Perl 5 Internals

Simon Cozens

Perl 5 Internals
by Simon Cozens

Open Publications License 1.0

Table of Contents

1. Preliminaries 1
1.1. Course Outline 1
1.2. Assumed Knowledge 1
1.3. Objectives 2
1.4. The course notes 2
2. Perl Development Structure 1
2.1. Perl Versioning 1
2.2. The Development Tracks 1
2.3. Perl 5 Porters 2
2.4. Pumpkins and Pumpkings 2
2.5. The Perl Repository 3
2.6. Summary 4
2.7. Exercises 4
3. Parts of the Interpreter 6
3.1. Top Level Overview 6
3.2. The Perl Library 7
3.3. The XS Library 7
3.4. The IO Subsystem 7
3.5. The Regexp Engine 8
3.6. The Parser and Tokeniser 8
3.7. Variable Handling 9
3.8. Run-time Execution 9
3.9. Support Functions 9
3.10. Testing 10
3.11. Other Utilities 10
3.12. Documentation 10
3.13. Summary 11
3.14. Exercises 11
4. Internal Variables 12
4.1. Basic SVs 12

4.1.1. Basics of an SV 12
4.1.1.1. sv_any 12
4.1.1.2. Reference Counts 13
4.1.1.3. Flags 14
4.1.2. References 15
4.1.3. Integers 16
4.1.4. Strings 18
4.1.5. Floating point numbers 23
4.2. Arrays and Hashes 23
4.2.1. Arrays 24
4.2.2. Hashes 27
4.2.2.1. What is a "hash" anyway? 27
4.2.2.2. Hash Entries 29
4.2.2.3. Hash arrays 30
4.3. More Complex Types 32
4.3.1. Objects 32
4.3.2. Magic 34
4.3.3. Tied Variables 38
4.3.4. Globs and Stashes 40
4.3.5. Code Values 42
4.3.6. Lexical Variables 44
4.4. Inheritance 46
4.5. Summary 46
4.6. Exercises 47
5. The Lexer and the Parser 48
5.1. The Parser 48
5.1.1. BNF and Parsing 48
5.1.2. Parse actions and token values 50
5.1.3. Parsing some Perl 51
5.2. The Tokeniser 53
5.2.1. Basic tokenising 53
5.2.1.1. Tokeniser State 54
5.2.1.2. Looking ahead 54

5.2.1.3. Keywords 55
5.2.2. Sublexing 55
5.3. Summary 56
5.4. Exercises 57
6. Fundamental Operations 59
6.1. The basic op 59
6.1.1. The different operations 60
6.1.2. Different "flavours" of op 61
6.1.3. Tying it all together 63
6.1.3.1. "Tree" order 63
6.1.3.2. Execution Order 64
6.2. PP Code 67
6.2.1. The argument stack 67
6.2.2. Stack manipulation 68
6.3. The opcode table and `opcodes.pl` 71
6.4. Scatchpads and Targets 72
6.5. The Optimizer 73
6.6. Summary 74
6.7. Exercises 74
7. The Perl Compiler 76
7.1. What is the Perl Compiler? 76
7.2. `B::` Modules 77
7.2.1. `B::Concise` 77
7.2.2. `B::Debug` 79
7.2.3. `B::Deparse` 80
7.3. What `B` and `O` Provide 81
7.3.1. `O` 81
7.3.2. `B` 82
7.4. Using `B` for Simple Things 83
7.5. Summary 87
7.6. Exercises 87
A. Unix cheat sheet 89
B. Editor cheat sheet 90

B.1. vi 90
B.1.1. Running 90
B.1.2. Using 90
B.1.3. Exiting 91
B.1.4. Gotchas 91
B.1.5. Help 91
B.2. pico 91
B.2.1. Running 92
B.2.2. Using 92
B.2.3. Exiting 92
B.2.4. Gotchas 92
B.2.5. Help 92
B.3. joe 93
B.3.1. Running 93
B.3.2. Using 93
B.3.3. Exiting 93
B.3.4. Gotchas 93
B.3.5. Help 94
B.4. jed 94
B.4.1. Running 94
B.4.2. Using 94
B.4.3. Exiting 94
B.4.4. Gotchas 94
B.4.5. Help 94
C. ASCII Pronunciation Guide 96

List of Tables

A-1. Simple Unix commands....89
B-1. Layout of editor cheat sheets....90
C-1. ASCII Pronunciation Guide....96

Chapter 1. Preliminaries

Welcome to NetThink's *Perl 5 Internals* training course. This is a three-hour course which provides a hands-on introduction to how the `perl` interpreter works internally, how to go about testing and fixing bugs in the interpreter, and what the internals are likely to look like in the future of Perl, Perl 6.

1.1. Course Outline

- Development Structure
- Parts of the Interpreter
- Internal Variables
- The Lexer and the Parser
- Fundamental operations
- The Runtime Environment
- The Perl Compiler
- Hacking on `perl`
- Perl 6 Internals

1.2. Assumed Knowledge

On this course, it is assumed that you will:

- be able to program Perl to at least an "intermediate" level; completing NetThink's "Intermediate Perl" course is regarded as an adequate standard.
- have some familiarity with the C programming language.
- be able to use a compiler and, if necessary, symbolic debugger, without prompting.

NOTE: Knowledge of XS is not required, but is beneficial.

1.3. Objectives

The aim of this course is to give you not just an understanding of the workings of the `perl` interpreter, but also the means to investigate more about it, to analyze and solve bugs in the Perl core, and to take part in the Perl development process.

1.4. The course notes

These course notes contain material which will guide you through the topics listed above, as well as appendices containing other useful information.

The following typographic conventions are used in these notes:

System commands appear in **this typeface**

Literal text which you should type in to the command line or editor appears as `monospaced font`.

Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL-D**

```
Program listings and other literal listings of what appears on the
screen appear in a monospaced font like this.
```

Parts of commands or other literal text which should be replaced by your own specific values appears *`like this`*

> **NOTE:** Notes and tips appear offset from the text like this.

> **ADVANCED:** Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.

> **README:** Notes marked with "Readme" are pointers to more information which can be found in your textbook or in online documentation such as manual pages or websites.

Chapter 2. Perl Development Structure

The aim of this section is to familiarize you with the process by which the Perl interpreter is developed and maintained. Most internals hacking is carried out on the "bleeding edge" of the Perl sources, and so you need to understand what these are and how to get them.

It's also important to understand the structure of the Perl development community; how it's organized, and how it works.

2.1. Perl Versioning

Perl has two types of version number: versions before 5.6.0 used a number of the form `x.yyyy_zz`; `x` was the major version number, (Perl 4, Perl 5) `y` was the minor release number, and `z` was the patchlevel. Major releases represented, for instance, either a complete rewrite or a major upheaval of the internals; minor releases sometimes added non-essential functionality, and releases changing the patchlevel were primarily to fix bugs. Releases where `z` was 50 or more were unstable, developers' releases working towards the next minor release.

Now, since, 5.6.0, Perl uses the more standard open source version numbering system - version numbers are of the form `x.y.z`; releases where `y` is even are stable releases, and releases where it is odd are part of the *development track*.

2.2. The Development Tracks

Perl development has four major aims: extending portability, fixing bugs, optimizations, and adding language features. Patches to Perl are usually made against the latest copy of the development release; the very latest copy, stored in the Perl repository (see Section 2.5 below) is usually called 'bleadperl'.

The bleadperl eventually becomes the new minor release, but patches are also picked up by the maintainer of the stable release for inclusion. While there are no hard and fast rules, and everything is left to the discretion of the maintainer, in general, patches which are bug fixes or address portability concerns (which include taking advantage of new features in some platforms, such as large file support or 64 bit integers) are merged into the stable release as well, whereas new language features tend to be left until the next minor release. Optimizations may or may not be included, depending on their impact on the source.

2.3. Perl 5 Porters

In February 2001, there were nearly 200 individuals involved in the development of Perl; these developers, or 'porters', communicate through the use of the `perl5-porters` mailing list; if you are planning to get involved in helping to develop or maintain Perl, a subscription to this list is essential.

You can subscribe by sending an email to `perl5-porters-subscribe@perl.org`; you'll be asked to send an email to confirm, and then you should start receiving mail from the list. To send mail, to the list, address the mail to `perl5-porters@perl.org`; you don't have to be subscribed to post, and the list is not moderated. If, for whatever reason, you decide to unsubscribe, simply mail `perl5-porters-unsubscribe@perl.org`.

The list usually receives between 200 and 400 mails a week. If this is too much for you, you can subscribe instead to a daily digest service by mailing `perl5-porters-digest-subscribe@perl.org`. Alternatively, I write a weekly summary of the list, published on the Perl home page (http://www.perl.com/).

There is also a `perl5-porters` FAQ (http://simon-cozens.org/writings/p5p-faq) which explains a lot of this, plus more about how to behave on P5P and how to submit patches to Perl.

2.4. Pumpkins and Pumpkings

Development is very loosely organised around the release managers of the stable and the development tracks; these are the two “pumpkings”.

Perl development can also be divided up into several smaller sub-systems: the regular expression engine, the configuration process, the documentation, and so on. Responsibility for each of these areas is known as a “pumpkin”, and hence those who semi-officially take responsibility for are called “pumpkings”.

At the time of writing, the Pumpking for 5.6.x is Gurusamy Sarathy, and the Pumpking for 5.7.x is Jarkko Hietaniemi.

You’re probably wondering why the silly names. It stems from the days before Perl was kept under version control, and people had to manually ‘check out’ a chunk of the Perl source to avoid conflicts by announcing their intentions to the mailing list; while they were discussing what this should be called, one of Chip Salzenburg’s co-workers told him about a system they had used for preventing two people using a tape drive at once: there was a stuffed pumpkin in the office, and nobody could use the drive unless they had the pumpkin.

2.5. The Perl Repository

Now Perl is kept in a version control system called Perforce (http://www.perforce.com/), which is hosted by ActiveState, Inc. There is no public access to the system itself, but various methods have been devised to allow developers near-realtime access.

Firstly, there is the Archive of Perl Changes. (ftp://ftp.linux.activestate.com/pub/staff/gsar/APC/) This FTP site contains both the current state of all the maintained Perl versions, and also a directory of changes made to the repository.

Since it’s a little inconvenient to keep up to date using FTP, the directories are also available via the software synchronisation protocol rsync (http://rsync.samba.org/). If

you have **rsync** installed, you can synchronise your working directory with the bleeding-edge Perl tree (usually called ‘bleadperl’) in the repository by issuing the command

```
% rsync -avz rsync://ftp.linux.activestate.com/perl-current/ .
```

There are also periodic snapshots of bleadperl released by the development pumpking, particularly when some important change happens. These are usually available from a variety of URLs, and always from ftp://ftp.funet.fi/pub/languages/perl/snap/.

Finally, there is a repository browser available at http://public.activestate.com/cgi-bin/perlbrowse which can tell you the current status of individual files, as well as provide an annotated ‘blame log’ cross-referencing each line in a file to the latest patch to affect it.

2.6. Summary

- Perl versions are numbers of the form x.y.z, where y is odd for development and even for stable versions.
- Perl development takes place on the perl5-porters mailing list (mailto:perl5-porters@perl.org)

2.7. Exercises

1. Obtain a copy of the development sources to Perl from CPAN. Unpack the archive, and familiarize yourself with the layout of its contents.

2. Use **rsync** to update the copy to `bleadperl`. How many bytes changed?
3. Subscribe to perl5-porters, if you haven't already done so. Spend a few moments reading through the FAQ. If you have already subscribed, read through back issues of the summaries.

Chapter 3. Parts of the Interpreter

This chapter will take you through the various parts of the `perl` interpreter, giving you an overview of its operation and the stages that a Perl program goes through when executed. By the end of this chapter you should be comfortable with the structure of the `perl` source and be able to locate functions and routines in the source tree based on a brief description of their operation.

3.1. Top Level Overview

`perl` is not exactly an interpreter and it's not exactly a compiler: it's a bytecode compiler. First compiles the input source code to an internal representation or *bytecode*, and then it executes the operations that the bytecode specifies on a virtual machine.

> **ADVANCED:** How does this differ from, say, Java? Java's virtual machine is designed to represented an idealised version of a computer's processor. In Perl's case, however, the individual operations that can be performed are considerably higher-level. For instance, a regular expression match is a single "instruction" in Perl's virtual machine.
>
> Again, like a real hardware processor, Java's VM stores its calculations in registers; Perl, on the other hand, uses a stack to co-ordinate and communicate results between operations.

The name we give to the first stage is "parsing", although, as we'll see, parsing refers to a specific operation. The input to this stage is your Perl source code; the output is a tree data structure which represents what that code "means".

One of the nodes in this tree is designated the "start" node; every node will have an operation to perform, and a pointer to the node that the interpreter must execute next.

Hence, the second phase of the operation is to execute the start node and follow the chain of pointers around the tree, executing each operation in the correct order. In later

parts of this course, we'll examine exactly how the operations are executed and what they mean.

First, however, we will examine the various distinct areas of the Perl source tree.

3.2. The Perl Library

The most approachable part of the source code, for Perl programmers, is the Perl library. This lives in `lib/`, and comprises all the standard, pure Perl modules and pragmata that ship with `perl`.

There are both Perl 5 modules and unmaintained Perl 4 libraries, shipped for backwards compatibility. In Perl 5.6.0 and above, the Unicode tables are placed in `lib/unicode`.

3.3. The XS Library

In `ext/`, we find the XS modules which ship with Perl. For instance, the Perl compiler (see Chapter 7) `B` can be found here, as can the DBM interfaces. The most important XS module here is `DynaLoader`, the dynamic loading interface which allows the runtime loading of every other XS module.

