

Zahlen

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.¹

– John von Neumann (1951)

2.0 Einführung

Die Zahl, der elementarste Datentyp nahezu jeder Programmiersprache, kann überraschend knifflig sein. Zufallszahlen, Zahlen mit Dezimalpunkten, Zahlenreihen und die Umwandlung zwischen Strings und Zahlen können eine Menge Ärger bereiten.

Perl arbeitet hart daran, Ihnen das Leben einfach zu machen, und die Möglichkeiten, die es zur Manipulation von Zahlen bereitstellt, bilden hier keine Ausnahme. Wenn Sie einen skalaren Wert als Zahl betrachten, wandelt Perl ihn entsprechend um. Wenn Sie also Altersangaben aus einer Datei einlesen, Ziffern aus einem String extrahieren oder Zahlen aus einer der vielen Textquellen filtern, die Ihnen im echten Leben begegnen, müssen Sie nicht die Klimmzüge machen, die Ihnen andere Sprachen durch ihre umständlichen Anforderungen beim Umwandeln von ASCII-Strings in Zahlen auferlegen.

Perl versucht sein Möglichstes, einen String als Zahl zu interpretieren, wenn Sie ihn so verwenden (etwa in mathematischen Ausdrücken), besitzt aber keine direkte Möglichkeit, Ihnen mitzuteilen, dass ein String keine gültige Zahl repräsentiert. Perl wandelt nicht-numerische Strings stillschweigend in eine Null um und bricht die Umwandlung des Strings ab, sobald das erste nicht-numerische Zeichen erkannt wird – somit bleibt "A7" 0, während "7A" zu einer 7 wird. (Allerdings warnt das Flag `-w` Sie bei solch unsauberen Konvertierungen.) Manchmal, etwa bei der Validierung von Eingaben, müssen Sie wissen, ob ein String eine gültige Zahl repräsentiert. Wie Sie das herausfinden, zeigen wir Ihnen in Rezept 2.1.

¹ »Jeder, der arithmetische Methoden zur Erzeugung von Zufallszahlen in Betracht zieht, befindet sich in einem Zustand der Sünde.«

Wir runden das Kapitel mit Rezepten zu Trigonometrie, Logarithmen, Matrixmultiplikation, komplexen Zahlen und anderen häufig gestellten Fragen (»Wie kann ich Punkte in Zahlen einfügen?«) ab.

2.1 Prüfen, ob ein String eine gültige Zahl darstellt

Problem

Sie möchten überprüfen, ob ein String eine gültige Zahl repräsentiert. Das ist ein gängiges Problem bei der Validierung von Eingaben, beispielsweise in CGI-Skripten, Konfigurationsdateien und Kommandozeilenargumenten.

Lösung

Vergleichen Sie den String mit einem regulären Ausdruck, der die Art von Zahlen erkennt, an denen Sie interessiert sind:

```
if ($string =~ /MUSTER/) {
    # Zahl
} else {
    # keine Zahl
}
```

Oder nutzen Sie die vom CPAN-Modul Regexp::Common bereitgestellten Muster:

```
if ($string =~ m{^$RE{num}{real}$}) {
    # Realzahl
} else {
    # oder nicht
}
```

Diskussion

Das Problem geht der Frage auf den Grund, was wir als Zahl verstehen. Selbst sich einfach anhörende Dinge wie Ganzzahl (*Integer*) lassen Sie schwer darüber grübeln, was erlaubt ist und was nicht. Zum Beispiel: »Ist ein führendes + für positive Zahlen optional, obligatorisch oder verboten?« Die vielen Arten, auf die Fließkommazahlen dargestellt werden, könnten Ihr Gehirn überhitzen.

Entscheiden Sie, was Sie wollen und was Sie nicht akzeptieren können. Danach müssen Sie einen regulären Ausdruck entwickeln, der genau diesen Sachverhalt erkennt. Hier sind einige vorgekochte Lösungen (die Cookbook-Variante von Fertiggerichten) für die gängigsten Fälle:

```
warn "Keine reine Zahl"          if    /\D/;
warn "Keine natürliche Zahl"    unless /\d+$/;           # verwirft -3
warn "Keine Integerzahl"       unless /\d+$/;           # verwirft +3
warn "Keine Integerzahl"       unless /^[+-]\d+$/;
warn "Keine Dezimalzahl"       unless /^-?\d+\.\d*$/; # verwirft .2
```

```
warn "Keine Dezimalzahl"      unless /^-?(?:\d+(?:\.\d*)?|\.\d+)$/;
warn "Kein C-Float"          unless /^( [+ ]? ) ( ? = \d | \. \d ) \d * ( \. \d * ) ? ( [ Ee ] ( [ + - ] ? \d + ) ) ? $ /;
```

Diese Zeilen decken die IEEE-Notationen für »Infinity« (»unendlich«) und »NaN« (»not a number«, also »keine Zahl«) zwar nicht ab, aber solange Sie sich keine Sorgen machen müssen, dass Mitglieder des IEEE-Komitees an Ihrem Arbeitsplatz haltmachen und Ihnen die Dokumente mit den relevanten Standards über den Schädel ziehen, dürfen Sie diese seltsamen Formen durchaus vergessen.

Besitzt Ihre Zahl führende oder anhängende Whitespaces, funktionieren diese Muster nicht mehr. Fügen Sie die entsprechende Logik direkt ein, oder verwenden Sie die `trim`-Funktion aus Rezept 1.19.

Das CPAN-Modul `Regexp::Common` stellt eine Vielzahl fertiger Muster zur Verfügung, mit denen Sie prüfen können, ob ein String wie eine Zahl aussieht. Abgesehen davon, dass Sie die Muster nicht selbst bestimmen müssen, wird Ihr Code auch leserlicher. Standardmäßig exportiert dieses Modul einen Hash namens `%RE`, den Sie entsprechend des gewünschten regulären Ausdrucks indizieren. Achten Sie darauf, bei Bedarf mit Ankeren zu arbeiten, da das Muster sonst überall im String gesucht wird. Hier ein Beispiel:

```
use Regexp::Common;
$string = "Gandalf verliess die Anfurten in 3021 DZ.";
print "Ist eine Integerzahl\n"      if $string =~ / ^ $RE{num}{int} $ /x;
print "Enthält die Integerzahl $1\n" if $string =~ / ( $RE{num}{int} ) /x;
```

Die folgenden Beispiele zeigen weitere Muster, die das Modul zum Matching von Zahlen verwenden kann:

```
$RE{num}{int}{-sep=>','?'}          # erkennt 1234567 oder 1,234,567
$RE{num}{int}{-sep=>'.'}{-group=>4} # erkennt 1.2345.6789
$RE{num}{int}{-base => 8}           # erkennt 014, nicht aber 99
$RE{num}{int}{-sep=>','}{-group=3}  # erkennt 1,234,594
$RE{num}{int}{-sep=>'?'}{-group=3}  # erkennt 1,234 oder 1234
$RE{num}{real}                     # erkennt 123.456 oder -0.123456
$RE{num}{roman}                     # erkennt xvii oder MCMXCVIII
$RE{num}{square}                   # erkennt 9 oder 256 oder 12321
```

Einige dieser Muster, etwa `square`, waren in früheren Modul-Versionen nicht verfügbar. Die allgemeine Dokumentation zu diesem Modul finden Sie in der `Regexp::Common`-Manpage, eine detailliertere Dokumentation zu numerischen Mustern aber in der `Regexp::Common::number`-Manpage.

Einige Techniken zur Identifizierung von Zahlen basieren nicht auf regulären Ausdrücken. Diese Techniken nutzen stattdessen Funktionen der Systembibliotheken oder von Perl, um zu ermitteln, ob ein String eine akzeptable Zahl enthält. Natürlich schränken diese Funktionen Sie auf die Definition von »Zahl« ein, die Ihre Bibliotheken und Perl zur Verfügung stellen.

