CHALMERS UNIVERSITY OF TECHNOLOGY

Examination in Bioinformatics, MVE360

Monday 16 March 2015, 08:30-12:30

Examiner:	Graham Kemp (telephone 772 54 11, room 6475 EDIT) The examiner will visit the exam room at 09:30 and 11:30.
Results:	Will be published by 8 April 2015 at the latest.
Exam review:	See course web page for time and place: http://www.cse.chalmers.se/edu/year/2015/course/MVE360/
Grades:	Grades for Chalmers students are normally determined as follows: ≥ 48 for grade 5; ≥ 36 for grade 4; ≥ 24 for grade 3. Grades for GU students are normally determined as follows: ≥ 42 for grade VG; ≥ 24 for grade G.
Help material:	A Chalmers-approved calculator is permitted. English language dictionaries are allowed.

Specific instructions:

- Check that you have received:
  - question paper (5 pages)
  - extract from EMBL nucleotide sequence database entry AC009360 (1 page)
  - example output for question 5 (1 page)
  - Perl reference guide (4 pages)
- Please answer in English where possible. You may clarify your answers in Swedish if you are not confident you have expressed yourself correctly in English.
- Begin the answer to each question on a new page.
- Write clearly; unreadable = wrong!
- Fewer points are given for unnecessarily complicated solutions.
- Indicate clearly if you make any assumptions that are not given in the question.
- Write the page number and question number on every page.

Question 1. 4 p

a) Consider phylogeny with four "operational taxonomic units" (OTUs) or external nodes (A,B,C and D). Draw all possible unrooted and rooted trees.

b) Consider the sequence alignment below. What columns are "informative" when considering a maximum parsimony method of tree construction.

```
A.  GACCTG
B.  GACCTG
C.  GTCGCC
D.  GTCACG
```

(4p)

Question 2. 4 p

Using a gap score of -2 and match/mismatch scores taken from the PAM250 substitution matrix (given below), derive the score matrix for a global alignment of "RFSN" with "YTQC".

In this case, what is the score of an optimal global alignment?
Give the alignment(s) with this score.

PAM250 substitution matrix:

```
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V
A  2
R -2  6
N  0  0  2
D  0 -1  2  4
C -2 -4 -4 -5  4
Q  0  1  1  2 -5  4
E  0 -1  1  3 -5  2  4
G  1 -3  0  1 -3 -1  0  5
H -1  2  2  1 -3  3  1 -2  6
I -1 -2 -2 -2 -2 -2 -2 -3 -2  5
L -2 -3 -3 -4 -6 -2 -3 -4 -2  2  6
K -1  3  1  0 -5  1  0 -2  0 -2 -3  5
M -1  0 -2 -3 -5 -1 -2 -3 -2  2  4  0  6
F -4 -4 -4 -6 -4 -5 -5 -5 -2  1  2 -5  0  9
P  1  0 -1 -1 -3  0 -1 -1  0 -2 -3 -1 -2 -5  6
S  1  0  1  0  0 -1  0  1 -1 -1 -3  0 -2 -3  1  3
T  1 -1  0  0 -2 -1  0  0 -1  0 -2  0 -1 -2  0  1  3
W -6  2 -4 -7 -8 -5 -7 -7 -3 -5 -2 -3 -4  0 -6 -2 -5 17
Y -3 -4 -2 -4  0 -4 -4 -5  0 -1 -1 -4 -2  7 -5 -3 -3  0 10
V  0 -2 -2 -2 -2 -2 -2 -1 -2  4  2 -2  2 -1 -1 -1  0 -6 -2  4
```

(4p)

Question 3. What output is printed when the following program is run?

4 p

```
#!/usr/bin/perl

$s = "aabbacbccba"; if ( $s =~ /^(.*)c/ )       { print "a) $1\n"; }
$s = "aabbacbccba"; if ( $s =~ /^([^c]*)c/ )    { print "b) $1\n"; }
$s = "aabbacbccba"; if ( $s =~ /b(..b)/ )       { print "c) $1\n"; }
$s = "aabbacbccba"; if ( $s =~ /(.{2})(.*)\1/ ) { print "d) $1, $2\n"; }

$s = "atgaattgtt" ; $s =~ s/.tt/ctt/g  ; print "e) $s\n";
$s = "atgaattgtt" ; $s =~ s/g[ta]+/x/  ; print "f) $s\n";
$s = "atgaattgtt" ; $s =~ s/t[^g]*/a/g ; print "g) $s\n";
$s = "atgaattgtt" ; $s =~ tr/at/cg/    ; print "h) $s\n";
```

(4p)

Question 4. (Refer to the extract from EMBL entry AC009360 — see attached sheet.)

6 p

The OC (Organism Classification) lines in an EMBL format file contain the taxonomic classification of the source organism:

> "The classification is listed top-down as nodes in a taxonomic tree in which the most general grouping is given first. The classification may be distributed over several OC lines, but nodes are not split or hyphenated between lines. Semicolons separate the individual items and the list is terminated by a period."

