CGI Programmierung

Kristian Köhntopp

27. November 1996

Kapitel 1

Der Apache WWW-Server

Der Apache WWW–Server ist ein Server der zweiten Generation. Das bedeutet, er entstand auf der Grundlage der Erfahrungen, die beim Betrieb von WWW–Servern unter sehr hoher Last gemacht wurden, und er bietet erweiterte Möglichkeiten zur dynamischen Erzeugung von Seiten über CGI (*Common Gateway Interface*) und über eine servereigene Programmierschnittstelle (*Server API*).

Der Server selbst ist auf praktisch jeder modernen UNIX–Plattform einsetzbar, und Portierungen auf Windows NT und OS/2 sind ebenfalls verfügbar.

1.1 Apache übersetzen

Wie es sich für ein frei verfügbares UNIX–Programm gehört, ist der Server in Quelltextform erhältlich. Zusätzlich werden auf `http://www.apache.org/` auch Binärversionen für die verbreitesten UNIX–Versionen bereitgestellt. Ist die eigene UNIX–Version nicht unter diesen zu finden oder möchte man den Server mit experimentellen Optionen betreiben, muß man ihn sich selbst übersetzen. Zum Glück ist das nicht sehr schwierig.

Apache kommt, wie üblich, als mit `gzip` gepacktes `tar`–Archiv. Nach dem Auspacken mit

```
/tmp$ gzip -dc apache*tar.gz | tar xvf -
```

entsteht ein Verzeichnis mit dem Namen `apache_x.y.z`, wobei `x.y.z` die Versionsnummer des Servers angibt. Es enthält die folgenden Unterverzeichnisse:

cgi-bin Dieses Verzeichnis enthält einen Satz Beispielprogramme zur CGI–Programmierung in Form von Scriptdateien.

cgi-src Dieses Verzeichnis enthält weitere Beispielprogramme zur CGI–Programmierung, die in C geschrieben wurden. Diese Programme müssen zunächst compiliert werden, bevor sie im Verzeichnis `cgi-bin` installiert werden können.

conf Dieses Verzeichnis enthält Beispiel-Konfigurationsdateien. Die Namen dieser Konfigurationsdateien enden auf `-dist`. Zum Gebrauch muß die jeweilige Konfigurationsdatei kopiert und unbedingt für den lokalen Bedarf angepaßt werden.

icons Die eingebaute Apache–Funktion zur Anzeige von Verzeichnissen kann Dateien mit dateitypspezifischen Icons anzeigen. Dieses Verzeichnis enthält einen Satz Beispiel-Icons, die vom Server dazu verwendet werden.

logs Dieses Verzeichnis ist zunächst leer. Der Server wird später im Betrieb seine Logbücher hier ablegen. Diese Logbücher können je nach Serverauslastung sehr umfangreich werden.

src Dieses Verzeichnis enthält den C–Quelltext des Servers. Es wird für den Betrieb nicht benötigt.

support Dieses Verzeichnis enthält weitere C-Quelltexte zu Hilfsprogrammen für den Serverbetrieb. Sie müssen compiliert werden, wenn man Eigenschaften des Servers für Fortgeschrittene nutzen möchte: Benutzeridentifikation mit Paßworten, DBM–Benutzerdatenbanken für sehr große Benutzerzahlen und verschiedene andere.

1.1.1 Konfigurationsdatei anpassen

Im `src`–Verzeichnis befindet sich die Datei `Configuration`. Dies ist die einzige Datei, die angepaßt werden muß, um den Apache–Server für eine unterstützte Plattform zu übersetzen. In dieser Datei ist auch einzustellen, welche Module in den Server eingebunden werden sollen.

Folgende Einstellungen sind mit einem Texteditor an dieser Datei vorzunehmen:

`CC=` Wahl des C–Compilers. Der Server kann entweder mit dem normalen Systemcompiler `cc` oder mit dem GNU–C–Compiler `gcc` übersetzt werden, wenn dieser auf dem System vorhanden ist.

`CFLAGS=` Optionen für den Aufruf des C–Compilers. Mit dieser Variable können die Optionen für den Lauf des Compilers festgelegt werden. Normalerweise wird man hier den Optimierungslevel des Compilers angeben, also beispielsweise `-O2`.

Für den späteren Betrieb des Servers mit *Server Side Includes* kann hier die Option `-DXBITHACK` mit angegeben werden. Für schnelleren Betrieb auf Kosten der Lesbarkeit von Logfiles kann hier die Option `-DMINIMAL_DNS` mit angegeben werden. Die Option `-DMAXIMAL_DNS` führt dagegen für alle Clients eine doppelte Abfrage des Nameservers durch, was Fälschungen von Hostnamen erschwert, aber sehr langsam ist.

Für den normalen Betrieb sollten diese Optionen zunächst einmal weggelassen werden. Sie sind nicht unmittelbar notwendig und haben Nebeneffekte, die verstanden sein müssen, bevor die Optionen sinnvoll einsetzbar sind.

`LFLAGS=` Optionen für den Aufruf des Linkers. Mit dieser Variable können die Optionen für den Lauf des Linkers festgelegt werden. Normalerweise ist diese Variable leer.

`EXTRA_LIBS=` Optionen für das linken zusätzlicher Bibliotheken. Mit dieser Variable können zusätzliche Bibliotheken zum Linken des Programmes angegeben werden. Normalerweise ist diese Variable leer.

`AUX_*=` Betriebssystemspezifische Optionen und Bibliotheken. Diese Variablen stehen in Blöcken nach Betriebssystemen sortiert beisammen. Sie legen fest, für welche Betriebssystemversion der Server übersetzt werden soll. Im Auslieferungszustand ist der Server für SunOS 4 konfiguriert. Diese Einstellung **muß** für das Zielbetriebssystem angepaßt werden. Dazu ist die Zeile für SunOS 4 zu deaktivieren und die Zeile für das gewünschte Betriebssystem zu aktivieren. Um den Apache–Quelltext zum Beispiel für die Übersetzung auf Solaris 2 zu konfigurieren, muß der Block

```
# For SunOS 4
AUX_CFLAGS= -DSUNOS4
# For Solaris 2.
#AUX_CFLAGS= -DSOLARIS2
#AUX_LIBS= -lsocket -lnsl
```

folgendermaßen geändert werden:

```
# For SunOS 4
#AUX_CFLAGS= -DSUNOS4
# For Solaris 2.
AUX_CFLAGS= -DSOLARIS2
AUX_LIBS= -lsocket -lnsl
```

Weiter unten in der Konfigurationsdatei befindet sich der Abschnitt zur Konfiguration der Module, aus denen der Server zusammengesetzt wird. Die Einstellungen in diesem Bereich sind in der Regel für den Normalbetrieb korrekt und brauchen nicht verändert zu werden. Die Kommentare in diesem Abschnitt der Konfigurationsdatei geben weitere Hinweise.

1.1.2 Den Server übersetzen

Nachdem die Konfigurationdatei angepaßt ist, müssen die Informationen aus dieser Datei in den C–Quelltext des Programmes eingearbeitet werden. Dies geschieht automatisch durch den Aufruf des Scriptes `Configure`:

```
/tmp/apache_1.0.3/src$ ./Configure
Using 'Configuration' as config file
```

Danach kann der Server übersetzt werden. Der Übersetzungsprozeß wird durch das UNIX–Werkzeug `make` automatisch gesteuert. Es genügt, im `src`–Verzeichnis `make` aufzurufen:

```
/tmp/apache_1.0.3/src$ make
gcc -c -O2 -DLINUX alloc.c
...
```

Nach erfolgreichem Abschluß des Übersetzungsvorgangs wird im `src`–Verzeichnis eine Datei `httpd` hinterlassen, die ausführbar ist und mit dem `file`–Kommando als Programm identifiziert wird.

```
/tmp/apache_1.0.3/src$ ls -l ./httpd
-rwxr-xr-x   1 root     root        82260 Nov  4 10:42 ./httpd
/tmp/apache_1.0.3/src$ file ./httpd
./httpd: ELF 32-bit LSB executable i386 (386 and up) Version 1
```

1.1.3 Die Hilfsprogramme übersetzen

Für den einfachen Serverbetrieb genügt es, das Serverprogramm `httpd` zur Verfügung zu haben. Für den fortgeschrittenen Betrieb ist es jedoch nützlich, auch die optionalen Hilfsprogramme in den Verzeichnissen `cgi-src` und `support` zu übersetzen und zu installieren. Zum Glück sind diese Programme kaum systemspezifisch und können mit einem einfachen Aufruf von `make` in den betreffenden Verzeichnissen übersetzt werden:

```
/tmp/apache_1.0.3/cgi-src$ make
gcc -c -g query.c
...
/tmp/apache_1.0.3/cgi-src$ cd ../support/
/tmp/apache_1.0.3/support$ make
gcc -g htpasswd.c -o htpasswd
...
```

Eventuell bei der Übersetzung auftretende Warnungen sind normal und können ignoriert werden.

1.1.4 Den Server installieren

Neben dem eigentlichen Serverprogramm `httpd` werden zum Betrieb des Servers noch verschiedene zusätzliche Verzeichnisse benötigt. Alle diese Dateien und Verzeichnisse sollen in einem gemeinsamen Sammelverzeichnis installiert werden, das in der Terminologie von Apache als *ServerRoot* bezeichnet wird.

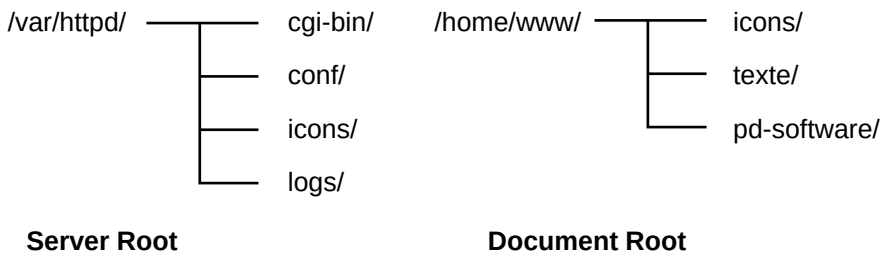


Abbildung 1.1: Verzeichnisstruktur der *ServerRoot*– und *DocumentRoot*–Verzeichnisse.

Viele Serverbetreiber wählen als ServerRoot das Verzeichnis `var/httpd`. Dieser Text geht ebenfalls davon aus, daß mit dieser ServerRoot installiert wird.

Außerdem wird der Server in Zukunft auf einen weiteren Verzeichnisbaum zugreifen, der die HTML–Dokumente enthält, die geservt werden sollen. Diese zweite Verzeichnishierarchie wird als *DocumentRoot* bezeichnet. Im Gegensatz zur ServerRoot ist die Verzeichnisstruktur der DocumentRoot selbst vollkommen frei wählbar. Die in Abbildung 1.1 gegebene Aufteilung ist nur ein Beispiel.

Beide Verzeichnishierarchien sind unterschiedlich und sollen sich nicht überlappen. Auf gar keinen Fall sollte die ServerRoot unterhalb der DocumentRoot liegen, denn sonst könnte man über den WWW-Server auf seine Konfigurationsdateien und Paßworte zugreifen, und das ist eine ausgesprochen schlechte Idee. Auch anders herum sollte die DocumentRoot nicht unterhalb der ServerRoot liegen, obwohl das nicht unmittelbar schädlich ist. Viele Serverbetreiber wählen als DocumentRoot das Verzeichnis `/home/www`, und das ist auch das Verzeichnis, das in diesem Text verwendet wird.

Zur Installation ist es notwendig, die Verzeichnisse für die ServerRoot und die DocumentRoot anzulegen und ihnen die notwendigen Rechte zu geben. Zunächst wird das ServerRoot–Verzeichnis angelegt und der Server selber in ihm installiert:

```
/var# mkdir httpd
/var# chown root.root httpd
/var# chmod 755 httpd
/var# cp /tmp/apache_1.0.3/src/httpd httpd/httpd
/var# chown root.root httpd/httpd
/var# chmod 111 httpd/httpd
```

Danach müssen die vorgefertigten Dateien aus den Verzeichnissen `cgi-bin`, `conf` und `icons` übernommen werden.

```
/var# cd httpd
/var/httpd# mkdir cgi-bin conf icons logs support
/var/httpd# chown root.root cgi-bin conf icons logs support
/var/httpd# chmod 755 cgi-bin conf icons logs support
/var/httpd# cp /tmp/apache_1.0.3/cgi-bin/* cgi-bin/
/var/httpd# cp /tmp/apache_1.0.3/icons/* icons/
/var/httpd# cp /tmp/apache_1.0.3/conf/* conf/
/var/httpd# cp /tmp/apache_1.0.3/support/htpasswd support/
/var/httpd# cp /tmp/apache_1.0.3/support/httpd_monitor support/
/var/httpd# cp /tmp/apache_1.0.3/support/unescape support/
```

Schließlich muß auch die DocumentRoot noch angelegt werden. Dokumente auf dem WWW–Server sollen unter Umständen auch durch normale Benutzer erzeugt werden dürfen. Es ist also sinnvoll, eine Benutzergruppe `webmaster` anzulegen und dieser Benutzergruppe Schreibrechte an der DocumentRoot zu geben. Dies ist recht einfach auf die folgende Weise möglich:

```
/home# echo "webmaster::60001:" >> /etc/group
/home# mkdir www
/home# chown kris.webmaster www
/home# chmod 775 www
```

Unterhalb dieses Verzeichnisses kann dann die benötigte Dokumentenstruktur nach Bedarf durch beliebige Benutzer der Gruppe `webmaster` angelegt werden. Benutzer werden der Benutzergruppe `webmaster` zugefügt, indem ihre Benutzernamen im vierten (letzten) Feld der entsprechenden Zeile in der Datei `/etc/group` eingetragen werden. Die betreffenden Benutzer müssen sich neu einloggen, damit die Änderung wirksam wird.