Wenn Sie an einem POSIX-System sitzen, können Sie die von Perl unterstützte `POSIX::strtod`-Funktion nutzen. Die Semantik ist allerdings etwas seltsam, weshalb Sie unten eine Wrapper-Funktion namens `getnum` finden, die den Zugriff etwas bequemer

gestaltet. Die Funktion verlangt einen String als Argument und gibt entweder die erkannte Zahl zurück oder undef, wenn es sich nicht um eine Zahl vom C-Typ float handelt. Die Funktion `is_numeric` dient einfach als Frontend zu `getnum`, falls Sie einfach nur wissen möchten, ob es sich um einen float-Wert handelt.

```

sub getnum {
    use POSIX qw(strtod);
    my $str = shift;
    $str =~ s/^\s+//;          # führenden Whitespace entfernen
    $str =~ s/\s+$//;        # angehängten Whitespace entfernen
    $! = 0;
    my($num, $unparsed) = strtod($str);
    if (($str eq '') || ($unparsed != 0) || $!) {
        return;
    } else {
        return $num;
    }
}

sub is_numeric { defined scalar &getnum }

```

Das `Scalar::Util`-Modul, neuer Standard seit Perl v5.8.1, exportiert eine Funktion namens `looks_like_number()`, die eine interne Funktion des Perl-Compilers gleichen Namens verwendet (siehe *perlapi(1)*). Diese Funktion gibt wahr für jede Zahl zu Basis 10 zurück, die für Perl selbst akzeptabel ist, etwa 0, 0.8, 14.98 und 6.02e23 – nicht aber bei 0xb1010, 077, 0x392 oder bei Zahlen, die Unterstriche enthalten. Wenn Sie eine andere Basis unterstützen wollen, müssen Sie also prüfen, ob sie vorliegt und diese dann selbst decodieren. Beispiel 2-1 zeigt eine mögliche Lösung:

Beispiel 2-1: Zahlen decodieren

```

#!/usr/bin/perl -w
use Scalar::Util qw(looks_like_number);
print "$0: Beenden mit ^D (Ihrem EOF-Zeichen)\n";
for (;;) {
    my ($on, $n);          # Originalstring und dessen numerischer Wert
    print "Geben Sie eine beliebige Zahl ein: ";
    $on = $n = <STDIN>;
    last if !defined $n;
    chomp($on,$n);
    $n =~ s/_//g;          # erlaube 186_282.398_280_685
    $n = oct($n) if $n =~ /^0/; # erlaube 0xFF, 037, 0b1010
    if (looks_like_number($n)) {
        printf "$on mal 2 ist %g dezimal\n", 2*$n;
    } else {
        print "Das sieht für Perl nicht wie eine Zahl aus.\n";
    }
}
print "\nAuf Wiedersehen.\n";

```

Siehe auch

Die Syntax regulärer Ausdrücke in *perlre(1)* und in Kapitel 5 von *Programmieren mit Perl*; die *strtod(3)*-Manpage Ihres Systems; die *perlapi(1)*-Manpage; die Dokumentation für das CPAN-Modul `Regexp::Common`, inklusive der Manpage `Regexp::Common::number`; die Dokumentation der Standardmodule `POSIX` und `Scalar::Util` (auch in Kapitel 32 von *Programmieren mit Perl*).

2.2 Runden von Fließkommazahlen

Problem

Sie möchten eine Fließkommazahl auf eine bestimmte Zahl von Dezimalstellen runden. Dieses Problem resultiert aus den gleichen Ungenauigkeiten in der Darstellung, die auch die Prüfung auf Gleichheit schwierig machen (siehe Rezept 2.3), kommt aber auch in Situationen vor, in denen die Genauigkeit Ihrer Antwort aus Gründen der Lesbarkeit reduziert werden muss.

Lösung

Verwenden Sie die Perl-Funktion `sprintf` oder `printf`, wenn Sie nur etwas ausgeben wollen:

```
# Auf zwei Stellen runden
$rounded = sprintf("%.2f", $unrounded);
```

Oder verwenden Sie eine der anderen in der Diskussion beschriebenen Rundungsfunktionen.

Diskussion

Ob sichtbar oder nicht, irgendeine Form der Rundung ist bei der Arbeit mit Fließkommazahlen nahezu unvermeidlich. Sorgfältig definierte Standards (namentlich IEEE 754, der Standard für binäre Fließkomma-Arithmetik), gekoppelt mit vernünftigen Voreinstellungen in Perl, eliminieren solche Rundungsfehler (oder verdecken sie zumindest).

Tatsächlich ist die implizite Rundung bei Perl üblicherweise gut genug, um Überraschungen zu vermeiden. Meist ist es am besten, die Zahlen bis zur Ausgabe ungerundet zu lassen und dann mit `printf` oder `sprintf` selbst ein Format für eine explizite Rundung festzulegen, wenn Ihnen die Ausgabe von Perl nicht gefällt. Die Formate `%f`, `%e` und `%g` ermöglichen es Ihnen alle festzulegen, wie die Argumente zu runden sind. Das folgende Beispiel zeigt, wie sich die drei Formate verhalten. In allen Fällen verwenden wir ein zwölf Zeichen breites Feld, aber mit einer Genauigkeit von nicht mehr als vier Ziffern rechts vom Dezimalpunkt.

```
for $n ( 0.0000001, 10.1, 10.00001, 100000.1 ) {
    printf "%12.4e %12.4f %12.4g\n", $n, $n, $n;
}
```

Das Beispiel erzeugt die folgende Ausgabe:

```
1.0000e-07      0.0000      1e-07
1.0100e+01     10.1000     10.1
1.0000e+01     10.0000     10
1.0000e+05    100000.1000    1e+05
```

Wenn das alles wäre, dann wäre das Runden recht einfach. Sie würden einfach das gewünschte Ausgabeformat wählen und fertig.

Leider ist es nicht so einfach: Manchmal müssen Sie genauer darüber nachdenken, was Sie genau wollen und was eigentlich passiert. Wie in der Einführung erläutert, kann selbst eine einfache Zahl wie 10.1 oder 0.1 als binäre Fließkommazahl nur genähert werden. Die einzigen Dezimalzahlen, die als Fließkommazahlen *genau* dargestellt werden können, sind diejenigen, die wir als endliche Summe von Brüchen zusammenfassen können, deren Nenner alle die Basis 2 haben. Ein Beispiel:

```
$a = 0.625;          # 1/2 + 1/8
$b = 0.725;          # 725/1000 oder 29/40
printf "$_ ist %.30g\n", $_ for $a, $b;
```

gibt Folgendes aus:

```
0.625 ist 0.625
0.725 ist 0.724999999999999977795539507497
```

Die Zahl in `$a` kann binär genau dargestellt werden, die in `$b` nicht. Wenn Perl eine Fließkommazahl ausgeben soll, aber nichts über die zu verwendende Genauigkeit weiß (wie das für den interpolierten Wert von `$_` der Fall ist), dann wird die auszugebende Zahl auf so viele Stellen genau gerundet, wie es von der Maschine unterstützt wird. Typischerweise entspricht dies dem Format `%.15g`, was bei der Ausgabe die gleiche Nummer erzeugt, die Sie `$b` zugewiesen haben.

Üblicherweise ist der Rundungsfehler so klein, dass Sie ihn nicht bemerken, und wenn doch, dann können Sie die gewünschte Genauigkeit in der Ausgabe festlegen. Weil aber die zu Grunde liegende Näherung immer ein klein wenig von dem abweicht, was ein einfaches `print` zeigt, kann es zu unerwarteten Ergebnissen kommen. Während also zum Beispiel solche Zahlen wie 0.125 und 0.625 genau abgebildet werden können, ist das bei Zahlen wie 0.325 und 0.725 nicht der Fall. Nehmen wir an, Sie wollen auf zwei Dezimalstellen runden. Wird 0.325 zu 0.32 oder zu 0.33? Wird 0.725 zu 0.72 oder zu 0.73?

```
$a = 0.325;          # 1/2 + 1/8
$b = 0.725;          # 725/1000 oder 29/40
printf "%s ist %.2f oder %.30g\n", ($_) x 3 for $a, $b;
```

Das liefert:

```
0.325 ist 0.33 oder 0.325000000000000011102230246252
0.725 ist 0.72 oder 0.724999999999999977795539507497
```

Weil die Näherung für 0.325 etwas über diesem Wert liegt, wird auf 0.33 aufgerundet. Weil andererseits die Näherung für 0.725 tatsächlich etwas unter diesem Wert liegt, wird abgerundet, und das Ergebnis lautet 0.72.

Was passiert aber, wenn die Zahl genau dargestellt werden *kann*, wie etwa 1.5 oder 7.5, die einfach ganze Zahlen plus ein Halb darstellen? Die in diesem Fall verwendete Rundungsregel macht wahrscheinlich nicht das, was Sie in der Schule gelernt haben:

```
for $n (-4 .. +4) {
    $n += 0.5;
    printf "%.1f %2.0f\n", $n, $n;
}
```

Das erzeugt:

```
-3.5 -4
-2.5 -2
-1.5 -2
-0.5 -0
 0.5  0
 1.5  2
 2.5  2
 3.5  4
 4.5  4
```

Was hier passiert, spiegelt die Tatsache wider, dass die numerische Analyse nicht die Rundungsregel »Runde ab fünf auf«, sondern stattdessen »Runde zu einer geraden Zahl« bevorzugt. Auf diese Weise neigen die Rundungsfehler dazu, sich gegenseitig aufzuheben.