Write a Perl program that reads an EMBL database file whose name is specified on the command line and writes the names of the nodes in the taxonomic classification of the source organism. The output should have one name per line, and each line should contain a number that indicates the node's depth in the tree. For example, the output for EMBL entry AC009360 should be:

```
1 Eukaryota
2 Viridiplantae
3 Streptophyta
4 Embryophyta
5 Tracheophyta
6 Spermatophyta
7 Magnoliophyta
8 eudicotyledons
9 Gunneridae
10 Pentapetalae
11 rosids
12 malvids
13 Brassicales
14 Brassicaceae
15 Camelineae
16 Arabidopsis
```

(6p)

Question 5. (Refer to the extract from EMBL entry AC009360 — see attached sheet.)

12 p

A recent study has looked for 'aaag$X_N$cttt' motifs in *Arabidopsis thaliana*, where X_N represents any string of N nucleotides. For example, in EMBL entry AC009360 this motif occurs between positions 157 and 180, where 'aaag' and 'cttt' are separated by 16 nucleotides ("ccagggtatcttcctg").

a) Write a Perl program that reads an EMBL database file whose name is specified on the command line and finds all 'aaag$X_N$cttt' motifs where N is between 1 and 25. For each motif found, the program should write out the matching subsequence and the value of N for that motif. The program should also count how often each separation N occurs in that database entry. Example output that your program should produce for AC009360 is shown on an attached sheet.

(8p)

b) Suppose we now want to generalise the program written in part (a) so that it finds all SX_NR motifs, where S is a string typed in by the user (e.g. 'aaag') and R is the reverse complement of S (e.g. 'cttt').

Modify your solution to part (a) so that it prompts the user to type in a string S, and then finds all SX_NR motifs.

(You do not have to rewrite the entire program — just write the new code and any parts of the solution to (a) that are changed.)

(4p)

Question 6. a) What is a stochastic process?

5 p

Markov chains are stochastic processes that have the Markov property. Write down and explain the Markov property.

(2p)

b) Briefly explain how Markov chains work as models.

Now explain hidden Markov models. What are the main difference(s) between MCs and HMMs? When do you use one or the other?

(3p)

Question 7. Explain what a pair HMM is and what it is used for. Describe its states and parameters and how it differs from a HMM. Draw a figure to represent your model.

3 p

(3p)

Question 8. 8 p

A certain motif, that we will call X, is responsible for controlling the expression of genes involved in the repair of cellular damage. Because the lack of repair of cellular damage is a crucial activity, we are interested in studying the motifs X in different species and mutants. So, we want to start by locating the motifs. However, the motifs X have low conservation in such a way that a simple pattern matching won't work. In this task you should sketch a HMM for the identification of motifs X.

a) Propose a HMM for the identification of motifs X. Describe its components and its parameters. You are welcome to draw your model here.
(2p)

b) Once you have the model, describe the next three main steps you need to achieve your goal, which is to use your HMM to search for motifs X. Explain why the steps are important and what you want to achieve in each step. Mention also the algorithms or the calculations you need in each step, if applicable.
(6p)

Question 9. 14 p

a) Describe briefly what paired-end sequencing means.
(3p)

b) You want to find mutations causing a specific type of cancer. You have tumor samples from 11 patients. Describe with a few sentences how you can use resequencing to get a list of candidate variants (SNPs and indels) that might be responsible for tumor development.
(4p)

c) Give one reason for the need of normalizing count data from metagenomics before identification of differentially abundant genes?
(3p)

d) In an RNA-seq experiment you want to identify differentially abundant transcripts in an organism with only 5 genes. Table 1 shows raw transcript abundances after mapping (number of reads mapped per gene and sample). Suggest one way of normalizing the data and calculate the normalized counts matrix. If you don't have a calculator it is okay to only explain how you do the normalization.

Table 1: Read counts per gene and sample

	Sample 1	Sample 2	Sample 3	Sample 4
Gene 1	665	292	186	217
Gene 2	113	29	11	61
Gene 3	450	129	146	104
Gene 4	71	48	11	3
Gene 5	343	144	137	61

(4p)

