Hauptseminar

Die Programmiersprache PERL

Perl Version 5

Andreas Koppenhöfer

Student an der Fakultät Informatik
Universität Stuttgart
email: koppenas@informatik.uni-stuttgart.de

7. Februar 1995

Inhaltsverzeichnis

1 Einführung

Perl ist eine mächtige Programmiersprache, die viele Einsatzmöglichkeiten bietet. Obwohl der Sprachumfang, als auch die Flexibilität sehr groß sind, entstehen bei der Arbeit mit Perl immer wieder Wünsche nach Verbesserungen und Erweiterungen der Sprache.

Seit der Veröffentlichung der Perl Version 4 wurde viel über Verbesserungen und Erweiterungen der Sprache diskutiert. Die Wunschliste war lang. Der Autor Larry Wall hat viel Zeit investiert, um möglichst viele dieser Wünsche zu realisieren. Dazu gehören foglende...

- Erleicherungen zum Schreiben "lesbarer" Programme;
- aussagekräftigere Fehlermeldungen;
- Deklaration von Variablen mit lexikalischer Gültigkeit, ähnlich der "auto"-Variablen in C.
- mehrstufige Datenstrukturen: Jede skalare Variable oder array-Element darf Zeiger auf beliebige andere Variablen oder Unterprogrammen enthalten.
- Modularität und Wiederverwendbarkeit: Aufteilung der Perl-Bibliothek in einzelne Module; Einfaches Einbinden eines Moduls in ein package; Module können dynamisch während der Laufzeit nachgeladen werden.
- Objektorientierte Programmierung: Ein package kann als Objektklasse fungieren. Methoden und Instanzen werden mit wenigen neuen syntaktischen Elementen implementiert. Filehandles können als Objekte behandelt werden.
- Vereinfachte Zusammenarbeit von C/C++–Applikationen und Perl. Beide Sprachen, Perl sowie C/C++, können Unterprogramme der jeweils anderen Sprache verwenden.
- POSIX-Unterstützung: Das POSIX-Modul erlaubt den Zugriff auf alle POSIX-Funktionen und -Definitionen.
- Unterprogramme können nachträglich bei Bedarf definiert werden: autoloading.
- Verbesserung der regulären Ausdrücke.

Larry Wall, hat schon Ende 1992 die ersten Alpha-Releases von Perl–5 für Frühjahr 1993 angekündigt. Es zeigte sich aber, daß diese Ankündigung verfrüht war. In der Zwischenzeit hat er Perl–5 von Grund auf neu geschrieben. Erst dadurch war es ihm möglich einen Großteil der Wünsche zu implementieren.

`Manualeintrag:` *Überschrift*

Dieser Text basiert auf dem ersten offiziellen Release 5.000 vom Oktober 1994. Die angegebenen Randnotizen verweisen auf weitergehende Informationen zum jeweiligen Thema. Die *"Überschrift"* dient zu Orientierung in umfangreicheren Manualeinträgen.

2 Perl-Manual – Suchen und Finden von Informationen

Die Online verfügbare Dokumentation über Perl–5 wurde nicht nur erweitert, sondern auch gründlich überarbeitet. Der Umfang von 240 Seiten machte es nötig, den Inhalt auf einzelne Themen aufzuspalten.

Das UNIX-Kommando "`man perl`"[1] gibt eine Übersicht über die verschiedenen zur Verfügung stehenden Manualeinträge aus.

`perl`

`perl/perl5`	Perl overview
`perldata`	Perl data structures
`perlsyn`	Perl syntax
`perlop`	Perl operators and precedence
`perlre`	Perl regular expressions
`perlrun`	Perl execution and options
`perlfunc`	Perl builtin functions
`perlvar`	Perl predefined variables
`perlsub`	Perl subroutines
`perlmod`	Perl modules
`perlref`	Perl references and nested data structures
`perlobj`	Perl objects
`perlbot`	Perl OO tricks and examples
`perldebug`	Perl debugging
`perldiag`	Perl diagnostic messages
`perlform`	Perl formats
`perlipc`	Perl interprocess communication
`perlsec`	Perl security
`perltrap`	Perl traps for the unwary
`perlstyle`	Perl style guide
`perlapi`	Perl application programming interface
`perlguts`	Perl internal functions for those doing extensions
`perlcall`	Perl calling conventions from C
`perlovl`	Perl overloading semantics
`perlbook`	Perl book information

Aus dieser Liste ist zu erfahren, daß zum Beispiel die Syntax der Sprache Perl im Manualeintrag "`perlsyn`" beschrieben ist.

Weitere Informationen sind in der im Anhang beschreibenen Literatur zu finden. Als wertvolle Quelle erweist sich dabei die Perl-FAQ, die Antworten auf häufig gestellte Fragen zum Thema Perl enthält. Ihr großer Vorteil ist die Aktualität, da sie monatlich über Internet verteilt wird.

[1] Bei einigen Installationen an der Fakultät erhält man bei "`man perl`" die Dokumentation der Version 4, mit "`man perl5`" dagegen die der Version 5.

3 Unverträglichkeiten der beiden Versionen

Perl–5 ist nicht hundertprozentig aufwärtskompatibel zu Perl–4. Einige wenige Details haben sich geändert. Dennoch werden sich die meisten Perl–4-Programme ohne Änderung mit Perl–5 verwenden lassen.

array-Manipulationen

`perldata:` *Scalar Values*

Das folgende Beispiel zeigt array-Manipulationen, wie sie mit Perl–4 möglich waren:

Beispiel 3.1

```
@array = ('A' .. 'E');
print join(' ', @array)."\n";        ☞ A B C D E
@#array = 2;
print join(' ', @array)."\n";        ☞ A B
@#array = 4;
print join(' ', @array)."\n";        ☞ A B C D
```

In Perl–4 bleiben alle Elemente eines Arrays erhalten, die durch eine Verkleinerung unsichtbar werden. Dagegen werden in Perl–5 die Elemente tatsächlich gelöscht. Der letzte `print`-Befehl in Beispiel 3.1 ergibt mit Perl–5 die Ausgabe "`A B`".

