Perl

Practical Extraction and Report Language

Scalar Data

- Number or string of characters
- Numbers
  - Internally, all numbers are double-precision floating point
- Single-quoted strings
  - Any character, with 2 exceptions, is legal between quotes
    - \' prints as '
    - \\ prints as \

Scalar Data

- Double-quoted strings
  - Backslash specifies control characters, or, through octal and hex notation, any character
    - \n - newline
    - \r - carriage return
    - \t - tab
    - \b - backspace
    - \a - bell
    - \e - escape
    - \007 - octal ASCII value
    - \x87 - hex ASCII value

Scalar Data

- Double-quoted strings
  - \cC - control character (here, control C)
  - \\ - backslash
  - \" - double quote
  - \l - lowercase next letter
  - \L - lowercase all following letters until \E
  - \u - uppercase next letter
  - \U - Uppercase all following letters until \E
  - \E - terminates \L or \U

Scalar Data

- Numeric operators
  - Addition (+)
  - Subtraction (-)
  - Multiplication (*)
  - Division (/)
  - Exponentiation (**)
  - Modulus (%)
  - Comparison operators
    - <, <=, ==, >=, >, !=

Scalar Data

- String operators
  - Concatenation (.)
  - String repetition (x)
    - "bill"x4 is "billbillbillbill"
    - (4+2) x 3 is 6 x 3 or "6" x 3 or "666"
    - "a"x3.7 is "aaa"
    - "a"x0 is ""

Scalar Data

- String operators
  - Comparison operators
    - lt, le, eq, ge, gt, ne
      - 7 < 30 is true
      - 7 lt 30 is false

Scalar Data

- Conversion between strings and numbers
  - String used as operand for numeric operator
    - String converted to its equivalent numeric value
      - " 45.67bill" converts to 45.67
      - "bill" converts to 0

Scalar Data

- Conversion between strings and numbers
  - Numeric value used as operand for string operator
    - Value calculated and converted into string
      - "z" . (3 + 5) is "z8"

Scalar Data

- Scalar variables
  - Begin with $, followed by letter, possibly followed by more letters, digits, or underscores
  - Case sensitive

Scalar Data

- Variable assignment
  - Value of a scalar assignment is the value of the right-hand side
  - $z = 3 * ($y = 2);
  - $a = $b = 7;

Scalar Data

- Binary assignment operators
  - $a = $a + 8;

    $a += 8;
  - $string = $string . "cat";

    $string .= "cat";

Scalar Data

- Autoincrement and autodecrement
  - $z = 8;
    $y = ++$z; #$z and $y are both 9
  - $z = 8;
    $y = $z++; #$y is 8, but $z is 9
  - $z = 8.5;
    $y = ++$z; #$z and $y are both 9.5

Scalar Data

- chop( )
  - Returns last character of string argument
  - Has side effect of removing this last character
- Interpolation
  - $a=7; print "Value is $a";
    - Prints **Value is 7**
  - $b='Bill'; print "Name is $b";
    - Prints **Name is Bill**
  - $b='Bill'; print "Name is \$b";
    - Prints **Name is $b**

Scalar Data

- <STDIN> as a scalar value
  - $a = <STDIN>;
    chop($a);
  - chop($a=<STDIN>);
- Output
  - print("Hi there\n");
  - print "Hi there\n";

Scalar Data

- undef
  - Value of undefined variable
    - Looks like zero when used as a number
    - Looks like the empty string when used as a string
    - <STDIN> returns undef when input is control-D

Arrays

- Literals
  - (4,5,6)
  - ("Bill",7,8,9)
  - ($a,8)
  - ($z+$h,7,8)
  - (1..5,4,7) # (1,2,3,4,5,4,7)
  - (1.3..5.3) # (1.3,2.3,3.3,4.3,5.3)

Arrays

- Literals
  - (2.3..7.1) # (2.3,3.3,4.3,5.3,6.3)
  - () # the empty array
  - print (5, 6, "cat", $a)

