PERL & CGI

Proseminar

Referenten:

Ümit Kezer 20421002

Selimhan Arayan

Überblick über Perl

"In a nutshell, Perl is designed to make the easy jobs easy, without making the hard jobs impossible." [WALL 1996]

"Practical Extraction and Report Language" wurde von einem einzelnen Entwickler entworfen, von Larry Wall. Der die Sprache im Zuge einer an ihn gestellte Aufgabe entwickelte. Perl stammt aus der Unix-Welt und wurde 1987 in der ersten Version veröffentlicht.
Ursprünglich wurde die Sprache dafür konzipiert um große Mengen Text schnell zu durchsuchen und zu analysieren, mittlerweile wird Perl für nahezu alle denkbaren Zwecke eingesetzt. Daher kommt auch der Name "Practical Extraction and Report Language".
Die Abarbeitung von Log-Dateien um kumulierte Berichte zu erstellen war schon immer eine Perl-Domäne. Aber auch in der Systemverwaltung können mit Perl eine Fülle von Tätigkeiten sehr elegant bewältigt werden.
Ein Grund warum sich diese Sprache solcher Beliebtheit erfreut, könnte sicherlich auch in der Tatsache zu finden sein, dass sich Perl vieler Elemente anderer Programmiersprachen bedient, z. B. C und der Tools *sed* und *awk*. Somit kann jemand, der diese Tools anwenden kann, ohne allzu viel Aufwand Perlprogramme schreiben.
Perl gilt als eine interpretierte Sprache, d. h. Perlprogramme werden als Bytecode kompiliert und zur Laufzeit interpretiert. Des Weiteren kommen viele alltägliche englische Wörter zum Einsatz, die dann auch noch oft genau das tun, was man erwarten würde. So steht dem C-Befehl "include" das für den "Amateur" vielleicht klarer "use" gegenüber.

Entsprechend der Unix-Philosophie ist Perl eine offene Sprache. Ihr Umfang und ihre Möglichkeiten wachsen mit neuen Versionen des Perl-Interpreters. Seit der Versions-Serie 5.x unterstützt Perl z.B. auch den Ansatz der Objektorientierten Programmierung. Jedoch ist es eine Script-Sprache, deren Haupteinsatz nicht umfangreiche Anwendungen sind, sondern trickreiche Automatismen in der täglichen Datenverarbeitung. Einen wahren Boom erlebt die Sprache aber vor allem im Nutzen der CGI-Programmierer im World Wide Web.
Weitere Aufgaben für die Perl sehr gut geeignet ist sind z.B Datenbankmanipulation, Text- und Dokumentenverarbeitung, Datenextraktion und -reduktion, Netzwerkadministration, objektorientierte und funktionale Programmierung etc.

Arbeitsweise

Es handelt sich aus Sicht des Anwenders um eine Interpretersprache, obwohl es eher eine Kreuzung aus Interpreter und Compiler (ähnelt damit Java). Zur Laufzeit wird das Perlskript vom Interpreter relativ schnell in ein internes, bytecode-ähnliches Format umgesetzt. Dabei werden, wie bei jedem Compiler, Syntaxchecks und Optimierungen durchgeführt (syntaktische und zum Kompilierungszeitpunkt erkennbaren semantische Fehler, Probleme beim Einbinden von Bibliotheken, usw.). Nach dem erfolgreichen Durchlauf des Compilers wird der Zwischencode an den Perl-Interpreter zur Ausführung weitergereicht. Compiler und Interpreter sind effizient und der typische Übersetzungszyklus dauert meist nur Sekunden.
Programme werden ähnlich wie MS-DOS-Batch-Programme oder Unix/Linux-Shell-Skripte in Form einer Textdatei und im Gegensatz zu anderen Sprachen wird kein eigenständiges Programm im Binärcode erstellt. Zur Ausführung ist immer ein installierter Perl-Interpreter erforderlich.

Sprachkontext

Perl setzt keine Vereinbarung von Typen, Datenstrukturen, Variablen, Funktionen wie in anderen Programmiersprachen voraus. Für komplexe Probleme ist es jedoch sinnvoll, Variablen am Programmanfnag zu definieren, vorzubesetzen und zu kommentieren. So werden viele Fehler im Vorfeld vermieden - oder schnell gefunden. Wie leicht vertippt man sich an irgendeiner Stelle im Programm bei einem Variablennamen. Dann läuft es natürlich nicht korrekt und gerade solche Fehler sind schwer zu finden. Man kann diese strenge Überprüfung auf korrekt definierte Variablen- und Unterprogrammnamen mit der Anweisung

```
use strict;
```

im Programm erreichen. Andererseits hat man in Perl für die vielen einfachen, alltäglichen Probleme eine Programmiersprache in der man einfach und sofort sagen kann, was man tun will. Es genügt beispielsweise die Zeile

```
print "Hello world!\n":
```

und schon wird `Hello world!` auf der Konsole ausgeben.

Der *Kontext*, in dem ein Literal oder eine Variable verwendet wird, wird durch die verwendeten Operatoren erzwungen. Es gibt zunächst keinen Unterschied zwischen Zahl und Zeichenkette. Paßt der Kontext, wird eine Zeichenkette als Zahl interpretiert. Zum Beispiel ergibt `"33" + "44"` die Zahl 77. Dagegen ergibt `"33" . "44"` die Zeichenkette "3344".

