Practical Perl Programming

Universität Potsdam, Institut für Linguistik
Wintersemester 2004-2005

Bryan Jurish
moocow@ling.uni-potsdam.de

Contents

Programs

1 Course Syllabus

- **Week 1** (14.10.) **Administrivia**
  Topics: prerequisites, grading policies, acquiring perl, perl resources, and shameless plugs.
- **Weeks 2-4** (21.10. – 04.11.) **The Mollusc of Your Choice**
  Topics: more shameless plugs with an eensy bit of content, anatomy of a perl script. Reading: Camel, ch. 1.

  04.11.: *Assignment Block 1 release*
- **Weeks 5-10** (11.11. – 16.12.) **Gory Details**
  - **Perl Datatypes**
    Topics: scalars, arrays, hashes, references, and maybe even globs.
  - **Perl Control Structures**
    Topics: expressions, statements, blocks, conditionals, loops, subroutines, declarations.
  - **Perl I/O**
    Topics: files, pipes, sockets, IO::File.
  - **Perl Regular Expressions**
    Topics: regex operators, modifiers, character classes, anchors, ...

  Reading: Camel, ch. 2.

  18.11.: *Assignment Block 1 due*
  16.12.: *Assignment Block 2 release*
- **Weeks 11-12** (06.01. – 13.01.) **Perl References**
  Topics: hard- vs. soft-refs, (anonymous) reference creation, dereferencing, reference counting. Reading: Camel, ch. 4.

  06.01.: *No class*
  13.01.: *Assignment Block 2 due*
- **Weeks 13-15** (20.01. – 03.02.) **Perl Modules**
  Topics: Packages, modules, objects. Reading: Camel, ch. 5.

  03.02.: *Assignment Block 3 release*
  24.02.: *Assignment Block 3 due*

2 Administrivia

2.1 Prerequisites

- Experience with at least one programming language, preferably of the imperative syntax variety (i.e. C(++), PASCAL), **and/or**
- Experience with at least one command-line interpreter or "shell" (i.e. bash, (t)csh, `DOS command.com`).

2.2 New Versions of This Document

The most current version of this document should always be available in a number of formats over the internet, at:

http://www.ling.uni-potsdam.de/~moocow/class/perl

2.3 Questions

Questions, comments, flames, and rotten fruit should be addressed to me: `moocow@ling.uni-potsdam.de`, and should contain the word "Perl" in the "Subject" header. I repeat, **e-mails to me regarding this course should contain the word "Perl" in the "Subject" header**. Otherwise, they run the risk drowing in a sea of spam.

2.4 Grading Policies

Only students interested in acquiring a *Leistungschein* need concern themselves with the grading policies described below.

2.4.1 Blocks

3 assignment-blocks, more or less equally spaced throughout the semester.

2.4.2 Problems

- 3 problems given per block.
- each student may attempt to solve at most 2 problems per block.

2.4.3 Points

Each problem will be assigned a value in "points", which depends on the assignment-block of which that problem is a member:

Block Number	Points per Problem	Maximum Points
1	8	16
2	16	32
3	32	64
Total	–	**112**

2.4.4 Deadlines

- Solutions for each assignment block are due exactly two weeks from the date that block was released.
- $\frac{1}{8}$ of the possible points for each problem will be deducted from the grade for each day past the deadline a late assignment is received.

2.4.5 Revisions

- Each student not receiving the highest possible number of points for his or her solution to a given problem will be given exactly 1 chance to revise that solution. Revisions are due no later than one week after reception of the graded initial solution.
- Choosing to revise an initial solution cannot lower a student's grade.

2.4.6 Platforms

Students are allowed and encouraged to write their programs on the computer of their choice. Solutions will be evaluated on a linux/x86 machine running perl version 5.8.4 – a good test machine in the Institute is `helios`.

2.4.7 Delivery

The solution to each problem should be sent by e-mail (ONE MAIL PER PROBLEM) to me: moocow@ling.uni-potsdam.de

Solution e-mails should contain the word "Perl" in the "Subject:" header (see Section 2.3).

2.4.8 Format

Each solution program should contain a comment near the top of the file containing the student's email address, immatriculation number, and the number of the problem.

2.4.9 Collective Work

Students choosing to work together may do so, provided that all students contributing to the solution of a problem turn in exactly one collective solution to that problem. Each student contributing to a solution submitted collectively by $N > 1$ students will receive $\frac{3}{2N}$ of the total points awarded to that solution. Collective solutions which are not declared as such are hereafter referred to as "cheating" – students contributing to a cheating solution will receive 0 (zero) points for that solution, and forfeit the right to revision for the corresponding assignment block.

2.5 Acquiring Perl

Perl itself is available free of charge for many platforms, including (but not limited to) Linux, BSD, Solaris, HP-UX, VMS, MS-DOS, MS-Windows (3.x, 9x, 2000, XP), MacOS, ...

The official distribution site for perl itself is CPAN, the Comprehensive Perl Archive Network. Binary ports for various operating systems and architectures are available at:

http://www.cpan.org/ports

For users of MS-Windoof, I reccommend the perl distributed as part of the cygwin project,

http://www.cygwin.com

The most popular Perl for Windoof is probably "ActivePerl", available in binary form from:

http://www.activestate.com/ActivePerl

... but any perl will suffice, as long as your programs run under perl 5.8.4 on a linux/x86 machine.

2.6 Perl Resources

Many Perl-related resources are available on the internet. Some of them are:

- **Perl Online Documentation**
  Your Perl should have come with a good deal of online documentation, hereafter referred to as "manpages". See the `perltoc` manpage for a list of the available topics.
- **CPAN**, the Comprehensive Perl Archive Network
  The definitive source for Perl itself, the standard library of Perl Modules, and user-contributed perl modules.
  URL: http://www.cpan.org

- **learn.perl.org**
  Contains much useful information for beginning perl programmers.
  URL: http://learn.perl.org
- **perldoc.com**
  When in doubt, RTFM[1]: most of the documentation available here should have shipped with your perl distribution.
  URL: http://www.perldoc.com/perl5.8.4
- **Perl Mongers**
  Contains a number of handy links.
  URL: http://www.perl.org
- **Perl Mailing Lists**
  Here, one can subscribe to and/or view the archives of the many perl-specific mailing lists.
  URL: http://lists.perl.org
- **perl.com**
  A useful site run by the publishing house O'Reilly and Associates, the employers of Perl's author, Larry Wall.
  URL: http://www.perl.com

2.7 Shameless Plugs

2.7.1 Copying

Perl is freely available under the terms of the Free Software Foundation's standard GNU public license (http://www.gnu.org), or optionally under the terms of the somewhat less restrictive Perl Artistic License. This is a Good Thing.

2.7.2 FAQs and Factoids

- Q: What does perl stand for?
  A: Choose one of the following:
  - **P**ractical **E**xtraction and **R**eport **L**anguage
  - **P**athologically **E**clectic **R**ubbish **L**ister

  ..., or don't.
- Q: Who wrote Perl?
  A: The linguist, system administrator, and All-Around Nifty Guy Larry Wall wrote Perl (to solve a problem that `awk` couldn't handle.)
- Q: How old is Perl?
  A: Perl-1.0 was first released to the usenet newsgroup comp.sources on 17 October, 1987.
- Q: What the heck is Perl good for anyways? A: Read on ...

[1] Read The Friendly Manual

2.7.3 Uses of Perl

- "Perl is a language for getting your job done. ... Perl is designed to make the easy jobs easy, without making the hard jobs impossible."
  [WCS96, p. ix]
- **Easy Jobs**
  - *Reading*
    Finding and reading files, directories, the output of other programs (pipes), or other computers on a network (sockets) and turning that data into something your program can use.

    Also, reading perl programs themselves: Perl's syntax allows programmers to write clear code.
  - *Writing*
    Writing your program's data to files, pipes, sockets, creating and manipulating files and directories, and generating human-readable reports.

    Also, writing a Perl program itself: Perl's flexibility allows programmers many ways to accomplish any given task.
  - *Arithmetic*
    Not just numeric operations, but the manipulation of, conversion between, and agglomeration of basic data structures such as strings, lists, and associative arrays ("hashes").
- **Harder Jobs**
  - *Social Interaction*
    Generating dynamic program code, integration of Perl interpreters into external applications and programs.
  - *Literature*
    Implementing a full-scale all-singing all-dancing slicing dicing paint-your-house opus for *(insert function here)*.
  - *Engineering*
    Calling an arbitrary C function, (via the XS sublayer, BTSOTD[2]), tweaking perl itself, ...

[2]Beyond the Scope of This Document

3 The Mollusc of Your Choice

3.1 The Very Basics: hello.perl

See code in Program 3.1 (hello.perl).

Program 3.1: hello.perl

```
#!/usr/bin/perl -w
print "Hello, Perl.\n";
```

3.1.1 Running perl

- **UNIX**: the "shebang" line
  - For interpreted scripts, the first line of a file has a special format:
    `#!/full/path/to/interpreter arg1 ... argN`
  - Set executable file permission bit:
    `bash$ chmod u+rx hello.perl`
  - Run the program[3]
    `bash$ ./hello.perl`
- **Everywhere**: call perl "by hand"

  `C:> perl -w hello.perl`

  ... where "`C:>`" represents the shell prompt of some arbitrary hypothetical non-UNIX operating system.
- **perl Options**
  `-w` print verbose warnings to the terminal
  `-d` run in debugging mode (see Section 5.3)
  ... and many, many more.
- **More Information**
  See the `perlrun(1)` manpage for more options and details on your own particular flavor of Perl.

  In most cases, Perl documentation should be available via the program `perldoc`, which is usually distributed with Perl itself:

  `bash$ perldoc perlrun`

[3] The UNIX shell prompt is displayed here and forever after as "`bash$`".

3.1.2 Compiling *vs.* Interpreting

First, some working definitions:

- **Compile** (*verb*)
  To translate source code (i.e. of a program) into a binary machine-readable format ("object code") in preparation for the execution or further linking of that code.

  Commonly compiled languages: C, C++, FORTRAN.
- **Interpret** (*verb*)
  To incrementally parse and execute program code.

  Commonly interpreted languages: LISP, Scheme, sh, DOS Batch.

So what does Perl do? Both!

1. Compile
   - Good: detects common errors on program startup.
   - Good: compiled code executes faster.
   - Not so good: program startup is slow.
2. Interpret
   - Good: no explicit compilation required when source code changes.
   - Good: no object code taking up extra disk space.
   - Not so good: interpreter required to run programs.

3.1.3 Program Elements

- `print`
  A builtin perl subroutine[4] which causes its argument(s) to be printed to the standard output (usually the user's terminal).
- `"Hello, Perl.\n"`
  A literal double-quoted string. The sequence "`\n`" translates to a newline character.
- `()`
  Subroutine arguments may or may not be enclosed by round parentheses. In some more complex cases, parentheses are required.
- `;`
  Perl programs are just sequences of *statements*. Each statement[5] is terminated by a semicolon.
- *whitespace*
  Whitespace[6] is ignored everywhere **except** inside strings.

[4] Perl's "subroutines" are also known as "functions", or sometimes "procedures".
[5] Actually, just *simple* statements – see the `perlsyn(1)` manpage for details.
[6] Whitespace includes spaces, TABs, newlines, carriage returns, and form-feeds.

3.2 Scalars: hello-name

What if we want to greet someone by name? See code in Program 3.2 (hello-name.perl).

Program 3.2: hello-name.perl

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

print 'Who are you? ';
$name = <STDIN>;            # read from standard input
chomp ($name);              # remove that pesky newline
print "Hello, $name!\n";
```

3.2.1 Scalar Variables

- **Variables**
  A variable is just a placeholder for some program data (the variable's *value*) which has a symbolic name (the variable's *identifier* or *name*).
- **Scalar Values**
  A scalar variable holds a single elementary datum (a *scalar value*), similar to the "atoms" of PROLOG or LISP. Valid scalar values include strings, numbers, and references.
- **Perl Scalar Variables**
  Scalar variables in Perl ("scalars" for short) are prefixed with a dollar-sign (`$`). Perl variable names must begin with a letter or underscore, and may be followed by up to about 250 letters, digits, or underscores. Examples:

  ```
  $_
  $x
  $var42
  $some_long_variable_name
  ```

  Unlike more restrictive languages such as C or PASCAL, Perl's variables spring into existence as needed.

  See the `perldata(1)` manpage for details.

3.2.2 Program Elements

- `'Who are you? '`
  A single-quoted literal string. No interpolation of variables or backslash-escapes is performed on single-quoted strings.
- `$name`
  A scalar variable whose identifier is "name".

- `STDIN`
  The standard input stream. One of the three builtin filehandles[7], STDIN usually gets its data from the user's keyboard.

- `<STDIN>`
  Line input operator: reads and returns (a single line of) data from a filehandle (here STDIN), including the terminating newline.

  If not used in an assignment (see below), implicitly assigns to the default scalar variable "`$_`".

- `$name = <STDIN>`
  An instance of variable-assignment,

  VAR `=` *EXPR*

  The variable *VAR* on the left-hand side of the "=" is assigned the result of evaluating the expression *EXPR* on the right-hand side; here, the scalar `$name` gets as its value the next string read from STDIN[8].

- `chomp($name)`
  A builtin Perl subroutine which removes the trailing newline character (if any) from its argument, here `$name`.

  If no argument is given, `chomp()` uses the variable "`$_`".

- `"Hello, $name!\n"`
  A double-quoted interpolated string. Variables will be replaced by their respective values in this string when it is evaluated.

- `# read from standard input`
  A comment. Perl comments begin with a hash-mark (#), and continue until the end of the line.

3.3 Lists and Arrays: hello-folks

Maybe we want our program to greet more than just one person.

See code in Program 3.3 (hello-folks.perl).

3.3.1 Lists and Arrays

- **List**
  A list is just an ordered (possibly empty) sequence of data (the *elements* of the list).

- **Array**
  Conceptually almost identical to a "list", arrays typically allow efficient (constant time) access to their elements.

[7] The other two builtin filehandles are STDOUT and STDERR, and only allow output.
[8] Technically, the left-hand-side of an assignment must be an *lvalue*.

Program 3.3: hello-folks.perl

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

@folks = ('moocow', 'Bryan');        # literal list

print 'Who is here? ';               # no interpolation
while (defined($name = <STDIN>)) {
  chomp($name);
  push (@folks, $name);              # add name to array
  print 'Who else is here? ';
}
print "\n";

foreach $name (sort(@folks)) {       # alphabetical sort
  print ("Hello, ", $name, "!\n");   # print a list
}
```

- **Perl Lists**
  Perl lists may contain only scalar values as elements[9]. For Perl, a list is just a type of *value*[10], which provides an evaluation *context*[11].
- **Perl Arrays**
  A Perl array is a variable whose value is a list. Perl array variables are prefixed with an "at-sign" (@). Examples:

  ```
  @_
  @a
  @array42
  @some_long_array_name
  ```

  As with scalars, Perl array variables pop into existence as needed. See the `perldata(1)` manpage for details.

3.3.2 Basic Array Operations

- **Array Indexing Operations**
  - `$array[$n]`
    Returns the `$n`th element of the array `@array`, where `$array[0]` is the first element in `@array`.
    You can use this syntax to assign to a list element, too:

    ```
    $array[$n] = $value;
    ```
  - `$#array`
    Returns the numeric index of the last element of the array `@array`.

[9] Recursive nesting of data structures is only possible in Perl by using *references*.
[10] Other basic value types include scalars, hashes, and (in a polymorphic sort of way) typeglobs.
[11] Other evaluation contexts include scalar context and boolean context.

- **Stack Operations**
  - `pop` *ARRAY*
    Removes and returns a single element from the end of *ARRAY*.
  - `push` *ARRAY*, *LIST*
    Adds the elements of *LIST* to the end of *ARRAY*.
- **Queue Operations**
  - `shift` *ARRAY*
    Removes and returns a single element from the front of *ARRAY*.
  - `unshift` *ARRAY*, *LIST*
    Adds the elements of *LIST* to the front of *ARRAY*.
- **Conversion Operations**
  - `split` *REGEX*, *SCALAR*
    Returns a list resulting from splitting up the string *SCALAR* wherever the regular expression *REGEX* matches.
  - `join` *SEP*, *LIST*
    Returns a string built by concatenating the string-representations of the elements of *LIST*, separated by the string *SEP*.

3.3.3 Program Elements

- `('moocow', 'Bryan')`
  A literal list. Commas (`,`) are used to separate the elements of a Perl list, and the list itself may be optionally enclosed by round parentheses[12].
- `@folks`
  An array variable whose identifier is "`folks`".
- `@folks = ('moocow', 'Bryan')`
  Assignment to an array variable – here, the variable `@folks` gets as its value the list consisting of the strings `"moocow"` and `"Bryan"`, in that order.
- `while ($name = <STDIN>) { ... }`
  An instance of the simple loop construct

  `while (`*EXPR*`)` *BLOCK*

  which behaves as follows:

  1. Evaluate the expression *EXPR* – here, `$name = <STDIN>`.
  2. If *EXPR* evaluates to a true value, then evaluate the sequence of statements in *BLOCK*. and then goto Step 1.
  3. Otherwise, if *EXPR* evaluates to a false value, then exit the loop and continue program execution with the first statement following *BLOCK*.

[12] In many cases, round parentheses are required to resolve precedence conflicts.

This loop works because:

- For Perl, a "false value" is one of the following:
  - The empty string, `''`.
  - The number `0` (zero).
  - The undefined value, `undef`.
- The value of an assignment is the value assigned – here, the line read by `<STDIN>`.
- On end-of-file[13], the `<>` operator returns the undefined value, `undef`.
- If there is still data to be read from `STDIN`, since the terminating newline is included in the value that the `<STDIN>` returns, the expression `$name = <STDIN>` will never evaluate to a truly empty string.

- `push(@folks, $name)`
  A builtin Perl function whose first argument (here, `@folks`) must be an array, and which adds (the values of) all remaining arguments (here, just `$name`) in order onto the end of that array.

- `sort(@folks)`

  `sort` *LIST*

  A builtin Perl function which returns the elements of its argument (a list, here the contents of the array `@folks`) sorted in alphabetical order.

- `foreach $name (sort(@folks)) { ... }`
  An instance of the loop construct:

  `foreach` *VAR* `(`*LIST*`)` *BLOCK*

  which sets the scalar variable *VAR* to each element of the list *LIST* in turn, evaluating *BLOCK* once for each element.

- `("Hello, ", $name, "!\n")`
  Yet another literal list.

- `print("Hello, ", $name, "!\n")`
  Shows that you can `print()` a list just as easily as a scalar.

3.3.4 Black Magic

An elegant but somewhat cryptic variant of the above program (without prompts): See code in Program 3.4 (hello-folks2.perl).

Things we haven't seen yet:

- `map` *BLOCK LIST*
  Returns the result of mapping the scalar `$_` to each element of *LIST* in turn and evaluating *BLOCK* for that element.

[13] On UNIX systems, end-of-file can be input on the keyboard by typing "Ctrl-d". On DOS-based systems, the end-of-file character is "Ctrl-z".

Program 3.4: hello-folks2.perl

```
#!/usr/bin/perl -w
print                # print a list
  map {              # resulting from mapping $_
    chomp;           # ... and chomping it (implicit $_)
    "Hello, $_!\n";  # ... to a greeting string
  } sort(<STDIN>);   # from each input line (in alphabetical order)
```

- *BLOCK* values
  The value of a block is the value of the last statement in the block.

- <>
  The line-input operator <> without a filehandle reads from all files specified on the command-line (as recorded in the array `@ARGV`), or from standard input if no files were specified. See the `perlop(1)` manpage for details.