Arrays

- Array variables
  - Begins with @
- Array assignment
  - @a=(1,2,3);
    @b=@a;
  - @z=5; # @z is (5)
  - @z=("b",6);
    @w=(4,5,@z,9); # @w is (4,5,"b",6,9)

Arrays

- Array assignment
  - ($a,$b,$c)=(4,5,6) # $a=4, $b=5, $c=6
  - ($a,$b)=($b,$a) # switches values of $a and $b
  - ($z,@w)=($a,$b,$c) # $z is $a, @w is ($b,$c)

Arrays

- Array assignment
  - $a=(4,5,6); # $a is 6, from the comma operator

    ($a)=(4,5,6); # $a is 4

    @x=(4,5,6);

    $a=@x; # $a is 3, the length of @x

    ($a)=@x; # $a is 4

Arrays

- Element access

```
@x=(4,5,6);
$a=$x[0];  # $a is 4
$x[0]=9;  # @x is (9,5,6)
$x[2]++;  # @x is (9,5,7)
$x[1] += 5;  # @x is (9,10,7)
($x[0],$x[1])=($x[1],$x[0]);  # @x is
 (10,9,7)
```

Arrays

- Element access

```
@x[0,1]=@x[1,0];  # @x is (9,10,7)
@x[0,1,2]=@x[1,1,1];  # @x is (10,10,10)
$b=$x[8];  # $b is undef
$x[4] = "b";  # @x is (10,10,10,undef, "b")
$#x=3;    #  $#x  is  index  value  of  last
element of @x, so @x is (10,10,10,undef)
```

Arrays

- push( ) and pop( ) operators

```
push(@x,$a);  # @x = (@x,$a)
push(@x,6,7,8);  # @x = (@x,6,7,8)
$last=pop(@x);   #  removes  last  element
of @x
```

- shift( ) and unshift( ) operators

```
unshift(@x,6,7,8);  # @x = (6,7,8,@x)
$first=shift(@x);  # ($first,@x) = @x
```

Arrays

- reverse( ) operator

```
@x = (6,7,8);
@y = reverse(@x);  # @y = (8,7,6)
```

- sort( ) operator

```
@x = (1,2,4,8,16,32,64);
@x = sort(@x);  #@x = (1,16,2,32,4,64,8)
```

Arrays

- chop( ) operator

```
@x = ("bill\n", "frank\n");
chop(@x);  # @x = ("bill", "frank")
```

Arrays

- Scalar and array contexts
  - Operator expecting an operand to be scalar is being evaluated in *scalar context*
  - Operator expecting an operand to be an array is being evaluated in *array context*

Arrays

- Scalar and array contexts
  - Concatenating a null string to an expression forces it to be evaluated in scalar context

```
@x = ("x", "y", "z");
print ("There are ",@x, " elements\n");  # prints "xyz" for @x
print ("There are ", ""."@x, " elements\n");  # prints 3 for @x
```

Arrays

- <STDIN> as an array
  - In scalar context, <STDIN> returns next line of input
  - In array context, <STDIN> returns all the remaining lines of input up to the end of file
    - Each line returned as a separate element of list

      @x = <STDIN>

      If user types 4 lines, then presses control-D (to indicate end-of-file), the array has 4 elements

Control Structures

- Statement blocks

```
{
  first_statement;
  ...
  last_statement;
}
```

Control Structures

- if/unless statement

```
if (expression) {
        first_true_statement;
        ...
        last_true_statement;
}
```

Control Structures

- if/unless statement

```
if (expression) {
        first_true_statement;
        ...
        last_true_statement;
} else {
        first_false_statement;
        ...
        last_false_statement;
}
```

Control Structures

- if/unless statement

```
if (expression_one) {
        one_true_statement_first;
        ...
        one_true_statement_last;
} elseif (expression_two) {
        two_true_statement_first;
        ...
        two_true_statement_last;
} elseif (expression_three) {
        three_true_statement_first;
        ...
        three_true_statement_last;
} else {
        false_statement_first;
        ...
        false_statement_last;
}
```

Control Structures

- if/unless statement

```
unless (expression) {
        statement_one;
} else {
        statement_two;
}
```