Zeichenketten ohne "&" und ohne Anführungszeichen

`perldata:` *Scalar value constructors*

Eine Wort in einem Perl-Programm, das keine von der Grammatik bestimmte Bedeutung hat, wird als "bareword" bezeichnet. In Beispiel 3.2 ist das Wort `hallo` in der zweiten Zeile ein solches bareword. Ein bareword wird in beiden Perl-Versionen üblicherweise als Zeichenkette interpretiert. Wenn jedoch ein Unterprogramm (`sub`) mit gleichem Namen definiert ist, interpretiert Perl–5 dieses bareword als Unterprogrammaufruf. Beispiel 3.2 zeigt das Verhalten von Perl–4.

Beispiel 3.2

```
sub hallo {high}
$text = hallo."\n";
print $text;                         ☞ hallo
```

`perlsub`

Perl–5 dagegen gibt mit dem Programm aus Beispiel 3.2 "`high`" aus, da es das Wort `hallo` nicht als String sondern als Unterprogrammaufruf interpretiert. Es ist deshalb empfehlenswert, Worte, die als Strings interpretiert werden sollen, grundsätzlich in Anführungszeichen zu setzen. Unterprogrammaufrufe schreibt man am besten mit einem vorangestelltem "`&`". Beispiel 3.3 ist die verbesserte Version von Beispiel 3.2.

Beispiel 3.3

```
sub hallo {"high"}
$text = &hallo."\n";
print $text;                         ☞ high
```

Variablen, die nicht verändert werden können

In Perl–5 sind einige spezielle Variablen "read-only", das heißt, daß sie nicht auf der linken Seite einer Zuweisung verwendet werden dürfen (als "lvalue"). Beispiel 3.4 war mit Perl–4 erlaubt, führt aber unter Perl–5 zur Fehlermeldung "`Modification of a read-only value attempted at line 4`".

Beispiel 3.4

```
while(<>) {
      chop;
      if (m/<(.*)>/) { # suche pattern
            $1 .= "\n"; # hänge newline an
            print $1;
      }
}
```

Geänderte Package-Bezeichner

In Perl–4 wurden Unterprogramme und Variablen aus anderen Packages in der Form

```
$Package'Variable
```

geschrieben. Diese Form ist unter Perl–5 zwar noch erlaubt, jedoch bevorzugt Perl–5 aus verschiedenen Gründen die Schreibweise

`perlmod:` *Packages*

```
$Package::Variable.
```

Zusätzlich wurde eine Kurzform eingeführt: Fehlt die Package-Angabe, so wird die "`main`"-Package verwendet wie z. B. in

```
$::Variable.
```

Verschiedenes

Der Manualeintrag zu `perltrap` beschreibt noch einige weniger wichtige Unverträglichkeiten der beiden Perl-Versionen.

`perltrap:` *Perl4 Traps*

In der Perl-FAQ wird unter dem Stichwort *"Will perl5 break my perl4 scripts?"* ebenfalls über die Unterschiede von Perl–4 und Perl–5 diskutiert.

4 Perl-Programme: Übersetzung und Ausführung

Ein Perl-Programm wird bei jedem Start zunächst in einen Zwischencode übersetzt, compiliert. Dieser Zwischencode wird dann von einem Interpreter ausgeführt. Das Verhalten von Compiler und Interpreter läßt sich z. B. mit Kommandozeilenoptionen beeinflussen. Eine Liste aller Kommandozeilenoptionen und ihre Bedeutung ist im Manualeintrag `perlrun` zu finden.

Ein für die Fehlersuche wichtige Option ist "`-w`".

4.1 Fehlersuche mit Hilfe der Option “-w”

`perlrun`

Die Kommandozeilenoption “`-w`” veranlaßt den Interpreter auf mögliche Probleme im Perl-Programm hinzuweisen. Wird das Programm aus Beispiel 3.2 mit Option “`-w`” gestartet, so erhält man die Warnung

```
Unquoted string 'high' may clash with future reserved word.
```

An vielen Stellen in der Perl-Dokumentation stehen Empfehlungen, die Option “`-w`” zu verwenden. Wird die Option eingeschaltet, dann wird das Perlprogramm auf typische Programmierfehler überprüft. Dazu gehören Bezeichner, die nur einmal verwendet werden, und skalare Variablen, die verwendet werden, bevor sie einen definierten Wert enthalten. Geprüft wird auch, ob Unterprogrammnamen mehrfach definiert werden, die Verwendung von undefinierten Filehandles und einiges mehr.

`perldiag`

Im Manualeintrag `perldiag` sind alle möglichen Warnungstexte und Fehlermeldungen des Perl-Compilers und -Interpreters ausführlich erläutert.

4.2 Compilersteuerung mit pragmas

`perlmod:`
Pragmatic Modules

Einen Befehl im Quelltext eines Programm, der das Verhalten des Compilers verändert, nennt man pragma. Pragmas werden in Perl–5 mit dem `use`-Befehl eingeschaltet.

pragma “`use strict 'subs'`”

Das Beispiel 4.1 zeigt, wie der Compiler auf einen Programmierfehler hinweisen kann: fehlende Anführungszeichen. Perl bricht das Programm mit einer Fehlermeldung ab.

Beispiel 4.1
```
use strict 'subs';
$text = hallo;
     ☞ Bareword 'hallo' not allowed while 'strict subs' in use
```

pragma “`use integer`”

Perl rechnet normalerweise mit Gleitkommazahlen. Das bedeutet, daß auch eine Zuweisung wie z. B.

```
$wert = $anfang + $anzahl * $groesse;
```

mit Gleitkommaarithmetik berechnet wird, selbst dann, wenn nur ganze Zahlen in den Rechnungen vorkommen. Perl–5 bietet nun die Möglichkeit reine Ganzzahlarithmetik zu verwenden. Je nach Situation (Art des Programms, der Daten und der verfügbaren Hardware) kann das zu erheblichen Geschwindigkeitsvorteilen führen.

Beispiel 4.2
```
use integer;
$endwert = $anfang + $anzahl * $groesse;
if ($index < 10) {
     no integer;
```

```
        $prozent = $index / $endwert * 100;
    }
    $index += $groesse;
```

In Beispiel 4.2 werden zunächst die Zuweisung für `$endwert` und die Testbedingung der `if`-Anweisung mit Ganzzahlarithmetik zu berechnet. Der Befehl "`no integer`" bewirkt das Gegenteil des `use`-Befehls: die Compilerdirektive wird ausgeschaltet. Die Berechnung der nachfolgenden `$prozent`-Zuweisung erfolgt wieder mit gewohnter Gleitkommaarithmetik.

