cs3157 lecture #7 notes.

mon 3 mar 2003

http://www.cs.columbia.edu/˜cs3157

- news
  - homework #2 is due next monday (10 mar)
- today’s topic
  - perl

perl — history.

- written by Larry Wall
- designed to produce reports for a bug reporting system
- created on and developed for Unix, but Windows and Mac versions also exist
- intended to be a *useful* language
- see `http://www.perl.com`
  - you can download perl from there
  - and find documentation, etc.
- perl5 has more stuff in it, e.g.:
  - option to compile perl into C
  - threads
- but we'll just cover basic perl

perl — basics.

- first line of file is

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

- this is the path to the perl executable
- if it doesn't work, then do `which perl` to find out where perl is installed on your system
- the perl executable runs the perl interpreter, to interpret and execute your perl script
- the interpreter converts script to bytecode prior to execution, so it is sort of like a compiler (although bytecode is not stored anywhere)
- make the script executable (`chmod +x <filename>`), like your shell scripts from last week

perl — program structure.

- whitespace
  - only needed to separate terms
  - all whitespace (spaces, tabs, newlines) is the same
- semicolons
  - every simple statement must end with one
  - except compound statements enclosed in braces (i.e., no semicolon needed after the brace)
  - except final statements within braces
- declarations
  - only subroutines and report formats need explicit declarations
  - otherwise, variables in perl are like in shell scripts — they are declared and initialized all at once
- comments
  - from hash (#) to end of line

perl — data types and variables.

- three basic data types:
  - scalars
  - arrays
  - hashes

perl — variables.

- first letter indicates its type:
  - `$` — scalar
  - `@` — array
  - `%` — hash (key/value pair)
- names consist of letters, digits, underscores; up to 255 chars
- case sensitive
- should start with a letter or underscore (otherwise wierd rules apply)
- uninitialized variables have value `undef`

perl — scalars.

- begin with `$`
- numbers
  - integers
  - floating point
  - e.g., `123, -456, 0xff, 3.14, 4_567`
- strings
  - delimited by single or double quotes
  - e.g, `"123", "abc", ’alphabet’`

perl — arrays.

- begin with `@`
- ordered list of scalar values
- e.g.: `@fruit = ("apple", "orange", "pear");`
- refer to single element using `$` in front of name (in place of `@`) and index of element in square brackets
- e.g.: `$fruit[0]` is `"apple"`
- negative subscripts count backwards from the last element;
  -1 is the last element in the list

perl — hashes.

- begin with `%`
- name/value pair
- e.g.: `%fruit = ("apples", 3, "oranges", 7, "pears", 6);`
- pick out one by referring to its name
- e.g.: `$fruit{"apples"}` is `3`
- you can also define like this:

```
%fruit = {
  apples => 3,
  oranges => 7,
  pears => 6
};
```

perl — contexts.

- operations happen in one of two contexts:
  - scalar
  - list
- some operators return scalars and some return lists
- some can return either, depending on the context
- two examples...

perl — contexts, example 1.

- example:

```
#!/usr/bin/perl

($sec,$min,$hr,$mday,$mon,$yr,$wday,$yday,$isdst) = localtime();
print "s=",$sec," min=",$min," hr=",$hr," mday=",$mday,
  " mon=",$mon," yr=",$yr," wday=",$wday,
  " yday=",$yday," isdst=",$isdst,"\n";

$today = localtime();
print "today=",$today,"\n";
```

- output:

```
s=31 min=29 hr=21 mday=2 mon=2 yr=103 wday=0 yday=60 isdst=0
today=Sun Mar  2 21:29:31 2003
```

perl — contexts, example 2.

- example

```
#!/usr/bin/perl

$a = (2,4,6,8);
print '$a=',$a,"\n";

@b = (2,4,6,8);
print '@b=',@b,"\n";

$a = @b;
print '$a=',$a,"\n";
```

- output

```
$a=8
@b=2468
$a=4
```

perl — statements.

- simple statements are expressions that get evaluated
- they end with a semicolon (`;`)
- a sequence of statements can be contained in a *block*, delimited by braces (`{` and `}`)
- the last statement in a block does not need a semicolon
- blocks can be given labels:

```
myblock: {
  print "hello world\n";
}
```

perl — conditionals.