Control Structures

- These expressions are the only ones which evaluate to false
  - ""
  - ( )
  - "0"
  - undef
  - 0

Control Structures

- while/until statement

```
while (expression) {
        statement_first;
        ...
        statement_last;
}
```

Control Structures

- while/until statement

```
until (expression) {
        statement_first;
        ...
        statement_last;
}
```

Control Structures

- for statement

```
for (initial_exp; test_exp; increment_exp)
{
        statement_first;
        ...
        statement_last;
}
```

Control Structures

- for statement

```
initial_exp;
while (test_exp) {
        statement_first;
        ...
        statement_last;
        increment_exp;
}
```

Control Structures

- foreach statement

```
foreach $a (@some_list) {
        statement_first;
        ...
        statement_last;
}
```

Control Structures

- foreach statement

```
@x = (1,2,3,4,5);
 foreach $y (reverse @x) {
        print $y;
 }  # prints 54321
```

Control Structures

- foreach statement
  - If scalar variable is missing, $_ is assumed

```
@x = (1,2,3,4,5);
foreach (reverse @x) {
        print;
}  # prints 54321
```

Associative Arrays

- List of items indexed by arbitrary scalars, called *keys*
- %x is array
- {$key} is index
  - $x{"Bill"} = 45.67;
  - $x{45.56} = "fred";

Associative Arrays

- Literal representation

```
@x_list = %x;  # x_list = ("Bill", 45.67,
45.56, "fred") or (45.56, "fred", "Bill",
45.67)
%y = @x_list;
%y = %x;
%y = ("Bill", 45.67, 45.56, "fred");
```

Associative Arrays

- keys( ) operator

```
@z = keys(%x);  # @z=("Bill",45.56) or
                (45.56, "Bill")
```

- values( ) operator

```
@z = values(%x);  # @z=(45.67, "fred")
                  or ("fred",45.67)
```

Associative Arrays

- each( ) operator
  - each (%array) returns the nextkey-value pair

```
while (($first,$last) = each(%last)) {
        print "The last name of $first is
        $last\n"
```

- delete( ) operator

```
%x = ("aa", "bb", 5.6, 8.77);
delete $x{"aa"}
```

Basic I/O

- Input from <STDIN>

```
while ($_ = <STDIN>) {
        chop $_;
        ...
}

while (<STDIN>) {
        chop;
        ...
}
```

Basic I/O

- Input from the diamond operator <>
  - Gets data from files specified on the command line that invoked given Perl program
    - If no files on command line, the diamond operator reads from standard input

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

Basic I/O

- Input from the diamond operator <>
  - Actually, diamond operator gets its input by examining @ARGV, the array which is a list of command line arguments
    - Can set this array in program

```
@ARGV = ("aa", "bb", "cc");
while (<>) {  # processes files aa, bb, and cc
  print "This line of the file is: $_";
}
```

Basic I/O

- Output to STDOUT
  - print operator
    - Argument is a list of strings
    - Return value is true or false, depending on success of the output

```
$x = print("Hello", " world", "\n");
```

  - Formatted output

```
printf "%9s  %4d  %12.5f\n", $s, $n, $r;
```

Regular Expressions

- Enclosed in-between slashes
- Checking for occurrence in $_

```
if (/abc/) {
  print $_;
}
while (<>) {
  if (/ab*c/) {
        print $_;
  }
}
```

Regular Expressions

- Single-character patterns
  - A single character matches itself
  - The dot . matches any single character except the newline (\n)

Regular Expressions

- Single-character patterns
  - A character class is represented by a list of characters enclosed between [ ]
    - For the pattern to match, one and only one of these characters must be present
    - [abcde]
    - [aeiouAEIOU]
    - [0-9]
    - [a-zA-Z0-9]

Regular Expressions

- Single-character patterns
  - Negated character class
    - [^0-9]
      - Matches any character except a digit
    - [^aeiouAEIOU]
      - Matches any single non-vowel

Regular Expressions

- Single-character patterns
  - Predefined character classes
    - \d
      - digits
      - [0-9]
    - \D
      - not digits
      - [^0-9]