1.2 Apache konfigurieren

Dieser Abschnitt beschreibt die Konfiguration des WWW–Servers gerade weit genug, um den Server in Betrieb nehmen zu können. Eine vollständige Beschreibung der Apache–Konfiguration befindet sich im Apache Referenzhandbuch (`manual.ps`), das Bestandteil der Serverdistribution ist.

1.2.1 Aufrufoptionen

Der Apache WWW–Server versteht nur wenige Aufrufoptionen. Die eigentliche Konfiguration des Servers erfolgt über Konfigurationsdateien, und die Aufrufoptionen des Servers beschränken sich darauf, dem Server mitzuteilen, wo sich diese Konfigurationsdateien befinden. Die Optionen sind im Einzelnen:

`-d` *serverroot* Setzt den ursprünglichen Wert der Variablen `ServerRoot` auf *serverroot*. Dies kann in der Konfigurationsdatei mit dem Kommando `ServerRoot` überschrieben werden. Der Defaultwert ist `/usr/local/etc/httpd`.

`-f` *config* Führt die Kommandos in der Datei *config* beim Start des Servers aus. Wenn *config* nicht mit einem `/` beginnt, wird der Pfadname relativ zur ServerRoot interpretiert. Der Defaultwert ist `conf/httpd.conf`.

`-X` Der Server wird im Debugmodus gestartet. Er erzeugt keine Kindprozesse und löst sich auch nicht von der Console, um in den Hintergrundmodus zu gehen.

`-v` Druckt die Versionsnummer des `httpd` und beendet sich dann.

`-?` Druckt eine Liste der Optionen des Servers und beendet sich dann.

Um den Server zu starten, ist es also notwendig, die benötigten Konfigurationdateien zu erzeugen. Der Server kann dann entweder mit dem Kommando

```
~# /var/httpd/httpd -f /var/httpd/conf/httpd.conf
```

oder mit dem Kommando

```
~# /var/httpd/httpd -d /var/httpd
```

aktiviert werden.

1.2.2 Minimale Konfigurationsdateien

Der Server liest beim Start drei Konfigurationsdateien ein. Jede dieser Dateien kann Schlüsselworte enthalten, die die Werte bestimmter Variablen festlegen und so den Betrieb des Servers steuern. Die Pfadnamen dieser Dateien werden wie üblich relativ zur ServerRoot interpretiert, wenn sie nicht mit einem `/` beginnen. Standardmäßig sind dies die folgenden Dateien:

`conf/httpd.conf` Diese Datei enthält globale Steuerparameter für den WWW–Server.

`conf/srm.conf` Diese Datei steuert das Server Ressource Managment, d.h. sie legt fest, welche URLs auf welche Pfade im Dateisystem abgebildet werden und ob es sich bei den durch URLs referenzierten Dokumenten um statische Dateien oder um dynamisch erzeugte Dokumente handelt. Diese Variable kann mit dem Schlüsselwort `RessourceConfig` überschrieben werden.

`conf/access.conf` Mit Hilfe dieser Datei können global Zugriffsrechte für bestimmte Verzeichnisse vergeben werden. Mit dem Schlüsselwort `AccessConfig` kann eine alternative Konfigurationsdatei bestimmt werden.

Weiterhin legt der Server drei Dateien im Verzeichnis `logs/` an. Einige dieser Dateien sind Logbücher, die je nach Serverlast sehr schnell sehr groß werden können. Es ist notwendig, diese Dateien regelmäßig zu sichern und den Server danach mit einem Signal „zu wecken“, damit er neue Logdateien aufsetzt. Die Dateien sind:

`logs/httpd.pid` Diese Datei wächst nicht und braucht auch nicht gesichert zu werden. Sie enthält in ASCII die PID des Serverprozesses. Dadurch ist es leicht möglich, den Server mit dem Kommando

```
~# kill -HUP ‘cat /var/httpd/logs/httpd.pid‘
```

zu wecken. Der Hauptprozeß des Servers wird alle seine Kindprozesse von der Änderung informieren, so daß ein einziges Signal zur Steuerung des Servers ausreichend ist.

Mit dem Schlüsselwort `PidFile` ist es möglich, diesen Pfadnamen zu ändern.

`logs/error_log` In dieser ständig wachsenden Datei werden alle Fehler beim Zugriff auf Komponenten des Servers mitgeloggt. Insbesondere finden sich hier auch die Fehlerausgaben von CGI–Scripten. Sollten solche Scripte also einmal nicht funktionieren, kann man dieser Datei die dabei entstehenden Meldungen entnehmen.

Mit dem Schlüsselwort `ErrorLog` kann eine andere Logdatei festgelegt werden. Für virtuelle Hosts können unterschiedliche Logdateien bestimmt werden.

`logs/access_log` In dieser sehr schnell wachsenden Datei wird jeder Zugriff auf den Server geloggt. Es ist möglich, diese Datei zur Erzeugung von Zugriffstatistiken auszuwerten.

Mit dem Schlüsselwort `TransferLog` kann ein anderer Dateiname festgelegt werden. Für virtuelle Hosts können unterschiedliche Logdateien bestimmt werden.

Um den Server zu konfigurieren, kopiert man sich zweckmäßigerweise die mitgelieferten Beispiel–Konfigurationsdateien und paßt sie für die eigene Konfiguration an:

```
/var/httpd/conf# cp access.conf-dist access.conf
/var/httpd/conf# cp httpd.conf-dist httpd.conf
/var/httpd/conf# cp srm.conf-dist srm.conf
```

conf/httpd.conf

Die mit dem Server ausgelieferte beispielhafte Konfigurationsdatei `conf/httpd.conf` enthält eine Reihe von Schlüsselworten zur Konfiguration des WWW-Servers. Im Einzelnen:

`Port` Dieses Schlüsselwort legt fest, auf welchem TCP–Port der Server seine Dienste anbieten soll. In UNIX sind die Ports mit Portnummern kleiner 1024 geschützt und nur Programmen mit Systemverwaltungsrechten zugänglich. Der Server muß also vom Systemverwalter gestartet werden, wenn er auf dem HTTP–Standardport 80 serven soll.

Für Server, die mit Benutzerrechten gestartet werden, hat es sich eingebürgert, den frei verfügbaren Port 8080 zu verwenden. Die Konfigurationsdatei enthält dann eine Anweisung der Art `Port 8080`.

`User` **und** `Group` Wenn der Server durch einen Systemverwalter gestartet wird, verwendet er die Systemverwalterrechte nur, um sich an den angegebenen Port zu binden. Alle anderen Dinge, insbesondere den Zugriff auf Dateien und das Starten von externen Programmen, erledigt der Server nicht unter dieser Benutzerkennung, um böse Überraschungen zu vermeiden.

Mit den Anweisungen `User` und `Group` wird der Server statt dessen auf eine Benutzer– und Gruppennummer konfiguriert, die ungefährlicher ist. Die Angabe dieser Werte kann numerisch in der Form `#zahl` erfolgen, oder der Name eines Benutzers oder einer Benutzergruppe kann direkt gegeben werden. Beispiel:

```
User nobody
Group #60001
```

`ServerAdmin` Dieses Schlüsselwort teilt dem Server die Mailadresse der für die Serververwaltung zuständigen Person mit. Der Server nennt diese Adresse in Fehlermeldungen als Adresse eines Ansprechpartners. Zukünftige Versionen des Servers werden selbstständig Problemreports an diese Adresse senden.

`ServerRoot` Legt das ServerRoot–Verzeichnis fest. In unserer Beispielkonfiguration geschähe dies mit der Anweisung `ServerRoot /var/httpd`.

`PidFile`

`ErrorLog`

`TransferLog` Diese drei Schlüsselworte bestimmen, wie bereits im Abschnitt 1.2.2 beschrieben, die Lage der verschiedenen Logdateien.

`ServerName` Ein Webserver muß gelegentlich wissen, unter welchem kanonischen Namen er selbst anzusprechen ist. Apache versucht, dieses Namen automatisch zu ermittelt, aber wenn dies nicht gelingt oder wenn man sicher gehen will, ist es notwendig, dem Server seinen eigenen Namen in der Konfigurationsdatei mitzuteilen. Das erfolgt dann mit diesem Schlüsselwort. Der Parameter ist ein voll qualifizierter Domainname, zum Beispiel in `ServerName white.koehntopp.de`.

conf/srm.conf

Mit dieser Konfiguration legt der Serverbetreiber fest, wie der Server die verschiedenen Dokumententypen behandeln soll, die auf seinem Server verfügbar sind. Diese Datei definiert also die Zuordnung von Dateiendungen zu MIME–Typen, die Namen von Indexdateien für Verzeichnisse und den Umgang mit Landessprachen und dergleichen.

Minimal sollte die Datei die folgenden Anweisungen enthalten:

`DocumentRoot` Dieses Schlüsselwort legt die DocumentRoot des Servers fest. In unserer Beispielkonfiguration ist das: `DocumentRoot /home/www`.

`ScriptAlias` Dieses Schlüsselwort ist nur dann notwendig, wenn der Server in der Lage sein soll, CGI–Scripte zur dynamischen Erzeugung von Seiten auszuführen. Die Anweisung hat die Form

```
ScriptAlias <url-pfad> <realer Pfad>
```

und sie „verbindet“ den Pfad *url–pfad* mit dem wirklichen Pfad *realer Pfad*, d.h. immer dann, wenn ein Request für eine Seite unterhalb von *url–pfad* auf dem Server eingeht, wird das zugehörige Programm in *realer Pfad* gestartet und muß die angeforderte Seite berechnen.

Um das CGI-Verzeichnis `/var/httpd/cgi-bin` mit dem URL–Pfad `/cgi-bin` zu verbinden, muß also die Anweisung

```
ScriptAlias /cgi-bin/ /var/httpd/cgi-bin/
```

gegeben werden. Ein Request für die URL `/cgi-bin/dummy.pl` startet dann das Programm `/var/httpd/cgi-bin/dummy.pl`, das eine Seite erzeugen muß.

Es ist wichtig, daß das Verzeichnis mit den CGI–Programmen **nicht** unterhalb der DocumentRoot liegt. Überlappen sich die Verzeichnisse, kann es zu seltsamen Effekten kommen, weil der Server sich entscheiden muß, ob er das Script als Text ausliefert oder ob er es startet und die Ausgabe des Scriptes ausliefert.

Die beispielhafte `srm.conf` enthält weitere Anweisungen, die hier zunächst nicht behandelt werden sollen.

conf/access.conf

Die Konfigurationsdatei `conf/access.conf` regelt, wer auf welche Dateien des Servers zugreifen darf. Es ist möglich, einzelne Dateien oder ganze Teilbäume in der Verbreitung zu beschränken, so daß man nur von bestimmten Rechnern aus auf die Dateien zugreifen darf oder daß man sich als Benutzer zunächst mit einem Paßwort gegenüber dem Server identifizieren muß, bevor man auf Seiten zugreifen kann. Auch Kombinationen von Schutzmechanismen sind möglich.

Da sich die Konfigurationsanweisungen in dieser Datei immer auf Verzeichnisse beziehen, ist ihr Inhalt konsequent in

```
<directory /pfad/*/name>
...
</directory>
```

gegliedert. Dabei ist zu beachten, daß die `<directory>`–Anweisungen vielleicht wie HTML–Tags aussehen, aber keine sind. Sie müssen auf jeden Fall einzeln in einer Zeile stehen und dürfen auch keine optionalen Parameter wie HTML–Tags haben. Die Pfade, die in `<directory>`–Anweisungen genannt werden, sind immer physikalische Pfade des Dateisystems und keine URL–Pfade.

Innerhalb eines `<directory>`–Abschnittes können dann mit verschiedenen Anweisungen Optionen und Zugriffsrechte für die in diesem Abschnitt spezifizierten Verzeichnisse festgelegt werden.

Unsere Beispielkonfiguration soll ohne weitere Erklärungen zunächst einmal die folgenden Anweisungen enthalten:

```
<Directory /var/httpd/cgi-bin>
Options Indexes FollowSymLinks
</Directory>

<Directory /home/www>
Options Indexes
AllowOverride None
</Directory>
```

Kapitel 2

CGI–Programmierung

Oftmals möchte man HTML-Seiten mit dynamisch erzeugtem Inhalt haben. Solche Seiten sind nicht fest in Verzeichnissen des Servers abgelegt, sondern werden beim Abruf durch ein Programm erzeugt. Das erzeugende Programm muß vom Server auf eine bestimmte Weise aufgerufen werden, damit es weiß, welche Art von Seite es erzeugen soll und es muß seine Ausgabe auf eine bestimmte Weise erzeugen, damit der Server diese Ausgabe als eine WWW–Seite interpretieren kann. Die Norm, nach der das externe Programm und der WWW–Server zusammenspielen, wird als das *Common Gateway Interface* (*CGI*) bezeichnet.

2.1 Aktivierung von CGI–Programmen durch Apache

Grundsätzlich kennt der Apache WWW–Server vier verschiedene Möglichkeiten, externe Programme zur Erzeugung von WWW–Seiten einzusetzen:

CGI-Pfade mit `ScriptAlias` Apache sieht wie alle ernstzunehmenden WWW–Server Möglichkeiten vor, mit denen man man URL–Pfade mit Scriptverzeichnissen verbinden kann. In Apache geschicht dies, wie in Abschnitt 1.2.2 beschrieben, mit der Anweisung `ScriptAlias`.