As a special exception, the XS code to the methods in the `UNIVERSAL` class can be found in `universal.c`.

3.4. The IO Subsystem

Recent versions of Perl come with a completely new standard IO implementation, `PerlIO`. This allows for several "layers" to be defined through which all IO is filtered, similar to the line disciplines mechanism in `sfio`. These layers interact with modules such as `PerlIO::Scalar`, also in the `ext/` directory.

The IO subsystem is implemented in `perlio.c` and `perlio.h`. Declarations for defining the layers are in `perliol.h`, and documentation on how to create layers is in `pod/perliol.pod`.

Perl may be compiled without `PerlIO` support, in which case there are a number of abstraction layers to present a unified IO interface to the Perl core. `perlsdio.h` aliases ordinary standard IO functions to their `PerlIO` names, and `perlsfio.h` does the same thing for the alternate IO library `sfio`.

The other abstraction layer is the "Perl host" scheme in `iperlsys.h`. This is confusing. The idea is to reduce process overhead on Win32 systems by having multiple Perl interpreters access all system calls through a shared "Perl host" abstraction object. There is an explanation of it in `perl.h`, but it is best avoided.

3.5. The Regexp Engine

Another area of the Perl source best avoided is the regular expression engine. This lives in `re*.*`. The regular expression matching engine is, roughly speaking, a state machine generator. Your match pattern is turned into a state machine made up of various match nodes - you can see these nodes in `regcomp.sym`. The compilation phase is handled by `regcomp.c`, and the state machine's execution is performed in `regexec.c`.

> **ADVANCED:** The regular expression compiler and interpreter are actually switchable; it's possible to remove Perl's default regular expression engine and insert one's own custom engine. (This is done by changing the value of the global variables `PL_regcompp` and `PL_regexecp` to be function pointers to the required routines.) In fact, that's exactly what the `re` module does.

3.6. The Parser and Tokeniser

As mentioned above, the first stage in Perl's operation is to "understand" your program. This is done by a joint effort of the tokeniser and the parser. The tokeniser is found in `toke.c`, and the parser in `perly.c`. (although you're far, far better off looking at the YACC source in `perly.y`)

The job of the tokeniser is to split up the input into meaningful chunks, or *tokens*, and also to determine what type of thing they represent - a Perl keyword, a variable, a subroutine name, and so on. The job of the parser is to take these tokens and turn them into "sentences", understanding their relative meaning in context. We'll examine their operation in more detail in Chapter 5.

3.7. Variable Handling

Perl's data types - scalars, arrays, hashes, and so on - are far more flexible than C's, and hence have to be implemented quite carefully in terms of C equivalents. The code for handling arrays is in `av.*`, hashes are in `hv.*` and scalars are in `sv.*`. See also Chapter 4.

3.8. Run-time Execution

What about the code to Perl's built-ins - `print`, `foreach` and the like? These live in `pp.*`, and will be examine in much more detail in Section 6.2. Some of the functionality is shelled out to `doio.c`.

The actual main loop of the interpreter is in `run.c`.

3.9. Support Functions

There are a number of routines which help out to make the Perl internals easier to program. For instance, `scope.[ch]` contains functions which allow you to save away and restore values on a stack. `locale.c` handles locale functions, `malloc.c` is a Perl-specific memory allocation library, `utf8.c` handles all the Unicode manipulation, `numeric.c` contains many handy numeric functions and `util.c` has various other useful things.

3.10. Testing

Every aspect of Perl's operation has a related test, and these test files live in the `t/` directory. Tests for individual library and XS modules are slowly being relocated to `lib/` and `ext/` respectively, but at time of writing, there are over 23,000 separate tests in over 400 test files.

On a related note, functions for debugging Perl itself are to be found in `deb.c` and `dump.c`. The distinction is that functions in `deb.c` are typically accessible from the `-D` flag on the Perl command line, whereas things in `dump.c` may need to be used from a source-level debugger.

3.11. Other Utilities

Perl ships with a host of utilities: from the **sed**, **awk** and **find** to Perl translators in `x2p/`, to the various utilities such as `h2xs` and `perldoc` in `utils/`.

3.12. Documentation

The POD documentation that ships with Perl lives in `pod/`, along with some of the utilities for manipulating POD documents.

3.13. Summary

We've examined the layout of the Perl source as well as an overview of the Perl interpreter. Perl runs programs in two stages: firstly reading in the source and using the tokeniser and parser to "understand" it, and then running over a series of operations to execute the program.

3.14. Exercises

1. What and where is the function that implements the `tr///` operator? Be as precise as you can.
2. How does the way Perl executes a program different from the way the Unix shell executes one? Contrast shell, Perl, Java and C.
3. Without looking, where do you think the `Perl_keyword` function would be? Find it, and explain what it does.
4. Several files in the Perl source tree are generated from other files. Look at all the `*.pl` files in the root of the Perl source tree, and find out what each file is responsible for generating, and from what sources. Be extremely careful when looking at `embed.pl`.

Chapter 4. Internal Variables

Perl's variables are a lot more flexible than C's - C is a *strongly-typed* language, whereas Perl is weakly typed. This means that Perl's variables may be used as strings, as integers, as floating point values, at will.

Hence, when we're representing values inside Perl, we need to implement some special types. This chapter will examine how scalars, arrays and hashes are represented inside the interpreter.

4.1. Basic SVs

SV stands for *Scalar Value*, and it's the basic form of representing a scalar. There are several different types of SV, but all of them have certain features in common.

4.1.1. Basics of an SV

Let's take a look at the definition of the SV type, in `sv.h` in the Perl core:

```
struct STRUCT_SV {
    void*   sv_any;     /* pointer to something */
    U32     sv_refcnt;  /* how many references to us */
    U32     sv_flags;   /* what we are */
};
```

Every scalar, array and hash that Perl knows about has these three fields: "something", a reference count, and a set of flags. Let's examine these separately:

4.1.1.1. sv_any

This field allows us to connect another structure to the SV. This is the mechanism by which we can change between representing an integer, a string, and so on. The function inside the Perl core which does the change is called `sv_upgrade`.

As its name implies, this changing is a one-way process; there is no corresponding `sv_downgrade`. This is for efficiency: we don't want to be switching types every time an SV is used in a different context, first as a number, then a string, then a number again and so on.

Hence the structures we will meet get progressively more complex, building on each other: we will see an integer type, a string type, and then a type which can hold both a string and an integer, and so on.

4.1.1.2. Reference Counts

Perl uses *reference counts* to determine when values are no longer used. For instance, consider the following two pieces of code:

```
{
  my $a;
  $a = 3;
}
```

Here, the integer value 3, an SV, is assigned to a variable. Remember that variables are simply names for values: if we look up `$a`, we find the value 3. Hence, `$a` *refers to* the value. At this point, the value has a reference count of 1.

At the closing brace, the variable `$a` goes out of scope; that is to say, the name is destroyed, and the reference to the value 3 is broken. The value's reference count therefore decreases, becoming zero.

Once an SV has a reference count of zero, it is no longer in use and its memory can be freed.

Now our second piece of code:

```
my $b;
{
  my $a;
  $a = 3;
  $b = \$a;
```

```
}
```

In this case, once we assign a reference to the value into `$b`, the reference count of our value (the integer 3) increases to 2, as now two variables are able to reach the value.

When the scope ends, the value's reference count decreases as before because `$a` no longer refers to it. However, even though one name is destroyed, another name, `$b`, still refers to the value - hence, the resulting reference count is now 1.

Once the variable `$b` goes out of scope, or a different value is assigned to it, the reference count will fall to zero and the SV will be freed.

4.1.1.3. Flags

The final field in the SV structure is a flag field. The most important flags are stored in the bottom two bits, which are used to hold the SV's type - that is, the type of structure which is being attached to the `sv_any` field.

The second most important flags are those which tell us how much of the information in the structure is relevant. For instance, we previously mentioned that one of the structures can hold both an integer and a string. We could also say that it has an integer "slot" and a string "slot". However, if we alter the value in the integer slot, Perl does not change the value in the string slot; it simply unsets the flag which says that we may use the contents of that slot:

```
$a = 3;            # Type: Integer            | Flags: Can use in-
teger
... if $a eq "3";  # Type: Integer and String | Flags: Can use integer,
                                              |        can use string
$a++;              # Type: Integer and String | Flags: Can use integer
```

Retrieving and setting flags

You can get at an SV's flags using the `SvFLAGS(sv)` macro. This is lvaluable: that is to say, you can write

```
SvFLAGS(sv) |= SVf_UTF8;
```

to turn on the `UTF8` flag. However, there are macros in `sv.h` for testing and setting flags; for instance, the above is more clearly and frequently written

```
SvUTF8_on(sv);
```

As mentioned above, the type of the SV is encoded in its flags. Use `SvTYPE(sv)` to get at this, and compare the result with the values of the `svtype` enum in `sv.h`.

We'll see more detailed examples of this later on. First, though let's examine the various types that can be stored in an SV.

4.1.2. References

A reference, or RV, is simply a C pointer to another SV, as its definition shows:

```
struct xrv {
    SV *    xrv_rv;     /* pointer to another SV */
}
```

ADVANCED: Hence, the Perl statement **`$a = \$b`** is equivalent to the C statements:

```
sv_upgrade(a, SVt_RV); /* Make sure a is an RV */
a->sv_any->xrv_rv = b;
```

However, the SV fields are hidden behind macros, so an XS programmer or porter would write the above as:

```
sv_upgrade(a, SVt_RV); /* Make sure a is an RV */
SvRV(a) = b;
```

Functions for manipulating references

You may create a reference at the C level using `newRV_inc((SV*) thing)` or `newRV_noinc((SV*) thing)`; the `_noinc` form does not increase the reference count - use with caution!

As seen above, `SvRV(rv)` dereferences the RV; be sure to cast it into the appropriate type (SV*, AV*, HV*) before doing anything with it. You can check the type using `SvTYPE(SvRV(rv))` as expected.

4.1.3. Integers

Perl's integer type is not necessarily a C int; it's called an IV, or *Integer Value*. The difference is that an IV is guaranteed to hold a pointer.

ADVANCED: Perl uses the macros `PTR2INT` and `INT2PTR` to convert between pointers and IVs. The size guarantee means that, for instance, the following code will produce an IV:

```
$a = \1;
$a--;    # Reference (pointer) converted to an integer
```

Let's now have a look at an SV structure containing an IV: the SvIV structure. The core module `Devel::Peek` allows us to examine a value from the C perspective:

```
% perl  -MDevel::Peek -le '$a=10; Dump($a)'
SV = IV(0x81559b0) at 0x81584f0                    ❶
  REFCNT = 1                                       ❷
  FLAGS = (IOK,pIOK)                               ❸
  IV = 10                                          ❹
```

❶ The first line tells us that this SV is of type SvIV. The SV has a memory location of `0x814584f0`, and `sv_any` points to an IV at memory location `0x81559b0`.

❷ The value has only one reference to it at the moment, the fact that it is stored in `$a`.

❸ `Devel::Peek` converts the flags from a simple integer to a symbolic form: it tells us that the `IOK` and `pIOK` flags are set. `IOK` means that the value in the IV slot is OK to be used.

> **ADVANCED:** What about `pIOK`? `pIOK` means that the IV slot represents the underlying ("p" for "private") data. If, for instance, the SV is tied, then we may not use the "10" that is in the IV slot - we must call the appropriate `FETCH` routine to get the value - so `IOK` is not set. The "10", however, is private data, only available to the tying mechanism, so `pIOK` is set.

❹ This shows the IV slot with its value, the "10" which we assigned to `$a`'s SV.

> **ADVANCED:** There's also a sub-type of IVs called UVs which Perl uses where possible; these are the unsigned counterparts of IVs. The flag `IsUV` is used to signal that a value in an IV slot is actually an unsigned value.

Functions for manipulating SvIVs.

You can create a new integer SV with the function `newSViv(IV foo)`.

To get the integer value of an SV, the `SvIV(sv)` macro will first ensure that the scalar has a valid IV slot, converting it if necessary, and then return the value of that slot. To change the integer value of an existing SV, use `sv_setiv(sv, iv)`.

The `SvIOK(sv)` macro can be used to check whether or not a given SV has a valid IV slot.

You should note at this point that if you title-case the type of SV (we've seen `Sv`, and we'll also see `Av`, `Hv` referring to unique properties of those types) and then add the names of the fields produced in the `Devel::Peek::Dump` dump, (`FLAGS`, `REFCNT`, `IV`) you obtain a macro that can be used from C to retrieve that property: `SvFLAGS`, `SvREFCNT` and so on.

4.1.4. Strings

The next class we'll look at are strings. We can't call them "String Values", because the SV abbreviation is already taken; instead, remembering that a string is a pointer to an array of characters, and that the entry in the string slot is going to be that pointer, we call strings "PVs": *Pointer Values*

It's here that we start to see combination types: as well as the SvPV type, we have a SvPVIV which has string and integer slots.