Regular Expressions

- Single-character patterns
  - Predefined character classes
    - \w
      - words
      - [a-zA-Z0-9_]
    - \W
      - not words
      - [^a-zA-Z0-9_]
    - \s
      - space
      - [\r\t\n\f]
    - \S
      - not space
      - [^\r\t\n\f]

Regular Expressions

- Grouping patterns
  - Sequence
    - *abc* matches *a* followed by *b* followed by *c*
  - Multipliers
    - *
      - 0 or more
    - +
      - 1 or more
    - ?
      - 0 or 1

Regular Expressions

- Grouping patterns
  - Multipliers
    - {n,m}
      - Between n and m occurrences inclusive
    - {n,}
      - At least n occurrences
    - {n}
      - Exactly n occurrences
    - Thus, * is {0,}, + is {1,}, and ? is {0,1}

Regular Expressions

- Alternation
  - /bill|fred/
  - /a|b|c/ is the same as /[abc]/

Regular Expressions

- All matching is greedy
  - Longest expression matches first
  - In "a yyy c yyyyyyyy c yyy d", the pattern "a.*c.*d" induces a match of the first ".*" on " yyy c yyyyyyyy "
  - Consider the pattern "a.*cx.*d" matching the string "a yyy cx yyyyyyyy cy yyy d"
    - The first ".*" first matches the substring " yyy cx yyyyyyyy ", and then backtracking occurs

Regular Expressions

- Parentheses as memory
  - Parentheses around a pattern causes the part of the string which matches this pattern to be remembered, so that it can be later referenced
  - Consider /bill(.)fred\1/ versus /bill.fred./
  - /a(.)b(.)c\2d\1/
  - /a(.*)b\1c/

Regular Expressions

- Anchoring patterns
  - Normally, beginning of pattern is shifted through the string from left to right
  - Anchoring ensures that parts of the pattern match with particular parts of the string

Regular Expressions

- Anchoring patterns
  - \b requires a word boundary at the given spot in order for the pattern to match
    - Word boundary is place between \w and \W or between \w and the start or end of the string
      - /car\b/ matches car but not cars
      - /\bbarb/ matches "barb" and "barbara", but not "ebarb"
      - /\bport\b/ matches "port", but neither "sport" nor "ports"

Regular Expressions

- Anchoring patterns
  - \B requires that there not be a word boundary
    - /\bFred\B/ matches "Frederick" but not "Fred Flintstone"
  - The caret (^) matches the beginning of the string if it makes sense
    - /^a/ matches "a" if it is the first character of the string
    - /a^/ matches "a^" anywhere in the string

Regular Expressions

- Anchoring patterns
  - The dollar sign ($) matches the end of the string if it makes sense

Regular Expressions

- Precedence
  - Parentheses
    - ( )
  - Multipliers
    - + * ? {n,m}
  - Sequence and anchoring
    - abc ^ $ \b \B
  - Alternation
    - |

Regular Expressions

- Selecting a different target
  - Want to match a variable other than $_
  - The operator =~ matches the left-hand argument

```
$a = "hello world";
$a =~ /^he/;  # true
$a =~ /(.)\1/;    # true, matches the double l
if ($a =~ /(.)\1/) {  # true
        ...
}
```

Regular Expressions

- Selecting a different target

```
if (<STDIN> =~ /^[yY]/) {
  ...
}

if (<STDIN> =~ /^y/i) {   # i ignores case
  ...
}
```

Regular Expressions

- Using a different delimiter
  - If regular expression contains "/", precede it with a "\"

```
$path = <STDIN>;
if ($path =~ /^\/usr\/etc/) {
        ...
}
```

Regular Expressions

- Using a different delimiter

```
$path = <STDIN>;
        if ($path =~ m#^/usr/etc#) {
            ...
        }
```

Regular Expressions

- Using variable interpolation

```
$what = "is";
$sentence = "This is good.";
if ($sentence =~ /\b$what\b/) {
        print "This sentence contains the
                word $what.\n"
}
```