Drei nützliche Funktionen zur Rundung von Fließkommawerten in Integralzahlen sind `int`, `ceil` und `floor`. In Perl fest eingebaut, gibt `int` den integralen Teil der übergebenen Fließkommazahl zurück. Dies wird als »Runden zur Null« bezeichnet. Sie wird auch zur Umwandlung in ganze Zahlen verwendet, weil die Nachkommastellen ignoriert werden: Die Funktion `int` rundet bei positiven Zahlen ab und bei negativen auf. Die `floor`- und `ceil`-Funktionen des POSIX-Moduls ignorieren die Nachkommastellen ebenfalls, runden aber immer zum nächsten Integerwert auf bzw. ab, und zwar unabhängig vom Vorzeichen.

```
use POSIX qw(floor ceil);
printf "%8s %8s %8s %8s %8s\n",
    qw(Zahl gerade null ab auf);
for $n (-6 .. +6) {
    $n += 0.5;
    printf "%8g %8.0f %8s %8s %8s\n",
        $n, $n, int($n), floor($n), ceil($n);
}
```

Dieser Code erzeugt die folgende Tabelle. Jede Spaltenüberschrift zeigt, was passiert, wenn die Zahl in die angegebene Richtung gerundet wird.

Zahl	gerade	null	ab	auf
-5.5	-6	-5	-6	-5
-4.5	-4	-4	-5	-4
-3.5	-4	-3	-4	-3
-2.5	-2	-2	-3	-2
-1.5	-2	-1	-2	-1

-0.5	-0	0	-1	0
0.5	0	0	0	1
1.5	2	1	1	2
2.5	2	2	2	3
3.5	4	3	3	4
4.5	4	4	4	5
5.5	6	5	5	6
6.5	6	6	6	7

Wenn Sie die Spalten zusammenaddieren, werden Sie feststellen, dass Sie bei sehr unterschiedlichen Gesamtsummen landen:

6.5	6	6	0	13
-----	---	---	---	----

Das sagt Ihnen, dass Ihr Rundungsstil – also genau genommen die Wahl des Rundungsfehlers – einen enormen Einfluss auf das Endergebnis haben kann. Das ist ein Grund, warum Sie mit dem Runden bis zur Ausgabe warten sollten. Doch selbst dann sind einige Algorithmen für die Akkumulation von Rundungsfehlern anfälliger als andere. Bei besonders kritischen Anwendungen (zum Beispiel für Finanzberechnungen oder thermonukleare Raketen) werden umsichtige Programmierer eigene Rundungsfunktionen implementieren, anstatt sich auf die in den Computer integrierte Logik (zum Beispiel deren Fehlen) zu verlassen. (Ein gutes Buch zur numerischen Analysis ist ebenfalls zu empfehlen.)

Siehe auch

Die Funktionen `sprintf` und `int` in *perlfunc*(1) und in Kapitel 29 von *Programmieren mit Perl*; die Abschnitte zu `floor` und `ceil` in der Dokumentation des Standardmoduls `POSIX` (auch in Kapitel 32 von *Programmieren mit Perl*). Die `sprintf`-Technik führen wir in Rezept 2.3 ein.

2.3 Vergleich von Fließkommazahlen

Problem

Die Fließkomma-Arithmetik ist nicht präzise. Sie möchten zwei Fließkommazahlen vergleichen und wissen, ob sie, bezogen auf eine bestimmte Zahl von Dezimalstellen, gleich sind. Meistens *sollten* Sie die Gleichheit von Fließkommazahlen auf diese Weise prüfen.

Lösung

Verwenden Sie `sprintf`, um die Zahlen mit einer bestimmten Zahl von Dezimalstellen aufzubereiten, und vergleichen Sie die resultierenden Strings:

```
# equal(NUM1, NUM2, PRECISION) : gibt wahr zurück, wenn NUM1 und NUM2
# auf PRECISION Dezimalstellen genau sind.
sub equal {
    my ($A, $B, $dp) = @_ ;
    return sprintf("%.${dp}g", $A) eq sprintf("%.${dp}g", $B);
}
```

Alternativ können Sie die Zahlen als Integerwerte abspeichern und eine Dezimalstelle annehmen.

Diskussion

Sie benötigen die `equal`-Routine, weil die Fließkomma-Darstellung des Computers für die meisten realen Zahlen nur eine Näherung ist (wie wir es in der Einführung zu diesem Kapitel erläutert haben). Die normalen Ausgaberroutinen von Perl geben Zahlen auf etwa 15 Stellen gerundet aus, aber numerische Tests machen das nicht. Es kann also manchmal sein, dass Zahlen bei der Ausgabe (nach dem Runden) gleich aussehen, bei Tests (ohne Rundung) aber nicht.

Dieses Problem ist besonders bei Schleifen zu beachten, bei denen sich Rundungsfehler leise akkumulieren können. Zum Beispiel könnten Sie glauben, dass Sie eine Variable mit null initialisieren, zehnmal ein Zehntel aufaddieren und dann bei eins laden. Nun, das ist nicht der Fall, weil ein binär arbeitender Computer ein Zehntel nicht genau darstellen kann:

```
for ($num = $i = 0; $i < 10; $i++) { $num += 0.1 }
if ($num != 1) {
    printf "Seltsam, $num ist nicht 1, sondern %.45f\n", $num;
}
```

Das ergibt:

Seltsam, 1 ist nicht 1, sondern 0.9999999999999999888977697537484345957636833191

Die Interpolation von `$num` erfolgt über das (bei den meisten Systemen als Standard verwendete) Umwandlungsformat `%.15g`, sieht also wie eine 1 aus. Intern aber ist das nicht der Fall. Hätten Sie nur bis auf ein paar Dezimalstellen, zum Beispiel fünf, geprüft ...

```
!equal($num, 1, 5)
```

wäre alles in Ordnung gewesen.

Falls Sie eine feste Anzahl von Dezimalstellen besitzen, etwa bei monetären Werten, können Sie das Problem oftmals umgehen, indem Sie die Zahlen als Integerwerte abspeichern. 3,50 EUR als 350 statt als 3.5 abzuspeichern hebt die Notwendigkeit von Fließkommazahlen auf. Den Dezimalpunkt können Sie bei der Ausgabe wieder einfügen:

```
$lohn = 536;           # 5.36 EUR/Std
$woche = 40 * $lohn;  # 214.40 EUR
printf("Der Wochenlohn beträgt: %.2f EUR\n", $woche/100);
```

Der Wochenlohn beträgt: 214.40 EUR

Es macht nur selten Sinn, mehr als 15 Dezimalstellen zu vergleichen, weil Ihre Hardware sehr wahrscheinlich nur eine dementsprechende Genauigkeit aufweist.

Siehe auch

Die Funktion `sprintf` in *perlfunc(1)* und in Kapitel 29 von *Programmieren mit Perl*; den Abschnitt zu `$OFMT` in der *perlvar(1)*-Manpage und in Kapitel 28 von *Programmieren mit Perl*; die Dokumentation des Standardmoduls `Math::BigFloat` (auch in Kapitel 32 von *Programmieren mit Perl*); wir nutzen `sprintf` in Rezept 2.2; Band 2, Abschnitt 4.2.2 von *The Art of Computer Programming*.

2.4 Mit einer Reihe von Integerwerten arbeiten

Problem

Sie möchten eine Operation auf alle Integerwerte zwischen `X` und `Y` anwenden, etwa bei der Verarbeitung eines zusammenhängenden Bereichs eines Arrays oder wo auch immer Sie alle Zahlen² innerhalb eines Bereichs verarbeiten wollen.