Before we get into that, though, let us examine the SvPV structure, again from `sv.h`:

```
struct xpv {
    char *  xpv_pv;     /* pointer to malloced string */
    STRLEN  xpv_cur;    /* length of xpv_pv as a C string */
    STRLEN  xpv_len;    /* allocated size */
};
```

C's strings have a fixed size, but Perl must dynamically resize its strings whenever the data going into the string exceeds the currently allocated size. Hence, Perl holds both the length of the current contents and the maximum length available before a resize must occur. As with SVs, allocated memory for a string only increases, as the following example shows:

```
% perl -MDevel::Peek -le '$a="abc"; Dump($a);print;
$a="abcde"; Dump($a);print; $a="a"; Dump($a)'
SV = PV(0x814ee44) at 0x8158520                                  ❶
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x815c548 "abc"\0                                         ❷
  CUR = 3                                                        ❸
  LEN = 4                                                        ❹

SV = PV(0x814ee44) at 0x8158520                                  ❺
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x815c548 "abcde"\0
  CUR = 5
  LEN = 6

SV = PV(0x814ee44) at 0x8158520                                  ❻
  REFCNT = 1
```

```
FLAGS = (POK,pPOK)
PV = 0x815c548 "a"\0
CUR = 1
LEN = 6
```

❶ This time, we have a SV whose `sv_any` points to an SvPV structure at address `0x814ee44`

❷ The actual pointer, the string, lives at address `0x815c548`, and contains the text `"abc"`. As this is an ordinary C string, it's terminated with a null character.

❸ x `SvCUR` is the length of the string, as would be returned by `strlen`. In this case, it is 3 - the null terminator is not counted.

❹ However, it is counted for the purposes of allocation: we have allocated 4 bytes to store the string, as reflected by `SvLEN`.

❺ So what happens if we lengthen the string? As the new length is more than the available space, we need to extend the string.

> **ADVANCED:** The macro `SvGROW` is responsible for extending strings to a specified length. It's defined in terms of the function `sv_grow` which takes care of memory reallocation:
>
> ```
> #define SvGROW(sv,len) (SvLEN(sv) < (len) ? sv_grow(sv,len) :
>     SvPVX(sv))
> ```

After growing the string to accomodate the new value, the value is assigned and the `CUR` and `LEN` information updated. As you can see, the SV and the SvPV structures stay at the same address, and, in this case, the string pointer itself has remained at the same address.

❻ And what if we shrink the string? Perl does not give up any memory: you can see that `LEN` is the same as it was before. Perl does this for efficiency: if it reallocated storage every time a string changed length, it would spent most of its time in memory management!

Now let’s see what happens if we use a value as number and string, taking the example in Section 4.1.1.3:

```
% perl -Ilib -MDevel::Peek -le '$a=3; Dump($a);print;
$a eq "3"; Dump($a);print; $a++; Dump($a)'
SV = IV(0x81559d8) at 0x8158518
  REFCNT = 1
  FLAGS = (IOK,pIOK)
  IV = 3

SV = PVIV(0x814f278) at 0x8158518                          ❶
  REFCNT = 1
  FLAGS = (IOK,POK,pIOK,pPOK)
  IV = 3
  PV = 0x8160350 "3"\0
  CUR = 1
  LEN = 2

SV = PVIV(0x814f278) at 0x8158518                          ❷
  REFCNT = 1
  FLAGS = (IOK,pIOK)
  IV = 4
  PV = 0x8160350 "3"\0
  CUR = 1
  LEN = 2
```

❶ In order to perform the string comparison, Perl needs to get a string value. It calls `SvPV`, the ordinary macro for getting the string value from an SV. PV notices that we don’t have a valid PV slot, so upgrades the SV to a SvPVIV. It also converts the number "3" to a string representation, and sets `CUR` and `LEN` appropriately. Because the values in both the IV and PV slots are available for use, both IOK and POK flags are turned on.

❷ When we change the integer value of the SV by incrementing it by one, Perl updates the value in the IV slot. Since the value in the PV slot is invalidated, the POK flag is turned off. Perl does not remove the value from the PV slot, nor does it downgrade to an SvIV because we may use the SV as a string again at a later time.

> **ADVANCED:** There's one slight twist here: if you ask Perl to remove some characters from the beginning of the string, it performs a (rather ugly) optimization called "The Offset Hack". It stores the number of characters to remove (the offset) in the IV slot, and turns on the `OOK` (offset OK) flag. The pointer of the PV is advanced by the offset, and the `CUR` and `LEN` fields are decreased by that many. As far as C is concerned the string starts at the new position; it's only when the memory is being released that the real start of the string is important.

Functions for manipulating strings

To create a SvPV from an ordinary string, use either `newSVpvn(char*, STRLEN)` or `newSVpvf(char* format, ...)` for `sprintf`-like formatting. `sv_setpvn(sv, char*, STRLEN)` and `sv_setpvf(sv, char* format, ...)` can be used to alter the string value of an SV. Analogous functions `sv_catpvn` etc. add to the end of the string.

As mentioned above, SvPV(sv) will return the string value, converting the SV to something which has a valid PV if necessary.

4.1.5. Floating point numbers

Finally, we have floating point types, or NVs: *Numeric Values*. Like IVs, NVs are guaranteed to be able to hold a pointer. The SvNV structure is very like the corresponding SvIV:

```
% perl -MDevel::Peek -le '$a=0.5; Dump($a);'
SV = NV(0x815d058) at 0x81584e8
```

```
  REFCNT = 1
  FLAGS = (NOK,pNOK)
  NV = 0.5
```

However, the combined structure, SvPVNV has slots for floats, integers and strings:

```
% perl -MDevel::Peek -le '$a="1"; $a+=0.5; Dump($a);'
SV = PVNV(0x814f9c0) at 0x81584f0
  REFCNT = 1
  FLAGS = (NOK,pNOK)
  IV = 0
  NV = 1.5
  PV = 0x815b5c0 "1"\0
  CUR = 1
  LEN = 2
```

Functions for manipulating NVs

By now, you should be able to guess the functions needed for manipulating NVs: `SvNV(sv)` will return the NV, converting if necessary; `sv_newSVnv(float)` will create a new SvNV; `sv_setnv(sv, float)` will change the NV.

4.2. Arrays and Hashes

Now we've looked at the most common types of scalar, (there are a few complications, which we'll cover in Section 4.3) let's examine array and hash structures. These, too, are build on top of the basic SV structure, with reference counts and flags, and structures hung off `sv_any`.

4.2.1. Arrays

Arrays are known in the core as AVs. Their structure can be found in `av.h`:

```
struct xpvav {
    char*   xav_array;  /* pointer to first array element */
    SSize_t xav_fill;   /* Index of last element present */
    SSize_t xav_max;    /* max index for which array has space */
    IV      xof_off;    /* ptr is incremented by offset */
    NV      xnv_nv;     /* numeric value, if any */
    MAGIC*  xmg_magic;  /* magic for scalar array */
    HV*     xmg_stash;  /* class package */
    SV**    xav_alloc;  /* pointer to malloced string */
    SV*     xav_arylen;
    U8      xav_flags;
};
```

We're going to skip over `xmg_magic` and `xmg_stash` for now, and come back to them in Section 4.3.

Let's use `Devel::Peek` as before to examine the AV, but we must remember that we can only give one argument to `Devel::Peek::Dump`; hence, we must pass it a reference to the AV:

```
% perl -MDevel::Peek -e '@a=(1,2,3); Dump(\@a)'
SV = RV(0x8106ce8) at 0x80fb380      ❶
  REFCNT = 1
  FLAGS = (TEMP,ROK)
  RV = 0x8105824
  SV = PVAV(0x8106cb4) at 0x8105824  ❷
    REFCNT = 2
    FLAGS = ()
    IV = 0
    NV = 0
    ARRAY = 0x80f7de8                ❸
    FILL = 2                         ❹
```

```
MAX = 3                              ❺
ARYLEN = 0x0                         ❻
FLAGS = (REAL)                       ❼
Elt No. 0
SV = IV(0x80fc1f4) at 0x80f1460      ❽
  REFCNT = 1
  FLAGS = (IOK,pIOK,IsUV)
  UV = 1
Elt No. 1
SV = IV(0x80fc1f8) at 0x80f1574
  REFCNT = 1
  FLAGS = (IOK,pIOK,IsUV)
  UV = 2
Elt No. 2
SV = IV(0x80fc1fc) at 0x80f1370
  REFCNT = 1
  FLAGS = (IOK,pIOK,IsUV)
  UV = 3
```

❶ We're dumping the reference to the array, which is, as you would expect, an RV.

❷ The RV contains a pointer to another SV: this is our array; the `Dump` function helpfully calls itself recursively on the pointer.

❸ The AV contains a pointer to a C array of SVs. Just like a string, this array must be able to change its size; in fact, the expansion and contaction of AVs is just the same as that of strings.

❹ To facilitate this, `FILL` is the highest index in the array. This is usually equivalent to `$#array`.

❺ `MAX` is the maximum allocated size of the array; if `FILL` has to become more than `MAX`, the array is grown with `av_extend`.

❻ We said that `FILL` was usually equivalent to `$#array`, but the exact equivalent is `ARYLEN`. This is an SV that is created on demand - that is, whenever `$#array` is read. Since we haven't read `$#array` in our example, it's currently a null pointer. The distinction between `FILL` and `$#array` is important when an array is tied.

❼ The `REAL` flag is set on "real" arrays; these are arrays which reference count their contents. Arrays such as `@_` and the scratchpad arrays (see below) are fake, and do not bother reference counting their contents as an efficiency hack.

❽ `Devel::Peek::Dump` shows us some of the elements of the array; these are ordinary SVs.

ADVANCED: Something similar to the offset hack is performed on AVs to enable efficient shifting and splicing off the beginning of the array; while `AvARRAY` (`xav_array` in the structure) points to the first element in the array that is visible from Perl, `AvALLOC` (`xav_alloc`) points to the real start of the C array. These are usually the same, but a shift operation can be carried out by increasing `AvARRAY` by one and decreasing `AvFILL` and `AvLEN`. Again, the location of the real start of the C array only comes into play when freeing the array. See `av_shift` in `av.c`.

Functions for manipulating arrays

You can create a new array simply with the `newAV` macro. `AvARRAY(av)` will return the underlying C array of SVs; `av_len` returns the index of the highest element, and `av_fill(av, index)` can be used to ensure that an array is grown to at least the size of `index`.

For more array manipulation functions, see `perlapi` in the Perl documentation, or *Using Perl and C* by Tim Jenness and Simon Cozens.

4.2.2. Hashes

Hashes are represented in the core as, you guessed it, HVs. Before we look at how this is implemented, we'll first see what a hash actually is...

4.2.2.1. What is a "hash" anyway?

A hash is actually quite a clever data structure: it’s a combination of an array and a linked list. Here’s how it works:

1. The hash key undergoes a transformation to turn it into a number called, confusingly, the *hash value*. For Perl, the C code that does the transformation looks like this: (from `hv.h`)

```
register const char *s_PeRlHaSh = str;
register I32 i_PeRlHaSh = len;
register U32 hash_PeRlHaSh = 0;
while (i_PeRlHaSh--) {
    hash_PeRlHaSh += *s_PeRlHaSh++;
    hash_PeRlHaSh += (hash_PeRlHaSh << 10);
    hash_PeRlHaSh ^= (hash_PeRlHaSh >> 6);
}
hash_PeRlHaSh += (hash_PeRlHaSh << 3);
hash_PeRlHaSh ^= (hash_PeRlHaSh >> 11);
(hash) = (hash_PeRlHaSh += (hash_PeRlHaSh << 15));
```

   Converting that to Perl and tidying it up:

```
sub hash {
    my $string = shift;
    my $hash;
    for (map {ord $_} split //, $string) {
         $hash += $_; $hash += $hash << 10; $hash ^= $hash >> 6;
    }
    $hash += $hash << 3; $hash ^= $hash >> 1;
    return ($hash + $hash << 15);
}
```

2. This hash is distributed across an array using the modulo operator. For instance, if our array has 8 elements, ("Hash buckets") we’ll use **`$hash_array[$hash % 8]`**

3. Each bucket contains a linked list; adding a new entry to the hash appends an element to the linked list. So, for instance, **`$hash{"red"}="rouge"`** is implemented similar to

```
push @{$hash->[hash("red") % 8]},
    { key   => "red",
      value => "rouge",
      hash  => hash("red")
    };
```

> **ADVANCED:** Why do we store the key as well as the hash value in the linked list? The hashing function may not be perfect - that is to say, it may generate the same value for `"red"` as it would for, say, `"blue"`. This is called a *hash collision*, and, while it is rare in practice, it explains why we can't depend on the hash value alone.

As usual, a picture speaks a thousand words:

4.2.2.2. Hash Entries

Hashes come in two parts: the HV is the actual array containing the linked lists, and is very similar to an AV; the things that make up the linked lists are *hash entry* structures, or HEs. From `hv.h`:

```
/* entry in hash value chain */
struct he {
    HE      *hent_next; /* next entry in chain */
    HEK     *hent_hek;  /* hash key */
    SV      *hent_val;  /* scalar value that was hashed */
};

/* hash key -- defined separately for use as shared pointer */
struct hek {
```

```
    U32     hek_hash;   /* hash of key */
    I32     hek_len;    /* length of hash key */
    char    hek_key[1]; /* variable-length hash key */
};
```

As you can see from the above, we simplified slightly when we put the hash key in the buckets above: the key and the hash value are stored in a separate structure, a HEK.

The HEK stored inside a hash entry represents the key: it contains the hash value and the key itself. It's stored separately so that Perl can share identical keys between different hashes - this saves memory and also saves time calcu.llating the hash value. You can use the macros `HeHASH(he)` and `HeKEY(he)` to retrieve the hash value and the key from a HE.