Alle `use`- und `no`-Befehle haben nur Gültigkeit bis zum Ende des Blocks, in dem Sie definiert wurden. Deshalb wird im Beispiel 4.2 die Zuweisung an `$index` und alle nachfolgenden Berechnungen bis zum Blockende wieder mit Ganzzahlarithmetik berechnet.

Weitere Anwendungen des `use`-Befehls werden im Abschnitt 7 beschrieben.

5 Daten- und Objekttypen

In diesem Abschnitt werden einige Begriffe aus der Perl-Dokumentation verwendet, ohne sie ins Deutsche zu übersetzen.

scalar: eine einfache Variable, die immer nur ein Wert enthalten kann.

array: ein Feld, das mehrere skalare Elemente enthalten kann. Der Zugriff auf einzelne Elemente erfolgt über eine Zahl als Index.

hash: assoziatives array – wie ein array, jedoch erfolgt die Adressierung der einzelnen Elemente über frei wählbare Schlüssel (keys), z. B. über Strings.

Werte die ein scalar, array oder hash enthalten können, sind nicht auf Zahlen und Zeichenketten beschränkt. Perl–5 kennt als Wert auch noch Referenzen, die in Abschnitt 5.3 beschrieben werden.

5.1 Deklaration und Gültigkeit

`perlfunc:` *local*

Mit dem Befehlen "`local`" und "`my`" lassen sich lokale Variablen deklarieren. Sie sind gültig bis zum Ende des Blocks in dem sie definiert wurden. Alle mit "`local`" deklarierten Variablen sind zusätzlich in allen Unterprogrammen sichtbar, die innerhalb ihrer Gültigkeit aufgerufen werden ("dynamic scoping"). Mit "`my`" deklarierte Variablen sind in Unterprogrammen nicht sichtbar. Die Gültigkeit von "`my`-Variablen ("lexical scoping") entspricht der Gültigkeit der Klasse der `auto`-Variablen in der Sprache C.

Das Beispiel 5.1 verdeutlicht den Unterschied von "`local`" und "`my`".

Beispiel 5.1

```
    sub printall {
        print "$text1 $text2!\n";
    }
    sub callprint {
```

```
        local($text1) = 'Die Sonne';
        my($text2) = 'scheint';
        &printall;
    }
    $text1 = 'Der Mond';
    $text2 = 'geht auf';
    &callprint;                          ☞ Die Sonne geht auf!
```

`perlfunc:` *my*

Eine Deklaration mit "`my`" ist im Vergleich mit "`local`" sicherer in der Handhabung und scheller in der Ausführung. Jedoch müssen in den folgenden Situationen unbedingt "`local`"-Deklarationen eingesetzt werden.

- Bei Variablen, die innerhalb von `format`-Befehlen verwendet werden;
- Bei Variablen, deren Name nicht mit einem Buchstaben beginnt, wie zum Beispiel "`$_`".

5.2 Assoziative Arrays

`perldata:` *List value constructors*

Es gibt in Perl–4 mehrere Möglichkeiten (Beispiel 5.2), einem assoziativen Array Werte zuzuweisen.

Beispiel 5.2

```
$map{'red'}   = 0x00f;
$map{'blue'}  = 0x0f0;
$map{'green'} = 0xf00;
# use atomic assignment instead
($map{'red'},$map{'blue'},$map{'green'}) = (0x00f,0x0f0,0xf00);
# same as map assignment above
%map = ('red',0x00f,'blue',0x0f0,'green',0xf00);
```

Die letzte Zeile aus Beispiel 5.2 weist `%map` eine Liste zu. Diese Liste hat die Form

```
(key0, value0, key1, value1, key2, value2, ...)
```

Perl–5 kennt zusätzlich die in Beispiel 5.3 gezeigte Form, deren verbesserte Lesbarkeit besonders bei großen Listen von Vorteil ist.

Beispiel 5.3

```
%map = ('red'   => 0x00f,
        'blue'  => 0x0f0,
        'green' => 0xf00,
);
```

5.3 Referenzen und mehrstufige Datenstrukturen

Mit Perl–4 war es schwierig komplexe Datenstrukturen zu definieren, da es keine Zeiger (Referenzen) auf Objekte und Daten gab.

Referenzen auf Symbole

Perl–5 kennt zwei Typen von Referenzen auf Objekte (Verweis, Zeiger, pointer). Die eine ist symbolischer Natur und entspricht im Vergleich mit dem UNIX-Dateisystem einem "symolic link". Die Schreibweise "`*name`" ergibt einen Verweis auf die Symboltabelle, der alle Objekte umfaßt, die den angegebenen Namen haben. In der Perldokumentation wird eine symbolische Referenz als "`type glob`" oder "`glob`" bezeichnet. Wie in Beispiel 5.4 demonstiert wird, können über eine einzige symbolische Referenz Objekte verschiedenen Typs angesprochen werden.

`perlsub:` *Passing Symbol Table Entries*

Beispiel 5.4
```
sub doubleall {
     local(*someglob) = @_;
     foreach $elem (@someglob) {
          $elem *= 2;
     }
     foreach $key (keys %someglob) {
          $someglob{$key} *= 3;
     }
}
@foo = (1, 3, 5, 7);
%foo = ('zehn' => 10, 'elf' => 11);
doubleall(*foo);
print join ' ', @foo, "\n";           ☞ 2 6 10 14
print "zehn: $foo{'zehn'}, elf: $foo{'elf'}\n";
                                      ☞ zehn: 30, elf: 33
```

Referenzen auf Objekte

Die zweite Art von Referenz ist nicht symbolisch, sondern "hart". Beim Vergleich mit dem UNIX-Dateisystem entspricht sie einem "hard link". Eine harte Referenz zeigt also nicht auf die Definition eines Objekts, sondern auf das Objekt selbst.

`perlref`

In der Perldokumentation und im folgenden Text werden harte Referenzen kurz als Referenzen (`ref`) bezeichnet. Symbolische Referenzen werden dagegen "`type glob`" oder nur "`glob`" genannt.

Jede skalare Variable kann als Wert eine Referenz enthalten. Da die Elemente von eindimensionalen Arrays und assoziativer Arrays skalarer Natur sind, lassen sich mehrstufig verschachtelte Objektstrukturen bilden, indem array- und hash-Elementen Referenzen zu anderen Objekten zugewiesen werden.

Entstehung von Referenzen

Es gibt verschiedenen Möglichkeiten, Referenzen zu erzeugen.

1. Wird einer Variablen, einem Unterprogrammnamen (sub) oder einem Wert ein "\" vorangestellt, entsteht eine Referenz auf das entsprechende Objekt.