Regular Expressions

- Special read-only variables
  - After successful match, $1, $2, ... are set to same values as \1, \2, ...

```
$_ = "This is good";
/(\w+)\W+(\w+)/;  # matches first two
                  words
# $1 is "This" and $2 is "is"
```

Regular Expressions

- Special read-only variables

```
$_ = "This is good";
($first,$second) = /(\w+)\W+(\w+)/;
# $first is "This" and $second is "is"
```

Regular Expressions

- Special read-only variables
  - $& is the part of the string that matches the regular expression
  - $` is the part of the string before the part that matched the regular expression
  - $' is the part of the string after the part that matched the regular expression

Regular Expressions

- Special read-only variables

```
$_ = "this is a sample string";
/sa.*le/;  # matches "sample"
# $` is "this is a "
# $& is "sample"
# $' is " string"
```

Regular Expressions

- Substitutions
  - s/regular_expression/replacement_string/
  - Replaces first occurrence

```
$_ = "bill yyyyyyy fred";
s/y*/tom/;  # $_ is now "bill tom fred"
```

  - Flag "g" substitutes all occurrences

```
$_ = "foot fool buffoon";
s/foo/bar/g;    # $_ is now "bart barl bufbarn"
```

Regular Expressions

- Substitutions

```
$_ = "hello world";
$new = "goodbye";
s/hello/$new/;  # $_ is now "goodbye
                world"
```

Regular Expressions

- Substitutions

```
$_ = "this is a test";
 s/(\w+)/<$1>/g;   # $_ is now "<this>
<is> <a> <test>"

$x[$j] =~ s/here/there/;

$d{"abc"} =~ s/there/here/;
```

Regular Expressions

- split( ) operator
  - Takes as arguments a regular expression and a string, looks for all occurrences of the regular expression in the string, and the parts of the string that don't match the regular expression are returned in sequence as a list of values

Regular Expressions

- split( ) operator

```
$line="67;bill;;/usr/bin;joe";
@fields=split(/;/,$line);
# @fields is ("67","bill","","/usr/bin","joe")

@fields=split(/;+/,$line);
# @fields is ("67", "bill", "/usr/bin", "joe")
```

Regular Expressions

- split( ) operator
  - *split(/regex/)* is the same as *split(/regex/,$_)*
  - *split* is the same as *split(/\s+/,$_)*

Regular Expressions

- join ( ) operator
  - Inverse of split( )
  - Takes a list of values and puts them together with a separator element between each pair of items

```
$line = join(";",@fields);  # see previous slides
```

Functions

- Defining a function

```
sub repeat {
   print "Hello, $word\n"
}
```

  - Subroutine definitions can be anywhere in program text
  - Subroutine definitions are global
    - For two subroutines with same name, the latter one overrides the former one
  - By default, any variable reference inside a subroutine is global

Functions

- Invoking a function
  - &repeat;
  - $x = 5+&repeat;
- Return Values
  - Return value of a subroutine is the value of the logically last expression evaluated

```
sub sum_of_a_and_b {
  $a+$b;
}
```

Functions

- Return values

```
$a=7; $b=8;
$x=3*&sum_of_a_and_b;

sub list_of_a_and_b {
  ($a,$b);
}

$a=7; $b=8;
@x=&list_of_a_and_b;
```

Functions

- Return values

```
sub a_or_b {
  if ($a>0) {
        print "Write a: $a\n";
        $a;
  } else {
        print "Write b: $b\n";
        $b;
  }
}
```

Functions

- Arguments
  - Invoking a subroutine with arguments in parentheses puts these arguments into a list denoted by @_ for the duration of the subroutine

Functions

- Arguments

```
sub say {
        print "$_[0], $_[1]!\n";
  }

&say("goodbye", "cruel world");
&say("hello", "world");
```

Functions

- Arguments
  - @_ is local to the subroutine

```
sub add {
   $sum = 0;
   foreach $_ (@_) {
            $sum += $_;
   }
   $sum;
}

$a = &add(4,5,6);  # adds 4, 5, 6 and assigns 15 to $a
print &add(6,7,8,9);
print &add(6..9);
```