Lösung

Verwenden Sie eine `for`-Schleife oder `..` zusammen mit einer `foreach`-Schleife:

```
foreach ($X .. $Y) {
    # $_ wird auf jeden Integerwert von X bis einschließlich Y gesetzt
}

foreach $i ($X .. $Y) {
    # $i wird auf jeden Integerwert von X bis einschließlich Y gesetzt
}

for ($i = $X; $i <= $Y; $i++) {
    # $i wird auf jeden Integerwert von X bis einschließlich Y gesetzt
}

for ($i = $X; $i <= $Y; $i += 7) {
    # $i wird in Siebener-Schritten auf jeden Integerwert von X bis Y gesetzt
}
```

Diskussion

Die ersten beiden Ansätze verwenden eine `foreach`-Schleife zusammen mit dem Konstrukt `$X .. $Y`, das eine Liste von Integerwerten zwischen `$X` und `$Y` erzeugt. Würden wir dieses Konstrukt einfach an ein Array zuweisen, dann würde das sehr viel Speicher verbrauchen, wenn `$X` und `$Y` weit auseinander liegen. Aber Perl erkennt die `foreach`-Schleife und verschwendet weder Zeit noch Arbeitsspeicher, um eine temporäre Liste anzulegen. Bei der Iteration über aufeinander folgende Integerwerte wird die `foreach`-Schleife schneller ausgeführt als die gleichwertige `for`-Schleife.

² Okay, ganze Zahlen. Alle reelle Zahlen zu finden ist schwer, fragen Sie Cantor.

Ein weiterer Unterschied zwischen den beiden Konstrukten besteht darin, dass die `foreach`-Schleife die Schleifenvariable im Body implizit lokalisiert, was bei der `for`-Schleife nicht der Fall ist. Das bedeutet, dass die Schleifenvariable nach Abschluss der `for`-Schleife den Wert der letzten Iteration enthält. Im Fall der `foreach`-Schleife ist dieser Wert hingegen nicht zugänglich, und die Variable enthält, was immer sie (wenn überhaupt) vor dem Eintritt in die Schleife enthielt. Sie können allerdings eine lexikalisch beschränkte Variable als Schleifenvariable verwenden:

```
foreach my $i ($X .. $Y) { ... }
for (my $i=$X; $i <= $Y; $i++) { ... }
```

Der nachfolgende Code verdeutlicht jede Technik. Dabei geben wir nur die von uns erzeugten Zahlen aus:

```
print "Babies mit: ";
foreach (0 .. 2) {
    print "$_ ";
}
print "\n";

print "Krabbelgruppe mit: ";
foreach $i (3 .. 4) {
    print "$i ";
}
print "\n";

print "Kinder mit: ";
for ($i = 5; $i <= 12; $i++) {
    print "$i ";
}
print "\n";

Babies mit: 0 1 2
Krabbelgruppe mit: 3 4
Kinder mit: 5 6 7 8 9 10 11 12
```

Siehe auch

Die `for`- und `foreach`-Operatoren in *perlsyn*(1) und die Abschnitte »for-Schleifen« und »foreach-Schleifen« in Kapitel 4 von *Programmieren mit Perl* (S. 120 und 122).

2.5 Mit römischen Zahlwörtern arbeiten

Problem

Sie möchten reguläre Zahlen in römische Zahlwörter umwandeln und umgekehrt. Sie benötigen dies bei Aufzählungen in Skizzen, Seitennummern im Vorwort und Copyright-Vermerken von Filmen.

Lösung

Verwenden Sie das Roman-Modul aus dem CPAN:

```
use Roman;
$roman = roman($arabic);           # Umwandlung in römische Zahlwörter
$arabic = arabic($roman) if isroman($roman); # Umwandlung römischer Zahlwörter
```

Diskussion

Das Roman-Modul stellt `Roman` und `roman` zur Verfügung, mit denen arabische (»normale«) Zahlen in die entsprechenden römischen Zahlwörter umgewandelt werden können. `Roman` erzeugt Großbuchstaben, während `roman` Kleinbuchstaben generiert.

Das Modul arbeitet nur mit römischen Zahlen von 1 bis einschließlich 3999. Die Römer haben weder negative Zahlen noch die Null dargestellt, und die Zahl 5000 (mit der auch 4000 dargestellt wird) verwendet ein Symbol außerhalb des ASCII-Zeichensatzes.

```
use Roman;
$roman_fifteen = roman(15);           # "xv"
print "Das römische Zahlwort für 15 ist $roman_fifteen\n";
$arabic_fifteen = arabic($roman_fifteen);
print "Umgekehrt entspricht $roman_fifteen der Zahl $arabic_fifteen\n";
```

***Das römische Zahlwort für 15 ist xv
Umgekehrt entspricht xv der Zahl 15***

Um das aktuelle Jahr auszugeben:

```
use Time::localtime;
use Roman;
printf "Wir zählen das Jahr %s\n", Roman(1900 + localtime->year);
```

Wir zählen das Jahr MMIII

Falls Ihnen Unicode-Fonts zur Verfügung stehen, dann finden Sie an den Codepunkten U+2160 bis U+2183 römische Zahlwörter, einschließlich derer unter den typischen ASCII-Werten.

```
use charnames ":full";
print "2003 ist ", "\N{ROMAN NUMERAL ONE THOUSAND}" x 2, "\N{ROMAN NUMERAL THREE}\n";
2003 ist MMIII
```

Allerdings besitzt das Roman-Modul bisher keine Möglichkeit, diese Zeichen zu nutzen.

Ob Sie es glauben oder nicht, es gibt sogar ein CPAN-Modul, mit dessen Hilfe Sie römische Zahlwörter beim Rechnen benutzen können.

```
use Math::Roman qw(roman);
print $a = roman('I'); # I
print $a += 2000;      # MMI
print $a -= "III";    # MCMXCVIII
print $a -= "MCM";    # XCVIII
```

Siehe auch

Den Artikel »Mathematics, History of« in der *Encyclopaedia Britannica*; die Dokumentation der CPAN-Module `Roman` und `Math::Roman`; Rezept 6.23.

2.6 Zufallszahlen erzeugen

Problem

Sie möchten Zufallszahlen in einem bestimmten Wertebereich erzeugen, um beispielsweise zufällig einen Arrayindex auszuwählen, das Würfeln in einem Spiel zu simulieren oder um ein zufälliges Passwort zu erzeugen.

Lösung

Verwenden Sie die Perl-Funktion `rand`:

```
$random = int( rand( $Y-$X+1 ) ) + $X;
```

Diskussion

Der folgende Code generiert einen zufälligen Integerwert zwischen 25 und 75 (einschließlich):

```
$random = int( rand(51) ) + 25;  
print "$random\n";
```

Die `rand`-Funktion gibt eine reelle Zahl zurück, die von (einschließlich) 0 bis zu (aber nicht einschließlich) dem übergebenen Argument reichen kann. Wir übergeben als Argument die 51, so dass die erzeugte Zahl zwischen null und darüber liegt, aber niemals 51 oder mehr ergibt. Wir nutzen dann den ganzzahligen Teil, um eine Zahl zwischen 0 und 50 (einschließlich) zu erhalten (50.99999... wird von `int` als 50 zurückgegeben). Danach addieren wir 25 auf, um eine Zahl zwischen 25 und 75 zu erzeugen.

Eine typische Anwendung ist die zufällige Auswahl eines Elements aus einem Array:

```
$elt = $array[ rand @array ];
```

Das entspricht dem folgenden Code:

```
$elt = $array[ int( rand(0+@array) ) ];
```

Weil der `rand`-Prototyp nur ein Argument akzeptiert, erzwingt `rand` implizit einen skalaren Kontext für dieses Argument, das bei einem benannten Array die Anzahl der Elemente im Array zurückliefert. Die Funktion gibt dann eine Fließkommazahl zurück, die kleiner ist als das Argument und größer oder gleich null. Eine als Arrayindex genutzte Fließkommazahl wird automatisch in eine Integerzahl umgewandelt (wobei zu null gerundet wird), was letztendlich ein gleichmäßig verteiltes, zufällig gewähltes Arrayelement ergibt, das an `$elt` zugewiesen wird.

Die Erzeugung eines zufälligen Passworts aus einer Reihe von Zeichen ist ebenso einfach:

```
@chars = ( "A" .. "Z", "a" .. "z", 0 .. 9, qw(! @ $ % ^ & * ) );  
$password = join("", @chars[ map { rand @chars } ( 1 .. 8 ) ] );
```

Wir nutzen `map`, um acht zufällige Indizes auf `@chars` zu generieren, extrahieren die entsprechenden Zeichen mit einem `Slice` und vereinen sie mittels `join` zu einem zufälligen Passwort. Das ist keine *gute* Zufallszahl, weil ihre Sicherheit von der Initialisierung des (Zufallszahlen-)Generators abhängt, die (in älteren Perl-Versionen) wiederum von der Zeit abhängt, zu der das Programm gestartet wurde. In Rezept 2.7 finden Sie eine bessere Möglichkeit, den Startwert des Zufallszahlengenerators einzustellen.

