18

Kompilierung

Wenn Sie hier sind, um nach einem Perl-Compiler zu suchen, werden Sie vielleicht überrascht sein zu entdecken, daß Sie bereits einen besitzen – Ihr *perl*-Programm (üblicherweise */usr/bin/perl*) enthält bereits einen Perl-Compiler. Das ist möglicherweise nicht das, woran Sie dachten, und wenn dem so ist, werden Sie erfreut sein zu hören, daß wir auch *Codegeneratoren* bereitstellen (die einige Leute wohlmeinend als »Compiler« bezeichnen). Diese werden aber erst gegen Ende dieses Kapitels behandelt. Zuerst möchten wir über das reden, was wir als »den Compiler« betrachten. Es ist unvermeidlich, daß es in diesem Kapitel eine ganze Reihe auf niedriger Ebene angesiedelter Details gibt, die einige Leute interessieren werden, andere hingegen nicht. Wenn Sie das nicht interessiert, betrachten Sie dieses Kapitel einfach als Möglichkeit, Ihre Fähigkeiten im Schnellesen zu trainieren.

Stellen Sie sich vor, ein Dirigent zu sein, der die Partitur für ein großes Orchesterwerk geordert hat. Sobald das Paket ankommt, finden Sie mehrere Dutzend Hefte vor, eins für jedes Mitglied des Orchesters, in dem nur dessen jeweiliger Teil enthalten ist. Seltsamerweise fehlt aber das Hauptbuch, in dem alle Teile enthalten sind. Und merkwürdigerweise sind die Teile, *die* vorhanden sind, in Deutsch und nicht in einer für Musik geeigneten Notation. Bevor Sie die Aufführung zusammenstellen oder die Musik überhaupt dem Orchester geben können, müssen Sie den Text erst einmal in das normale Notensystem übersetzen. Dann müssen Sie die einzelnen Teile zu einer großen Partitur zusammenfassen, um überhaupt eine Vorstellung von der Aufführung zu bekommen.

Ähnlich verhält es sich, wenn Sie den Quellcode Ihres Perl-Skripts an *perl* zur Ausführung übergeben. Das Skript ist keinen Deut nützlicher für den Computer, als es die deutsche Beschreibung der Symphonie für die Musiker war. Bevor Ihr Programm ausgeführt werden kann, muß Perl diese englisch aussehenden Anweisungen in eine spezielle symbolische Darstellung kompilieren[1]. Ihr Programm läuft immer noch nicht, weil der Compiler nur kompiliert. Wie bei der Partitur des Dirigenten wird – selbst nachdem Ihr Programm in ein für die Interpretation geeignetes Format übersetzt wurde – immer noch ein aktiver Agent benötigt, der diese Anweisungen interpretiert.

1 Oder übersetzen oder transformieren oder umgestalten oder umwandeln oder ummodeln.

Der Lebenszyklus eines Perl-Programms

Sie können den Lebenszyklus eines Perl-Programms in vier verschiedene Phasen aufteilen, von denen jede wiederum eigene Phasen besitzt. Die erste und die letzte Phase sind die interessantesten, und die beiden mittleren sind optional. Diese Phasen werden in Abbildung 18-1 dargestellt.

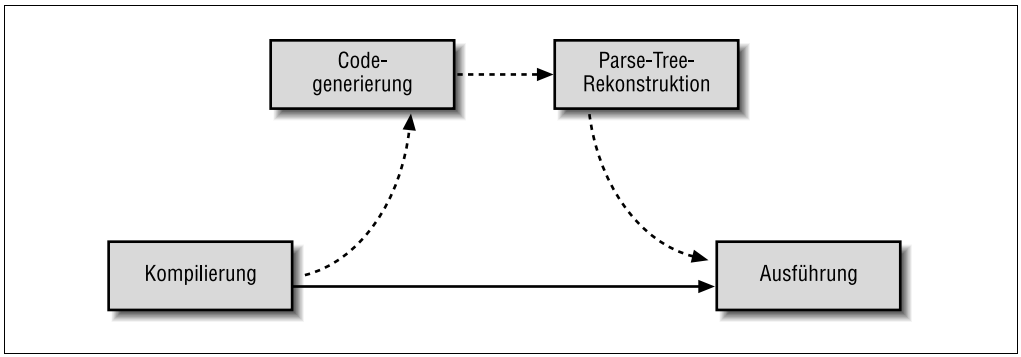

Abbildung 18-1: Der Lebenszyklus eines Perl-Programms

1. *Die Kompilierungsphase*

   Während der ersten Phase, der *Kompilierungsphase*, wandelt der Perl-Compiler Ihr Programm in eine Datenstruktur um, die als *Parse Tree* bezeichnet wird. Neben den Standard-Parsing-Techniken setzt Perl eine wesentlich leistungsfähigere ein: Es verwendet `BEGIN`-Blöcke zur Lenkung der weiteren Kompilierung. `BEGIN`-Blöcke werden an den Interpreter zur Ausführung übergeben, sobald diese vom Parser verarbeitet wurden, was dazu führt, daß sie in einer FIFO-Reihenfolge (»first in, first out«) abgearbeitet werden. Das schließt alle `use`- und `no`-Deklarationen ein, die eigentlich nur verschleierte `BEGIN`-Blöcke sind. Bei allen `CHECK`-, `INIT`- und `END`-Blöcken verzögert der Compiler hingegen die Ausführung.

   Lexikalische Deklarationen werden notiert, entsprechende Zuweisungen werden aber nicht ausgeführt. Jedes `eval` *`BLOCK`*, alle `s///e`-Konstrukte und alle nicht interpolierten regulären Ausdrücke werden hier kompiliert, und konstante Ausdrücke werden direkt ausgewertet. Der Compiler ist nun fertig, sofern seine Dienste später nicht wieder benötigt werden. Am Ende dieser Phase wird der Interpreter erneut aufgerufen, um alle `CHECK`-Blöcke in LIFO-Reihenfolge (»last in, first out«) auszuführen. Das Vorhandensein bzw. Fehlen eines `CHECK`-Blocks bestimmt, ob wir nun mit Phase 2 weitermachen oder direkt zu Phase 4 springen.

2. *Die Codegenerierungsphase (optional)*

   `CHECK`-Blöcke werden von Codegeneratoren installiert, so daß diese optionale Phase durchlaufen wird, wenn Sie explizit einen der später im Abschnitt »Codegeneratoren«) beschriebenen Codegeneratoren verwenden. Diese wandeln das kompilierte

(aber noch nicht ausgeführte) Programm entweder in C-Quellcode oder in serialisierten Perl-Bytecode (eine Reihe von Werten, die interne Perl-Anweisungen ausdrücken) um. Wenn Sie sich für die Generierung von C-Quellcode entscheiden, kann letztendlich eine Datei erzeugt werden, die als *ausführbares Image* (in der maschineneigenen Sprache) bezeichnet wird.[2]

An diesem Punkt wird Ihr Programm zum Leben erweckt. Bei einem ausführbaren Image geht es direkt mit Phase 4 weiter, anderenfalls muß in Phase 3 der gefriergetrocknete Bytecode wiederhergestellt werden.

3. *Die Parse-Tree-Rekonstruktionsphase (optional)*

   Um Ihr Programm zu reanimieren, muß der Parse-Tree rekonstruiert werden. Diese Phase existiert nur, wenn die Codegenerierung erfolgt ist und Sie sich entschieden haben, Bytecode zu generieren. Perl muß zuerst seine Parse-Trees aus dieser Bytecode-Sequenz wiederherstellen, bevor das Programm ausgeführt werden kann. Perl führt diese Bytecodes nicht direkt aus, weil das langsam wäre.

4. *Die Ausführungsphase*

   Schließlich kommt das, worauf wir alle gewartet haben: Ihr Programm wird ausgeführt. Daher auch der Name *Ausführungsphase*. Der Interpreter nimmt sich den Parse-Tree (den er entweder direkt vom Compiler oder indirekt von einer Codegenerierung und der darauffolgenden Parse-Tree-Rekonstruktion erhalten hat) und führt ihn aus. (Wenn Sie ein ausführbares Image erzeugt haben, kann das Programm auch selbständig ausgeführt werden, weil es einen eingebetteten Perl-Interpreter enthält.)

   Zu Beginn dieser Phase, bevor Ihr Hauptprogramm startet, werden alle vorgemerkten `INIT`-Blöcke in FIFO-Reihenfolge ausgeführt. Dann erst wird das Hauptprogramm ausgeführt. Der Interpreter kann wieder auf den Compiler zurückgreifen, wenn er ein `eval` *`STRING`*, ein `do` *`FILE`*, eine `require`-Anweisung, ein `s///ee`-Konstrukt oder ein Pattern-Matching mit einer interpolierten Variablen entdeckt, die eine zulässige Code-Zusicherung (Assertion) enthält.

