Institute of Bioinformatics
Johannes Kepler University Linz

Perl

A Short Introduction for Bioinformaticians

What is Perl?

- Perl is an interpreted (scripting) language
- Perl is (almost) platform-independent
- Perl is free of charge
- Perl is a common standard in bioinformatics, language processing, and Web programming

Perl's Advantages

- Platform-independent
- Free of charge
- Only minor hardware and software requirements
- Powerful elements for string processing (regular expressions) and hash tables allow concise algorithms
- Quick and easy solutions are possible

Perl's Disadvantages

- Richness and compactness of language facilitates difficult-to-read programs
- No stand-alone programs without additional software
- Slower than compiled code
- Perl is sometimes wasteful with memory
- Perl's built-in strings, lists, and hash tables sometimes hide potential performance problems
- Therefore, Perl cannot handle as large problems as some other programming languages

Use Perl for...

- Small rapid prototyping solutions
- Applications that require nifty string processing
- Small to medium-sized problems
- Applications with moderate performance and memory requirements

Do Not Use Perl for...

- Large software projects (performance, stability, quality, maintainability)
- Applications in which performance and memory consumption are most important factors

Why Perl in Bioinformatics

- String processing capabilities and hash tables
- Easy, also for biologists and other people outside computer science

What Else is Perl Used For?

- Very common in Web programming (offers very good database and networking integration)
- Perl can also serve as a more powerful replacement of UNIX shell scripts or DOS/Windows batch files
- *In this introduction, we concentrate on standard elements that will be necessary for bioinformatics applications*

Setting Up Your Perl System

- What you need: (1) Perl software, (2) text editor
- On UNIX/Linux systems, probably everything is pre-installed
- Windows/MacOS
  - Get ActivePerl and install it:
    `http://www.activestate.com/Products/ActivePerl/`
  - Choose your favorite text editor (e.g. UltraEdit, TextPad, XEmacs)
  - Make sure that perl is in your default search path

Documentation

- Online:

  `http://www.perl.org/docs.html`

  `http://perldoc.perl.org/perlintro.pdf`

  `http://perldoc.perl.org/index-tutorials.html`

  `http://perldoc.perl.org/index-functions.html`

- Program `perldoc` that is part of every Perl system

First Steps

- Comments start with `#`
- For compatibility with UNIX/Linux systems, it is advisable to start the program with the so-called She'Bang line:

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

- For better checking, you are advised to add the following two lines to your program (right after the She'Bang line):

```
use strict;
use warnings;
```

- Statements need to be closed with a semicolon, whitespaces are irrelevant
- In the simplest case, terminal output is issued with the `print` command

Data — Variables

- There are three basic types of variables in Perl. Unlike other programming languages, the type is identified with a special character that prefixes the variable name:

  Scalars: prefixed with "`$`" e.g. `$num`

  Arrays: prefixed with "`@`" e.g. `@list`

  Hashes: prefixed with '`%`" e.g. `%hashtable`

- If using `use strict;`, variables have to be declared before they are used. This is done with `my`.

- The most basic operator is the assignment operator `=`. It copies the content of one variable into the other (possibly with a conversion).

Scalars

- Among scalars, there is no specific kind of type checking!
- Scalars can have three different kinds of meanings:
  1. Numbers
  2. Strings
  3. References

Numbers

- There is no explicit distinction between integer and floating point numbers, this is handled implicitly
- Examples:

```
$num = 3;
$pi = 3.141592654;
$mio = 1.e6;
```

- Arithmetic operators: $+$, $-$, $*$, $/$, $\%$, may be used in conjunction with assignments, i.e. $+=$, $-=$, $*=$, $/=$, $\%=$
- Increment/decrement operators: $++$, $--$

String Constants

- Double quotes or single quotes may be used around strings:

```
'Hello world'
"Hello world"
```