Siehe auch

Die Funktionen `int`, `rand`, `map` und `join` in *perlfunc*(1) und in Kapitel 29 von *Programmieren mit Perl*. Wir erläutern Zufallszahlen weiterhin in den Rezepten 2.7, 2.8 und 2.9 und nutzen sie in Rezept 1.13.

2.7 Reproduzierbare Folgen von Zufallszahlen erzeugen

Problem

Bei jeder Ausführung Ihres Programms erhalten Sie eine andere Folge von (Pseudo-)Zufallszahlen. Sie benötigen aber eine reproduzierbare Folge (was zum Beispiel für eine Simulation nützlich ist), weshalb Perl jedes Mal die gleichen Zufallszahlen erzeugen soll.

Lösung

Verwenden Sie die Perl-Funktion `srand`:

```
srand AUSDR; # Verwenden Sie eine Konstante für reproduzierbare Sequenzen
```

Diskussion

Zufallszahlen zu generieren ist schwierig. Das Beste, was Computer erreichen können, ohne spezielle Hardware zu verwenden, ist die Generierung von »Pseudo-Zufallszahlen«, die innerhalb eines Wertebereichs gleichmäßig verteilt sind. Diese werden über eine mathematische Formel erzeugt, was bedeutet, dass beim gleichen Startwert (dem sog. *seed*) zwei Programme die gleichen Pseudo-Zufallszahlen erzeugen.

Die `srand`-Funktion erzeugt einen neuen Startwert für den Pseudo-Zufallszahlengenerator. Wird ihr ein Argument übergeben, verwendet sie diese Zahl als Startwert. Ohne Argument verwendet `srand` einen Startwert, der nur recht schwer zu ermitteln ist.

Wenn Sie `rand` aufrufen, ohne vorher `srand` aufgerufen zu haben, ruft Perl `srand` für Sie auf und wählt dabei einen »guten« Startwert. Auf diese Weise erhalten Sie bei jeder Ausführung Ihres Programms eine andere Folge von Zufallszahlen. Ältere Perl-Versionen

haben `srand` nicht aufgerufen, das heißt, das Programm hat immer die gleiche Sequenz von Zufallszahlen erzeugt. Bestimmte Arten von Programmen wünschen aber keine andere Folge von Zufallszahlen, sondern immer die gleiche Folge. Wenn Sie dieses Verhalten benötigen, rufen Sie `srand` selbst auf und stellen einen entsprechenden »Seed-Wert« zur Verfügung.

```
srand( 42 ); # wählen Sie einen festen Startwert
```

Rufen Sie `srand` innerhalb eines Programms nicht erneut auf, da die Sequenz dann an diesem Punkt wieder beginnt (es sei denn, Sie wollen genau das).

Dass Perl versucht, einen guten Startwert bereitzustellen, garantiert nicht, dass die erzeugten Zahlen, kryptographisch gesehen, vor cleveren Hackern geschützt sind. Bücher zur Kryptographie sind üblicherweise gute Quellen für kryptographisch sichere Zufallszahlengeneratoren.

Siehe auch

Die `srand`-Funktion in *perlfunc*(1) und in Kapitel 29 von *Programmieren mit Perl*; die Rezepte 2.6 und 2.8; Bruce Schneiers exzellentes Buch *Angewandte Kryptographie* (Addison-Wesley).

2.8 Noch zufälliger Zahlen

Problem

Sie möchten Zahlen generieren, die noch zufälliger sind als die von Perl erzeugten Zufallszahlen. Beschränkungen der »Seed-Werte« des Zufallszahlengenerators Ihrer C-Bibliothek können manchmal zu Problemen führen. Die Sequenz der erzeugten Zufallszahlen kann sich für manche Anwendungen zu schnell wiederholen.

Lösung

Verwenden Sie einen anderen Zufallszahlengenerator, beispielsweise einen, den die CPAN-Module `Math::Random` und `Math::TrulyRandom` zur Verfügung stellen:

```
use Math::TrulyRandom;
$random = truly_random_value();

use Math::Random;
$random = random_uniform();
```

Diskussion

Perl versucht während der Kompilierung, die beste verfügbare C-Bibliothek zur Generierung von Pseudo-Zufallszahlen zu nutzen, und sucht dabei nach `rand(3)`, `random(3)` und `drand48(3)`. (Sie können dies aber auch von Hand festlegen.) Die Standard-Bibliotheks-

funktionen sind recht gut, aber einige veraltete Implementierungen der `rand`-Funktion geben nur 16-Bit-Werte zurück oder weisen andere algorithmische Schwächen auf und könnten daher für Ihre Zwecke nicht »zufällig« genug sein.

Das Modul `Math::TrulyRandom` nutzt Unzulänglichkeiten des Timers Ihres Systems zur Generierung von Zufallszahlen. Das dauert eine Weile, weshalb sich dieses Modul nicht zur Generierung vieler Zufallszahlen eignet.

Das Modul `Math::Random` verwendet die `randlib`-Bibliothek zur Generierung von Zufallszahlen. Es umfasst auch eine große Anzahl verwandter Funktionen zur Generierung von Zufallszahlen entsprechend bestimmter Verteilungen wie Binomial-, Poisson- und Exponential-Verteilungen.

Siehe auch

Die Funktionen `srand` und `rand` in *perlfunc*(1) und in Kapitel 29 von *Programmieren mit Perl*; die Rezepte 2.6 und 2.7; die Dokumentation der CPAN-Module `Math::Random` und `Math::TrulyRandom`.

2.9 Generierung gewichteter Zufallszahlen

Problem

Sie möchten Zufallszahlen erzeugen, bei denen die Verteilung nicht gleichmäßig ist, das heißt, bei denen die Wahrscheinlichkeiten nicht gleich sind. Beispielsweise könnten Sie ein zufälliges Banner auf einer Webseite ausgeben wollen, wobei relative Gewichtungen vorgeben, wie häufig ein Banner zu erscheinen hat. Alternativ könnten Sie das Verhalten einer Normalverteilung (der Glockenkurve nach Gauß) simulieren wollen.

Lösung

Wenn Sie eine Zufallszahl benötigen, die entsprechend einer bestimmten Funktion verteilt ist (zum Beispiel eine Gaußsche Normalverteilung), müssen Sie die entsprechende Funktion bzw. den richtigen Algorithmus in einem Statistikbuch nachschlagen. Die folgende Subroutine generiert normal verteilte Zufallszahlen mit einer Standardabweichung von 1 und einem Mittelwert von 0:

```
sub gaussian_rand {
    my ($u1, $u2); # Gleichmäßig verteilte Zufallszahlen
    my $w;        # Varianz, dann Gewichtung
    my ($g1, $g2); # Normalverteilte Zahlen

    do {
        $u1 = 2 * rand() - 1;
        $u2 = 2 * rand() - 1;
        $w = $u1*$u1 + $u2*$u2;
    } while ($w >= 1 || $w == 0);
}
```

```

    $w = sqrt( (-2 * log($w)) / $w );
    $g2 = $u1 * $w;
    $g1 = $u2 * $w;
    # Falls gewünscht, beide zurückgeben, sonst nur einen Wert
    return wantarray ? ($g1, $g2) : $g1;
}

```

Wenn Sie eine Liste der Gewichtungen und Werte besitzen, aus denen Sie zufällig einen Wert wählen wollen, können Sie das folgende, aus zwei Schritten bestehende Verfahren anwenden: Zuerst wandeln Sie die Gewichtungen mit `weight_to_dist` in eine Wahrscheinlichkeitsverteilung um, und dann verwenden Sie diese Verteilung mit `weighted_rand`, um zufällig einen Wert auszuwählen:

```

# weight_to_dist: erwartet einen Hash, der den Schlüssel auf die
# Gewichtung abbildet, und gibt einen Hash zurück,
# der den Schlüssel auf die Wahrscheinlichkeit abbildet.
sub weight_to_dist {
    my %weights = @_;
    my %dist = ();
    my $total = 0;
    my ($key, $weight);
    local $_;

    foreach (values %weights) {
        $total += $_;
    }

    while ( ($key, $weight) = each %weights ) {
        $dist{$key} = $weight/$total;
    }

    return %dist;
}

# weighted_rand: erwartet einen Hash, der Schlüssel auf Wahrscheinlichkeit
# abbildet, und liefert das entsprechende Element zurück
sub weighted_rand {
    my %dist = @_;
    my ($key, $weight);

    while (1) {
        # zur Vermeidung von Fließkomma-Ungenauigkeiten
        my $rand = rand;
        while ( ($key, $weight) = each %dist ) {
            return $key if ($rand -= $weight) < 0;
        }
    }
}

