“Scripting: Higher Level Programming for the 21st Century” (John Ousterhout)

http://www.tcl.tk/doc/scripting.html

For the last fifteen years a fundamental change has been occurring in the way people write computer programs. The change is a transition from system programming languages such as C or C++ to scripting languages such as Perl or Tcl. Although many people are participating in the change, few people realize that it is occurring and even fewer people know why it is happening. This article is an opinion piece that explains why scripting languages will handle many of the programming tasks of the next century better than system programming languages.

Scripting languages are designed for different tasks than system programming languages, and this leads to fundamental differences in the languages.

“Scripting: Higher Level Programming for the 21st Century” (John Ousterhout)

In deciding whether to use a scripting language or a system programming language for a particular task, consider the following questions:

- Is the application’s main task to connect together pre-existing components?
- Will the application manipulate a variety of different kinds of things?
- Does the application include a graphical user interface?
- Does the application do a lot of string manipulation?
- Will the application’s functions evolve rapidly over time?
- Does the application need to be extensible?

"Yes" answers to these questions suggest that a scripting language will work well for the application.

“Scripting: Higher Level Programming for the 21st Century” (John Ousterhout)

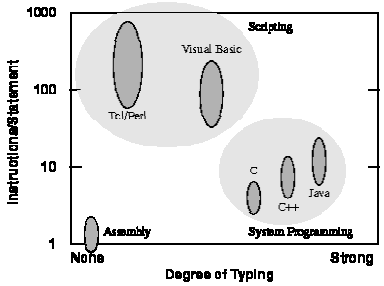

Figure 1. A comparison of various programming languages based on their level (higher level languages execute more machine instructions for each language statement) and their degree of typing. System programming languages like C tend to be strongly typed and medium level (5-10 instructions/statement). Scripting languages like Tcl tend to be weakly typed and very high level (100-1000 instructions/statement).

“Scripting: Higher Level Programming for the 21st Century” (John Ousterhout)

"Yes" answers to the following questions suggest that an application is better suited to a system programming language:

- Does the application implement complex algorithms or data structures?
- Does the application manipulate large datasets (e.g. all the pixels in an image) so that execution speed is critical?
- Are the application’s functions well-defined and changing slowly?

Scripting and system programming are symbiotic. Used together, they produce programming environments of exceptional power: system programming languages are used to create exciting components which can then be assembled using scripting languages.

Perl

Practical Extraction and Report Language

scalar1.pl

```
#!/usr/bin/perl -w

$a = 3;
$b = 5;

$rem1 = $a % $b;            # 3
$rem2 = $b % $a;            # 2

$a++;                       # 4
$b--;                       # 4

$n1 = $a + $b * 2;          # 12
$n2 = ($a + $b) * 2;        # 16
$n3 = 12 / $a / 2;          # 1.5
$n4 = 12 / ($a / 2);        # 6
$n5 = (2*2)**($b-2)**2;     # 256
```


hello.pl

```
#!/usr/bin/perl

print "Hello world\n";
```

simple.pl

```
#!/usr/bin/perl

$a = 2;
$b = 3;
$result = $a + $b;
print "Result is: $result\n";
```


Loops in Perl

```
$i = 1;
while ( $i <= 4 ) {
    print "$i\n";
    $i++;
}

$i = 1;
until ( $i > 4 ) {
    print "$i\n";
    $i++;
}

for ( $i = 1 ; $i <= 4 ; $i++ ) {
    print "$i\n";
}

foreach $i ( (1,2,3,4) ) {
    print "$i\n";
}
```

countdown.pl

```
#!/usr/bin/perl

#
# file:         countdown.pl
# purpose:      a 10 second countdown
#

$countdown = 10;
while ( $countdown != 0 ) {
        print "$countdown...\n";
        sleep 1;
        --$countdown;
}
print "BOOM!\n";
```


string1.pl

```
#!/usr/bin/perl

$empty = "";
$a = "Bioinformatics";
$b = "\"Perl Programming\"\n";
$me = "Graham\tChalmers\t6475\n";

print "$a $empty $b";
print $me;
print "\n";
```

```
Bioinformatics  "Perl Programming"
Graham  Chalmers        6475
```