   Beispiel 5.5
   ```
   $scalarref = \$foo;
   ```

```
$arrayref  = \@ARGV;
$hashref   = \%ENV;
$coderef   = \&handler;
```

2. Eine Referenz zu einem anonymen Array kann mittels eckiger Klammern konstruiert werden:

   Beispiel 5.6
   ```
   $arrayref = [1, 2, ['a', 'b', 'c']];
   ```

   Es handelt sich hier um ein dreielementiges Array, dessen letztes Element eine Referenz auf ein weiteres Array ist. Ein Zugriff auf den Wert "`b`" ist z. B. mit `$arrayref->[2][1]` möglich.

3. Eine Referenz zu einer hash-Variablen wird mit geschweiften Klammern wie in Beispiel 5.7 dargestellt.

   Beispiel 5.7
   ```
   $hashref = {
        'Adam' => 'Eve',
        'Clyde' => 'Bonnie',
   };
   ```

   Da eine öffnende geschweifte Klammer u. a. auch den Anfang eines Befehlsblocks anzeigen kann, müssen hash-Referenzen gegebenenfalls mit "`+`" oder "`return`" markiert werden.

   Wenn sie in einer Funktion ein neues hash erzeugen und als Ergebnis eine hash-Referenz zurückgeben wollen, dann haben sie folgende Möglichkeiten.

   Beispiel 5.8
   ```
   sub hashem {        { @_ } } # falsch
   sub hashem {      + { @_ } } # ok
   sub hashem { return{ @_ } } # ok
   ```

4. Eine Referenz zu einem Unterprogramm erhalten sie, indem Sie den Befehl "`sub`" ohne Namen verwenden.

   Beispiel 5.9
   ```
   $coderef = sub { print "Hello World!\n" };
   ```

   Zu beachten ist dabei, daß zu dem Zeitpunkt der Zuweisung das Unterprogramm noch nicht ausgeführt wird.

Verwendung von Referenzen

Wie Referenzen entstehen ist nun bekannt. Es folgen nun Methoden, wie Referenzen verwendet werden können.

1. An jeder Stelle, an der ein Name als Bestandteil einer Variablen oder Unerprogrammaufruf stehen darf, können sie statt des Namens eine skalare Variable einsetzen, die als Wert eine Referenzen vom entsprechenden Typ enthält.

Beispiel 5.10

```
$bar = $$scalarref;
push(@$arrayref, $filename);
$$arrayref[0] = "January";
$$hashref{"KEY"} = "VALUE";
&$coderef(1,2,3);
```

Es ist wichtig zu verstehen, daß hier kein Zugriff auf `$arrayref[0]` oder `$hashref{"KEY"}` erfolgt. Die Auswertung (dereferencing) der skalaren Variablen wird *zuerst* durchgeführt. Wie das folgende Beispiel 5.11 zeigt, können Referenzen auch rekursiv eingesetzt werden.

Beispiel 5.11

```
$refrefref = \\\"Kompliziert?\n";
print $$$$refrefref;                ☞ Kompliziert?
```

2. An jeder Stelle, an der ein Name als Bestandteil einer Variablen oder eines Unterprogrammaufruf stehen darf, können sie statt des Namens einen Befehlsblock einsetzen, der als Wert eine Referenz vom entsprechenden Typ liefert.

Beispiel 5.12

```
$bar = ${$scalarref};
push(@{$arrayref}, $filename);
${$arrayref}[0] = "January";
${$hashref}{"KEY"} = "VALUE";
&{$coderef}(1,2,3);
```

Es ist im Beispiel 5.12 nicht nötig geschweifte Klammern einzusetzen. Da jedoch der Befehlsblock beliebige Ausdrücke enthalten darf, ist der Einsatz zusätzlicher Klammern manchmal unumgänglich, wie im Beispiel 5.13 mit einem indizierten Ausdruck demonstriert wird.

Beispiel 5.13

```
&{ $dispatch{$index} }(1,2,3); # call correct routine
```

3. Es gibt noch eine zweite äquivalente Notation für Referenzen. Die beiden Zeilen aus Beispiel 5.14 entsprechen denen aus 5.15.

Beispiel 5.14

```
${$arrayref}[0] = "January";
${$hashref}{"KEY"} = "VALUE";
```

Beispiel 5.15

```
$arrayref->[0] = "January";
$hashref->{"KEY"} = "VALUE";
```

Das syntaktische Element "`->`" wird in Perl–5 zur Auflösung einer Referenz eingesetzt, wie es auch in der Sprache C für Zeiger (pointer) verwendet wird. Zu beachten ist, daß "`$array[$x]`" *nicht* dasselbe bedeutet wie "`$array->[$x]`"!

Beispiel 5.16

```
$array[$x]->{"foo"}->[0] = "January";
```

Mit Hilfe von Beispiel 5.16 soll gezeigt werden, wie bisher noch nicht existierende Referenzen benutzt werden, falls "`$array[$x]`" vor diesem Befehl noch undefiniert war. Wenn das der Fall ist, dann wird "`$array[$x]`" automatisch mit einer Referenz initialisiert, so daß der nächste Teil des Ausdrucks "`{"foo"}`" ausgewertet werden kann. Folglich wird auch "`$array[$x]->{"foo"}`" automatisch mit einer array-Referenz initialisiert, so daß darüber der Rest des Ausdrucks "`[0]`" noch bewertet werden kann.

Wenn ein syntaktisches Element "`->`" wie in Beispiel 5.16 zwischen zwei Klammern steht, dann darf es weggelassen werden. Beispiel 5.17 ist deshalb gleichbedeutend mit 5.16.

Beispiel 5.17

```
$array[$x]{"foo"}->[0] = "January";
```

4. Mehrdimensionale Arrays können leicht mit den bisher gezeigten Anwendungsmöglichkeiten von Referenzen konstruiert werden. Die Schreibweise entspricht dabei den mehrdimensionalen Feldern der Sprache C.

Beispiel 5.18

```
$score[$x][$y][$z] += 42;
```

Gemeinsame Verwendung von `globs` und `refs`

`perlref`: *Symbolic referenzes*

Im Manualeintrag zu `perlref` finden sich noch weitere Einsatzmöglichkeiten für symbolische und harte Referenzen.

5.4 Objektklassen und Instanzen

Eine andere Leistungsfähige Erweiterung in Perl–5 sind Objektklassen und Methoden.