Bsp. „Hello world“ in Perl

```
1.      #!/usr/bin/perl
        #
2.      # Das typische erste Programm
        #
3.4.    print 'Hello world';        # Ausgabe eines Textes
```

1. Fast jedes Perl-Programm beginnt unter UNIX mit der Zeile
   #!/usr/bin/perl
   obwohl die Zeile von System zu System unterschiedlich aussehen kann, sagt sie dem Computer, was er bei der Ausführung zu tun hat. In diesem Fall soll er das Programm durch den Perl-Interpreter schicken.
2. Das **#**-Symbol eröffnet einen Kommentar. Alles zwischen **#** und dem Zeilenende wird vom Interpreter ignoriert (Ausnahme: erste Zeile). Kommentare können überall im Programm verwendet werden. Der einzige Weg um Kommentare über mehrere Zeilen ausdehnen zu können, ist die Verwendung von **#** in jeder Zeile.
3. Alles Übrige sind Perl-Anweisungen, welche mit einem Strichpunkt (Semikolon) beendet werden müssen, wie die letzte Zeile oben.
4. Die print Funktion gibt Information aus. Im obigen Fall gibt sie die Zeichenkette ***Hello world*** aus.

Ein Programm ausführen

Schreiben Sie das Beispielprogramm mit einem Texteditor und speichern Sie es ab. Danach muß es ausführbar gemacht werden. Beispielsweise mit dem UNIX-Kommando
Um das Programm zu starten, können Sie eine der folgenden Möglichkeiten wählen:
perl *progname*
./progname
progname

Variablen

Eine Variable ist ein Behälter für Werte, ein Behälter mit einem Namen, so daß man ihn (und seinen Inhalt) wiederfinden kann. Dabei gibt es Variablen, die nur einen Wert enthalten können, genannt *Skalar*, und Variablen, die mehrere Werte enthalten können, genannt *Array*.
In Perl werden die verschiedenen Arten von Variablen durch spezielle Zeichen markiert:

Typ	**Zeichen**	**Beispiel**	**bezeichnet:**
Skalar	$	$count	einzelnen Wert (Zahl oder String)
Array	@	@namen	Liste von Werten, über numerischen Index ansprechbar
Hash	%	%eingabe	Assoziatives Array aus (String-)Wertepaaren, erster String ist Zugriffsschlüssel
Unterprogramm	&	&convert	Aufrufbarer Perl-Code
Typeglob	*	*irgendwas	Alles was "irgendwas" genannt wird

Reguläre Ausdrücke

Perl beherrscht als eingebautes Feature reguläre Ausdrücke. Dabei gibt es zwei Klassen und zwar die Mustervergleichs-Operatoren "/foo/" und die Ersetzungs-Operatoren "s/foo/bar/". In Vergleichen kann Mustervergleich (pattern matching) mit dem Operator " =~ " angefordert werden. Reguläre Ausdrücke in Perl basieren auf einem NFA (nicht-deterministischer finiter Automat), der folgendermaßen vorgeht in dem er sich die Stellen merkt, an denen mehr als eine Möglichkeit zu kontrollieren ist. Stellt er beim Testen einer Variante fest, dass der Gesamtausdruck nicht mehr zutrifft, geht er zurück zum "Scheideweg" und prüft die Alternative. Erst wenn alle abgehakt sind, entscheidet der NFA, ob der Ausdruck zutrifft oder nicht. Durch dieses "Backtracking" genannte Vorgehen beherrscht Perl nummerierte Rückbezüge wie `s/(Eins) (Zwei)/\2\1/g`. Hier sorgen die Klammern dafür, dass Perl sich jedes "Eins" und jedes "Zwei" merkt. Im zweiten Teil vertauscht dann `\2\1` die beiden miteinander.
Perl versucht normalerweise, den frühesten Treffer im String zu finden. Kommt aber ein Quantifizierer wie `*` ins Spiel, will der NFA soviel wie möglich finden, er wird gierig ("greedy"). Dabei ist die "Gierigkeit" stärker als die "Links-Bindung".
Will man in HTML-Code einen bestimmten Tag erwischen, heißt der erste Versuch vermutlich: `/<.*>/`. Übersetzt: "Suche beliebig viele (auch gar kein) Zeichen, umschlossen von spitzen Klammern."

Was würde Perl nun in der Zeile <B>Wir</B> sind die <B>Champions</B>! finden?
Alles von der ersten spitzen Klammer bis zur letzen vor dem Ausrufezeichen. Da ein Quantifizierer dabei ist, gilt nicht mehr "Treffer soweit links wie möglich" (also `<B>`) sondern "Soviel wie möglich". Perls Gierigkeit lässt sich jedoch durch ein hinter `+` oder `*` gesetztes Fragezeichen beschränken. Benutzt man im obigen Beispiel `<.*?>`, wird es `<B>` finden. In solchen Fällen hilft ebenfalls: `/<[^>]+>/`. Dieser Ausdruck sucht ein <, dann etwas, was kein > ist, davon mindestens eines, schließlich ein >. Ähnlich geht man zum Beispiel vor, um Worte in Anführungszeichen zu finden: `/"[^"]+"/` erledigt diesen Job besser als `/".*"/`.

Die folgende Übersicht fasst die verschiedenen Möglichkeiten zusammen.

