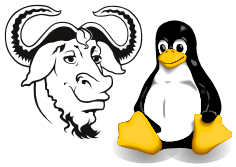

Summary of Perl

1 Main Topics

Shebang Each Perl program begins with a "shebang":

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

It tells the operating system which interpreter to use to execute the program.

You can add options to this, such as the `-w` above, which switches on additional warnings. I strongly recommend always using this while developing the program.

1.1 Variables, Operators

Scalars and non-scalars There are two categories of variables: *scalars* and *non-scalars*.

- scalars have a single value, such as `"a string"`, and
- non-scalars have a list of values, such as `( 1, 2, "a string")`

Non-scalars There are two types of non-scalars: *arrays* and *hashes*.

- *arrays* are much like arrays in Java or C (though much more versatile).
- *hash*es are like arrays that are indexed by strings, a bit like `java.util.Hashtable`, but simpler and more flexible.

`$, @ and %` Scalar variable values always start with a `$`, such as `$var = 1;`

Arrays variable values always start with a `@`, such as `@array = (2, 4, 6);`

Hash variable values always start with a `%`, such as `%hash = ( 'NL' => 'Netherlands', BE => 'Belgium' );`

Note that it is a *value*. For example, in `@array`, there is a scalar value `$array[0]`, and in `%hash`, there is a scalar value `$hash{"BE"}`.

Variable Interpolation A variable can be put right into a string like this: `"The value of \$var is $var.\n"`

If you print that string, the value of `$var` will be printed in the string, instead of the four characters `$var`. Notice that, just as in C, the backslash hides the special meaning of special characters such as `$`.

Operators: Perl has all the operators of C, in the same priority as in C.

Note Perl also has special operators for comparing strings:

Comparison	*Numeric*	*String*
equal	`==`	`eq`
not equal	`!=`	`ne`
greater than	`>`	`gt`
less than	`<`	`lt`
greater than or equal to	`>=`	`ge`
less than or equal to	`<=`	`le`

`use strict;` Turns on compile-time checks for lots of possible error conditions, such as undeclared variables, and other possible typing errors. I strongly recommend using this in all your programs that are longer than half a page.

`my` and `our`: are used to declare local variables and static variables, respectively. Necessary if you put

```
use strict;
```

in your program.

1.2 Statements

`if`, `while`, `for` need braces: You must use braces in a normal `if` statement, unlike in C or Java.

`if` statement: The `if` statement is similar to C or Java, except that there is a keyword "`elsif`":

```
if ( $age > $max ) {
    print "Too old\n";
} elsif ( $age < $min ) {
    print "Too young\n";
} else {
    print "Just right\n";
}
```

`for` loops: There are two types of `for` loop, one as in C and Java, the other is more useful in Perl:

```
for ( $i = 0; $i < $max; ++$i ) {
   $sum += $array[ i ];
}
```

But this `for` loop is much more useful. Here is an example that adds 1 to each element of an array:

```
foreach $a ( @array ) {
    ++$a;
}
```

Notice that `$a` here is made a reference to each element of the array, so changing `$a` actually changes the array element. You can write "`for`" or "`foreach`", Perl won't mind.

Special variable: `$_`: this special variable appears as the default argument of many built-in functions, including `print`, so this `foreach` loop prints all elements of `array`:

```
foreach ( @array ) {
    print;
}
```

`while` **loops:** are rather like in C or Java.

```
while ( $i < $max ) {
    ++$i;
}
```

Reading each line from input files: We often use a `while` loop to read each line from each of the files listed on the command line:

```
while ( <> ) {
    print $_;
}
```

What this does is:

- If there are command line parameters to this script, then it assumes that they are file names, and opens each in turn, and loops once for each line in the file, setting `$_` to that line
- Otherwise, it reads standard input, setting `$_` to each line.

Note that you could achieve the same result as above with:

```
print <>;
```

Reading from standard input only: is very similar to using `<>`. This example prints each line of standard input:

```
while ( <STDIN> ) {
    print $_;
}
```

`next` and `last` `next` is like `continue` in C; `last` is like `break` in C.

1.3 Array operations

`push` add a value at the end of an array

`pop` remove and return value from end of an array

`shift` remove and return value from the beginning of an array

`unshift` add value to the beginning of an array

1.4 `split` and `join`

`split` splits a string into an array:

```
my $pwline = "nicku:x:500:500:Nick Urbanik:/home/nicku:/
my ( $userid, $pw, $userid_number, $group_id_number,
     $name, $home_dir, $shell ) = split /:/, $pwline;
```

`join` is the opposite of `split` and joins an array into a string:

```
my $pwline = join ':', @pwfields;
```

1.5 Executing External Programs

Perl provides many ways of doing this, but we just used the `system` built-in function. In the laboratory in creating user accounts, I have written solutions that pass an array to `system`:

```
system:         my @cmd = (
                       'useradd',
                       '-c', "\"$name\"",
                       '-p', $hashed_passwd,
```

```
                $id
               );
        print "@cmd\n";
        system @cmd;
```

This also works:

```
system "useradd -c \"$name\" -p \"$hashed_passwd\
```

The difference is that the second form is usually passed to a command shell (such as `/bin/sh` or `CMD.EXE`) to execute, whereas the first form is executed directly.

Was the command successful? You can tell if the command was successful by checking that the return value was zero:

```
if ( system( "useradd -c \"$name\" -p \"$hashed_passwd\"
    print "useradd failed";
    exit;
}
```

This is usually written in Perl more simply using the built in function `die`, and the `or` operator:

```
system( "useradd -c \"$name\" -p \"$hashed_passwd\" $id"
   or die "useradd failed";
```

2 Regular Expressions

We spent most time in the laboratory and in the lectures studying and using regular expressions. Regular expressions are an important part of Perl. Regular expressions just been incorporated into Java 1.4, and are based directly on Perl regular expressions. Regular expressions are also used in many other programming languages, text editors, programs. . . even Microsoft Word. They will be an important part of the exam.

You should be familiar with character classes, matching the beginning and end of a line, and selecting part of a match. At an absolute minimum, you *must* be familiar with the application of:

- `\` Quote the next metacharacter
- `^` Match the beginning of the line
- `.` Match any character (except newline)
- `$` Match the end of the line (or before newline at the end)
- `|` Alternation
- `()` Grouping
- `[]` Character class
- `*` Match 0 or more times
- `+` Match 1 or more times

3 Perl Regular Expression Symbols: extracted from `perlre` manual page

[Note: this table will be provided in the exam].

`\`	Quote the next metacharacter
`^`	Match the beginning of the line
`.`	Match any character (except newline)
`$`	Match the end of the line (or before newline at the end)
`|`	Alternation
`()`	Grouping
`[]`	Character class
`*`	Match 0 or more times
`+`	Match 1 or more times
`?`	Match 1 or 0 times
$\{n\}$	Match exactly n times
$\{n,\}$	Match at least n times
$\{n,m\}$	Match at least n but not more than m times
`\w`	Match a "word" character (alphanumeric plus "_")
`\W`	Match a non-"word" character
`\s`	Match a whitespace character
`\S`	Match a non-whitespace character
`\d`	Match a digit character
`\D`	Match a non-digit character
`(?:pattern)`	This is for clustering, not capturing; it groups subexpressions like "`()`", but doesn't make back references as "`()`" does.

Regular Expression Modifiers

`i` Do case-insensitive pattern matching.

`x` Extend your pattern's legibility by permitting whitespace and comments.