Zugriffe auf Seiten unterhalb der durch den `ScriptAlias` spezifizierten URL bewirken, daß das angegebene Programm gestartet wird und die Seite berechnet.

Programme mit der Endung `.cgi` Anstatt CGI–Scripte in bestimmten Verzeichnissen unterzubringen, kann Apache solche Scripte auch an einer bestimmten Endung erkennen, unabhängig davon, in welchem Verzeichnis sie sich befinden. Standardmäßig verwendet man für solche Scripte die Endung `.cgi`, aber es kann auch jede beliebige andere noch nicht vergebene Dateiendung konfiguriert werden.

Verantwortlich für den Start von Scripten mit der Endung `.cgi` ist das Apache–Modul `mod_cgi`. Es bewirkt, daß alle Dateien, die über die Erkennung von Dateiendungen mit dem MIME–Type `application/x-httpd-cgi` gekennzeichnet werden, nicht als statische Texte ausgeliefert werden. Der Server versucht statt dessen die Datei als Programm zu starten und liefert die Ausgabe des Programms an den Abrufer zurück. Mit der Anweisung

```
AddType application/x-httpd-cgi .cgi
```

in der `conf/srm.conf`–Konfigurationsdatei wird dem Server mitgeteilt, welche Dateiendungen diesen MIME–Type haben. Im Beispiel wird die Standardzuordnung definiert, die `.cgi`–Dateien als ausführbar markiert, aber es ist prinzipiell möglich, andere und auch mehrere Dateiendungen zu CGI–Endungen zu erklären. Bis auf die bequemere Installation (kein besonderes Verzeichnis notwendig) unterscheidet sich dieses Verfahren nicht vom Programmstart mit `ScriptAlias`.

Server Side Includes (SSI) Manchmal möchte man nicht eine ganze WWW–Seite durch ein Script berechnen lassen, sondern nur einfache Textersetzungen auf einer Seite vornehmen oder nur Teile einer Seite von einem Script erzeugen lassen. Solche Seiten sind ein typischer Anwendungsfall für SSI. SSI–Dateien werden vom Server nicht unmittelbar ausgeliefert, sondern der Server erwartet in der Datei HTML–Kommentare. Einige, spezielle Kommentare werden vom Server als Steueranweisungen interpretiert. Der Server ersetzt in diesem Fall den gesamten betroffenen Kommentar durch das Ergebnis der Steueranweisung.

SSI wird durch das Apache–Module `mod_include` verwaltet. Es wird vom Server auf alle Dateien mit dem MIME–Type `text/x-server-parsed-html` angewendet. Wieder kann man mit der `AddType`–Anweisung in der Ressourcenkonfiguration bestimmten Dateiendungen diesen MIME–Type zuordnen:

```
AddType text/x-server-parsed-html .shtml
```

Wie im Beispiel gezeigt, wird standardmäßig die Dateiendung `.shtml` verwendet, um Seiten mit SSI zu markieren. Wenn die Servererweiterung `XBITHACK` bei der Übersetzung des Servers aktiviert war und in der Ressourcenkonfiguration durch die Anweisung

```
XBitHack on
```

aktiviert ist, dann werden auch normale `.html`–Dateien als SSI–Dateien erkannt, sobald das `x`–Recht für User an der betreffenden Datei gesetzt ist.

Falls die Variable auf den Wert `full` statt auf `on` gesetzt wird, testet Apache außerdem noch das `x`–Recht für Gruppen an der Datei. Ist es gesetzt, wird ein `Last-Modified`–Header mit dem Dateidatum erzeugt. Ist es nicht gesetzt, wird auch kein `Last-Modified`–Header erzeugt. Auf diese Weise kann man das Verhalten von Caches und Proxies auf dem Weg zum Client beeinflussen.

Server Module (AAPI) In besonderes komplizierten Anwendungsfällen oder wenn es ganz besonders auf Geschwindigkeit ankommt, ist es nicht mehr ausreichend, den Server durch externe Programme zu erweitern. In diesem Fall muß die Erweiterung des Servers um weitere Eigenschaften statt dessen Bestandteil des Servers selbst werden.

Apache ist ein modularer Server, und da er im Quelltext vorliegt, ist sehr leicht möglich, den Server durch weitere Module zu erweitern. Module lassen sich besonders leicht einfügen, wenn sie sich an die vom Server bereitgestellte Programmierschnittstelle halten, das *Apache API*. Solche Module lassen sich außerdem auch mit dem experimentellen Modul `mod_dld` zur Laufzeit nachladen und ohne erneute Übersetzung des Servers einbinden.

Auch andere Server haben Modulschnittstellen, die leider meistens zueinander inkompatibel sind. Bekannt ist vor allen Dingen die *Netscape Server API* zur Programmierung von Modulen (*Plugins*) für die Netscape Serverfamilie.

2.2 Einfache CGI–Programme

Einfache CGI–Programme werden über einen normalen Link aktiviert und übernehmen keine Eingabeparameter. Sie können aber Informationen aus anderen Datenquellen beziehen, etwa aus den Logbuchdateien des Servers oder aus anderen Dateien auf dem Serverrechner.

Das Beispiel 2.1 zeigt das einfachste CGI–Programm überhaupt. Dieses Programm tut nichts weiter, als ein Dokument mit dem MIME–Type `text/plain` zu erzeugen und dann seine Ablaufumgebung auszudrucken. Es eignet sich damit ausgezeichnet als „Hello, World!“–Programm und dient gleichzeitig zum Erkunden der Ablaufumgebung für CGI–Programme:

Das CGI–Programm ist in der Scriptsprache der UNIX–Shell `/bin/sh` geschrieben, aber jede andere Programmiersprache ist ebenso geeignet. CGI–Programme müssen ihre Ausgabe auf der Standardausgabe

```
#! /bin/sh --

echo "Content-Type: text/plain"
echo
echo "Pfad:"
echo "$PATH"
echo
echo "Aktuelles Verzeichnis:"
pwd
echo
echo "UID/GID:"
id
echo
echo "Umgebungsvariablen:"
env | sort
echo
echo "Prozessliste:"
ps alxwww
```

Abbildung 2.1: Dieses einfache CGI–Programm ist das CGI–Äquivalent von „Hello, World!“. Zusätzlich druckt es noch Informationen über die Ablaufumgebung des Scriptes aus.

liefern und sie müssen der eigentlich nutzbaren Ausgabe einen Absatz voranstellen, der zwingend den MIME–Type der Ausgabe festlegt und weitere optionale Angaben über das erzeugte Dokument enthalten darf. Dieser Absatz muß durch eine Leerzeile vom nutzbaren Teil der Ausgabe getrennt sein.

Um das Programm ausprobieren zu können, muß es als CGI–Programm auf dem Server installiert werden. Das bedeutet: Entweder muß es als Datei mit einem beliebigen Namen im `cgi-bin`–Verzeichnis des Servers installiert werden, oder es muß als Datei mit der Endung `.cgi` in einem Verzeichnis der DocumentRoot installiert werden. In beiden Fällen muß die Datei als für das Betriebssystem ausführbar installiert werden, damit der Server das Programm starten kann.

In unserer Beispielkonfiguration kann das Programm wie folgt installiert werden:

```
/var/httpd# cp /tmp/hello cgi-bin/
/var/httpd# chmod 755 cgi-bin/hello
```

Mit einem Browser kann nun die URL `http://server/cgi-bin/hello` abgerufen werden, um das Script zu starten. Seine Ausgabe ist etwas länglich, aber sehr informativ: Wir erfahren den Pfadnamen, der vom Server und von unseren eigenen CGI–Programmen durchsucht wird, um externe Programme zu starten, und das Script nennt uns das aktuelle Verzeichnis, in dem unsere CGI–Programme ablaufen. Außerdem bekommen wir mitgeteilt, mit welchen Benutzerrechten das Script abläuft und welche Umgebungsvariablen ihm zur Verfügung stehen. Letztere Information ist besonders aufschlußreich, denn offenbar definiert das Common Gateway Interface eine ganze Liste von Umgebungsvariablen, die dem externen Programm vom WWW–Server zur Verfügung gestellt werden. Abbildung 2.2 zeigt eine leicht gekürzte, beispielhafte Ausgabe des Scriptes.

Die Bedeutung der verschiedenen Umgebungsvariablen eines CGI–Programmes wird in Abschnitt 2.3 erklärt. Die meisten Programme werten jedoch nur die Variable `QUERY_STRING` oder eine der (im Beispiel nicht erzeugten) Variablen `PATH_INFO` und `PATH_TRANSLATED` aus.

```
Pfad:
/bin:/usr/bin:/usr/local/bin

Aktuelles Verzeichnis:
/var/httpd/cgi-bin

UID/GID:
uid=65534(nobody) gid=65535(nogroup) groups=65535(nogroup)

Umgebungsvariablen:
DOCUMENT_ROOT=/home/www
GATEWAY_INTERFACE=CGI/1.1
HOSTTYPE=i386
HTTP_ACCEPT=image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
    image/png, image/tiff, multipart/x-mixed-replace, */*
HTTP_ACCEPT_LANGUAGE=de, en, fr, es, it
HTTP_HOST=white
HTTP_PRAGMA=no-cache
HTTP_USER_AGENT=OmniWeb/2.0.1 OWF/1.0
OSTYPE=Linux
PATH=/bin:/usr/bin:/usr/local/bin
QUERY_STRING=
REMOTE_ADDR=193.102.57.1
REMOTE_HOST=black
REQUEST_METHOD=GET
SCRIPT_FILENAME=/var/httpd/cgi-bin/hello
SCRIPT_NAME=/cgi-bin/hello
SERVER_ADMIN=kris@koehntopp.de
SERVER_NAME=white.koehntopp.de
SERVER_PORT=80
SERVER_PROTOCOL=HTTP/1.0
SERVER_SOFTWARE=Apache/1.0.3
SHELL=/bin/bash
SHLVL=1
TERM=dumb
_=/usr/bin/env

Prozessliste:
...
```

Abbildung 2.2: In der Ausgabe des „Hello, World“–Programmes ist die umfangreiche Liste der durch den Server bereitgestellten CGI–Variablen erkennbar.

2.3 Umgebung von CGI–Programmen

Webserver unter UNIX versorgen ihre CGI–Programme mittels Umgebungsvariablen mit den verschiedensten Fakten über den Aufrufer. Unter anderen Betriebssystemen werden andere Mechanismen zur Übermittlung der Information verwendet, aber die Art und die Namen der übermittelten Funktionen bleibt gleich.

2.3.1 Standard–Variablen

Der Client, den der Anwender benutzt, um das CGI–Programm auf dem Server zu aktivieren, liefert dem Server eine ganze Reihe von Informationen, die dieser auch dem CGI–Programm selbst zur Verfügung stellt. Das CGI–Programm kann diese Informationen auswerten und evtl. für die Gestaltung der Seite verwenden.

`SERVER_SOFTWARE` Der Name und die Versionsnummer des Servers, der das CGI–Programm gestartet hat.

Beispiel: Apache/1.0.3

`SERVER_NAME` Der Hostname des Servers, der das CGI–Programm gestartet hat. Ist dieser Name nicht verfügbar, wird hier eine IP–Nummer übergeben.

Beispiel: white.koehntopp.de

`SERVER_PORT` Die Portnummer des Servers, der das CGI–Programm gestartet hat.

Beispiel: 80

`SERVER_ADMIN` Die Mailadresse des Serveradministrators. Das CGI–Programm kann in Notfällen Fehlermeldungen an diese Adresse mailen.

Beispiel: kris@koehntopp.de

`GATEWAY_INTERFACE` Die Version der CGI–Spezifikation, nach diese Umgebungsvariablen erzeugt wurden.

Beispiel: CGI/1.1

`SERVER_PROTOCOL` Der Name und die Versionsnummer des Protokolls, mit dem der Request empfangen wurde, der das CGI–Programm aktiviert hat.

Beispiel: HTTP/1.0

`REQUEST_METHOD` Die Request–Methode, die für das Starten des CGI–Programms verantwortlich war. Diese Methode legt fest, auf welche Weise das CGI–Programm an seine Eingabeparameter kommt.

Beispiel: GET, HEAD oder POST.

Bei jedem Datentausch zwischen einem WWW–Browser und Server werden fast immer auch optionale Informationen in Form von zusätzlichen Request–Headerzeilen ausgetauscht. Anstatt für alle Eventualitäten vorauszuplanen und Header zu definieren, hat man in der CGI–Spezifikation festgelegt, daß solche zusätzlichen Headerzeilen in Uppercase mit dem Prefix `HTTP_` versehen werden sollen und an das CGI–Programm weitergegeben werden. Wenn der Browser also die optionale HTTP–Informationszeile *Bla: Fasel* einliefert, dann wird dem CGI–Programm die Umgebungsvariable `HTTP_BLA` mit dem Wert `Fasel` vorbelegt übergeben.

Die folgenden Variablen findet man in nahezu jeder CGI–Anfrage:

`HTTP_REFERER` Die URL des Dokumentes, das auf das CGI–Programm verwiesen hat. Auf dieser Seite hat der Benutzer also ein Link angeklickt, der auf das Programm verwiesen hat, oder ein Formular abgesendet, das das CGI–Programm aktiviert hat.

Beispiel: http://www.koehntopp.de/cgistarter.html

`HTTP_FROM` Die Mail–Addresse des Benutzers des Browsers. Die meisten Browser unterdrücken diese Information inzwischen, um die Privatsphäre des Benutzers zu schützen.

Beispiel: kris@koehntopp.de

`HTTP_HOST` Viele Serverrechner betreiben nicht nur einen einzigen Webserver, sondern eine ganze Palette von Servern für unterschiedliche Anbieter unter verschiedenen Namen. Diese Variable enthält, wenn sie vorhanden ist, den Namen, unter dem der Server angesprochen wurde.