Extract from EMBL nucleotide sequence database entry AC009360

```
ID   AC009360; SV 3; linear; genomic DNA; STD; PLN; 28334 BP.
XX
AC   AC009360;
XX
DT   19-AUG-1999 (Rel. 60, Created)
DT   14-APR-2005 (Rel. 83, Last updated, Version 4)
XX
DE   Arabidopsis thaliana chromosome 1 BAC F16G16 sequence, complete sequence.
XX
OS   Arabidopsis thaliana (thale cress)
OC   Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
OC   Spermatophyta; Magnoliophyta; eudicotyledons; Gunneridae; Pentapetalae;
OC   rosids; malvids; Brassicales; Brassicaceae; Camelineae; Arabidopsis.
XX
CC   On Jan 5, 2001 this sequence version replaced gi:6226999.
CC   The sequence of BAC F16G16 from Arabidopsis thaliana chromosome 1.
CC   In order to facilitate the jointing of overlapping clones in the
CC   future for creation of larger contigs, we provide overlaps between
CC   overlapping submitted clones.  The 5' end of this sequence overlaps
CC   by 200 bp to the 3' end of the sequence of the clone F13O11.  The
CC   3' end of this sequence overlaps by 2000 bp to the 5' end of the
CC   clone T23K8.
XX
FH   Key             Location/Qualifiers
FH
FT   source          1..28334
FT                   /organism="Arabidopsis thaliana"
FT                   /chromosome="1"
FT                   /cultivar="Columbia"
FT                   /mol_type="genomic DNA"
FT                   /clone="F16G16"
FT                   /db_xref="taxon:3702"
XX
SQ   Sequence 28334 BP; 9034 A; 5335 C; 5126 G; 8839 T; 0 other;
     aatcattttt tttattaatt acatagctca tttgtaaatg taaacacaat cgcgaacgca        60
     aacaaataat caaggtctaa agagaattta cctaatccaa acatgcttgt tttctgaata       120
     gagacacggt cgacttttga ccttttgata tttcttaaag ccagggtatc ttcctgcttt       180
     cctttaaact cgaggaattc ccctaacaca attttgctac ttttaagctc cactctgcag       240
     cataatcaac ttttaatgaa aactctttga acagaactac ttcatcaagg gttcaagaga       300
     tgagatgtat gaccaactca gacaattttt aaattgaacg tcaataaaga tgagatgtat       360
     gacctatccc acgttggtgc tgtgccaagg atgaaattaa gtttttgatg cgaagattcc       420
     tgcatctctt ttccggtgct gaatgcgcct ttaacccaac cagaaatttt acttttgat        480
     actggtttta gctgttgcca atcttcatag gtaattacct cactcgggaa tgcatgaatt       540
     gtactgaaaa aaaacaaaaa gaaaacatat atttatgcac cagtaggaaa ataaagaatg       600
     tctttacaga ccaagatagc cacgcccatg cacgcactac aagatgtcct gtacttcaaa       660
     ttgattgctt ttttgtctac ctcttgcaag caggaagata ctgacggtag caaggataac       720
     gataacatac cgaaaacttg gattgggaat gacaactttg caagacacat ttccttctag       780
     taaagtaact cccctctgtt caagatatga ttcaagaact gaagcttgtt tcttgacctc       840
     cgccagctga gaaaacacac atccattagt cataaacaag attgcataga tgcaagatgc       900
     aggtcaactt tcctaagtcg ataaaaatac aggagatttg atcaagagac ttgaaatgaa       960
     acgtgtaaga acgatccagg agaaaagaaa ggataaccaa gcacattcgt ataaataaca      1020
     agaatattct gtttcataat ctactcaagc gacaatcata ctctctactt tgccaaaaca      1080
     tatacacatt tacaattcga tggagacctt acataaaatg aacatatcca attcactgtg      1140
     gtcgtgcaaa gagaaaaacc ctataagcct ctagaagtaa atcatcagtg ctattttctc      1200
     acaatgacat aattatgcaa caaggagata gcagcagagt tccaatatat gtatccttta      1260
     caatctagaa cagaaaatta tagagaatca ctcacaagca ctaaaacgac actctaataa      1320
     cttggatgtt caaactataa gcattgtgtg taatcacaaa gaatcaaaca aaacttacag      1380
     gatcagggaa agtctcagta gtatgatgct tgccaccttc acaaacccag ctaccatcac      1440
     tagttaccgt aacaatcccc gatagattct tcacagaaat cactacaaca tctcctttag      1500
     taacaagaac aacatcaatg tcttgtctag aagcagtgtc ggcatctgga attcgaagtc      1560
     ctacataagc ttttccacca taaagtttct caagtctgca ataacaaatg gaaatcagaa      1620
     tcaatttcag tggaaatcga gaatcgaaaa atggagtatg gggaaaatcg ggaacctgtg      1680
     agcaacggag aagagagcag tggaatcgga agtttcgacg tcggagaatt cgtcgtcgta      1740
     gaagaaacgg cggacgagtt tgtagatgac gagaccgcag atgatttcaa tccacatcgt      1800
```