`perlobj`

In Perl–5 gelten folgende drei einfache Definitionen:

1. Ein Objekt ist eine einfache Referenz, die "weiß", zu welcher Klasse sie gehört.
2. Eine Klasse ist nur ein Package, das Methoden zur Manipulation von Objektreferenzen bereitstellt.
3. Eine Methode ist ein Unterprogramm (sub), das als erstes Argument eine Objektreferenz erwartet.

`perlbot`

In den Manualeinträgen `perlobj` und `perlbot` werden Perl-Objekte anhand von Beispielen ausführlich erklärt. Deshalb wird hier nicht näher darauf eingegangen.

6 Neue Funktionen

`perlfunc`

In Perl–5 sind eine Reihe neuer Funktionen implementiert worden. Die wichtigsten davon enthält die folgende Aufzählung.

`abs EXPR` — Als Ergebnis wird der Absolutwert des Arguments zurückgegeben.

`chomp EXPR` — Arbeitet ähnlich wie die Funktion `chop`. `chop` entfernt das letzte Zeichen eines Strings. `chomp` dagegen entfernt alle am Ende eines Strings stehenden Zeilenendezeichen (newline), sofern welche vorhanden sind. Beispiel:

Beispiel 6.1
```
while (<>) {
      chomp# avoid "\n" on last field
      @array = split(/:/);
      ...
}
```

`chr EXPR` — Wandelt eine Zahl in das entsprechende Zeichen um. Zum Beispiel ergibt "`chr(65)`" das Zeichen "`A`" im ASCII-Zeichensatz.

`exists EXPR` — Liefert den Wert `true`, wenn das als Argument angegebene Element eines assoziativen Arrays existiert, auch wenn der zugehörige Wert undefiniert ist. Beispiel 6.2 vergleicht `exists` mit anderen ähnlichen Funktionen.

Beispiel 6.2
```
print "Exists\n"  if exists $array{$key};
print "Defined\n" if defined $array{$key};
print "True\n"    if $array{$key};
```

`map BLOCK LIST; map EXPR,LIST` — Für jedes Element aus `LIST` wird `BLOCK` bzw. `EXPR` genau einmal ausgewertet. Vor der Ausführung wird `$_` auf den Wert des jeweils aktuellen Elements gesetzt. Die beiden folgenden Beispiele 6.3 und 6.4 führen zu demselben Ergebnis.

Beispiel 6.3
```
@nums = (80, 101, 114, 108);
@chars = map(chr, @nums);
print join(':', @chars),"\n";        ☞ P:e:r:l
```

Beispiel 6.4
```
@nums = (80, 101, 114, 108);
{     local($_);
      @chars = ();
      foreach $_ (@nums) {
      push @chars, chr;
}
print join(':', @chars),"\n";        ☞ P:e:r:l
```

`pos` — Arbeitet zusammen mit dem "pattern match"-Befehl "`m/PATTERN/g`". Die Option "g" veranlaßt die Matchoperation jedes passende Pattern einer Zeichenkette zu finden. Es liefert bei der ersten Verwendung das erste Pattern, bei jeder weiternen Verwendung das jeweils nächste. Die Funktion `pos` gibt an, bis zu welcher Position das jeweils gefundene Pattern in der Zeichenkette reicht. Im Beispiel 6.5 werden alle Worte und ihre Position ermittelt, die in einem String enthalten sind.

`perlop:` *Quote and Quotelike Operators*

Beispiel 6.5

```
$_ = '7. Februar 1995';
while(m/(\w+)/g) {
    print "$1 (".(pos).") ";  ☞ 7 (1) Februar (10) 1995 (15)
}
```

perlref

`ref EXPR` — liefert einen wahren Wert (`true`), wenn `EXPR` eine Referenz ist, ansonsten falsch (`false`). Der zurückgegebene Wert ist vom Typ des referenzierten Objekts abhängig. Folgende Typen kennt Perl–5: `REF`, `SCALAR`, `ARRAY`, `HASH`, `CODE` und `GLOB`. Damit entspricht `ref()` in etwa dem `typeof()`-Operator der Sprache C.

Beispiel 6.6

```
if (ref($r) eq "HASH") {
    print "r is a reference to an associative array.\n";
}
if (!ref ($r) {
    print "r is not a reference at all.\n";
}
```

7 Packages und Module

7.1 Konstruktoren und Destruktoren

`perlmod:` *Package Constructors and Destructors*

Ein Konstruktor ist ein spezieller Programmblock, der zur Compilezeit des Perlprogramms ausgeführt wird. Er wird mit dem Schlüsselwort `BEGIN` markiert. Ein Destruktor, markiert mit `END`, wird dagegen erst unmittelbar vor Ende eines Programms ausgeführt.

Beispiel 7.1

```
package serial;
# Initialisieren der Seriennummer mit einer Zufallszahl
BEGIN { srand; $no = int(rand 1e6) };
END { print "next: $no\n"; }
# no() liefert aufeinanderfolgende Seriennummern
sub no { $no++ };
package main;
# Ausgabe von zwei Seriennummern
print &serial::no," ",
    &serial::no,"\n";                ☞ 568796 568797
exit(0);                             ☞ next: 568798
```

Beispiel 7.1 demonstiert das Verhalten von `BEGIN` und `END`. Zu beachten ist, dass der `END`-Block auch nach (!) einem `die`-Befehl noch ausgeführt wird. Es ist erlaubt, mehrere `BEGIN`- und `END`-Blöcke in einem Programm zu verwenden. `BEGIN`-Blöcke werden einfach nacheinander in der Reihenfolge ihrer Definition ausgeführt. Bei Programmende werden `END`-Blöcke in umgekehrter Reihenfolge ihrer Definition bearbeitet.

7.2 Module

Ein Module ist ein `package`, das in einer eigenen Datei als Bibliothek abgelegt wurde. Der Hauptzweck eines Moduls ist die einfache Wiederverwendbarkeit des Programmcodes. Das wird erreicht durch einen Mechanismus, der Symbole eines Moduls exportieren kann, so daß sie von ausserhalb sichtbar werden. Alle aus einem Modul importieren Unterprogramme können als "`list operator`" verwendet werden. Den Unterschied zwischen "`use`" und "`require`" versucht Beispiel 7.2 deutlich zu machen.