Beispiel: www.netuse.de

`HTTP_USER_AGENT` Die Typbezeichnung und Versionsnummer der Browser–Software. Einige CGI–Programme werten diese Variable aus, um die Fähigkeiten des Browsers zu bestimmen und je nach Bedarf die Ausgabe anzupassen (Tabellen, Frames, Javascript, ActiveX, Java).

Beispiel: OmniWeb/2.0.1 OWF/1.0

`HTTP_ACCEPT` Eine Liste aller MIME–Types, die der Client als Antworttyp akzeptiert.

Beispiel: text/plain, text/html, image/*

Und schließlich gibt es noch eine ganze Gruppe von Variablen, die direkt Informationen über den Request enthalten, der das CGI–Programm gestartet hat:

`PATH_INFO` CGI-Programme können nach dem Pfadnamen zum Aufruf des Scriptes noch weitere Pfadkomponenten enthalten. Zum Beispiel könnte das Beispielprogramm aus Abbildung 2.1 mit der URL `http://server/cgi-bin/hello/more/info` aufgerufen werden. Die Variable hält in diesem Fall dann die „überstehenden“ Komponenten des Pfades zur Auswertung durch das Script bereit.

Beispiel: /more/info

`PATH_TRANSLATED` Falls das CGI–Programm mit einem verlängerten CGI–Pfad aufgerufen wurde, kann dieser Pfad vom WWW–Server auf die übliche Weise in einen wirklichen Dateinamen übersetzt werden. Das Ergebnis dieser Übersetzung ist dann in dieser Variable zu finden.

Beispiel: /home/www/more/info

`SCRIPT_NAME` Der Teil der URL, der zum Aufruf des CGI–Programmes geführt hat, ist in dieser Variablen in URL–Form zu finden.

Beispiel: /cgi-bin/hello

`SCRIPT_FILENAME` Die Information aus `SCRIPT_NAME` wird für das CGI–Programm ebenfalls in wirkliche Dateinamen übersetzt bereitgehalten.

Beispiel: /var/httpd/cgi-bin/hello

`QUERY_STRING` Diese Variable enthält eine Liste von Parametern, die dem CGI–Programm mitgegeben worden sind. Die Parameter liegen in codierter Form vor (Siehe Abschnitt 2.4) und müssen durch das CGI–Programm decodiert werden, bevor sie verwendet werden können.

Beispiel: name=Otto+Normaluser&phone=441777

`REMOTE_HOST` Der Name des Rechners, der den Request gemacht hat, soweit er verfügbar ist.

Beispiel: mahaki.koehntopp.de

`REMOTE_ADDRESS` Die IP–Nummer des Rechners, der den Request eingeliefert hat.

Beispiel: 192.103.57.2

`AUTH_TYPE` Die Methode, mit der die Useridentität in der Variablen `REMOTE_USER` verifiziert worden ist, falls eine Verifikation überhaupt stattgefunden hat.

Beispiel: Basic

`REMOTE_USER` Die verifizierte Identität des Benutzers, der den Request eingeliefert hat, wenn eine Verifikation überhaupt stattgefunden hat.

Beispiel: marit

`CONTENT_TYPE` Der MIME–Type der Daten, die mit einem PUT– oder POST–Request eingeliefert worden sind.

Beispiel: text/plain

`CONTENT_LENGTH` Die Länge der Daten in Bytes, die mit einem PUT– oder POST–Request eingeliefert worden sind.

Beispiel: 17321

2.3.2 Apache–Spezialvariablen

Apache stellt zusätzlich zu den im CGI–Standard definierten Variablen noch einige zusätzliche Informationen bereit.

`DOCUMENT_ROOT` Diese Variable enthält den Pfad zur `DocumentRoot` aus der Serverkonfiguration.

Beispiel: /home/www

`REDIRECT_...` Apache ist in der Lage, nach Maß zugeschnittene Fehlermeldungen zu erzeugen. Geschickterweise erzeugt man diese Fehlermeldungen durch ein CGI–Script. Damit das richtig funktioniert, muß Information über den alten Request vorhanden sein, der den Fehler hervorgerufen hat. Apache stellt also alle Variablen des alten Requests mit dem zusätzlichen Prefix `REDIRECT_` in der Fehlerbehandlung zur Verfügung.

Außerdem tauchen zwei zusätzliche neue Variablen auf, nämlich `REDIRECT_URL`, die URL, die den Fehler hervorgerufen hat, und `REDIRECT_STATUS`, der HTTP–Statuscode, der Auskunft über die Art des Fehlers gibt.

2.3.3 Zusätzliche Variablen

Mit dem Schlüsselwort `PassEnv` ist es möglich, weitere Variablen aus der Umgebung des Serverprozesses in die Umgebung des CGI–Scriptes weiterzugeben. Die Anweisung

```
PassEnv HOSTNAME
```

würde zum Beispiel bewirken, daß die Umgebungsvariable `HOSTNAME` aus der Serverumgebung auch in CGI–Programmen sichtbar wäre, wenn sie überhaupt definiert ist.

Mit dem Schlüsselwort `SetEnv` ist es außerdem möglich, gezielt Variablen in der Umgebung des CGI–Programmes zu erzeugen. Die Anweisung

```
SetEnv DEBUG_ZONE parser
```

würde für alle CGI–Programme die Umgebungsvariable `DEBUG_ZONE` mit dem Wert `parser` belegen. Dies ist, wie der Name der Variablen schon andeuten soll, ausgesprochen nützlich, um Debugging in CGI–Scripten zu aktivieren.

2.4 Parameterübergabe an CGI–Programme

CGI–Programme können durch direkte Links oder durch Formulare (siehe Kapitel 3) aktiviert werden. In beiden Fällen ist es möglich, dem gestarteten CGI–Programm Parameter mitzugeben. Für den Programmierer des CGI–Programmes sind dabei zwei Dinge interessant: die HTTP–*Methode*, mit der die Daten übergeben werden, und die *Codierung* der übergebenen Daten.

2.4.1 HTTP–Methoden

Wie die meisten Protokolle, die im Internet eingesetzt werden, ist auch HTTP ein Protokoll, das Daten in ASCII und zeilenorientiert übermittelt. Auf diese Weise ist ein Client oder Server für das Protokoll schnell mit Hilfe der C-Bibliothek programmiert, und das Programm läßt sich auch ohne Gegenstelle schnell mit `telnet` testen.

```
GET /cgi-bin/hello/more/info HTTP/1.0
Accept: text/html, image/gif, image/jpeg
User-Agent: Mozilla/2.02 (Windows; I; 32bit)
<Leerzeile>
```

Abbildung 2.3: Ein Beispiel für einen HTTP–Request: Ein Netscape–Client fordert den URL–Pfad `/cgi-bin/hello/more/info` an.

Ein Request, den ein HTTP–Client, etwa ein WWW–Browser, an den Server sendet, sieht beispielweise so wie in Abbildung 2.3 aus. Das Schlüsselwort `GET` signalisiert, daß eine Seite vom WWW–Server abgeholt werden soll, daß also in der Terminologie von HTTP die `GET`–Methode verwendet werden soll. Dahinter werden der URL–Pfad und die benutzte Protokollversion angegeben.

Die folgenden Zeilen des Requests bis zur ersten Leerzeile können aus beliebig vielen Parametern bestehen, die dem `GET`–Request noch mitgegeben werden. Der Browser teilt dem Server so mit, welche Datentypen er in der Antwort unterstützt und um welche Programmversion welchen Herstellers es sich bei dem Browser handelt. Prinzipiell ist der Request durch die Einführung neuer Zeilen beliebig erweiterbar.

Der Server beantwortet einen solchen Request wie in Abbildung 2.4 gezeigt auf eine ganz ähnliche Weise: Auch er erzeugt zunächst eine formalisierte Antwortzeile, in der er die verwendete Protokollversion, einen Statuscode und einen Freitext, der den Statuscode erläutert, zurückgibt. Danach folgt wieder ein relativ frei formatierte Headerblock, der Informationen über das gelieferte Dokument enthält, und dann, nach einer Leerzeile, das eigentliche Dokument.

```
HTTP/1.0 200 OK
Date: Mon, 04-Nov-1996 19:54:32 GMT
Server: Apache 1.0.3
Mime-Version: 1.0
Last-Modified: Mon, 04-Nov-1996 19:54:32 GMT
Content-Type: text/plain
<Leerzeile>
...
```

Abbildung 2.4: Ein Beispiel für eine HTTP–Response. Der Server beantwortet den Request zunächst mit einem Headerblock mit Metainformationen über die gewünschten Daten. Nach einer Leerzeile kommen dann die angeforderten Informationen.

Die `GET`–Methode

Die `GET`–Methode kann nicht nur verwendet, um Informationen von einem Server anzufordern, sondern sie kann zugleich auch Informationen zum Server übertragen, falls die angeforderte URL ein CGI–Programm bezeichnet.

Eine solche URL hat die Form

```
protokoll://server/path/file/extra_daten?query_string
```

Die Informationen aus `extra_daten` werden dem CGI–Programm in den Variablen `PATH_INFO` und `PATH_TRANSLATED` zur Verfügung gestellt, wie weiter oben beschrieben. Die Informationen aus dem `query_string` tauchen im CGI–Programm dagegen in der gleichnamigen Variablen auf. Sie liegen in einer codierten Form vor, die als *urlencoded* bezeichnet wird und die im Abschnitt 2.4 beschrieben wird.

Daten, die in `GET`-Methoden codiert werden, sind in der URL sichtbar. Solche Anfragen an CGI–Programme lassen sich also in Bookmarks oder Hotlists mit Parametern abspeichern. Das ist sehr nützlich, wenn man Benutzern vorgefertigte Anfragen an Suchmaschinen oder andere Roboter präsentieren möchte. Solche URLs können aber sehr lang und sehr häßlich werden, wenn viele Parameter zu übergeben sind. Für diese Zwecke ist die im folgenden beschriebene `POST`–Methode besser geeignet.

Die `POST`–Methode

Die `POST`–Methode sendet die Parameter für das CGI–Programm in einer gesonderten Übertragung an den Server. Die Daten werden für das Programm zur Standardeingabe. Da die Daten nicht Bestandteil der URL sind, besteht das Problem der zu langen Eingaben hier nicht. Auf der anderen Seite ist es nicht möglich, `POST`–Anfragen mit Parametern in Hotlists zu speichern. Die übertragenen Daten sind auch hier wieder *urlencoded*, obwohl es prinzipiell nicht unbedingt notwendig wäre.

2.4.2 URL–Encoding

Die Daten, die einem CGI–Programm übergeben werden, sind unabhängig von der verwendeten Transfermethode in jedem Fall urlencoded gesendet. Die Codierung wurde notwendig, weil man vor der Erfindung der `POST`–Methode mehrere Variablen mit Zeichen aller Art an den Server senden wollte und die Daten dazu in die mittels eines `GET` angeforderte URL eincodiert werden mußten. Eine solche URL darf bestimmte Zeichen nicht enthalten: Unter anderem kann eine URL keine Leerzeichen enthalten.

Die Trennung zwischen eigentlicher URL und zusätzlichen Daten für einen `QUERY_STRING` erfolgt wie oben schon angedeutet mit einem Fragezeichen (?). Die Daten im `QUERY_STRING` können aus mehreren Feldern der Form `name=wert` bestehen. Die einzelnen Felder werden durch Und–Zeichen (&) voneinander getrennt. Ein `QUERY_STRING` kann also so aussehen:

```
name1=wert1&name2=wert2&name3=wert3...
```

Um jetzt innerhalb eines Strings in urlencoding trotzdem Fragezeichen, Undzeichen und andere Sonderzeichen verwenden zu können, bedient man sich folgender Ersetzungsregeln:

- Leerzeichen werden durch Pluszeichen (+) dargestellt. Der String „Dies ist ein Test“ wird also zu „Dies+ist+ein+Test“.
- Andere Zeichen, die in einer URL nicht mehr vorkommen dürfen, werden durch das Zeichen Prozent (%) gefolgt vom zweistelligen hexadezimalen Zeichencode des Zeichens dargestellt. Der String „#$%“ wird also zu „%23%24%25“.

Jedes CGI–Programm muß in irgendeiner Form Funktionen zum Decodieren von Daten enthalten, die urlencoded vorliegen.

Kapitel 3

Formulare mit HTML

Natürlich ist es möglich, Eingaben für CGI–Programme zusammenzustellen, indem man von Hand einen GET–Request zusammenbaut. Schöner und bunter erzeugt man solche Requests jedoch mit HTML–Formularen, den FORMS.

3.1 Der <FORM>–Tag

Man kann Teile einer HTML–Seite zu Formularen erklären, indem man sie in die speziellen Tags <FORM> und </FORM> einschließt. Innerhalb dieses Bereiches dürfen dann zusätzliche Tags verwendet werden, um Bedienelemente und Eingabefelder zu definieren. Eine einzige HTML–Seite kann mehrere Forms enthalten, und diese verschiedenen Forms können unterschiedliche CGI–Programme starten, wenn dies notwendig sein sollte.

Der <FORM>–Tag muß in jedem Fall ein ACTION=–Attribut haben. Dieses Attribut gibt die URL eines CGI–Programmes an, das aktiviert werden soll, wenn das Formular abgeschickt wird. Optional kann der <FORM>–Tag noch mit einem Attribut METHOD= ausgezeichnet werden, das die Methode der Parameterübergabe spezifiziert. Mögliche Werte für diese Methode sind GET oder POST. Fehlt die Angabe einer Methode, wird GET angenommen. Abbildung 3.1 zeigt ein leeres Formular, das unser Beispielprogramm hello aufrufen soll.

```
<html>
<head><title>Testformular</title></head>
<body>
<h1>Testformular</h1>
<form action="http://server/cgi-bin/hello" method="get">
Hier Formulardaten einfügen.
</form>
</html>
```

Abbildung 3.1: Dieses leere Formular ist der Ausgangspunkt für die Gestaltung von HTML–Seiten mit Formularen.