- <> in list context
  In list context, the line-input operator (with or without a specified filehandle) returns a list of all remaining lines in the filehandle(s) from which it reads.

3.4 Hashes: hello-dialect

Anyone care for a multilingual greeting program? See code in Program 3.5 (hello-dialect.perl).

3.4.1 Hashes and Associative Arrays

- **Associative Arrays**
  An associative array is an unordered set of *values*, each of which is *indexed* (or *keyed*) by some *key*.

- **Hash Tables**
  A hash table is a low-level data structure which allows a scalable and efficient implementation of associative arrays.

- **Perl Hashes**
  Perl hashes are variables whose values are associative arrays[14]. Keys of a perl hash must be strings, and each key has exactly one scalar value per hash. Perl hash variables are prefixed with a percent-sign (`%`). Examples:

  ```
  %_
  %h
  %hash42
  %some_long_hash_name
  ```

[14] Internally, Perl hashes are indeed implemented as hash tables.

Program 3.5: hello-dialect.perl

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

# literal hash of greeting-strings, keyed by dialect
%greetings = ('english' => 'Hello!',  'deutsch' => 'Tag!');

print "What is your favorite dialect? ";
while ($dialect = <STDIN>) {
  chomp ($dialect);
  if (exists($greetings{$dialect})) {        # existence check
    print $greetings{$dialect}, "\n";        # hash lookup
  }
  else {
    print ("Sorry, I don't speak '$dialect' ",
           " -- how should I greet you? ");
    $greeting = <STDIN>;
    chomp ($greeting);
    $greetings{$dialect} = $greeting;        # hash assignmentg
    print "Well, then -- ", $greeting, "\n";
  }
  print "What is your next favorite dialect? ";
}

# Summarize what we know
print "\n\nI now know the following greetings:\n";
foreach $dialect (keys(%greetings)) {        # hash iteration
  print "\t $dialect:\t $greetings{$dialect}\n";
}
```

Yup, you guessed it – Perl's hash variables come to be as they are needed. See the `perldata(1)` manpage for details.

Hashes are one of the most useful (and most common) elements of Perl. Get used to them.

3.4.2 Basic Hash Operations

- `$hash{$key}`
  Returns the value associated with the string value of `$key` in the hash `%hash`.

  You can use this syntax to assign the value for `$key` in `%hash`, too:

  ```
  $hash{$key} = $value;
  ```

- `keys(%hash)`
  Returns an unordered list of all the keys of the hash `%hash`.

3.4.3 Program Elements

- `('english' => 'Hello!', 'deutsch' => 'Tag!')`
  Actually just a list, the operator `=>` can be used to associate hash-keys (on the left of the `=>`) with their values (on the right).

- `%greetings = ('english' => 'Hello!', 'deutsch' => 'Tag!')`
  Literal hash-assignment. The hash `%greetings` associates the key-string "`english`" with the scalar value "`Hello!`" (also a string), and the key-string "`deutsch`" with the scalar value "`Tag!`" (yet another string).

- `exists($greetings{$dialect})`
  A builtin perl function which returns a true value iff[15] the hash[16] variable `%greetings` contains a value for the key `$dialect`.

- `if (exists($greetings{$dialect})) { ... } else { ... }`
  An instance of the conditional construct:

  `if (|` *EXPR* `)` *BLOCK* `else` *BLOCK*

  which first evaluates *EXPR*, evaluating the first *BLOCK* if *EXPR* returns a true value, otherwise evaluating the second *BLOCK*.

  Other variations of the conditional construct include:

  `if (` *EXPR* `)` *BLOCK*
  `if (` *EXPR* `)` *BLOCK* `elsif (` *EXPR* `)` *BLOCK* ... `else` *BLOCK*

- `keys(%greetings)`
  Returns a list containing all the keys of the hash `%greetings`.

[15] "If and only if"
[16] This works for arrays, too: `exists($array[$index])`

- `"\t $dialect:\t $greetings{$dialect}\n"`
  Yet another double-quoted string literal. The hash-indexing operation `$greetings{$dialect}` will be replaced by the value for the key `$dialect` in the hash `%greetings` in this string when it is evaluated – just as the scalar `$dialect` will be replaced by its value[17].

3.5 Filehandles: hello-file.perl

What if we want to store different greetings for each person we know in a separate file?

See code in Program 3.6 (hello-file.perl).

Program 3.6: hello-file.perl

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

## -- open the greetings file
open(GREETINGS, "<greetings.txt")
  or die("$0: could not open 'greetings.txt': $!");

## -- read in known greetings
while (<GREETINGS>) {                      # assign to $_
  ($name,$greeting) = split(/\s+/,$_,2);  # ... convert to list
  chomp($name,$greeting);                  # ... chomp() a list
  $greetings{$name} = $greeting;           # ... associate
}
close(GREETINGS);

## -- say hello
print STDERR 'Who are you? ';
$name = <STDIN>;
chomp($name);
if (exists($greetings{$name})) {         # do I know you?
  print STDOUT $greetings{$name}, "\n";  # ... use greeting from file
} else {
  print STDOUT "Nice to meet you, $name!\n";   # default greeting
}
```

Here's the greetings file itself:

See Datafile 3.1 (greetings.txt).

3.5.1 Streams and Filehandles

- **Streams**
  A stream is just a "channel" through which data may (or may not) flow.

[17] As you might expect, value interpolation into double-quoted strings works for array-indexing operations as well.

Datafile 3.1: greetings.txt

```
moocow      Moo!
Bryan       Hey now!
```

Streams come in two basic flavors: *input* streams and *output* streams. Intuitively, you may read data from an input stream, and you may write data to an output stream.

Typical examples of streams: files, pipes, sockets.

- **Perl Filehandles**
  Perl uses filehandles to represent streams. By convention, filehandle names are written in upper-case. "Pure" Perl filehandles are always literals – see the `IO::File(3pm)` module manpage for one workaround.

- **Standard Filehandles**
  UNIX operating systems (and many others) define the following three standard streams for each process:

C Name	Perl Name	Description
stdin	STDIN	Standard input (keyboard)
stdout	STDOUT	Standard output (terminal)
stderr	STDERR	Standard error-output (terminal)

3.5.2 Basic Filehandle Operations

- `open(HANDLE,"<$filename")`
  Open the filehandle HANDLE for reading from the file `$filename`. Returns a true value iff `$filename` was opened successfully.

  And yes, HANDLE will be created if it did not already exist.

- `open(HANDLE,">$filename")`
  Open HANDLE for writing to the file `$filename`, overwriting the file's previous contents (if any). Returns a true value iff `$filename` was opened successfully.

- `close(HANDLE)` Closes the filehandle HANDLE.

3.5.3 Program Elements

- `GREETINGS`
  A filehandle name.

- `open(GREETINGS,"<greetings.txt")`
  Opens the filehandle `GREETINGS` for reading from the file `greetings.txt`. See Section 3.5.2, above.

- `or`
  The binary boolean disjunction operator. Perl also accepts C-style boolean operators. Note that as in C or LISP, Perl performs "short-circuit" evaluations of booleans – this means that in the expression "`1 or 0`", only the "1" gets evaluated, while in the expression "`0 or 1`", both terms must be evaluated.

- `$0`
  A builtin Perl variable whose value is the name of the running program.

- `$!`
  A builtin Perl variable whose value is the operating system's current error message – useful when builtin OS-interface functions fail.

- `die("$0: could not open 'greetings.txt': $!")`
  A builtin Perl function which causes the running program to print an error message to STDERR and immediately terminate.

- `open(...) or die(...)`
  A common idiom for opening a filehandle safely which causes the program to exit if the file could not be opened[18].

- `chomp($name,$greeting)`
  The builtin function `chomp()` takes lists, as well.

- `close(GREETINGS)`
  Closes the filehandle `GREETINGS`. See Section 3.5.2, above.

- `STDERR`
  One of the three standard filehandles, see Section 3.5.1, above.

- `print STDERR 'Who are you? '`
  An instance of the builtin perl function:

  `print` *HANDLE LIST*

  which prints its argument *LIST* to the filehandle *HANDLE*. Note the lack of a comma between *HANDLE* and *LIST*. Up until now, we have seen the `print()` function without a *HANDLE* argument, which implicitly prints to `STDOUT`.

3.6 Subroutines: hello-sub

Maybe our Perl script meets a lot of people at a lot of different places, and we don't want to have to retype our greeting script from Section 3.2 every time we see someone we know. We can write a subroutine to do that for us:

See code in Program 3.7 (hello-sub.perl).

[18] Why could the file not be opened? Maybe it didn't exist, maybe the user lacks the required permissions, maybe the OS is feeling vindictive – in any case, the contents of `$!` should give the official reason.

Program 3.7: hello-sub.perl

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

## undef = hello($name)
##  + subroutine to greet $name abstractly
sub hello {
  my $name = shift(@_);        # get our first and only argument
  print "Hello, $name!\n";     # ... and greet 'em
}

## undef = do_something(@what)
##   + sets global $name to $what[0]
sub do_something {
  print "(doing something: ", @_, ")\n";
  $name = $_[0];              # side effect: set $name
}


# ... meet some folks, assign the variable $name ...
do_something('whatever');
hello($name);                 # greet someone


# ... meet some other folks, re-assign $name
do_something('else');
hello($name);                 # greet someone else
```

3.6.1 Subroutines and Other Animals

- **Procedures**
  A procedure is a chunk of program code that you, the programmer, can assign a name and save to call later, abstracting over one or more variable values (*parameters*, or *arguments*).

  A good example is the builtin Perl procedure `print()`.

- **Functions**
  A function is just like a procedure, but also "returns" a meaningful value[19] to its caller – usually, this value is computed from the function's parameters.

  A good example of a function is the builtin Perl function `exists()`.

- **Perl subroutines**
  A Perl subroutine can be either a procedure or a function[20] – that is, it *can* return a meaningful value, but is not obliged to do so.

  Unlike more restrictive languages such as C or PROLOG, Perl subroutines do not need to pre-declare the number or types of their parameters or return values.

  See the `perlsub(1)` manpage for details on Perl subroutines.

3.6.2 Program Elements

- `sub hello { ... }`
  An instance of the subroutine definition schema:

  `sub` *IDENTIFIER BLOCK*

  which defines a subroutine named *IDENTIFIER* – in this case, "hello" – whose body is *BLOCK*. All this really means is that you can later write something like `hello($arg1,$arg2)` and the code *BLOCK* will be evaluated with the special parameter-array variable `@_` set to the list `($arg1,$arg2)`.

- `@_`
  A special array variable which holds the parameters (if any) given in the call to the currently executing subroutine.

- `my $arg`
  An instance of the declaration schema:

  `my` *VAR*

  The keyword "my" causes its argument variable(s) to be *lexically scoped*[21]. All this basically means is that a new scalar `$arg` magically pops into existence when perl evaluates the declaration "`my $arg`", which then ceases

[19] Or values.
[20] Or even an object method, but we'll get to that later, maybe.
[21] As opposed to *globally scoped* or *global* variables.

to exist as soon as evaluation passes out of the smallest block[22] containing that declaration. Usefully, you can assign to a "my" declaration, as in Program 3.7 (hello-sub.perl):

```
my $arg = shift(@_);
```

Which is a common idiom for assigning the first argument of a subroutine to a lexically scoped scalar variable. Also commonly seen is something like the following:

```
my ($arg1,$arg2,$arg3) = @_;
```

. . . which uses a list-context variant of the "my" keyword to simultaneously declare and assign three subroutine arguments to three scalar variables.

Other useful idioms include:

```
my ($req1,@opt) = @_;  # required and optional arguments
my ($req1,%kws) = @_;  # keyword arguments as a hash
```

- `hello($name)`
  A call to the subroutine named `hello`, with the argument `$name`. Sometimes appears in older[23] code as: `&hello($name)`.

[22] Or subroutine, or `eval()`, or package.
[23] pre Perl-5

4 The Gory Details

4.1 Perl Syntax

> "A Perl script consists of a sequence of declarations and statements."
>
> the `perlsyn(1)` manpage

Well, that's just great, but what the heck are "declarations" and "statements"? Anyone seeking some working definitions of these animals, read on – but note that the working definitions in this section are *post-hoc* generalizations based on my own personal experience with Perl, and do not give a full and complete account of Perl's syntax or semantics.

4.1.1 Comments and Whitespace

Perl comments begin with a "`#`" (hash-mark) character, and continue to the end of the line. Example:

```
# This text is ignored by Perl.
```

Whitespace is ignored just about everywhere by Perl, except in strings, and in regular program text where it may be used to separate tokens. Whitespace is also significant in formats.

4.1.2 Terms and Values

Terms come in three typological flavors: scalars, lists, and hashes.

- **Simple Terms**
  A *simple term* may be either a literal value or variable. Some simple examples:

Simple Term	Type	Description
`"moo"`	scalar	String literal
`42`	scalar	Integer constant
`420.2407`	scalar	Floating-point constant
`$x`	scalar	Scalar variable
`@a`	array	Array variable
`%h`	hash	Hash variable

- **Complex Terms**
  There are also creatures commonly called *complex terms* – terms which contain operators (such as the array-indexing bracket operator "`[]`") and expressions (such as an index).

Complex Term	type	Description
`$a[0]`	scalar	Array element
`$h{moo}`	scalar	Hash element
`("moo", "cow")`	list	List literal
`@a[0,1]`	list	Array slice
`@h{'moo','cow'}`	list	Hash slice

- **Values**
  Every term has a semantic *value*, which is usally what you would expect it to be. The integer constant `42` for example, has as its value the (abstract, conceptual) number 42.

- **Lvalues**
  An *lvalue* is just a term which can appear on the left-hand (receiving) side of an assignment. Valid Perl lvalues include all types of variable, indexing operations, and additionally list literals whose every member is an lvalue. Examples:

Lvalue	Type	Description
`$x`	scalar	Scalar variable
`@a`	array	Array variable
`%h`	hash	Hash variable
`$a[0]`	scalar	Array element
`$h{'moo'}`	scalar	Hash element
`($x,$y)`	list	List of lvalues

4.1.3 Expressions

An *expression* is (for practical purposes) any Perl syntactic construct that has a value. Valid Perl expressions include terms, operator applications, subroutine calls, and parenthesized expressions. Examples:

Expression	Description
`$x`	Term (scalar variable)
`$x + $y`	Addition operation
`$x = 42`	Assignment operation
`foo('bar')`	Subroutine call
`($x + 1)`	Parenthesized expression

4.1.4 Statements

A *statement* is either an expression whose value we choose to ignore (by use of the semicolon "`;`" operator), or a directive to the Perl compiler, known as a *pragma*. For most purposes, pragmata may be considered a type of statement.

Statement	Description
`$foo = 'bar';`	Semicolon-terminated expression
`use CGI;`	External module inclusion pragma
`no warn;`	Syntax-warning disablement pragma

4.1.5 Blocks

A *block* is just a sequence of statements, enclosed by curly brackets "`{}`". The value of a block is the value of the last expression in that block. Blocks may be understood as "compound statements". Example:

```
{
  $bar = 42;
  $foo = $bar;
}
```

4.1.6 Declarations

A *declaration* is a statement which defines and creates a variable or subroutine.

Declaration	Description
`my ($foo,$bar);`	Variable declarations
`sub add { return $_[0] + $_[1]; }`	Subroutine declaration

While global variables do not need to be declared in Perl, it is usually a good idea to lexically scope your variables by declaring them with `my`.

4.1.7 Control Structures

Other kinds types of complex statements include conditional expressions and various looping constructs. Details on these can be found in Section 4.3.

4.2 Perl Datatypes

"This may seem a bit weird, but that's okay, because it is weird."

[WCS96, p. 37]

4.2.1 Scalars

Scalars represent "singular" data – numbers, strings, and references.

- **Scalar Literals** (aka "constants")
  - *Numbers*

Literal	Description
`undef`	Undefined value
`-3.14195`	Floating-point constant ($-\pi$)
`2.997925e8`	SI float constant (speed of light)
`42`	Decimal integer constant (the answer)
`010`	Octal integer constant (decimal 8)
`0xff`	Hexidecimal integer constant (decimal 255)

  - *Character*[24] *Strings*

Literal	Description
`'moo'`	Single-quoted string
`"cow"`	Double-quoted string
`q(moo)`	Single-quoted string (alternate)
`qq(cow)`	Double-quoted string (alternate)

- **Scalar Variables**

  Scalar variables are identified by the funny character prefix "`$`" (dollar sign).

- **Scalar Terms**

  Scalar terms include scalar literals, scalar variables, indexing operations on arrays and hashes, as well as certain builtin perl subroutines and declarations.[25]

Scalar Term	Description
`$s`	Scalar variable
`$a[0]`	Array indexing operation
`$h{'moo'}`	Hash indexing operation

- **Common Operations**

[24] For those who might be wondering, a Perl "character" is a creature at least 8 bits wide, and might sometimes be wider. See the documentation of your Perl – especially the `perllocale(1)` manpage and the `perlunicode(1)` manpage – for details.

[25] I frequently lump some of these notions together and refer to scalar terms as simply "scalars", and to non-literal scalar terms as "scalar variables" (or even just "scalars") – if this seems confusing, it is: when in doubt, construct a test case and feed it to perl.

– *Assignment*

All assignments in Perl have the same format:

Schema	Example	Description
LVALUE `=` *EXPR*	`$x = $y`	Assignment: read "`$x` gets `$y`"

The value of an assignment is the value assigned.

– *Implicit Conversion*

Scalars are implicitly converted from numbers to strings and vice versa, depending on how they are used – guaranteeing that a variable used as a number actually contains a meaningful numeric value is the programmer's problem.

A single scalar used in list context looks just like a list with only one element, which is exactly how it's treated.

The undefined value "`undef`" evaluates to zero when used as a number, and evaluates to the empty string when used as a string – in either of these cases, expect "`perl -w`" to complain.

- **Logical Operations** (aka "boolean operations")

– *Truth*

∗ "False" scalar values are all and only the following:

Value	Description
''	Empty string
0	Number zero
undef	Undefined value (reduceable)

∗ Every scalar value which is not a "false" value is considered to be a "true" value. By convention, the canonical "true" value is the number 1.

– *Scalar Comparisons*

Schema	Example	Description
EXPR `eq` *EXPR*	`$x eq $y`	String equality
EXPR `ne` *EXPR*	`$x ne $y`	String inequality
EXPR `gt` *EXPR*	`$x gt $y`	String greater-than
EXPR `ge` *EXPR*	`$x gt $y`	String greater-or-equal
EXPR `lt` *EXPR*	`$x lt $y`	String less-than
EXPR `le` *EXPR*	`$x lt $y`	String less-or-equal
EXPR `==` *EXPR*	`$x == $y`	Numeric equality
EXPR `!=` *EXPR*	`$x != $y`	Numeric inequality
EXPR `>` *EXPR*	`$x > $y`	Numeric greater-than
EXPR `>=` *EXPR*	`$x >= $y`	Numeric greater-or-equal
EXPR `<` *EXPR*	`$x < $y`	Numeric less-than
EXPR `<=` *EXPR*	`$x <= $y`	Numeric less-or-equal

– *Logical Expressions*

Schema	Example	Description
EXPR `&&` *EXPR*	`$x && $y`	C-style logical "and"
EXPR `||` *EXPR*	`$x || $y`	C-style logical "or"
`!` *EXPR*	`! $x`	C-style logical "not"
EXPR `and` *EXPR*	`$x and $y`	PASCAL-style logical "and"
EXPR `or` *EXPR*	`$x or $y`	PASCAL-style logical "or"
`not` *EXPR*	`not $x`	PASCAL-style logical "not"

Perl's logical expressions are subject to "short-circuit" evaluation – this means that as few expressions as possible are evaluated by the Perl interpreter to find their value.

Expression	Value
`$x && $y`	`$x` if `$x` is false, otherwise `$y`
`$x || $y`	`$x` if `$x` is true, otherwise `$y`
`!$x`	true if `$x` is not true

- **Numeric Operations**

Schema	Example	Description
EXPR `+` *EXPR*	`$x + $y`	Addition
EXPR `-` *EXPR*	`$x - $y`	Subtraction
EXPR `*` *EXPR*	`$x * $y`	Multiplication
EXPR `/` *EXPR*	`$x / $y`	Division
EXPR `%` *EXPR*	`$x % $y`	Modulus (remainder)
EXPR `**` *EXPR*	`$x ** $y`	Exponentiation
VAR `+=` *EXPR*	`$x += $y`	Addition with assignment
VAR `-=` *EXPR*	`$x -= $y`	Subtraction with assignment
⋮	⋮	⋮
VAR`++`	`$x++`	Autoincrement, returns old value
VAR`--`	`$x--`	Autodecrement, returns old value
`++`*VAR*	`++$x`	Autoincrement, returns new value
`--`*VAR*	`--$x`	Autodecrement, returns new value

- **String Operations**

  - *String Concatenation*

Schema	Example	Description
EXPR `.` *EXPR*	`'moo' . 'cow'`	String concatenation
EXPR `x` *EXPR*	`'moo' x 3`	String repetition

  - *Substring Access (string indexing)*

Schema	Description
`length` *STR*	String length
`index` *STR*, *SUB*	Substring search
`index` *STR*, *SUB*, *POS*	Substring search (from *POS*)
`substr` *STR*, *OFFSET*	Suffix extraction (lvalue)
`substr` *STR*, *OFFSET*, *LEN*	Substring extraction (lvalue)
`vec` *STR*, *OFFSET*, *BITS*	Binary substrings (lvalue)

– *String Interpolation*

Various funny characters and variable values can be "interpolated" (read "substituted") into string literals. What gets interpolated where depends on the flavor of string you're interpolating.

∗ *Single-Quoted Strings*

Escape	Example	Description
`\'`	`'moo\'s cow'`	Embedded quote
`\\`	`'C:\\DOS'`	Escaped backslash

∗ *Double-Quoted Strings*

· *Escapes*

Escape	Description
`\"`	Embedded quote
`\n`	Newline
`\r`	Carriage Return
`\t`	Horizontal tab
`\$`	Escaped dollar-sign
`\@`	Escaped at-sign
`\\`	Escaped backslash

Perl allows other C-style escapes in double-quoted strings as well. See the `perlop(1)` manpage for details.

· *Variable Interpolation*

Double-quoted strings (but not single-quoted strings) are subject to "variable interpolation", substituting variables' values for the respective variables where those variables occur in the string, thus the following code:

```
$x = 42; $str = "The answer is $x";
```

is equivalent[26] to:

```
$str = "The answer is 42";
```

Variable identifiers may also be surrounded by curly brackets: this can be useful to delimit variables from the surrounding text:

```
print "2 to the ${x}th power is ", 2**$x;
```

∗ *Backtic Strings*

Backtic strings are interpreted as the command-lines of external programs which Perl should call. The value of a backtic string

[26] As far as `$str` is concerned, that is.

is the text printed by external program called to its standard output (if any). Example:

```
$str = `echo moo`;
```

* *"Here" Documents*
  Shell-style "here" document string literals are supported by Perl, and may specify their quoting style:

```
print <<"EOF";
This is a test.
This is ONLY a test.
EOF
```

- **Bitwise Operations**

  Perl's bitwise operations follow C syntax, and can operate on numbers or binary strings. See the `perlop(1)` manpage and the entries for pack() and unpack() in the `perlfunc(1)` manpage for details.

- **Special Scalars**

  Perl has a plethora of special builtin scalar variables, the values of which you, the programmer, may (or may not) examine, redefine, or otherwise manipulate.

Variable	Description
`$_`	General-purpose default variable
`$&`	Last regex pattern match
`$.`	Line number of current input filehandle
`$|`	Output autoflush flag
`$!`	Last OS error
`$^E`	OS-specific error information
`$@`	Last eval() error
`$0`	Name of current program
`$^W`	Warning flag

  See the `perlvar(1)` manpage for more.

4.2.2 Context

Perl knows about two major "contexts": scalar context and list context. An expression may appear to have different values depending on which context it gets evaluated in.

How do I know which context my expression is being evaluated in?

- **Easy Cases**

  – *Assignment*
  If the left-hand side of an assignment looks like a list[27], Perl will evaluate the right-hand side in list context, otherwise the right-hand side will be evaluated in scalar context.

[27] Any left-hand side which is an array, hash, slice, or list of lvalues looks like a list to Perl.

- *Force Scalar Context:* `scalar()`

  Perl's builtin `scalar()` function forces its argument to be evaluated in scalar context.

- *Encourage List Context:* `(EXPR)`

  Enclosing your expression with round parentheses is a good way to encourage perl to evaluate it in list context – this doesn't always work, however.

- **Not Quite so Easy Cases**

  - *Operator Expressions*

    Operators "supply" either list or scalar context to their operands – see the documentation for the operator in question in the `perlop(1)` manpage for details.

  - *Subroutine Calls*

    Context propagates into subroutine calls. You can use the builtin `wantarray()` function within a subroutine to determine in which context your subroutine was called. See the `wantarray()` entry in the `perlfunc(1)` manpage for details.

- **Other Cases**

  You may hear and/or read about such animals as "void context", "don't-care context", "boolean context", "interpolative context", and/or "object context" – these can generally be considered variations on the theme of "scalar context", described above.

4.2.3 Lists and Arrays

Lists are ordered (possibly empty) sequences of scalar values, indexed by number.

- **List Literals** (aka "constants")

  - *General*

    List literals are usually enclosed by round parentheses, elements are separated by commata. Example:

    ```
    ('moo', 'the', 'cow')
    ```

  - *Word-lists*

    A special form of list literal can be used for lists of space-separated literal strings:

    ```
    qw(moo the cow)
    ```

- **List Variables: Arrays**

  Arrays are identified by the funny character prefix "`@`" (at sign).

- **List Terms**

  List terms include list literals, arrays, slice operations on arrays and hashes, as well as certain builtin perl subroutines.[28]

List Term	Description
`@a`	Array variable
`@a[0,1]`	Array slice operation
`@h{'moo','cow'}`	Hash slice operation

- **Common Operations**

  - *Assignment*

    List/array assignments have the common assignment format:

    LVALUE `=` *EXPR*

    List lvalues include array variables, slices, and list literals containing only lvalues.

  - *Implicit Conversions*

    In scalar context, a list evaluates to the number of elements it contains (the empty list evaluates to zero).

- **List Truth**

  The only false list value is a list without any elements. This is because:

  1. Boolean context acts like scalar context.
  2. The value of a list in scalar context is the number elements it contains.
  3. The only false numeric (scalar) value is the number zero.

- **Indexing Operations**

Schema	Example	Description
`$#`*ARRAY*	`$#ary`	Array length (final index)
`$`*ARRAY* `[` *INDEX* `]`	`$ary[$i]`	Array element access
`(` *LIST* `)` `[` *INDEX* `]`	`(qw(a b))[$i]`	List element lookup
`@`*ARRAY* `[` *LIST* `]`	`@ary[$i,$j]`	Array slice

- **List Interpolation**

  - *Scalars*

    The value of a scalar variable replaces that variable whenever the variable occurs in a list literal (except for `qw()` word-lists, of course). Similarly the value of an array-, list-, or hash-element lookup term replaces that term in the value of a list literal. This means that the following code:

    ```
    $x = 42;
    @ary = ("The answer is: ", $x);
    ```

    is equivalent[29] to:

    ```
    @ary = ("The answer is: ", 42);
    ```

[28] I lump these notions together and refer to list terms as "lists".

[29] Regarding only `@ary`, that is.

– *Lists*

Since lists may contain only scalar values, no embedding of list or array values is allowed[30] – whole lists are "interpolated" flatly into other lists, thus the following code:

```
@ary1 = qw(moo);
@ary2 = qw(cow);
@ary3 = (@ary1, qw(says the), @ary2);
```

is equivalent[31] to:

```
@ary3 = qw(moo says the cow);
```

- **Some Useful List Functions**

Schema	Description
`split` */PATTERN/*`,` *EXPR*	scalar → list conversion
`join` *EXPR*`,` *LIST*	list → scalar conversion
`sort` *LIST*	Alphabetical sort

See the `perlfunc(1)` manpage for more.

- **Special Arays**

Perl has a number of special builtin array variables to enrich your programming experience.

Variable	Description
`@ARGV`	Command-line script arguments
`@_`	Subroutine parameters
`@INC`	Module search path

See the `perlvar(1)` manpage for more.

4.2.4 Hashes

Hashes (or "associative arrays") are unordered collections of scalar values indexed (or "keyed") by character strings[32]. To put it another way, "a hash is just a funny kind of array in which you look values up using key strings instead of numbers" [WCS96, p. 50].

- **Hash Literals** (aka "constants")

There is no such thing a hash literal. Lists of *key,value* pairs serve the same purpose – but see the description of the "`=>`" operator, below.

- **Hash Variables**

Hash variables are identified by the funny character "`%`" (percent sign).

- **Common Hash Operations**

[30] You need references to do neat recursive things like this.

[31] Well, at least as far as `@ary3` is concerned.

[32] Of course, since Perl implcitly converts all scalar values to strings as needed, you can use any scalar value as a hash key. In the case of references, you'll need to fiddle a bit if you want to get the original value back, though.

– *Implicit Conversions*

In list context, hashes are evaluated as lists of *key,value* pairs. Conversely, a list[33] of *key,value* pairs can be directly used to populate a hash, by just assigning it to the hash.

In scalar context, empty hashes evaluate to the number 0 (zero); nonempty hashes evaluate to a funny looking string – have a look at the `perldata(1)` manpage to learn what it means.

– *Assignment*

Hash assignments have the common assignment format:

LVALUE `=` *EXPR*

The only[34] hash lvalues are hash variables. The right-hand-side of a hash assignment is evaluated in list context, and is interpreted as a series of *key,value* pairs which are used to populate the hash on the left.

– *The "=>" Operator*

To improve readability, you can use the "arrow" operator "`=>`" when constructing lists which will be used as hashes. It works almost just like a comma "," except that "`=>`" forces the term immediately preceeding it (assumed to be a hash key) to be interpreted as a string, so you can omit quotes – providing your hash keys don't contain spaces. Example:

```
%h = (
   cow            => 'moo',
   cat            => 'meow',
   'deep thought' => 42,
);
```

- **Hash Truth**

  The only false hash value is an empty hash. The reasoning is analagous to the case of truth for lists.

- **Indexing Operations**

Schema	Example	Description
`$`*HASH* `{` *KEY* `}`	`$h{'moo'}`	Hash value access
`@`*HASH* `{` *LIST* `}`	`@h{@keys}`	Hash slice

Usefully, the hash indexing operator brackets "`{}`" force their arguments to be interpreted as strings – so you can omit the quotes on literal hash keys – providing these don't contain spaces, just like with the "`=>`" operator:

```
$h{moo} = 'cow';
```

is equivalent to:

```
$h{"moo"} = 'cow';
```

[33] Or array, or slice, ...

[34] I'm ignoring references here for the sake of convenience.

- **Hash Interpolation**

  Since there is no such thing as a hash literal, there is no such thing as hash interpolation. Implicit conversion to and from lists of key-value pairs tells you pretty much everything you need to know, though.

- **Some Useful Hash Functions**

Schema	Example	Description
`keys` *HASH*	`keys(%h)`	Get list of hash keys
`values` *HASH*	`values(%h)`	Get list of hash values
`each` *HASH*	`each(%h)`	(key,value) iterator for loops
`delete` *INDEX*	`delete($h{moo})`	Remove a hash entry

- **Special Hash Variables**

  Perl has a number of special builtin hash variables to bring joy to your existence:

Variable	Description
`%ENV`	OS environment variables
`%SIG`	Signal-handling subs
`%INC`	Keeps track of included modules

  See the `perlvar(1)` manpage for more.

4.2.5 Subs and Code

Subroutines will be dealt with in more detail in Section 4.3. You should know now that subroutines (and `CODE` references) are identified by the funny character "`&`" (ampersand).

4.2.6 Typeglobs and Filehandles

Typeglobs can be understood as a "meta-datatype" which stores whole symbol-table entries. Typeglobs are identified by the funny character prefix "`*`" (asterisk). Their main use is to create aliases for variables or subroutines:

```
$x = 42;
*answer = \$x;              # Variable alias

sub foo { return 42; }
*bar = \&foo;               # Subroutine alias

$baz = 'bonk';
@baz = qw(foo bar);
*bonk = *baz;               # Multiple aliases
```

Another, older use for for typeglobs is for saving filehandles to a variable. These days, there are lovely modules to handle such things. Filehandles will be dealt with in more detail in Section 4.4.

4.2.7 Regular Expression Patterns

For present purposes, a regular expression pattern ("regex" or "pattern" for short) is just a funny-looking string – either a literal or a scalar. Regular expressions will be dealt with in more detail in Section 4.5.

4.2.8 References

References work much like C "pointers" to one of the basic Perl datatypes – `SCALAR`, `ARRAY`, `HASH`, or `CODE`. References are themselves scalar values, which means that you can use them to construct nested data structures. You can construct a reference from an existing variable by prefixing the variable with a backslash "\" in addition to that variable's normal funny-character prefix. References will be covered in more detail in Section 4.6.

4.3 Perl Control Structures

4.3.1 Conditionals

- **Compound Conditionals**

  `if (`*EXPR*`)` *BLOCK*
  `if (`*EXPR*`)` *BLOCK* `else` *BLOCK*
  `if (`*EXPR*`)` *BLOCK* `elsif (`*EXPR*`)` *BLOCK* ...
  `if (`*EXPR*`)` *BLOCK* `elsif` ... `else` *BLOCK*

  Evaluates the conditional expressions *EXPR* in boolean context in the order they appear until some *EXPR* evaluates to a true value, in which case the *BLOCK* immediately following that *EXPR* is evaluated. If none of the *EXPR*s evaluates to a true value and the final *BLOCK* of the "`if`" statement is introduced by the keyword "`else`" , evaluates the final *BLOCK*. At most one of the *BLOCK*s is evaluated. Returns the value of the *BLOCK* evaluated, if any.

  The "`unless`" statement provides a handy shorthand for the first two forms of the "`if`" statement with an implicitly negated conditional expression:

  `unless (`*EXPR*`)` *BLOCK*
  `unless (`*EXPR*`)` *BLOCK* `else` *BLOCK*

  are equivalent to, respectively:

  `if (!(`*EXPR*`))` *BLOCK*
  `if (!(`*EXPR*`))` *BLOCK* `else` *BLOCK*

- **Conditional Modifiers**

  STATEMENT `if` *EXPR*`;`
  STATEMENT `unless` *EXPR*`;`

  Simple statements may occur with conditional modifiers just before the statement-terminating semicolon. The schemata above are equivalent to:

  `if (`*EXPR*`) { `*STATEMENT*`; }`
  `unless (`*EXPR*`) { `*STATEMENT*`; }`

- **Conditional Operators**

  TEST `?` *IF-EXPR* `:` *ELSE-EXPR*

  The C-style conditional test operator "`?...:`" provides a useful shorthand for conditional statements which can be easily nested within other expressions. It first evaluates the expression *TEST* in boolean context, and then evaluates and returns the value of the expression *IF-EXPR* just in case *TEST* evaluated to a true value, otherwise evaluates and returns the value of the expression *ELSE-EXPR*.

- **Examples**

```
### A simple test
if ($x < 42) {
  print "x is less than forty-two!\n";
}

### Simple test with default case
if ($x % 2 == 0) {
  print "$x is even.\n";
}
else {
  print "$x is odd.\n";
}

### Even/odd test using the conditional operator
print "$x is ", $x % 2 == 0 ? "even" : "odd", ".\n";

### If the Clash were Perl hackers...
if ($I_go) {
  print "there will be trouble.\n";
}
elsif ($I_stay) {
  print "it will be double.\n";
}
```

4.3.2 Loops

- **"while" Loops**

  `while (`*EXPR*`)` *BLOCK*
  `while (`*EXPR*`)` *BLOCK* `continue` *BLOCK*
  LABEL`: while (`*EXPR*`)` *BLOCK*
  LABEL`: while (`*EXPR*`)` *BLOCK* `continue` *BLOCK*

  Keeps evaluating the first *BLOCK* as long as the conditional expression *EXPR* evaluates to a true value (in boolean context). "`while`" loops may be introduced by an optional *LABEL*, which is just some identifier followed by a colon ":". Additionally, "`while`" loops may occur with an optional "`continue`" *BLOCK*, which is executed between loop iterations.

  You can use the keyword "`until`" in place of "`while`" as a shorthand for a negated loop condition:

  `until (`*EXPR*`)` *BLOCK*

  is equivalent to:

  `while (!(`*EXPR*`))` *BLOCK*

- **"for" Loops**

  `for (`*INIT*`;` *TEST*`;` *INCR*`)` *BLOCK*
  LABEL`: for (`*INIT*`;` *TEST*`;` *INCR*`)` *BLOCK*

C-style loop construct. First evaluates the initialization expression *INIT*. Then executes *BLOCK* as long as the conditional expression *TEST* evaluates to a true value. Executes the loop-incrementation expression *INCR* between loop iterations. Any of the expressions *INIT*, *TEST*, and/or *INCR* may be empty. The "`for`" loop is really just shorthand for:

INIT;
LABEL:
`while (`*TEST*`)` *BLOCK*
`continue {` *INCR*`; }`

- **"foreach" Loops**

  `foreach` *VAR* `(`*LIST*`)` *BLOCK*
  `foreach` *VAR* `(`*LIST*`)` *BLOCK* `continue` *BLOCK*
  LABEL`: foreach` *VAR* `(`*LIST*`)` *BLOCK*
  LABEL`: foreach` *VAR* `(`*LIST*`)` *BLOCK* `continue` *BLOCK*

  Executes *BLOCK* once for each element of the list value *LIST* with that value bound to the scalar loop variable *VAR*. If *VAR* is omitted, implicitly uses the default loop variable `$_`. *VAR* is always local to the loop – it regains its old value (if any) when the loop exits. As for "`while`" loops, "`foreach`" loops can take "`continue`" blocks, which are executed between loop iterations.

  You *can* use the "`for`" keyword to mean "`foreach`", but I don't recommend it.

- **Loop Modifiers**

  STATEMENT `while` *EXPR*`;`
  STATEMENT `until` *EXPR*`;`

  As for conditionals, simple statements may occur with a loop modifier just before the statement-terminating semicolon.

- **Loop Control**

  - **Loop Labelling**
    All loops may be labelled. By convention, label names are all uppercase. A loop-label identifies the loop as a whole, not just its entry point.
  - **"last": Loop Termination**

    `last`
    `last` *LABEL*

    Like the "`break`" statement in C, Perl's "`last`" command causes immediate termination of the innermost enclosing loop (first form), or of the loop labelled by *LABEL* (second form). "`continue`" blocks for the terminated loop are not evaluated.
  - **"next": Loop Continuation**

    `next`
    `next` *LABEL*

Like the "`continue`" statement in C, Perl's "`next`" command causes Perl to skip the rest of the current iteration and start with the next iteration. If the loop in question has a "`continue`" block, that block is evaluated before the loop re-evaluates its conditional. As for "**last**", the optional *LABEL* refers to the loop to be restarted, which defaults to the innermost enclosing loop.

– **"redo": Loop Repetition**

`redo`
`redo` *LABEL*

Causes Perl to restart the current loop iteration without evaluating the loop's "`continue`" block (if any) or conditional. *LABEL* names the loop to be restarted, default is the innermost enclosing loop.

- **Pseudo-Loops**

BLOCK
BLOCK `continue` *BLOCK*
LABEL: *BLOCK*
LABEL: *BLOCK* `continue` *BLOCK*

You can arbitrarily distribute blocks throughout your program, giving them labels and/or "`continue`" blocks, if you so desire. This allows you to use loop control primitives such as `next`, `last`, and `redo` to control program execution when even a "`while`" loop is too restrictive.

- **Looping Expressions**

`grep` *BLOCK LIST*
`map` *BLOCK LIST*

Looping expressions differ from loop statements mostly in the fact that looping expressions have a meaningful value, which loop statements such as "`while`" and "`for`" do not.[35]

The "`grep`" function evaluates *BLOCK* for each element of *LIST* in turn, locally binding the variable "`$_`" to each successive element, and returns a list of all and only those values from *LIST* for which (the last statement of) *BLOCK* returns a true value. Note that you can use the "`$_`" variable as an lvalue within a "`grep`" block to modify the original list values, but such tricks are not likely to improve the readability of your code, and may cause much wailing and gnashing of teeth in case your original list values were not themselves lvalues.

Like "`grep`", the "`map`" function evaluates *BLOCK* for each element of *LIST* in turn, locally binding the variable "`$_`" to each successive element. Also as for "`grep`", the "`$_`" variable may be used within *BLOCK* as an lvalue to modify the original contents of *LIST*.[36] Unlike "`grep`", "`map`" returns a list composed of the results of the evaluations of *BLOCK*.[37]

[35] The value of a loop statement such as "`while`" is usually the value of the last test condition evaluated, which is by definition a false value.

[36] Here too, use of $_ as an lvalue can be dangerous as well as cryptic.

[37] Note that you can produce non-monotonic results by returning list values of varying length from *BLOCK*. This is a Really Cool Thing.

Since "`map`" and "`grep`" (and, for that matter, "`sort`") return list values, you can chain them together to form truly monstrous-looking but remarkably efficient[38] expressions. Enjoy, and see the `perlfunc(1)` manpage for more details.

- **Iterators**

  `keys` *HASH*
  `values` *HASH*
  `each` *HASH*

  While these functions are not strictly speaking looping constructs (see Section 4.2.4 for details), Perl is able to process them especially efficiently when used within loops – used as the *LIST* value in a `foreach` loop, for example, the `keys()` function need not extract and allocate space for **all** the keys from *HASH* – it only needs to be able to enumerate those keys one at a time – one key for each iteration of the loop. This is pretty much exactly what Perl does when you say someting like:

  ```
  foreach $key (keys(%myhash)) {
    do_something_with($key);
  }
  ```

  ... which is a Good Thing, since that's usually the most intuitive way to express what you want to do, anyways.

- **Examples**

  ```
  ### Read lines from stdin
  while ($line = <STDIN>) {
    print "got line: $line";
  }


  ### As above with modifier
  print "got line: $line" while $line = <STDIN>;


  ### Iterate over a list literal
  foreach $day (qw(Sun Mon Tue Wed Thu Fri Sat)) {
    print "Today is $day.\n";
    sleep 60*60*24;
  }


  ### Array iteration, keeping track of indices
  @days = qw(Sun Mon Tue Wed Thu Fri Sat);
  for ($i = 0; $i <= $#days; $i++) {
    print "The ${i}th day of the week is $days[$i].\n";
  }
  ```

[38] Since the "`grep`" and "`map`" functions also fall into the category of "iterators" – see below.

```
### Infinite loop with label & explicit exit condition
SHAMPOO:
  while (1) {
    foreach $act (qw(wash rinse)) {
      perform_action($act);
      last SHAMPOO if (out_of_water());
    }
  }


### Filter out undefined values
@defined_things = grep { defined($_) } @things;


### Normalize all list elements to upper-case
@capital_stuff = map { uc($_) } @stuff;
```

4.3.3 Jumps

`goto` *LABEL*
`goto` *EXPR*
`goto &`*SUBNAME*

Perl supports control "jumping" with the "`goto`" command. Such commands are rarely (if ever) necessary, and generally do not help readability, re-usablity, maintainability, or efficiency. Try to avoid them. See the `perlsyn(1)` manpage if you need to know more about them.

4.3.4 Subroutines

- **Declaration**

  `sub` *NAME BLOCK*

  Declares a subroutine (read "function", "procedure", or "method") named *NAME* as a shorthand for the command-block *BLOCK* (also known as the "body" of the subroutine). You can later cause the commands in *BLOCK* to be executed by calling *NAME* as you would any builtin Perl command, such as `print`.

- **Parameters**
  Within a subroutine's body, the special array `@_` holds the actual parameter values with which that subroutine was called, thus:

  ```
  sub printus {
    print "my arguments were: ", @_, "\n";
  }
  printus(1,2,3);
  ```

prints:

```
my arguments were:  123
```

You can even use `@_` as an lvalue, and the changes will be propagated to the parameters in the calling context – providing those paremeters were lvalues, of course:

```
sub plural {
  $_[0] .= 's';
}
plural($word = 'cow');
print "word is now '$word'\n";
```

prints:

```
word is now 'cows'
```

- **Return Values**

  `return`
  `return` *EXPR*

  User-defined subroutines can specify how to compute their values by use of the "`return`" command, which causes the subroutine currently executing to terminate immediately and evaluate to the value of *EXPR* if specified, and otherwise to `undef`. Thus,

```
sub pow2 {
  return 2**$_[0];
}
print "2 to the 4th power is: ", pow2(4), "\n";
```

  prints:

```
2 to the 4th power is: 16
```

  If your subroutine doesn't explicitly `return` any value, it will return the value (if any) of the last expression in its body *BLOCK* that it evaluates. Note that you can return list values as well as scalar values from a subroutine – and even use `wantarray()` to determine what kind of value your caller expects.

- **Anonymous Subroutines**

  `sub` *BLOCK*

  Omitting the *NAME* part of a subroutine declaration causes an anonymous subroutine to be created and returned as a `CODE` reference (a scalar value). You can later call such a reference by using the "`&`" operator:

```
$mysub = sub { print "I'm anonymous.\n"; }
&$mysub();
```

  prints:

```
I'm anonymous.
```

It is always a good idea to use explicit parentheses when calling subroutines using the "&" operator, since omitting such parentheses causes the current value of `@_` to be passed. See the `perlsub(1)` manpage for details.

4.3.5 Declarators and Scope

- **Global Scope**
  By default, Perl variables are automatically created with *global scope* – this means they can be "seen" (their values can be referred to) from anywhere with your program. This can be handy, but often makes a program harder to maintain.

- **"my": Lexical Scope**

  `my (`*LIST*`)`

  The most common alternative to globally scoped variables are *lexically scoped* variables. A lexically scoped variable in Perl must be declared with the keyword "`my`". Lexically scoped variables (henceforth, "`my`" variables) cease to exist when the innermost block enclosing their declaration exits, and are "invisible" outside of that block, although they can be seen by sub-blocks. In particular, "`my`" variables in a subroutine are local[39] to a particular instance of that subroutine: each time you call the subroutine, a new set of "`my`" variables gets allocated.

```
### Wrong: $n gets overwritten
sub factorial_global {
  $n = shift(@_);                       # global scope
  return 1 if ($n < 1);
  return $n * factorial_global($n-1);   # oops!
}

### Right: "my $n" gets re-allocated
sub factorial_lexical {
  my $n = shift(@_);                    # lexical scope
  return 1 if ($n < 1);
  return $n * factorial_lexical($n-1);  # better
}
```

- **"local": Dynamic Scope**

  `local` *LIST*

  Dynamically scoped variables (henceforth, "`local`" variables) are like "`my`" variables, in that they cease to exist when the innermost block enclosing their declaration exits, but are like global variables in that while they exist, they can be "seen" from anywhere within your program. You can think of "`local`" variables as temporary global variables.

[39] In the conceptual sense, not in the sense that a variable declared with "`local`" is "local".

- **Closures**

  ```
  my VAR;
  $closure = sub { ... VAR ... };
  ```

  Closures are anonymous subroutines with a "dangling" reference to one or more lexically scoped variables. The nifty thing about closures is that as long as the closure (which is technically speaking a `CODE` reference) hangs around, the "`my`" variable of closure (above, "*VAR*") continues to exist. This is useful for doing tricky object-oriented things, implementing callbacks, or just to improve modularity (and sometimes even efficiency). Closures are Totally Cool Things commonly used in functional programming languages sub as LISP.

4.3.6 Errors and Warnings

- **Program Termination**

  `exit` *EXPR*

  Evaluating this statement causes the currently running Perl program to immediately terminate with an exit status of *EXPR* (which should have an integer value). The conventional exit status for program termination under normal conditions is 0 (zero).

- **Fatal Errors**

  `die` *LIST*

  Evaluating this statement causes the currently running Perl program to print the value of *LIST* to `STDERR` and then immediately terminate with an exit status indicating abnormal program termination. You can use the variable `$!` within *LIST* to refer to the most recent error message from the host operating system, if any.

- **Warnings**

  `warn` *LIST*

  If `perl` is running with the `-w` switch, evaluating this statement prints *LIST* to `STDERR`, much as `die()` does, but unlike `die()`, does not cause the running program to terminate.

- **Roll-Your-Own Error Handling**

  ```
  $SIG{__WARN__} = CODE;
  $SIG{__DIE__}  = CODE;
  ```

  You can specify your own warning- and error-handling routines by assigning them (as `CODE` references) as values to the `__WARN__` and/or `__DIE__` keys of the special global hash `%SIG`. Your handlers will be passed the error or warning message as their first argument if and when `warn()` or `die()`[40] are called.

[40] or `carp()` or `croak()` *etc.*

- **Examples**

```
### Produce a fatal error if 'myfile.txt' can't be opened
open(OUT,">myfile.txt")
  or die("open failed for 'myfile.txt': $!");

### Warn the user about an undefined variable
warn("Hey - you forgot to define '\$x'!\n")
  if (!defined($x));

### DIY Handlers: Step 1: define a handler
sub my_warn_handler {
  my $msg = $_[0];
  warn("my_warn_handler: $msg") if ($DOWARN);
}

### DIY Handlers: Step 2: install the handler
$SIG{__WARN__} = \&my_warn_handler;

### DIY Handlers: Alternative: use anonymous sub
$SIG{__DIE__}  = sub {
                   my_warn_handler(@_);
                   cleanup_temporary_files();
                   exit(1);
                 };
```

4.3.7 Dynamic Evaluation

`eval` *EXPR*
`eval` *BLOCK*

- **Expression Evaluation: Dynamic Code**
  The form "`eval` *EXPR*" causes *EXPR* (any expression, expected to return a string value) to be parsed and evaluated as Perl code in scalar context, and returns the result of evaluating this code. It is not terribly efficient, but can be quite handy if you need to generate program content dynamically at run-time.

- **Block Evaluation: Exception Trapping**
  The form "`eval` *BLOCK*" causes the contents *BLOCK* to be parsed as usual at compile-time, but blocks propagation of fatal errors out of the *BLOCK*. The return value of the `eval` statement is the value of the last statement evaluated in *BLOCK*.

- **eval and Errors**
  If any errors occur while parsing or evaluating an `eval` statement, `eval` returns `undef`, and sets the global special variable `$@` to the error message. If no error occurred, then `$@` is guaranteed to be an empty string.

- **Examples**

```
### Dynamic variable naming (dangerous!)
$xref = "\$x";
eval "$xref = 42;";

### Make divide-by-zero errors non-fatal
eval { $answer = $a / $b; };
warn $@ if $@;
```

4.3.8 External Code

- **External Scripts**

  `do` *EXPR*

  Interprets the expression *EXPR* as the name of a file and attempts to evaluate the contents of that file as Perl code. Returns a true value on success, otherwise `undef`.

  If the file cannot be read, the variable `$!` will be set to an appropriate error message. Otherwise, if the file can be read but not compiled, then the variable `$@` will be set to an appropriate error message. Otherwise, if the file is successfully read and compiled, then the `do` statement returns the value of the last expression evaluated in the file.

  Typical uses of `do` include reading in libraries of Perl subroutines, or reading in program-specific configuration files. These days, it is more common to write full-blown Perl modules and include these with either the `use` pragma or a `require` statement.

- **External Modules**
  Since we haven't encountered packages yet, I cannot adequately explain exactly what a Perl Module is ... for now, you can think of a module as a "black box" full of various algorithmic goodies provided by those nice folks on CPAN for your fun and enjoyment. Most modules come with their own documentation, which you can read with the `perldoc` program.

  - *Compile-Time Inclusion*

    `use` *Module*
    `use` *Module LIST*

    Attempts to load the module *Module* (a bareword) at compile time by searching the root directories of the default module include path `@INC`, replacing any `::` in *Module* with a directory separator, and looking for the file *Module*`.pm`. Causes a fatal error if *Module* cannot be found, or if the last expression evaluated in *Module*`.pm` does not return a true value.

    If *LIST* is specified, it is (usually) interpreted as a list of variable-, subroutine-, and or "tag"-names which should be imported from *Module* into the calling namespace. See the `perlmod(1)` manpage for details.

  - *Runtime Inclusion*

`require` *EXPR*

Causes the module named by evaluating the expression *EXPR* to be loaded (by searching for it in the default module search path) if it is not loaded already. Causes a fatal runtime error if the module cannot be found, or if evaluating the module file fails to return a true value. Unlike `use`, the `require` statement gets evaluated at run-time, and does not perform any implicit import of symbols from the module loaded.

- **Version Checking**

  `use` *Module VERSION LIST*
  `use` *Module VERSION*
  `use` *VERSION*
  `require` *VERSION*

  If you specify a *VERSION* (usually a funny-looking numeric literal) to the "`use`" statement, the Perl interpreter will attempt to ensure that the version of *Module* loaded is at least *VERSION*.

  If you omit a *Module* and simply "`use` *VERSION*" or "`require` *VERSION*", then *VERSION* will be interpreted as the minimum version of Perl necessary to run your script, and will generate a fatal error if the interpreter's own version number is less than *VERSION*.

  See the entries for "require EXPR" and "use Module" in the `perlfunc(1)` manpage, as well as the `perlmod(1)` manpage for details.

4.4 Perl I/O

4.4.1 Filehandles

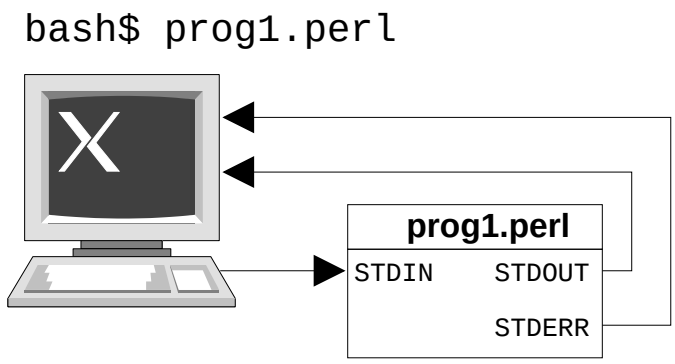
A Perl *Filehandle* (or simply *Handle*) is just a Perl syntactic element for representing a *stream* to or from which data may flow.[41] Perl filehandles behave like an additional elementary datatype (like scalars, arrays, and hashes) which are not prefixed by any funny character – thus, "pure" filehandle names always appear as barewords.

- **Standard Filehandles**
  Perl defines the following three standard streams for each process:

C Name	Perl Name	Description
stdin	STDIN	Standard input (keyboard)
stdout	STDOUT	Standard output (screen)
stderr	STDERR	Standard error-output (screen)

  The "normal" configuration of the standard streams for a simple program call is shown in Figure 1.

Figure 1: Program call without redirection



- **Typical Filehandle Operations**
  - **Opening**

    `open` *HANDLE*, *MODE*, *NAME*
    `open` *HANDLE*, *EXPR*

    Opens the filehandle *HANDLE* for I/O in mode *MODE* to and/or from *NAME*, which is evaluated as a string. Exactly what this string should contain depends on what you are trying to open – see Sections 4.4.2 and 4.4.3 below, the "open" entry in the `perlfunc(1)` manpage, and also the `perlopentut(1)` manpage.
  - **Reading Text**

[41] See Section 3.5 for a brief introduction to streams and handles.

`<`*HANDLE*`>`
`<>`

The `<`*HANDLE*`>` operator reads and returns a single line of input from the filehandle *HANDLE* in scalar context. In list context, it reads and returns a list of all remaining lines from *HANDLE*, which in any case should be opened for reading. On end-of-file, `<`*HANDLE*`>` returns `undef`.

As a special case, you can omit the *HANDLE* argument on the line-input operator, which causes input to be read from the files named by your script's command-line arguments[42], or from `STDIN` if no arguments were given.

– **Writing Text**

`print` *HANDLE LIST*
`print` *LIST*

Prints the elements of *LIST* to *HANDLE* in the order specified. *HANDLE* should be a filehandle opened for writing. If *HANDLE* is omitted, prints to *STDOUT*.

– **Writing Text, C-Style**

`printf` *HANDLE FORMAT*`,` *LIST*
`printf` *FORMAT*`,` *LIST*

C-style `printf()` function. If *HANDLE* is omitted, prints to *STDOUT*. You will never need this function (because Perl has `sprintf()`, too). When you want it, it is very nice to have it available. See the `sprintf(3)` manpage or the `printf(3)` manpage for details on the `printf()` available on your system, and see the "sprintf()" entry in the `perlfunc(1)` manpage for details on Perl's builtin `sprintf()` function.

– **Setting Binary Mode**

`binmode` *HANDLE*

On silly systems that distinguish between "binary" and "text" files[43], you will need to call `binmode()` on any handles which may contain non-text data before performing any read or write operations on those handles.

– **Reading Binary Data**

`read` *HANDLE*`,` *SCALAR*`,` *LENGTH*

Attempt to read *LENGTH* bytes of data from *HANDLE* into the variable *SCALAR*. Returns the number of bytes actually read, 0 (zero) at end-of-file, or `undef` if there was an error.

You might also just want to set the `$/` variable to your favorite record separator and use the line-input operator "`<`*HANDLE*`>`".

– **Writing Binary Data**

`print` *HANDLE LIST*
`print` *LIST*

[42] Remember the `@ARGV` array?
[43] Such as DOS and Windoof.

Perl will happily `print()` binary data – for Perl, such data can be easily encoded as a funny-looking string. See the entry for "pack" in the `perlfunc(1)` manpage for a good way to create such strings, and check out the `Storable` module if you want to save Perl data structures. If you just want a persistent hash or array, check out the `DB_File` , `GDBM_File` , `NDBM_File` , `ODBM_File` , `SDBM_File` , and/or `AnyDBM_File` modules.

– **Setting Autoflush**

`require IO::Handle;`
HANDLE`->autoflush();`

Causes output to *HANDLE* to be unbuffered: that is, anything written to *HANDLE* will be written immediately. Useful for pipes and other IPC related handles.

– **Closing**

`close` *HANDLE*

Closes the handle *HANDLE*. Returns true on success, false if there was an error.

It is good practice to close your handles when you are done with them. Any opened handles are automatically closed when your program terminates, though.

- **Examples**

```
### Open a filehandle
open(FH, $myfile)
  or die("open failed for '$myfile': $!");