- The difference is that strings with single quotes are taken literally. Strings with double quotes are interpreted, i.e. variable names and special characters are translated first
- Common operator: concatenation operator `.`, may also be used in conjunction with assignments `.=`

Arrays

- Arrays need not have a predefined size
- Assignment operators work on individual elements (scalars) or with whole lists, e.g.

```
@list = ("Zicke", "Zacke", 123);
```

- The index starts with 0
- Memory allocation is done on demand; Be aware: when you first create the fifth element, the whole list from the first to the fifth element is allocated (i.e. elements 0...4)
- Accessing single elements:

```
$elem = $list[2];
```

- Accessing multiple elements is also possible, e.g. `@list[0,1]`, `@list[1..3]`

Arrays (cont'd)

- Assignments also work in the following way:

```
($firstelem, @remaining) = @list;
```

- Special operators:
  - Assignment to scalar gives number of elements, e.g.

```
$no = @list;
```

  - Index of last element:

```
$index = $#list;
$elem = $list[$#list];
```

- Note that lists are always flattened, e.g.

```
@list = ("a", "b", ("c", "d"));
```

  is the same as

```
@list = ("a", "b", "c", "d");
```

Arrays: Stacks and Queues

- Special functions for adding/removing elements from arrays:
  - `push` appends (a) new element(s) at the end of the array, e.g.

```
push(@list, "tralala");
push(@list, ("hui", 1));
```

  - `pop` returns the last element and removes it from the list, e.g.

```
$elem = pop(@list);
```

  - `shift` returns the first element and removes it from the

```
$elem = shift(@list);
```

  - `unshift` inserts (a) new element(s) at the beginning of an array

```
unshift(@list, "tralala");
unshift(@list, ("hui", 1));
```

- "Killing" an array: `@list = ();`

Arrays: The `splice` Function

The `splice` function allows to remove a part of an array or to replace it by another list.

- Example with four arguments:

```
my @list = ("u", "v", "w", "x", "y", "z", "0", "1", "2");
splice(@list, 3, 4, ("a", "b"));
```

  removes the four elements from no. 3 on (i.e. elements [3..6]) and replaces them by two elements, "a" and "b". Finally, `@list` is (“u”, “v”, “w”, “a”, “b”, “1”, “2”).

- Example with three arguments:

```
my @list = ("u", "v", "w", "x", "y", "z", "0", "1", "2");
splice(@list, 3, 4);
```

  removes the four elements from no. 3 onwards. Finally, `@list` is (“u”, “v”, “w”, “1”, “2”).

Arrays: The `splice` Function (cont'd)

- Example with two arguments:

```
my @list = ("u", "v", "w", "x", "y", "z", "0", "1", "2");
splice(@list, 3);
```

  removes *all* elements from no. 3 onwards. Finally, `@list` is ("u", "v", "w").

- A negative offset $-i$ as second argument means to start from the i-th to the last element.

- See also
  `http://perldoc.perl.org/functions/splice.html`

Hashes

- Like arrays, hashes are collections of scalars, however, with the difference that they are not ordered; individual elements can be accessed by arbitrary scalars, so-called *keys*
- Assignment operators work on individual elements (scalars), e.g.,

```
$color{"apple"} = "red";
```

or with whole lists, e.g.

```
%color = ("apple", "red", "banana",
"yellow");
```

which is equivalent to the more readable

```
%color = (apple => "red", banana =>
"yellow");
```

Hashes (cont'd)

- Memory allocation is done on demand
- Special functions:
  - List of keys:

```
@keylist = keys %color;
```

  - List of values:

```
@vallist = values %color;
```

  - Deleting a hash entry:

```
delete $color{'apple'};
```

  - Checking whether a hash entry exists:

```
exists $color{'apple'}
```

Control Structures: if and unless

- Single if:

```
if (⟨expression⟩)
{
  ...
}
```

- unless (expression negated):

```
unless (⟨expression⟩)
{
  ...
}
```

Control Structures: if and unless (cont'd)