Example output for question 5

```
aaagccagggtatcttcctgcttt: 16
aaagccagggtatcttcctgctttccttt: 21
aaagaatgtcttt: 5
aaagtatgcaaccctagactctgattactcttt: 25
aaaggttgcttcttt: 7
aaagaaagaacttt: 6
aaagaaagaacttttcctctgttcttt: 19
aaagaacttt: 2
aaagaacttttcctctgttcttt: 15
aaaggaaaaatttgcattaaaaccctttt: 20
aaagaacattataaaaatccaaaacttt: 20
aaagccttaagcttt: 7
aaagcccaactatttcttt: 11
aaagttcaatcataacaagatttcatgaacttt: 25
aaagacatggattgaggaagacccatctcttt: 24
aaagccccttccattgtcagcttcgtttccttt: 25
aaagatgtgaacttt: 7
aaagatgtaaacttcatgatgtaaacttt: 21
aaagatgtaaacttt: 7
aaagatgtgaacttt: 7
aaagttttgtgcttt: 7
aaaggattttgattactatatctgcttt: 20
aaaggattttgattactatatctgctttacttt: 25
aaagaactacattcaacaaagcttagcttt: 22
aaagcttagcttt: 5
aaagcttagctttgaggtccttt: 15
aaagatttgtacctgtttggcaacatattcttt: 25
aaagccttt: 1
aaagcctttgaacttgatgacttcttt: 19
aaagagtattaaagcttt: 10
aaagagtattaaagctttcattgtcacaacttt: 25
aaagctttcattgtcacaacttt: 15
aaagagataaaacaaaaacgaaccttt: 19
aaagaagaataaatacttt: 11
aaagtcaaatgaattagatgcagtatccttt: 23
aaagtaaaaacacaagtcttgagtgctccttt: 24
aaagctgcagtcttagttgcctgaagcttt: 22
aaagagtgagccttt: 7
aaaggttgaaagacttt: 9
aaagacttt: 1
aaagcggtcttt: 4
aaagtcatgaccttt: 7
aaagagagtcgagcaattgggtgcccctccttt: 25
aaagaatccttcttt: 7
aaagaaaaatgtgttagagagttgacttt: 21
aaagaacagacccttt: 8
aaagcaaagagcattacttt: 12
aaagagcattacttt: 7
aaagttctctcagaggtttcttt: 15
aaagatatcctgcaaaatgcttt: 15
Count 1: 2
Count 2: 1
Count 3: 0
Count 4: 1
Count 5: 2
Count 6: 1
Count 7: 10
Count 8: 1
Count 9: 1
Count 10: 1
Count 11: 2
Count 12: 1
Count 13: 0
Count 14: 0
Count 15: 5
Count 16: 1
Count 17: 0
Count 18: 0
Count 19: 3
Count 20: 3
Count 21: 3
Count 22: 2
Count 23: 1
Count 24: 2
Count 25: 7
```

Perl 5 Reference Guide (Extract)

Literals

Numeric:
- **123**
- **1_234**
- **123.4**
- **5E-10**
- **0xff** (hex)
- **0377** (octal)

String:
- **'abc'**
  literal string, no variable interpolation or escape characters, except **\'** and ****. Also: **q/abc/**. Almost any pair of delimiters can be used instead of **/.../**.
- **"abc"**
  Variables are interpolated and escape sequences are processed. Also: **qq/abc/**.
  Escape sequences: **\t** (Tab), **\n** (Newline), **\r** (Return), **\f** (Formfeed), **\b** (Backspace), **\a** (Alarm), **\e** (Escape), **\033** (octal), **\x1b** (hex), **\c[** (control)
  \l and **\u** lowercase/uppercase the following character. **\L** and **\U** lowercase/uppercase until a **\E** is encountered. **\Q** quote regular expression characters until a **\E** is encountered.
- **'**COMMAND**'**
  evaluates to the output of the COMMAND. Also: **qx/**COMMAND**/**.

Array:
- **(1, 2, 3)**. **()** is an empty array.
- **(1..4)** is the same as **(1,2,3,4)**,
  likewise **('a'..'z')**.
- **qw/foo bar .../** is the same as **('foo','bar',...)**.

Array reference:
- **[1,2,3]**

Hash (associative array):
- **(**KEY1**,** VAL1**,** KEY2**,** VAL2**,...)**
  Also **(**KEY1**=>** VAL1**,** KEY2**=>** VAL2**,...)**

Hash reference:
- **{**KEY1**,** VAL1**,** KEY2**,** VAL2**,...}**

Code reference:
- **sub {**STATEMENTS**}**

Filehandles:
- **<STDIN>**, **<STDOUT>**, **<STDERR>**, **<ARGV>**, **<DATA>**.
  User-specified: HANDLE, **$**VAR.

Globs:
- **<**PATTERN**>** evaluates to all filenames according to the pattern. Use **<${**VAR**}>** or **glob $**VAR to glob from a variable.

Here-Is:
- **<<**IDENTIFIER
  Shell-style "here document."

Special tokens:
- **__FILE__**: filename; **__LINE__**: line number; **__END__**: end of program; remaining lines can be read using the filehandle **DATA**.