   Wenn Ihr Programm endet, werden alle bis jetzt verzögerten `END`-Blöcke schließlich ausgeführt, diesmal in LIFO-Reihenfolge. Der allererste Block wird zuletzt ausgeführt, und dann sind Sie fertig. (`END`-Blöcke werden nur übergangen, wenn Sie mit `exec` arbeiten oder wenn Ihr Prozeß aufgrund eines nicht abzufangenden katastrophalen Fehlers abstürzt. Normale Ausnahmen werden nicht als Katastrophen betrachtet.)

Nun wollen wir diese Phasen etwas detaillierter und in einer etwas anderen Reihenfolge erläutern.

2 Ihr Originalskript ist ebenfalls eine ausführbare Datei, aber nicht in Maschinensprache, weshalb wir sie nicht als Image bezeichnen. Eine Imagedatei wird so genannt, weil sie eine exakte Kopie der Maschinencodes enthält, die Ihre CPU direkt ausführen kann.

Ihren Code kompilieren

Perl befindet sich immer in einem von zwei Betriebsmodi: Entweder kompiliert es Ihr Programm, oder es führt es aus – niemals erfolgt beides zur gleichen Zeit. Im gesamten Buch verweisen wir darauf, daß bestimmte Dinge während des Kompilierens passieren, oder wir sagen, der Perl-Compiler täte dieses und jenes. An anderen Stellen erwähnen wir, daß etwas zur Laufzeit passiert oder daß der Perl-Interpreter dieses oder jenes tut. Obwohl Sie sich den Compiler und den Interpreter zusammen einfach als »Perl« vorstellen können, ist es wichtig zu verstehen, welche dieser beiden Rollen Perl an einem bestimmten Punkt spielt. Nur so ist zu verstehen, warum viele Dinge so geschehen, wie sie geschehen. Das *perl*-Executable implementiert beide Rollen: zuerst den Compiler, dann den Interpreter. (Andere Rollen sind ebenfalls möglich: *perl* ist auch ein Optimizer und ein Codegenerator. Gelegentlich ist es sogar ein Schwindler – aber es meint es nur gut.)

Es ist auch wichtig, die Unterscheidung zwischen der Compilerphase und dem Zeitpunkt der Kompilierung sowie zwischen der Ausführungsphase und der Laufzeit zu verstehen. Ein typisches Perl-Programm besitzt eine Compilerphase und eine Ausführungsphase. Eine »Phase« ist ein Konzept großen Umfangs, während die Kompilierungszeit und die Laufzeit Konzepte kleineren Umfangs darstellen. Eine gegebene Compilerphase erledigt größtenteils Dinge, die zur Kompilierungszeit erfolgen, erledigt aber auch einige Laufzeit-Arbeiten über `BEGIN`-Blöcke. Eine bestimmte Ausführungsphase erledigt größtenteils Dinge, die zur Laufzeit passieren, kann aber auch Dinge der Kompilierungszeit enthalten, beispielsweise durch Operatoren wie `eval` *`STRING`*.

Beim typischen Lauf der Ereignisse geht Perl den gesamten Quellcode Ihres Programms durch, bevor die Ausführung beginnt. Zu dieser Zeit untersucht der Perl-Parser die Deklarationen, Anweisungen und Ausdrücke auf deren syntaktische Richtigkeit.[3] Findet der Compiler einen Syntaxfehler, versucht er, aus diesem wieder herauszukommen, damit er auch noch weitere Fehler melden kann, die später im Quelltext vorkommen. Manchmal funktioniert das, manchmal aber auch nicht. Syntaxfehler haben die unangenehme Neigung, eine ganze Reihe von Fehlalarmen hinter sich herzuziehen. Perl gibt nach etwa zehn Fehlern frustriert auf.

Neben dem Interpreter, der die `BEGIN`-Blöcke übernimmt, verarbeitet der Compiler Ihr Programm mit Hilfe dreier verschwiegener Gesellen. Der *Lexer* sucht Ihr Programm nach den kleinsten sinngebenden Einheiten ab. Diese werden manchmal als »Lexeme« bezeichnet; in Büchern über Programmiersprachen werden Sie aber häufiger den Begriff *Token* antreffen. Der Lexer wird manchmal als Tokenizer oder Scanner bezeichnet, und das, was er tut, nennt man manchmal entsprechend Lexing oder Tokenizing. Der *Parser* versucht dann, sich aus Gruppen dieser Tokens etwas sinnvolles zusammenzureimen, indem er sie, basierend auf der Grammatik von Perl, zu größeren Konstrukten wie etwa Ausdrücken und Anweisungen zusammensetzt. Der *Optimizer* reduziert diese größeren

3 Nein, es gibt kein formales Syntaxdiagramm wie BNF, aber Sie sind herzlich eingeladen, sich die Datei *perly.y* im Perl-Quellbaum anzusehen, die die von Perl verwendete yacc(1)-Grammatik enthält. Wir raten Ihnen davon ab, sich den Lexer anzusehen, von dem bekannt ist, daß er bei Leberkranken zu Freßstörungen geführt hat.

Gruppierungen und ordnet sie zu effizienteren Sequenzen an. Er wählt seine Optimierungen sorgfältig aus und verschwendet seine Zeit nicht mit marginalen Optimierungen, weil der Perl-Compiler verdammt schnell sein muß, wenn man ihn als Load-and-Go-Compiler verwendet.

Das läuft alles nicht in unabhängigen Schritten ab, sondern alles auf einmal mit sehr viel Informationsaustausch zwischen den Agenten. Der Lexer benötigt gelegentlich Hinweise vom Parser, um zu wissen, um welchen der verschiedenen möglichen Token-Typen es sich gerade handelt. (Seltsamerweise ist der lexikalische Geltungsbereich eines der Dinge, die die lexikalische Analyse *nicht* versteht, weil das die andere Bedeutung von »lexikalisch« darstellt.) Auch der Optimizer muß nachhalten, was der Parser so macht, weil einige Optimierungen nicht erfolgen können, solange das Parsing nicht einen bestimmten Punkt erreicht hat, wie etwa das Ende eines Ausdrucks, einer Anweisung, eines Blocks oder einer Subroutine.

Sie könnten nun denken, daß es merkwürdig ist, daß der Perl-Compiler all diese Dinge auf einmal und nicht nacheinander macht. Aber eigentlich ist das der gleiche schwierige Prozeß, den Sie durchlaufen, um eine natürliche Sprache zu verstehen, während Sie sie hören oder lesen. Auch Sie warten nicht bis zum Ende des Kapitels, um herauszubekommen, was der erste Satz bedeutet hat. Sie können sich die folgenden Entsprechungen vor Augen halten:

Computersprache	Natürliche Sprache
Zeichen	Buchstabe
Token	Morphem
Term	Wort
Ausdruck	Satzteil
Anweisung	Satz
Block	Absatz
Datei	Kapitel
Programm	Erzählung

Wenn das Parsing erfolgreich verlaufen ist, hält der Compiler Ihre Eingabe für eine gültige Erzählung, äh, ein gültiges Programm. Wenn Sie bei der Ausführung des Programms den Switch **`-c`** verwenden, wird die Meldung »syntax OK« ausgegeben und die Ausführung endet. Anderenfalls übergibt der Compiler die Früchte seiner Bemühungen an die anderen Agenten. Diese »Früchte« werden in Form eines Parsing-Baums oder *Parse-Tree* geliefert. Jede Frucht des Baums – jeder *Knoten* (node), wie man ihn nennt – repräsentiert einen der internen Perl-*Opcodes*, während die Verzweigungen im Baum das historische Wachstumsmuster des Baums darstellen. Schließlich werden die Knoten linear, einer nach dem anderen aneinandergereiht, um die Ausführungsreihenfolge festzulegen, in der das Laufzeitsystem die Knoten besucht.

Jeder Opcode bildet die kleinste einigermaßen autonome Anweisungen, die Perl kennt. Sie sehen einen Ausdruck wie `$a = -($b + $c)` vielleicht als eine Anweisung an, aber für Perl

sind das sechs verschiedene Opcodes. In einem etwas vereinfachten Format dargestellt, würde der Parse-Tree für diesen Ausdruck etwa so aussehen wie in Abbildung 18-2. Die Zahlen repräsentieren dabei die Reihenfolge des Besuchs, der das Laufzeitsystem von Perl folgen würde.