4.2.2.3. Hash arrays

Now to turn to the HVs themselves, the arrays which hold the linked lists of HEs. As we mentioned, these are not too dissimilar from AVs.

```
% perl -MDevel::Peek -e '%a = (red => "rouge", blue => "bleu"); Dump(\%a);'
SV = RV(0x8106c80) at 0x80f1370                                    ❶
  REFCNT = 1
  FLAGS = (TEMP,ROK)
  RV = 0x81057a0
  SV = PVHV(0x8108328) at 0x81057a0
    REFCNT = 2
    FLAGS = (SHAREKEYS)                                            ❷
    IV = 2
    NV = 0
    ARRAY = 0x80f7748  (0:6, 1:2)                                  ❸
    hash quality = 150.0%                                          ❹
    KEYS = 2                                                       ❺
    FILL = 2
    MAX = 7                                                        ❸
    RITER = -1                                                     ❻
```

```
EITER = 0x0                                                    ❻
Elt "blue" HASH = 0x8a5573ea                                   ❼
SV = PV(0x80f17b0) at 0x80f1574
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x80f5288 "bleu"\0
  CUR = 4
  LEN = 5
Elt "red" HASH = 0x201ed
SV = PV(0x80f172c) at 0x80f1460
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x80ff370 "rouge"\0
  CUR = 5
  LEN = 6
```

❶ As before, we dump a reference to the AV, since `Dump` only takes one parameter.

❷ The `SHAREKEYS` flag means that the key structures, the HEKs can be shared between hashes to save memory. For instance, if we have **`$french{red} = "rouge"; $german{red} = "rot"`**, the key structure is only created once, and both hashes contain a pointer to it.

❸ As we mentioned before, there are eight buckets in our hash initially - the hash gets restructured as needed. The numbers in brackets around `ARRAY` tell us about the population of those buckets: six of them have no entries, and two of them have one entry each.

❹ The "quality" of a hash is related to how long it takes to find an element, and this is in turn related to the average length of the hash chains, the linked lists attached to the buckets: if there is only one element in each bucket, you can find the key simply by performing the hash function. If, on the other hand, all the elements are in the same hash bucket, the hash is particularly inefficient.

❺ `HvKEYS(hv)` returns the number of keys in the hash - in this case, two.

❻ These two values refer to the hash iterator: when you use, for instance, `keys` or `each` to iterate over a hash, Perl uses these values to keep track of the current entry.

The "root iterator", `RITER`, is the array index of the bucket currently being iterated, and the "entry interator", `EITER`, is the current entry in the hash chain. `EITER` walks along the hash chain, and when it gets to the end, it increments `RITER` and looks at the first entry in the next bucket. As we're currently not in the middle of a hash iteration, these are set to "safe" values.

❼ As with an array, the `Dump` function shows us some of the elements; it also shows us the hash key: the key for `"blue"` is `0x3954c8`. (You can confirm that this is correct by running **`hash("blue")`** using the Perl subroutine given above.)

4.3. More Complex Types

Sometimes the information provided in an ordinary SV, HV or AV isn't enough for what Perl needs to do. For instance, how does one represent objects? What about tied variables? In this section, we'll look at some of the complications of the basic SV types.

> **ADVANCED:** The entirety of this section should be considered advanced material; it will not be covered in the course. Readers following the course should skip to the next section, Section 4.4 and study this in their own time.

4.3.1. Objects

Objects are represented relatively simply. As we know from ordinary Perl programming, an object is a reference to some data which happens to know which package it's in. In the definitions of AVs and HVs above, we saw the line

```
HV*     xmg_stash;  /* class package */
```

As we'll see in Section 4.3.4, packages are known as "stashes" internally and are represented by hashes. The `xmg_stash` field in AVs and HVs is used to store a pointer to the stash which "owns" the value.

Hence, in the case of an object which is an array reference, the dump looks like this:

```
% perl -MDevel::Peek -e '$a=bless [1,2]; Dump($a)'
SV = RV(0x81586d4) at 0x815b7a0
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x8151b0c
  SV = PVAV(0x8153074) at 0x8151b0c
    REFCNT = 1
    FLAGS = (OBJECT)                                          ❷
    IV = 0
    NV = 0
    STASH = 0x8151a34 "main"                                  ❸
    ARRAY = 0x815fcf8
    FILL = 1
    MAX = 1
    ARYLEN = 0x0
    FLAGS = (REAL)
    Elt No. 0
    SV = IV(0x815833c) at 0x8151bc0
      REFCNT = 1
      FLAGS = (IOK,pIOK)
      IV = 1
    Elt No. 1
    SV = IV(0x8158340) at 0x8151c44
      REFCNT = 1
      FLAGS = (IOK,pIOK)
      IV = 2
```

❶ We create an array reference and bless it into the main package.

❷ The `OBJECT` flag is turned on to signify that this SV is an object.

❸ And now we have a pointer to the appropriate stash in the `STASH` field.

4.3.2. Magic

This works for AVs and HVs which have a `STASH` field, but what about ordinary scalars? There is an additional, more complex type of scalar, which can hold both stash information and also permits us to hang additional, miscellaneous information onto the SV. This miscellaneous information is called "magic", (partially because it allows for clever things to happen, and partially because nobody *really* knows how it works) and the complex SV structure is a PVMG. We can create a PVMG by blessing a scalar reference:

```
% perl -MDevel::Peek -le '$b="hi";$a=bless \$b, main; print Dump($a)'
SV = RV(0x8106ca4) at 0x810586c
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x81058c0
  SV = PVMG(0x810e628) at 0x81058c0
    REFCNT = 2
    FLAGS = (OBJECT,POK,pPOK)
    IV = 0
    NV = 0
    PV = 0x80ff698 "hi"\0
    CUR = 2
    LEN = 3
    STASH = 0x80f1388 "main"
```

As you can see, this is similar to the PVNV structure we saw in Section 4.1.5, with the addition of the `STASH` field. There's also another field, which we can see if we look at the definition of `xpvmg`:

```
struct xpvmg {
    char *  xpv_pv;     /* pointer to malloced string */
```

```
    STRLEN  xpv_cur;    /* length of xpv_pv as a C string */
    STRLEN  xpv_len;    /* allocated size */
    IV      xiv_iv;     /* integer value or pv offset */
    NV      xnv_nv;     /* numeric value, if any */
    MAGIC*  xmg_magic;  /* linked list of magicalness */
    HV*     xmg_stash;  /* class package */
};
```

The `xmg_magic` field provides us with somewhere to put a magic structure. What's a magic structure, then? For this, we need to look in `mg.h`:

```
struct magic {
    MAGIC*  mg_moremagic;                                   ❶
    MGVTBL* mg_virtual; /* pointer to magic functions */    ❷
    U16     mg_private;                                     ❸
    char    mg_type;                                        ❹
    U8      mg_flags;                                       ❺
    SV*     mg_obj;                                         ❻
    char*   mg_ptr;                                         ❼
    I32     mg_len;                                         ❽
};
```

❶ First, we have a link to another magic structure: this creates a linked list, allowing us to hang multiple pieces of magic off a single SV.

❷ The magic virtual table is a list of functions which should be called to perform particular operations on behalf of the SV. For instance, a tied variable will automagically call the C function `magic_getpack` when its value is being retrieved. (This function will, in turn, call the `FETCH` method on the appropriate object.)

> **ADVANCED:** The magic virtual tables are provided by Perl - they're in `perl.h` and all begin `PL_vtbl_`. For instance, the virtual table for `%ENV` is `PL_vtbl_env`, and the table for individual elements of the `%ENV` hash is `PL_vtbl_envelem`.

In theory, you can create your own virtual tables by providing functions to fill the `mgvtbl` struct in `mg.h`, to allow for really bizarre behaviour to be triggered by accesses to your SVs. In practice, nobody really does that, although it's conceivable that you can improve the speed of pure-C tied variables that way. See also the discussion of "U" magic in Section 4.3.3 below.

❸ This is a storage area for data private to this piece of magic. The Perl core doesn't use this, but you can if you're building your own magic types. For instance, you can use it as a "signature" to ensure that this magic was created by your extension, not by some other module.

❹ Magic comes in a number of varieties: as well as providing for tied variables, magic propagates taintedness, makes special variables such as `%ENV` and `%SIG` work, and allows for special things to happen when expressions like `substr($a,0,10)` or `$#array` are assigned to.

Each of these different types of magic have a different "code letter" - the letters in use are shown in `perlguts`.

❺ There are only four flags in use for magic; the most important is `MGf_REFCOUNTED`, which is set if `mg_obj` had its reference count increased when it was added to the magic structure.

❻ This is another storage area; it's normally used to point to the object of a tied variable, so that tied functions can be located.

❼ The pointer field is set when you add magic to an SV with the `sv_magic` function. (see below) You can put anything you like here, but it's typically the name of the variable. Built-in magical virtual table functions such as `magic_get` check this to process Perl's special variables.

❽ This is the length of the string in `mg_ptr`.

What happens when the value of an SV with magic is retrieved? Firstly, a function should call `SvGETMAGIC(sv)` to cause any magic to be performed. This in turn calls `mg_get` which walks over the linked list of magic. For each piece of magic, it looks in the magic virtual table, and calls the magical "get" function if there is one.

Let’s assume that we’re dealing with one of Perl’s special variables, which has only one piece of magic, "\0" magic. The appropriate magic virtual table for "\0" magic is `PL_vtbl_sv`, which is defined as follows: (in `perl.h`)

```
EXT MGVTBL PL_vtbl_sv = {MEMBER_TO_FPTR(Perl_magic_get),
                         MEMBER_TO_FPTR(Perl_magic_set),
                         MEMBER_TO_FPTR(Perl_magic_len),
                         0,      0};
```

Magic virtual tables have five elements, as seen in `mg.h`:

```
struct mgvtbl {
    int     (CPERLscope(*svt_get))  (pTHX_ SV *sv, MAGIC* mg);
    int     (CPERLscope(*svt_set))  (pTHX_ SV *sv, MAGIC* mg);
    U32     (CPERLscope(*svt_len))  (pTHX_ SV *sv, MAGIC* mg);
    int     (CPERLscope(*svt_clear))(pTHX_ SV *sv, MAGIC* mg);
    int     (CPERLscope(*svt_free)) (pTHX_ SV *sv, MAGIC* mg);
};
```

So the above virtual table means "call `Perl_magic_set` when we want to get the value of this SV; call `Perl_magic_set` when we want to set it; call `Perl_magic_len` when we want to find its length; do nothing if we want to clear it or when it is freed from memory."

In this case, we are getting the value, so `magic_get` is called. [1] This function looks at the value of `mg_ptr`, which, as noted above, is often the name of the variable. Depending on the name of the variable, it determines what to do: for instance, if `mg_ptr` is `"!"`, then the current value of the C variable `errno` is retrieved.

A similar process is performed by `SvSETMAGIC(sv)` to call functions that need to be called when the value of an SV changes.

Adding magic to an SV

Magic is added by calling the function `sv_magic(SV* sv, SV* object, char how, char* name, STRLEN len)`. `sv` is the SV to add magic to; `object` is the SV to be placed in `mg_obj`. `how` is the character representing the "code letter" for the type of magic you wish to add. `name` and `len` will get stored in `mg_ptr` and `mg_len` respectively. This will also assign the appropriate virtual table for the type of magic - see the list in `perlguts`.

Note that for user-defined magic, "~" magic, you must set the virtual table manually. (Good luck.)

4.3.3. Tied Variables

Tied arrays and hashes are implementing by adding type "P" magic to their AVs and HVs; individual elements of the arrays and hashes have "p" magic. Tied scalars and filehandles have type "q" magic. The virtual tables for, for instance, "p" magic scalars look like this:

```
EXT MGVTBL PL_vtbl_packelem =   {MEMBER_TO_FPTR(Perl_magic_getpack),
                                 MEMBER_TO_FPTR(Perl_magic_setpack),
                                 0,
                                 MEMBER_TO_FPTR(Perl_magic_clearpack),
                                 0}
```

That’s to say, the function `magic_getpack` is called when the value of an element of a tied array or hash is retrieved. This function in turn performs a `FETCH` method call on the object stored in `mg_obj`.

We can invent our own "pseudo-tied" variables, using the user-defined "U" magic. "U" magic only works on scalars, and allows us to call a function when the value of the scalar is got or set. The virtual table for "U" magic scalars is as follows:

```
EXT MGVTBL PL_vtbl_uvar =   {MEMBER_TO_FPTR(Perl_magic_getuvar),
```

```
                                MEMBER_TO_FPTR(Perl_magic_setuvar),
                                0,  0,  0};
```

As you should by now expect, these functions are called when the value of the scalar is accessed. They in turn call our user-defined functions. But how do we tell them what our functions are? In this case, we pass a pointer to a special structure in the `mg_ptr` field; the structure is defined in `perl.h`, and looks like this:

```
struct ufuncs {
    I32 (*uf_val)(IV, SV*);
    I32 (*uf_set)(IV, SV*);
    IV uf_index;
};
```

Here are our two function pointers: `uf_val` is called with the value of `uf_index` and the scalar when the value is sought, and `uf_set` is called with the same parameters when it is set.

Hence, the following code allows us to emulate `$!`:

```
I32 get_errno(IV index, SV* sv) {
    sv_setiv(sv, errno);
}

I32 set_errno(IV index, SV* sv) {
    errno = SvIV(sv); /* Some Cs don't like us setting errno, but hey */
}

struct ufuncs uf;

/* This is XS code */

void
magicify(sv)
    SV *sv;
CODE:
    uf.uf_val = &get_errno;
    uf.uf_set = &set_errno;
```

```
    uf.uf_index = 0;
    sv_magic(sv, 0, 'U', (char*)&uf, sizeof(uf));
```

If you need any more flexibility than that, it's time to look into "~" magic.