Variables

$var	a simple scalar variable.
$var[28]	29th element of array **@var**.
$p = \@var	now **$p** is a reference to array **@var**.
$$p[28]	29th element of array referenced by **$p**. Also, **$p->[28]**.
$var[-1]	last element of array **@var**.
$var[$i][$j]	**$j**th element of the **$i**th element of array **@var**.
$var{'Feb'}	one value from hash (associative array) **%var**.
$p = \%var	now **$p** is a reference to hash **%var**.

$$p{'Feb'}	a value from hash referenced by **$p**. Also, **$p->{'Feb'}**.
$#var	last index of array **@var**.
@var	the entire array; in a scalar context, the number of elements in the array.
@var[3,4,5]	a slice of array **@var**.
@var{'a','b'}	a slice of **%var**; same as **($var{'a'},$var{'b'})**.
%var	the entire hash; in a scalar context, **true** if the hash has elements.
$var{'a',1,...}	emulates a multidimensional array.
('a'...'z')[4,7,9]	a slice of an array literal.
PKG**::**VAR	a variable from a package, e.g., **$pkg::var**, **@pkg::ary**.
\OBJECT	reference to an object, e.g., **\$var**, **\%hash**.
***NAME**	refers to all objects represented by NAME. ***n1 = *n2** makes **n1** an alias for **n2**. ***n1 = $n2** makes **$n1** an alias for **$n2**.

You can always use a **{**BLOCK**}** returning the right type of reference instead of the variable identifier, e.g., **${...}**, **&{...}**. **$$p** is just a shorthand for **${$p}**.

Operators

******	Exponentiation
+ - * /	Addition, subtraction, multiplication, division
%	Modulo division
& | ^	Bitwise AND, bitwise OR, bitwise exclusive OR
>> <<	Bitwise shift right, bitwise shift left
|| &&	Logical OR, logical AND
.	Concatenation of two strings
x	Returns a string or array consisting of the left operand (an array or a string) repeated the number of times specified by the right operand

All of the above operators also have an assignment operator, e.g., **.=**

->	Dereference operator
****	Reference (unary)
! ~	Negation (unary), bitwise complement (unary)
++ --	Auto-increment (magical on strings), auto-decrement
== !=	Numeric equality, inequality
eq ne	String equality, inequality
< >	Numeric less than, greater than
lt gt	String less than, greater than
<= >=	Numeric less (greater) than or equal to
le ge	String less (greater) than or equal to
<=> cmp	Numeric (string) compare. Returns -1, 0, 1.
=~ !~	Search pattern, substitution, or translation (negated)
..	Range (scalar context) or enumeration (array context)
?:	Alternation (if-then-else) operator
,	Comma operator, also list element separator. You can also use **=>**.
not	Low-precedence negation
and	Low-precedence AND
or xor	Low-precedence OR, exclusive OR

All Perl functions can be used as list operators, in which case they have very high or very low precedence, depending on whether you look at the left or the right side of the operator. Only the operators **not**, **and**, **or** and **xor** have lower precedence.

A "list" is a list of expressions, variables, or lists. An array variable or an array slice may always be used instead of a list.

Parentheses can be added around the parameter lists to avoid precedence problems.

Statements

Every statement is an expression, optionally followed by a modifier, and terminated by a semicolon. The

semicolon may be omitted if the statement is the final one in a BLOCK.

Execution of expressions can depend on other expressions using one of the modifiers **if**, **unless**, **while** or **until**, for example:

```
EXPR1 if EXPR2 ;
EXPR1 until EXPR2 ;
```

The logical operators **||**, **&&** or **?:** also allow conditional execution:

```
EXPR1 || EXPR2 ;
EXPR1 ? EXPR2 : EXPR3 ;
```

Statements can be combined to form a BLOCK when enclosed in **{}**. Blocks may be used to control flow:

```
if (EXPR) BLOCK [ [ elsif (EXPR) BLOCK ... ] else BLOCK ]
unless (EXPR) BLOCK [ else BLOCK ]
[ LABEL: ] while (EXPR) BLOCK [ continue BLOCK ]
[ LABEL: ] until (EXPR) BLOCK [ continue BLOCK ]
[ LABEL: ] for (EXPR; EXPR; EXPR) BLOCK
[ LABEL: ] foreach VAR² (LIST) BLOCK
[ LABEL: ] BLOCK [ continue BLOCK ]
```

Program flow can be controlled with:

goto LABEL	Continue execution at the specified label.
last [LABEL]	Immediately exits the loop in question. Skips continue block.
next [LABEL]	Starts the next iteration of the loop.
redo [LABEL]	Restarts the loop block without evaluating the conditional again.

Special forms are:

```
do BLOCK while EXPR ;
do BLOCK until EXPR ;
```

which are guaranteed to perform BLOCK once before testing EXPR, and

```
do BLOCK
```

which effectively turns BLOCK into an expression.