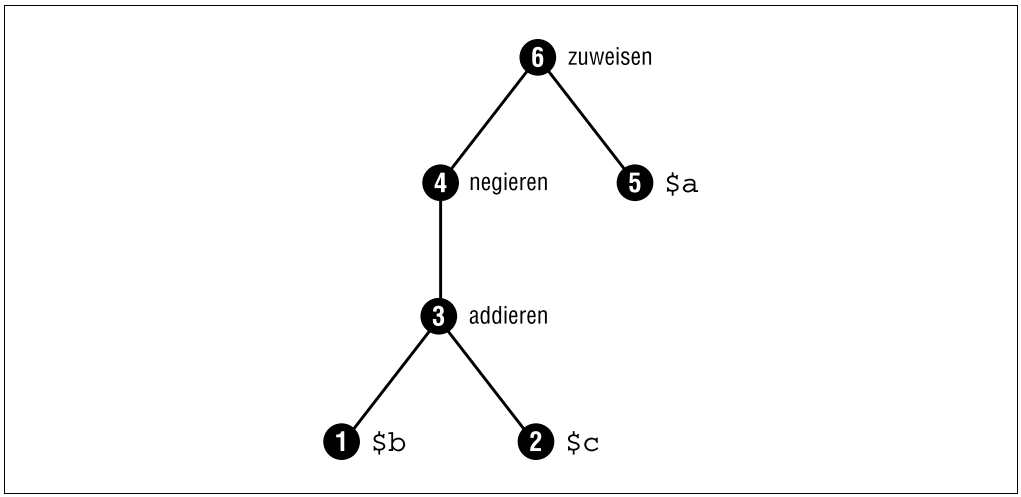

Abbildung 18-2: Opcode-Besuchsfolge bei $a = –($b + $c)

Perl ist kein One-Pass-Compiler, wie Sie vielleicht glauben. (One-Pass-Compiler machen die Dinge für den Computer leicht und für den Programmierer schwierig.) Das heißt, daß der Compiler den Quelltext nicht linear liest und übersetzt, sondern nach Bedarf vor und zurück springt, um einzelne Teile in mehreren Phasen (»Pass«) weiter- oder nachzubearbeiten. Genauer gesagt handelt es sich bei Perl um einen optimierenden Multipass-Compiler, der aus mindestens drei logischen Phasen besteht, die in der Praxis alle untereinander verknüpft sind. Die Phasen 1 und 2 laufen abwechselnd ab, während der Compiler den Parse-Tree während seiner Konstruktion wiederholt auf und ab durchläuft. Der dritte Durchlauf erfolgt, wenn der Parser eine Subroutine oder eine Datei vollständig abgearbeitet hat. Es gibt folgende Phasen:

Phase 1: Bottom-Up-Parsing

Während dieser Phase wird der Parse-Tree mit Hilfe des *yacc*(1)-Parsers aufgebaut. Die Tokens werden vom darunterliegenden Lexer geliefert (den man auch als eigene logische Phase betrachten könnte). Bottom-Up bedeutet einfach, daß der Parser die Blätter des Baums kennt, bevor er dessen Verzweigungen und die Wurzel sieht. Die Dinge werden dabei in Abbildung 18-2 wirklich von unten (bottom) nach oben erkannt, weil wir die Wurzel in der für Informatiker (und Linguisten) typischen Art nach oben gestellt haben.

Während jeder Opcode-Knoten erzeugt wird, stellen entsprechende Prüfungen die korrekte Semantik sicher, beispielsweise die korrekte Anzahl und Art der Argumente beim Aufruf eingebauter Funktionen. Während ein Teilbereich des Baums Gestalt annimmt, überlegt der Optimizer, welche Transformationen er auf den vor ihm liegenden Teilbaum anwenden kann. Beispielsweise kann er, sobald er weiß, daß eine Liste von Werten an eine Funktion mit einer festen Anzahl von Argumenten übergeben wird, den Opcode entfernen, der die Anzahl von Argumenten für Funktionen mit einer variierenden Anzahl von Argumenten enthält. Eine wichtigere Optimierung, das sogenannte *Constant Folding*, wird später in diesem Abschnitt beschrieben.

Dieser Durchgang konstruiert auch die Reihenfolge, in der die Knoten später während der Ausführung besucht werden, was wirklich ein netter Trick ist, weil der erste zu besuchende Punkt fast nie der oberste Knoten ist. Der Compiler erzeugt eine temporäre Schleife von Opcodes, wobei der oberste Knoten auf den ersten zu besuchenden Opcode verweist. Wird der oberste Opcode in etwas Größeres eingebettet, wird diese Opcode-Schleife aufgebrochen, und eine größere Schleife mit einem neuen obersten Knoten wird generiert. Schließlich wird die Schleife endgültig aufgebrochen, wenn der Start-Opcode in eine andere Struktur wie etwa einen Subroutinen-Deskriptor geschrieben wird. Der Subroutinen-Aufrufer findet immer noch diesen ersten Opcode, obwohl er so weit unten im Baum liegt, wie das auch in Abbildung 18-2 der Fall ist. Für den Interpreter besteht keine Notwendigkeit, sich im Parse-Tree nach unten zu bewegen, um den Startpunkt zu ermitteln.

Phase 2: Top-Down-Optimizer

Wenn jemand ein Perl-Codefragement liest, kann er den Kontext nicht bestimmen, ohne die darumliegenden lexikalischen Elemente zu untersuchen. Manchmal kann man nicht entscheiden, was wirklich vorgeht, ehe man nicht mehr Informationen besitzt. Nehmen Sie es nicht zu schwer, denn Sie sind nicht allein: Auch der Compiler kann das nicht. In dieser Phase geht der Compiler den gerade aufgebauten Teilbaum noch einmal von oben nach unten durch, um einige lokale Optimierungen anzuwenden. Die erwähnenswerteste ist dabei die *Kontext-Propagation*. Der Compiler markiert die vorhandenen Knoten basierend auf dem aktuellen Knoten mit dem entsprechenden Kontext (void, Skalar, Liste, Referenz oder Lvalue). Ungewollte Opcodes werden jetzt »ausgenullt«, aber nicht gelöscht, weil es nun zu spät ist, die Ausführungsreihenfolge umzustellen. Wir überlassen es der dritten Phase, sie aus der provisorischen Ausführungsreihenfolge zu entfernen, die vom ersten Durchgang ermittelt wurde.

Phase 3: Peephole-Optimizer

Bestimmte Codeeinheiten besitzen ihren eigenen Speicherraum, in dem sie Variablen mit lexikalischem Geltungsbereich vorhalten. (Dieser Raum wird bei Perl als *Scratchpad*, zu deutsch etwa Schmierzettel, bezeichnet.) Zu diesen Einheiten gehören `eval` *`STRING`*`s`, Subroutinen und ganze Dateien. Was aus der Sicht des Optimizers aber noch wichtiger ist: Sie besitzen alle ihren eigenen Einstiegspunkt. Wir kennen also die Ausführungsreihenfolge von diesem Punkt an, wissen aber nicht, was vorher passiert ist, weil das Konstrukt von überall hätte aufgerufen werden können. Wenn also das Parsing einer solchen Einheit erfolgt ist, läßt Perl einen Peephole-Optimizer über diesen Code laufen. Im Gegensatz zu den beiden vorherigen Durchgängen, die

der Aststruktur des Parse-Tree gefolgt sind, geht dieser Durchgang den Code in linearer Ausführungsreihenfolge durch, weil das grundsätzlich die letzte Möglichkeit ist, das zu tun, bevor die Opcode-Liste vom Parser abgekoppelt wird. Die meisten Optimierungen sind bereits in den ersten beiden Durchgängen vorgenommen worden, aber bei einigen war das nicht möglich.

Verschiedene späte Optimierungen finden hier statt, darunter auch die Zusammenstellung der endgültigen Ausführungsreihenfolge durch das Überspringen ausgenullter Opcodes und die Erkennung verschiedener Opcode-Kombinationen, die auf etwas Einfacheres reduziert werden können. Die Erkennung aneinandergereihter Stringverkettungen ist eine wichtige Optimierung, weil man es möglichst vermeiden will, einen String hin- und herzukopieren, nur weil ein bißchen am Ende angehängt wird. Diese Phase optimiert nicht nur, sie erledigt auch einen Großteil der eigentlichen Arbeit: das Abfangen von Barewords, die Generierung von Warnungen bei fragwürdigen Konstrukten, die Überprüfung von nicht zu erreichendem Code, die Auslösung von Pseudohash-Schlüsseln und die Suche nach Subroutinen, die aufgerufen werden, bevor ihre Prototypen kompiliert wurden.

Phase 4: Codegenerierung

Dieser Durchgang ist optional und wird beim normalen Lauf der Dinge nicht genutzt. Wird aber einer der drei Codegeneratoren – `B::Bytecode`, `B::C` oder `B::CC` – aufgerufen, wird der Parse-Tree ein letztes Mal angefaßt. Die Codegeneratoren liefern entweder serialisierten Perl-Bytecode, der zur späteren Rekonstruktion des Parse-Tree verwendet wird, oder literalen C-Code, der den Zustand des kompilierten Parse-Tree repräsentiert.

Die Generierung von C-Code erfolgt in zwei verschiedenen Varianten. `B::C` rekonstruiert einfach den Parsing-Baum und führt ihn in der üblichen `runops()`-Schleife aus, die Perl selbst während der Ausführung verwendet. `B::CC` erzeugt ein linearisiertes und optimiertes C-Äquivalent des Laufzeit-Codepfads (der einer riesigen Sprungtabelle ähnelt) und führt dieses aus.