/string/	adressiert die nächste Zeile, die 'string' enthält (rückwärts suchen mit ?string?)
^	steht für den Zeilenbeginn. `/^Meier/` adressiert Zeile, die mit "Meier" beginnt
$	steht für das Zeilenende. `/Meier$/` adressiert Zeile, die mit "Meier" endet. `/^$/` adressiert die nächste Leerzeile (Achtung! Doppelbedeutung! Im Suchmuster eines reg. Ausdrucks Zeilenende, als Adresse im ed-Kommando Dateiende).
[]	definiert einen Buchstaben aus dem Bereich in []. Beginnt der Bereich mit ^, wird nach einen Zeichen gesucht, das **nicht** im Breich enthalten ist (Bei Dateinamen war dies das !-Zeichen). `[ABC]`: einer der Buchstaben A, B oder C `[A-Z]`: Großbuchstaben `[A-Za-z]`: alle Buchstaben `[^0-9]`: keine Ziffer
.	der Punkt steht für ein beliebiges Zeichen
*	der Stern "*" steht für eine beliebige Folge des **vorhergehenden** Zeichens (auch Null Zeichen!). `a*`: Leerstring oder beliebige Folge von a's `aa*`: eine beliebige Folge von a's (mindestens eines) `[a-z]*`: Leerstring oder eine beliebige Folge von Kleinbuchstaben `[a-z][a-z]*`: eine beliebige Folge von Kleinbuchstaben (mindestens einer) `.*`: jede beliebige Zeichenfolge
*?	der Stern "*" ist recht "gefräßig" (greedy), d. h. es wird versucht, maximal viele Zeichen in den reg. Ausdruck einzuschließen. Bei der Zeichenkette "aaa:bbb:ccc" würde der Ausdruck ".*:" die Zeichenkette "aaa:bbb:" finden. Durch das nachgestellte Fragezeichen wird diese Eingenschaft umgekehrt, es wird die minimal Teizeichenkette genommen - also im obigen Beispiel "`aaa:`".
+	das Pluszeichen "+" steht für eine beliebige Folge des **vorhergehenden** Zeichens, jedoch mindestens eines. `a+`: ein a oder beliebige Folge von a's
+?	Mindestens einmal das **vorhergehende** Zeichen.
?	Nullmal oder einmal das **vorhergehende** Zeichen.
\	hebt den Metazeichen-Charakter für das folgende Zeichen auf `a*` steht für Leerstring oder beliebige Folge von a's `a\*` steht für die Zeichenfolge "a*"
( )	Reguläre Ausdrücke können mit Klammern gruppiert werden. `([A-Za-z]*)` gruppiert beispielsweise ein Wort aus beliebig vielen Buchstaben, wobei auch ein leeres Wort (0 Buchstaben) dazugehört. Soll das Wort mindestes einen Buchstaben enthalten, muß man `([A-Za-z][A-Za-z]*)` oder `([A-Za-z]+)` schreiben. Die Anwendung solcher Gruppen wird weiter unten gezeigt - es kann in den Editor-Befehlen nämlich Bezug auf die Gruppen genommen werden.

Perl wird in Verbindung mit den folgenden Operatoren verwendet:

- `=~ /.../`: ist enthalten,
- `!~ /.../`: ist nicht enthalten
- `=~ s/.../.../`: Textersetzung und
- `=~ tr/././`: Zeichenersetzung

Beispiele:

```
if ($value =~ /Meier/)        # "Meier" in der Zeichenketten enthalten?
$value =~ tr/+/ /;            # Ersetze "+" durch Leerzeichen
$value =~ tr/A-Z/a-z/;        # Ersetze alle Grossbuchstaben durch
Kleinbuchstaben
$value =~ s/Meier/Huber/;     # Ersetze einmal "Meier" durch "Huber"
$value =~ s/Meier/Huber/g;    # Ersetze alle "Meier" durch "Huber"
```

Der folgende Ausdruck ersetzt alle Zeichenfolgen, die auf das Muster "%-Zeichen, gefolgt von zwei Sedezimalziffern" passen, durch den Buchstaben, der den ASCII-Wert der Hexzahl hat:

```
$value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
```

Wie man sehen kann, ist im Ersetzungsteil auch ein Perl-Ausdruck enthalten.

Kontrollstrukturen

Weitere interessante Möglichkeiten werden mit Kontrollstrukturen und Schleifen eröffnet. Perl unterstützt viele verschiedene Arten von Kontrollstrukturen, welche zum Teil C ähnlich sehen, aber auch an Pascal erinnern.

foreach

Um jedes Element eines Arrays oder einer andern Listenstruktur (wie zun Beispiel die Zeilen eines Files) zu durchlaufen, kann foreach verwendet werden. Das sieht folgendermassen aus:

```
foreach $morsel (@food)                 # Gehe der Reihe nach durch
                                        # @food und nenne das
                                        # Element $morsel
{
        print "$morsel\n";              # Ausgabe des Elementes
        print "Yum yum\n";              # That was nice
}
```

Die Anweisungen, welche jedesmal durchgeführt werden sollen, müssen in geschweiften Klammern angegeben werden. Falls @food leer wäre, würde der Block in geschweiften Klammern nie durchlaufen.

Bool'sche Operatoren

Die nächsten Anweisungen testen die Gleichheit zweier Ausdrücke. In Perl wird jede Zahl ungleich Null und jeder nicht leere String als true betrachtet. Die Zahl Null, Null als String und der leere String sind false. Hier sind ein paar Tests auf Gleichheit für Zahlen und Strings:

```
$a == $b        # $a numerisch gleich $b?
                # Achtung: Nicht = verwenden
$a != $b        # $a numerically ungleich $b?
$a eq $b        # $a gleicher String wie $b?
$a ne $b        # $a und $b nicht der gleiche String?
```

Es gibt auch die logischen AND, OR und NOT:

```
($a && $b)      # ist $a und $b true?
($a || $b)      # ist entweder $a oder $b true?
!($a)           # ist $a false?
```

for

Die **for**-Struktur von Perl ist vergleichbar mit C:

```
for (init; test; inkr)
{
        first_action;
        second_action;
        etc
}
```