scalar2.pl

```
#!/usr/bin/perl

$str1 = "Merry";
$str2 = "_Christmas! ";
$a = $str1 . "_Christmas!_";  # Merry_Christmas!_
$b = $str1 . $str2;           # Merry_Christmas!_
$c = "$str1$str2";            # Merry_Christmas!_
$b .= $b;                     # Merry_Christmas!_Merry_Christmas!_
$d = $c x 2;                  # Merry_Christmas!_Merry_Christmas!_
$e = chop($str1);             # y
$f = length($str1);           # 4
$g = lc($str1);               # merr
$h = uc($str1);               # MERR
$i = substr($a,0,3);          # Mer
$j = substr($a,-4,2);         # as
$k = index($a,"m");           # 12
```


string2.pl

```
#!/usr/bin/perl

#
# demonstrate single-quoted strings
#

$empty = '';
$a = 'Bioinformatics';
$b = '\"Perl Programming\"\n';
$me = 'Graham\tChalmers\t6475\n';

print "$a $empty $b";
print $me;
print "\n";
```

```
Bioinformatics  \"Perl Programming\"\nGraham\tChalmers\t6475\n
```

circle.pl

```
#!/usr/bin/perl -w

$pi = 3.1415925;

print "Please type in the radius: ";
$radius = <STDIN>;
chomp($radius);

$area = $pi * $radius * $radius;
$circ = 2 * $pi * $radius;

print "A circle of radius $radius has area $area\n",
      "and circumference $circ\n";
```

```
Please type in the radius: 4
A circle of radius 4 has area 50.26548
and circumference 25.13274
```


copyfile.pl

```
#!/usr/bin/perl -w

open(SOURCE, "file_A")  || die "cannot open file_A: $!";
open(TARGET, ">file_B") || die "cannot open file_B: $!";
while ( $line = <SOURCE> ) {
       print TARGET $line;
}
close(SOURCE);
close(TARGET);
```

```
#!/usr/bin/perl -w

open(SOURCE, "file_A")  || die "cannot open file_A: $!";
open(TARGET, ">file_B") || die "cannot open file_B: $!";
while ( <SOURCE> ) {
       print TARGET; }
close(SOURCE);
close(TARGET);
```


Opening files

```
open(SOURCE1, "file1");       # reading

open(SOURCE1, "<file2");      # reading

open(RESULT1, ">output1");    # writing (create or overwrite)

open(RESULT2, ">>output2");   # writing (create or append)

open(RESULT3, "+<inoutfile"); # reading/writing

open(SOURCE1, "file1") or die "Unable to open file: $!";
open(SOURCE1, "file1") || die "Unable to open file: $!";

close(SOURCE1);
```


Command line arguments

```
#!/usr/bin/perl

#
# file:         arguments.pl
# purpose:      prints the command line arguments
#

print "Command line arguments are: @ARGV\n";
print "The first argument is: $ARGV[0]\n";
```

Variables beginning with an @ symbol are array variables.
(Scalar) element at position i within an array @a is accessed by $a[i-1]

mycat.pl

```
#!/usr/bin/perl
while ( $_ = <ARGV> ) {
        print $_;
}
```

```
#!/usr/bin/perl
while ( <ARGV> ) {
        print;
}
```

```
#!/usr/bin/perl
while ( <> ) {
        print;
}
```


Comparison operators

Operation	Numeric	String
equal	==	eq
not equal	!=	ne
less than	<	lt
greater than	>	gt
less than or equal	<=	le
greater than or equal	>=	ge

What is true?

- anything except "" and "0"
- any number except 0
- any non-empty array

Conditional statements

```
if ( expression ) {
    # do if true
}

if ( expression ) {
    # do if true
} else {
    # do if flase
}

if  ( expression1 ) {
    # do if expression1 is true
} elsif ( expression2 ) {
    # do if expression1 is false and expression2 is true
} else {
    # do if expression1 is false and expression2 is false
}
```


Executing Perl programs

You can invoke the Perl interpreter directly, e.g.

```
perl program.pl
```

Or, if the first line of the program contains "#!" followed by the path of the Perl interpreter, and the program file is executable, you can just type the name of the program file on the command line, e.g.

```
./program.pl
```

To make a program file executable, use the chmod command, e.g.

```
chmod u+x program.pl
```