### Read from command-line files or STDIN
while ($line = <>) {
  do_something($line);

### Write to a file
print FH "Hello, handle!\n";

### Set binary mode
binmode(BINFH);

### Read up to 42 bytes of binary data from BINFH
$nbytes = read(BINFH, $data, 42);

### Write binary data to BINFH
print BINFH pack('i*', 420, 24, 7);
```

4.4.2 Files

- **Shell Redirection**

`bash$` *PROGRAM* `<` *FILE*
`bash$` *PROGRAM* `>` *FILE*

Most command-line shells enable the user to redirect the standard streams to/from file(s) (known as "file redirection").[44] Although shell redirection is not strictly part of Perl, many Perl programs are written to be used as "filters": programs which read data from STDIN and write (modified) data to STDOUT, so that they may be used within a shell pipeline. Some examples of shell redirection of the standard streams are given in Figures 2 and 3.

Figure 2: Input redirection from file

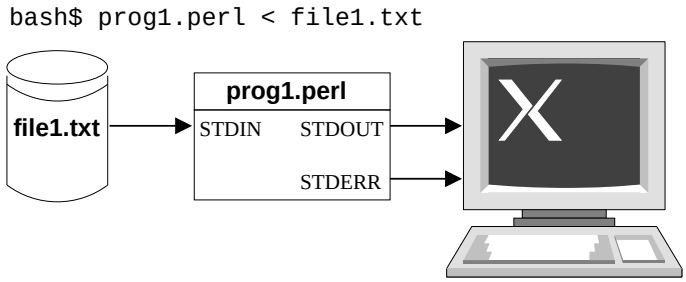
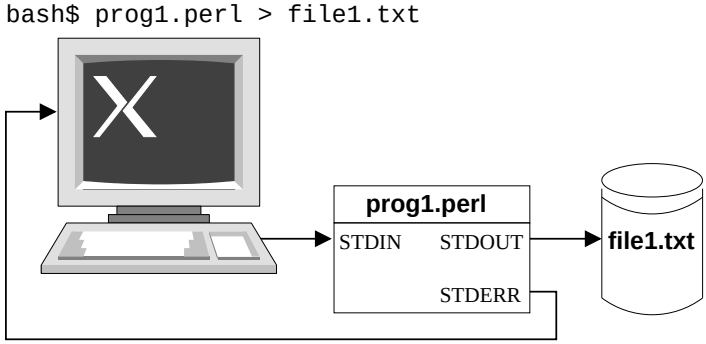


Figure 3: Output redirection to file



- **Opening**

  `open` *HANDLE*, *MODE*, *FILENAME*
  `open` *HANDLE*, *FILESPEC*

  Opens the handle *HANDLE* for I/O in mode *MODE* to/from the file named *FILENAME*. The *MODE* argument specifies how the file should be opened: for reading, writing, or appending. In the two-argument form, the *FILESPEC* should be a string resulting from concatenating a *MODE* string (the "mode-prefix") and a *FILENAME*.

[44] Note that some non-POSIX operating systems (such as DOS and its ilk) stubbornly refuse to redirect "binary" data.

Mode	Description
`<`	Read only
`>`	Write only, overwrites old data
`>>`	Write only, append
`+<`	Read and write (dangerous!)
(none)	Input only, like "`<`"

- **Testing**

  `-`*X* *HANDLE*
  `-`*X* *EXPR*

  Perl has some handy builtin shell-like filetest operators (above abstracted as "*X*") to test the status of files based on either expressions evaluating to their filenames or on opened handles.

Operator	Description
`-r`	File is effectively readable
`-w`	File is effectively writable
`-x`	File is effectively executable
`-e`	File exists
`-z`	File has zero size (is empty)
`-s`	File has nonzero size (returns size in bytes)
`-f`	File is a plain file
`-d`	File is a directory
`-t`	File is a tty (terminal)
`-T`	File is an ASCII text file
`-B`	File is a "binary" file

  See the entry for "-X" in the `perlfunc(1)` manpage for more.

- **Pseudo-Filenames**
  The pseudo-filename "-" (minus) is special when used in a call to `open()`. If used as an input filename, it refers to STDIN. If used as an output filename, it refers to STDOUT. This jives well with the command-line syntax of many common UNIX utility programs, such as `cat(1)` or `grep(1)`.

- **Examples**

```
### Open 'file1.txt' for reading
open(FILE1, "<", "file1.txt")
  or die("open failed for 'file1.txt': $!");

### Same thing, but shorter
open(FILE1, "<file1.txt")
  or die("open failed for 'file1.txt': $!");

### Same thing, even shorter
open(FILE1, "file1.txt")
  or die("open failed for 'file1.txt': $!");
```

```
### Open "output.txt" for writing, overwriting old data
open(OUT, ">output.txt")
  or die("open failed for 'output.txt': $!");

### Open "output.txt" for writing, appending to old data
open(OUT, ">>output.txt")
  or die("open failed for 'output.txt': $!");

### See if 'whosit.txt' already exists
die("where is 'whosit.txt'?\n") if (! -e 'whosit.txt');

### Check whether we are in a pipeline
print("We're not in a pipeline")
  if (-t STDIN && -t STDOUT);
```

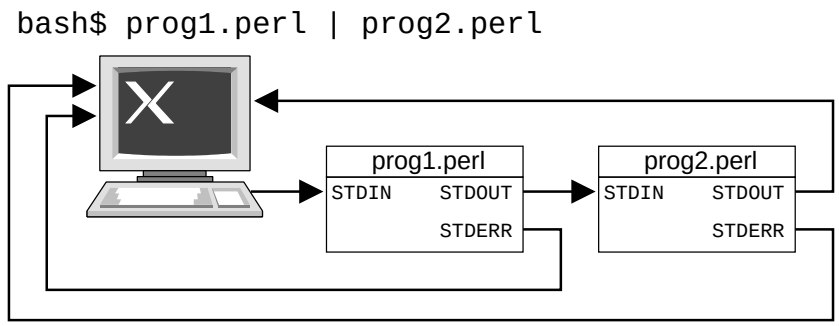
4.4.3 Pipes

- **Shell Pipelines**

  `bash$` *PROGRAM1* `|` *PROGRAM2*

  Most command-line shells enable the user to redirect the standard streams to/from other programs (known as a "pipeline").[45] A graphical example of shell pipelining is given in Figure 4.

Figure 4: A simple pipeline



- **Opening Pipes**

  `open` *HANDLE*, *MODE*, *COMMAND*
  `open` *HANDLE*, *CMDSPEC*

  Opens the handle *HANDLE* for I/O in mode *MODE* to/from the command *COMMAND* (possibly with arguments). The *MODE* argument specifies whether a pipe should be opened to (`|-`) or from (`-|`) *COMMAND*. In

[45] Again, some non-POSIX operating systems (such as DOS and its ilk) inexplicably refuse to redirect "binary" data.

the two-argument form, the *CMDSPEC* should be a string resulting from replacing the "-" (minus) character in the plain *MODE* value with the string *COMMAND*.

Mode	Description
`|-`	Write to *COMMAND*'s `STDIN`
`-|`	Read from *COMMAND*'s `STDOUT`

You cannot open a single pipe for both reading and writing, but see the `IPC::Open2` module for a workaround.

- **Examples**

```
### Open a pipe from 'perldoc -u perltoc'
open(PERLTOC, "-|", "perldoc -u perltoc")
  or die("open failed for pipe from 'perldoc': $!");

### Open a pipe from 'perldoc -u perltoc', shorter
open(PERLTOC, "perldoc -u perltoc |")
  or die("open failed for pipe from 'perldoc': $!");

### Read from a pipe (like any other handle)
@perltoc = <PERLTOC>;

### Open a pipe to 'gfsmcompile'
open(FSMCOMP, "|-", "gfsmcompile -F myfsm.gfst")
  or die("open failed for pipe to 'gfsmcompile': $!");

### Open a pipe to 'gfsmcompile', shorter
open(FSMCOMP, "|gfsmcompile -F myfsm.gfst")
  or die("open failed for pipe to 'gfsmcompile': $!");

### Write to a pipe (like any other handle)
print FSMCOMP "$qfrom $qto $labin $labout\n";
```

4.4.4 IO::File

The `IO::File` module provides an object-oriented interface to many types of Perl filehandles, including files and pipes. The really nifty thing about `IO::File` is that unlike "pure" filehandles, and `IO::File` object can be stored in a scalar variable, so you can pass them to subroutines as parameters, store them in arrays or hashes, etc.

See the `IO::Handle(3pm)` module manpage , the `IO::File(3pm)` module manpage , and the `IO::Pipe(3pm)` module manpage for details.

- **Prerequisites**

```
use IO::File;
```

You should `use` or `require` the IO::File module before trying to call any of the methods listed below.

- **Opening Files**

  `IO::File->new(`*SPEC*`)`

  The `new()` method creates, opens, and returns a new `IO::File` object (a scalar value) for I/O to/from *SPEC* which should be an expression as used by the two-argument form of the builtin `open()` function. Returns `undef` on failure.

- **Line Input**

  ```
  $line = <$fh>;
  @lines = <$fh>;
  ```

  You can use the built-in line input operator on an IO::File object, too.

- **Setting Binary Mode**

  ```
  $fh->binmode();
  ```

  The `binmode()` method works just like the built-in function of the same name.

- **Reading Binary Data**

  `$fh->read(`*SCALAR*`, `*LENGTH*`, `*OFFSET*`)`

  The `read()` method works just like the built-in function of the same name.

- **Writing Data**

  `$fh->print(`*LIST*`)`
  `print $fh `*LIST*

  The `print()` method works just like the built-in function of the same name.

  You can use the built-in `print()` function on an IO::File object, too.

- **Closing**

  ```
  $fh->close()
  ```

  The `print()` method works just like the built-in function of the same name.

  You can use the built-in `close()` function on an IO::File object, too.

- **Examples**

  ```
  ### Open the file 'input.txt' for reading
  $infh = IO::File->new("<input.txt")
    or die("open failed for 'input.txt': $!");

  ### Read a line from the input IO::File
  $line = <$infh>;
  ```

```
### Open the file 'output.txt' for writing
$outfh = IO::File->new(">output.txt")
  or die("open failed for 'output.txt': $!");

### Write to an output IO::File
$outfh->print("Hello, IO::File!\n");
```

4.4.5 Sockets

Sockets are just generalized interprocess communication (IPC) channels. This Section is about internet sockets, also known as "INET sockets", which are a type of socket designed for passing data between two computers on a network. To use INET sockets, one machine (the "server") must `listen()` on a given *port*, and the other machine (the "client") must `connect()` to that port on the server – usually, the client must "know" the server's IP address for this to work.

INET sockets are additionally characterized by their *protocol*, the most typical of which are TCP[46] and UDP[47], as well as by the *style* (or *type*) of communication they allow.

Protocol	Perl Name	Typical Use
TCP	`"tcp"`	Lossless data exchange
UDP	`"udp"`	Lossy data exchange

Type	C/Perl Constant	Typical Use
stream	`SOCK_STREAM`	Lossless data exchange
datagram	`SOCK_DGRAM`	Lossy data exhchange

The parallels between the common *protocols* and *types* lead to the two most common combinations:

Protocol	Type	Typical Use
TCP	stream	Lossless data exchange
UDP	datagram	Lossy data exchange

All the socket flavors described here are bidirectional: that is, you can use them for both reading and writing, but be warned that your program needs to figure out for itself when and how much it can read, otherwise it may well hang ("block") forever!

- **Prerequisite: the IO::Socket::INET module**

  ```
  use IO::Socket::INET;
  ```

  You can do everything in the rest of this section without this module, but it's a lot uglier without it.

- **Creating a Client Socket**

  ```
  $c_sock = IO::Socket::INET->new(
      PeerAddr => $server,   # IP address or hostname
  ```

[46] Transmission Control Protocol
[47] User Datagram Protocol

```
    PeerPort => $port,     # number or service name
    Proto    => $protocol, # protocol number or name
    Type     => $type      # socket type constant
  );
```

Creates and returns a new client socket for communicating with port `$port` on the server `$server` using protocol `$protocol` with type `$type`. Returns undef on failure.

You can use the client socket `$c_sock` pretty much like any other filehandle.

- **Creating a Server Socket**

```
$s_sock = IO::Socket::INET->new(
    LocalAddr  => $server,   # IP or hostname (optional)
    LocalPort  => $port,     # number or service name
    Proto      => $protocol, # protocol number or name
    Type       => $type,     # socket type constant
    Listen     => $qsize,    # client queue length
    ReuseAddr  => $bool,     # reuse socket address?
    Timeout    => $seconds   # timeout (optional)
  );
```

Creates and returns a new server socket for listening on the port `$port` of the local machine `$server` using protocol `$protocol` with type `$type`. Returns undef on failure.

The "Listen" argument specifies the maximum number of clients which should be allowed to queue for the port on the server before the server refuses new incoming connections.

The "ReuseAddr" argument specifies whether the local address for the socket should be re-used. Some high-level internet protocols (such as FTP) require this option to bet set to a true value – it's usually a good idea.

The "Timeout" argument specifies the timeout in seconds for various socket operations (such as `accept()`) – leaving it undefined or specifying 0 (zero) as the timeout will cause such operations to wait forever (or until your program receives a signal for which a handler is defined).

If you do not specify the optional "LocalAddr" argument, then the socket constant `INADDR_ANY` will be used, which will bind any valid IP address your machine may have.

- **Accepting Client Connections**

```
$c_sock = $s_sock->accept();
```

The `accept()` method causes a server socket `$s_sock` to wait (until its "Timeout" expires) for incoming client connections, returning a new IO::-Socket::INET object for the client when such a connection occurs.

You can use the newly returned client socket `$c_sock` filehandle.

- **Reading, Writing, etc.**
  Like `IO::File`, the `IO::Socket` class inherits from the abstract class `IO::Handle`, so you can use the standard methods such as `print()`, `read()`, `close()`, as well as the line-input operator on connected (client) sockets. See the `IO::Handle(3pm)` module manpage , the `IO::Socket(3pm)` module manpage , and the `IO::Socket::INET(3pm)` module manpage for details.

- **Client Example: Get time of day**

```
### Connect to port for service 'daytime' on 'localhost'
$dsock = IO::Socket::INET->new(
           PeerAddr => 'localhost',
           PeerPort => 'daytime',
           Proto    => 'tcp',
           Type     => SOCK_STREAM,
         )
  or die("could not open socket: $!");

### read time of day
$daytime = <$dsock>;

### and disconnect
$dsock->close();
```

- **Server Example: Provide time of day**

```
### Set up a server socket on port 42024
$sock = IO::Socket::INET->new(
           LocalPort => 42024,
           Listen    => 5,
           Proto     => 'tcp',
           Type      => SOCK_STREAM,
           ReuseAddr => 1,
         )
  or die("could not create socket: $!");

### Listen for incoming connections
###   + NOTE: you will need to explicitly kill
###     this program yourself!
while ($client = $sock->accept()) {
  chomp($date = `date`);           # call the 'date' program
  $client->print("$date\r\n");     # ... inform the client
  $client->close();                # ... make them go away
}
```

4.5 Perl Regular Expressions

4.5.1 Friends and Relations

Regular expressions are known to formal linguists as a notational variant of *finite state automata*. UNIX hackers may be familiar with their implementations in such common utilities as `grep`, `egrep`, `ed`, `sed`, `awk`, `vi`, or `emacs`. Perl's regular expressions are very similar to those of the common UNIX utilities in their syntax and semantics, the conventions for which differ slightly from those commonly used by formal linguists. Still, it should be remembered that a regular expression is just syntactic sugar for a finite state machine – this is generally a Good Thing, since finite state machines are notoriously fast.

See the `perlre(1)` manpage and the "Regexp Quote-Like Operators" section in the `perlop(1)` manpage for the full story.

4.5.2 Common Uses

Those readers familiar with finite state automata will be unsurprised that the elementary operation associated with regular expressions is that of *recognition*, more commonly known to UNIX and Perl hackers as *pattern matching*, where the regular expression being "matched" constitutes the "pattern" in question. I will adopt this terminology here, reserving the term "expression" for any Perl expression as described in Section 4.1.3. The basic Perl schema for pattern matching is:[48]

STRING `=~` */PATTERN/*

... where *STRING* is some string expression, and *PATTERN* is a Perl regular expression. The slashes (/) surrounding *PATTERN* are literals. Such a matching operation returns a true value if and only if *STRING* "matches" *PATTERN* – in formal terms, iff *STRING* belongs to the language defined by *PATTERN*.[49]

PATTERN itself may contain any text, and interpolates Perl variables in string context – for other syntactic and semantic oddities, see below. A common case is:

```
if ($name =~ /moo/) {          # ... ok, you found me
   do_something($name);
}
```

... which matches any string `$name` which contains some occurrence of the substring "moo".

Another common use of Perl regular expressions is the "substitution" pattern operator `s///`, which performs what are known to linguists as regular *transductions*: mappings from input strings to output strings:

[48] Actually, you can omit *STRING* and the matching operator "`=~`" altogether, in which case the match is performed against the contents of the default variable `$_`.

[49] Exactly what this value is depends on sub-patterns, context, and other niceties – see the `perlop(1)` manpage for details.

STRING =~ s/*SEARCH*/*REPLACEMENT*/

This causes the first occurrence of a substring which matches *SEARCH* in *STRING* to be replaced by *REPLACEMENT*. Of course, *STRING* must be an lvalue in order for this to work.

```
$str = 'moocow';
$str =~ s/moo/cow/;      # $str is now 'cowcow'
```

4.5.3 Single-Character Patterns

This Section describes the major regular expression pattern elements for matching single characters in the input string.

- **Literal Characters**
  Almost any literal character can be included as-is in a regex pattern. Literal characters match themselves.

  ```
  /abc/    # match substring abc
  /a b c/  # match substring 'a b c' (with spaces)
  ```

- **Wildcard Character**
  The special "wildcard" character "." (dot) matches any single character in the input string which is not a newline:

  ```
  /a.c/    # match aac, abc, acc, adc, ...
  ```

- **Escaped Literal Characters**
  Characters which have a special meaning within regex patterns can be "escaped" (treated as literal characters) by preceeding them with a backslash:

  ```
  /\/home\/moocow/  # matches /home/moocow
  /c:\\dos\\run/    # matches c:\dos\run
  /www\.cpan\.org/  # matches www.cpan.org
  ```

- **Simple Character Classes**
  "Character classes" are common in the UNIX world, but less frequently used by formal linguists, who like to reduce them to large unwieldy disjunctions. A "character class" is just a set of characters specified in the pattern, the presence of any one of which in the input string counts as a match. Character classes are enclosed in square brackets:

  ```
  /[Aa]b[Cc]/    # matches AbC, Abc, abC, or abc
  /[0123456789]/ # matches any digit
  ```

- **Ranged Character Classes**
  Character classes can also be specified by "ranges"[50], by specifying the endpoints of the range separated by a "-" (minus):

[50] Note that exactly what any given range contains depends on which character set you happen to be using. Generally, you can consider yourself on safe ground with 7-bit ASCII, but beyond that, you're on your own.