- three forms
- simple if

```
if (expression) {block} else {block}
```

- unless

```
unless (expression) {block} else {block}
```

- compound if

```
if (expression1) {block}
elsif (expression2) {block}
 ...
elsif (expressionN) {block}
else {block}
```

perl — conditionals, example.

```
#!/usr/bin/perl

@b = (2,4,6,8);
$a = @b;

if ( $a > 0 ) { print "a is greater than 0!\n" }
else { print "a is NOT greater than 0!\n" }

unless ( $a > 0 ) { print "a is NOT greater than 0!\n" }
else { print "a is greater than 0!\n" }

if ( $a > 0 ) { print "a is greater than 0!\n" }
elsif ( $a < 0 ) { print "a is less than 0!\n" }
else { print "a is exactly 0!\n" }
```

perl — loops.

- while
- for
- foreach

perl — while loops.

- syntax:

```
while (expression) {block}
```

- example

```
#!/usr/bin/perl

@b = (2,4,6,8);
$a = @b;

$i=0;
while ( $i < $a ) {
    print "i=",$i," b[i]=",$b[$i],"\n";
    $i++;
}
```

perl — for loops.

- syntax:

```
for ( expression1; expression2; expression3 ) {block}
```

- example:

```
#!/usr/bin/perl

@b = (2,4,6,8);
$a = @b;

for ( $i=0; $i<$a; $i++ ) {
    print "i=",$i," b[i]=",$b[$i],"\n";
}
```

perl — foreach loops.

- syntax:

```
foreach var (list) {block}
```

- example:

```
#!/usr/bin/perl

@b = (2,4,6,8);
$a = @b;

foreach $e (@b) {
    print "e=",$e,"\n";
}
```

perl — modifiers.

- you can follow a simple statement by an if, unless, while or until modifier:

```
statement if expression;
statement unless expression;
statement while expression;
statement until expression;
```

- example:

```
#!/usr/bin/perl

@b = (2,4,6,8);
$a = @b;

print "hello world!\n" if ($a < 10);
print "hello world!\n" unless ($a < 10);
#print "hello world!\n" while ($a < 10);
print "hello world!\n" until ($a < 10);
```

perl — special variables.

- there’s a (long) list of global special variables...
- a few important ones:
- `$_` = default input and pattern-searching string
- example:

```
#!/usr/bin/perl

@b = (2,4,6,8);
$a = @b;

foreach (@b) {
    print $_,"\n";
}
```

perl — other global special variables.

- there are lots of shortcuts; here are some (note that some also have an “English” equivalent, if you load in a special perl module):
- `$/` = input record separator (default is newline)
- `$$` = process id of the perl process running the script
- `$<` = real user id of the process running the script
- `$0` = (0=zero) name of the perl script
- `@ARGV` = list of command-line arguments
- `%ENV` = hash containing current environment
- `STDIN` = standard input
- `STDOUT` = standard output
- `STDERR` = standard error

perl — operators.

- unary:
  `!` : logical negation
  `-` : arithmetic negation
  `~` : bitwise negation
- arithmetic
  `+,-,*,/,%` : as you would expect
  `**` : exponentiation
- relational
  `>, <=, <=, <=` : as you would expect
- equality
  `==, !=` : as you would expect
  `<=>` : comparison, with signed result:
  - returns -1 if the left operand is less than the right;
  - returns 0 if they are equal;
  - returns +1 if the left operand is greater than the right

perl — more operators.

- assignment, increment, decrement

```
=
+=, ++
-=, --
*=, **=, /=, %=
&&=, ||=
```

- just like in C

perl — regular expressions.

- simplest regular expression is a literal string
- complex regular expressions use *metacharacters* to describe various options in building a pattern... *"I never metacharacter I didn't like"*
- metacharacters:

`\`	escapes the character immediately following it
`.`	matches any single character except newline
`^`	matches at the beginning of a string
`$`	matches at the end of a string
`*`	matches the preceding element 0 or more times
`+`	matches the preceding element 1 or more times
`?`	matches the preceding element 0 or 1 times
`{ ... }`	specifies a range of occurrences for the element preceding it
`[ ... ]`	matches any one of the class of characters in the brackets
`( ... )`	groups expressions
`|`	matches either the expression before or after it

note that there are some exceptions to these rules

perl — pattern matching.