- Note that the curly braces are obligatory; conditional statements, however, can also be written in single lines:

```
...  if (⟨expression⟩);
...  unless (⟨expression⟩);
```

Control Structures: if/else

- if/else

```
if (⟨expression⟩)
{
   ...
}
else
{
   ...
}
```

Control Structures: if/elsif/else

- if/elsif/else:

```
if (⟨expression1⟩)
{
  ...
}
elsif (⟨expression2⟩)
{
  ...
}
else
{
  ...
}
```

Control Structures: while and until

- while:

```
while (⟨expression⟩)
{
  ...
}
```

- until (expression negated):

```
until (⟨expression⟩)
{
  ...
}
```

Control Structures: while and until (cont'd)

- Note that the curly braces are obligatory; while and until loops, however, can also be written in single lines:

```
... while (⟨expression⟩);
... until (⟨expression⟩);
```

Control Structures: for and foreach (1/3)

- for (the same as in C):

```
for (⟨init⟩; ⟨condition⟩; ⟨increment⟩)
{
  ...
}
```

- foreach:

```
foreach $variable (@list)
{
  ...
}
```

Control Structures: for and foreach (2/3)

- foreach with fixed list:

```
foreach $key ("tralala", 3, 2, 1, 0)
{
  ...
}
```

- Simple repetions:

```
foreach  (1..15)
{
  ...
}
```

Control Structures: for and foreach (3/3)

- Note that the curly braces are obligatory; foreach loops, however, can also be written in single lines:

```
... foreach $variable (@list));
```

- Not allowed for `for` loops!

Expressions

- Logical operators: `&&`, `||`, `!`
- Numeric comparison: `==`, `!=`, `<`, `>`, `<=`, `>=`
- String comparison: `eq`, `ne`, `lt`, `gt`, `le`, `ge`

Truth and falsehood:

- 0, ’0’ , ”, () and `undef` are interpreted as *false* if they are the result of the evaluation of an expression, all other values are interpreted as *true*

Special Variables

`$_`: default input; many (if not most) built-in Perl functions use this variable as default input if no argument is specified; also used as default variable in `foreach` loops

`@_`: list of input arguments in sub-routines (see later)

`@ARGV`: list of command line arguments; NOTE: unlike in `C`, `$ARGV[0]` is the first command line argument, not the program name

`$0`: the name of the program being executed (like `argv[0]` in `C`)

`$1`, `$2`, `...`: pattern matches (see later)

Note that there are a whole lot more special variables, but they are not important for us at this point.

Control Structures: sub-routines

- Subroutines are simply written as follows:

```
sub ⟨name⟩
{
   ...
}
```

- Recursions are allowed
- Note that there is no typechecking, not even the number of arguments needs to be fixed
- Arguments are passed through the special list `@_`
- Return values are optional and need to be passed with `return`

Control Structures: sub-routines (cont'd)

- Calls of sub-routines can be prefixed with `&`
- Arguments need to be separated by commas and can (but need not!) be embraced with parentheses; however, the use of parentheses is highly recommended
- Like in C, Perl uses *call by value*

Control Structures: sub-routines example

```
#!/usr/bin/perl -w
use strict;

sub divide
{
  my ($enumerator,$denominator) = @_;
  return $enumerator/$denominator;
}

...
my $quotient = &divide(4, 2);
...
```

Variable Scoping

- Declaring/using a variable in the main program outside any block creates a *global variable*
- Declarations inside a sub-routine create a *local variable*
- Declaring/using a variable in a block creates a *temporary variable*
- Use `use strict` to enforce declarations
- Try to use temporary/local variables where possible and avoid the use of global variables where possible
- Local variables are destroyed/de-allocated as soon as the execution of their scope (block/sub-routine) is finished (with the only exception if there are references to this local variable)