`perlmod:` *Perl Modules*

Beispiel 7.2

```
require Cwd; # make Cwd:: accessible
$here = Cwd::getcwd();
use Cwd; # import names from Cwd::
$here = getcwd();
require Cwd; # make Cwd:: accessible
$here = getcwd(); # oops! no main::getcwd()
```

Ein Modul kann auch als Definition einer Objektklasse dienen und macht seine Semantik implizit über Methoden auf den Klassen und seinen Objekten bekannt, ohne spezielle Symbole zu exportieren.

`perlobj:` *Method Invocation*

Perlmodule werden in ein Programm mit Befehlen wie in Beispiel 7.3 eingebunden.

Beispiel 7.3

```
use FileHandle;
autoflush STDOUT 1;
use POSIX 'uname';
@ident = qw(sysname nodename release version machine);
@uname = uname;
while($#ident >= 0) {
    printf "%-8s: %s\n", (shift @ident), (shift @uname);
}
```

Als Ergebnis liefert Beispiel 7.3 auf dem Rechner des Autors:

```
sysname : Linux
nodename: ako
release : 1.1.88
version : #3 Fri Feb 3 12:28:24 MET 1995
machine : i486
```

Die beiden `use`-Befehle aus Beispiel 7.3 können auch wie im folgenden Beispiel 7.4 umschrieben werden.

Beispiel 7.4

```
BEGIN { require "FileHandle.pm"; import FileHandle; }
BEGIN { require "POSIX.pm"; import POSIX 'uname'; }
```

Wie ein Modul geschrieben wird, ist über die Manualeinträge `perlapi` und `perlguts` zu erfahren. Die Perldokumentation empfiehlt ein bereits existierendes Modul zu analysieren und es als Prototyp für eigene Entwicklungen zu verwenden.

`perlapi`
`perlguts`

7.3 Verfügbare Bibliotheken

Die in der Perldokumentation enthaltenen Beispiele enthalten häufig Fehler.

Deshalb wird in diesem Abschnitt der Schwerpunkt auf funktionsfähige Beispiele gelegt. Detailierte Informationen sind aus den Manualeinträgen und dem Quellcode zu bekommen.

abbrev

`Abbrev` — generiert aus einer Liste eine hash-Tabelle mit Abkürzungen. In der hash-Tabelle sind als Schlüssel alle eindeutigen Abkürzungen zu finden. Die dazugehörigen Werte sind die ausgeschreibenen Worte der Liste. Beispiel 7.5 zeigt ein Beispiel.

Beispiel 7.5
```
use Text::Abbrev;
abbrev *ab, qw(fischers fritze faengt frische fische);
foreach $key (sort keys %ab) {
     printf "%-10s=> %s\n", $key, $ab{$key};
}
```

Als Ausgabe liefert das Programm aus Beispiel 7.5 folgende Tabelle:

```
fa        => faengt        fris     => frische
fae       => faengt        frisc    => frische
faen      => faengt        frisch   => frische
faeng     => faengt        frische  => frische
faengt    => faengt        frit     => fritze
fische    => fische        fritz    => fritze
fischer   => fischers      fritze   => fritze
fischers  => fischers
```

Basename

`Basename` — ist ein Modul, das drei nützliche Routinen zur Dateibearbeitung enthält.

`basename` — trennt einen Dateiname von einer Pfadangabe ab. Es entspricht in seiner Funktion dem UNIX-Kommando "`basename`".

Beispiel 7.6
```
use File::Basename;
print basename("/usr/lib/libc.a", '\..*'),"\n";
                                          ☞ libc
```

`dirname` — ist das Gegenstück von "`basename`, da es als Ergebnis den Pfad zum Verzeichnis einer Datei liefert ähnlich dem UNIX-Kommando "`dirname`".

Beispiel 7.7
```
use File::Basename;
print dirname("/usr/lib/libc.a"),"\n";
                                          ☞ /usr/lib
```

`fileparse` — ist eine Kombination von `dirname` und `basename`. Das Ergebnis ist hier eine dreielementige Liste, die den gekürzten Dateienamen ohne Suffix, den Pfad und den Suffix selbst enthält.

Beispiel 7.8

```
use File::Basename;
while (</usr/lib/libc*>) {
    ($name, $path, $suffix) = fileparse($_, '\..*');
    printf "%-10s %-10s %s\n", $name, $path, $suffix;
}
```

Auf dem Linux-Rechner des Autors liefert das Programm aus Beispiel 7.8 folgende Ausgabe:

```
libc      /usr/lib/ .a
libc      /usr/lib/ .sa
libc_p    /usr/lib/ .a
libcurses /usr/lib/ .a
libcurses /usr/lib/ .sa
```

Interessant ist, daß das Modul “`Basename`” bei der Zerlegung eines Dateinamens die Konventionen des jeweiligen Betriebssystems (UNIX, VMS, MSDOS, MacOS) beachtet.

`Benchmark` — ermöglicht auf einfache Weise Benchmarktests von Perl-Programmen. Beispiel 7.9 zeigt wie gering in diesem speziellen Fall der Zeitaufwand für das Sortieren eines arrays ist. Weitere Informationen sind über den Manualeintrag “`Benchmark`” zu finden. `Benchmark`

Beispiel 7.9

```
use Benchmark;
use Text::Abbrev;
sub sort_ab {
    abbrev *ab, @_;
    sort keys %ab;
}
sub nosort_ab {
    abbrev *ab, @_;
    keys %ab;
}
@data = qw(fischers fritze faengt frische fische);
timethese(5000, {
    'sorted' => 'sort_ab @data',
    'unsorted' => 'nosort_ab @data'});
☞ Benchmark: timing 5000 iterations of sorted, unsorted...
      sorted: 34 secs (34.00 usr 0.00 sys = 34.00 cpu)
    unsorted: 33 secs (32.49 usr 0.00 sys = 32.49 cpu)
```

`carp` — hilft die Ausagekraft von Fehlermeldungen zu verbessern. Wenn in Unterprogrammen statt “`warn`” der Befehl “`carp`” eingesetzt wird, erhält man im Fehlerfall auch die Stelle gemeldet, von der das Unterprogramm aufgerufen wurde. `Carp`

Beispiel 7.10

```
use Carp;
sub somesub {
```

```
    warn "Einfache Warnung";          ☞ Einfache Warnung at line 3.
    carp "Aufuehrliche Warnung";      ☞ Aufuehrliche Warnung at line 4
                                         main::somesub called at line 6
}
somesub;
```

Der “die”-Befehl kann entsprechend durch “croak ersetzt werden.