```

Diskussion

Die Funktion `gaussian_rand` implementiert die *Box-Muller*-Polarmethode, um zwei unabhängige, gleichmäßig verteilte Zufallszahlen zwischen 0 und 1 (wie sie `rand` generiert) in

zwei Zahlen mit einem Mittelwert von 0 und einer Standardabweichung von 1 (d.h. einer Normalverteilung) umzuwandeln. Um Zufallszahlen mit anderen Mittelwerten und Standardverteilungen zu erzeugen, multiplizieren Sie die Ausgabe von `gaussian_rand` mit der neuen Verteilung und addieren den neuen Mittelwert:

```
# gaussian_rand, wie bereits gezeigt
$mean = 25;
$sdev = 2;
$salary = gaussian_rand() * $sdev + $mean;
printf("Sie wurden eingekauft zu \$.2f\n", $salary);
```

Das Modul `Math::Random` implementiert dies und andere Distributionen für Sie:

```
use Math::Random qw(random_normal);
$salary = random_normal(1, $mean, $sdev);
```

Die Funktion `weighted_rand` pickt sich eine Zufallszahl zwischen 0 und 1 heraus. Danach verwendet sie die von `weight_to_dist` generierten Wahrscheinlichkeiten, um zu bestimmen, welchem Element die Zufallszahl entspricht. Aufgrund der Launenhaftigkeit von Fließkomma-Darstellungen können akkumulierte Darstellungsfehler dazu führen, dass wir kein Element finden. Aus diesem Grund fangen wir den Code in einer `while`-Schleife ab, um eine neue Zufallszahl zu wählen und es erneut zu versuchen.

Zudem bietet das CPAN-Modul `Math::Random` Funktionen an, mit denen Zufallszahlen unterschiedlicher Verteilungen zurückgegeben werden.

Siehe auch

Die Funktion `rand` in *perlfunc(1)* und in Kapitel 29 von *Programmieren mit Perl*; Rezept 2.6; die Dokumentation des CPAN-Moduls `Math::Random`.

2.10 Trigonometrie im Gradmaß statt im Bogenmaß

Problem

Sie möchten Ihre trigonometrischen Routinen mit Gradangaben arbeiten lassen statt mit dem bei Perl üblichen Bogenmaß.

Lösung

Wandeln Sie Bogenmaße und Gradangaben um (2π entspricht 360 Grad):

```
use constant PI => (4 * atan2 (1, 1));

sub deg2rad {
    my $degrees = shift;
    return ($degrees / 180) * PI;
}
```

```

sub rad2deg {
    my $radians = shift;
    return ($radians / PI) * 180;
}

```

Alternativ können Sie das Standardmodul `Math::Trig` verwenden:

```

use Math::Trig;

$radians = deg2rad($degrees);
$degrees = rad2deg($radians);

```

Diskussion

Wenn Sie viel mit trigonometrischen Funktionen arbeiten, sollten Sie sich die Standardmodule `Math::Trig` oder `POSIX` näher ansehen. Sie enthalten wesentlich mehr trigonometrische Funktionen als der Perl-Kern. Ansonsten verwenden Sie die in der ersten Lösung definierten Funktionen `rad2deg` und `deg2rad`. Der Wert für π ist nicht direkt in Perl integriert, kann aber so genau berechnet werden, wie Ihre Fließkomma-Hardware das erlaubt. In der Lösung ist die `PI`-Funktion eine mit `use constant` erzeugte Konstante. Statt uns merken zu müssen, dass π den Wert 3.14159265358979 (oder so) hat, verwenden wir den fest eingebauten Funktionsaufruf (der bei der Kompilierung aufgelöst wird), der es uns nicht nur erspart, sich einen langen String aus Ziffern merken zu müssen, sondern auch garantiert die Genauigkeit bietet, die die Plattform unterstützt.

Wenn Sie den Sinus in Grad wünschen, verwenden Sie Folgendes:

```

# deg2rad und rad2deg wie oben definiert oder aus Math::Trig
sub degree_sine {
    my $degrees = shift;
    my $radians = deg2rad($degrees);
    my $result = sin($radians);

    return $result;
}

```

Siehe auch

Die Funktionen `sin`, `cos` und `atan2` in *perlfunc(1)* und in Kapitel 29 von *Programmieren mit Perl*; die Dokumentation der Standardmodule `POSIX` und `Math::Trig` (auch in Kapitel 32 von *Programmieren mit Perl*).

2.11 Berechnung anderer trigonometrischer Funktionen

Problem

Sie möchten Werte für trigonometrische Funktionen wie Sinus, Tangens oder Arkus-Kosinus berechnen.

Lösung

Perl stellt standardmäßig nur die Funktionen `sin`, `cos` und `atan2` zur Verfügung. Aus diesen können Sie `tan` und alle anderen Funktionen ableiten (wenn Sie mit esoterischen trigonometrischen Funktionen vertraut sind):

```
sub tan {
    my $theta = shift;

    return sin($theta)/cos($theta);
}
```

Das POSIX-Modul stellt eine größere Auswahl trigonometrischer Funktionen zur Verfügung:

```
use POSIX;

$y = acos(3.7);
```

Das Standardmodul `Math::Trig` stellt einen vollständigen Satz von Funktionen bereit und unterstützt auch komplexe Zahlen:

```
use Math::Trig;

$y = acos(3.7);
```

Diskussion

Die `tan`-Funktion führt zu einer Division-durch-null-Ausnahme, wenn `$theta` die Werte $\pi/2$, $3\pi/2$ und so weiter annimmt, weil der Kosinus dieser Werte 0 ist. Entsprechend können `tan` und viele andere Funktionen aus `Math::Trig` den gleichen Fehler generieren. Um ihn abzufangen, verwenden Sie `eval`:

```
eval {
    $y = tan($pi/2);
} or return undef;
```

Siehe auch

Die Funktionen `sin`, `cos` und `atan2` in *perlfunc(1)* und in Kapitel 29 von *Programmieren mit Perl*; die Dokumentation des Standardmoduls `Math::Trig`; Trigonometrie im Kontext imaginärer Zahlen wird in Rezept 2.14 besprochen; die Nutzung von `eval` zum Abfangen von Ausnahmen wird in Rezept 10.12 erläutert.

2.12 Logarithmen mit unterschiedlicher Basis

Problem

Sie möchten Logarithmen mit unterschiedlicher Basis verwenden.

Lösung

Für Logarithmen zur Basis e können Sie die integrierte `log`-Funktion verwenden:

```
$log_e = log(VALUE);
```

Für Logarithmen zur Basis 10 können Sie die `log10`-Funktion des POSIX-Moduls verwenden:

```
use POSIX qw(log10);
$log_10 = log10(VALUE);
```

Für andere Basen müssen Sie die mathematische Identität verwenden:

$$\log_n(x) = \frac{\log_e(x)}{\log_e(n)}$$

wobei x die Zahl ist, deren Logarithmus Sie benötigen. n ist die gewünschte Basis, und e ist die natürliche Basis des Logarithmus.

```
sub log_base {
    my ($base, $value) = @_ ;
    return log($value)/log($base);
}
```

Diskussion

Die Funktion `log_base` lässt Sie Logarithmen zu einer beliebigen Basis berechnen. Wenn Sie die Basis im Voraus kennen, ist es effizienter, den Logarithmus der Basis zwischenspeichern, anstatt ihn jedes Mal neu zu berechnen.

```
# log_base wie bereits definiert
$answer = log_base(10, 10_000);
print "log10(10,000) = $answer\n";
log10(10,000) = 4
```

Das Modul `Math::Complex` übernimmt diese Zwischenablage über seine `logn()`-Routine für Sie, so dass Sie Folgendes schreiben können:

```
use Math::Complex;
printf "log2(1024) = %lf\n", logn(1024, 2); # Achten Sie auf die Reihenfolge der Argumente!
log2(1024) = 10.000000
```

Dies ist möglich, obwohl hier keine komplexe Zahl im Spiel ist. Diese Variante ist nicht sehr effizient, aber es gibt Pläne, `Math::Complex` der Geschwindigkeit halber in C umzuschreiben.

Siehe auch

Die Funktion `log` in *perlfunc(1)* und in Kapitel 29 von *Programmieren mit Perl*; die Dokumentation der Standardmodule `POSIX` und `Math::Complex` (auch in Kapitel 32 von *Programmieren mit Perl*).

2.13 Matrizen multiplizieren

Problem

Sie möchten ein Paar zweidimensionaler Arrays multiplizieren. Mathematiker und Ingenieure benötigen das oft.