Contexts

Every operation that you invoke in a Perl script is evaluated in a specific *context*. How the operator behaves may be determined by the respective context. Note that the context is determined by how the return value is used. There are two major contexts in Perl, the scalar and the list context

- Scalar context:
  - Numerical context
  - String context
  - Boolean context
- List context

References

- A *reference* is a scalar that “points” to a scalar, list or hash.
- A reference is created by prefixing with a backslash “\”, e.g.

```
$ref_to_scalar = \$scalar;
$ref_to_array = \@array;
$ref_to_hash = \%hash;
```

- De-referencing, i.e. getting back the value, is done by prefixing the reference with the appropriate prefix, e.g.

```
$scalar = ${$ref_to_scalar};
@array = @{$ref_to_array};
%hash = %{$ref_to_hash};
```

- The curly braces may also be omitted when de-referencing.

References (cont'd)

- With references more complex data structures can be builded, e.g. multidimensional arrays and hashes.
- References are useful to realize *call by reference* in sub-routines. This allows for passing references to arrays and hashes to sub-routines.
- Since Perl5 references are used for objectoriented programming in order to access the methods (functions and sub-routines) and the properties (variables) of an object.
- Accessing items in de-referenced arrays and hashes may be clumsy, therefore `${$ref_to_array}[2]` and `${$ref_to_hash}{"key"}` may also be written as `$ref_to_array->[2]` and `$ref_to_hash->{"key"}`

Matrices (1/2)

- A matrix can be created by arrays which contain references to arrays, e.g.

```
@row1 = (1, 2, 3);
@row2 = (4, 5, 6);
@row3 = (7, 8, 9);
@array = (\@row1, \@row2, \@row3);
print "$array[0]"; # prints 1 2 3
print "$array[1]->[2]"; # prints 6
```

- This can be written more easily because perl automatically includes an arrow operator between tow subsequent brackets, e.g.

```
print "$array[1][2]"; # prints 6
```

Matrices (2/2)

- Brackets create a reference to an anonymous array; by this trick, matrices can be realized easily, e.g.

```
$array[0] = [1, 2, 3];
$array[1] = [4, 5, 6];
$array[2] = [7, 8, 9];
print "$array[1][2]"; # prints 6
```

  More easily:

```
@array = ([1, 2, 3], [4, 5, 6], [7, 8, 9]);
```

- This also works without a one-for-all assignment, only by assigning values to individual elements, e.g.

```
$array[1][2] = 7;
```

Hashes of Hashes

- Curly braces create a reference to an anonymous hash; by this trick, hashes of hashes can be realized relatively easily, e.g.

```
%hash = (j => {a=>1, b=>2, c=>3},
         k => {d=>6, e=>5, f=>10});
```

- This also works without a one-for-all assignment, only by assigning values to individual elements, e.g.

```
$hash{j}{a} = 1;
$hash{k}{e} = 5;
```

- Note that, in such a case, `$hash{j}` is not a hash, but a reference; therefore, something like `keys $hash{j}` will not work. To get the keys of the entries in the second level, one has to use something like

```
keys %{$hash{j}}
```

A Note on the Lifetime of Local Variables

- Usually, local variables are destroyed/de-allocated as soon as the execution of their scope (block/sub-routine) is finished
- This means that references to local variables would point to nirvana after their scope's execution is finished
- That is not how it works; instead, Perl uses a reference counter for each variable. A local variable remains existing until there is no reference pointing to it anymore

A Note on the Lifetime of Local Variables (cont'd)

Consider the following example:

```
my @matrix;
for(my $i = 0; $i < 10; $i++)
{
   my @line = (0..9);
   $matrix[$i] = \@line;
}
```

The full 10×10 matrix can be used safely, even if the array's lines are lists created as local variables.