- `=~` binds a scalar to a patterm match, substitution or translation
- `!~` just like above, except that the return value is negated in the logical sense
- operators:
  - `m/pattern/gimosx` : match
    - g = match globally (all instances)
    - i = do case insensitive matching
    - note that first m is optional
  - `s/pattern/replacement/egimosx` : search
    - e = evaluate right side as an expression
    - g = match globally (all instances)
    - i = do case insensitive matching
  - `y/pattern1/pattern2/cds` : translate
    - c = complement pattern1
    - d = delete found but unreplaced characters
    - s = squash duplicate replaced characters

perl — pattern matching, example 1.

- example

```
#!/usr/bin/perl

$s = "hello world";
print '$s=[',$s,"]\n";

if ($s =~ m/x/) { print "there's an x in ",$s,"\n" }
else { print "there isn't\n" }

if ($s =~ m/L/i) { print "there's an l in ",$s,"\n" }
else { print "there isn't\n" }
```

- output:

```
$s=[hello world]
there isn't
there's an l in hello world
```

perl — pattern matching, example 2.

- example

```
#!/usr/bin/perl

$s = "hello world";
print '$s=[',$s,"]\n";

$t = ($s =~ s/l/x/g);
print '$t=[',$t,"]\n";
print '$s=[',$s,"]\n";
```

- output:

```
$s=[hello world]
$t=[3]
$s=[hexxo worxd]
```

perl — pattern matching, example 3.

- example

```
#!/usr/bin/perl

$s = "hello world";
print '$s=[',$s,"]\n";

$u = ($s =~ y/l/o/c);
print '$u=[',$u,"]\n";
print '$s=[',$s,"]\n";
```

- output:

```
$s=[hello world]
$u=[8]
$s=[oolloooolo]
```

perl — subroutines.

- syntax for defining:

```
sub name {block}
sub name (proto) {block}
```

- where `proto` is like a prototype, where you put in sample arguments
- syntax for calling:

```
name(args);
name args;
```

- any arguments passed to a subroutine come in as the array `@_`
- you can use the `return` statement, like in C

perl — files, aka filehandles.

- `open( FILEHANDLE, filename );` : to open a file for reading
  `open( FILEHANDLE, >filename );` : to open a file for writing
  `open( FILEHANDLE, >>filename );` : to open a file for appending
- use `|| warn print "message";` or `|| die print "message";` for error checking
- `print FILEHANDLE, ...;`
- `close( FILEHANDLE );`
- example:

```
#!/usr/bin/perl

open( MYFILE,">a.dat" );
print MYFILE "hi there!\n";
print MYFILE "bye-bye\n";
close( MYFILE );
```

perl — filehandles, another example.

```
#!/usr/bin/perl

open( MYFILE2,"b.dat" ) || warn "file not found!";
open( MYFILE2,"a.dat" ) || die "file not found!";
while ( <MYFILE2> ) { print "$_\n" }
close( MYFILE2 );
```

perl — built-in functions.

- here are a few:
- `chomp $var`
  `chomp @list`
  removes any line-ending characters
- `chop $var`
  `chop @list`
  removes last character
- `chr number`
  returns the character represented by the ASCII value number
- `eof filehandle`
  returns true if next read on filehandle will return end-of-file
- `exists $hash{$key}`
  returns true if specified hash key exists, even if its value is undefined
- `exit`
  exits the perl process immediately

- `getc filehandle`
  reads next byte from filehandle
- `index string, substr [, start]`
  returns position of first occurrence of substr in string, with optional starting position; also `rindex` which is index in reverse
- `opendir dirhandle, dirname`
  opens a directory for processing, kind of like a file; use `readdir` and `closedir` to process
- `split /pattern/, string [, limit]`
  splits string into a list of substrings, by finding delimiters that match pattern;
  example: `split /([-,])/,"1-10,20";` returns `(1, '-', 10, ',', 20)`
- `substr string, pos [, n, replacement]`
  returns substring in string starting with position pos, for n characters

perl — etc.

- there are lots and lots of advanced and funky things you can do in perl; this is just a start!
- here's a quick start reference:
  `http://www.comp.leeds.ac.uk/Perl/`
- the main perl page is:
  `http://www.perl.com`
- documentation is here (linked from above):
  `http://www.perl.com`
- function reference list is here:
  `http://www.perldoc.com/perl5.6/pod/perlfunc.html`