Zuerst wird *init* ausgeführt. Falls *test* true ist, wird der Block von Anweisungen in geschweiften Klammern ausgeführt. Nach jedem Durchgang wird *inkr* ausgeführt. Mit der folgenden **for**-Schleife werden die Zahlen 1 bis 9 ausgegeben:

```
for ($i = 0; $i < 10; ++$i)     # Starte mit $i = 1
                                # falls $i < 10
                                # Inkr. $i vor Block
{
        print "$i\n";
}
```

Die reservierten Wörter for und foreach können für beide Strukturen verwendet werden. Perl merkt dann schon, was gemeint ist!

while und until

Das folgende Programm liest eine Eingabe von Keyboard und wiederholt die Anfrage, bis das korrekte Passwort eingegeben wurde.

```
#!/usr/local/bin/perl
print "Password? ";             # Fragt nach Input
$a = <STDIN>;                   # liest input
chop $a;                        # löscht \n am Ende
while ($a ne "fred")            # Solange $a falsch ist
{
   print "sorry. Again? ";      # Fragt wieder
   $a = <STDIN>;                # liest
   chop $a;                     # löscht \n am Ende
}
```

Die **while**-Schleife sollte soweit klar sein. Wie immer haben wir einen Block von Anweisungen in geschweiften Klammern. Ein paar Dinge können wir hier noch erwähnen: Erstens können wir von STDIN lesen, ohne es zuerst geöffnet zu haben. Zweitens wird mit der Eingabe des Passwortes auch ein newline-character in $a abgespeichert. Drittens ist die **chop**-Function dazu da, den letzten Buchstaben eines Strings abzuschneiden, in diesem Fall den newline-character.

Um das Gegenteil zu testen, können wir die **until**-Schleife in genau gleicher Weise verwenden. Damit wird der Block solange ausgeführt, *bis* der Test true ist und nicht *solang*e er true ist.

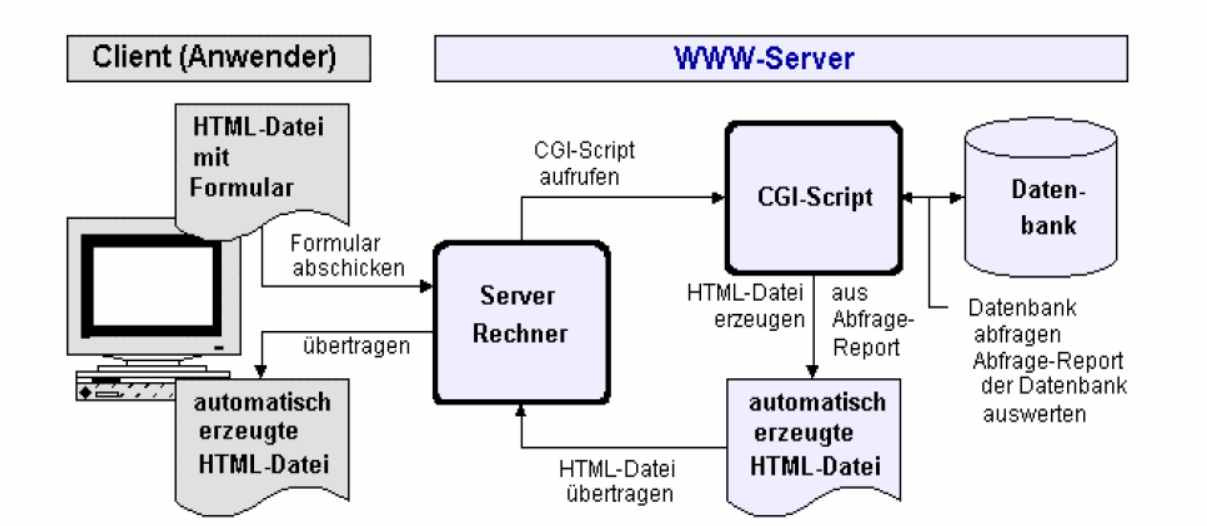
Eine andere Art vorzugehen, ist die Verwendung des **while**- oder **until**-Tests am Ende des Blocks anstatt am Anfang. Dazu benötigt man noch den **do**-Operator, welcher den Blockanfang markiert. Falls wir die Meldung *sorry. Again* weglassen, könnten wir das Passwortprogramm folgendermassen schreiben:

```
#!/usr/local/bin/perl
do
{
        print "Password? ";            # Fragt nach Input
        $a = <STDIN>;                  # liest Input
        chop $a;                       # löscht \n
}
while ($a ne "fred");                  # Nochmals solange falscher Input
```

Common Gateway Interface (CGI)