Functions

- Local Variables

```
sub bigger_than {
  local($n,@values);
  ($n,@values) = @_;  # or local($n,@values)
        =@_
  local(@result);
  foreach $_ (@values) {
        if ($_ > $n) {
              push(@result,$_);
        }
  }
  @result;
}
```

Functions

- Local variables

  @new=&bigger_than(100,@list); #@new gets elements of @list > 100

  @x = &bigger_than(5,1,5,15,67); #@x= (15, 67)

Miscellaneous Control Structures

- last
  - Exits a loop
- next
  - Begins another loop iteration
- redo
  - Jumps to the top of a loop without beginning another iteration

Miscellaneous Control Structures

```
@movies = ("Star Wars", "Porky's", "Rocky 5",
  "Terminator", "Babe");
@ratings = ("PG", "R", "PG-13", "R", "G");

for ($j=0, $j<=4, $j++) {
  next if $ratings[$j] eq 'R';
  print "Would you like to see $movie[$j]?";
  chop($answer = <>);
  last if $answer eq 'yes';
}
if ($answer eq 'yes') {print "That will be
  \$4.25\n"}
else {print "Sorry you don't see anything you
  like.\n"}
```

Miscellaneous Control Structures

```
print "Type in 4 digits (0 through 9)\n\n"
for ($j=1; $j<=4; $j++) {
   print "Choice $j:";
   chop($digit[$j] = <>);
   redo if $digit[$j] > 9;
   redo if $digit[$j] < 0;
}
print "Your choices: @digit \n";
```

Miscellaneous Control Structures

- Labelled Blocks

```
OUTER: for ($i=1; $i<=10; $i++) {
  INNER: for ($j=1; $j<=10; $j++) {
      if ($i*$j == 25) {
          print "$i times $j is 25!\n";
          last OUTER;
      }
      if ($j > $i) {
          next OUTER;
      }
  }
}
```

Miscellaneous Control Structures

- Expression Modifiers
  - exp2 if exp1; # if (exp1) {exp2;}
  - exp2 unless exp1; # unless (exp1) {exp2;}
  - exp2 while exp1; # while (exp1) {exp2;}
  - exp2 until exp1; # until (exp1) {exp2;}

Miscellaneous Control Structures

- &&, ||, and ?: as control structures
  - exp1 && exp2
    - if (exp1) {exp2;}
  - exp1 || exp2
    - unless (exp1) {exp2;}
  - exp1 ? exp2 : exp3;
    - if (exp1) {exp2;} else {exp3;}

Filehandles and File Tests

- Filehandles
  - I/O connection name
  - STDIN
  - STDOUT
  - STDERR

Filehandles and File Tests

- Opening and closing a filehandle
  - open(FILEHANDLE, "external_file_name")
    - Opens file for reading
  - open(FILEHANDLE, ">external_file_name")
    - Opens file for writing
  - open(FILEHANDLE, ">>external_file_name")
    - Opens file for appending

Filehandles and File Tests

- Opening and closing a filehandle
  - close(FILEHANDLE)
    - Closes file
  - These operations return true or false, depending on whether or not the operation was successful
  - A filehandle that hasn't been successfully opened can be read (you get the end-of-file at the start) or written to (data disappears)

Filehandles and File Tests

- Opening and closing a filehandle
  - The die( ) operator prints out its arguments on STDERR and then ends the process

```
unless (open(FILE, ">/x/y/data")) {
        die "File couldn't be opened\n";
}

open(FILE, ">/x/y/data") || die “File couldn’t
        be opened\n”;
```

Filehandles and File Tests

- Using filehandles

```
open(IN,"$a")  ||  die  "Cannot  open  $a  for
  reading";
open(OUT, ">$b") || "Cannot create $b";
while (<IN>) {
  print OUT $_;
}
close(IN);
close(OUT);
```

Filehandles and File Tests

- File tests

```
print "What file? ";
$filename=<STDIN>;
chop($filename);
if (-r $filename && -w $filename) {
   # file exists and I can read and write it
   ...
}
```