Während der Kompilierung optimiert Perl Ihren Code auf die verschiedensten Weisen. Es arrangiert den Code um, damit er während der Ausführung effizienter ist. Es löscht Code, der während der Ausführung nie erreicht werden kann, etwa `if (0)`-Blöcke oder die `elsif` und das `else` in einem `if (1)`-Block. Wenn Sie lexikalische Variablen mit `my ClassName $var` oder `our ClassName $var` deklarieren und das Paket `ClassName` mit dem Pragma `use fields` arbeitet, werden Zugriffe auf konstante Felder aus dem zugrundeliegenden Pseudohash während der Compilierung auf ihre Schreibweise geprüft und in lineare Arrayzugriffe umgewandelt. Wenn Sie den `sort`-Operator mit einer ausreichend einfachen Vergleichsroutine wie `{$a <=> $b}` oder `{$b cmp $a}` versorgen, wird diese durch einen Aufruf kompilierten C-Codes ersetzt.

Die wohl dramatischste Optimierung ist aber die Art und Weise, in der Perl konstante Ausdrücke so schnell wie möglich auflöst. Betrachten wir zum Beispiel den Parse-Tree in Abbildung 18-2. Wären die Knoten 1 und 2 literale Werte oder konstante Funktionen, wären die Knoten 1 bis 4 durch das Ergebnis dieser Berechnung ersetzt worden, beispielsweise wie in Abbildung 18-3.

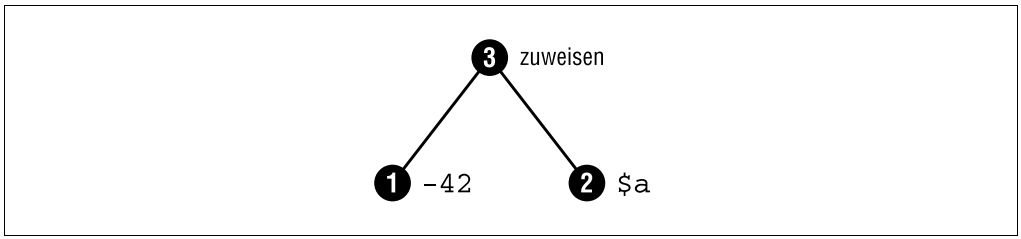

Abbildung 18-3: Constant Folding

Das wird als *Constant Folding* bezeichnet. Das Constant Folding beschränkt sich nicht auf einfache Fälle wie die Umwandlung von `2**10` während der Kompilierung in `1024`. Sie löst auch Funktionsaufrufe aus – eingebaute und benutzerdefinierte Subroutinen, die die Kriterien aus dem Abschnitt »Inlining konstanter Funktionen« in Kapitel 6, *Subroutinen*, erfüllen. So wie der FORTRAN-Compiler bekanntermaßen um seine ihm innewohnenden Funktionen weiß, weiß auch ein Perl-Compiler, welche seiner eingebauten Funktionen er während der Kompilierung aufrufen muß. Aus diesem Grund erhalten Sie bei `log 0.0` oder dem `sqrt` einer negativen Konstante einen Fehler während der Kompilierung und keinen Laufzeitfehler, und der Interpreter wird gar nicht erst aufgerufen.[4]

Selbst beliebig komplizierte Ausdrücke werden früh aufgelöst und führen zur Löschung kompletter Blöcke wie des folgenden:

```
if (2 * sin(1)/cos(1) < 3 && somefn()) { whatever() }
```

Es wird kein Code für etwas generiert, was niemals evaluiert werden kann. Weil der erste Teil immer falsch ist, werden weder `somefn` noch `whatever` jemals aufgerufen. (Erwarten Sie also nicht, mittels `goto`-Labels in den Block um `whatever` springen zu können, weil diese zur Laufzeit gar nicht existieren.) Wenn `somefn` eine Inlining-fähige konstante Funktion wäre, würde auch der folgende Wechsel der Evaluierungsreihenfolge

```
if (somefn() && 2 * sin(1)/cos(1) < 3)) { whatever() }
```

nichts am Ergebnis ändern, weil der gesamte Ausdruck immer noch während der Kompilierung aufgelöst wird. Wäre `whatever` Inlining-fähig, würde es weder zur Laufzeit noch während der Kompilierung aufgerufen werden. Ihr Wert würde einfach eingefügt werden, als handelte es sich um eine literale Konstante. Sie würden nur eine Warnung über die »Unnütze Verwendung einer Konstante im void-Kontext« (»Useless use of a constant in void context«) erhalten. Das könnte einen überraschen, wenn man nicht erkennt, daß es sich um eine Konstante handelt. Ist `whatever` hingegen die letzte Anweisung, die in einer Funktion in einem Nicht-void-Kontext evaluiert wird (zumindest in den Augen des Optimizers), erscheint keine Warnung.

4 Tatsächlich vereinfachen wir die Dinge hier sehr stark. Der Interpreter wird aufgerufen, weil das Constant Folding auf diese Weise implementiert ist. Er wird aber, ähnlich wie bei BEGIN-Blöcken, während der Kompilierung aufgerufen.

Sie können sich das endgültige Ergebnis des erzeugten Parse-Tree nach allen Optimierungsphasen mit *perl -Dx* ansehen. (Der Switch **-D** verlangt eine spezielle, Debugging-fähige Perl-Version.) Beachten Sie auch den noch folgenden Abschnitt über `B::Deparse`.

Alles in allem arbeitet der Perl-Compiler hart (aber nicht *zu* hart) an der Optimierung des Codes, damit die Ausführung zur Laufzeit zügig erfolgt. Es wird Zeit, Ihr Programm laufen zu lassen, was wir nun tun wollen.

Ihren Code ausführen

Sparc-Programme laufen nur auf Sparc-Maschinen, Intel-Programme laufen nur auf Intel-Maschinen, und Perl-Programme laufen nur auf Perl-Maschinen. Eine Perl-Maschine besitzt all die Attribute, die ein Perl-Programm an einem Computer ideal finden würde: Speicher, der automatisch alloziert und dealloziert wird, fundamentale Datentypen, die dynamische Strings, Arrays und Hashes sind und keinerlei Größenbeschränkungen unterworfen sind, und Systeme, die sich alle auf die gleiche Weise verhalten. Die Aufgabe des Perl-Interpreters besteht nun darin, jeden Computer, auf dem er läuft, so erscheinen zu lassen, als wäre er eine dieser idealisierten Perl-Maschinen.

Diese fiktive Maschine präsentiert die Illusion eines Computers, der speziell entworfen wurde, um nichts anderes als Perl-Programme auszuführen. Jeder vom Compiler erzeugte Opcode ist ein elementarer Befehl in diesem emulierten Befehlssatz. Anstelle eines Hardware-Programmzählers hält der Interpreter nur den momentan auszuführenden Opcode nach. Anstelle eines Hardware-Stackpointers besitzt der Interpreter seinen eigenen virtuellen Stack. Dieser Stack ist sehr wichtig, weil die virtuelle Perl-Maschine (wir lehnen es ab, sie als PVM zu bezeichnen) eine stackbasierte Maschine ist. Perl-Opcodes werden intern als *PP-Codes* (eine Abkürzung für »Push-Pop-Codes«) bezeichnet, weil sie den virtuellen Stack des Interpreters manipulieren, um alle Operanden zu finden, temporäre Werte zu verarbeiten und die Ergebnisse abzulegen.

Wenn Sie jemals in Forth oder PostScript programmiert oder einen HP-Taschenrechner mit RPN-Eingabe (»Reverse Polish Notation«, umgekehrte polnische Notation) verwendet haben, dann wissen Sie, wie eine Stackmaschine funktioniert. Selbst wenn nicht, das Konzept ist einfach: Um die Werte 3 und 4 zu addieren, gehen Sie in der Reihenfolge `3 4 +` vor, anstatt wie sonst meist üblich `3 + 4` zu schreiben. Aus der Sicht des Stacks bedeutet das, daß Sie zuerst den Wert `3` und dann den Wert `4` auf den Stack schieben (Push-Operation). Der Operator `+` entfernt dann diese beiden Argumente vom Stack (Pop-Operation), addiert sie und schiebt dann den Wert `7` zurück auf den Stack, wo er so lange liegen bleibt, bis Sie etwas anderes mit ihm machen.

Verglichen mit dem Perl-Compiler ist der Perl-Interpreter ein geradliniges, geradezu langweiliges Programm. Er geht einfach schrittweise die kompilierten Opcodes durch und übergibt sie an die Perl-Laufzeitumgebung, das heißt, an die virtuelle Perl-Maschine. Der Interpreter ist nur ein Haufen C-Code, richtig?

Tatsächlich ist er überhaupt nicht langweilig. Eine virtuelle Perl-Maschine überwacht für Sie einen Großteil des dynamischen Kontextes, damit Sie sich nicht darum kümmern müssen. Perl pflegt eine ganze Reihe von Stacks, die Sie nicht alle verstehen müssen, die wir an dieser Stelle aber trotzdem aufzählen, nur um Sie zu beeindrucken:

Operanden-Stack
: Über diesen Stack haben wir bereits gesprochen.