```
/[0-9]/    # matches any digit
/[A-Za-z]/ # matches any ASCII letter
```

- **Negated Character Classes**
  Character classes may be "negated" – that is, you may specify a character class which matches any character in the input string which is *not* a member of the character class, by specifying a carat "^" as the first character of the class:

```
/[^z]/    # matches any character except z
/[^0-9]/  # matches any non-digit
```

- **Predefined Character Classes**
  Perl has several very useful predefined character classes, which you may use either directly within your patterns, or include them within other character classes:

Construct	Equivalent Class	Description
\d	[0-9]	Digit character
\w	[A-Za-z0-9_]	Word character
\s	[\r\t\n\f]	Whitespace character
\D	[^0-9]	Non-digit character
\W	[^A-Za-z0-9_]	Non-word character
\S	[^ \r\t\n\f]	Non-whitespace character

4.5.4 Multi-Character Patterns

This Section describes the major regular expression pattern elements for matching multiple characters in the input string.

- **Sequence** (*Concatenation*)
  Sequences of regular expression pattern elements match corresponding substring sequences in the input string. Many of the examples above implicitly made use of sequenced patterns to match whole substrings by specifying sequences of characters. This is a handy thing, but it gets even cooler when you realize that any two regex patterns may be concatenated.

```
/abc/ # match a followed by b followed by c
```

- **Optionality: ?**
  The "?" (question mark) pattern operator matches zero or one occurrences in the input string of the pattern element immediately preceeding it.

```
/a?b/ # match b or ab
```

- **Multipliers: *** (*Transitive and Reflexive Closure*)
  The "*" (asterisk) pattern operator matches zero or more occurrences in the input string of the pattern element immediately preceeding it.

```
/a*b/         # match b, ab, aab, aaab, aaaab, ...
/[a-zA-Z]\w*/ # match a perl identifier
```

- **Multipliers: +** (*Transitive Closure*)
  The "+" (plus) pattern operator matches one or more occurrences in the input string of the pattern element immediately preceeding it.

```
/a+b/           # match ab, aab, aaab, aaaab, ...
/0x[\da-fA-F]+/ # match C-style hex literals
```

- **General Multiplier** (*N-ary Closure*)
  The "$\{m,n\}$" pattern operator matches at least m but not more than n occurrences in the input string of the pattern element immediately preceeding it. You may omit n, in which case no upper limit is imposed (as for the "*" and "+" pattern operators). You may also omit the comma, which is equivalent to $\{m,m\}$.

```
/a{2,4}/        # match aa, aaa, or aaaa
/a{2,}/         # match aa, aaa, aaaa, aaaaa, ...
/a{2}/          # match aa
```

- **Alternation** (*Union, Disjunction*)
  Two regular expression pattern elements separated by the alternation pattern operator "|" (vertical bar) match any string which matches one or both of the two pattern elements:

```
/a|b/  # match a or b
/a+|b/ # match a, aa, aaa, ..., or b
```

  For disjunctions of single characters, it is usually easier and more readable to use character classes.

- **Anchors**
  Perl regular expressions may contain a number of so-called "zero-width anchors", which can be understood as matching points in the input string which lie *between* individual characters. Some common anchors are listed below.

Anchor	Description
^	Beginning of string or post-newline
$	End of string or pre-newline
\b	Word boundary
\B	Non-boundary
(?=*PATTERN*)	Positive lookahead assertion
(?!*PATTERN*)	Negative lookahead assertion

```
/^abc/       # match abc only at beginning of string
/def$/       # match def only at end of string

/\bmoo\b/    # match whole word moo
/\bmoo\B/    # match moocow, moose, ..., but NOT moo

/moo(?=cow)/ # match moo if followed by cow
/moo(?!cow)/ # match moo if not followed by cow
```

4.5.5 Grouping Patterns

It is often necessary to group subpatterns within a regular expression, since the builtin precedence rules for the pattern operators (see below) don't always do what you want. Perl's pattern grouping operators allow you to do much more than just disambiguate operator scope conflicts, however ...

- **Grouping: ()**
  Any subpattern may be enclosed in round parentheses[51] "()", which take precedence over all other pattern operators.

  /... (SUBPATTERN) .../

  This allows you to write expressions such as the following:

```
/ab+/           # matches ab, abb, abbb, ...
/(ab)+/         # matches ab, abab, ababab, ...

/foo|bazbar/    # matches foo or bazbar
/(foo|baz)bar/  # matches foobar or bazbar
```

- **Subpattern Memory: () and $\backslash N$**
  The really cool part about subpattern grouping with round parentheses is that Perl "remembers" the part of the input string which matched the parenthesized subpattern, so you can refer to it later:

  /... (SUBPATTERN) ... \N .../

  where N is the number of the parenthesized *SUBPATTERN* (starting from 1) to which you wish to refer.

```
/(foo)bar\1/  # matches foobarfoo
/a(.)b\1/     # matches aXbX, aYbY, ..., not aXbY
```

- **Subpattern Memory and Substitution: s///, (), and $\backslash N$**
  Parenthesized subpattern memory is extremely useful in "substitution" (`s///`) regex patterns, especially in the *REPLACEMENT* part:

```
$str = 'cows say moo';
$str =~ s/(\w+) say (\w+)/\2 is said by \1/;
print $str, "\n";
```

  prints:

```
moo is said by cows
```

- **Grouping Only: (?:)**
  If you aren't interested in remembering what part of the input string matched, and just want to group your subpatterns, you can use a `(?:)` construct instead of plain round parentheses:

  /... (?:SUBPATTERN) .../

[51] Note that this is a major difference between Perl's regular expressions and those found in other UNIX utilities, which typically use backslashed parentheses "\(\)" for this purpose.

It is usually a good idea to use `(?:)` when you don't need to "remember" the matched substring, simply because `(?:)` constructs are a bit more efficient.

4.5.6 Matching Miscellany

- **Greedy** *vs.* **Lazy Matching**
  By default, Perl's multiplier regular expressions are "greedy" – that is, a pattern such as `/a*/` will match as many "`a`" characters in the input string as it can. Any pattern multiplier (`?`, `+`, `*`, or `{`m, n`}`) may be forced to be non-greedy (or *lazy*) by following it with a question mark (`?`). This can be useful for efficiency reasons (failed attempts at greedy matching cause Perl's matching engine to backtrack), or just to control more closely how your pattern is matched.

- **Pattern Operator Precedence**
  The following table lists the various pattern-internal operators in order of precedence (highest to lowest):

Name	Representation
Parentheses	`( )` `(?: )`
Multipliers	`? + * {m,n} ?? +? *? {m,n}?`
Sequences and Anchors	`abc ^ $ (?= ) (?! )`
Alternation	`|`

- **Variable Interpolation**
  You can interpolate the string-values of scalar variables into your patterns just as you would into double-quoted strings. This even works if you have special regular expression meta-characters (such as "`.`", "`*`", or "`|`") in your variables – they are interpreted as usual within the pattern:

```
$pattern = 'moo|cow';
foreach $s (qw(cow moose bovine)) {
  if ($s =~ /$pattern/) {               # Interpolate!
    print "/$pattern/ matched '$s' \n";
  }
  else {
    print "/$pattern/ did NOT match '$s'\n";
  }
}
```

prints:

```
/moo|cow/ matched 'cow'
/moo|cow/ matched 'moose'
/moo|cow/ did NOT match 'bovine'
```

For efficiency reasons, it is usually best **not** to use variable interpolation in regular expression patterns, since interpolation causes Perl to re-compile

the regular expression every time it is matched (in case the interpolated variable's value has changed).[52]

- **Quoting Escapes:** `\Q...\E`
  Sometimes a string is just a string – sometimes you don't want special regular expression pattern operators to have a special meaning. In these cases, you can always "escape" (or "quote") them with a backlash, as mentioned above. This can get ugly, and if the special characters happen to be in the value of a variable that gets interpolated into the pattern, things can get quite ugly indeed. Perl provides the `\Q \E` ("quoting escape") construct for just such cases:

```
$cpan='cpan.org'; # just a string variable

/cpan.org/        # matches cpan.org, cpanAorg, ...
/$cpan/           # matches cpan.org, cpanAorg, ...

/cpan\.org/       # matches cpan.org only
/\Qcpan.org\E/    # matches cpan.org only
/\Q$cpan\E/       # matches cpan.org only
```

- **Alternative Delimiters**
  Just as you can choose your own string delimiters with `q{}` and `qq{}`, you can choose a regular expression delimiter other than slash (/) by using the "matching operator" `m{}` for simple matches. As for `q{}` and `qq{}`, the `m{}` operator respects "natural pairs" of delimiters. Similary, you can choose your own delimiters for substitution operations by simply using them with the `s{}{}` operator.

```
/\/home\/moocow/   # my home directory
m/\/home\/moocow/  # ... with 'm' operator
m#/home/moocow#    # ... using # for delimiters
m(/home/moocow)    # ... using () for delimiters

s/foo/bar/         # substitute 'bar' for 'foo'
s#foo#bar#         # ... using # for delimiters
s(foo)(bar)        # ... using () for delimiters
```

- **Special Variables**
  There are a number of special read-only match-related variables which you can refer to after a successful match. Most useful are the special variables `$1`, `$2`, `$3` and so on, which refer to the same strings as `\1`, `\2`, `\3` and so on refer to within the matched pattern. Also useful are:

Variable	Description
`$&`	Matched substring
`$‘`	Unmatched prefix ("pre-match")
`$’`	Unmatched suffix ("post-match")

[52] You can get some explicit control over re-compilation with the `qr//` operator – see the `perlop(1)` manpage for details.

- **Pattern Modifiers**
  Perl provides a few handy modifiers for whole patterns. One or more modifiers may follow the closing delimiter of an `m//` or `s///` pattern.

Modifier	Description
`g`	Match globally: find all occurrences
`i`	Do case-insensitive matching
`m`	Treat string as multiple lines
`o`	Only compile pattern once
`s`	Treat string as single line
`x`	Ignore whitespace and comments within pattern

```
/abc/i       # matches abc, abC, aBc, ..., ABC
s/foo/bar/g  # replace all 'foo's, not just the first
/$regex/o    # assume value of $regex is constant
/^a\nb$/m    # embedded newline ok, $ matches EOS
```

4.6 Perl References

4.6.1 What is a Reference?

Perl's "references" are similar to "pointers" as found in C or PASCAL. PROLOG implicitly incorporates the notion of reference in the unification of uninstantiated variables. Intuitively, a reference *refers* to some Perl datum – be it a scalar, array, hash, or subroutine. A reference may refer to a local or global variable, a constant, or (in the case of "hard" references) even to an "anonymous" Perl datum – one for which the reference is the only access.

References themselves however are always scalar values – the act of recovering the datum referred to (or "pointed at") by a reference is known as *dereferencing*.

A tutorial on Perl references can be found in the `perlreftut(1)` manpage, and details are available in the `perlref(1)` manpage.

4.6.2 Why References?

- **Shared Data**
  References allow multiple scalars, arrays, and/or hashes to refer to the same underlying datum – thus, changing the underlying value for one instance causes that value to change for all instances.
- **Nested Data Structures**
  Since references are scalar values, they can be inserted directly as single elements into lists, or as values into hashes, without causing the implicit "flattening" of the list (hash) in question known as "list interpolation". This allows you to build nested data structures such as:
  - lists of (references to) lists,
  - lists of (references to) hashes,
  - hashes of (references to) lists,
  - hashes of (references to) hashes,
  - lists of (references to) lists of (references to) lists,
  - *etc.*
- **Abstraction & Encapsulation**
  The ability to construct nested data structures allows a greater degree of hierarchical organization – complicated data structures can be organized into a single (recursive) reference.
- **Efficiency**
  Passing references to arrays or hashes into and/or out of your subroutines is almost always more efficient than passing whole "flat" lists or hashes. It also allows you to pass more than one list or hash into (out of) a subroutine.
- **Objects**
  Perl's objects (class instances) are really just references which have been

`bless()`ed into some *package* (read "class"). See the `perltoot(1)` manpage for a tutorial and the `perlobj(1)` manpage for details on Perl's object system.

4.6.3 Symbolic References

"Symbolic" references are also known as "soft" references or "fake" references, and should be familiar to shell programmers. Although Perl allows the use of symbolic references, these can be dangerous (not to mention cryptic) to use in an actual program.

- **Construction**
  - *Symbolic reference from a variable*
    Assign the name of the variable as a string to the symbolic reference.

    ```
    $x = 42;     # Existing variable
    $xr = "x";   # Symbolic reference
    ```

  - *New symbolic reference*
    Pick an unused variable name and assign it as a string to the symbolic reference.

    ```
    $newref = "unused1"; # New(?) symbolic reference
    ```

- **Dereferencing**
  The string-form of `eval()` can be used to access and/or manipulate symbolic references.

  ```
  $xrval = eval "\$$xr";   # Get value
  eval "\$$xr = 24;";      # Set value
  ```

- **Dangers**
  Aside from being cumbersome, symbolic references can be dangerous – the named variable underlying a symbolic reference may go out of scope before the reference is used, or its value may get "clobbered" by another variable of the same name.

- **Upshot**
  Don't use symbolic references if you can avoid it.

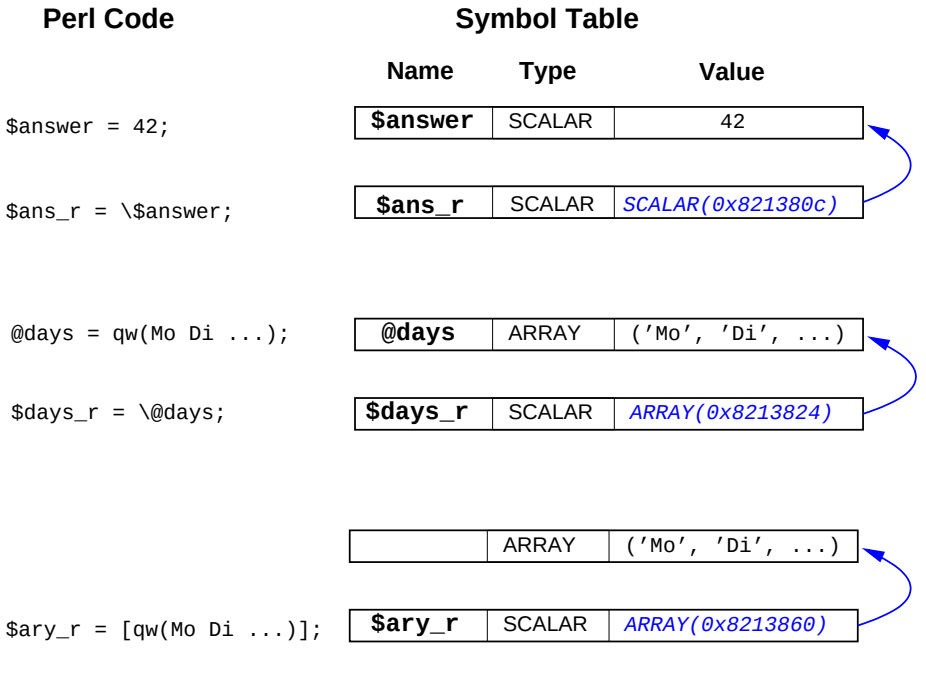
4.6.4 Hard References

> **Thingy** (*noun*): Any Perl value residing in a chunk of physical memory.

Unlike symbolic references, which are really just variable names stored as strings, and therefore only work with named variables, "hard" references (also known as "real" references) refer to a thingy itself – an actual Perl datum residing somewhere in memory. This means that a hard reference can still be used even after a named variable containing the thingy referred to has gone out of scope.

Hard references can even be created independently of any named variable – such references are called "anonymous" references. Some examples of named and anonymous references are given in Figure 5.

Figure 5: Symbol Tables and Hard References



- **Construction**
  - *Construction from a named value*
    A reference to any named value can be created by use of the backslash operator[53]
    - Scalars
      ```
      $x = 42;
      $scalar_ref = \$x;
      ```
    - Scalar Constants
      ```
      $constant_ref = \42;
      ```
    - Arrays
      ```
      @days = qw(Mo Di Mi Do Fr Sa So);
      $array_ref= \@days;
      ```
    - Hashes
      ```
      %noises = (cow => 'moo', cat => 'meow');
      $hash_ref = \%noises;
      ```

[53] This use of Perl's backslash operator shows its similarity to C's "addressing" operator, "&".

    * Subroutines
      ```
      sub foo { return 'bar'; }
      $code_ref = \&foo;
      ```
    * Typeglobs
      ```
      open(HANDLE, ">file.txt");
      $glob_ref = \*HANDLE;
      ```
  - *Anonymous References*
    An anonymous reference can be created together with the thingy it points by using one of the "reference composer" opertors below:
    * Arrays: `[` *LIST* `]`
      ```
      $array_ref = [ 'a', 'b', 'c' ];
      ```
    * Hashes: `{` *LIST_OF_PAIRS* `}`
      ```
      $hash_ref = { a=>1, b=>2, c=>3 };
      ```
    * Subroutines: `sub` *BLOCK*
      ```
      $code_ref = sub { return 'bonk'; };
      ```
  - *Implicit Reference Construction*
    Hard references (together with the thingies they point to) are created implicitly if they are dereferenced appropriately in an assignment expression, much like named Perl variables spring into existence the first time they are assigned to.

- **Dereferencing**
  Now that we've made some references, how do we get at their values?

  - *Reference Variables as Variable Names*
    Perl's type-identifying "funny character" variable prefixes (`$`, `@`, `%`, `&`) work as dereferencing operators, too.[54] Used as dereferencing operators, these funny characters appear as (additional) prefixes to a scalar variable containing a hard reference. Anywhere you can put an identifier as part of a variable or subroutine name, you can replace that identifier with a simple scalar variable containing a reference to a thingy of the appropriate type:

    ```
    ### Basic Dereferencing: Lookup
    $scalar_val = $$scalar_ref; # scalar dereference
    @array_val  = @$array_ref;  # array dereference
    %hash_val   = %$hash_ref;   # hash dereference

    ### Basic Dereferencing: Assignment
    $$scalar_ref = 42;          # scalar dereference
    @$array_ref = qw(a b c);    # array dereference
    %$hash_ref = (a=>1, b=>2);  # hash dereference

    ### Dereferencing + Element Access: Lookup
    $elt_val = $$array_ref[0];  # array derefernce
    $key_val = $$hash_ref{cow}; # hash dereference

    ### Dereferencing + Element Access: Assignment
    $$array_ref[0] = 'a';    # array dereference
    $$hash_ref{cow} = 'moo'; # hash dereference
    ```

[54] Similar to C's dereferencing operator, "*".

You can think of the funny-character prefixes as being evaluated right-to-left: thus, "`$$array_ref[0]`" is the first element of the array referred to by the hard reference "`$array_ref`", and **not** the scalar referred to by the first element of the (normal) array "`@array_ref`".

This is a handy (and more or less intuitive) syntax, but only allows you access to a single level of reference-nesting at a time, which can be frustrating when dealing with nested data structures.

– *BLOCKs as Variable Names*
Anywhere you can use an alphanumeric identifier as part of a variable or subroutine name, you can also use a *BLOCK* which returns a reference to a thingy of the appropriate type:

```
### Block Dereferencing: Assignment
${$scalar_ref} = 42;          # scalar dereference
@{$array_ref} = qw(a b c);    # array dereference
%{$hash_ref} = (a=>1, b=>2);  # hash dereference

### Dereferencing + Element Access: Assignment
${$array_ref}[0] = 'a';    # array dereference
${$hash_ref}{cow} = 'moo'; # hash dereference
```

The useful things about *BLOCK*s as variable names are:

1. A *BLOCK* itself can contain any arbitrary expression, and
2. *BLOCK*s can themselves be nested.