DB_FILE
dbm(3)

DB_FILE — bietet eine Schnittstelle zum Zugriff auf Datenbanken der dbm-C-Bibliothek. An dieser Stelle soll nur am Beispiel 7.11 gezeigt werden, wie überraschend einfach der Zugriff auf z. B. die mail-aliases-Datenbank des Betriebssystems zugegriffen werden kann. Der `tie`-Befehl bindet eine Assoziation (hash) mit einer Datenbank. Der Zugriff auf die einzelnen hash-Elemente wird danach über die dbm-C-Biliothek abgewickelt.

Beispiel 7.11

```
use AnyDBM_File;
use Fcntl;
$aliases = -r '/etc/mail/aliases.pag'
          ? '/etc/mail/aliases'
          : '/etc/aliases';
tie %alias, AnyDBM_File, $aliases, O_RDONLY, 0;
# get all aliases containing /berger/
foreach $key (grep /berger/, keys %alias) {
    my ($val) = $alias{$key};
    map { s/\0$// } ($val,$key); # remove trailing nul char
    printf "%s: %s\n", $key, $val;
}
# check for specific alias
print "eggenberger: ",$alias{"eggenberger\0"}||"<???>","\n";
untie %alias;
print "eggenberger: ",$alias{"eggenberger\0"}||"<???>","\n";
```

Die Ausgabe könnte zum Beispiel so aussehen:

```
uwe.berger: Uwe.Berger@ifi
reissenberger: Reissenberger@ifi
eggenberger: Eggenberger@ifi
wolfgang.reissenberger: Wolfgang.Reissenberger@ifi
otto.eggenberger: Otto.Eggenberger@ifi
berger: berger@ifi
eggenberger: Eggenberger@ifi
eggenberger: <???>
```

Die beiden letzten Zeilen zeigen, wie der Inhalt von `%alias` nach dem `untie`-Befehl “verschwindet”. `untie` löst die Bindung von der Datenbank. Das Beispiel 7.11 funktioniert in dieser Form unter den Betriebssystemen Linux und Solaris, sofern entsprechende aliases-Daten verfügbar sind.

`Sx` — Schnittstelle zur Athena- und Xlib-Bibliothek des X11-Fenstersystems. Sx bietet einfachen Zugang zur X11-Programmierung. Das Modul ist nicht im Perl–5-Paket enthalten, sondern muß getrennt installiert werden. Die Bezugsquellen sind in der Perl-FAQ aufgeführt.

Ein einfaches "Hello World"-Programm ist Beispiel 7.12.

Beispiel 7.12

```
use Sx;
MakeLabel("Hello World!");
MainLoop;
```

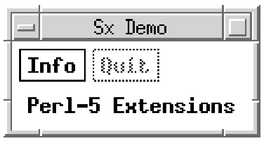
☞ 

Ein umfangreicheres Beispiel ist 7.13. Es beinhaltet zwei Buttons und einen Textbereich.

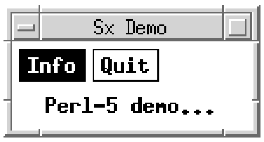
Beispiel 7.13

```
use Sx;
# — CallBack Routines —
sub quit {
     exit if GetYesNo('Really quit?');
}
sub info {
     local($w, $t, $s) = @_;
     $state ^= 1;
     SetLabel($w_label, $state ? "Perl-5 demo..." : "... of Sx");
     SetWidgetState($w_quit, 1);
}
# — Setup —
OpenDisplay('Sx Demo', @ARGV)
          or die "$0: cannot open display";
$w_info = MakeToggle("Info", 0, undef, 'info', 1);
$w_quit = MakeButton("Quit", 'quit');
SetWidgetPos($w_quit, $Sx::PLACE_RIGHT,
          $w_info, $Sx::NO_CARE, undef);
SetWidgetState($w_quit, 0);
$w_label = MakeLabel('Perl-5 Extensions');
SetWidgetPos($w_label, $Sx::PLACE_UNDER,
          $w_info, $Sx::NO_CARE, undef);
ShowDisplay();
MainLoop();
```

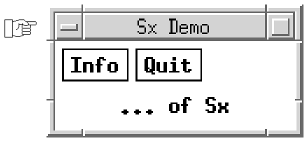
Nach dem Start des Programms aus Beispiel 7.13 erscheint dieses Fenster mit zwei Buttons und einem Textfeld. Der "`Quit`"-Button ist (noch) nicht auswählbar.

☞ 

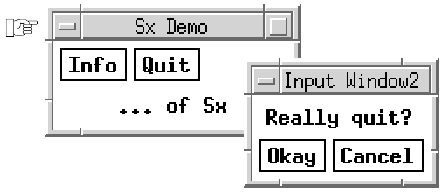
Wird mit der Maus der "`Info`"-Button angewählt wird, ändert sich der Text und der "`Quit`"-Button wird aktiv.

☞ 

Nach wiederholtem Anklicken des "`Info`"-Buttons wechselt der Text im Fenster zwischen "`Perl-5 demo...`" und "`... of Sx`".



Sobald der "`Quit`"-Button aktiviert wird, erscheint ein weiteres Fenster, das eine Bestätigung verlangt. Das Programm wird beendet, durch Anklicken von "`Okay`".



Die zur Zeit verfügbare Version 1.6 von Sx enthält leider ein Handicap: die Dokumentation der Schnittstelle ist lückenhaft und z. T. sogar fehlerhaft. Ebenfalls fehlt eine Installationsanweisung.

Liste aller Standardmodule

Es gibt noch mehr Module, deren Beschreibung jedoch den Umfang dieses Seminarbeitrags bei weitem sprengen würde.

Module lassen sich in die drei folgenden Gruppen einteilen.

`perlmod:` *Pragmatic Modules*

Pragmatic Modules — beeinflussen das Verhalten des Perl-Compilers.

`perlmod:` *Standard Modules*

Standard Modules — gehören zum Perl–5-Standard. Sie sind zusammen mit den *Pragmatic Modules* in jeder Perl–5-Installation verfügbar.

`perlmod:` *Extension Modules*

Extension Modules — Module, die größtenteils in C (oder einer Mixtur aus Perl und C) geschrieben sind und während der Laufzeit von Perl dynamisch nachgeladen werden können. Über die Verfügbarkeit dieser Module gibt die Perl-FAQ Auskunft. Zum Beispiel existieren Module zur Einbindung von Perl in SQL-Datenbanken oder für die Verwendung der curses-C-Bibliothek.