Save-Stack
: Hier werden lokalisierte Werte für die spätere Wiederherstellung gesichert. Auch viele interne Routinen lokalisieren Werte, ohne daß Sie es wissen.

Scope-Stack
: Der leichtgewichtige dynamische Kontext, der bestimmt, wann ein Element vom Save-Stack entfernt werden soll.

Kontext-Stack
: Der schwergewichtige dynamische Kontext; wer hat wen aufgerufen, um dorthin zu kommen, wo Sie jetzt sind. Die Funktion `caller` untersucht diesen Stack. Schleifenkontrollfunktionen untersuchen diesen Stack, um herauszufinden, welche Schleife es zu kontrollieren gilt. Wenn Sie sich im Kontext-Stack zurückbewegen, wird auch der Scope-Stack entsprechend korrigiert, was Ihre ganzen lokalen Variablen aus dem Save-Stack wiederherstellt, selbst wenn Sie den früheren Kontext durch ruchlose Methoden wie durch das Auslösen einer Ausnahme und ein *longjmp*(3) verlassen haben.

Jumpenv-Stack
: Der Stack mit *longjmp*(3)-Kontexten, der es uns erlaubt, Ausnahmen auszulösen und bei Bedarf das Programm zu verlassen.

Return-Stack
: Dieser Stack gibt an, wohin wir zurückspringen, wenn die Subroutine verlassen wird.

Mark-Stack
: Dieser Stack gibt an, an welcher Stelle die aktuelle Argumentenliste auf dem Operanden-Stack beginnt.

Rekursive lexikalische Hilfs-Stacks
: Dieser Stack gibt an, wo lexikalische Variablen und andere »Zwischenregister« festgehalten werden, wenn Subroutinen rekursiv aufgerufen werden.

Und natürlich gibt es den C-Stack, auf dem alle C-Variablen abgelegt werden. Perl versucht allerdings, sich für die Speicherung gesicherter Werte nicht auf den C-Stack zu verlassen, weil *longjmp*(3) die korrekte Wiederherstellung solcher Werte umgeht.

All das zeigen wir Ihnen nur, um zu verdeutlichen, daß die übliche Ansicht, ein Interpreter sei ein Programm, das ein anderes Programm interpretiert, wirklich völlig ungeeignet ist, um das zu beschreiben, was hier vorgeht. Ja, es gibt etwas C-Code, der einige Opcodes implementiert, aber wenn wir »Interpreter« sagen, meinen wir etwas mehr als das.

Das gleiche gilt für den »Musiker«, der weit mehr als eine DNA-Sequenz zur Umwandlung von Noten in Musik ist. Musiker sind reale, lebende Organismen und besitzen einen »Zustand«. So verhält es sich auch mit Interpretern.

Genaugenommen ist dieser dynamische und lexikalische Kontext zusammen mit den globalen Symboltabellen, den Parse-Trees und einem Ausführungsstrang das, was wir als Interpreter bezeichnen. Als Kontext für die Ausführung beginnt ein Interpreter sein Leben eigentlich schon, bevor der Compiler soweit ist. Er kann in rudimentärer Form schon laufen, während der Compiler den Interpreter-Kontext noch aufbaut. Tatsächlich ist das genau das, was passiert, wenn der Compiler den Interpreter aufruft, um solche Dinge wie `BEGIN`-Blöcke auszuführen. Und der Interpreter kann die Sache umkehren und den Compiler aufrufen, um die Dinge voranzutreiben. Jedesmal, wenn Sie eine weitere Subroutine aufrufen oder ein weiteres Modul laden, definiert sich die jeweilige virtuelle Perl-Maschine, die wir als Interpreter bezeichnen, neu. Sie können nicht sagen, daß der Compiler oder der Interpreter die Kontrolle besitzt, weil beide kooperieren, um den Startprozeß zu steuern, den wir allgemein als »Ausführung eines Perl-Skripts« bezeichnen. Es ist wie bei der Ausbildung des Gehirns eines Kindes. Ist die DNA verantwortlich oder die Neuronen? Wir glauben, daß ein wenig von beidem erforderlich ist – zusammen mit ein paar Eingaben von externen Programmierern.

Es ist möglich, mehrere Interpreter im gleichen Prozeß auszuführen. Diese können Parse-Trees gemeinsam nutzen (oder auch nicht), je nachdem, ob ein vorhandener Interpreter geklont oder vollständig neu aufgebaut wurde. Es ist auch möglich, mehrere Threads in einem einzigen Interpreter auszuführen. In diesem Fall werden nicht nur die Parse-Trees, sondern auch die globalen Symbole gemeinsam benutzt (siehe Kapitel 17, *Threads*).

Die meisten Perl-Programme verwenden aber nur einen einzigen Perl-Interpreter zur Ausführung des kompilierten Codes. Und obwohl Sie mehrere unabhängige Perl-Interpreter innerhalb eines Prozesses ausführen können, ist die hierfür zuständige aktuelle API nur über C erreichbar.[5] Jeder einzelne Interpreter spielt die Rolle eines vollständig separaten Prozesses, kostet aber nicht ganz so viel, wie die Erzeugung eines vollständig neues Prozesses. Auf diese Weise erreicht die Apache-Erweiterung `mod_perl` ihre hohe Performance: Wenn Sie ein CGI-Skript unter `mod_perl` starten, wurde es bereits in Perl-Opcodes übersetzt, was eine Rekompilierung überflüssig macht. Sie eliminieren aber auch – und das ist der eigentliche Flaschenhals – die Notwendigkeit, einen neuen Prozeß starten zu müssen. Apache initialisiert einen neuen Perl-Interpreter in einem existierenden Prozeß und übergibt diesem Interpreter den bereits kompilierten Code zur Ausführung. Natürlich hängt hier, wie immer im Leben, wesentlich mehr dran, als wir hier vorgeben. Weitere Informationen zu `mod_perl` finden Sie in *Writing Apache Modules with Perl and C* (O'Reilly, 1999).

Viele andere Anwendungen wie *nvi*, *vim* und *innd* können Perl-Interpreter einbetten (wir können sie hier nicht alle aufzählen). Es gibt auch eine ganze Reihe kommerzieller Produkte, die gar nicht erst erwähnen, daß sie Perl-Engines eingebettet haben. Sie verwenden sie nur intern, um ihre Aufgabe elegant erledigen zu können.

5 Mit bislang einer Ausnahme: Perl 5.6.0 kann Interpreter klonen, um fork unter Microsoft Windows zu emulieren. Wenn Sie diese Zeilen lesen, kann es durchaus schon eine Perl-API auf die sogenannten »Ithreads« geben.

Compiler-Backends

Wenn Apache also ein Perl-Programm jetzt kompilieren und später ausführen kann, warum nicht auch Sie? Apache und andere Programme mit eingebetteten Perl-Interpretern haben es leicht – sie speichern den Parse-Tree niemals in einer externen Datei. Wenn Sie mit diesem Ansatz zufrieden sind und nicht die C-API verwenden wollen, um etwas zu erreichen, können Sie das gleiche tun. Sehen Sie sich den Abschnitt »Perl einbetten« in Kapitel 21, *Internes und Externes*, an, um zu verstehen, wie Sie auf Perl aus einem umschließenden C-Rahmen heraus zugreifen können.

Wenn Sie diesen Weg nicht gehen wollen oder andere Bedürfnisse haben, stehen Ihnen ein paar Optionen zur Verfügung. Anstatt die Opcode-Ausgabe des Perl-Compilers direkt an den Perl-Interpreter zu übergeben, können Sie eines der verschiedenen alternativen Backends aufrufen. Diese Backends können die kompilierten Opcodes serialisieren und in einer externen Datei ablegen oder sie sogar in eine Reihe verschiedener C-Varianten umwandeln.

Bitte seien Sie sich dessen bewußt, daß die Codegeneratoren extrem experimentelle Utilities sind, auf die man sich in einer Produktionsumgebung nicht unbedingt verlassen sollte. Tatsächlich sollten Sie nicht einmal erwarten, daß sie alle Jubeljahre mal in einer Nicht-Produktionsumgebung funktionieren. Nachdem wir Ihre Erwartungen also so heruntergeschraubt haben, daß jeder Erfolg an sich schon eine Überraschung ist, können wir Ihnen beruhigt sagen, wie die Backends arbeiten.

Einige Backend-Module sind Codegeneratoren, beispielsweise `B::Bytecode`, `B::C` und `B::CC`. Andere sind in Wirklichkeit Codeanalyse- oder Debugging-Tools, zum Beispiel `B::Deparse`, `B::Lint` und `B::Xref`. Neben diesen Backends enthält das Standard-Release verschiedene andere Low-Level-Module, die für Autoren von Perl-Entwicklungswerkzeugen von Interesse sein könnten. Weitere Backend-Module sind im CPAN zu finden, darunter (während dies geschrieben wird) `B::Fathom`, `B::Graph`, `B::JVM::Jasmin` und `B::Size`.