Nachdem die HTTP-Spezifikation entwickelt wurde, welche es erstmals ermöglichte, Daten zur Anzeige als Website zu bringen, wurde schnell klar, dass dies nicht ausreichend ist, um die schnell steigenden Bedürfnisse zu befriedigen. Man musste die Eigenschaft eines Servers herausbilden, Clienten zu erlauben auch Daten auf ihm speichern und modifizieren zu können. Aus dieser Idee heraus entstand das Common Gateway Interface, das sich bis heute behaupten konnte.
Die Sprache Perl ist meist bevorzugt beim schreiben von CGI-Programmen, da sie plattformübergreifend verfügbar ist und sich besonders zur Manipulation von Texten eignet - und das ist meist die Hauptanwendung für CGI-Programme.
Das CGI beschreibt nur ein Protokoll, nicht eine bestimmte Programmiersprache. CGI-Programme lassen sich somit in jeder auf dem Server verfügbaren Programmiersprache erstellen. Übersetzt heißt CGI eine allgemeine Vermittlungsrechner-Schnittstelle und ist eine Möglichkeit, Programme im WWW bereitzustellen, die von HTML-Dateien aus aufgerufen werden können, und die selbst HTML-Code erzeugen und an einen WWW-Browser senden können.
CGI-Programme liegen auf einem Server-Rechner im Internet und bei Aufruf verarbeiten sie bestimmte Daten. Die Datenverarbeitung geschieht auf dem Server-Rechner. CGI-Programme können auf dem Server-Rechner Daten speichern, zum Beispiel, wie oft auf eine WWW-Seite zugegriffen wurde oder was ein Anwender in ein Gästebuch geschrieben hat.
Bei entsprechendem Aufruf kann ein CGI-Programm gespeicherte Daten auslesen und daraus HTML-Code generieren. Dieser "dynamisch" erzeugte HTML-Code wird an den aufrufenden WWW-Browser eines Anwenders betragen und kann dort individuelle Daten in HTML-Form anzeigen, zum Beispiel den aktuellen Zugriffszählerstand einer WWW-Seite oder die bisherigen Einträge in einem Gästebuch.

Typische CGI-Situation, wie man zum Beispiel für Suchdienste im WWW



In dem Beispiel kann der Anwender in einer angezeigten HTML-Datei in einem Formular Daten eingeben, zum Beispiel eine Suche in einer Datenbank formulieren. Nach dem Abschicken des Formulars an den Server-Rechner wird ein CGI-Programm aufgerufen. Das CGI-Programm setzt die vom Anwender eingegebenen Daten in eine Datenbankabfrage um. Wie das genau funktioniert, hängt von der Datenbank ab.

Es gibt eine international standardisierte Datenbankabfragesprache, SQL, die hierbei sehr häufig zum Einsatz kommt. Die Datenbankanwendung liefert die Suchergebnisse an das aufrufende CGI-Programm zurück (oder schreibt sie in eine Datei, die das CGIProgramm dann auslesen kann). Das CGI-Programm erzeugt nun HTML-Code, wobei es die Suchergebnisse als Daten in den HTML-Code einbaut. Den HTML-Code sendet das CGI-Programm an den WWW-Browser, der die Suchabfrage gestartet hat. Am Bildschirm des Anwenders verschwindet die WWW-Seite mit dem Suchformular. Statt dessen erscheint eine neue Seite mit den Suchergebnissen, dynamisch generiert von dem CGI-Programm.

CGI-Schnittstelle

Die CGI-Schnittstelle besteht aus einem bestimmten Verzeichnis auf dem Server-Rechner, das CGI-Programme erhalten darf. Meist erhält dieses Verzeichnis den Namen cgi-bin oder cgi-local.Desweiteren von einer Reihe von Daten, die der WWW-Server speichert, und die ein CGI-Script auslesen kann, um Daten verarbeiten zu können. Diese Daten speichert WWWServer in sogenannten CGI-**Umgebungsvariablen**. Es gibt auch andere Schnittstellen für ausführbare Programme im WWW, die von kommerziellem Hersteller wie Netscape (API-Schnittstelle) oder Microsoft (ISAPISchnittstelle) eingeführt wurden, machen der klassischen CGI-Schnittstelle jedoch zunehmend Konkurrenz. Ein entscheidbarer Vorteil der CGI-Schnittstelle bleibt die Tatsache, daß es sich – ähnlich wie bei HTML – um einen kommerziell unabhängigen, kostenlosen, Produktübergreifenden Standard handelt.

Daten empfangen

Damit man dem Script Anfragen mitteilen kann müssen einfach Parameter an die URL angehängt werden. Mit dem Fragezeichen werden die Parameter vom Programmnamen getrennt. Alles, was danach folgt, wird als Parameter erfasst. In einer Parameterzeile dürfen keine Leerzeichen stehen, weshalb sie durch ein "+"-Zeichen ersetzt werden. Sonderzeichen (Code > 127) werden durch ein Prozentzeichen (%) und den hexadezimalen Code des Zeichens ersetzt. Man kann also z. B. für ein Shop-Programm ein Bestell-Link der folgenden Form in eine Webseite einfügen:

```
<A HREF="http://shop.server.de/cgi-
bin/shopper.pl?prodid=23481227">Bestellen</A>
```

Genauso häufig sind Daten, die durch die Eingabe in ein Formular erzeugt werden. Wenn der Client seine Daten übermittelt (Submit-Knopf im Formular), erhält das Script alle erzeugten Daten als einen Satz von Name-Wert-Paaren. Der Name ist jeweils der, den man beim `INPUT`-Tag (bzw. beim `SELECT`- oder `TEXTAREA`-Tag) festgelegt hat, die Werte sind das, was der Anwender ins Formular eingetragen oder gewählt hat. Dieser Satz von Name-Wert-Paaren wird in einem einzigen langen String übermittelt, den das CGI-Programm auflösen muss. Das ist nicht sehr kompliziert, und es gibt viele fertige Routinen dazu. Der Aufbau des Strings ist recht einfach:

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

Man muss den String also einfach beim &-Zeichen zerlegen. Dann erhält man die Paare

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

So muss man nun noch

- alle "+"-Zeichen in Leerzeichen wandeln
- alle "%xx"-Sequenzen in einzelne Buchstaben mit dem ASCII-Wert "xx" wandeln. "xx" ist da bei eine Hexadezimalzahl (Beispiel: "%3D" wird zu "=")