Lösung

Verwenden Sie das im CPAN verfügbare PDL-Modul. PDL steht für *Perl Data Language* und bietet schnellen Zugriff auf kompakte Matrizen und mathematische Funktionen:

```
use PDL;
# $a und $b sind beides pdl-Objekte
$c = $a x $b;
```

Alternativ können Sie den Algorithmus zur Matrix-Multiplikation auf Ihr zweidimensionales Array anwenden:

```
sub mmult {
    my ($m1,$m2) = @_ ;
    my ($m1rows,$m1cols) = matdim($m1);
    my ($m2rows,$m2cols) = matdim($m2);

    unless ($m1cols == $m2rows) { # Ausnahme auslösen
        die "Indexfehler: Matrizen stimmen nicht überein: $m1cols != $m2rows";
    }

    my $result = [];
    my ($i, $j, $k);

    for $i (range($m1rows)) {
        for $j (range($m2cols)) {
            for $k (range($m1cols)) {
                $result->[$i][$j] += $m1->[$i][$k] * $m2->[$k][$j];
            }
        }
    }
    return $result;
}

sub range { 0 .. ($_[0] - 1) }

sub veclen {
    my $ary_ref = $_[0];
    my $type = ref $ary_ref;
    if ($type ne "ARRAY") { die "$type ist ungültige Arrayreferenz für $ary_ref" }
    return scalar @$ary_ref;
}

sub matdim {
    my $matrix = $_[0];
```

```

    my $rows = veclen($matrix);
    my $cols = veclen($matrix->[0]);
    return ($rows, $cols);
}

```

Diskussion

Wenn Sie die PDL-Bibliothek installiert haben, können Sie die sehr schnellen Manipulationsmöglichkeiten für Zahlen nutzen. Das verlangt wesentlich weniger Speicher und CPU-Zeit als die Perl-Funktionen zur Arrayverarbeitung. Bei der Verwendung von PDL-Objekten werden viele numerische Operatoren (wie + und *) überladen. Sie arbeiten auf der Basis von Elementen (zum Beispiel ist * der Operator für die so genannte *skalare Multiplikation*). Für eine echte Matrixmultiplikation müssen Sie den überladenen Operator x verwenden.

```

use PDL;

$a = pdl [
    [ 3, 2, 3 ],
    [ 5, 9, 8 ],
];

$b = pdl [
    [ 4, 7 ],
    [ 9, 3 ],
    [ 8, 1 ],
];

$c = $a x $b; # x ist überladen

```

Wenn Sie die PDL-Bibliothek nicht besitzen oder sie für ein kleines Problem nicht installieren wollen, können Sie die Arbeit immer noch auf die althergebrachte Weise erledigen.

```

# mmult() und andere Subroutinen, wie bereits gezeigt

$x = [
    [ 3, 2, 3 ],
    [ 5, 9, 8 ],
];

$y = [
    [ 4, 7 ],
    [ 9, 3 ],
    [ 8, 1 ],
];

$z = mmult($x, $y);

```

Siehe auch

Die Dokumentation des CPAN-Moduls PDL.

2.14 Verwendung komplexer Zahlen

Problem

Ihre Anwendung muss komplexe Zahlen verarbeiten können. Diese werden häufig im Ingenieurwesen, in der Wissenschaft und in der Mathematik benötigt.

Lösung

Entweder Sie verwalten die realen und imaginären Komponenten selbst:

```
# $c = $a * $b manuell
$c_real = ( $a_real * $b_real ) - ( $a_imaginary * $b_imaginary );
$c_imaginary = ( $a_real * $b_imaginary ) + ( $b_real * $a_imaginary );
```

Oder Sie verwenden das Modul `Math::Complex` (das Bestandteil der Perl-Standarddistribution ist):

```
# $c = $a * $b mit Math::Complex
use Math::Complex;
$c = $a * $b;
```

Diskussion

Dies ist die manuelle Multiplikation von $3+5i$ und $2-2i$:

```
$a_real = 3; $a_imaginary = 5;          # 3 + 5i;
$b_real = 2; $b_imaginary = -2;        # 2 - 2i;
$c_real = ( $a_real * $b_real ) - ( $a_imaginary * $b_imaginary );
$c_imaginary = ( $a_real * $b_imaginary ) + ( $b_real * $a_imaginary );
print "c = ${c_real}+${c_imaginary}i\n";
```

$c = 16+4i$

Mit `Math::Complex` sieht die Berechnung so aus:

```
use Math::Complex;
$a = Math::Complex->new(3,5);          # oder Math::Complex->new(3,5);
$b = Math::Complex->new(2,-2);
$c = $a * $b;
print "c = $c\n";
```

$c = 16+4i$

Sie können komplexe Zahlen über den Konstruktor `cplx` oder über die exportierte Konstante `i` definieren:

```
use Math::Complex;
$c = cplx(3,5) * cplx(2,-2);          # einfacher fürs Auge
$d = 3 + 4*i;                          # 3 + 4i
printf "sqrt($d) = %s\n", sqrt($d);
```

$\text{sqrt}(3+4i) = 2+i$

Das Modul `Math::Trig` verwendet intern das Modul `Math::Complex`, weil einige Funktionen aus der realen Achse in die komplexe Ebene ausbrechen können – beispielsweise der inverse Sinus von 2.

Siehe auch

Die Dokumentation des Standardmoduls `Math::Complex` (auch in Kapitel 32 von *Programmieren mit Perl*).

2.15 Umwandlung von Binär-, Oktal- und Hexadezimalzahlen

Problem

Sie möchten einen String (zum Beispiel `"0b10110"`, `"0x55"` oder `"0755"`), der eine Binär-, Oktal- oder Hexadezimalzahl enthält, in die entsprechende Zahl umwandeln.

Perl versteht Zahlen in Binär- (Basis 2), Oktal- (Basis 8) und Hexadezimalschreibweise (Basis 16) nur, wenn sie innerhalb des Programms als Literale erscheinen. Werden sie als Dateien eingelesen – wie etwa beim Auslesen aus Dateien oder Umgebungsvariablen oder wenn sie als Kommandozeilenargumente übergeben werden –, findet keine automatische Umwandlung statt.

Lösung

Verwenden Sie die Perl-Funktion `hex` für hexadezimale Strings wie `"2e"` oder `"0x2e"`:

```
$number = hex($hexadezimal);      # nur hexadezimal ("2e" wird 47)
```

Verwenden Sie die `oct`-Funktion für einen hexadezimalen String wie `"0x2e"`, einen oktalen String wie `"047"` oder einen Binärstring wie `"0b101110"`:

```
$number = oct($hexadecimal);      # "0x2e" wird 47  
$number = oct($octal);            # "057" wird 47  
$number = oct($binary);           # "0b101110" wird 47
```

Diskussion

Die `oct`-Funktion wandelt Oktalzahlen mit oder ohne führende `"0"` um; zum Beispiel `"0350"` oder `"350"`. Trotz seines Namens konvertiert `oct` nicht nur Oktalzahlen: Sie wandelt auch hexadezimale Zahlen (`"0x350"`) um, wenn diesen ein `"0x"` vorangestellt ist, und binäre Zahlen (`"0b101010"`), wenn diesen ein `"0b"` vorangestellt ist. Die `hex`-Funktion wandelt hingegen nur hexadezimale Zahlen mit oder ohne führendes `"0x"` um (`"0x255"`, `"3A"`, `"ff"` oder `"deadbeef"`). Buchstaben dürfen dabei groß- und kleingeschrieben werden.

Nachfolgend ein Beispiel, das einen Integerwert in Dezimal-, Binär-, Oktal- oder Hexadezimalschreibweise einliest und diesen Integerwert zu jeder der vier Basen ausgibt. Beginnt

die Zahl mit einer 0, wird die oct-Funktion verwendet, um Binär-, Oktal- und Hexadezimalzahlen umzuwandeln. Es nutzt dann printf, um die Konvertierungen in alle vier Basen durchzuführen.

```
print "Geben Sie einen Integerwert in Dezimal-, Binär-, Oktal- oder Hex-
Schreibweise ein: ";
$num = <STDIN>;
chomp $num;
exit unless defined $num;
$num = oct($num) if $num =~ /^0/;      # erkennt 077 0b10 0x20
printf "%d %#x %#o %#b\n", ($num) x 4;
```

Das #-Symbol zwischen dem Prozentzeichen und den drei nicht-dezimalen Basen sorgt dafür, dass printf eine Ausgabe erzeugt, die die Basis des Integerwertes angibt. Geben Sie beispielsweise die Zahl 255 ein, würde die Ausgabe wie folgt aussehen:

```
255 0xff 0377 0b11111111
```

Ohne das #-Zeichen würden Sie Folgendes erhalten:

```
255 ff 377 11111111
```

Der folgende Code wandelt Unix-Zugriffsrechte um.

```
print "Geben Sie die Zugriffsrechte als Oktalzahl ein: ";
$permissions = <STDIN>;
die "Abbruch ...\n" unless defined $permissions;
chomp $permissions;
$permissions = oct($permissions); # Rechte immer oktal
print "Der Dezimalwert ist $permissions\n";
```

Siehe auch

Den Abschnitt »Scalar Value Constructors« in *perldata*(1) und den Abschnitt »Numerische Literale« in Kapitel 2 von *Programmieren in Perl* (S. 62); die Funktionen oct und hex in *perlfunc*(1) und in Kapitel 29 von *Programmieren in Perl*.