Wenn Sie den Perl-Compiler für etwas anderes als die Übergabe an den Interpreter einsetzen, steht das `O`-Modul (d.h. die Verwendung der Datei *O.pm*) zwischen dem Compiler und den verschiedenen Backend-Modulen. Sie rufen das Backend nicht direkt auf, sondern rufen den mittleren Teil auf, der dann letztendlich das gewünschte Backend aufruft. Wenn Sie also ein Modul namens `B::Backend` aufrufen wollen, würden Sie es für ein bestimmtes Skript wie folgt starten:

```
% perl -MO=Backend SKRIPINAME
```

Einige Backends erkennen Optionen, die wie folgt übergeben werden:

```
% perl -MO=Backend,OPTIONEN SKRIPINAME
```

Einige Backends besitzen bereits eigene Frontends zum richtigen Aufruf des Mittelteils, so daß Sie sich deren MOs nicht merken müssen. Besonders erwähnenswert ist hier *perlcc*(1), das den entsprechenden Code-Generator aufstößt, dessen Start anderenfalls etwas beschwerlich sein kann.

Codegeneratoren

Es sei noch einmal ausdrücklich betont, daß die drei aktuellen Backends, die Perl-Opcodes in irgendein anderes Format umwandeln, experimenteller Natur sind. (Ja, wir haben das bereits erwähnt, aber wir wollen nicht, daß Sie das vergessen.) Selbst wenn die erzeugte Ausgabe korrekt läuft, können die resultierenden Programme mehr Plattenplatz, mehr Speicher und mehr CPU-Zeit verbrauchen, als das normalerweise der Fall wäre. Das ist ein Bereich fortwährender Forschung und Entwicklung. Die Dinge werden sich bessern.

Der Bytecode-Generator

Das Modul `B::Bytecode` schreibt die Opcodes der Parsing-Bäume in einer plattformunabhängigen Form heraus. Sie können ein auf Bytecodes reduziertes Perl-Skript nehmen und auf jede andere Maschine kopieren, auf der Perl installiert ist.

Der standardmäßig verfügbare, aber immer noch experimentelle Befehl *perlcc*(1) weiß, wie man Perl-Quellcode in ein bytekompiliertes Perl-Programm konvertiert. Sie müssen nur folgendes angeben:

```
% perlcc -b -o pbyscript srcscript
```

Und schon sollten Sie in der Lage sein, das resultierende *pbyscript* »auszuführen«. Der Anfang dieser Datei sieht ungefähr wie folgt aus:

```
#!/usr/bin/perl
use ByteLoader 0.03;
^C^@^E^A^C^@^@^@^A^F^@^C^@^@^@^B^F^@^C^@^@^@^C^F^@^C^@^@^@
B^@^@^@^H9^A8M-^?M-^?M-^?M-^?7M-^?M-^?M-^?M-^?6^@^@^@^A6^@
^G^D^D^@^@^@^KR^@^@^@^HS^@^@^@^HV^@M-2W<^FU^@^@^@^@X^Y@Z^@
...
```

Sie sehen einen kleinen Skript-Header, auf den reine Binärdaten folgen. Das hat etwas von schwarzer Magie, ist aber bestenfalls ein wenig »grau«. Das `ByteLoader`-Modul verwendet eine als *Sourcefilter* bezeichnete Technik, um den Code umzuwandeln, bevor Perl ihn überhaupt zu Gesicht bekommt. Ein Sourcefilter ist eine Art Präprozessor, der auf alles in der Datei Folgende angewandt wird. Anstatt auf einfache Transformationen von Makroprozessoren wie *cpp*(1) und *m4*(1) beschränkt zu sein, gibt es hier keinerlei Beschränkungen. Sourcefilter wurden verwendet, um die Perl-Syntax zu erweitern, um Quellcode zu komprimieren und zu verschlüsseln, und sogar, um Perl-Programme in Latein zu schreiben. E perlibus unicode; cogito, ergo substr; carp dbm, et al.

Das `ByteLoader`-Modul ist ein Sourcefilter, der weiß, wie die serialisierten Opcodes von `B::Bytecode` zu entschlüsseln sind, um den ursprünglichen Parse-Tree wiederherzustellen. Der so wiederhergestellte Perl-Code wird ohne Zutun des Compilers in den aktuellen Parse-Tree eingespeist. Trifft der Interpreter auf diese Opcodes, führt er sie aus, als würden sie schon die ganze Zeit nur darauf warten.

Die C-Codegeneratoren

Die restlichen Codegeneratoren `B::C` und `B::CC` erzeugen beide C-Code anstelle serialisierter Perl-Opcodes. Der von ihnen generierte Code ist alles andere als lesbar, und wenn Sie es doch versuchen, laufen Sie Gefahr zu erblinden. Das ist nichts, um kleine, von Perl nach C übersetzte Codefragmente in größere C-Programme einzubinden. Wenn Sie das wollen, lesen Sie Kapitel 21.

Das Modul `B::C` schreibt einfach alle C-Datenstrukturen heraus, die zum Aufbau der gesamten Perl-Laufzeitumgebung notwendig sind. Sie erhalten einen dezidierten Interpreter, bei dem alle compilergenerierten Datenstrukturen vorinitialisiert sind. In gewisser Weise ist der generierte Code dem von `B::Bytecode` ähnlich. Beide erzeugen eine direkte Übersetzung der vom Compiler aufgebauten Opcode-Bäume, aber während `B::Bytecode` diese in eine symbolische Form packt, die später wiederhergestellt und in einen laufenden Perl-Interpreter eingebunden wird, schreibt `B::C` die Opcodes in C herunter. Wenn Sie diesen C-Code mit Ihrem C-Compiler kompilieren und mit der Perl-Bibliothek linken, benötigt das resultierende Programm keinen Perl-Interpreter auf dem Zielsystem. (Es könnte allerdings einige Shared Libraries benötigen, wenn Sie nicht alles statisch gelinkt haben.) Tatsächlich unterscheidet sich das Programm aber nicht von einem regulären Perl-Interpreter, der Ihr Skript ausführt. Es wurde nur in ein eigenständiges ausführbares Image gepackt.

Das Modul `B::CC` versucht hingegen, etwas mehr zu tun. Der Anfang des von ihm erzeugten C-Codes ähnelt stark dem von `B::C` erzeugten Code,[6] aber diese Ähnlichkeiten enden recht schnell. Im `B::C`-Code haben Sie eine große Opcode-Tabelle in C, die so manipuliert wird, wie das auch der Interpreter tun würde. Der von `B::CC` generierte C-Code ist hingegen so ausgelegt, daß er den Laufzeitfluß Ihres Programms abbildet. Es gibt sogar eine C-Funktion für jede Funktion Ihres Programms. Einige Optimierungen erfolgen auf Grundlage des Variablentyps. Einige Benchmarks laufen doppelt so schnell wie beim Standardinterpreter. Das ist der ehrgeizigste der aktuellen Codegeneratoren, der für die Zukunft am meisten verspricht. Aber ohne Frage ist er auch der instabilste von allen.

Informatik-Studenten auf der Suche nach einer Diplomarbeit können die Suche einstellen. Auf diesem weiten Feld gibt es einige Diamanten, die darauf warten, geschliffen zu werden.

Code-Entwicklungstools

Das `O`-Modul bietet außer dem Füttern der ärgerlich experimentellen Codegeneratoren noch viele weitere Betriebsmodi. Da es einen vergleichsweise einfachen Zugriff auf die Ausgabe des Perl-Compilers bietet, ist es mit Hilfe dieses Moduls möglich, auf relativ einfache Art und Weise andere Tools zu entwickeln, die alles über ein Perl-Programm wissen müssen.

6 Andererseits sieht alles gleich aus, wenn man erblindet ist. Haben wir Sie nicht davor gewarnt zu gucken?

Das `B::Lint`-Modul ist nach *lint*(1) benannt, einem Programm zur Prüfung von C-Programmen. Es untersucht Programme auf fragwürdige Konstrukte, über die Anfänger häufig stolpern, die aber normalweise keine Warnungen auslösen. Sie rufen das Modul direkt auf:

```
% perl -MO=Lint,all myprog
```

Momentan sind nur einige wenige Prüfungen definiert, etwa die Verwendung eines Arrays in einem implizit skalaren Kontext, das Vertrauen auf Standardvariablen und der Zugriff auf (normalerweise private) Bezeichner eines anderen Pakets, die mit _ beginnen. Details finden Sie in *B::Lint*(3).