4.3.4. Globs and Stashes

SVs that represent variables are kept in the symbol table; as you'll know from your Perl programming, the symbol table starts at `%main::` and is an ordinary Perl hash, with the package and variable names as hash keys. But what are the hash values? Let's have a look:

```
% perl -le '$a=5; print ${main::}{a}'
*main::a
```

Well, that doesn't tell us very much - at first sight it just looks like an ordinary string. But if we use `Devel::Peek` on it, we find it's actually something else - a glob, or GV:

```
% perl -MDevel::Peek -e '$a=5; Dump ${main::}{a}'
SV = PVGV(0x80fe3e0) at 0x80fb3ec
  REFCNT = 2
  FLAGS = (GMG,SMG)                                          ❶
  IV = 0
  NV = 0
  MAGIC = 0x80fea50
    MG_VIRTUAL = &PL_vtbl_glob                               ❶
    MG_TYPE = '*'
    MG_OBJ = 0x80fb3ec                                       ❷
    MG_LEN = 1
    MG_PTR = 0x81081d8 "a"
  NAME = "a"                                                 ❸
  NAMELEN = 1
  GvSTASH = 0x80f1388 "main"                                 ❹
```

```
GP = 0x80ff2b0                                  ❺
  SV = 0x810592c                                ❻
  REFCNT = 1                                    ❼
  IO = 0x0                                      ❽
  FORM = 0x0                                    ❽
  AV = 0x0                                      ❽
  HV = 0x0                                      ❽
  CV = 0x0                                      ❽
  CVGEN = 0x0                                   ❾
  GPFLAGS = 0x0                                 (10)
  LINE = 1
  FILE = "-e"
  FLAGS = 0x0
  EGV = 0x80fb3ec "a"
```

❶ Globs have get and set magic to handle glob aliasing as well as the conversion to strings we saw above.

❷ The glob’s magic object points back to the GV itself, so that the magic functions can easily access it.

❸ The "name" is simply the variable’s unqualified name; this is combined with the "stash" below to make up the "full name".

❹ The stash itself is a pointer to the hash in which this glob is contained.

❺ This structure, a GP structure, actually holds the symbol table entry. It’s separated out so that, in the case of aliased globs, multiple GVs can point to the same GP.

❻ As we know, globs have several different "slots", for scalars, arrays, hashes and so on. This is the scalar slot, which is a pointer to an SV.

❼ The GP is refcounted because we need to know how many GVs point to it, so it can be safely destroyed when no longer needed.

❽ The other slots are a filehandle, a form, an array, a hash and a code value. (see Section 4.3.5)

❾ This stores the "age" of the code value. Every time a subroutine is defined, Perl increments the variable `PL_sub_generation`. This can be used as a way of checking the method cache: if the current value of `PL_sub_generation` is equal to the one stored in a GP, this GP is still valid.

(10) The GP's flags are currently unused.

Symbol tables are considered some of the hairiest voodoo in the Perl internals.

> **ADVANCED:** From C, the variable `PL_defstash` is the HV representing the `main::` stash; `PL_curstash` contains the current package's stash.

4.3.5. Code Values

The final data type we will examine is the CV, a code value used for storing subroutines. Both Perl and XSUB subroutines are stored in CVs, and blocks are also stored in CVs. The CV structure can be found in `cv.h`:

```
struct xpvcv {
    char *  xpv_pv;     /* pointer to malloced string */
    STRLEN  xpv_cur;    /* length of xp_pv as a C string */
    STRLEN  xpv_len;    /* allocated size */
    IV      xof_off;    /* integer value */
    NV      xnv_nv;     /* numeric value, if any */
    MAGIC*  xmg_magic;  /* magic for scalar array */
    HV*     xmg_stash;  /* class package */

    HV *    xcv_stash;                                  ❶
    OP *    xcv_start;                                  ❷
```

```
    OP *    xcv_root;                                       ❷
    void    (*xcv_xsub) (pTHXo_ CV*);                       ❸
    ANY     xcv_xsubany;                                    ❹
    GV *    xcv_gv;                                         ❺
    char *  xcv_file;                                       ❻
    long    xcv_depth;  /* >= 2 indicates recursive call */❼
    AV *    xcv_padlist;                                    ❽
    CV *    xcv_outside;                                    ❾
#ifdef USE_THREADS
    perl_mutex *xcv_mutexp;                                 (10)
    struct perl_thread *xcv_owner;  /* current owner thread */(10)
#endif /* USE_THREADS */
    cv_flags_t  xcv_flags;                                  (10)
}
```

❶ Although it might look like this provides the CV's stash, it is important to note that this is a pointer to the stash in which the CV was *compiled*; for instance, given

```
package First;
sub Second::mysub { ...}
```

then `xcv_stash` points to `First::`. This is why, for instance,

```
package One;
$x = "In One";
package Two;
$x = "In Two";
sub One::test { print $x }
package main;
One::test();
```

will print `"In Two"`.

❷ For a subroutine defined in Perl, these two pointers hold the start and the root of the compiled op tree; this will be further in Chapter 6.

❸ For an XSUB, on the other hand, this field contains a function pointer pointing to the C function implementing the subroutine.

❹ This is how constant subroutines are implemented: Perl can arrange for the SV representing the constant to be returned by a constant XS routine, which is hung here.

❺ This simply holds a pointer to the glob by which the subroutine was defined.

❻ This stores the name of the file in which the subroutine was defined. For an XSUB, this will be the `.c` file rather than the `.xs` file.

❼ This is a counter which is incremented each time the subroutine is entered and decremented when it is left; this allows Perl to keep track of recursive calls to a subroutine.

❽ Explained below, `xcv_padlist`, the pad list, contains the lexical variables declared in a subroutine or code block.

❾ Consider the following code:

```
{
   my $x = 0;
   sub counter { return ++$x; }
}
```

When inside `counter`, where does Perl "get" the SV `$x` from? It's not a global, so it doesn't live in a stash. It's not declared in `counter`, so it doesn't belong in `counter`'s pad list. It actually belong to the pad list for the CV "outside" of `counter`. To enable Perl to get at these variables and also at lexicals used in closures, each CV contains a pointer to CV of the enclosing scope.

4.3.6. Lexical Variables

Global variables live, as we've seen, in symbol tables or "stashes". Lexical variables, on the other hand, are tied to blocks rather than packages, and so are stored inside the CV representing their enclosing block.

As mentioned briefly above, the `xcv_padlist` element holds a pointer to an AV. This array, the padlist, contains the names and values of lexicals in the current code block. Again, a diagram is the best way to demonstrate this:

The first element of the padlist - called the "padname" - is an array containing the names of the variables, and the other elements are lists of the current values of those variables. Why do we have several lists of current values? Because a CV may be entered several times - for instance, when a subroutine recurses. Having, essentially, a stack of frames ensures that we can restore the previous values when a recursive call ends. Hence, the current values of lexical variables are stored in the last element on the padlist.

> **ADVANCED:** From inside perl, you can get at the current pad as `PL_curpad`. Note that this is the pad itself, not the padlist. To get the padlist, you need to perform some awkwardness:
>
> ```
> I32 cxix    = dopoptosub(cxstack_ix) /* cxstack_ix is a macro */
> AV* padlist = cxix ? CvPADLIST(cxstadck[ix].blk_sub.cv) : PL_comppadlist;
> ```
>
> We'll visit pads again when we look at operator targets in Section 6.4.

4.4. Inheritance

As we have seen, some types of SV deliberately build on and extend the structure of others. The SV code is written to attempt to provide an object-oriented style of programming inside C, and it is fair to say that some SV "classes" inherit from others. In the compiler module `B`, we see these inheritance relationships spelt out:

```
@B::PV::ISA = 'B::SV';
@B::IV::ISA = 'B::SV';
@B::NV::ISA = 'B::IV';
@B::RV::ISA = 'B::SV';
```

```
@B::PVIV::ISA = qw(B::PV B::IV);
@B::PVNV::ISA = qw(B::PV B::NV);
@B::PVMG::ISA = 'B::PVNV';
@B::PVLV::ISA = 'B::PVMG';
@B::BM::ISA = 'B::PVMG';
@B::AV::ISA = 'B::PVMG';
@B::GV::ISA = 'B::PVMG';
@B::HV::ISA = 'B::PVMG';
@B::CV::ISA = 'B::PVMG';
@B::IO::ISA = 'B::PVMG';
@B::FM::ISA = 'B::CV';
```

4.5. Summary

Perl uses several variable types in its internal representation to achieve the flexibility that is needed for its external types: scalars, (SVs) arrays, (AVs) hashes (HVs) and code blocks. (CVs)

The module `Devel::Peek` allows us to examine how Perl types are repesented internally. The field names produced by `Devel::Peek` can be easily turned into macros which allow us to get and set the values of the fields from C.

The key files from the Perl source tree which deal with Perl's internal variables are `sv.c`, `av.c` and `hv.c`; the documentation in the associated header files (`sv.h`, `av.h` and `hv.h`) is extremely helpful for understanding how to deal with Perl's internal variables.

4.6. Exercises

1. One thing we didn't do in this chapter was run `Devel::Peek` on a subroutine. Try it on a named subroutine reference, an anonymous subref and a subref to an XS

routine.

2. See if you can work out what ’FM’, ’IO’, ’BM’ and ’PVLV’ are in the above; try creating them in Perl and dumping them out with `Devel::Peek`. Use `sv.h` to explain the new fields.

Notes

1. We’ll see later that Perl uses the `Perl_` prefix internally for function names, but that prefix can be omitted inside the Perl core. Hence, we’ll call `Perl_magic_get` "`magic_get`".

Chapter 5. The Lexer and the Parser

In this chapter, we're going to examine how Perl goes about turning a piece of Perl code into an internal representation ready to be executed. The nature of the internal representation, a tree of structures representing operations, will be looked at in the next chapter, but here we'll just concern ourselves with the lexer and parser which work together to "understand" Perl code.

5.1. The Parser

The parser lives in `perly.y`. This is code in a language called Yacc, which is converted to C using the **byacc** command.

> **ADVANCED:** In fact, Perl needs to do some fixing up on the **byacc** output to have it deal with dynamic rather than static memory allocation. Hence, if you make any changes to `perly.y`, just running **byacc** isn't enough - you need to run the Make target `run_byacc`, which will do the fixups that Perl requires.

In order to understand this language, we need to understand how grammars work and how parsing works.

5.1.1. BNF and Parsing

Computer programmers define a language by its grammar, which is a set of rules. They usually describe this grammar in a form called "Backhaus-Naur Form" [1] or *BNF*. BNF tells us how phrases fit together to make sentences. For instance, here's a simple BNF for English - obviously, this isn't going to describe the whole of the English grammar, but it's a start:

```
sentence   : nounphrase verbphrase nounphrase;
```

```
verbphrase : VERB;

nounphrase : NOUN
| ADJECTIVE  nounphrase
| PRONOMINAL nounphrase
| ARTICLE    nounphrase;
```

Here is the prime rule of BNF: you can make the thing on the left of the colon if you see all the things on the right in sequence. So, this grammar tells us that a sentence is made up of a noun phrase, a verb phrase and then a noun phrase. The vertical bar does exactly what it does in regular expressions: you can make a noun phrase if you have a noun, or an adjective plus another noun phrase, or an article plus a noun phrase. Turning the things on the right into the thing on the left is called a *reduction*. The idea of parsing is to reduce all of the input down to the first thing in the grammar - a sentence.

You'll notice that things which can't be broken down any further are in capitals - there's no rule which tells us how to make a noun, for instance. This is because these are fed to us by the lexer; these are called *terminal symbols*, and the things which aren't in capitals are called *non-terminal symbols*. Why? Well, let's see what happens if we try and parse a sentence in this grammar.

The text right at the bottom - "my cat eats fish" - is what we get in from the user. The tokeniser then turns that into a series of tokens - "PRONOMINAL NOUN VERB NOUN". From that, we can start performing some reductions: we have a pronominal, so we're looking for a noun phrase to satisfy the `nounphrase : PRONOMINAL nounphrase` rule. Can we make a noun phrase? Yes, we can, by reducing the `NOUN` ("cat") into a `nounphrase`. Then we can use `PRONOMINAL nounphrase` to make another `nounphrase`.

Now we've got a `nounphrase` and a `VERB`. We can't do anything further with the `nounphrase`, so we'll switch to the `VERB`, and the only thing we can do with that is turn it into a `verbphrase`. Finally, we can reduce the noun to a `nounphrase`, leaving us with `nounphrase verbphrase nounphrase`. Since we can turn this into a `sentence`, we've parsed the text.

5.1.2. Parse actions and token values

It’s important to note that the tree we’ve constructed above - the "parse tree" - is only a device to help us understand the parsing process. It doesn’t actually exist as a data structure anywhere in the parser. This is actually a little inconvenient, because the whole point of parsing a piece of Perl text is to come up with a data structure pretty much like that.