Zum leichten Auffinden der Dokumentation dieser Module werden im folgenden die englischen Kurzbeschreibungen aus den Manualeinträgen aller Module aufgeführt.

`Abbrev`	`abbrev` — create an abbreviation table from a list
`AnyDBMFile`	`AnyDBM_File` — provide framework for multiple DBMs
	`NDBM_File, ODBM_File, SDBM_File, GDBM_File` — various DBM implementations
`AutoLoader`	`AutoLoader` — load functions only on demand
`AutoSplit`	`AutoSplit` — split a package for autoloading
`Basename`	`Basename` — parse file specifications
	`fileparse` — split a pathname into pieces
	`basename` — extract just the filename from a path
	`dirname` — extract just the directory from a path
`Benchmark`	`Benchmark` — benchmark running times of code

`timethis`	— run a chunk of code several times	
`timethese`	— run several chunks of code several times	
`timeit`	— run a chunk of code and see how long it goes	
`carp`	— warn of errors (from perspective of caller)	`Carp`
`croak`	— die of errors (from perspective of caller)	
`confess`	— die of errors with stack backtrace	
`validate`	— run many filetest checks on a tree	`CheckTree`
`Collate`	— compare 8-bit scalar data according to the current locale	`Collate`
`Config`	— access Perl configuration option	`Config`
`getcwd`	— get pathname of current working directory	`Cwd`
`DB_File`	— Perl5 access to Berkeley DB	`DB_File`
`DynaLoader`	— Dynamically load C libraries into Perl code	`Dynaloader`
`English`	— use nice English (or awk) names for ugly punctuation variables	`English`
`Env`	— Perl module that imports environment variables	`Env`
`Exporter`	— module to control namespace manipulations	`Exporter`
`import`	— import functions into callers namespace	
`Fcntl`	— load the C Fcntl.h defines	`Fcntl`
`FileHandle`	— supply object methods for filehandles	`FileHandle`
`cacheout`	— keep more files open than the system permits	
`find`	— traverse a file tree	`Find`
`finddepth`	— traverse a directory structure depth-first	`Finddepth`
`getopt`	— Process single-character switches with switch clustering	`Getopt`
`getopts`	— Process single-character switches with switch clustering	
`Getopt::Long, GetOptions`	— extended getopt processing	`GetOptions`
`integer`	— Perl pragma to compute arithmetic in integer instead of double	`integer`
`less`	— Perl pragma to request less of something from the compiler	`less`
`MakeMaker`	— generate a Makefile for Perl extension	`MakeMaker`
`IPC::Open2, open2`	— open a process for both reading and writing	`Open2`
`IPC::Open3, open3`	— open a process for reading, writing, and error handling	`Open3`
`POSIX`	— Perl interface to IEEE 1003.1 namespace	`POSIX`

`Net::Ping, pingecho` — check a host for upness `Ping`

`sigtrap` — Perl pragma to enable stack backtrace on unexpected signals `sigtrap`

`strict` — Perl pragma to restrict unsafe constructs `strict`

`Socket` `Socket` — load the C socket.h defines

`subs` `subs` — Perl pragma to predeclare sub names

`Sx` `Sx` — front to all the Athena and Xlib garbage for Perl GUI progamming

Bis auf das Modul "`Sx`" gehören alle aufgeführten Module zum perl5.000-Paket.

7.4 Schnittstelle zwischen Perl und C

`perlapi`

Die Sprachen Perl und C/C++ können Unterprogramme der jeweils anderen Sprache verwenden. Dazu wurde das *"Perl–5 application programming interface for C extensions"* eingeführt.

`perlcall`
`perlguts`

Da es an dieser Stelle nicht sinnvoll ist, die Details dieser Schnittstelle und ihre Anwendung zu diskutieren, sei hier nur auf die entsprechenden Manualeinträge verwiesen: `perlapi`, `perlcall` und `perlguts`.

8 Schluß

Nachdem Perl Version 4 schon sehr leistungsfähig war, wurden die Einsatzmöglichkeiten der Sprache Perl mit der Version 5 nochmals um einiges vergrößert. Auch wenn die z. Z. verfügbare Dokumentation noch einige Schwachstellen, Lücken und Fehler hat, kann ich Perl als universelle Skript-Sprache unter UNIX empfehlen.

Bevor Sie demnächst eine Aufgabe mit "Bourne Shell", "C-shell", "awk", "sed" oder anderen Skript-Sprachen zu lösen versuchen, sollten Sie prüfen, ob Sie mit Perl schneller und eleganter zu einer Lösung gelangen können.

A Literatur

Ausgewählte Büchern und Dokumente zum Thema Perl:

1. Online verfügbare Dokumentation über den UNIX-Befehl "`man perl`".

2. Diskussionsforen `comp.lang.perl` (englisch) und `de.comp.lang.perl` (deutsch).

3. Perl-FAQ – Antworten auf häufig gestellte Fragen zum Thema Perl: wird monatlich über `comp.lang.perl` verteilt.

4. Larry Wall, Randal Schwartz. *"Programming Perl"* (Nutshell Handbook). O'Reilly & Associates. Bisherige Ausgaben des "Camel Book" behandeln nur Perl–4.

5. Randal Schwartz. *"Learning Perl"* (Nutshell Handbook). O'Reilly & Associates. Das "Llama Book" beschränkt sich auf Perl–4.

6. Ellie Quigley. *"Perl by Example"*. Prentice Hall. Ein Perl–4 Tutorial mit vielen Beispielen.

Im Laufe des Jahres 1995 sind durch die Freigabe von Perl Version 5 einige Neuerscheinungen und überarbeitete Perl-Handbücher zu erwarten. Dazu gehören...

1. eine überarbeitete Auflage des "Camel Book", *"Programming Perl"*; soll alle Perl–5 Neuheiten enthalten. Voraussichtlicher Erscheinungstermin: Frühjahr/Sommer 1995.

2. das "Llama Book", *"Learning Perl"*, wird nicht überarbeitet, sondern unverändert neu aufgelegt. Beschänkt sich also weiterhin auf Perl–4.

3. ein neues Buch, das zur Zeit unter dem Arbeitstitel *"Learning More Perl"* entsteht. Es soll die Lücke zwischen *"Learning Perl"* und Perl–5 schließen. Voraussichtlicher Erscheinungstermin: Frühjahr/Sommer 1995.