```
### 2-dimensional array access
$a2d = [ [qw(a b c)], [qw(d e f)] ];
$elt = ${ ${$a2d}[0] }[1];          # $elt is now 'b'
```

– *The Arrow Infix Operator:* `->`
Use of *BLOCK*s returning references as variable names is extensible but quite ugly. For this reason, Perl provides the arrow infix operator "`->`" to allow readable direct access to the elements of nested array and hash references:

```
### 2-dimensional array access, with ->
$a2d = [ [qw(a b c)], [qw(d e f)] ];
$elt = $a2d->[0]->[1];              # $elt is now 'b'

### 2-dimensional hash access, with ->
$h2d = {
        num2txt => { 0=>'zero', 1=>'one' },
        txt2num => { zero=>0, one=>1, },
       };
$num = $h2d->{txt2num}->{zero};    # $num is now 0
```

Even niftier is the fact that you can omit the "`->`" between braced or bracketed subscripts (nested element access), so the above examples become:

```
$elt = $a2d->[0][1];
$num = $h2d->{txt2num}{zero};
```

– *Dereferencing Summary*
The following three statements are equivalent:

```
$  $array_ref  [0] = 'thingy';
${ $array_ref }[0] = 'thingy';
   $array_ref->[0] = 'thingy';
```

As are these:

```
$  $hash_ref  {key} = 'thingy';
${ $hash_ref }{key} = 'thingy';
   $hash_ref->{key} = 'thingy';
```

- **Type-Checking:** `ref()`
  You can use the builtin `ref()` function to learn what kind of thingy a hard reference points to:

  `ref` *EXPR*

  If *EXPR* evaluates to a hard reference, returns a string indicating the type of thingy to which that reference refers.

  Returns a false value if *EXPR* does not evaluate to a hard reference. Typical return values are:

  ```
  'SCALAR'
  'ARRAY'
  'HASH'
  'CODE'
  'GLOB'
  ```

  You can think of `ref()` as a *typeof()* operator for hard references.

4.6.5 Reference Counts and Memory Management

Every thingy that comes to be in the course of a Perl interpreter's lifetime has an internal *reference count* associated with it. The reference count for a thingy is just a natural number – the number of hard references (or named variables) currently pointing to the thingy in question.

Perl uses reference counts for memory management: each thingy continues to exist in memory as long as at least one hard reference points to it. A thingy get de-allocated[55] when its reference count reaches zero – when no more hard references point to it. This is why closures[56] work: a closure (anonymous subroutine) may contain a reference to a lexical variable which has since gone out of scope, but whose thingy continues to exist because of the reference to it encoded in the closure.

Usually this memory-management scheme works well, since it handles even nested references correctly:

```
$nested =  [ 'foo', 'bar', [ 'baz', 'bonk' ] ];
# ... stuff happens ...
$nested = undef;  # recursive cleanup
```

When the variable `$nested` is re-assigned, Perl correctly traverses the array

[55] Read: "cast into the eternal void".
[56] See Section 4.3.4.

thingy to which it refers and de-allocates even the nested array `['baz','bonk']` – providing that there are no other references to it lurking about, of course.

Perl's reference-counting strategy has intrinsic difficulty dealing with "circular" data structures, such as "`$tree`", below:

```
## $tree : a traversable tree structure
##   + each node is a hash-ref of the form:
##       {
##         lab  => $node_label,
##         mom  => $mother_node,     # undef for root node
##         dtrs => [$daughter_node_1, ...],
##       }
##   + whole tree is represented by root node
$tree =
  {
   lab  => 'S',                     # root label
   dtrs => [                        # daughters of root
            { lab=>'NP', dtrs=>[] }, # ... first daughter
            { lab=>'VP', dtrs=>[] }  # ... second daugher
           ]
  };
$tree->{dtrs}[0]{mom} = $tree;       # Circular reference!
$tree->{dtrs}[1]{mom} = $tree;       # ... and another!
```

As it stands, this structure will never get de-allocated by Perl, even if the variable `$tree` goes out of scope and even if you explicitly remove the reference to the root node that it contains, for instance by changing its value:

```
$tree = undef;   # Wrong: reference-count is still nonzero!
```

This is because each daugher node (with labels 'NP' and 'VP') contains a reference to the root node as the value of the key 'mom' for the daugher node, which leaves the reference count for the root node at 2, even after the reference held by the variable `$tree` has gone out of scope. Since the value of the 'dtrs' key for the root node contains references to the daughters, the daughters themselves will not get de-allocated either. Even worse, after `$tree` has gone out of scope (or been re-assigned), there is no way for you, the programmer, to access the old reference – it just sits around and takes up memory. Currently, the only way to convince Perl to clean up such circular reference chains is to break the circular references yourself:

```
foreach $d ( @{$tree->{dtrs}} ) {
  delete $d->{mom}; # Break 1 level of circular refs.
}
$tree = undef;      # Right (here): no more circular refs.
```

Of course, in a "real" tree, you are likely to have more than one level of depth, so a simple `foreach` loop such as that shown above would not suffice to break all circular references, and you would have to do something like:

```
@nodes = ($tree);             # Breadth-first search queue
while ($n = shift(@nodes)) {
  delete $n->{mom};           # Break circular references
  push(@nodes, @{$n->{dtrs}}) # Enqueue daughters
    if (exists($n->{dtrs});   # ... if possible
}
$tree = undef;     # Better: no more circular references.
```

4.6.6 Stringification

If you use a hard reference in some context which calls for a string value, the reference will appear as a funny-looking string made up of a type-name and a hexidecimal number:

```
$scalar_ref = \$x;
$array_ref  = [ qw(test 123) ];
$hash_ref   = { foo => 'bar' };
$code_ref   = sub { return undef; };
$glob_ref   = \*STDOUT;
print
  "Scalar ref : $scalar_ref\n",
  "Array ref  : $array_ref\n",
  "Hash ref   : $hash_ref\n",
  "Code ref   : $code_ref\n",
  "Glob ref   : $glob_ref\n";
```

prints something like:

```
Scalar ref : SCALAR(0x825ffbc)
Array ref  : ARRAY(0x8261738)
Hash ref   : HASH(0x8266320)
Code ref   : CODE(0x826638c)
Glob ref   : GLOB(0x80fd4c8)
```

Although it is impossible to recover the actual reference from such a string value (since the reference-count information gets lost in the conversion to a string), two references will produce the same string-form if and only if they point to the same underlying thingy, which means that you can test for reference-equality with the `eq` operator:

```
$x = $y = 42;
$xref = \$x;
$xref2= \$x;
$yref = \$y;
sub eqcheck {
  return $_[0] eq $_[1] ? 'yup' : 'nope';
}
print
```

```
    '$$xref eq $$yref  ? ', eqcheck($$xref, $$yref),  "\n",
    ' $xref eq  $yref  ? ', eqcheck($xref,   $yref),  "\n",
    ' $xref eq  $xref2 ? ', eqcheck($xref,   $xref2), "\n";
```

prints:

```
$$xref eq $$yref  ? yup
 $xref eq  $yref  ? nope
 $xref eq  $xref2 ? yup
```

Since a reference cannot be recovered from its string-form, references do not always work as you would expect when used as hash-keys, since the keys of a hash must be strings.[57] This means that something like the following will not work:

```
$hash{\$x} = 'foo';     # Legal but non-recoverable!
# ... stuff happens ...
foreach $key (keys(%hash)) {
  print "key='$key', deref='$$key'\n";      # Wrong!
}
```

If you need to use references as hash keys and you need to be able to dereference those references later, you can always use another hash to store the actual references as hash *values* (which do not need to be strings), keyed by the references' string-forms:

```
$hash{\$x} = 'foo';        # Keys cannot be dereferenced.
$vals{\$x} = \$x;          # ... but values can.
# ... stuff happens ...
foreach $key (keys(%hash)) {
  print "key='$key', deref='${$vals{$key}}'\n";  # Works.
}
```

[57] See Section 4.2.4 for details.

4.7 Perl Modules *etc.*

4.7.1 What's it All About?

> *The first step toward ecologically sustainable programming is simply: don't litter in the park.*
>
> [WCS96, Ch. 5]

Perl provides several useful mechanisms for ensuring that the metaphorical park stays nice and tidy – chief among these are *Packages*, *Modules*, and *Objects*; each of which are discussed individually in the following sections. See the `perlmod(1)` manpage for details on Perl packages and modules, and see the `perltoot(1)` manpage and the `perlobj(1)` manpage for details on Perl objects.

4.7.2 Packages

A Perl "package" is just a *namespace* in which variables, subroutine names, and filehandles may be located. Placing specialized code in its own package reduces the chance of *name conflicts* ("clobbering") arising between various pieces of code.

- **Package Declaration:** The `package` *NAME* declaration makes *NAME* the current package, creating the package *NAME* if it doesn't already exist.

  `package` *NAME*`; # NAME is a bareword`

- **Package Scope:**
  NAME remains the current package until the next `package` declaration or until the end of the innermost enclosing *BLOCK*, or until end of file (in the case of modules).

- **Default Package `main`:**
  The "default" package in which perl programs run is called `main`.

- **Lookup (qualified):**
  Package variables may be accessed even from outside the package by "qualifying" their identifiers: preceeding them with the name of the containing package followed by two colons "::":

```
$A::x    = 'foo';  # sets $x in package 'A'
$B::x    = 'bar';  # sets $x in package 'B'

$main::x = 'baz';  # sets $x in package 'main'
$::x     = 'baz';  # ... ditto
```

- **Lookup (unqualified):**
  All (non-lexical) unqualified variable and subroutine lookups are performed in the (symbol tables of the) current package:

```
package A;        # package is now 'A'
$x = 'bar';       # sets $A::x
print "$x\n";     # prints value of $A::x

package B;        # package is now 'B'
$x = 'baz';       # sets $B::x
print "$x\n";     # prints value of $B::x
```

- **Nested Packages:**
  Packages may be nested. Nested package names are of the form *OUTER::INNER*, where *OUTER* is the (possibly nested) "parent" package and *INNER* is the identifier of the nested package. No special rules apply to lookup from within nested packages.

```
package A::B::C;  # package is now 'A::B::C'
$x = 'blop';      # sets $A::B::C::blop
```

- **Symbol Tables:**
  The actual contents of a package's symbol table can be accessed via the hash `%`*NAME*`::` for a package named *NAME*, which contains typeglob values keyed by identifiers.

- **Package Maintainence:**
  Packages may contain `BEGIN` and/or `END` blocks, which are evaluated at compile-time and at exit-time, respectively. Such blocks can be useful for package initialization and/or cleanup.

```
package foo;
BEGIN { require 'bar'; }         # runs at compile time
END   { cleanup_temp_files(); }  # runs at exit time
```

- **Autoloading:**
  If a package subroutine is called which does yet exist, but the package defines a special subroutine called `AUTOLOAD`, then the package's `AUTOLOAD` will be called with the arguments passed to the "missing" subroutine, and the package-global variable `$AUTOLOAD` will contain the fully qualified name of the "missing" subroutine, as a string.

- **Current Package**
  The fully qualified name of the current package is contained in the special bareword "`__PACKAGE__`" (two leading and two trailing underscores). This can be especially useful for error messages and for object methods.

- **Tips and Pitfalls:**
  - Certain variables are *always* evaluated in package `main`; these include: `$!`, `$_`, `STDIN`, `STDOUT`, `STDERR`, `ARGV`, `ARGVOUT`, `ENV`, `INV`, and `SIG`.
  - All package identifiers begin with an upper-case letter by convention.

4.7.3 Modules

A Perl Module is just a package containing useful and re-useable abstract goodies stored in a library file. Modules can be loaded at compile time with the `use` pragma, and at runtime with the `require` function.

- **Module Filenames:**
  A module for a package named *PARENT::NAME* should be stored in a file *NAME*`.pm` in a directory *LIBDIR/PARENTDIR*, where
  - *LIBDIR* is some system directory in Perl's global module search path (`@INC`).
  - *PARENTDIR* is like *PARENT* except that package boundaries in *PARENT* (indicated by double-colons "::") are realized as directory boundaries (indicated here by "/").

  For example, on a system whose Perl searches "`/usr/local/lib/site_perl`" for user modules, a module for the package `Foo::Bar::Baz` could be stored in the file "`/usr/local/lib/site_perl/Foo/Bar/Baz.pm`" and imported into a program at compile time by:

  ```
  use Foo::Bar::Baz;
  ```

  and at runtime by:

  ```
  require Foo::Bar::Baz;
  ```

- **Symbol Sharing:**
  The strict distinction between symbol tables of different packages is indeed safe and clean, but often not very comfortable, as it requires the user either to frequently change the current package or to type often large and unwieldy fully qualified package identifiers for subroutines, variables, etc. Perl's `Exporter` module provides a flexible mechanism by which package-internal symbols may be shared between various packages.

- **Exporting Symbols:**

```
package MyPackage;           # A package
require Exporter;            # ... with symbols to share

# @ISA
#  + inheritance-related black magic (for now)
our @ISA    = qw(Exporter);

# @EXPORT
#  + array of symbol identifiers to share by default
our @EXPORT = qw(mysub1);

# @EXPORT_OK
#  + list of symbol identifiers to share if requested
our @EXPORT_OK = qw($myscalar @myarray %myhash);

# %EXPORT_TAGS
```

```
#  + hash of the form ( $tag => \@symbols, ... ) of symbols
#    to be shared by symbolic tag name ":$tag"
our %EXPORT_TAGS = (
                    mytag1 => [qw($myscalar @myarray)],
                    mytag2 => [qw(mysub1 %myhash)],
                    all    => [@EXPORT, @EXPORT_OK],
                   );
```

- **Importing Symbols (Defaults):**
  Assuming you have a package `MyPackage` as above which exports some symbols, and you would like to use (some of) the exported symbols in another package `MyUser` in an unqualified manner:

  The symbols in `@MyPackage::EXPORT` are exported by default:

  ```
  package MyUser;
  use MyPackage;  # load package & import symbols
  ```

  The above code is equivalent to the runtime code:

  ```
  package MyUser;
  BEGIN {
    require MyPackage;   # load package
    MyPackage->import(); # import default symbols
  }
  ```

- **Importing Symbols (Explicit Requests)**
  Suppose that from `MyUser` we wanted to access one or two symbols that `MyPackage` exports, but which are not exported by default – i.e. symbols in `@MyPackage::EXPORT_OK` but not in `@MyPackage::EXPORT`. For such cases, the *LIST* argument of the `use` *MODULE LIST* pragma is extremely useful:

  ```
  package MyUser;
  use MyPackage qw($myvar @myarray);
  ```

  or:

  ```
  package MyUser;
  BEGIN {
    require MyPackage;
    MyPackage->import( qw($myvar @myarray) );
  }
  ```

  Note that in the above cases, **only** the requested symbols are imported, and not the default symbols, if there were any.

- **Importing Symbols (Tags):**
  If, as above, the package `MyPackage` exports groups of symbols under some symbolic tag name *TAG* (a key in `%MyPackage::EXPORT_TAGS`), then the tag name preceeded by a symbol colon ":" can appear in the *LIST* argument to `use` or `import` in order to import all of the symbols assigned to *TAG* by `MyPackage`:

```
package MyUser;
use MyPackage qw(:mytag1 :mytag2);
```

or:

```
package MyUser;
BEGIN {
  require MyPackage;
  MyPackage->import( qw(:mytag1 :mytag2) );
}
```

- **The `CORE` package**
  Builtin Perl functions (such as `print`, `chdir`, *etc.*) may be overridden by a subroutine within a given package, and the overrides may even be propagated out of the package by means of a symbol-sharing mechanism. The original version of any built-in Perl function should always be available in the package `CORE` however:

```
package Foo;

# undef = print(@args)
#   + prefixes the current package name and passes
#     @args to Perl's builtin print() function.
sub print {
  CORE::print( __PACKAGE__ , ": ", @_ );
}
```

- **Tips, Caveats,** *etc.*
  - Avoid exporting any symbols by default, since that pretty much defeats the purpose of packages as "safe" havens for your identifiers.
  - Generally try to avoid overriding builtin Perl functions – you can still get at them using fully qualified names in the `CORE` package, but overridden builtin names are usually very difficult to read.
  - When writing your own modules, use the `ExtUtils::MakeMaker` module to create a portable and intuitive build environment. See the `ExtUtils::MakeMaker(3pm)` module manpage for details.

4.7.4 Objects

- **Terminology:**
  - **Class** *(noun)*: a collection of things-in-the-world (in the case of programming, a collection of data-in-the-program) which share a common set of *properties* or *behaviors*.
  - **Object** *(noun)*: an *instance* of a *class*: in programming, a single (possibly complex) datum with some properties or behaviors given by its associated class.

- **Method** *(noun)*: in programming, a chunk of code implementing some property or behavior common to a class of objects, either as a function of the class itself (a *class method*) or as a function of a single instance of that class (a *object method* or *instance method*).

  Common class methods include *constructors*, which create and return new object instances, and *destructors*, which are responsible for destroying object instances which are no longer needed.
- **Parent Class** *(noun phrase)*: comparative term indicating a "more general" class or *superclass*, also known as a *base class*. Compare *child class*.
- **Child Class** *(noun phrase)*: comparative term indicating a "more specific" class or *subclass*, also known as a *derived class*. Compare *parent class*.
- **Inherit** *(verb)*: to receive an item, property, or behavior by virtue of one's genealogy – in programming, to receive access to a *method* defined for a *parent class* (also indirectly, e.g. defined for a grandparent class, *etc.*).

• **Perl Classes**
Perl classes are just packages.

```
package Rectangle;  # declares a class 'Rectangle'
```

• **Perl Objects**
Perl objects are just references which have been `bless()`ed into a class package.

```
my $obj = {};             # create a reference
bless $obj, 'Rectangle';  # ... bless it into an object
```

• **Perl Methods**
Perl methods are just subroutines declared in a class package which take as their first argument either a class name (class methods) or an object (instance methods).

- **Perl Constructors**
  Perl constructors are conventionally called `new`. They should create a reference, `bless()` it, and return the blessed reference:

```
# $rect = Rectangle->new($width,$height)
#   + create and return a new rectangle
sub new {
  my ($class,$width,$height) = @_;
  return bless({
                width  => $width,
                height => $height,
               }, $class);
}
```

- **Perl Destructors**
  Perl destructors are always called `DESTROY`, and receive as their only argument the object instance to be destroyed. They may perform

user-defined cleanup operations, such as breaking circular references within the object.