Not a problem. Yacc allows us to extend BNF by adding actions to rules - every time the parser performs a reduction using a rule, it can trigger a piece of C code to be executed. Here’s an extract from Perl’s grammar in `perly.y`:

```
term    :   term ASSIGNOP term
  { $$ = newASSIGNOP(OPf_STACKED, $1, $2, $3); }
        |   term ADDOP term
  { $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

The pieces of code in the curlies are actions to be performed. Here’s the final piece of the puzzle: each symbol carries some additional information around. For instance, in our "cat" example, the first `NOUN` had the value "cat". You can get at the value of a symbol by a Yacc variable starting with a dollar sign: in the example above, `$1` is the value of the first symbol on the right of the colon (`term`), `$2` is the value of the second symbol (either `ASSIGNOP` or `ADDOP` depending on which line you’re reading) and so on. `$$` is the value of the symbol on the left. Hence information is propagated "up" the parse tree by manipulating the information on the right and assigning it to the symbol on the left.

5.1.3. Parsing some Perl

So, let’s see what happens if we parse the Perl code **`$a = $b + $c`**. We have to assume that **`$a`**, **`$b`** and **`$c`** have already been parsed a little; they’ll turn into `term` symbols. Each of these symbols will have a value, and that will be an "op". An "op" is a data structure representing an operation, and the operation to be represented will be that of retrieving the storage pointed to by the appropriate variable.

Let's start from the right[2] , and deal with **`$b + $c`**. The **`+`** is turned by the lexer into the terminal symbol `ADDOP`. Now, just like there can be lots of different nouns that all get tokenised to `NOUN`, there can be several different `ADDOP`s - concatenation is classified as an `ADDOP`, so **`$b . $c`** would look just the same to the parser. The difference, of course, is the value of the symbol - this `ADDOP` will have the value `'+'`.

Hence, we have `term ADDOP term`. This means we can perform a reduction, using the second rule in our snippet. When we do that, we have to perform the code in curlies underneath the rule - `{ $$ = newBINOP($2, 0, scalar($1), scalar($3)); }`. `newBINOP` is a function which creates a new binary "op". The first argument is the type of binary operator, and we feed it the value of the second symbol. This is `ADDOP`, and as we have just noted, this symbol will have the value `'+'`. So although `'.'` and `'+'` look the same to the parser, they'll eventually be distinguished by the value of their symbol. Back to `newBINOP`. The next argument is the flags we wish to pass to the op. We don't want anything special, so we pass zero.

Then we have our arguments to the binary operator - obviously, these are the value of the symbol on the left and the value of the symbol on the right of the operator. As we mentioned above, these are both "op"s, to retrieve the values of `$b` and `$c` respectively. We assign the new "op" created by `newBINOP` to be the value of the symbol we're propagating upwards. Hence, we've taken two ops - the ones for `$b` and `$c` - plus an addition symbol, and turned them into a new op representing the combined action of fetching the values of `$b` and `$c` and then adding them together.

Now we do the same thing with `$a = ($b+$c)`. I've put the right hand side in brackets to show that we've already got something which represents fetching `$b` and `$c` and adding them. `=` is turned into an `ASSIGNOP` by the tokeniser in the same way as we turned `+` into an `ADDOP`. And, in just the same way, there are various different types of assignment operator - `||=` and `&&=` are also passed as `ASSIGNOP`s. From here, it's easy: we take the `term` representing `$a`, plus the `ASSIGNOP`, plus the `term` we've just constructed, reduce them all to another `term`, and perform the action underneath the rule. In the end, we end up with a data structure a little like this:

You can find a hypertext version of the Perl grammar at http://simon-cozens.org/hacks/grammar.pdf

5.2. The Tokeniser

The tokeniser, in `toke.c` is one of the most difficult parts of the Perl core to understand; this is primarily because there is no real "roadmap" to explain its operation. In this section, we'll try to show how the tokeniser is put together.

5.2.1. Basic tokenising

The core of the tokeniser is the intimidatingly long `yylex` function. This is the function called by the parser, `yyparse`, when it requests a new token of input.

First, some basics. When a token has been identified, it is placed in `PL_tokenbuf`. The file handle from which input is being read is `PL_rsfp`. The current position in the input is stored in the variable `PL_bufptr`, which is a pointer into the PV of the SV `PL_linestr`. When scanning for a token, the variable `s` advances from the start of `PL_bufptr` towards the end of the buffer (`PL_bufend`) until it finds a token.

The first thing the parser does is test whether the next thing in the input stream has already been identified as an identifier; when the tokeniser sees `'%'`, `'$'` and the like as part of the input, it tests to see whether it introduces a variable. If so, it puts the variable name into the token buffer. It then returns the type sigil (`%`, `$`, etc.) as a token, and sets a flag (`PL_pending_ident`) so that the next time `yylex` is called, it can pull the variable name straight out of the token buffer. Hence, right at the top of `yylex`, you'll see code which tests `PL_pending_ident` and deals with the variable name.

5.2.1.1. Tokeniser State

Next, if there's no identifier in the token buffer, it checks its tokeniser state. The tokeniser uses the variable `PL_lex_state` to store state information.

One important state is `LEX_KNOWNEXT`, which occurs when Perl has had to look ahead one token to identify something. If this happens, it has tokenised not just the next token, but the one after as well. Hence, it sets `LEX_KNOWNEXT` to say "we've already tokenised this token, simply return it."

The functions which set `LEX_KNOWNEXT` are `force_word`, which declares that the next token has to be a word, (for instance, after having seen an arrow in `$foo->bar`) `force_ident`, which makes the next token an identifier, (for instance, if it sees a `*` when not expecting an operator, this must be a glob) `force_version`, (on seeing a number after `use`) and the general `force_next`.

Many of the other states are to do with interpolation of double-quoted strings; we'll look at those in more detail in the next section.

5.2.1.2. Looking ahead

After checking the lexer state, it's time to actually peek at the buffer and see what's waiting; this is the start of the giant `switch` statement in the middle of `yylex`, just following the label `retry`.

One of the first things we check for is character zero - this signifies either the start or the end of the file or the end of the line. If it's the end of the file, the tokeniser returns zero and the game is one; at the beginning of the file, Perl has to process the code for command line switches such as `-n` and `-p`. Otherwise, Perl calls `filter_gets` to get a new line from the file through the source filter system, and calls `incline` to increase the line number.

The next test is for comments and new lines, which Perl skips over. After that come the tests for individual special characters. For instance, the first test is for minus, which could be unary minus if followed by a number or identifier, or the binary minus operator if Perl is expecting an operator, or the arrow operator if followed by a `>`, or the start of a filetest operator if followed by an appropriate letter, or a quoting option such as `(-foo => "bar" )`. Perl tests for each case, and returns the token type using one of the upper-case token macros defined at the beginning of `toke.c`: `OPERATOR`, `TERM`, and so on.

If the next character isn't a symbol that Perl knows about, it's an alphabetic character which might start a keyword: the tokeniser jumps to the label `keylookup` where it checks for labels and things like `CORE::function`. It then calls `keyword` to test whether it is a valid built-in or not - if so, `keyword` turns it into a special constant (such as `KEY_open`) which can be fed into the `switch` statement. If it's not a keyword, Perl

has to determine whether it's a bareword, a function call or an indirect object or method call.

5.2.1.3. Keywords

The final section of the `switch` statement deals with the `KEY_` constants handed back from `keyword`, performing any actions necessary for using the builtins. (For instance, given `__DATA__`, the tokeniser sets up the `DATA` filehandle.)

5.2.2. Sublexing

"Sublexing" refers to the the fact that inside double-quoted strings and other interpolation contexts (regular expressions, for instance) a different type of tokenisation is needed.

This is typically started after a call to `scan_str`, which is an exceptionally clever piece of code which extracts a string with balanced delimiters, placing it into the SV `PL_lex_stuff`. Then `sublex_start` is called which sets up the data structures used for sublexing and changes the lexer's state to `LEX_INTERPPUSH`, which is essentially a scoping operator for sublexing.

Why does sublexing need scoping? Well, consider something like `"Foo\u\LB\uarBaz"`. This actually gets tokenized as the moral equivalent of `"Foo" . ucfirst(lc("B" . ucfirst("arBaz")))`. The push state (which makes a call to `sublex_push`) quite literally pushes an opening bracket onto the input stream.

This in turn changes the state to `LEX_INTERPCONCAT`; the concatentation state uses `scan_const` to pull out constant strings and supplies the concatenation operator between them. If a variable to be interpolated is found, the state is changed to `LEX_INTERPSTART`: this means that `"foo$bar"` is changed into `"foo".$bar` and `"foo@bar"` is turned into `"foo".join($",@bar)`.

There are times when it is not sure when sublexing of an interpolated variable should end - in these cases, the horrifyingly scary function `intuit_more` is called to make an

educated guess on the likelihood of more interpolation.

Finally, once sublexing is done, the state is set to `LEX_INTERPEND` which fixes up the closing brackets.

5.3. Summary

In this chapter, we've briefly examined how Perl turns Perl source input into a tree data structure suitable for executing; in the next chapter, we'll look more specifically at the nature of the nodes in that tree.

There are two stages to this operation: the tokeniser, `toke.c`, chops up the incoming program and recognises different token types; the parser `perly.y` then assembles these tokens into phrases and sentences. In reality, the whole task is driver by the parser - Perl calls `yyparse` to parse a program, and when the parser needs to know about the next token, it calls `yylex`.

While the parser is relatively straightforward, the tokeniser is somewhat more tricky. The key to understanding it is to divide its operation into checking tokeniser state, dealing with non-alphanumeric symbols in ordinary program code, dealing with alphanumerics, and dealing with double-quoted strings and other interpolation contexts.

Very few people actually understand the whole of how the tokeniser and parser work, but this chapter should have given you a useful insight into how Perl understands program code, and how to locate the source of particular behaviour inside the parsing system.

5.4. Exercises

1. What do you think the `LEX_FORMLINE` state is for? Work out what it does.

2. You can put `#!perl -p` at the top of your file and Perl will behave as though the `-p` command-line switch was given. Since exactly the same mechanism handles incoming code from a file and from `eval`, why won't it do that if you say `eval qq[#!perl -p]`?
3. Why is `--$a++` a syntax error? Explain in terms of how it should be parsed. Look for the `PREDEC` and `POSTINC` types in the grammar. What would you need to change to make it parse?
4. In the current Perl source, just after `case "#":`, you'll find test marked "Found by Ilya", which tests for a buffer overflow. How could that conceivably occur? Work out what would trigger the error message and produce some Perl code which would do so.

Notes

1. Sometimes "Backhaus Normal Form"
2. This is slightly disingenous, as parsing is always done from left to right, but this simplification is easier than getting into the details of how Yacc grammars recognise the precendence of operators.

Chapter 6. Fundamental Operations

So we've seen that the job of the parsing stage is to reduce a program to a tree structure, and each node of the tree represents an operation. In this chapter, we'll look more closely at those operations: what they are, how they're coded, and how they fit together.

6.1. The basic op

Just AVs and HVs are "extensions" of the basic SV structure, there are a number of different "flavours" of ops, built on a basic OP structure; you can find this structure defined as `BASEOP` in `op.h`:

```
    OP*         op_next;
    OP*         op_sibling;
    OP*         (CPERLscope(*op_ppaddr))(pTHX);
    PADOFFSET   op_targ;
    OPCODE      op_type;
    U16         op_seq;
    U8          op_flags;
   U8          op_private;
```

Some of these fields are easy to explain, so we'll deal with them now.

The `op_next` field is a pointer to the next op which needs to be executed. We'll see later, in Section 6.1.3, how the "thread of execution" is derived from the tree.

`op_ppaddr` is the address of the C function which carries out this particular operation. It's stored here so that our main execution code can simply dereference the function pointer and jump to it, instead of having to perform a lookup.

Each unique operation has a different number; this can be found in the `enum` in `opnames.h`:

```
typedef enum opcode {
```

```
    OP_NULL,        /* 0 */
    OP_STUB,        /* 1 */
    OP_SCALAR,      /* 2 */
    OP_PUSHMARK,    /* 3 */
    OP_WANTARRAY,   /* 4 */
    OP_CONST,       /* 5 */
    OP_GVSV,        /* 6 */
    OP_GV,          /* 7 */
    ...
};
```

The number of the operation to perform is stored in the `op_type` field. We'll examine some of the more interesting operations in Section 6.1.1.

`op_flags` is a set of flags generic to all ops; `op_private` stores flags which are specific to the type of op. For instance, the `repeat` op which implements the `x` operator has the flag `OPpREPEAT_DOLIST` set when it's repeating a list rather than a string. This flag only makes sense for that particular operation, so is stored in `op_private`. Private flags have the `OPp` prefix, and public flags begin with `OPf`.

`op_seq` is a sequence number allocated by the optimizer. It allows for, for instance, correct scoping of lexical variables by storing the sequence numbers of the beginning and end of scope operations inside the pad.

As for the remaining fields, we'll examine `op_sibling` in Section 6.1.2 and `op_targ` in Section 6.4

6.1.1. The different operations

Perl has currently 351 different operations, implementing all the built-in functions and operators, as well as the more structural operations required internally - entering and leaving a scope, compiling regular expressions and so on.

The array `PL_op_desc` in `opcode.h` describes each operation: it may be easier to follow the data from which this table is generated, at the end of `opcode.pl`. We'll take a longer look at that file later on in this chapter.

Many of the operators are familiar from Perl-space, such as `concat` and `splice`, but some are used purely internally: for instance, one of the most common, `gvsv` fetches a scalar variable; `enter` and `leave` are block control operators, and so on.

6.1.2. Different "flavours" of op

There are a number of different "flavours" of op structure, related to the arguments of an operator and how it fits together with other ops in the op tree. For instance, `scalar` is a unary operator, a UNOP. This extends the basic op structure above with a link to the argument:

```
struct unop {
    BASEOP
    OP *    op_first;
};
```

Binary operators, such as `i_add`, (integer addition) have both a `first` and a `last`:

```
struct binop {
    BASEOP
    OP *    op_first;
    OP *    op_last;
};
```

List operators are more interesting; they too have a `first` and a `last`, but they also have some ops in the middle, too. This is where `op_sibling` above comes in; it connects ops "sibling" ops on the same level in a list. For instance, look at the following code and the graph of its op tree:

```
open FILE, "foo";
print FILE "hi\n";
close FILE;
```

The dashed lines represent `op_sibling` connections. The root operator of every program is the list operator `leave`, and its children are the statements in the program, separated by `nextstate` (next statement) operators. `open` is also a list operator, as is `print`. The first child of `print` is `pushmark`, which puts a mark on the stack (see Section 6.2.1) so that Perl knows how many arguments on the stack belong to `print`. The `rv2gv` turns a reference to the filehandle `FILE` into a GV, so that `print` can print to it, and the final child is the constant `"hi\n"`.

Some operators hold information about the program; these are COPs, or "code operators". Their definition is in `cop.h`:

```
struct cop {
    BASEOP
    char *  cop_label;  /* label for this construct */
#ifdef USE_ITHREADS
    char *  cop_stashpv;    /* package line was compiled in */
    char *  cop_file;   /* file name the following line # is from */
#else
    HV *    cop_stash;  /* package line was compiled in */
    GV *    cop_filegv; /* file the following line # is from */
#endif
    U32     cop_seq;    /* parse sequence number */
    I32     cop_arybase;    /* array base this line was compiled with */
    line_t      cop_line;       /* line # of this command */
    SV *    cop_warnings;   /* lexical warnings bitmask */
    SV *    cop_io;     /* lexical IO defaults */
};
```

COPs are inserted between every statement; they contain the label (for `goto`, `next` and so on) of the statement, the file name, package and line number of the statement and lexical hints such as the current value of `$[`, warnings and IO settings. Note that this doesn’t contain the current CV or the padlist - these are kept on a special stack called the "context stack".

The final type of op is the null op: any op with type zero means that a previous op has been optimized away; we’ll look at how this is done later in this chapter, but for now, you should skip over the null op when you see it in op trees.

6.1.3. Tying it all together

We've so far seen a little of how the op tree is connected together with `op_first`, `op_last`, `op_sibling`, and so on. Now we'll look at how the tree gets manufactured, as how it gets executed.

6.1.3.1. "Tree" order

After our investigation of the parser in the previous chapter, it should now be straightforward to see how the op tree is created. The parser calls routines in `op.c` which create the op structures, passing ops further "down" the parse tree as arguments. This threads together a tree as shown in the diagram above. For comparison, here is the what the example in that chapter (`$a = $b + $c`) really looks like as an op tree:

Again, you can see the places where an op was optimized away and became a null op. This is not so different from the simplified version we gave earlier.

6.1.3.2. Execution Order

The second thread through the op tree, indicated by the dotted line in our diagrams, is the execution order. This is the order in which Perl must actually perform the operations in order to run the program. The main loop of Perl is very, very simple, and you can see it in `run.c`:

```
while ((PL_op = CALL_FPTR(PL_op->op_ppaddr)(aTHX))) {
    PERL_ASYNC_CHECK();
}
```

That's it. That's all the Perl interpreter is. `PL_op` represents the op that's currently being executed. Perl calls the function pointer for that op and expects another op to be returned; this return value is then set to `PL_op`, which is executed in turn. Since everything apart from conditional operators (for obvious reasons) just return `PL_op->op_next`, the execution order through a program can be found by chasing the trail of `op_next` pointers from the start node to the root.

We can trace the execution order in several ways: if Perl is built with debugging, then we can say

```
perl -Dt -e 'open ...'
```

Alternatively, and perhaps more simply, the compiler module `B::Terse` (see Chapter 7) has an option to print the execution order, `-exec`. For instance, in our "open-print-close" example above, the execution order is:

```
% perl -MO=Terse,-exec -e 'open FILE, "foo"; ...'
OP (0x8111510) enter
COP (0x81121c8) nextstate
OP (0x8186f30) pushmark
SVOP (0x8186fe0) gv  GV (0x8111bd8) *FILE
SVOP (0x8186f10) const  PV (0x810dd98) "foo"
LISTOP (0x810a170) open [1]
COP (0x81114d0) nextstate
OP (0x81114b0) pushmark
SVOP (0x8118318) gv  GV (0x8111bd8) *FILE
UNOP (0x8111468) rv2gv
SVOP (0x8111448) const  PV (0x8111bfc) "hi\n"
LISTOP (0x8111488) print
COP (0x8111fe0) nextstate
SVOP (0x8111fc0) gv  GV (0x8111bd8) *FILE
UNOP (0x8111fa0) close
LISTOP (0x8111420) leave [1]
```

This program, just like every other program, starts with the `enter` and `nextstate` ops to enter a scope and begin a new statement respectively. Then a mark is placed on the argument stack: marks represent the start of a set of arguments, and a list operator can retrieve all the arguments by pushing values off the stack until it finds a mark. Hence, we're notifying Perl of the beginning of the arguments to the `open` operator.

The arguments in this case are merely the file handle to be opened and the file name; after operators put these two arguments on the stack, `open` can be called. This is the end of the first statement.

Next, the arguments to `print` begin. This is slightly more tricky, because while `open` can only take a true filehandle, `print` may take any sort of reference. Hence, `gv` returns the GV and then this is turned into the appropriate filehandle type by the `rv2gv` operator. After the filehandle come the arguments to be printed; in this case, a constant (`"hi\n"`). Now all the arguments have been placed on the stack, `print` can be called. This is the end of the second statement.

Finally, a filehandle is put on the stack and closed. Note that at this point, the connections between the operators - unary, binary, etc. - are not important; all manipulation of values comes not by looking at the children of the operators but by looking at the stack. The types of op are important for the construction of the tree in "tree order", but the stack and the `op_next` pointers are the only important things for the execution of the tree in execution order.

> **ADVANCED:** How is the execution order determined? The function `linklist` in `op.c` takes care of threading the `op_next` pointers in prefix order. It does so by recursively applying the following rule:
>
> - If there is a child for the current operator, visit the child first, then its siblings, then the current op.
>
> Hence, the starting operator is always the first child of the root operator, (always `enter`) the second op to be executed is its sibling, `nextstate`, and then the children of the next op are visited. Similarly, the root itself (`leave`) is always the last operator to be executed. Null operators are skipped over during optimization.

6.2. PP Code

We know the order of execution of the operations, and what some of them do. Now it’s time to look at how they’re actually implemented - the source code inside the

interpreter that actually carries out `print`, `+`, and other operations.

The functions which implement operations are known as "PP Code" - "Push / Pop Code" - because most of their work involves popping off elements from a stack, performing some operation on it, and then pushing the result back. PP code can be found in several files: `pp_hot.c` contains frequently used code, put into a single object to encourage CPU caching; `pp_ctl.c` contains operations related to flow control; `pp_sys.c` contains the system-specific operations such as file and network handling; `pack` and `unpack` recently moved to `pp_pack.c`, and `pp.c` contains everything else.

6.2.1. The argument stack

We've already talked a little about the argument stack. The Perl interpreter makes use of several stacks, but the argument stack is the main one.

The best way to see how the argument stack is used is to watch it in operation. With a debugging build of Perl, the `-Ds` command line switch prints out the contents of the stack in symbolic format between operations. Here is a portion of the output of running **`$a=5; $b=10; print $a+$b;`**:

```
(-e:1)  nextstate
    =>
(-e:1)  pushmark
    =>  *
(-e:1)  gvsv(main::a)
    =>  *  IV(5)
(-e:1)  gvsv(main::b)
    =>  *  IV(5)  IV(10)
(-e:1)  add
    =>  *  IV(15)
(-e:1)  print
    =>  SV_YES
```

At the beginning of a statement, the stack is typically empty. First, Perl pushes a mark onto the stack to know when to stop pushing off arguments for `print`. Next, the values of `$a` and `$b` are retrieved and pushed onto the stack.

The addition operator is a binary operator, and hence, logically, it takes two values off the stack, adds them together and puts the result back onto the stack. Finally, `print` takes all of the values off the stack up to the previous bookmark and prints them out. Let's not forget that `print` itself has a return value, the true value `SV_YES` which it pushes back onto the stack.

6.2.2. Stack manipulation

Let's now take a look at one of the PP functions, the integer addition function `pp_i_add`. The code may look formidable, but it's a good example of how the PP functions manipulate values on the stack.

```
PP(pp_i_add)                                                    ❶
{
    dSP; dATARGET; tryAMAGICbin(add,opASSIGN);                  ❷
    {
      dPOPTOPiirl_ul;                                           ❸
      SETi( left + right );                                     ❹
      RETURN;                                                   ❺
    }
}
```

❶ In case you haven't guessed, *everything* in this function is a macro. This first line declares the function `pp_i_add` to be the appropriate type for a PP function.

❷ Since following macros will need to manipulate the stack, the first thing we need is a local copy of the stack pointer, `SP`. And since this is C, we need to declare this in advance: `dSP` declares a stack pointer. Then we need an SV to hold the return value, a "target". This is declared with `dATARGET`; see Section 6.4 for more on how targets work. Finally, there is a chance that the addition operator has been overloaded using the `overload` pragma. The `tryAMAGICbin` macro tests to see if it is appropriate to perform "A" (overload) magic on either of the scalars in a binary operation, and if so, does the addition using a magic method call.

❸ We will deal with two values, `left` and `right`. The `dPOPTOPiirl_ul` macro pops two SVs off the top of the stack, converts them to two integers (hence `ii`) and stores them into automatic variables `right` and `left`. (hence `rl`)

> **ADVANCED:** The `_ul`? Look up the definition in `pp.h` and work it out...

❹ We add the two values together, and set the integer value of the target to the result, pushing the target to the top of the stack.

❺ As mentioned above, operators are expected to return the next op to be executed, and in most cases this is simply the value of `op_next`. Hence `RETURN` performs a normal return, copying our local stack pointer `SP` which we obtained above back into the global stack pointer variable, and then returning the `op_next`.

As you might have guessed, there are a number of macros for controlling what happens to the stack; these can be found in `pp.h`. The more common of these are:

`POPs`

Pop an SV off the stack and return it.

`POPpx`

Pop a string off the stack and return it. (Note: requires a variable "STRLEN `n_a`" to be in scope.)

`POPn`

Pop an NV off the stack.

`POPi`

Pop an IV off the stack.

`TOPs`

Return the top SV on the stack, but do not pop it. (The macros `TOPpx`, `TOPn`, etc. are analogous)

`TOPm1s`

Return the penultimate SV on the stack. (There is no `TOPm1px`, etc.)

`PUSHs`

Push the scalar onto the stack; you must ensure that the stack has enough space to accommodate it.

`PUSHn`

Set the NV of the target to the given value, and push it onto the stack. `PUSHi`, etc. are analogous.

There is also an `XPUSHs`, `XPUSHn`, etc. which extends the stack if necessary.

`SETs`

This sets the top element of the stack to the given SV. `SETn`, etc. are analogous.

`dTOPss`, `dPOPss`

These declare a variable called `sv`, and either return the top entry from the stack or pop an entry and set `sv` to it.

`dTOPnv`, `dPOPnv`

These are similar, but declare a variable called `value` of the appropriate type. `dTOPiv` and so on are analogous.

In some cases, the PP code is purely concerned with rearranging the stack, and the PP function will call out to another function in `doop.c` to actually perform the relevant operation.

6.3. The opcode table and `opcodes.pl`

The header files for the opcode tables are generated from a Perl program called `opcode.pl`. Here is a sample entry for an op:

```
index             index                        ck_index isT@     S S S?
```

The entry is in five columns.

The first column is the internal name of the operator. When `opcode.pl` is run, it will create an enum including the symbol `OP_INDEX`.

The second column is the English description of the operator which will be printed during error messages.

The third column is the name of the "check" function which will be used to optimize this tree; see Section 6.5.

Then come additional flags plus a character which specifies the "flavour" of the op: in this case, `index` is a list op, since it can take more than two parameters, so it has the symbol `@`.

Finally, the "prototype" for the function is given: `S S S?` translates to the Perl prototype `$$;$`, which is indeed the prototype for `CORE::index`.

While most people will never need to edit the op table, it is as well to understand how Perl "knows" what the ops look like. There is a full description of the format of the table, including details of the meanings of the flags, in `opcodes.pl`.

6.4. Scatchpads and Targets

PP code is the guts of Perl execution, and hence is highly optimized for speed. One thing that you don't want to do in time-critical areas is create and destroy SVs, because allocating and freeing memory is a slow process. So Perl allocates for each op a *target* SV which is created at compile time. We've seen above that PP code gets the target and uses the `PUSH` macros to push the target onto the stack.

Targets live on the scratchpad, just like lexical variables. `op_targ` for an op is an offset in the current pad; it is the element number in the pad's array which stores the SV that should be used as the target. Perl arranges that ops can reuse the same target if they are not going to collide on the stack; similarly, it also directly uses lexical variables on the pad as targets if appropriate instead of going through a `padsv` operation to extract them. (This is a standard compiler technique called "binding".)

You can tell if an SV is a target by its flags: targets (also known as temporaries) have the `TEMP` flag set, and SVs bound to lexical variables on the pad have the `PADMY` flag set.

6.5. The Optimizer

Between compiling the op tree and executing it, Perl goes through three stages of optimization.

The first stage actually happens as the tree is being constructed. Once Perl creates an op, it passes it off to a check routine. We saw above how the check routines are assigned to operators in the op table; an `index` op will be passed to `ck_index`. This routine may manipulate the op in any way it pleases, including freeing it, replacing it with a different op, or adding new ops above or below it. They are sometimes called in a chain: for instance, the check routine for `index` simply tests to see if the string being sought is a constant, and if so, performs a Fast Boyer-Moore string compilation to speed up the search at runtime; then it calls the general function-checking routine `ck_fun`.

Secondly, the constant folding routine `fold_constants` is called if appropriate. This tests to see whether all of the descendents of the op are constants, and if they are, runs the operator as if it was a little program, collects the result and replaces the op with a

constant op reflecting that result. You can tell if constants have been folded by using the "deparse" compiler backend (see Section 7.2.3):