Das kommt daher, dass der übermittelte String in der URL des CGI-Programms codiert ist. Bei einer Form der Übermittlung zum Server wird er durch "?" getrennt an die URL angehängt, was man beispielsweise beim Bedienen einer Suchmaschine in der URL-Zeile des Browsers sehen kann. Der Anwender muss aber nach wie vor die Möglichkeit haben, &-Zeichen und Gleichheitszeichen zu übergeben - deshalb die Codierung durch "%xx". Die folgende Tabelle zeigt die Zeichencodierung:

Zeichen	Code	Zeichen	Code	Zeichen	Code	Zeichen	Code
Leerz.	+	!	%21	"	%22	#	%23
$	%24	%	%25	&	%26	'	%27
(	%28	)	%29	+	%2B	,	%2C
/	%2F	:	%3A	;	%3B	<	%3C
=	%3D	>	%3E	?	%3F	[	%5B
\	%5C	]	%5D	^	%5E	"	%60
{	%7B	|	%7C	}	%7D	~	%7E
°	%A7	Ä	%C4	Ö	%D6	Ü	%DC
ß	%DF	ä	%E4	ö	%F6	ü	%FC

Von wo das CGI-Programm den Formularinhalt erhält, hängt von der Methode ab, mit der die HTTP-Form übermittelt wurde:

Bei der **GET-Methode** wird die in der <FORM ACTION="URL"> angegebene URL genommen und die codierten Eingabedaten angehängt, getrennt durch ein "?". Dies könnte zum Beispiel so aussehen:

```
http://.../cgi-bin/testcgi.pl?Text=Hallo+dies+ist+ein+Test&Zeichen=%25
```

Diese URL wird beim Absenden der Anfrage auch durch den Browser angezeigt. Die Nachteile der GET-Methode sind leicht ersichtlich:
-Es eignet sich nur bei relativ geringen Datenmengen
-Die URLs werden unleserlich.
Der Vorteil liegt darin, dass diese Darstellung für den CGI-Programmierer sehr hilfreich sein kann beim Debugging seinen Scriptes. Außerdem ist GET schneller als POST.
Bei GET-Übermittlungen liegt das Ergebnis in der Environment-Variable QUERY_STRING. Normalerweise benutzt man GET um kleine Datenmengen zu übertragen, z. B. einen Suchbegriff - vielleicht mit ein paar Parametern versehen. Dabei dürfen nicht zu viele Daten übertragen werden, denn die Maximallänge der URL variiert von Browser zu Browser.

Bei der **POST-Methode** bleibt die in <FORM ACTION="URL"> angegebene URL unverändert. Die Daten werden für den Benutzer unsichtbar an den Server übertragen. Dies hat aber den Nachteil, dass man die URL nicht bookmarken kann. Während man z. B. bei einer Suchmaschine, welche mit GET arbeitet, bestimmte Begriffe als Link in die Bookmarks eintragen kann, und so immer einen aktuellen Link auf einen Suchbegriff haben kann, kann man dies bei Post nicht machen.
Bei **POST-Übermittlungen** muss man die Formulardaten von `STDIN` lesen. Die genaue Anzahl übermittelter Bytes steht dabei in der Variablen `CONTENT_LENGTH`.
POST wird normalerweise dafür verwendet, dem Server eine größere Menge Daten zu übermitteln. Die Menge der übertragenen Daten ist im Gegensatz zu GET nicht begrenzt.

All das geschieht hinter den Kulissen. Für den CGI-Programmierer funktionieren GET und POST fast gleich und sind gleich einfach zu benutzen. Der Vorteil von POST ist, dass man beliebig viele Daten übertragen kann. Der Vorteil von GET ist, dass alle Daten in eine URL gebastelt sind, man kann auf sie also verweisen oder sie in die Bookmarks aufnehmen.

Hier sind noch einmal die Schritte aufgeführt, die ein CGI-Script normalerweise durchlebt, wenn es vom Benutzer aufgerufen wird:

- Wird das Script ohne Parameter gerufen, gibt es eine Standardseite aus. Meist handelt es sich dabei um ein Formular.
- Der Benutzer füllt das Formular aus und schickt es ab. Das Formular kann natürlich auch auf einer HTML-Seite angeboten werden (dann entfällt der erste Punkt).
- Der Server empfängt die Anfrage und ruft das Script - diesmal mit Parametern - auf.
- Das Script verarbeitet die Parameter und gibt eine Ausgabe dazu aus.

Dem Client eine Antwort geben

Der Browser sendet die Anfrage an ein CGI-Programm genauso ab, wie die Anforderung eines HTML-Dokuments und er werwartet natürlich auch, daß der Server mit entsprechenden Daten antwortet. Bei CGI wird jedoch keine HTML-Seite gesendet, sondern das Programm erzeugt die Daten dynamisch. Als erstes gibt es die Zeile

```
Content-Type: text/html
```

gefolgt von einer Leerzeile auf `die Standardausgabe` aus. Nun läßt es eine normale HTML-Antwort folgen, die es ebenfalls auf `die Standardausgabe` ausgibt. Wenn das Script beendet ist, sieht der Anwender die so erzeugte Seite. Das Script liegt in einem speziellen Verzeichnis auf dem Server (meist "cgi-bin"). Der Content-Type kann natürlich variieren, es sind alle gebräuchlichen und vom Browser akzepierten MIME-Typen erlaubt. Versuchen wir es mal mit einem einfachen Beispiel.