3.2 Der <INPUT>–Tag

Innerhalb des Formularteils einer HTML–Seite kann normaler HTML–Code stehen. Außerdem ist eine Reihe von zusätzlichen Tags erlaubt, die die Bedienelemente des Formulars definieren. Der vielseitigste Tag

zur Definition dieser Bedienelemente ist <INPUT>. Mit ihm lassen sich außer Feldern zur Eingabe von Texten und Paßworten auch Check– und Radiobuttons erzeugen. Außerdem kann er versteckte Eingabefelder definieren und Dateiauswahlboxen zum Upload von Dateien auf den Server starten.

3.2.1 Text– und Paßwortfelder

Die Formen des <INPUT>–Tags zur Erzeugung von Text– und Paßwortfeldern sind identisch. Die Tags werden vom Browser auch identisch dargestellt. Der einzige Unterschied ist, daß ein Textfeld die Benutzereingaben anzeigt, während ein Paßwortfeld sie zwar aufnimmt, aber die Eingabe verbirgt. **Warnung:** Ein Paßwortfeld verbirgt die Daten zwar bei der Eingabe, aber der Transport über das Internet erfolgt offen und unverschlüsselt, wenn nicht besondere Vorkehrungen zur sicheren Kommunikation getroffen werden.

Die volle Form eines <INPUT>–Tags lautet:

```
<input type={feldtyp} name={variablenname} value={defaultwert}
       size={feldbreite} maxlength={eingabelänge}
```

Bis auf die Felder TYPE= und NAME= sind die Attribute des Tags optional. Sie bedeuten im Einzelnen:

TYPE= Der Typ des Feldes bestimmt das Aussehen des Kontrollelementes auf der HTML–Seite. Er kann für die hier diskutierten Eingabefelder entweder TEXT oder PASSWORD sein.

NAME= Der Name des Feldes bestimmt den Namen der Variablen, die später dem CGI–Programm übergeben wird. Der Name dieser Variablen ist nicht frei wählbar, sondern wird vom Programmierer des CGI–Programmes festgelegt. Es ist sehr wichtig, hier Groß- und Kleinschreibung des Namens genau zu beachten.

VALUE= Wenn das optionale Attribut VALUE= vorhanden und mit einem Wert belegt ist, wird dieser Wert als Defaultwert des Feldes vorgegeben. Fehlt das Attribut, ist das Eingabefeld zu Beginn leer.

SIZE= Dieses Attribut legt fest, wie viele Zeichen Breite das Eingabefeld am Bildschirm haben soll. Der Defaultwert ist 20 Zeichen.

Ältere CGI–Formulare haben die Form SIZE=x,y verwendet, um mehrzeilige Eingabefelder zu erzeugen. Das ist zwar immer noch prinzipiell möglich, aber der weiter unten diskutierte Tag <TEXTAREA> bietet sehr viel bessere Editiermöglichkeiten.

MAXLENGTH= Dieses Attribut legt fest, wie lang die Eingabe im Textfeld wirklich sein darf. Dies ist unabhängig von der mit SIZE= festgelegten sichtbaren Breite des Textfeldes.

Abbildung 3.2 zeigt das mögliche Aussehen eines <INPUT TYPE=''text'' >–Tags.

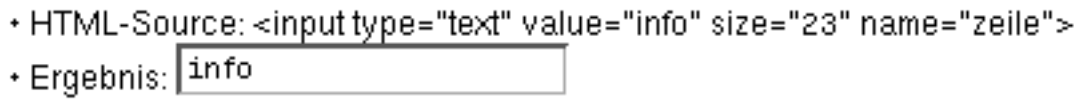


Abbildung 3.2: Ein Beispiel mit einem <INPUT TYPE=''text'' >-Tag.

3.2.2 Check– und Radioboxes

Wenn man dem Benutzer keine Eingabe in freier Form erlauben möchte, bietet es sich für kleine Mengen auszuwählender Optionen an, je nach Art der Auswahl entweder Checkboxen oder Radioboxen zu definieren.

Checkboxen sind kleine Kästchen, die der Benutzer ankreuzen kann und die eine Option entweder ein- oder ausschalten. Radioboxen bieten dagegen eine Eins–aus–n–Auswahl an: Unter einer Menge von Optionen ist nur genau eine auswählbar.

In beiden Fällen sieht der <INPUT>–Tag zur Erzeugung dieser Auswahl so aus:

```
<input type={typ} name={variablenname} value={wert} checked>
```

Außer dem Attribut CHECKED sind alle Attribute dieses Tags vorgeschrieben und dürfen nicht weggelassen werden. Sie bedeuten:

TYPE= Dieses Attribut ist entweder CHECKBOX oder RADIO und legt damit den Typ des Knopfes fest.

NAME= Dieses Attribut bestimmt wie immer den Namen der Variable, die dem CGI–Programm übergeben wird.

VALUE= Dieses Attribut ist im Gegensatz zu TEXT–Eingaben **nicht** optional. Ist die entsprechende Box selektiert, wird die Eingabevariable des CGI–Programmes mit dem hier angegebenen Wert belegt, sonst ist sie leer.

CHECKED Dieses optionale Attribut markiert den Defaultwert der Box. Ist es vorhanden, erscheint die Box angeklickt.

Abbildung 3.3 zeigt das mögliche Aussehen dieser Tags.

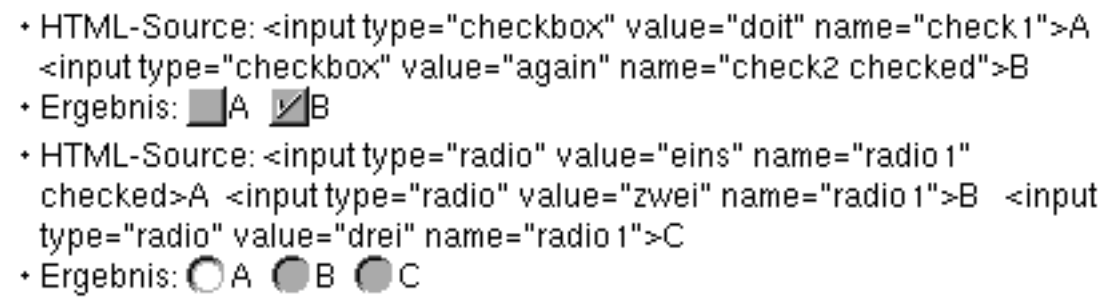


Abbildung 3.3: Ein Beispiel mit Radio– und Checkboxes.

Ein wichtiger Unterschied zwischen Checkboxen und Radioboxen existiert: Checkboxen müssen immer verschiedene NAME=–Attribute haben, denn jede Checkbox existiert für sich alleine und unabhängig von anderen Checkboxen.

Radioboxen treten dagegen in Gruppen auf und innerhalb einer Gruppe von Radioboxen kann immer nur eine selektiert sein. Eine Gruppe von Radioboxen wird durch gleiche NAME=–Attribute definiert.

3.2.3 Versteckte Eingabefelder

Mit einem Eingabefeld vom Typ HIDDEN lassen sich versteckte Eingabefelder anlegen:

```
<INPUT TYPE="HIDDEN" NAME={variablennamen} VALUE={wert}>
```

Solche Eingabefelder sind nicht sichtbar und auch nicht veränderbar. Sie werden in jedem Fall unverändert an das CGI–Programm übergeben. Versteckte Eingabefelder sind nützlich, wenn man ein CGI–Programm mit mehreren Funktionen hat, bei dem über ein Eingabefeld eine Funktion ausgewählt werden muß, oder wenn man einen Zustand in einer dynamisch erzeugten Seite über eine Transaktion hinweg retten muß.

3.2.4 Dateiupload

Es ist möglich, über ein Eingabefeld vom Typ `FILE` ganze Dateien auf den WWW-Server zu uploaden. Es ist dringend angeraten, für eine solche Transaktion ein Formular mit der Methode `POST` zu verwenden. Außerdem muß dem Formular mit dem Attribut `ENCTYPE=multipart/form-data` klargemacht werden, auf welche Weise der Upload zu geschehen hat. *Hinweis:* `FILE` ist eine Netscape–Erweiterung.

```
<FORM ACTION="URL" METHOD="POST" ENCTYPE="multipart/form-data">
Dateinamen angeben: <INPUT TYPE="FILE" NAME="upload">
</FORM>
```

3.2.5 Absenden und Rücksetzen

Mit den gezeigten Tags lassen sich zwar unfangreiche Formulare erzeugen, aber diese Formulare können noch nicht in Betrieb genommen werden, denn es fehlt ein Knopf zum Absenden des Formulars. HTML sieht für diesen Zweck Eingabefelder vom Typ `SUBMIT` vor und bietet außerdem noch Eingabefelder vom Typ `RESET` an, mit dem sich ein Formular auf den Startzustand zurücksetzen läßt.

```
<INPUT TYPE="SUBMIT" VALUE="Formular absenden">
<INPUT TYPE="RESET" VALUE="Eingabe löschen">
```

Jedes Formular muß einen `SUBMIT`–Button haben, und ein Formular kann nur einen `SUBMIT`–Button haben. Abbildung 3.4 zeigt mögliche Darstellungen dieser Knöpfe.

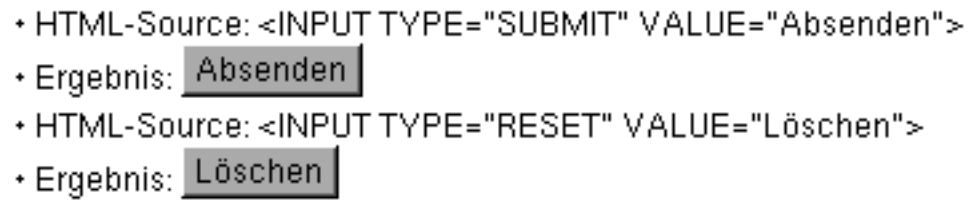


Abbildung 3.4: Ein Beispiel mit `SUBMIT` und `RESET`.

3.3 Der <TEXTAREA>–Tag

Felder zur freien Texteingabe lassen sich mit dem Tag `<TEXTAREA>` definieren. Der Tag hat diese Form:

```
<TEXTAREA NAME={variablenname} ROWS={zeilen} COLS={spalten}>
Textvorgabe
</TEXTAREA>
```

Die Attribute `ROWS=` und `COLS=` sind optional. Die Attribute bedeuten:

`NAME=` Das Attribut legt wie üblich den Namen der CGI–Variablen fest.

`ROWS=` Das Attribut bestimmt die Höhe des Eingabefeldes in Zeilen.

`COLS=` Das Attribut bestimmt die Breite des Eingabefeldes in Spalten.

Das Ergebnis dieses Tags sieht dann möglicherweise so aus wie in Abbildung 3.5.

Es ist angeraten, `<TEXTAREA>` zusammen mit der Methode `POST` zu verwenden.

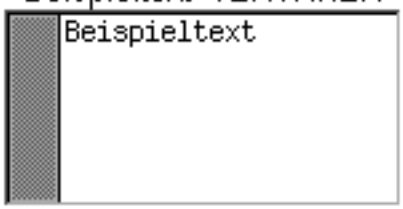
- HTML-Source: <TEXTAREA NAME="text" ROWS=5 COLS=20>Beispieltext<TEXTAREA>
- Ergebnis: 

Abbildung 3.5: Ein Beispiel mit TEXTAREA.

3.4 Der <SELECT>–Tag

Größere Auswahlen von vorgegebenen Werten erzeugt man wegen des großen Platzverbrauches und im Interesse der Übersichtlichkeit besser nicht mit Checkboxen oder Radiobuttons, sondern mit dem <SELECT>–Tag. Je nach Konfiguration erzeugt dieser Tag ein Popup-Menü oder eine Auswahlbox für Mehrfachauswahlen.

Das Format dieses Tags ist

```
<SELECT NAME={variablenname} SIZE={größe} MULTIPLE>
<OPTION SELECTED VALUE={wert}>Option 1
<OPTION ...>
</SELECT>
```

Der <SELECT>–Tag erfordert nur die Angabe des Variablennamens mit NAME=, die anderen Attribute sind optional. Ein <SELECT>–Container kann beliebig viele <OPTION>–Tags enthalten. Diese können optional SELECTED sein und es kann optional ein VALUE= festgelegt werden, mit dem die Variable belegt wird, wenn dieses Element ausgewählt ist.

Wenn das optionale Attribut MULTIPLE angegeben ist, ist die Mehrfachauswahl von Optionen möglich.

Kapitel 4

Perl

CGI–Programme können in jeder beliebigen Programmiersprache geschrieben werden, solange sie in der Lage ist, einen `QUERY_STRING` zu decodieren. Programmiersprachen, die eine ausgereifte Stringbehandlung und eine automatische Speicherverwaltung haben, sind besonders geeignet, denn CGI–Programme benutzen diese Spracheigenschaften meistens sehr intensiv. Ein Datenbankinterface oder andere Möglichkeiten, große Datenmengen zu durchsuchen, sind ein zusätzliches Plus, denn oft müssen CGI–Programme vorhandene Daten für die Darstellung im Web aufbereiten.