Structure Conversion

pack TEMPLATE, LIST
: Packs the values into a binary structure using TEMPLATE.

unpack TEMPLATE, EXPR
: Unpacks the structure EXPR into an array, using TEMPLATE.

TEMPLATE is a sequence of characters as follows:

a / **A**	ASCII string, null- / space-padded
b / **B**	Bit string in ascending / descending order
c / **C**	Native / unsigned char value
f / **d**	Single / double float in native format
h / **H**	Hex string, low / high nybble first
i / **I**	Signed / unsigned integer value
l / **L**	Signed / unsigned long value
n / **N**	Short / long in network (big endian) byte order
s / **S**	Signed / unsigned short value
u / **P**	Uuencoded string / pointer to a string
v / **V**	Short / long in VAX (little endian) byte order
x / **@**	Null byte / null fill until position
X	Backup a byte

Each character may be followed by a decimal number that will be used as a repeat count; an asterisk (*****) specifies

all remaining arguments. If the format is preceded with **%**N, **unpack** returns an N-bit checksum instead. Spaces may be included in the template for readability purposes.

String Functions

chomp LIST²
: Removes line endings from all elements of the list; returns the (total) number of characters removed.

chop LIST²
: Chops off the last character on all elements of the list; returns the last chopped character.

crypt PLAINTEXT, SALT
: Encrypts a string.

eval EXPR²
: EXPR is parsed and executed as if it were a Perl program. The value returned is the value of the last expression evaluated. If there is a syntax error or runtime error, an undefined string is returned by **eval**, and **$@** is set to the error message. See also **eval** in section Miscellaneous.

index STR, SUBSTR [, OFFSET]
: Returns the position of SUBSTR in STR at or after OFFSET. If the substring is not found, returns **-1** (but see **$[** in section Special Variables).

length EXPR²
: Returns the length in characters of the value of EXPR.

lc EXPR
: Returns a lowercase version of EXPR.

lcfirst EXPR
: Returns EXPR with the first character lowercase.

quotemeta EXPR
: Returns EXPR with all regular expression metacharacters quoted.

rindex STR, SUBSTR [, OFFSET]
: Returns the position of the last SUBSTR in STR at or before OFFSET.

substr EXPR, OFFSET [, LEN]
: Extracts a substring of length LEN out of EXPR and returns it. If OFFSET is negative, counts from the end of the string. May be assigned to.

uc EXPR
: Returns an uppercased version of EXPR.

ucfirst EXPR
: Returns EXPR with the first character uppercased.

Array and List Functions

delete $HASH**{**KEY**}**
: Deletes the specified value from the specified hash. Returns the deleted value unless HASH is **tie**d to a package that does not support this.

each %HASH
: Returns a 2-element array consisting of the key and value for the next value of the hash. Entries are returned in an apparently random order. After all values of the hash have been returned, a null array is returned. The next call to **each** after that will start iterating again.

exists EXPR²
: Checks if the specified hash key exists in its hash array.

grep EXPR, LIST

grep BLOCK LIST
: Evaluates EXPR or BLOCK for each element of the LIST, locally setting **$_** to refer to the element. Modifying **$_** will modify the corresponding element from LIST. Returns the array of elements from LIST for which EXPR returned **true**.

join EXPR, LIST
: Joins the separate strings of LIST into a single string with fields separated by the value of EXPR, and returns the string.

keys %HASH
: Returns an array with all of the keys of the named hash.

map EXPR, LIST

map BLOCK LIST
: Evaluates EXPR or BLOCK for each element of the LIST, locally setting **$_** to refer to the element. Modifying **$_** will modify the corresponding element from LIST. Returns the list of results.

pop @ARRAY
: Pops off and returns the last value of the array.

push @ARRAY, LIST

Pushes the values of LIST onto the end of the array.

reverse LIST
: In array context, returns the LIST in reverse order. In scalar context: returns the first element of LIST with bytes reversed.

scalar @ARRAY
: Returns the number of elements in the array.

scalar %HASH
: Returns a **true** value if the hash has elements defined.

shift [@ARRAY]
: Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If @ARRAY is omitted, shifts **@ARGV** in main and **@_** in subroutines.

sort [SUBROUTINE] LIST
: Sorts the LIST and returns the sorted array value. SUBROUTINE, if specified, must return less than zero, zero, or greater than zero, depending on how the elements of the array (available to the routine as **$a** and **$b**) are to be ordered. SUBROUTINE may be the name of a user-defined routine, or a BLOCK.

splice @ARRAY, OFFSET [, LENGTH [, LIST]]
: Removes the elements of @ARRAY designated by OFFSET and LENGTH, and replaces them with LIST (if specified). Returns the elements removed.

split [PATTERN [, EXPR[2] [, LIMIT]]]
: Splits a string into an array of strings, and returns it. If LIMIT is specified, splits into at most that number of fields. If PATTERN is also omitted, splits at the whitespace. If not in array context, returns number of fields and splits to @_. See also Search and Replace Functions.

unshift @ARRAY, LIST
: Prepends list to the front of the array, and returns the number of elements in the new array.

values %HASH
: Returns a normal array consisting of all the values of the named hash.