Das Formular im Beispiel mit CGI

```
1:  <HTML>
2:  <HEAD>
3:  <TITLE>Tell Me Your Name</TITLE>
4:  </HEAD>
5:  <BODY>
6:  <FORM ACTION="/cgi-bin/name.pl">
7:  <P>Geben Sie Ihren Namen ein: <INPUT NAME="name">
8:  <P><INPUT TYPE="SUBMIT" VALUE="Abschicken!">
9:  </FORM>
10: </BODY>
11: </HTML>
```

Anmerkungen zum HTML-Code:

- In Zeile 6 verweist das Attribut ACTION auf das CGI-Skript, das dieses Formular verarbeiten wird, wenn es an den Server geschickt wird. In unserem Beispiel heißt das Skript ***name.pl*** und steht auf dem Server im Verzeichnis ***cgi-bin*** (dem regulären Speicherort für CGI-Skripts; das Verzeichnis muss jedoch nicht bei allen Servern gleich lauten; es kann auch erforderlich sein, dass Sie erst die Erlaubnis einholen müssen, bevor Sie Ihre Skripts dort ablegen). Passen Sie den Pfad gegebenenfalls an.

- Zeile 7 definiert ein Textfeld als Formularelement (<INPUT>-Tag). Mit Hilfe des Attributs NAME geben Sie diesem Element einen Namen. Dies wird von Bedeutung sein, wenn Sie das CGI-Skript für das Formular erstellen.

Das CGI-Skript fürs Formular

Ein CGI-Skript wird in Perl grundsätzlich in der gleichen Weise aufgesetzt wie ein normales Perl-Skript, das von der Befehlszeile aus ausgeführt wird. Es gibt jedoch einige wesentliche Unterschiede, die darauf beruhen, dass Ihr Skript vom Webserver aufgerufen wird und nicht von Ihnen. So erhalten Sie zum Beispiel keine Optionen oder Dateinamen- Argumente über die Befehlszeile. Alle Daten, die Sie im Skript erhalten, kommen vom Webserver (oder werden von Ihnen selbst aus Dateien auf der Festplatte eingelesen). Die Ausgabe Ihres Skripts muss in einem bestimmten Format vorliegen - normalerweise HTML.

Wir fangen mit der obersten Zeilen unseres CGI-Skripts an:

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:standard);
```

Die Shebang-Zeile und `use strict` ist Bekannt und die dritte Zeile `use CGI kann man sich leicht denken wofür es steht.`
Die meisten CGI-Skripts bestehen aus zwei Teilen: Der erste Teil liest die Daten ein, die Sie von einem Formular oder dem Webbrowser erhalten haben, und verarbeitet sie. Der zweite Teil gibt eine Antwort aus, die in der Regel im HTML-Format erzeugt wird. Da es in unserem Beispiel kaum etwas zu verarbeiten gibt, gehen wir gleich zum Ausgabeteil über.

Das erste, was wir ausgeben ist ein besonderer Header an den Webserver, der ihm mitteilt, welche Art von Datei Sie zurücksenden. Senden Sie eine HTML- Datei zurück, lautet der Typ `text/html`. Ist es eine einfache Textdatei, lautet der Typ `text/plain`. Für eine Grafik ist es `image/gif`. Diese Dateitypen sind alle als Teil der MIME-Spezifikation standardisiert, und wenn man tiefer in die CGI-Skripterstellung einsteigt, wird man sich mit zumindest einigen dieser Formate näher vertraut gemacht. Das geläufigste Format ist zweifelsohne `text/html`. `CGI.pm` stellt Ihnen eine grundlegende Subroutine namens `print_header()` zur Verfügung, die den entprechenden Header für diesen Formattyp ausgibt.

```
print header();
```

Alle weiteren Ausgaben, die auf den Header folgen, müssen jetzt im HTML-Format sein. Sie können entweder reine HTML-Tags ausgeben, die Perl-Subroutinen von `CGI.pm` verwenden, um den HTML-Code zu erzeugen, oder beide Methoden miteinander kombinieren.
Man sollte die Subroutine von CGI.pm nur dann vorziehen, wenn man Tipparbeit spart, ansonsten sollte man normale `print`-Anweisungen verwenden.

```
print start_html('Hallo!');
print "<H1>Hallo, ", param('name'), "!</H1>\n";
print end_html;
```

Die erste Zeile ruft die `CGI.pm`-Subroutine `start_html()` auf, die den ersten Teil einer jeden HTML-Datei ausgibt (die Tags `<HTML>`, `<HEAD>`, `<TITLE>` und `<BODY>`). Das String-Argument für `start_html()` ist der Titel der Seite. Sie können dieser Subroutine aber auch andere Argumente übergeben, um zum Beispiel Hintergrundfarbe, Schlüsselwörter und andere Besonderheiten des Headers zu setzen (mehr dazu im nächsten Abschnitt).

Die zweite Zeile ist eine reguläre `print`-Anweisung, die eine HTML-Überschrift (`<H1>`- Tag) zum Hallo-Sagen ausgibt. Zwischen dem öffnenden und dem schließenden Tag befindet sich der interessante Teil. Die Subroutine `param()` ist ebenfalls Teil von `CGI.pm` und dient dazu, an die Informationen zu gelangen, die der Benutzer in das Formular eingegeben hat. Rufen man dazu `param()` mit dem Namen des Formularelements (aus dem `NAME`-Attribut des HTML-Tags des Formularelements) auf, so liefert die Routine den Wert zurück, den der Benutzer für das Element eingegeben hat. Mit dem Aufruf von `param('name')` erhalten wir den String, den der Benutzer in das Textfeld unseres Formulars eingegeben hat. Mit diesem Wert können wir dann die Antwort erzeugen.

Die dritte Zeile ist eine weitere Subroutine von `CGI.pm`, die lediglich die abschließenden HTML-Elemente (`</BODY>` und `</HTML>`) ausgibt und damit die Antwort komplettiert.

Das komplette Skript:

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:standard);

print header;
print start_html('Hallo!');
print "<H1>Hallo, ", param('name'), "!</H1>\n";
print end_html;
```