Die Programmiersprache Perl ist besonders geeignet zur Programmierung von CGI–Scripten, denn sie macht gerade die Bearbeitung von Texten und Dateien besonders einfach. Ursprünglich war Perl eine Sprache zur Reportgenerierung aus beliebig formatierten Systemlogbüchern. Daher stammen Perls umfangreiche Einbaufunktionen zum Suchen und Ersetzen von Texten und zur Formatierung von Reports. Die Weiterentwicklung hat Perl zu einem sehr mächtigen Werkzeug zur Systemprogrammierung und –steuerung in UNIX gemacht: Gerade die Systemverwalter größerer Installationen wissen Perl zu nutzen, um Wartungsabläufe auf ihren Systemen zu automatisieren. Auch die weite Verfügbarkeit von Perl auf praktisch allen Plattformen, die ausreichend Speicher zur Verfügung stellen, macht die Sprache für diese Zwecke interessant.

Mit dem Aufkommen des World Wide Web hat Perl eine Leistungsfähigkeit als Sprache zur schnellen Entwicklung von CGI–Programmen demonstriert. Ein sehr großer Teil aller CGI–Programme ist zunächst als Perl–Script entwickelt worden. Oft ist die Leistung diesr Scripte schon ausreichend, so daß eine Portierung in eine compilierte Sprache sich erübrigt.

4.1 Einfache Perl–Programme

4.1.1 Hello, World!

Das berüchtigte „Hello, World!“–Programm sieht in Perl so aus:

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

print "Hello, World\n";
```

Es sieht aus wie eine Kreuzung zwischen einem Shellscript und einem C–Programm. In der Tat ist Perl eine interpretierte Sprache, denn Perlprogramme müssen nicht compiliert werden, sondern können wie Shellscripte direkt aufgerufen werden. Genau wie bei UNIX–Shellscripten gibt die erste Zeile des Programms an, daß es sich um ein Script handelt und unter welchem Pfad der Interpreter für diese Scripte zu finden ist. Für das Script selbst ist eine Zeile, die mit einem # beginnt, ein Kommentar, so daß das Script selbst seinen eigenen Aufruf als Kommentar interpretiert.

Perlprogramme sind jedoch schneller als Shell–Scripte, denn der Interpreter übersetzt das Programm beim Aufruf in eine interne Form und bestimmt auch schon die Adressen der verschiedenen Funktionen und Variablen. In der Tat steht ein Perlprogramm einem C–Programm in der Geschwindigkeit nicht viel nach. Seine Ausführungsgeschwindigkeit ist der von P–Code vergleichbar, wie ihn einige Microsoft–Compiler zu Speicherersparniszwecken erzeugen.

Die Anweisung `print` druckt ihr Argument, einen String nach den Konventionen von C. Wie in C werden Anweisungen in Perl mit einem Semikolon beendet.

4.1.2 Benutzereingaben lesen

Die nächste Stufe der Komplexität nach „Hello, World!“ besteht in der Regel darin, den Namen der zu grüßenden Person einzulesen und diese Person namentlich zu begrüßen. Das erfordert die Verwendung von Operationen zur Eingabe von Daten und natürlich Variablen.

In Perl gibt es nur einen Typ von Variablen. Sie können entweder Strings, Integer oder Fließkommazahlen speichern und sie müssen nicht vereinbart werden. Perl–Variablen können jedoch in Form von einfachen Werten (*Skalaren*), numerisch indizierten Feldern (*Arrays*) oder beliebig indizierten Tabellen (*assoziativen Feldern, Hashtables*) auftreten. Skalare werden durch das Zeichen Dollar ($) gekennzeichnet, so daß `$name` eine skalare Variable mit dem Namen `name` bezeichnet. Diese Variable wird ins Leben gerufen, indem ihr einfach ein Wert zugewiesen wird.

Um nach einem Namen zu fragen, muß ein Prompt ausgegeben werden und auf irgendeine Weise eine Eingabe eingelesen werden. Den Prompt können wir bereits drucken, mit `print`. Eine Eingabe einzulesen, ist ebenfalls sehr einfach, denn wie in C stehen uns die vordefinierten Filehandles `<STDIN>`, `<STDOUT>` und `<STDERR>` zur Verfügung. Perl ist auf Stringverarbeitung optimiert: Um eine Zeile der Eingabe einzulesen, genügt es, einer Variablen das Filehandle zuzuweisen:

```
print "Wie ist Ihr Name? ";
$name = <STDIN>;
```

Das Ergebnis ist, daß der Skalar `$name` eine Eingabezeile inklusive des Zeilenendezeichens enthält. Wir können dieses letzte Zeichen abschneiden, indem wir die Funktion `chop()` verwenden, die das letzte Zeichen eines Strings zurückliefert und als Seiteneffekt den String um dieses Zeichen kürzt.

```
chop($name);
```

Wie in Shell auch können Variablen in Stringkonstanten verwendet werden: Sie werden vor der Ausgabe durch ihren Wert ersetzt:

```
print "Hello, $name!\n";
```

Und wie in Shell auch muß das Dollarzeichen in einem String durch einen Backslash geschützt werden, wenn es ausgegeben werden soll.

Hier noch einmal das ganze Programm:

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

print "Wie ist Ihr Name? ";
$name = <STDIN>;

chop($name);

print "Hello, $name!\n";
```

4.2 Variablen und Konstanten

4.2.1 Notation von Skalaren

Wie bereits erklärt, unterscheidet Perl nicht nach verschiedenen Datentypen, und Variablen müssen nicht vereinbart werden. Einer skalaren Variable kann so ziemlich jeder Wert zugewiesen werden, der sich in Perl darstellen läßt. Das können Zahlen wie 42 oder 3.1415926e00 sein oder Zeichenketten beliebiger Länge. Anders als Shell hat Perl keine Probleme mit binären Zeichen in Strings, ja nicht einmal Nullbytes machen Schwierigkeiten. Es ist kein Problem, in Perl Binärprogramme in Strings einzulesen, zu patchen und wieder zurückzuschreiben.

Perl wandelt skalare Werte intern automatisch in Zeichenketten oder in Fließkommazahlen um, je nachdem, welche Art von Operation mit dem Wert durchgeführt wird. Die Sprache kennt keine Ganzzahloperationen und arbeitet auch intern niemals mit ganzen Zahlen, sondern immer mit doppelt genauen Fließkommazahlen. Zahlen können in Perl wie in C notiert werden, Abbildung 4.1 gibt einen Überblick.

Format	Notation
Fließkomma	1.24
	7.24e17
	-7.3e17
	7.3e-17
Ganzzahl	17
Ganzzahl (okatal)	0377
Ganzzahl (hex)	-0xff

Abbildung 4.1: Zahlenformate in Perl.

Wie in der UNIX–Shell kann und muß man auch in Perl gelegentlich Zeichenketten in Anführungszeichen einschließen, etwa wenn sie Leerzeichen oder Sonderzeichen enthalten. In jedem Fall kann man einzelnen Sonderzeichen ihre Sonderbedeutung nehmen, indem man ihnen einen Backslash voranstellt. Das gilt natürlich auch für den Backslash (und die verschiedenen Sorten Anführungszeichen) selbst.

Einfache Anführungszeichen wirken so, als sei jedes einzelne Zeichen in der Zeichenkette von einem Backslash geschützt. Innerhalb der Zeichenkette darf jedes beliebige Zeichen auftauchen, auch Zeilenumbrüche, wenn sich die Zeichenkette über mehrere Zeilen erstreckt. Beispiele:

```
'Hello, World!'
'Don\'t do that, Dave!'
''   # Der Leerstring.
'Dies ist der Backslash: \\.'
'multiline
string'
```

Doppelte Anführungszeichen funktionieren eher so wie Strings in Shell und C: Innerhalb solcher Zeichenketten werden Variablen ersetzt, und einige Sonderzeichen können durch Backslash-Sequenzen notiert werden (Abbildung 4.2).

```
"Hello, World!"
"Aufwachen!\007"
"Spalte1\tSpalte2"
```

Zeichen	Bedeutung
\n	Newline.
\r	Return.
\t	Tab.
\f	Formfeed.
\b	Backspace.
\v	Vertical Tab.
\a	Bell.
\e	Escape.
\007	Zeichen mit dem oktalen Zeichencode xxx, hier: 7, Bell.
\x7f	Zeichen mit dem hexadezimalen Zeichencode xxx, hier: 127, Delete.
\cC	Controlzeichen, hier: Control-C.
\\	Backslash
\''	Doppeltes Anführungszeichen
\l	Nächstes Zeichen in Kleinschrift wandeln.
\L	Alle folgenden Zeichen bis zum \E in Kleinschrift wandeln.
\u	Nächstes Zeichen in Großschrift wandeln.
\U	Alle folgenden Zeichen bis zum \E in Großschrift wandeln.
\E	Beende \L oder \U.

Abbildung 4.2: Controlzeichen in Perl.

4.2.2 Operationen mit Skalaren

Perl kennt dieselben numerischen Operationen auf Skalaren, die C auch kennt: `+, -, *, /` und `%`. Außerdem ist noch ein Potenzoperation `**` definiert. Numerische Vergleiche werden mit den Operatoren `<, <=, ==, >=, >` und `!=` vorgenommen.

Außerdem ist auf Strings die Konkatenation definier; dazu wird der Operator Punkt (`.`) verwendet. Strings können mit dem Operator `x` auch vervielfacht werden. Strings werden alphabetisch verglichen, dafür stehen die Operatoren `lt, le, eq, ge, gt` und `ne` zur Verfügung. Die numerischen Operatoren würden `7 < 30` als wahr bewerten, aber `7 lt 30` ist falsch.

Skalare können einfach zugewiesen werden, Perl verhält sich dabei im wesentlichen wie C. Der einzige Unterschied ist, daß Variablen nicht deklariert werden müssen, sondern nach Bedarf ins Leben gerufen werden. Wie in C existiert zu jedem arithmetischen Operator auch ein zugehöriger rechnender Zuweisungsoperator, d.h. man kann `+=, -=, *=, /=` und so weiter verwenden. Ebenso wie in C sind `++` und `--` als Pre- und Postoperatoren erlaubt.

Wird ein Skalar benutzt, bevor er einen Wert zugewiesen bekommen hat, liefert Perl den speziellen Wert *undef*. Dieser Wert sieht aus wie der Leerstring oder wie eine numerische Null, je nachdem, in welchem Kontext er verwendet wird. *undef* kann jedoch mit dem Prädikat `defined()` abgefragt werden. Das ist nützlich im Zusammenhang mit Dateien, die am Dateiende *undef* zurückliefern.

4.2.3 Arrays

Außer skalaren Werten kann Perl noch Felder und assoziative Felder bearbeiten. Genau wie bei Strings gibt es auch bei diesen Datentypen keine Größenbegrenzungen, und genau wie gewöhnliche Variablen müssen auch die beiden Arten von Feldern in Perl nicht deklariert werden, sondern können einfach benutzt werden.

So wie skalare Variable durch ein `$` bezeichnet werden, kennzeichnet Perl Felder mit einem `@` und Hashlisten mit einem `%`–Zeichen. Demnach bezeichnet `$name` einen Skalar mit dem Namen `name` und `@name` ein Feld desselben Namens. Perl unterhält unterschiedliche Namensräume für alle drei Sorten von Objekten, so daß `$name`, `@name` und `%name` unterschiedliche Objekte bezeichnen.

Ebenso wie skalare Konstanten in Perl notiert werden können, existieren auch Notationen für Felder:

```
(1,2,3)   # Ein Feld mit dem drei Werten 1, 2 und 3.
("huhu", "du", "da") # Ein Feld mit drei Strings.
```

Es ist auch möglich, gemischte Felder zu notieren, und die Feldelemente brauchen keine Konstanten zu sein:

```
($name, 42)  # Der Wert von $name und 42.
($vorname + $nachname, $strasse + $hausnr) # Zwei Werte.
() # Das leere (nullelementige) Feld.
```

Um große Listen von aufzählenden Zahlenwerten zu notieren, existiert in Perl außerdem noch eine Abkürzung:

```
(1..10)      # Die ganzen Zahlen von 1 bis 10.
(11.9..15.9) # Die Zahlen 11.9, 12.9, .., 15.9.
(2, 3, 5..9) # Die Zahlen 2, 3, 5, 6, 7, 8 und 9.
($start..$stop) # Der Bereich von $start bis $stop.
```

Solche Listen zählen immer nur aufsteigend. Wenn der rechte Wert kleiner ist als der linke Wert, entsteht die leere Liste. Der Operator `print` kann ebenfalls Listen drucken; die Ausgabe erfolgt ohne Leerzeichen zwischen den Listenelementen:

```
print ("Der Name ist ", $name,"\n";
```

4.2.4 Operationen mit Arrays

Arrays werden in Perl genauso zugewiesen wie Skalare, nämlich mit `=`. Perl erkennt, ob es sich um eine Zuweisung von Arrays oder Skalaren handelt, indem es die linke Seite der Zuweisung analysiert und feststellt, ob es sich dabei um einen skalaren oder um einen Arraykontext handelt.

```
@lall  = (1..3); # Das Feld @lall besteht jetzt aus 3 Elementen.
@bla   = @lall;  # In Perl können ganze Felder zugewiesen werden.
@fasel = 1;      # 1 wird zu (1) und dann zugewiesen.
```

In der Elementliste auf der rechten Seite der Zuweisung können auch wieder Arrays als Elemente auftauchen. Sie werden bei der Zuweisung durch ihre Elemente ersetzt. Auf diese Weise ist es möglich, vorne und hinten in das Feld Elemente einzufügen.

```
@lall = (1..3);
@bla  = (0, @lall, 4); # wird zu (0..4).
```

Enthält eine Liste nur Variablen, kann sie auch als linke Seite einer Zuweisung verwendet werden. Auf diese Weise kann man bequem beliebig viele Elemente eines Feldes permutieren. Wenn die linke Liste einer Zuweisung weniger Elemente hat als das Feld auf der rechten Seite, werden die überzähligen Elemente verworfen. Es ist aber auch möglich, als letztes Element der linken Seite ein Feld zu verwenden, das diese Elemente dann aufnimmt:

```
($b, $a) = ($a, $b); # $a und $b vertauschen.
($a, $b) = ($c, $d, $e); # $b ist $d, $e ist verloren.
($a, @b) = ($c, $d, $e); # $b[0] ist $d, $b[1] ist $e.
```

Wird ein Feld in einem skalaren Kontext verwendet, wird die Länge des Feldes für das Feld eingesetzt. Die Länge eines Feldes `@name` kann aber auch mit der Variablen `$#name` abgefragt werden.