```
# undef = DESTROY($rect)
#   + destructor
sub DESTROY {
  my $obj = shift;
  print "$obj is being cast into the void.\n";
}

# $area = $rect->area();
#   + returns area of the rectangle object $rect
sub area {
  my $self = shift;
  return $self->{width} * $self->{height};
}
```

- **Perl Inheritance**
  Inheritance in Perl is implemented by means of the special package array `@ISA`, which holds in every child class package the names of the subclass's parent classes:

```
package Square;       # declare a class 'Square'
@ISA = qw(Rectangle); # ...inheriting from 'Rectangle'

# $sqr = Square->new($width)
#  + create and return a new Square object
sub new {
  my ($class,$width) = @_;

  # ... explicitly crafted call to Rectangle::new()
  return Rectangle::new($class, $width, $width);
}

# DESTROY() : inherited from 'Rectangle'
# area()    : inherited from 'Rectangle'
```

  For the truly gory details on inheritance search order and other such things, see the `perlobj(3pm)` module manpage .

- **Using Perl Methods**
  Perl methods are most intuitively called with the help of the "`->`" operator:

  CLASS_OR_INSTANCE `->` *METHODNAME* `(` *ARGS* `)`

  For example:

```
my $r = Rectangle->new(42, 2);
my $s = Square->new(10);

print
  ("Area of r is: ", $r->area(), "\n",
   "Area of s is: ", $s->area(), "\n");
```

There is another syntax available for calling Perl methods, the so-called "indirect object" syntax, but I don't recommend it.

5 Miscellaneous Bits

5.1 Efficiency

> TMTOWTDI (There's More Than One Way To Do It)
>
> [WCS96]

> NAWTDIACE (Not All Ways To Do It Are Created Equal)
>
> – *the author of this document*

The most important aspect of programming is (of course) getting your program to do whatever it is that it is intended to do. Later (or perhaps right away), you may wish to optimize your program for speed and/or space efficiency. I find that it is generally a good idea to do some "optimization" right from the start, which saves me some work later. Also, there are other flavors of "efficiency" not often discussed by theoreticians[58] which are best observed right from the outset.

This Section contains some brief descriptions of several varieties of "efficiency", along with some rules of thumb you can follow to make your Perl programs more efficient. [WCS96, Chapter 8] contains many more of these. Often, these rules conflict with one another – in such cases:

> Them's the breaks. If programming were easy, they wouldn't need anything as complicated as a human being to do it, now would they?
>
> [WCS96, p. 537]

5.1.1 Time Efficiency

Maximizing time efficiency simply means minimizing the time for which your program runs. This may seem unimportant for a program with an average lifetime of 0.004 seconds, but if that program is a CGI on a busy server which might get called 500 times in a second, you have a problem.

Generally, maximizing time efficiency means:

- Burn fewer CPU cycles
- Do less I/O

Idle Pontification:
I personally tend to rate time efficiency as very important, usually overriding the demands of space efficiency, at least until I run out of RAM.[59]

Rules of Thumb:

[58] ... who tend to behave as if $O = O(n)$ explains everything you need to know about a program.

[59] Yes, this has happened.[60]

[60] More often than I would like to think about, thank you.

- Use as few loops as possible – often, you can eliminate a loop by using list context.
- Only use nested loops if you absolutely must.
- Don't use recursion – there's usually a way to compute every recursive function iteratively.
- Use `foreach` loops instead of `for` loops over index variables.
- Use hashes instead of array searches.
- Use the builtin Perl functions – they're usually quite fast.
- Don't use the string form `eval`, especially inside a loop.
- Don't call too many user-defined subroutines.
- Just use fewer elementary operations.

5.1.2 Space Efficiency

Maximizing space efficiency simply means minimizing the amount of memory (especially RAM) that your program occupies. Again, this may seem unimportant for a program with an average "footprint" of 4 KB, but many program instances or more input data can often cause space efficiency to become extremely important.

Generally, maximizing space efficiency means:

- Use less RAM
- Store less data

Idle Pontification:
I personally tend to rate space efficiency as very important, but usually prefer time efficiency when the two conflict.

Rules of Thumb:

- Use fewer variables – actually, this helps with time efficiency, too.
- Don't store data (such as input) in an array when a loop over your source data (your input stream(s)) would suffice.
- Prefer numbers to strings for scalar values.
- Use iterators, especially `each`.
- `undef` variables when they are no longer needed.

5.1.3 Programmer Efficiency

Maximizing programmer efficiency just means minimizing the amount of work that you, the programmer, have to do in order to get your program working.

Generally, this means:

- Do more work with less code.

Idle Pontification:
Personally, I tend to rate programmer efficiency lower than just about every other flavor, except when I'm just trying to find some snippet of code that does what I need it to do in a hurry – i.e. whenever I really need Perl.

Rules of Thumb:

- Use the modules from CPAN – the best program is one you don't have to write yourself.
- Use the default variable `$_` – save your carpal tunnels!
- Use funky command-line switches such as `-p` or `-n`.
- Use whatever you think of first.

5.1.4 Maintainer Efficiency

Maximizing maintainer efficiency just means making your program easy to understand.

Generally, this means:

- Write readable code.
- Write modular (re-usable) code.
- Document.
- Document?
- **Document!**

Idle Pontification:
I personally tend to rate maintainer efficiency very highly indeed – often even overriding the demands of time- and space-efficiency, when they conflict.[61]

Rules of Thumb:

- Use meaningful variable names.
- Use meaningful subroutine names.

[61] This is largely because I have to maintain my own code.

- Use meaningful loop labels.
- Don't use global variables.
- Comment your code.
- Document your data-structure conventions, at the very least with commented initial declarations.
- Document your subroutines' argument types and return values, at the very least with commented subroutine declarations.
- Name your subroutine parameters using `my`.
- Use round parentheses for clarity – for grouping sub-lists, as well as for subroutine calls.
- Close your files when you are done with them.
- Use packages, modules, and classes.

5.1.5 Porter Efficiency

Maximizing porter efficiency just means minimizing the amount of work that you (or someone else) have (has) to do in order to get your program to run on another platform. See the `perlport(f)` manpageor details.

Generally, this means:

- Avoiding platform-specific hacks.
- Avoiding binary encodings.

Idle Pontification:
Generally, I tend not to think much about porter efficiency – then again, I have never attempted to port my Perl programs to a non-UNIX operating system.[62]

Rules of Thumb:

- Avoid functions that aren't implemented everywhere.
- Don't try to send binary data over a pipe.
- Don't use binary data and expect them to work the same everywhere.
- If you must use binary data, pick a byte-order (for `pack()` and `unpack()`) and stick with it.
- Use `require $VERSION` to ensure that the porter is running a sufficiently new version of the Perl interpreter.
- Put in the "shebang" line (`#!/usr/bin/perl`) as the first line of your script, even if your platform does not support it.

[62] Bad programmer. No biscuit.

5.1.6 User Efficiency

Maximizing user efficiency just means writing programs that are intuitive and easy to use.

Generally, this means:

- Not confusing your user(s).[63]
- Documenting user-level functionality.

Idle Pontification:
I personally tend to rate user efficiency quite highly; then again, I myself tend to be the primary user of my own programs.

Rules of Thumb:

- Use sensible default values, but allow the user to change them.
- Use the `Getopt` and/or `Getopt::Long` modules to parse command-line options.
- Use prompts to nudge beginning (or forgetful) users along.
- Allow advanced users to skip the prompts.
- Check for error conditions and produce helpful error messags with `warn()` and `die()`.
- Allow input to come from either STDIN or from files on the command-line, at the user's option.
- Allow output to go either to a file or to STDOUT, at the user's option.
- Be nice.

[63] Of course, you must have some idea of who your users are and what they expect in order to meet this criterion.

5.2 Coding With Style

This Section contains some rules of thumb to follow when writing your own Perl programs. While stylistic issues such as these are to a large extent aesthetic issues, they also have a significant influence on the maintainability and re-usability of your code: it is *much* easier to debug, maintain, and re-use *pretty* code than *ugly* code.

See [WCS96, Chapter 8] for more suggestions.

5.2.1 Indentation

- **Blocks**
  Indent the *contents* of every block by a constant amount – usually about 3 characters. Ensure that the closing brace for every block is at the same indentation level as the line containing the opening brace, which should be the same level as the keyword (if any) requiring the block.

Pretty	**Not So Pretty**

`if ($foo) {`
`  bar();`
`}`
	`if ($foo){`
	`bar();`
	`}`

`if ($foo)`
`{`
`  bar();`
`}`
	`if ($foo)`
	`{`
	`  bar();`
	`  }`

- **Function Arguments**
  When function or subroutine arguments extend over more than one line, use round parentheses "()" around your subroutine's arguments and indent the later lines to one character past the column of the opening "(" of that subroutine:

Pretty	**Not So Pretty**
`foo($bar,` `  $baz);`	`foo($bar,` `$baz);`

- **Multiline Statements**
  When simple statements continue over more than one line, indent later lines at least 2-3 characters. When multiline statements are complex expressions, break lines at sub-expression boundaries and indent according to sub-expression nesting level:

Pretty	Not So Pretty
`open(FOO,"<foo.txt")` `  or die("oops: $!");`	`open(FOO,"<foo.txt")` `or die("oops: $!");`

`if (!$x || ($x eq $y`
`  &&`
`  $y eq $z))`
	`if (!$x || ($x eq $y`
	`&& $y eq $z))`

5.2.2 Blank Lines

- **Statements**
  Start a new line for every statement.

- **Declarations**
  Use some blank lines between declarations (i.e. subroutine declarations) and the following code – especially if that code is another subroutine declaration!

- **Conceptual Grouping**
  Use blank lines to separate portions of your code into conceptual groupings.

5.2.3 Comments

- **File Comments**
  At the top of each file, add comments which name you as the author, and which briefly describe the program and what it does – for example:

```
#!/usr/bin/perl -w
#
# File: codecounter.perl
# Author: Bryan Jurish <moocow@ling.uni-potsdam.de>
# Description:
#   + Report relative amounts of code, comments,
#     whitespace, and PODs in perl source file(s).
#  Usage:
#   $0 [FILE(s)...]
```

- **Subroutine Comments**
  Introduce each subroutine with some comments that describe its arguments and return value:

```
# $sum = add(@numbers);
#   + returns the sum of the elements of @numbers
#   + undefined elements are mapped to 42.
```

- **Conceptual Groups**
  Separate portions of your program (groups of subroutine declarations, etc.) from one another by comments. I like to use the "=" and "-" characters to make long (60–70 characters) horizontal lines for larger groups, using shorter lines for smaller conceptual groups:

```
#=======================================
# Common Subroutines
#=======================================

#----------------------------
# I/O subs
#----------------------------

# undef = foo($bar);
#   + just a dummy sub
sub foo {
  my ($bar) = shift;

  #-- sanity check
  return undef if (!defined($bar));

  #-- the actual guts
  return baz($bar) if (blippo(bonk()));
  return $bar;
}
```

5.3 When Things Go Wrong

First of all ... **Don't Panic!**

Perl is a nightmarishly complex language to parse[64]. It is therefore a truly astounding fact that Perl is capable of producing such a broad range of diagnositic warning and error messages meant to help you, the programmer, in perfecting and debugging your code.

5.3.1 Grokking the Diagnostics

Grok (*verb*): To understand and appreciate, usually in a global sense.

1. Read the error message. Carefully.
2. If you haven't already done so, turn on the "-w" switch to the perl interpreter, or set the "$^W" variable to a true value, and see if the additional information gives you any clues.
3. Look at the line of your code mentioned in the error message – do you see any obvious causes? Typical candiates include: typographical errors ("typos"), missing commas, missing semicolons, missing parentheses, missing "$", "@", or "%" characters. If you don't see anything on the line mentioned, check the surrounding context (statement, block, subroutine, etc.).
4. Consult the `perldiag(1)` manpage to find your error message, and see if the information there gives you any additional clues.
5. Try running your program under the Perl debugger (see Section 5.3.4) Set a breakpoint shortly before the line that causes the error, and examine ("`x`") the relevant variables to see if their values are what you expect them to be.
6. Fiddle around with it. The debugger is a good place to try out small variations of single lines of code in the context of a running program.
7. Go to Step 1.

5.3.2 Common Warnings

The following are some common Perl warnings, using `printf()` style escapes (such as "%s"). These warnings only appear if you ran perl with the "-w" switch (or if you have set "$^W" to a true value). Their presence is not fatal to Perl, but might indicate that all is not well with your program.

- **Deep recursion on subroutine "%s"**

  Some subroutine called itself more than 100 times – did you code the right termination condition?

[64] Not as nightmarishly complex as, say, English, but pretty nightmarishly complex nonetheless.

- **%s (...) interpreted as function**

  Usually not really a problem. See the `perlop(1)` manpage for details.

- **Name "%s::%s" used only once: possible typo**

  You have coded a useless variable – did you mistype its name?

- **Unquoted string "%s" may clash with future reserved word**
  Did you forget quotes around a string? Or mistype a subroutine name? Or forget the "$" prefix for a scalar variable?

- **Useless use of %s in void context**
  Perl failed to parse your program as you intended. Usually a precedence problem – try some parentheses.

- **Use of uninitialized value**
  You tried to perform some operation on a scalar whose value is `undef`. This is legal, and often things will work out alright, but you should think about giving your variable some defined value.

5.3.3 Perl Errors

Here are some common Perl error messages, which will cause the Perl interpreter to terminate.

- **BEGIN failed–compilation aborted**

  Usually the result of an error in some package your program `use`s.

- **Can't find string terminator %s anywhere before EOF**

  You seem to have forgotten the closing quotes on string literal.

- **Can't locate %s**

  You probably tried to `use` a module which Perl couldn't find – do you have it installed? If so, check `@INC`; if not, check CPAN.

- **Can't modify %s in %s**

  You overtaxed Perl's capabilities. Assigning to a temporary variable usually solves this one.

- **%s found where operator expected**

  Probably the most common error message I see, this generally means you forgot a semicolon or a comma. Sometimes you will see the additional message:

  (Do you need to predeclare %s?)

  which means that Perl is interpreting "%s" as a subroutine – maybe you forgot a "$" or an "@" on some variable name?

- **Might be a runaway multi-line %s string starting on line %d**

  Indicates that the previous error might have been caused by a missing string-delimiter such as ' or ".

- **Missing right curly or square bracket**

  You probably forgot to close all of your blocks. Use an editor such as (x)emacs or vim which matches parentheses for you, or count more carefully.

- **Search pattern not terminated**

  You may have forgotten to terminate a regex pattern. Forgetting the leading "$" on a variable "$m" could cause this error.

- **Undefined subroutine &%s called**

  You tried to call a subroutine which you haven't defined. Possibly a typo.

5.3.4 The Perl Debugger

You can run the Perl debugger by using the "-d" switch to the Perl interpreter. I personally prefer to use an editor such as (x)emacs with builtin support for the perl debugger (in xemacs, right click on the buffer containing your perl code file, pick "debugger", and type the command-line into the minibuffer), so I can easily edit and debug my programs in parallel, and get pretty syntax highlighting and auto-indentation in the bargain.

Some common debugger commands include:

Command	Description
`h`	print a helpful summary of known debugger commands
`q`	exit the debugger
`b` *LINE*	set a breakpoint at *LINE*
`c`	continue until the next breakpoint
`l`	list current line
`s`	step into next line
`n`	next – step over subroutines
`x` *EXPR*	examine expression *EXPR* in list context
EXPR	evaluate any perl expression

For more information on the perl debugger, see the `perldebtut(1)` manpage and/or the `perldebug(1)` manpage.

A A Brief Review of Set Theory

- **Set**
  A *set* is simply a collection of objects (called the *elements* of the set) in which a given element may be present at most once. We write $x \in S$ to indicate that the object x is an *element of* the set S. Typically, two basic forms of set-designation are recognized: *explicit enumeration* and *restriction*.

- **Set Enumeration**
  The set containing all and only the objects $x_1, \ldots, x_n$ is representend by $\{x_1, \ldots, x_n\}$.

- **Set Restriction**
  $\{x \mid P(x)\}$ represents the set containing all and only those objects x which satisfy the logical predicate P.

- **Empty Set**
  The *empty set* is the set containing no objects, often written $\emptyset$. It can be defined in terms of enumeration as: $\emptyset := \{\}$.

- **Set Union**
  $S_1 \cup S_2 \quad := \quad \{x \mid x \in S_1 \text{ or } x \in S_2\}$

- **Set Difference**
  $S_1 - S_2 \quad := \quad \{x \mid x \in S_1 \text{ and } x \notin S_2\}$

- **Set Intersection**
  $S_1 \cap S_2 \quad := \quad S_1 - (S_1 - S_2)$
  $= \quad \{y \mid y \in S_1 \text{ and } y \in S_2\}$

- **Subset Relation**
  $S_1 \subseteq S_2 \quad :\Leftrightarrow \quad \forall x(x \in S_1 \Rightarrow x \in S_2)$

- **Set Identity**
  Two sets S_1 and S_2 are taken to be identical if they contain exactly the same objects:
  $S_1 = S_2 \quad :\Leftrightarrow \quad S_1 \subseteq S_2 \text{ and } S_2 \subseteq S_1$
  $\Leftrightarrow \quad \forall x(x \in S_1 \Leftrightarrow x \in S_2)$

B A Brief Review of Tree Domains

- **Tree Nodes**
  A *tree domain* is defined with respect to the free monoid $\langle U, \circ \rangle$, where U is the set of all finite strings of positive numbers separated by dots "." and $\circ$ is the concatenation operation. The identity element of the free monoid U is the empty string ε. The elements of U can be understood as "node identifiers" in concrete trees.

- **Node Dominance Relation**
  The relation $\trianglelefteq$: $U \times U$ is a binary *dominance relation* on node identifiers, defined as: $\forall u, v \in U : u \trianglelefteq v$ iff $(\exists w \in U : v = u.w)$

- **Node Precedence Relation**
  The relation $\prec: U \times U$ is a binary *precedence relation* on node identifiers, defined as: $\forall u, v \in U : u \prec v$ iff $\exists x, y, z \in U, \exists i, j \in \mathbb{N} : u = x.i.y \;\&\; v = x.j.z \;\&\; i < j$

- **Tree Domain**
  A *tree domain* is a finite subset D of U for which the following conditions hold:

  1. **Dominance Closure**

  $$\forall u, v \in U : ((v \in D \;\&\; u \trianglelefteq v) \Rightarrow u \in D)$$

  2. **Strict Precedence**

  $$\forall u \in U, \forall i, j \in \mathbb{N} : ((u.j \in D \;\&\; 1 \leq i \leq j) \Rightarrow u.i \in D)$$

- **Labelled Tree**
  A *labelled tree* is a triple $\langle D, \Sigma, L \rangle$ where:

  - D is a tree domain, a set of node identifiers.
  - Σ is a finite *node label alphabet*.
  - $L : D \rightarrow \Sigma$ is a total *node-labelling* function.

  For the purposes of the following definitions, let $t = \langle D, \Sigma, L \rangle$ be a labelled tree, let $u \in U$ be a (potential) node identifier, and let $i \in \mathbb{N}$ be a natural number.

- **Tree Root Node**
  $\mathrm{root}(t) = \varepsilon$

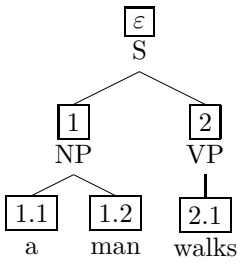
- **Node Daughters**
  $\mathrm{daughters}(t, u) = \{u.i \in (D \circ \mathbb{N}) \mid u.i \in D\}$

- **Node Mother**
  $\mathrm{mother}(t, u.i) = u$

- **Tree Leaves**
  $\mathrm{leaves}(t) = \{u \in D \mid \mathrm{daughters}(t, u) = \emptyset\}$

- **Example:**
  In the following example, node identifiers are shown boxed.

$$\begin{aligned} D &= \{\varepsilon, 1, 2, 1.1, 1.2, 2.1\} \\ \Sigma &= \{\mathrm{S}, \mathrm{NP}, \mathrm{VP}, \mathrm{a}, \mathrm{man}, \mathrm{walks}\} \\ L &= \left\{ \begin{array}{r} \varepsilon \mapsto \mathrm{S} \\ 1 \mapsto \mathrm{NP}, \\ 2 \mapsto \mathrm{VP}, \\ 1.1 \mapsto \mathrm{a}, \\ 1.2 \mapsto \mathrm{man}, \\ 2.1 \mapsto \mathrm{walks} \end{array} \right\} \end{aligned}$$



An alternative formal definition of labelled trees can be found in Barbara Partee's lecture notes.[65]

[65] Warning: The "nontangling condition" on page 3 should be:

$$(\forall w, x, y, z \in N)((\langle w, x\rangle \in P \;\&\; \langle w, y\rangle \in D \;\&\; \langle x, z\rangle \in D) \rightarrow \langle y, z\rangle \in P)$$

References

[SC97] Randal L. Schwartz and Tom Christiansen. *Learning Perl, Second Edition.* O'Reilly & Associates, Köln, 1997.

[WCS96] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl, Second Edition.* O'Reilly & Associates, Köln, 1996.