Das Modul `B::Xref` erzeugt Crossreferenzen der Deklaration und Verwendung aller Variablen (mit globalem und lexikalischem Geltungsbereich), Subroutinen und Formate eines Programms, geordnet nach Datei und Subroutine. Das Modul wird wie folgt aufgerufen:

```
% perl -MO=Xref myprog > myprof.pxref
```

Hier ein Ausschnitt aus einem solchen Report:

```
Subroutine parse_argv
  Package (lexical)
    $on               i113, 114
    $opt              i113, 114
    %getopt_cfg       i107, 113
    @cfg_args         i112, 114, 116, 116
  Package Getopt::Long
    $ignorecase       101
    &GetOptions       &124
  Package main
    $Options          123, 124, 141, 150, 165, 169
    %$Options         141, 150, 165, 169
    &check_read       &167
    @ARGV             121, 157, 157, 162, 166, 166
```

Sie sehen, daß die Subroutine `parse_argv` vier eigene lexikalische Variablen besitzt. Sie greift auch auf globale Identifier aus dem `main`-Paket und aus `Getopt::Long` zu. Die Zahlen geben die Zeilen an, in denen das Element verwendet wurde. Ein voranstehendes `i` gibt an, daß das Element an der folgenden Zeilennummer eingeführt (introduced) wurde. Ein führendes `&`-Zeichen bedeutet, daß eine Subroutine hier aufgerufen wurde. Dereferenzierungen werden separat aufgeführt, weshalb hier sowohl `$Options` als auch `%$Options` aufgeführt sind.

`B::Deparse` ist ein »Pretty Printer«, der Perl-Code demystifizieren kann und Ihnen zu verstehen hilft, welche Transformationen der Optimizer an Ihrem Code vorgenommen hat. Das folgende Beispiel zeigt beispielsweise, welche Standards Perl für verschiedene Konstrukte verwendet:

```
% perl -MO=Deparse -ne 'for (1 .. 10) { print if -t }'
LINE: while (defined($_ = <ARGV>)) {
    foreach $_ (1 .. 10) {
        print $_ if -t STDIN;
    }
}
```

Der Switch **-p** fügt Klammern ein, so daß Sie erkennen können, welche Vorstellung Perl von Vorrang besitzt:

```
% perl -MO=Deparse,-p -e 'print $a ** 3 + sqrt(2) / 10 ** -2 ** $c'
print(((($a ** 3) + (1.4142135623731 / (10 ** (-(2 ** $c))))));
```

Sie können **-q** verwenden, um zu sehen, in welche Primitive interpolierte Strings umgewandelt werden:

```
% perl -MO=Deparse,-q -e '"A $name and some @ARGV\n"'
'A ' . $name . ' and some ' . join($", @ARGV) . "\n";
```

Und hier können Sie sehen, wie Perl eine dreiteilige `for`-Schleife tatsächlich in eine `while`-Schleife umwandelt:

```
% perl -MO=Deparse -e 'for ($i=0;$i<10;$i++) { $x++ }'
$i = 0;
while ($i < 10) {
    ++$x;
}
continue {
    ++$i
}
```

Sie können `B::Deparse` sogar auf eine Perl-Bytecode-Datei anwenden, die mit *perlcc -b* erzeugt wurde und sich die Binärdatei auf diese Weise dekompilieren lassen. Serialisierte Perl-Opcodes sind sicher schwierig zu lesen, stark verschlüsselt sind sie aber nicht.

Avantgarde-Compiler, Retro-Interpreter

Es gibt die richtige Zeit, um über alles nachzudenken. Manchmal liegt diese Zeit vor uns, manchmal liegt sie hinter uns, und manchmal liegt sie irgendwo in der Mitte. Perl stellt keine Vermutungen darüber an, wann die Zeit zum Nachdenken gekommen ist, und gibt daher dem Programmierer eine Reihe von Optionen an die Hand, um ihm mitzuteilen, wann es denken soll. Zu anderen Zeiten weiß Perl, daß es denken sollte, weiß aber nicht genau, was es denken soll, und benötigt daher Möglichkeiten, Ihr Programm danach zu fragen. Ihr Programm beantwortet diese Fragen, indem es Subroutinen definiert, deren Namen dem entsprechen, was Perl herauszufinden versucht.

Nicht nur der Compiler kann den Interpreter aufrufen, wenn er ein wenig vorausdenken möchte, auch der Interpreter kann den Compiler aufrufen, wenn er den Lauf der Geschichte umkehren möchte. Ihr Programm kann unterschiedliche Operatoren verwenden, um wieder in den Compiler zu gelangen. Wie der Compiler kann auch der Interpreter benannte Subroutinen aufrufen, wenn er bestimmte Dinge herausfinden will. Aufgrund dieses ganzen Gebens und Nehmens zwischen Compiler, Interpreter und Ihrem Programm müssen Sie wissen, welche Dinge wann geschehen. Zuerst wollen wir darüber reden, wann diese Subroutinen angestoßen werden.

In Kapitel 10, *Pakete*, haben Sie gesehen, wie die `AUTOLOAD`-Subroutine eines Pakets angestoßen wird, wenn eine in diesem Paket undefinierte Funktion aufgerufen wird. In

Kapitel 12, *Objekte*, sind Sie der DESTROY-Methode begegnet, die immer dann aufgerufen wird, wenn der Speicher des Objekts von Perl automatisch wieder zurückgewonnen wird. Und in Kapitel 14, *Variablen und tie*, haben wir die vielen Funktionen vorgestellt, die implizit aufgerufen werden, wenn auf eine über tie gebundene Variable zugegriffen wird.

Diese Subroutinen folgen alle der Konvention, daß wir, wenn eine Subroutine automatisch vom Compiler oder vom Interpreter angestoßen wird, ihren Namen in Großbuchstaben schreiben. Gekoppelt mit den verschiedenen Lebensphasen Ihres Programms gibt es vier weitere dieser Subroutinen namens BEGIN, CHECK, INIT und END. Das Schlüsselwort sub vor deren Deklaration ist optional. Vielleicht sollte man sie besser als »Blöcke« bezeichnen, weil sie sich in mancher Hinsicht eher wie benannte Blöcke denn als echte Subroutinen verhalten.

Zum Beispiel gibt es, im Gegensatz zu regulären Subroutinen, keine Probleme, diese Blöcke mehrmals zu definieren, weil Perl darüber wacht, wann sie aufzurufen sind, und Sie sie daher nie namentlich aufrufen müssen. (Sie unterscheiden sich von regulären Subroutinen auch dadurch, daß shift und pop sich so verhalten, als wären sie im Hauptprogramm, und arbeiten daher standardmäßig mit @ARGV und nicht mit @_.)

Diese vier Blocktypen werden wie folgt abgearbeitet:

BEGIN
: Wird ASAP (»as soon as parsed«, also sofort nach dem Parsing) ausgeführt, sobald sie während der Kompilierung erkannt wird, d.h. bevor der Rest der Datei kompiliert wird.

CHECK
: Wird nach beendeter Kompilierung, aber vor dem Programmstart ausgeführt. (CHECK kann »Checkpoint« oder »Gegenprobe« oder auch einfach »Stop« bedeuten.)

INIT
: Läuft zu Beginn der Ausführung, direkt vor dem Hauptfluß Ihres Programms.

END
: Läuft am Ende der Ausführung, direkt nachdem das Hauptprogramm abgeschlossen wurde.

Wenn Sie mehr als eine dieser Routinen mit demselben Namen deklarieren, werden sie dennoch in einer geordneten Reihenfolge aufgerufen. Das trifft selbst dann zu, wenn sie in unterschiedlichen Modulen stehen. Stets gilt: Erst werden alle BEGIN-Blöcke ausgeführt, dann alle CHECK-Blöcke, dann die INIT-Blöcke und ganz am Schluß des Programms die END-Blöcke. Mehrere BEGINs und INITs werden in der Reihenfolge Ihrer Deklaration (FIFO) abgearbeitet, während CHECKs und ENDs in umgekehrter Deklarationsreihenfolge (LIFO) abgearbeitet werden.

Am einfachsten ist das wohl an einem Beispiel zu erkennen:

```
#!/usr/bin/perl -l
print       "main fängt hier an";
die         "main endet hier\n";
die         "XXX: nie erreicht\n";
END         { print "1. END: Interpreter beendet" }
CHECK       { print "1. CHECK: Kompilierung beendet" }
```

```
INIT        { print "1. INIT: Lauf gestartet"  }
END         { print "2. END: Lauf beendet"   }
BEGIN       { print "1. BEGIN: Kompilierung aktiv" }
INIT        { print "2. INIT: Lauf gestartet"  }
BEGIN       { print "2. BEGIN: Kompilierung aktiv" }
CHECK       { print "2. CHECK: Kompilierung beendet" }
END         { print "3. END: Lauf beendet"   }
```

Wenn wir es ausführen, erzeugt dieses Demoprogramm die folgende Ausgabe:

```
1. BEGIN: Kompilierung aktiv
2. BEGIN: Kompilierung aktiv
2. CHECK: Kompilierung beendet
1. CHECK: Kompilierung beendet
1. INIT: Lauf gestartet
2. INIT: Lauf gestartet
main fängt an
main endet hier
3. END: Lauf beendet
2. END: Lauf beendet
1. END: Lauf beendet
```

Weil ein BEGIN-Block sofort ausgeführt wird, kann er Subroutinendeklarationen, Definitionen und Importe einbinden, bevor der Rest der Datei überhaupt kompiliert wird. Das kann die Weise verändern, wie das Parsing des Compilers den Rest der aktuellen Datei verarbeitet, insbesondere wenn Sie Subroutinendefinitionen importieren. Zumindest erlaubt die Deklaration einer Subroutine deren Verwendung als Listenoperator, was die Klammerung optional macht. Wird die importierte Subroutine mit einem Prototyp deklariert, werden entsprechende Aufrufe vom Parser wie eingebaute Funktionen betrachtet und können solche gleichen Namens sogar überschreiben, um ihnen eine andere Semantik zu geben. Die `use`-Deklaration ist einfach ein BEGIN-Block mit bestimmtem Inhalt: Laden eines Moduls und Import von Namen in den lokalen Namensraum.