File/Console IO (1/5)

- Similar to other programming languages, Perl uses *file handles*
- The `open` function is used to open a file handle:

```
open(INPUT, "< $filename1"); # open for reading
open(OUTPUT, "> $filename2"); # write new content
open(LOGFILE, ">> $filename2"); # append
```

- `open` returns 0 on failure and a non-zero value on success
- The following file handles are open and usable by default: `STDIN`, `STDOUT`, `STDERR`

File/Console IO (2/5)

- File handles are closed with `close`, e.g.

```
close(INPUT);
close(OUTPUT);
close(LOGFILE);
```

- `close` returns 0 on failure and a non-zero value on success
- After writing to a file it is advisable to check whether the writing succeeded or not

File/Console IO (3/5)

- The `print` function can be used to write to a file, e.g.

```
print OUTPUT "Tralala";
```

  Note the missing comma after the file handle!

- In scalar context, the input operator `<>` reads one line from the specified file handle, e.g.

```
$line = <INPUT>;
```

  reads one line from the file handle `INPUT`, whereas

```
$line = <STDIN>;
```

  reads one line from the console input.

- Note that, in the above examples, `$line` still contains the trailing newline character; it can be removed with the `chomp` function, e.g.

```
chomp $line;
```

File/Console IO (4/5)

- In list context, the input operator `<>` reads the whole file at once, e.g.

  ```
  @whole = <INPUT>;
  ```

  reads the whole file `INPUT` line by line, where each line is one item in the list `@whole` is one line
- All lines in `@whole` still contain the the trailing newline characters; they can be removed with the `chomp` function, e.g.

  ```
  chomp @whole;
  ```

  removes the trailing newline character from all lines in `@whole`
- This way of reading a file may be very comfortable. Note, however, that it requires reading the whole file into the memory at once, which may be infeasible for larger files

File/Console IO (5/5)

- If the input operator <> is used without specifying a variable, the next line is placed in the default variable `$_`
- Of course, `$_` still contains the trailing newline character; it can be removed with the `chomp` function, this time without any arguments (because `chomp` takes `$_` by default anyway)
- Example of a program fragment that reads input from the console input line by line:

```
while (<STDIN>)
{
  ...  # $_ contains last input line with newline
  chomp;
  ...  # $_ contains last input line without newline
}
```

Regular Expressions: Basics (1/3)

- Regular expressions are a powerful tool for string pattern matching and replacement
- Simple regular expressions are embraced with slashes `/.../`
- Regular expression are most often used in conjunction with the two operators `=~` and `!~`; the former checks whether a pattern is found and the latter checks whether the pattern is not present, e.g.

```
('Hello world' =~ /Hell/) # evaluates to true
('Hello world' !~ /Hell/) # evaluates to false
```

Regular Expressions: Basics (2/3)

- The examples on the previous slide show how to search for certain constant string parts (“Hell” in these examples)
- The search pattern can also (partly) be a variable
- Search patterns, however, need not be string constants; there is a host of meta-characters for constructing more advanced searches: `{ } [ ] ( ) ^ $ . | * + ?`
- Using meta-characters as ordinary search expressions requires prefixing them with a backslash

Regular Expressions: Basics (3/3)

- A regular expression can also be written synonymously as the search operator `m//`, e.g. `/world/` and `m/world/` mean the same thing. The `m//` operator has the advantage that other characters for embracing the regular expression can also be used (instead of only slashes), e.g. `m!!`, `m{}`

Regular Expressions: Character Classes

- Character classes can be defined between brackets; as a simple example, `/[bcr]at/` matches “bat”, “cat” and “rat”
- Character classes may also be ranges in the present character coding (e.g. ASCII), e.g. `/index[0-2]/` matches “index0”, “index1” and “index2”; `/[0-9a-fA-F]/` matches a hexadecimal digit
- Using the meta-character `^` in the first place of a character class means negation, e.g. `/[^0-9]/` matches all non-numerical characters