Filehandles and File Tests

- File tests
  - -x
    - File is executable
  - -e
    - File or directory exists
  - -o
    - File or directory is owned by user

Filehandles and File Tests

- File tests
  - -z
    - File exists and has zero size
  - -s
    - File or directory exists and has nonzero size
      - Return value is the size in bytes
  - -f
    - Entry is a file
  - -d
    - Entry is a directory

Filehandles and File Tests

- stat( ) operator
  - stat(FILEHANDLE)
    - Returns 13-element array, where stat(FILEHANDLE)[7] is the size of the file in bytes

Formats

- Defining a Format

```
format ADDRESSLABEL =
==========================================
| @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$name
| @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$address
| @<<<<<<<<<<<<<<<<<<, @< @<<<< |
$city, $state, $zip
==========================================
.
```

Formats

- Invoking a Format

```
open(ADDRESSLABEL, ">labels") || die "Can't
  create";
open(ADDRESSES, "addresses") || die "Can't
  open addresses";
while (<ADDRESSES>) {
  ($name,$address,$city,$state,$zip) = split(/:/);
  write ADDRESSLABEL;  # This is filehandle
  # by default, format of same name is also used
}
```

Formats

- Text Fields
  - @<<<<
    - 5 spaces left-justified
  - @>>>>
    - 4 spaces right-justified
  - @| | | | |
    - 6 spaces centered
- Numeric Fields
  - @###.##

Formats

- sprintf operator
  - Returns as a string whatever printf would have printed

```
format MONEY =
Assets:          @<<<<<<<<<<          Liabilities
  @<<<<<<<<< Net: @<<<<<<<<<<
$func($assets,10), &func($liab,9), $func($assets
  - $liab,10)
.
```

Formats

- sprintf operator

```
sub func {
  local($n, $width) = @_;
  $width - = 2;
  $n = sprintf("%.2f", $n);
  if ($n < 0) {
        sprintf("[%$width.2f]", -$n);
  } else {
        sprintf(" %$width.2f ", $n);
  }
}
```

Formats

- Multiline fields

```
format STDOUT =
Text before.
@*
$long_string
Text after.
.

$long_string = "Bill\nFred\nSue\nTom\n";
write;
```

Formats

- Multiline fields

  Text before.
  Bill
  Fred
  Sue
  Tom
  Text after.

Formats

- Filled fields
  - Allows one to create a paragraph where words are broken at word boundaries and lines are wrapped as needed
  - Substitute ^ for @

Formats

- Filled fields

```
format try =
Name: @<<<<<<<<<< Comment:
      ^<<<<<<<<<<<<<<<<<
$name, $comment

      ^<<<<<<<<<<<<<<<<<
$comment

      ^<<<<<<<<<<<<<<<<<
$comment
.
```

  - $comment is changed each time to what is not yet printed

Formats

- Filled fields

```
format try =
Name:          @<<<<<<<<<<          Comment:
  ^<<<<<<<<<<<<<<<<<<
$name, $comment
~
  ^<<<<<<<<<<<<<<<<<<
$comment
~
  ^<<<<<<<<<<<<<<<<<<
$comment
.
```

  - ~ suppresses line if only blanks would print

Formats

- Filled fields

```
format try =
Name:          @<<<<<<<<<<          Comment:
  ^<<<<<<<<<<<<<<<<<
$name, $comment
~~                          ^<<<<<<<<<<<<<<<<<
                            $comment
.
```

  - ~~ repeats this line until nothing but blanks are left

Formats

- Changing the Filehandle

```
print "Hello world\n";  # prints to STDOUT
select(LOGFILE);  # select a new filehandle
print "Hello world\n";  # prints to LOGFILE
```

  - The return value from select( ) is the name of the old filehandle

```
$oldhandle = select(LOGFILE);
print "This goes to LOGFILE\n";
select($oldhandle);    # restores the previous
  filehandle
```