Auf die Elemente eines Feldes `@name` kann durch Indizierung wie in C zugegriffen werden. Die Elemente eines Feldes sind Skalare, daher sind sie mit dem Prefix `$` zu versehen. Es ist möglich, beliebigen einzelnen Feldelementen Werte zuzuweisen. Perl vergrößert das Feld nach Bedarf, eventuell vorhandene Lücken werden mit *undef* aufgefüllt.

```
print $name[0]; # das erste Element des Feldes @name.
$name[0] = 17;
```

Anders als in C kann in Perl auch auf eine Liste von Arrayelementen zugegriffen werden. Dies wird als eine `array slice` bezeichnet. In diesem Fall handelt es sich um einen Arraykontext, daher also das Prefix `@`.

```
@fasel[0, 1]; # entspricht ($fasel[0], $fasel[1])
@fasel[1, 0] = @fasel[0, 1]; # Feldelemente tauschen.
@fasel[0, 1, 2] = @fasel[1, 1, 1];
@fasel[9, 10] = (17, 4);
```

Mit den Operatoren `push` und `pop` kann Perl ein Array wie einen Stack verwenden. Es ist möglich, mehrere Werte (oder ganze Felder) auf einmal zu pushen.

```
push(@einStack, $einWert);
push(@einStack, 4, 5, 6);
$einWert = pop(@einStack);
```

Während `push` und `pop` die rechte Seite eines Arrays bearbeiten (d.h. an den hohen Indizes manipulieren), verändern `shift` und `unshift` die linke Seite des Feldes. `shift` entspricht dabei dem gleichnamigen Shell–Operator, und `unshift` ist seine Umkehrung.

`reverse` liefert die Elemente eines Feldes in umgekehrter Reihenfolge, `sort` liefert sie alphabethisch sortiert[1], und `chop` ist auch auf alle Elemente eines Feldes anwendbar.

Dateihandles wie `<STDIN>` können auch Arrays zugewiesen werden. Sie lesen dann die ganze Datei auf einmal ein und weisen jede Zeile einem Feldelement zu.

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

@lines = <STDIN>;
@reverselines = reverse(@lines);
@sortlines = sort(@lines);

print @sortlines;
print @reverselines;
```

4.3 Kontrollstrukturen

In Perl werden wie in C Blöcke von Anweisungen gebildet, indem man die Anweisungen in geschweifte Klammern einschließt. Diese Anweisungen werden linear ausgeführt.

```
{
    anweisung_1;
```

[1] Tatsächlich kann man die Sortierreihenfolge beliebig beeinflussen.

```
    anweisung_2;
    ...
    anweisung_n;
}
```

Wie in C ist es möglich, Bedingungen zu formulieren. Anders als in C **müssen** die geschweiften Klammern immer vorhanden sein, auch dann, wenn nur eine Anweisung von der Bedingung abhängig ist. Der `else`–Block kann weggelassen werden. Mit Hilfe von `elsif` können Entscheidungen kaskadiert werden, ohne daß sich die Einrücktiefe bis ins Unendliche steigert.

```
if (bedingung) {
    anweisungen;
} elsif (neue_bedingung) {
    anweisungen;
else {
    anweisungen;
}
```

In Perl werden Bedingungen als Strings berechnet. Bedingungen sind genau dann falsch, wenn sie entweder genau den String `0` oder den leeren String ergeben. Alle anderen Werte (1, 17, 0.000, 00) sind wahr.

Eine etwas seltsame Konstruktion in Perl ist die `unless`–Anweisung, die im Prinzip ein `if` ohne `then` darstellt (Die Anweisung `die` beendet ein Perl–Programm mit einer Fehlermeldung).

```
print "Wie ist Dein Name? ";
$name = <STDIN>;
unless ($name eq "Kristian") {
    die "Zugriff verweigert.\n";
}
```

Eine andere seltsame Konstruktion ist das einzelnen Anweisungen nachgestellte `if` oder `unless`.

```
die "Zugriff verweigert.\n" unless ($name eq "Kristian);
```

Perl ist in der Lage, einen Block von Anweisungen zu wiederholen, solange eine Bedingung wahr (`while`) oder falsch (`until`) ist. Ebenso ist eine zu C analoge `for`–Anweisung zum Zählen vorhanden. Die Syntax ist in allen Fällen wie in C, nur daß die geschweiften Klammern nicht entfallen dürfen.

```
while (bedingung) {
    anweisungen;
}

until (bedingung) {
    anweisungen;
}

for (start_anweisungen; test_anweisungen; zählanweisungen) {
    arbeits_anweisungen;
}
```

Zusätzlich ist Perl in der Lage, einen Block für jedes Element eines Feldes auszuführen. Das ist bequemer, als erst die Länge eines Feldes zu bestimmen und es dann durchzuzählen.

```
foreach $zähler (@eineListe) {
    anweisungen;
}
```

Schleifen können mit `last` und `next` vorwärts und rückwärts kurzgeschlossen werden, d.h. `last` entspricht dem `break` in C und `next` entspricht dem `continue` von C.

Blöcke können in Perl mit einem Label markiert werden. `last` und `next` können mit Hilfe dieser Labels mehrere Ebenen auf einmal verlassen oder kurzschließen.

4.4 Assoziative Arrays

Assoziative Arrays oder *Hashlisten* sind programmtechnisch eine Verallgemeinerung von Arrays. Statt ganzzahliger Ausdrücke kann in einer Hashliste ein beliebiger String als Index verwendet werden. Über diesen String kann jederzeit auf dem mit dem String assoziierten Wert zugegriffen werden. Man kann sich eine solche Hashliste auch als Tabelle mit zwei Spalten vorstellen: Jedem String aus der linken Spalte (den *Schlüsseln*) ist der zugehörige String auf der rechten Seite (ein *Wert*) zugeordnet. Anders als in einer richtigen Tabelle oder in einem Array haben die Elemente in einer Hashliste aber keine vorgegebene Ordnung. Würde man sie aufzählen (z.B. in einer `foreach`–Schleife), bekäme man irgendeine Reihenfolge.

In Perl werden assoziative Felder durch ein Prozentzeichen (%) markiert. In einem Arraykontext werden sie wie ein normales Array mit einer geraden Anzahl von Feldern behandelt. Die geraden Positionen dieses Arrays beginnend bei 0 sind mit Schlüsseln belegt, die ungeraden Positionen mit den zu diesen Schlüsseln gehörenden Werten. Die Funktion `keys()` liefert die Schlüssel eines solchen Feldes und die Funktion `values()` die Werte.

```
%vorname_von = ( "Köhntopp", "Kristian",
    "Seeger", "Martin",
    "Rump", "Birgit");
# Reihenfolge ist zufällig!
print keys(%vorname_von);   # Köhntopp, Seeger, Rump
print values(%vorname_von); # Kristian, Martin, Birgit
```

Der Zugriff auf einzelne Elemente erfolgt wie bei Feldern durch Indizierung, aber der Indexoperator besteht hier aus den geschweiften Klammern.

```
# Reihenfolge ist zufällig
foreach $i (keys(%vorname_von)) {
    print "Der Vorname von $i ist $vorname_von{$i}.\n";
}

foreach $i (sort keys %vorname_von) {
    print "Der Vorname von $i ist $vorname_von{$i}.\n";
}
```

Einzelne Elemente lassen sich mit dem Operator `delete` aus dem Feld löschen (man kann auch das ganze Feld löschen).

4.5 Eingabe und Ausgabe

4.5.1 Standardeingabe

Wie bereits gezeigt, kann man eine Zeile der Eingabe einlesen, indem man das Filehandle der Eingabe an eine skalare Variable zuweist. Wenn keine Eingabezeile mehr verfügbar ist, wird der Wert *undef* zugewiesen. Weist man ein Filehandle statt dessen an ein Array zu, wird die gesamte Datei bis zum Dateiende zugewiesen und jedem Arrayelement eine Zeile zugeordnet.

Oft möchte man eine Datei zeilenweise durchlesen und für jede Zeile einer Operation ausführen. Für die Standardeingabe sieht das so aus:

```
while ($i = <STDIN>) {
    print $i; # Zeile drucken
}
```

Die Schleife beendet sich am Dateiende, da der Wert *undef* zum Leerstring expandiert und dieser String als false–Wert gilt. Leere Eingabezeilen bestehen aber immerhin noch aus einem Newline–Zeichen, sind also keine Leerstrings.

In Perl ist es so, daß für viele Operatoren die Variable `$_` als Defaultvariable angenommen wird, wenn keine Variable angegeben wird. Tatsächlich kann man das Beispiel auch so schreiben:

```
while (<STDIN>) {
    print; # Zeile drucken
}
```

In diesem Fall wird statt der Variablen `$i` die Standardvariable `$_` verwendet.

4.5.2 @ARGV und %ENV

Die Kommandozeilenparameter eines Perlprogrammes stehen in der vordefinierten Arrayvariablen `@ARGV` zur Verfügung. Das spezielle Dateihandle `<>` interpretiert diese Namen der Reihe nach als Dateinamen und liest alle diese Dateien durch. Das Programm

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

while (<>) {
    print; # Zeile drucken
}
```

entspricht also dem UNIX–Kommando `cat`. Es kann `perlcat a b c` aufgerufen werden und gibt nacheinander die Inhalte aller drei Dateien als eine Datei auf der Standardausgabe aus.

Im assoziativen Array `%ENV` stellt Perl alle Umgebungsvariablen zur Verfühgung. Die Schlüssel sind jeweils der Name, die Werte sind die Werte der Umgebungsvariablen.

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

# Finde den Pfad des Homeverzeichnisses
# des Benutzers.
print $ENV{'HOME'};
```

4.5.3 printf

Wem die Ausgabe von Variablen mit `print` zu langweilig oder zu umständlich ist, der kann statt dessen `printf` verwendet. Diese Anweisung akzeptiert einen Formatstring wie die gleichnamige C–Funktion und eine Liste von Variablen, die entsprechend dem Formatstring formatiert werden.

```
printf "%10s %05d %7.2f\n", $s, $i, $f;
```

4.6 Reguläre Ausdrücke

Viele Perl–Anweisungen und Funktionen machen von regulären Ausdrücken Gebrauch. Ein regulärer Ausdruck ist ein Suchmuster, das gegen eine Zeichenkette verglichen wird und entweder paßt oder nicht. Oft kommt es nur darauf an, ob der Ausdruck paßt oder nicht, manchmal möchte man auch noch wissen, welcher Teil der Zeichenkette genau gepaßt hat, etwa um ihn zu ersetzen.

Reguläre Ausdrücke werden von vielen UNIX–Programmen verwendet, darunter `grep`, `sed`, `awk`, `ed`, `vi`, `emacs` und die verschiedenen Shells. Jedes dieser Programme hat eine leicht unterschiedliche Art der Notation für reguläre Ausdrücke, und auch die Leistungsfähigkeit unterscheidet sich gelegentlich etwas. Perls reguläre Ausdrücke sind eine Obermenge aller dieser Verfahren: Jeder reguläre Ausdruck, der in einem der genannten Werkzeuge formuliert werden kann, kann auch in Perl formuliert werden, muß aber evtl. leicht umformuliert werden.

In Perl werden reguläre Ausdrücke notiert, indem sie in Schrägstriche (*slashes*) eingeschlossen werden. Die einfachste Form von regulären Ausdrücken sieht so aus:

```
if (/abc/) {
    print "$_";
}
```

Im Beispiel wird die Einbauvariable `$_` gegen den regulären Ausdruck getestet und wenn eine Übereinstimmung erzielt wird, wird der String gedruckt. Die volle Syntax des *match*–Operators lautet