Regular Expressions: Predefined Character Classes

- `\d`: numerical character, short for `[0-9]`
- `\D`: non-numerical character, short for `[^0-9]`, equivalently `[^\d]`
- `\s`: whitespace character, short for `[ \t\r\n\f]`
- `\S`: non-whitespace character, short for `[^ \t\r\n\f]`, equivalently `[^\s]`
- `\w`: word character (alphanumeric or _), short for `[0-9a-zA-Z_]`
- `\W`: non-word character, short for `[^0-9a-zA-Z_]`, equivalently `[^\w]`
- The period `.` matches every character except the newline character `\n`
- `\b`: matches a boundary between a word and a non-word character or between a non-word and word character; this is useful for checking whether a match occurs at the beginning or end of a word

Regular Expressions: Matching Beginnings and Ends

- The `^` meta-character is used to require that the string begins with the search pattern, e.g.

```
('Hello world' =~ /^Hell/) # matches
('Hello world' =~ /^world/) # does not match
```

- The `$` meta-character is used to require that the string ends with the search pattern, e.g.

```
('Hello world' =~ /Hell$/) # does not match
('Hello world' =~ /rld$/) # matches
```

- The two meta-characters `^` and `$` can also be used together

Regular Expressions: Variants and Grouping

- The stroke character `|` is used as the so-called *alternation meta-character*, e.g. `/dog|cat|rat/` matches “dog”, “cat” and “rat”, and so does `/dog|[cr]at/`
- Parentheses that serve as so-called *grouping meta-characters* can be used for grouping alternatives in parts of the search pattern, e.g. `/house(dog|cat)/` matches “housedog” and “housecat”
- Alternatives may also be empty, e.g. `/house(cat|)/` matches “housecat” and “house”
- Groupings may also be nested, e.g. `/house(cat(s|)|)/` matches “housecats”, “housecat” and “house”

Regular Expressions: Quantifiers

- So far, we were only able to search only for patterns with a relatively fixed structure, but we were not able to deal with repetitions in a flexible way; that is what *quantifiers* are good for
- The following quantifiers are available:

  `?` match 1 or 0 times

  `*` match any number of times

  `+` match at least once

  `{n,m}` match at least n and at most m times

  `{n,}` match at least n times

  `{n}` match exactly n times

Regular Expressions: Quantifiers (cont'd)

Examples:

- `/0x[0-9a-fA-F]+/` matches hexadecimal numbers
- `/[\-\+]?\d+/` matches integer numbers
- `/[\-\+]?\d+\.\d+/` matches numbers
- `/\d{2}\.\d{2}\.\d{4}/` matches dates in DD.MM.YYYY format

Regular Expressions: Extracting Matches (1/3)

- Parentheses also serve for a different purpose: they allow extracting the relevant part of the string that matched; for that purpose, the special variables `$1`, `$2`, etc. are employed
- More specifically, Perl seeks the first match in the string and puts those parts into the special variables `$1`, `$2`, etc. that match
- Example: after evaluating

```
('Hello you!' =~ /Hello (world|you.)/)
```

  `$1` has the value "you!"

- Another example: after evaluating

```
('AGCTTATATGCATATATAT' =~ /T(.T|.A)T(.T|A)/)
```

  `$1` has the value "TA" and `$2` has the value "AT"

Regular Expressions: Extracting Matches (2/3)

- Assigning an evaluation of a regular expression to a list stores the matching parts in the list (i.e. the regular expression is evaluated in list context)
- Example: after

```
@list = ('Hello you!' =~ /Hello (world|you.)/)
```

`@list` has the value ("you!") and after

```
@list = ('AGCTTATATGCATATATAT' =~ /T(.T|.A)T(.T|A)/)
```

`@list` has the value ("TA", "AT")

Regular Expressions: Extracting Matches (3/3)

- Note that the extraction of matches discussed until now only concerns the first match of the regular expression in the string — extracting all matches is a different story (see later)!
- Example: after

```
$string = 'red = 0xFF0000, blue = 0x0000FF';
@list = ($string =~ /0x([0-9a-fA-F]{6})/)
```

