING FASTA HEADERS

One possible solution:

```
while (<>) {
  print isheader($_);
}
sub isheader {
  my ($input) = @_;
  if ($input =~ m/^>/) {
    return "Header found\n";
  } else {
    return "Not a header\n";
  }
}
```

CALLING A SUBROUTINE WITH NO RETURN VALUE

```
sub repeat_it {
    my ($input) = @_;
    print "You said: $input."\n";
}
repeat_it("Hello");
my $test = repeat_it ("Hello");
print $test; # What happens?
```

CALLING A SUBROUTINE WITH NO INPUTS

```
sub random_fruit {
    my @fruit = ("apple","banana","orange","raspberry");
    my $random = int(rand(3));
    return $fruit[$random];
}
my $choice = random_fruit();
print $choice."\n";
```

What happens if you wrote

`my $choice = random_fruit;` instead?

ADDITIONAL MATERIAL

PASSING ARRAYS/HASHES TO SUBROUTINE

- Use *references* (see Day 2)

```
my @arr = (1, 2, 3, 4, 5);
add_arr(\@arr); # Notice backslash before @arr
sub add_arr {
  my $arr_ref = @_;
  my @array = @$arr_ref; # Dereference
  my $sum = 0;
  foreach my $num (@array) {
    $sum += $num;
  }
  return ($sum);
}
```

PECULIARITIES OF SUBROUTINES IN PERL

- Called "subroutines" instead of "functions"
- Need parantheses even for functions without inputs
- Accepts only lists of scalars as inputs
- Input arguments are not directly defined
- In older versions of Perl, subroutines needed sigil `&`