```
$variable =~ m/regexp/optionen;
```

Dieser Ausdruck vergleicht den Inhalt der Variablen `$variable` gegen den regulären Ausdruck `regexp`. Dabei kann der Ablauf des Matchvorgangs mit einigen einbuchstabigen Operationen beeinflußt werden (die wichtigste Option ist `i`, ignore case). Auch die Trennzeichen (Slashes) können durch andere, frei gewählte Trennzeichen ersetzt werden. Der Match selbst ergibt einen Wahrheitswert, „1“ oder den Leerstring.

Perl kennt die folgenden Regeln zur Konstruktion von regulären Ausdrücken:

- Alle Zeichen, denen im folgenden keine besondere Bedeutung zugewiesen wird, stehen in einem regulären Ausdruck für sich selbst. Der reguläre Ausdruck `a` paßt also auf einen String, der `a` enthält.
- Das Zeichen Punkt steht für ein einzelnes, beliebiges Zeichen außer Newline. Der reguläre Ausdruck `.` paßt also auf alle Strings, die mindestens ein beliebiges Zeichen außer Newline enthalten.
- Eine in eckige Klammern eingeschlossene Zeichenmenge steht für ein einzelnes Zeichen aus dieser Menge. Beginnt die Menge mit einem Dach, steht die Menge für ein einzelnes Zeichen, das nicht in der Menge ist. Der Ausdruck `[abc]` paßt also entweder auf das Zeichen `a` oder das Zeichen `b` oder das Zeichen `c`.
- Perl kennt einige vordefinierte Zeichenklassen (und deren Negationen), die oft benötigt werden:

Kürzel	Bedeutung	Kürzel	Bedeutung
\d (digits)	[0–9]	\D (digits, not!)	[^0-9]
\w (words)	[a–zA–Z0–9_]	\W (words, not!)	[^a–zA–Z0–9_]
\s (space)	[\r\t\n\f]	\S (space, not!)	[^\r\t\n\f]

- Mehrere Teilausdrücke können einfach aneinandergereiht werden. Der resultierende Ausdruck paßt dann auf eine Folge von Zeichen, auf die jeweils die Teilausdrücke passen würden. Im Klartext bedeutet das, daß etwa `abc` auf ein `a` gefolgt von einem `b` gefolgt von einem `c` paßt und daß etwa `a.c` auf ein `a` gefolgt von einem beliebigen Zeichen außer Newline gefolgt von `c` paßt.
- Der Stern `*` paßt auf null oder mehr Wiederholungen des folgenden Zeichens. Der Ausdruck `ab*c` paßt also auf `ac`, `abc`, `abbc` und so weiter.

- Das Pluszeichen `+` paßt auf eine oder mehr Wiederholungen des folgenden Zeichens. Der Ausdruck `ab+c` paßt also auf `abc`, `abbc` und so weiter.

- Das Fragezeichen `?` paßt auf eine oder eine Wiederholung des folgenden Zeichens. Der Ausdruck `ab?c` paßt also auf `ac` oder `abc`.

- Hinter einem Zeichen kann in geschweiften Klammer ein genereller Multiplikator `{n,m}` angegeben werden. Der Ausdruck paßt dann auf `n` bis `m` Wiederholungen dieses Zeichens. Wird nur eine Zahl angegeben, paßt der Ausdruck auf genau `n` Wiederholungen des Zeichens. Wird die zweite Grenze `m` weggelassen, wird für sie unendlich eingesetzt.

  Der Stern ist also ein Kürzel für `{0,}`, das Pluszeichen steht für `{1,}` und das Fragezeichen für `{0,1}`.

- Mit runden Klammern `()` können Teile eines Ausdruckes markiert werden. Diese Markierungen stehen im selben Ausdruck unter den Namen `\1`, `\2` und so weiter zur Verfügung. Der reguläre Ausdruck `a(.+)b\1c` paßt also auf einen String, der ein `a` gefolgt von einer Anzahl beliebiger Zeichen, gefolgt von einem `b` gefolgt von exakt demselben String wie zwischen `a` und `b` gefolgt von einem `c` enthält. Passend wäre also `addbddc` oder `ahallobhalloc`, aber nicht `addbhalloc`. Letzterer String würde aber auf den Ausdruck `a.+b.+c` passen.

  Nach einem Match (und bis zum nächsten Gebrauch des Match–Operators) stehen die markierten Teilstrings in den besonderen Perl–Variablen `$1`, `$2` und so weiter zur Verfügung.

  Außerdem binden die Klammern Teilausdrücke zusammen: `abc*` paßt auf `ab`, `abc`, `abcc` und so weiter, während `(abc)*` auf den leeren String, `abc`, `abcabc` und so weiter paßt.

- Und schließlich lassen sich mit dem Pipelinezeichen noch Alternativen deklarieren. Der Ausdruck `lall|laber` paßt entweder auf den String `lall` oder auf den String `laber`.

- Ein Dach markiert den Anfang eines Strings und ein Dollarzeichen das Ende eines Strings. Der Ausdruck `^x$` paßt also auf einen String, der genau nur aus dem Zeichen `x` besteht.

Außer dem m–Operator zum Matchen kennt Perl auch noch einen s–Operator zum Suchen und Ersetzen in einem String:

```
$variable =~ s/alt/neu/optionen;
```

Diese Operation ersetzt in der Variablen `$variable` alle Vorkommen des regulären Ausdrucks `alt` durch `neu`. Der Ausdruck `neu` darf dabei ebenfalls Klammermarkierungen enthalten. So würde `s/a(.+)b/x\1y/` in einem String ein Vorkommen von paarigen `a` und `b` durch passende `x` und `y` ersetzen, ohne daß der Text dazwischen irgendwie beeinflußt wäre.

Außer der Option `i` (ignore case) kennt der s–Operator auch noch die Option `g` (global search and replace), die nicht nur ein einzelnes Vorkommen von `alt` ersetzt, sondern alle.

4.6.1 `split()` und `join()`

Die Perlfunktion `split()` kann dazu verwendet werden, einen Skalar mit Hilfe eines regulären Ausdrucks in ein Feld zu zerlegen. `split()` nimmt dazu den String und wendet den regulären Ausdruck wiederholt auf den String an. Alle Teile des Strings, die auf den regulären Ausdruck passen, werden zu Trennzeichen, und die Teile des Strings, die nicht auf den Ausdruck passen, werden zu Feldelementen.

Eine Zeile der Paßwortdatei in UNIX sieht zum Beispiel so aus:

```
$line = "kris:x:100:20:Kristian Köhntopp:/home/kris:/bin/bash";
```

Mit dem Aufruf `split(/:/, $line);` wendet Perl den regulären Ausdruck `/:/` auf diese Zeile an. Der Ausdruck paßt auf alle Doppelpunkte, diese werden also Trennzeichen. Die Worte `kris`, `x`, `100`, `20`, `Kristian Köhntopp`, `/home/kris` und `/bin/bash` werden die Feldelemente 0 bis 6 des Ergebnisfeldes von `split`.

Wendet man `split()` in einem Arraykontext an, bekommt man das gesamte Feld als Ergebnis (und kann seine Länge mit `$#feld` feststellen). In einem skalaren Kontext bekommt man von `split()` die Länge des Feldes als Ergebnis.

```
$laenge = split(/:/, $line); # ergibt 6
@feld   = split(/:/, $line); # ergibt $feld[0] = "kris" usw.
                             # $#feld ist auch 6.
```

Andererseits kann man mit `join()` ein Feld mit beliebigen Trennzeichen zu einem Skalar zusammenfügen.

```
$line = join(":", @feld);
```

würde das zerlegte Paßwortfeld von oben wieder zusammensetzen.

4.7 Funktionen

Es ist möglich, in Perl eigene Funktionen zu definieren. Die Syntax ist folgendermaßen:

```
sub name {
    anweisungen;

    $wert;
}
```

Diese Anweisungen definieren eine Funktion mit dem Namen `name`, die einen Rückgabewert in `$wert` berechnet. Die Namen von Funktionen sind von den Namen von Skalaren, Arrays und Hashlisten unabhängig. Funktionen werden mit einem Undzeichen aufgerufen. Die Funktion `name` kann also so aufgerufen werden:

```
&name;        # name aufrufen und das Ergebnis verwerfen.
$var = &name; # name aufrufen und das Ergebnis nutzen.
```

Funktionen können mit Argumenten aufgerufen werden. Diese Argumente stehen im Feld `@_` zur Verfügung. Sie können also mit `$_[0]` und so weiter angesprochen werden.

```
sub add {
    $summe = 0;
    foreach $i (@_) {
        $summe += $i;
    }

    $summe;
}

print &add(1, 42, 17, 4);
print &add(1..5);
```

Variablen in Funktionen sind global. Die Variable `$summe` in der Funktion oben würde also eine gleichnamige Variable anderswo im Programm überschreiben. Mit Hilfe des Operators `local()` ist es möglich, lokale Variablen zu deklarieren:

```
sub add {
    local($summe) = 0;

    foreach $i (@_) {
        $summe += $i;
    }

    $summe;
}
```

Auf diese Weise kann man auch die Funktionsparameter in ein bekanntes Feld umkopieren:

```
sub add {
    local(@werte) = @_;
    local($summe) = 0;

    foreach $i (@werte) {
        $summe += $i;
    }

    $summe;
}
```

Kapitel 5

CGI–Programme in Perl

5.1 `cgi-lib.pl` und `testcgi.pl`

Zur CGI–Programmierung stehen in Perl eine ganze Reihe von Funktionspaketen zur Verfügung. Ein sehr einfaches Funktionspaket ist die `cgi-lib.pl` von Steven E. Brenner (`S.E.Brenner@bioc.cam.ac.uk`). Dieses Paket kann die Eingabedaten von GET und POST–Methoden einlesen und decodieren. Es stellt die Werte der einzelnen Eingabevariablen dann in dem assoziativen Feld `%in` zur Verfügung. Neben dieser Grundfunktionalität sind zusätzlich noch einige Hilfsfunktionen zum Debugging vorhanden.

Der Gebrauch des Paketes ist wie folgt: Im Bibliotheksverzeichnis von Perl muß die Datei `cgi-lib.pl` installiert sein. Das eigene Perlscript muß dann so anfangen:

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

# Paket aktivieren.
require "cgi-lib.pl" || die "cgi-lib.pl nicht gefunden.";

# Einlesen und Zerlegen der GET/POST-Variablen.
&ReadParse;

# html-Header erzeugen.
print &PrintHeader;

# Debugausgabe
print "<html>
<head>
<title>Test CGI</title>
</head>

<body>
<h1>Test CGI</h1>
<h2>Eingabevariablen:</h2>\n";

print &PrintVariables(%in);

print "\n<h2>Umgebungsvariablen</h2>\n";

print &PrintVariables(%ENV);

print "<hr>
```

```
Erzeugt von test-cgi.pl
</body>
</html>\n";
```

Das Script muß ausführbar gemacht werden und dann unter einer passenden URL aufgerufen werden (zum Beispiel: `http://white/cgi-bin/testcgi.pl`). Ihm können von einem Formular oder manuell beliebige Parameter mitgegeben werden. Seine Ausgabe sieht dann zum Beispiel so aus (mit einigen Parametern aufgerufen):

```
Test CGI

Eingabevariablen:

lall
    laber
probe
    hallo

Umgebungsvariablen

DOCUMENT_ROOT
    /home/www
GATEWAY_INTERFACE
    CGI/1.1
... gekürzt ...
SERVER_SOFTWARE
    Apache/1.0.3

------------------------------------------------------------
Erzeugt von test-cgi.pl
```

Das Script funktioniert wie folgt:

- Die Anweisung `require` aktiviert das Perl Funktionspaket `cgi-lib.pl`. Schlägt das fehl, beendet die `die`–Anweisung das Script mit einer Fehlermeldung.
- Der Aufruf der im Paket definierten Unterfunktion `&ReadParse` liest die Eingaben der GET– und POST–Methoden ein und decodiert sie. Die einzelnen Variablenwerte werden im assoziativen Feld `%in` hinterlegt.
- Die Funktion `&PrintHeader` liefert den benötigten Content–Type–String zurück, der dann mit `print` gedruckt wird.
- Mit einem einfachen, mehrzeiligen `print` wird der Header eines HTML–Dokumentes erzeugt.
- Die Funktion `&PrintVariables` druckt das ihr übergebene assoziative Array `%in` zu Debugzwecken aus.
- Nach einem Zwischenheader wird dann auch noch das assoziative Array `%ENV` ausgedruckt. Dieses enthält alle Umgebungsvariablen des Programms.
- Ein Abspann–`print` druckt den Seitenfuß.

5.2 Ein einfacher Zähler

CGI–Scripte können und sollen nicht gecached werden. Es ist daher möglich, auf diese Weise einen einfachen Seitenzähler zu realisieren. Das Script muß dazu an einem geeigneten Ort eine Datei hinterlegen, in der es seine Aufrufe mitzählt. Die Ausgabe des Scriptes kann eine beliebige HTML–Seite sein, in der der Zählerstand eingebaut wird.

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

# Paket aktivieren.
require "cgi-lib.pl" || die "cgi-lib.pl nicht gefunden.";

# Einlesen und Zerlegen der GET/POST-Variablen.
&ReadParse;

# html-Header erzeugen.
print &PrintHeader;

# Zählerstand einlesen.
open(COUNTER, "</tmp/counter");
$counter = <COUNTER> + 1;
close(COUNTER);

# Zählerstand merken.
open(COUNTER, ">/tmp/counter");
print COUNTER "$counter\n";
close(COUNTER);

# Seite ausgeben.
print "<html>
<head>
<title>Test CGI</title>
</head>

<body>
<h1>Z&auml;hlerseite</h1>

Diese Seite wurde $counter mal aufgerufen.

</body>
</html>\n";
```

Das Script funktioniert so:

- Wie üblich werden die Eingabeparameter eingelesen. Sie werden in dieser Version des Zählscriptes aber noch nicht verwendet.
- Es wird ein Dateihandle `<COUNTER>` zum Lesen geöffnet und eine Zeile mit einer Zahl eingelesen. Diese Zahl wird um eins erhöht. Die Datei wird wieder geschlossen.
- Dasselbe Dateihandle wird noch einmal geöffnet, aber zum Schreiben. Der aktuelle Zählerstand wird in die Datei hineingedruckt. Beachte, daß beim Drucken in Filehandles zwischen dem Filehandle und dem eigentlichen Drucktext **kein** Komma gesetzt wird!

- Der Seitentext wird ausgegeben.

Aufgaben: Erweitere das Script so, daß es mehrere verschiedene Seiten und Zähler zu diesen Seiten bearbeiten kann. Die Seitenbezeichnung wird dem Script in der Variablen `cn` übergeben. Beachte die sich ergebenden Sicherheitsprobleme, wenn Pfadnamen aus Variablen direkt übernommen werden! Wie können diese vermieden werden?

Inhaltsverzeichnis