Regular Expressions

Each character matches itself, unless it is one of the special characters **+ ? . * ^ $ ( ) [ ] { } |** \. The special meaning of these characters can be escaped using a \.

.	matches an arbitrary character, but not a newline unless it is a single-line match (see **m//s**).
(...**)**	groups a series of pattern elements to a single element.
^	matches the beginning of the target. In multiline mode (see **m//m**) also matches after every newline character.
$	matches the end of the line. In multiline mode also matches before every newline character.
[... **]**	denotes a class of characters to match. **[^** ... **]** negates the class.
(... **|** ... **|** ... **)**	matches one of the alternatives.
(?# TEXT **)**	Comment.
(?: REGEXP **)**	Like **(**REGEXP**)** but does not make back-references.
(?= REGEXP **)**	Zero width positive look-ahead assertion.
(?! REGEXP **)**	Zero width negative look-ahead assertion.
(? MODIFIER **)**	Embedded pattern-match modifier. MODIFIER can be one or more of **i**, **m**, **s**, or **x**.

Quantified subpatterns match as many times as possible. When followed with a **?** they match the minimum number of times. These are the quantifiers:

+	matches the preceding pattern element one or more times.
?	matches zero or one times.
*****	matches zero or more times.
{N**,**M**}**	denotes the minimum N and maximum M match count. **{**N**}** means exactly N times; **{**N**,}** means at least N times.

A \ escapes any special meaning of the following character if non-alphanumeric, but it turns most alphanumeric characters into something special:

\w	matches alphanumeric, including _, **\W** matches non-alphanumeric.
\s	matches whitespace, **\S** matches non-whitespace.
\d	matches numeric, **\D** matches non-numeric.
\A	matches the beginning of the string, **\Z** matches the end.
\b	matches word boundaries, **\B** matches non-boundaries.
\G	matches where the previous **m//g** search left off.
\n, **\r**, **\f**, **\t**	etc. have their usual meaning.

\w, **\s** and **\d** may be used within character classes, **\b** denotes backspace in this context.

Back-references:

\1 ... **\9**	refer to matched subexpressions, grouped with **()**, inside the match.
\10	and up can also be used if the pattern matches that many subexpressions.

See also **$1** ... **$9**, **$+**, **$&**, **$`**, and **$'** in section Special Variables.

With modifier **x**, whitespace can be used in the patterns for readability purposes.

Search and Replace Functions

[EXPR **=~**] [**m**] **/**PATTERN**/** [**g**] [**i**] [**m**] [**o**] [**s**] [**x**]
: Searches EXPR (default: **$_**) for a pattern. If you prepend an **m** you can use almost any pair of delimiters instead of the slashes. If used in array context, an array is returned consisting of the subexpressions matched by the parentheses in the pattern, i.e., **($1,$2,$3,...)**.
Optional modifiers: **g** matches as many times as possible; **i** searches in a case-insensitive manner; **o** interpolates variables only once. **m** treats the string as multiple lines; **s** treats the string as a single line; **x** allows for regular expression extensions.
If PATTERN is empty, the most recent pattern from a previous match or replacement is used.
With **g** the match can be used as an iterator in scalar context.

?PATTERN**?**
: This is just like the **/**PATTERN**/** search, except that it matches only once between calls to the **reset** operator.

[**$**VAR **=~**] **s/**PATTERN**/**REPLACEMENT**/** [**e**] [**g**] [**i**] [**m**] [**o**] [**s**] [**x**]
: Searches a string for a pattern, and if found, replaces that pattern with the replacement text. It returns the number of substitutions made, if any; if no substitutions are made, it returns **false**.
Optional modifiers: **g** replaces all occurrences of the pattern; **e** evaluates the replacement string as a Perl expression; for any other modifiers, see **/**PATTERN**/** matching. Almost any delimiter may replace the slashes; if single quotes are used, no interpretation is done on the strings between the delimiters, otherwise the strings are interpolated as if inside double quotes.
If bracketing delimiters are used, PATTERN and REPLACEMENT may have their own delimiters, e.g., **s(foo)[bar]**. If PATTERN is empty, the most recent pattern from a previous match or replacement is used.

[**$**VAR **=~**] **tr/**SEARCHLIST**/**REPLACEMENTLIST**/** [**c**] [**d**] [**s**]
: Translates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced. **y** may be used instead of **tr**.
Optional modifiers: **c** complements the SEARCHLIST; **d** deletes all characters found in SEARCHLIST that do not have a corresponding character in REPLACEMENTLIST; **s** squeezes all sequences of characters that are translated into the same target character into one occurrence of this character.

pos SCALAR
: Returns the position where the last **m//g** search left off for SCALAR. May be assigned to.

study [**$**VAR[2]]
: Study the scalar variable **$**VAR in anticipation of performing many pattern matches on its contents before the variable is next modified.