Es fällt auf, dass wir für die Ausgabe eigentlich keine besonderen Schritte unternommen haben. Wir haben lediglich einfache `print`-Anweisungen verwendet. Und doch geht die Ausgabe nicht an den Bildschirm, sondern zurück an den Browser. Damit ist ein CGI-Skript ein Paradebeispiel dafür, dass die Standardeingabe und -ausgabe nicht unbedingt die Tastatur oder der Bildschirm sein müssen. CGI-Skripts lesen Ihre Eingaben von der Standardeingabe und schreiben ihre Ausgabe in die Standardausgabe, nur das in diesem Falle die Standardeingabe und - ausgabe der Webserver ist. Sie müssen keine besonderen Vorkehrungen treffen, um mit dem Server zu kommunizieren; über die Standardwege klappt das reibungslos.

Wenn der Benutzer in das Formular HANS eingegeben hätte, würde die Ausgabe des CGI-Skripts samt Header und HTML-Daten wie folgt aussehen:

```
Content-type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Hallo!</TITLE>
</HEAD><BODY><H1>Hallo, HANS!</H1>
</BODY></HTML>
```

Diese Daten werden über die Standardausgabe an den Webserver übergeben, der sie wiederum an den Webbrowser von HANS zurückgibt.

Das Skript testen

Bevor Sie ein Skript auf Ihren Webserver installieren, ist es angebracht, das Skript zu testen, um sicherzustellen, dass Sie keine groben Fehler gemacht haben. Mit Hilfe von `CGI.pm` kann man das CGI-Skript zum Test von der Befehlszeile aus starten:

```
% name.pl
(offline mode: enter name=value pairs on standard input)
```

Nach dieser Zeile kann man die Formulareingabe in Form von Name/Werte-Paaren simulieren. So müsste man zum Beispiel für das Hallo-Formular als Name `name` angeben, und der Wert wäre dann irgendein Name (zum Beispiel `HANS`). Ihre Eingabe sähe dann wie folgt aus:

```
name=HANS
```

Solange die Eingabe keine Leerzeichen enthält, muss man keine Anführungszeichen setzen. Nachdem man Ihre Name/Werte-Paare eingegeben haben, drücken Sie `[Strg]+[D]` (`[Strg]+[Z]` für Windows), um die Standardeingabe zu beenden. Das Skript wird dann ausgeführt, als ob es die Eingabe vom Formular erhalten hätte, und das Ergebnis wird auf dem Bildschirm ausgegeben.

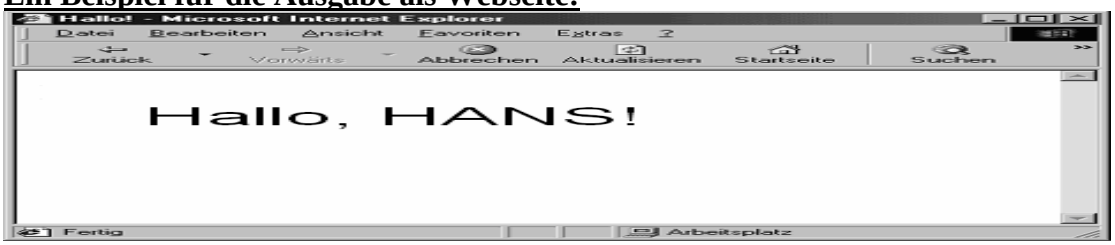
Alternativ kann man die Namen und Werte auch als Argumente in der Befehlszeile des Skripts eingeben:

```
% name.pl name=HANS
```

Das Skript wird dann diese Name/Werte-Paare als Eingabe betrachten und Sie nicht nach weiteren Eingaben fragen.

Nachdem man weiß, dass das CGI-Skript so funktioniert, wie wir es erwarten, besteht der letzte Schritt darin, das Skript auf dem Webserver zu installieren. Installation kann bedeuten, dass man den Skript in einen besonderen Verzeichnis namens `cgi-bin` ablegen muss, die Skriptextension in `.cgi` umbenennen oder auf anderweitige Art und Weise dem Webserver signalisieren, dass es sich bei Ihrem Skript um ein CGI-Skript handelt. (Auch hier gilt, dass sich diese Anforderungen von Server zu Server unterscheiden können, so dass Sie auf alle Fälle in Ihrer Server- Dokumentation über Einzelheiten informieren sollten). Eventuell müssen Sie auch sicherstellen, dass das Skript eine ausführbare Datei ist, oder spezielle Zugriffsberechtigungen vergeben sind. Abschließend ändern wir noch die Original- HTML-Datei, so dass wir auf die aktuelle Position des Skripts zeigen.
Nach all den Tätigkeiten sollte es jetzt möglich sein, in die HTML-Datei mit dem Formular einen Namen einzugeben, den **Abschicken**-Schalter anzuklicken und vom CGI-Skript eine Antwort zu erhalten.

Ein Beispiel für die Ausgabe als Webseite:



Quellen:

http://www.on-luebeck.de/doku/tek-perl/

http://homepage.ruhr-uni-bochum.de/Uwe.Hunz/perl/perl.html

http://www.rolandgeyer.at/kurse/german/perlsrc/ps001.html

http://www.html-world.de/program/cgi1.php

Farid Hajji: Perl - Einführung, Anwendungen, Referenz
2., aktualisierte und erweiterte Auflage
Addison-Wesley Longman,