`@list` will have the value ("FF0000"), but there is presently no way to access the second match

- Further note that quantifiers are greedy in the sense that they try to match as many items as possible; example: after

```
('<B>My homepage</B>' =~ /<(.*)>/)
```

`$1` has the value "B>My homepage</B"

Regular Expressions: Quantifiers Revisited

- Suffixing quantifiers by “?” makes them non-greedy

  `??` match 1 or 0 times, try 0 first

  `*?` match any number of times, but as few times as possible

  `+?` match at least once, but as few times as possible

  `{n,m}?` match at least n and at most m times, but as few times as possible

  `{n,}?` match at least n times, but as few times as possible

  `{n}?` is allowed, but obviously it has the same meaning as `{n}`

- Example: after

```
('<B>My homepage</B>' =~ /<(.*?)>/)
```

  `$1` has the value “B”

Regular Expressions: Modifiers

- A modifier can be appended to a regular expression to alter its interpretation; the following are the most important modifiers:
  - `i` case-insensitive matching
  - `m` treat string as collection of individual lines; interpret `\n` as newline character, `^` and `$` match individual lines
  - `s` treat string as one line; `.` meta-character also matches `\n`
  - `x` whitespaces and comments inside regular expression are not interpreted
  - `g` allow extraction of all occurrences (see later)
- Example: `/a/i` matches all strings containing ‘a’ or ‘A’

Regular Expressions: Extracting All Matches (1/3)

- If the `g` modifier is specified, Perl internally keeps track of the string position
- If a regular expression is applied to the same string again, the search starts at that position where the last search stopped
- This can be repeated until the last search has been found
- Before the regular expression is evaluated first, the string position is `undef`
- If the string is changed between the regular expression is applied, the position is also set to `undef`
- In scalar context, a regular expression with `g` modifier returns false if the pattern has not been found and true if it has been found

Regular Expressions: Extracting All Matches (2/3)

- Example:

```
while ($string =~ /0x([0-9a-fA-F]+)/g)
{
  print "$1\n";
}
```

  This loop extracts all hecadecimal numbers from `$string` and prints them line by line without the “0x” prefix

- For special purposes, the actual position can be determined with the function `pos`; however, note that, after each occurrence, the position points to the next character after the previous match (if counting of characters starts at 0)!

Regular Expressions: Extracting All Matches (3/3)

- In list context, the `g` modifier can be used to extract all matches and store them in a list
- If there are no groupings, all matches of the whole regular expression are placed in the list
- If there are nested groupings, all matches of all groupings are placed in the list in the order in which they are matched, where matches of outer groups precede matches of inner groups
- Example: after

```
@list = ("AGC GAT TGA GAG" =~ /(G(A(T|G)))/g);
```

`@list` has the value (“GAT”, “AT”, “T”, ‘GAG”, “AG”, “G”)

Regular Expressions: Replacements

- Regular expressions also facilitate powerful replacement mechanisms; this is accomplished with the `s///` operator
- The `s///` operator returns the number of replacements
- Example: after

```
$sequence = "AGCGTAGTATAGAG";
$sequence =~ s/T/U/;
```

`$sequence` has the value “AGCGUAGTATAGAG”

Regular Expressions: Replacements (cont'd)

- Modifiers as introduced above work analogously; not surprisingly, the `g` modifiers allows to replace all occurrences of the search string, e.g. after

```
$sequence = "AGCGTAGTATAGAG";
$sequence =~ s/T/U/g;
```

  `$sequence` has the value "AGCGUAGUAUAGAG"