Input / Output

In input/output operations, FILEHANDLE may be a filehandle as opened by the **open** operator, a predefined filehandle (e.g., **STDOUT**) or a scalar variable that evaluates to the name of a filehandle to be used.

<FILEHANDLE**>**
: In scalar context, reads a single line from the file opened on FILEHANDLE. In array context, reads the whole file.

< >
: Reads from the input stream formed by the files specified in **@ARGV**, or standard input if no arguments were supplied.

close FILEHANDLE
: Closes the file or pipe associated with the filehandle.

open FILEHANDLE [, FILENAME]
: Opens a file and associates it with FILEHANDLE. If FILENAME is omitted, the scalar variable of the same name as the FILEHANDLE must contain the filename.
The following filename conventions apply when opening a file.

"FILE" open FILE for input. Also "<FILE".
">FILE" open FILE for output, creating it if necessary.
">>FILE" open FILE in append mode.
"+<FILE" open FILE with read/write access (file must exist).
"+>FILE" open FILE with read/write access (file truncated).
"|CMD" opens a pipe to command CMD; forks if CMD is -.
"CMD|" opens a pipe from command CMD; forks if CMD is -.
FILE may be &FILEHND in which case the new filehandle is connected to the (previously opened) filehandle FILEHND. If it is &=N, FILE will be connected to the given file descriptor. **open** returns **undef** upon failure, **true** otherwise.
Returns a pair of connected pipes.

print [FILEHANDLE] [LIST[2]]
Equivalent to **print** FILEHANDLE **sprintf** LIST.

Special Variables

The following variables are global and should be localized in subroutines:

$_	The default input and pattern-searching space.
$.	The current input line number of the last filehandle that was read.
$/	The input record separator, newline by default. May be multicharacter.
$,	The output field separator for the print operator.
$"	The separator that joins elements of arrays interpolated in strings.
$\	The output record separator for the print operator.
$#	The output format for printed numbers. Deprecated.
$*	Set to 1 to do multiline matching within strings. Deprecated, see the **m** and **s** modifiers in section Search and Replace Functions.
$?	The status returned by the last `...` command, pipe **close** or **system** operator.
$]	The perl version number, e.g., 5.001.
$[	The index of the first element in an array, and of the first character in a substring. Default is 0. Deprecated.
$;	The subscript separator for multidimensional array emulation. Default is "\034".
$!	If used in a numeric context, yields the current value of **errno**. If used in a string context, yields the corresponding error string.
$@	The Perl error message from the last **eval** or **do** EXPR command.
$:	The set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format.
$0	The name of the file containing the Perl script being executed. May be assigned to.
$$	The process ID of the currently executing Perl program. Altered (in the child process) by **fork**.
$<	The real user ID of this process.
$>	The effective user ID of this process.
$(	The real group ID of this process.
$)	The effective group ID of this process.
$^A	The accumulator for **formline** and **write** operations.
$^D	The debug flags as passed to perl using -D.
$^F	The highest system file descriptor, ordinarily 2.
$^I	In-place edit extension as passed to Perl using -i.
$^L	Formfeed character used in formats.
$^P	Internal debugging flag.
$^T	The time (as delivered by **time**) when the program started. This value is used by the file test operators -M, -A and -C.
$^W	The value of the -w option as passed to Perl.
$^X	The name by which the currently executing program was invoked.

The following variables are context dependent and need not be localized:

$%	The current page number of the currently selected output channel.
$=	The page length of the current output channel. Default is 60 lines.
$-	The number of lines remaining on the page.
$~	The name of the current report format.
$^	The name of the current top-of-page format.
$|	If set to nonzero, forces a flush after every write or print on the currently selected output channel. Default is 0.
$ARGV	The name of the current file when reading from <>.

The following variables are always local to the current block:

$&	The string matched by the last successful pattern match.
$`	The string preceding what was matched by the last successful match.
$'	The string following what was matched by the last successful match.
$+	The last bracket matched by the last search pattern.
$1...$9...	Contain the subpatterns from the corresponding sets of parentheses in the last pattern successfully matched. $10... and up are only available if the match contained that many subpatterns.

Special Arrays

@ARGV	Contains the command-line arguments for the script (not including the command name).
@EXPORT	Names the methods a package exports by default.
@EXPORT_OK	Names the methods a package can export upon explicit request.
@INC	Contains the list of places to look for Perl scripts to be evaluated by the **do** FILENAME and **require** commands.
@ISA	List of base classes of a package.
@_	Parameter array for subroutines. Also used by **split** if not in array context.
%ENV	Contains the current environment.
%INC	List of files that have been included with **require** or **do**.
%OVERLOAD	Can be used to overload operators in a package.
%SIG	Used to set signal handlers for various signals.