END-Blöcke werden im Gegensatz dazu so *spät* wie möglich ausgeführt: wenn Ihr Programm den Perl-Interpreter verläßt, selbst wenn das aufgrund eines nicht abgefangenen `die` oder einer anderen fatalen Ausnahme passiert. Es gibt zwei Fälle, in denen ein END-Block (oder eine DESTROY-Methode) übersprungen werden: Die Ausführung erfolgt nicht, wenn der aktuelle Prozeß sich selbst durch ein `exec` in einen anderen verwandelt. Ein durch ein nicht abgefangenes Signal aus dem Gleichgewicht gebrachtes Programm überspringt ebenfalls seine END-Routinen. (Das `use sigtrap`-Pragma in Kapitel 31, *Pragma-Module*, beschreibt eine einfache Möglichkeit, abzufangende Signale in Ausnahmen umzuwandeln. Allgemeine Informationen zur Signalbehandlung finden Sie im Abschnitt »Signale« in Kapitel 16, *Interprozeß-Kommunikation*.) Um die gesamte END-Verarbeitung zu unterbinden, können Sie `POSIX::_exit` aufrufen, `kill -9, $$` verwenden oder einfach mittels `exec` jedes harmlose Programm (bei Unix-Systemen etwa */bin/true*) ausführen.

Innerhalb des END-Blocks enthält `$?` den `exit`-Status, mit dem das Programm endet. Sie können `$?` innerhalb des END-Blocks ändern, um den Rückgabewert des Programms zu ändern. Achten Sie darauf, `$?` nicht versehentlich durch Ausführung eines anderen Programms über `system` oder Backticks zu verändern.

Sind mehrere END-Blöcke innerhalb einer Datei definiert, werden diese in der *umgekehrten* Reihenfolge ihrer Definition ausgeführt. Das bedeutet also, daß der zuletzt definierte END-Block als erster ausgeführt wird, wenn Ihr Programm endet. Diese Umkehrung ermöglicht es, zusammengehörende BEGIN- und END-Blöcke so zu verschachteln, wie Sie es erwarten würden, wenn Sie sie zu Paaren verbinden würden. Besitzen zum Beispiel ein Hauptprogramm und ein Modul beide eigene, zusammengehörende BEGIN/END-Subroutinen

```
BEGIN { print "main beginnt" }
END { print "main endet" }
use Modul;
```

und stehen im Modul die Deklarationen

```
BEGIN { print "Modul beginnt" }
END { print "Modul endet" }
```

dann weiß das Hauptprogramm, daß sein BEGIN-Block immer zuerst, sein END-Block dagegen immer zuletzt ausgeführt wird. (Ja, BEGIN ist wirklich ein compilerbezogener Block, ähnliche Argumente gelten aber auch für paarweise verbundene INIT- und END-Blöcke zur Laufzeit.) Dieses Prinzip gilt auch für alle Dateien, die einander einbinden, wenn sie solche Deklarationen besitzen. Diese Schachtelungseigenschaft macht diese Blöcke zu guten Paket-Konstruktoren und -Dekonstruktoren. Jedes Modul kann seine eigenen Funktionen zum hoch- und herunterfahren besitzen, die Perl automatisch aufruft. Auf diese Weise muß sich der Programmierer nicht merken, welcher spezielle Initialisierungs- oder Cleanup-Code einer bestimmten Bibliothek wann aufgerufen werden muß. Die Deklarationen des Moduls stellen das sicher.

Wenn Sie sich ein `eval` *`STRING`* als Sprung *zurück* vom Interpreter in den Compiler vorstellen, können Sie sich ein BEGIN als einen Sprung *vorwärts* vom Compiler in den Interpreter denken. Beide halten die aktuelle Tätigkeit kurzfristig an und wechseln den Betriebsmodus. Wenn wir sagen, daß ein BEGIN-Block so früh wie möglich ausgeführt wird, bedeutet das, daß er ausgeführt wird, sobald er vollständig definiert ist, also noch bevor der Rest der Datei vom Parser verarbeitet wurde. BEGIN-Blöcke werden daher zum Zeitpunkt der Kompilierung, nie während der Laufzeit ausgeführt. Sobald ein BEGIN-Block ausgeführt wurde, gilt er sofort als undefiniert, und jeder von ihm verwendete Code wird wieder an den Speicherpool von Perl zurückgegeben. Sie können einen BEGIN-Block nicht als Subroutine aufrufen, selbst wenn Sie es versuchen, weil er schon wieder verschwunden ist, sobald er da ist.

Ähnlich wie BEGIN-Blöcke werden auch INIT-Blöcke ausgeführt, unmittelbar bevor die Perl-Laufzeit ihren Dienst antritt. Auch hier erfolgt die Ausführung in FIFO-Reihenfolge. Zum Beispiel machen sich die (in *perlcc* dokumentierten) Codegeneratoren INIT-Blöcke zunutze, um Zeiger auf XSUBs zu initialisieren und aufzulösen. INIT-Blöcke verhalten sich fast wie BEGIN-Blöcke, nur daß sie dem Programmierer eine Unterscheidung zwischen der Konstruktion während der Kompilierungsphase und der Konstruktion während der Laufzeitphase erlauben. Wenn Sie ein Skript direkt ausführen, ist das nicht so wahnsinnig wichtig, weil der Compiler sowieso kompiliert wieder aufgerufen wird. Wenn

die Kompilierung aber getrennt von der Ausführung erfolgt, kann diese Unterscheidung entscheidend sein. Der Compiler wird möglicherweise nur einmal aufgerufen, während die erzeugte Executable immer wieder ausgeführt wird.

Wie END-Blöcke werden CHECK-Blöcke direkt nach dem Ende der Kompilierungsphase, aber vor der Ausführungsphase in LIFO-Reihenfolge ausgeführt. CHECK-Blöcke sind zum »Herunterfahren« des Compilers so nützlich wie END-Blöcke für das Herunterfahren Ihres Programms. Speziell die Backends verwenden CHECK-Blöcke als Einstiegspunkt in ihre jeweiligen Codegeneratoren. Die Backends müssen nur einen CHECK-Block in ihr eigenes Modul einbinden, und es wird zur rechten Zeit ausgeführt, ohne daß Sie einen CHECK in Ihr Programm einfügen müssen. Aus diesem Grund werden Sie kaum selbst einen CHECK-Block schreiben, es sei denn, Sie entwickeln ein solches Modul.

Zusammenfassend führt Tabelle 18-1 die verschiedenen Konstrukte auf, zusammen mit Details zum Zeitpunkt von Kompilierung und Ausführung des durch "..." dargestellten Codes. (C steht für Compilerphase und A für Ausführungsphase.)

Tabelle 18-1: Was passiert wann

Block oder Ausdruck	Kompilierung während Phase	Fängt Compiler-Fehler ab	Läuft während Phase	Fängt Laufzeit-fehler ab	Zeitpunkt des Aufrufs
`use ...`	C	Nein	C	Nein	Sofort
`no ...`	C	Nein	C	Nein	Sofort
`BEGIN {...}`	C	Nein	C	Nein	Sofort
`CHECK {...}`	C	Nein	C	Nein	Spät
`INIT {...}`	C	Nein	A	Nein	Früh
`END {...}`	C	Nein	A	Nein	Spät
`eval {...}`	C	Nein	A	Ja	Inline
`eval "..."`	A	Ja	A	Ja	Inline
`foo(...)`	C	Nein	A	Nein	Inline
`sub foo {...}`	C	Nein	A	Nein	Jederzeit
`eval "sub {...}"`	A	Ja	A	Nein	Später
`s/pat/.../e`	C	Nein	A	Nein	Inline
`s/pat/"..."/ee`	A	Ja	A	Ja	Inline

Nachdem Sie die Partitur nun kennen, hoffen wir, daß Sie in der Lage sein werden, Ihre Perl-Stücke mit größerem Vertrauen komponieren und spielen zu können.