- The special variables `$1`, `$2`, `$3`, etc. allow very tricky replacements, e.g. with

```
s/(\d{2})\.(\d{2})\.(\d{4})/$3\-$2\-$1/g
```

  one can convert all dates from DD.MM.YYYY format to YYYY-MM-DD format

Regular Expressions: Transliteral Replacements

- The operators `tr///` and `y///` (both are equivalent) are available to perform so-called *transliteral replacements*, i.e. the translation of single characters according to a replacement list
- Example: `tr/AB/BC/`, at once, replaces all “A”s with “B”s and all “B”s with “C”s
- Note that this functionality cannot be realized easily with the replacement operator `s///`
- It is also possible to specify ranges, e.g. `tr/a-f/0-5/`

Regular Expressions: The `split` Function

- The highly useful `split` function can be used to split a string into parts that are separated by certain characters or patterns; it returns a list of split strings
- Example: after

```
@fragments = split(/TGA/, "GCATGACGATGATATA");
```

`@fragments` has the value (“GCA”, “CGA”, “TATA”)

- As obvious from the above example, the first argument is a regular expression at which the string is split; note that the split pattern is omitted in the split list
- Nor surprisingly, there is no restriction to fixed search patterns, e.g.

```
split(/\s+/, $string);
```

splits `$string` into single words

Regular Expressions: The `split` Function (cont'd)

- The split function has an optional third argument with which the maximal number of splits can be controlled
- Example: after

```
@fragments = split(/TGA/, "GCATGACGATGATATA", 2);
```

`@fragments` has the value ("GCA", "CGATGATATA")

- The `join` function is the converse function, i.e. it assembles a list of strings into one large string, where a separating character can be inserted; e.g. after

```
@list = ("tri", "tra", "tralala");
$string = join(':', @list);
```

`$string` has the value "tri:tra:tralala".

Useful Functions Not Previously Mentioned

`abs`, `atan2`, `cos`, `exp`, `log`, `sin`, `sqrt`: usual mathematical functions

`int`: converts a floating point number to an integer number (truncates!)

`printf`: allows more flexibility for formatted output than `print`

`length`: get the length of a string

`reverse`: reverse a string or array

`sort`: sort an array

`system`: run external program

`time`: get system time (mostly the number of seconds since Jan 1, 1970)

`localtime`: convert system time into the actual local time

Programming Style (1/2)

- Programming styles deal with improving the readability and therefore the maintainability of the program
- Declare the variables at the beginning of the sub-function, block or program. You should clearly separate the declaration part from the rest
- In order to avoid unnecessary errors declare all your variables before using them. By using `use strict;` you are forced to do so
- For a detailed warning report use the `use warnings;` pragma

Programming Style (2/2)

- Avoid the use of global variables where possible
- Put repetitive code segments into sub-routines
- Try to find meaningful names for the variables and sub-routines
- Try not to use non-local variables in sub-routines. Control the sub-routine by the input parameters.
- Comments!

Final Remarks

- The present slides are only an overview tutorial that concentrates on elements of Perl that are useful for small bioinformatics applications
- Perl actually offers a lot more possibilities
- For more details, discover the world of Perl at
  `http://www.perl.org/`
- Theory is good, but not sufficient — a programming language can only be learned by experience

References and Further Reading

1. Robert Kirrily: *perlintro — a brief introduction and overview of Perl*
   http://perldoc.perl.org/perlintro.html
2. *perlsyn — Perl syntax* (author unknown)
   http://perldoc.perl.org/perlsyn.html
3. Mark Kvale: *perlretut — Perl regular expressions tutorial*
   http://perldoc.perl.org/perlretut.html
4. Ian Truskett: *perlreref — Perl regular expressions reference*
   http://perldoc.perl.org/perlreref.html
5. *Perl Functions A–Z* (multiple unknown authors)
   http://perldoc.perl.org/index-functions.html

References and Further Reading (cont'd)

1. Rex A. Dwyer: *Genomic Perl: From Bioinformatics Basics to Working Code*. Cambridge University Press, 2003.
2. Martin Kästner: *Perl fürs Web*. Galileo Press, Bonn, 2003.