2.16 Punkte in Zahlen einfügen

Problem

Sie möchten eine Zahl mit Punkten an den richtigen Stellen ausgeben. Lange Zahlen (zum Beispiel in Berichten) werden übersichtlicher, wenn sie auf diese Weise dargestellt sind.

Lösung

Kehren Sie den String mittels reverse um. Auf diese Weise können Sie das Backtracking nutzen, um eine Substitution bei den Nachkommastellen der Zahl zu vermeiden. Verwenden Sie dann einen regulären Ausdruck, um die Positionen zu ermitteln, an denen

Punkte notwendig sind, und fügen Sie diese dann ein. Schließlich kehren Sie den String wieder in seine ursprüngliche Form um.

```
sub commify {
    my $text = reverse $_[0];
    $text =~ s/(\d\d\d)(?=\d)(?!*\d*\.)/$1,/g;
    return scalar reverse $text;
}
```

Diskussion

Bei regulären Ausdrücken ist es wesentlich einfacher, die Arbeit von vorne zu beginnen statt von hinten. Aus diesem Grund kehren wir den String mit `reverse` um und ändern eine Kleinigkeit in dem Algorithmus, der nach jeweils drei Ziffern immer einen Punkt einfügt. Nach dem Einfügen der Punkte kehren wir den String wieder um und geben ihn zurück. Weil `reverse` empfindlich auf seinen impliziten Rückkehr-Kontext reagiert, erzwingen wir einen skalaren Kontext.

Diese Funktion kann leicht an die Verwendung von Kommata (die in vielen Ländern, zum Beispiel in den USA, verbreitet sind) anstelle von Punkten angepasst werden.

Nachfolgend sehen Sie `commify` in Aktion:

```
# ein vernünftiger Web-Counter :-)
use Math::TrulyRandom;
$hits = truly_random_value();      # negative Hits!
$output = "Diese Webseite hatte letzten Monat $hits Zugriffe.\n";
print commify($output);
Diese Webseite hatte letzten Monat -1.740.525.205 Zugriffe.
```

Siehe auch

`perllocale(1)`; die `reverse`-Funktion in `perlfunc(1)` und in Kapitel 29 von *Programmieren mit Perl*; den Abschnitt »Große Zahlen in Dreiergruppen aufteilen: Lookaround« in Kapitel 2 von *Reguläre Ausdrücke*, 2. Auflage.

2.17 Den Plural korrekt ausgeben

Problem

Sie geben so etwas wie "Es dauerte \$time Stunden" aus, aber "Es dauerte 1 Stunden" ist nicht korrekt. Diesen Fehler möchten Sie beheben.

Lösung

Verwenden Sie `printf` und den Konditionaloperator (`X ? Y : Z`), um das Substantiv oder Verb zu korrigieren:

```
printf "Es dauerte %d Stunde%s\n", $time, $time == 1 ? "" : "n";
```

```
printf "%d Stunde%s %s genug.\n", $time,
       $time == 1 ? "" : "n",
       $time == 1 ? "ist" : "sind";
```

Bei englischem Text können Sie auch das CPAN-Modul `Lingua::EN::Inflect` verwenden, das wir in der folgenden Diskussion beschreiben.

Diskussion

Der einzige Grund, warum so alberne Meldungen wie "1 file(s) updated" überhaupt vorkommen, besteht einfach darin, dass der Programmierer zu faul ist zu prüfen, ob der Zähler nun 1 ist oder nicht.

Ändert sich, wie im folgenden englischen Beispiel, mehr als nur ein "-s", müssen Sie `printf` entsprechend anpassen:

```
printf "It took %d centur%s", $time, $time == 1 ? "y" : "ies";
```

In einfachen Fällen reicht das aus, aber Sie werden es leid sein, das immer wieder zu schreiben. Das führt zu lustig aussehenden Funktionen wie:

```
sub noun_plural {
    local $_ = shift;
    # Die Reihenfolge ist hierbei unbedingt zu beachten!
    s/ss$/sses/           ||
    s/([psc]h)$/${1}es/  ||
    s/z$/zes/            ||
    s/ff$/ffs/          ||
    s/f$/ves/           ||
    s/ey$/eys/          ||
    s/y$/ies/           ||
    s/ix$/ices/         ||
    s/([sx])$/${1}es/   ||
    s/$/s/              ||
    die "das kann nicht sein";
    return $_;
}
*verb_singular = \&noun_plural; # Funktionsalias
```

Je mehr Ausnahmen Sie entdecken, desto unübersichtlicher wird Ihre Funktion. Für englischsprachige Probleme dieser Art bietet das CPAN-Modul `Lingua::EN::Inflect` eine flexible Lösung.

```
use Lingua::EN::Inflect qw(PL classical);
classical(1); # Warum ist das nicht Standard?
while (<DATA>) { # Jede Zeile der Daten
    for (split) { # Jedes Wort der Zeile
        print "One $_, two ", PL($_), ".\n";
    }
}
# Noch eins dazu
$_ = 'secretary general';
print "One $_, two ", PL($_), ".\n";
```

```
__END__
fish fly ox
species genus phylum
cherub radius jockey
index matrix mythos
phenomenon formula
```

Das erzeugt die folgende Ausgabe:

```
One fish, two fish.
One fly, two flies.
One ox, two oxen.
One species, two species.
One genus, two genera.
One phylum, two phyla.
One cherub, two cherubim.
One radius, two radii.
One jockey, two jockeys.
One index, two indices.
One matrix, two matrices.
One mythos, two mythoi.
One phenomenon, two phenomena.
One formula, two formulae.
One secretary general, two secretaries general.
```

Ohne den Aufruf von `classical` würden die Zeilen anders ausgegeben werden als in der obigen Ausgabe:

```
One phylum, two phylums.
One cherub, two cherubs.
One radius, two radiuses.
One index, two indexes.
One matrix, two matrixes.
One formula, two formulas.
```

Dies ist nur eines der vielen Dinge, die dieses Modul tun kann. Es beugt und konjugiert auch andere Teile der Sprache, stellt zahlenunabhängige Vergleichsfunktionen zur Verfügung, findet heraus, ob ein *a* oder ein *an* zu verwenden ist, und vieles mehr.

Siehe auch

Den Konditionaloperator in `perlop(1)` und in Kapitel 3 von *Programmieren mit Perl* (S. 108); die Dokumentation zum CPAN-Modul `Lingua::EN::Inflect`.

2.18 Programm: Primfaktoren berechnen

Das folgende Programm erlaubt die Übergabe von einem oder mehreren Integer-Argumenten und ermittelt deren Primfaktoren. Es verwendet die Perl-eigene, numerische Darstellung, solange die Zahlen keine Fließkomma-Darstellung verwenden und daher an

Beispiel 2-2: *bigfact* (Fortsetzung)

```
# Hier wird $sqi zum Quadrat von $i. Wir nutzen die Tatsache aus,
# dass  $(i + 1)^2 = i^2 + 2 * i + 1$  ist.
for (my ($i, $sqi) = (2, 4); $sqi <= $n; $sqi += 2 * $i ++ + 1) {
    while ($n % $i == 0) {
        $n /= $i;
        print STDERR "<$i>" if $opt_d;
        $factors {$i} ++;
    }
}

if ($n != 1 && $n != $orig) { $factors{$n}++ }
if (! %factors) {
    print "PRIMZAHL\n";
    next ARG;
}
for $factor ( sort { $a <=> $b } keys %factors ) {
    print "$factor";
    if ($factors{$factor} > 1) {
        print "**$factors{$factor}";
    }
    print " ";
}
print "\n";
}

# Simuliert ein use, aber zur Laufzeit
sub load_biglib {
    require Math::BigInt;
    Math::BigInt->import();          # immateriell?
}
```