Formats

- Changing the Format Name
  - Default format name for a filehandle is the same as the filehandle
  - Contained in variable $~
  - Set format for REPORT filehandle to NEW

```
$oldhandle = select(REPORT);
$~ = "NEW";
select($oldhandle);
```

Data Transformations

- Finding a substring
  - $y = index($string, $substring);
    - Scans from left-to-right

```
$y = index("car", "a");  # $y is 1
$z = "fred";
$y = index($z, $z);  # $y is 0
$y = index($z, "ed");  # $y is 2
$y = index($z, "car");  # $y is -1
```

Data Transformations

- Finding a substring
  - $x = index($string, $substring, $skip);
    - Third parameter is the minimum value returned by index

```
$where=index("hello world", "l");  # $where is 2
$where=index("hello world", "l",0); #$where is 2
$where=index("hello world", "l",1); #$where is 2
$where=index("hello world", "l",3); #$where is 3
```

Data Transformations

- Finding a substring
  - $x = rindex("string", "substring", $skip)
    - Scans right-to-left

```
$x=rindex("hello world", "he");  # $w is 0
$x=rindex("hello world", "l");  # $w is 9
$x=rindex("hello world", "o",6);  # $w is 4
```

Data Transformations

- Extracting and replacing a string
  - $s = substr($string, $start, $length);

```
$x = "hello world!";
$y = substr($x, 3, 2);  # $y is "lo"
$y = substr($x, 6, 400);  # $y is "world!"
$z = substr(“10000000000”, 0, $power+1)
```

  - If length < 0, the empty string is returned

Data Transformations

- Extracting and replacing a string
  - $start can be less than zero
    - Start that many characters from the right

```
$y = substr("This is a string", -3, 3);  # $y is "ing"
$y = substr("This is a string", -3, 1);  # $y is "i"
```

  - Can omit $length
    - Takes entire string from $start position

```
$y = substr("This is a string", 5);  # $y is "is a string"
```

Data Transformations

- Extracting and replacing a string

```
$x = "hello world!";
substr($x, 0, 5) = "hi there";   # $x is "hi there
  world!"
substr($x, -6, 5) = "people";   # $x is "hello
  people!"
```

Data Transformations

- Transliteration
  - Translate one character to another
  - Operates by default on $_

```
$_ = "bill and fred";
tr/bf/fb/;  # $_ is "fill and bred"
tr/abcde/ABCDE/;  # $_ is "fill AnD BrED"
tr/a-z/A-Z/;  # $_ is "FILL AND BRED"
tr/A-Z/yx/;  # $_ is "xxx yxx xxxx"
$_ = "bill and fred";
tr/a-z/ABCDE/d;  # $_ is "B AD ED" (delete tag)
```

Data Transformations

- Transliteration
  - Return value is number of characters changed

```
$_ = "bill and fred";
$count = tr/a-z//;  # $_ is unchanged but $count is 11
```

    - If the second list is empty and there is no *d* option, then result list is same as the original list

Data Transformations

- Transliteration
  - Complement Tag
    - Any character in first string is removed from set of 256 possible characters and remaining characters, taken in sequence, form the resulting first string

```
$_ = "bill and fred";
$count = tr/a-z//c;  # $_ is unchanged, $count is 2 (non-lowercase letters are counted)
tr/a-z/_/c;      # $_ is "bill_and_fred", non-lowercase letters are converted to "_"
tr/a-z//cd;   # $_ is "billandfred", non-lowercase letters deleted
```

Data Transformations

- Transliteration
  - Squeeze tag

```
$_ = "aaabbbcccdefghi";
tr/defghi/abcddd/s;  # $_ is "aaabbbcccabcd"
$_ = "bill and fred, jerry and sue";
tr/a-z/X/s;  # $_ is "X X X, X X X"
$_ = "bill and fred, jerry   and sue";
tr/a-z/_/cs;  # $_ is
  'bill_and_fred,_jerry_and_sue"
```

Data Transformations

- Transliteration
  - Other targets

```
$x = "bill and fred";
$x ~= tr/aeiou/X/;  # $x is "bXll Xnd frXd"
```