```
 % perl -MO=Deparse -e 'print (3+5+8+$foo)'
print 16 + $foo;
```

Here, the `3+5` has been constant-folded into `8`, and then `8+8` is constant-folded to 16.

Finally, the peephole optimizer `peep` is called. This examines each op in the tree in execution order, and attempts to determine "local" optimizations by "thinking ahead" one or two ops and seeing if multiple operations can be combined into one. It also checks for lexical issues such as the effect of `use strict` on bareword constants.

6.6. Summary

Perl's fundamental operations are represented by a series of structures, analogous to the structures which make up Perl's internal values. These ops are threaded together in two ways - firstly, into an op tree during the parsing process, where each op dominates its arguments, and secondly, by a thread of execution which establishes the order in which Perl has to run the ops.

To run the ops, Perl uses the code in `pp*.c`, which is particularly macro-heavy. Most of the macros are concerned with manipulating the argument stack, which is the means by which Perl passes data between operations.

Once the op tree is constructed, there are a number of means by which it is optimized - check routines and constant folding which takes place after each op is created, and a peephole optimizer which performs a "dry run" over the execution order.

6.7. Exercises

1. The function `op_null` turns an op into a null op. Find all the occasions in which a null op is constructed, and explain in each case why the op has been nullified.
2. Explain what is going on in the bottom half of `Perl_utilize`. (after the comment `Fake up an import/unimport`)
3. Add a check for the range operator - if both sides are constant, ensure that the left is less than the right.

Chapter 7. The Perl Compiler

7.1. What is the Perl Compiler?

In 1996, someone

* *(I think it was Chip. Must check.)*

announced a challenge - the first person to write a compiler suite for Perl would win a laptop. Malcolm Beattie stepped up to the challenge, and won the laptop with his `B` suite of modules. Many of these modules have now been brought into the Perl core as standard modules.

The Perl compiler is not just for compiling Perl code to a standalone executable - in fact, some would argue that it's not *at all* for compiling Perl into a standalone executable. We've already seen the use of the `B::Terse` and `B::Tree` modules to help us visualise the Perl op tree, and this should give us a hint as to what the Perl compiler is actually all about.

The compiler comes in three parts: a frontend module, `O`, which does little other than turn on Perl's `-c` (compile only, do not run) flag, and loads up a backend module, such as `B::Terse` which performs a specific compiler task, and the `B` module which acts as a low-level driver.

The `B`, at the heart of the compiler, is a stunningly simple XS module which makes Perl's internal object-like structures - SVs, ops, and so on - into real Perl-space objects. This provides us with a degree of introspection: we can, for instance, write a backend module which traverses the op tree of a compiled program and dump out its state to a file. (This is exactly what the `B::Bytecode` module does.)

It's important to know what the Perl compiler is not. It's not something which will magically make your code go faster, or take up less space, or be more reliable. The backends which generate standalone code generally do exactly the opposite. All the compiler is, essentially, is a way of getting access to the op tree and doing something

potentially interesting with it. Let's now take a look at some of the interesting things that can be done with it.

7.2. B:: Modules

There are twelve backend modules to the compiler in the Perl core, and many more besides on CPAN. Here we'll briefly examine those which are particularly helpful to internals hackers or particularly interesting.

7.2.1. B::Concise

`B::Concise` was written quite recently by Stephen McCamant to provide a generic way of getting concise information about the op tree. It is highly customizable, and can be used to emulate `B::Terse` and `B::Debug`. (see below)

Here's the basic output from `B::Concise`:

```
    % perl -MO=Concise -e 'print $a+$b'
1r <@> leave[t1] vKP/REFC ->(end)
1k    <0> enter ->1l
1l    <;> nextstate(main 7 -e:1) v ->1m
1q    <@> print vK ->1r
1m       <0> pushmark s ->1n
1p       <2> add[t1] sK/2 ->1q
-           <1> ex-rv2sv sK/1 ->1o
1n             <$> gvsv(*a) s ->1o
-           <1> ex-rv2sv sK/1 ->1p
1o             <$> gvsv(*b) s ->1p
```

Each line consists of five main parts:

- a label for this operator (in this case, `1r`)

- a type signifier (`@` is a list operator - think arrays)
- the name of the op and its target, if any, plus any other information about it
- the flags for this operator. Here, `v` signifies void context and `K` shows that this operator has children. The private flags are shown after the slash, and are written out as a longer abbreviation than just one character: `REFC` shows that this op is refcounted.
- finally, the label for the next operator in the tree, if there is one.

Note also that, for instance, ops which have been optimized away to a null are left as "ex-...". The exact meanings of the flags and the op classes are given in the `B::Concise` documentation:

```
=head2 OP flags abbreviations

    v      OPf_WANT_VOID    Want nothing (void context)
    s      OPf_WANT_SCALAR  Want single value (scalar context)
    l      OPf_WANT_LIST    Want list of any length (list context)
    K      OPf_KIDS         There is a firstborn child.
    P      OPf_PARENS       This operator was parenthesized.
                             (Or block needs explicit scope entry.)
    R      OPf_REF          Certified reference.
                             (Return container, not containee).
    M      OPf_MOD          Will modify (lvalue).
    S      OPf_STACKED      Some arg is arriving on the stack.
    *      OPf_SPECIAL      Do something weird for this op (see op.h)

=head2 OP class abbreviations

    0      OP (aka BASEOP)  An OP with no children
    1      UNOP             An OP with one child
    2      BINOP            An OP with two children
    |      LOGOP            A control branch OP
    @      LISTOP           An OP that could have lots of children
    /      PMOP             An OP with a regular expression
```

```
    $      SVOP            An OP with an SV
    "      PVOP            An OP with a string
    {      LOOP            An OP that holds pointers for a loop
    ;      COP             An OP that marks the start of a state-
ment
```

As with many of the debugging `B::` modules, you can use the `-exec` flag to walk the op tree in execution order, following the chain of `op_next`'s from the start of the tree:

```
    % perl -MO=Concise,-exec -e 'print $a+$b'
1k <0> enter
1l <;> nextstate(main 7 -e:1) v
1m <0> pushmark s
1n <$> gvsv(*a) s
1o <$> gvsv(*b) s
1p <2> add[t1] sK/2
1q <@> print vK
1r <@> leave[t1] vKP/REFC
-e syntax OK
```

Amongst other options, (again, see the documentation) `B::Concise` supports a `-tree` option for tree-like ASCII art graphs, and the curious but fun `-linenoise` option.

7.2.2. `B::Debug`

`B::Debug` dumps out *all* of the information in the op tree; for anything bigger than a trivial program, this is just way too much information. Hence, to sensibly make use of it, it's a good idea to go through with `B::Terse` or `B::Concise` first, and find which ops you're interested in, and then grep for them.

Some output from `B::Debug` looks like this:

```
LISTOP (0x81121a8)
```

```
        op_next         0x0
        op_sibling      0x0
        op_ppaddr       PL_ppaddr[OP_LEAVE]
        op_targ         1
        op_type         178
        op_seq          6433
        op_flags        13
        op_private      64
        op_first        0x81121d0
        op_last         0x8190498
        op_children     3
OP (0x81121d0)
        op_next         0x81904c0
        op_sibling      0x81904c0
        op_ppaddr       PL_ppaddr[OP_ENTER]
        op_targ         0
        op_type         177
        op_seq          6426
        op_flags        0
        op_private      0
```

As you should know from the ops chapter, this is all the information contained in the op structure: the type of op and its address, the ops related to it, the C function pointer implementing the PP function, the target on the scratchpad this op uses, its type, sequence number, and public and private flags. It also does similar dumps for SVs. You may find the `B::Flags` module useful for "Englishifying" the flags.

7.2.3. `B::Deparse`

`B::Deparse` takes a Perl program and turns it into a Perl program. This doesn't sound very impressive, but it actually does so by decompiling the op tree back into Perl. While this has interesting uses for things like serializing subroutines, it's interesting for

internals hackers because it shows us how Perl understands certain constructs. For instance, we can see that logical operators and binary "if" are equivalent:

```
%  perl -MO=Deparse -e '$a and do {$b}'
if ($a) {
    do {
        $b;
    };
}
-e syntax OK
```

We can also see, for instance, how the magic that is added by command line switches goes into the op tree:

```
% perl -MO=Deparse -ane 'print'
LINE: while (defined($_ = <ARGV>)) {
    @F = split(" ", $_, 0);
    print $_;
}
-e syntax OK
```

7.3. What B and O Provide

To see how we can built compilers and introspective modules with `B`, we need to see what `B` and the compiler front-end `O` give us. We'll start with `O`, since it's simpler.

7.3.1. O

The guts of the `O` module are very small - only 48 lines of code - because all it intends to do is set up the environment ready for a back-end module. The back-ends are expected to provide a subroutine called `compile` which processes the options that are

passed to it and then returns a subroutine reference which does the actual compilation. `O` then calls this subroutine reference in a CHECK block.

CHECK blocks were specifically designed for the compiler - they're called after Perl has finished constructing the op tree and before it starts running the code. `O` calls the `B` subroutine `minus_c` which, as its name implies, is equivalent to the command-line `-c` flag to perl: compile but do not execute the code. It then ensures that any BEGIN blocks are accessible to the back-end modules, and then calls `compile` from the back-end processor with any options from the command line.

7.3.2. B

As we have mentioned, the B module allows Perl-level access to ops and internal variables. There are two key ways to get this access: from the op tree, or from a user-specified Perl "thing".

To get at the op tree, `B` provides the `main_root` and `main_start` functions. These return `B::OP`-derived objects representing the root of the op tree and the start of the tree in execution order respectively:

```
% perl -MB -le 'print B::main_root; print B::main_start'
B::LISTOP=SCALAR(0x8104180)
B::OP=SCALAR(0x8104180)
```

For everything else, you can use the `svref_2object` function which turns some kind of reference into the appropriate `B::SV`-derived object:

```
% perl -MB -l
  $a = 5; print B::svref_2object(\$a);
  @a=(1,2,3); print B::svref_2object(\@a);

  B::IV=SCALAR(0x811f9b8)
B::AV=SCALAR(0x811f9b8)
```

(Yes, it's normal that the objects will have the same addresses.)

In this tutorial we'll concentrate on the op-derived classes, because they're the most useful feature of `B` for compiler construction; the SV classes are a lot simpler and quite analogous.

7.4. Using `B` for Simple Things

OK, so now we have the objects - what can we do with them? `B` provides accessor methods similar to the fields of the structures in `op.h` and `sv.h`. For instance, we can find out the type of the root op like this:

```
$op=B::main_root; print $op->type;
178
```

Oops: `op_type` is actually an enum, so we can't really get much from looking at that directly; however, `B` also gives us the `name` method, which is a little friendlier:

```
$op=B::main_root; print $op->name;
leave
```

We can also use `flags`, `private`, `targ`, and so on - in fact, everything we saw prefixed by `op_` in the `B::Debug` example above.

What about traversing the op tree, then? You should be happy to learn that `first`, `sibling`, `next` and friends return the `B::OP` object for the related op. That's to say, you can follow the op tree in execution order by doing something like this:

```
#!/usr/bin/perl -cl
use B;
CHECK {
  $op=B::main_start;
  print $op->name while $op=$op->next;
}
```

```
print $a+$b;
...
```

Except that's not quite there; when you get to the last op in the sequence, the "enter" at the root of the tree, `op_next` will be a null pointer. `B` represents a null pointer by the `B::NULL` object, which has no methods. This has the handy property that if `$op` is a `B::NULL`, then `$$op` will be zero. So we can print the name of each op in execution order by saying:

```
  $op=B::main_start;
  print $op->name while $op=$op->next and $$op;
```

Walking the tree in normal order is a bit more tricky, since we have to make the right moves appropriate for each type of op: we need to look at both `first` and `last` links from binary ops, for instance, but only the `first` from a unary op. Thankfully, `B` provides a function which does this all for us: `walkoptree_slow`. This arranges to call a user-specified method on each op in turn. Of course, to make it useful, we have to define the method...

```
#!/usr/bin/perl -cl
use B;
CHECK {
      B::walkoptree_slow(B::main_root, "print_it", 0);
      sub B::OP::print_it { my $self = shift; print $self->name }
}

print $a+$b;
...
```

Since all ops inherit from `B::OP`, this duly produces:

```
leave
```

```
enter
nextstate
print
pushmark
add
null
gvsv
null
gvsv
```

We can also use the knowledge that `walkoptree_slow` passes the recursion level as a parameter to the callback method, and prettify the tree a little, like this:

```
      sub B::OP::print_it {
          my ($self,$level)=@_;
          print "    "x$level, $self->name
      }
```

```
leave
    enter
    nextstate
    print
        pushmark
        add
            null
                gvsv
            null
                gvsv
```

See how we're starting to approximate `B::Terse`? Actually, `B::Terse` uses the `B::peekop` function, a little like this:

```
      sub B::OP::print_it {
```

```
        my ($self,$level)=@_;
        print "    "x$level, B::peekop($self);
    }
```

```
LISTOP (0x81142c8) leave
    OP (0x81142f0) enter
    COP (0x8114288) nextstate
    LISTOP (0x8114240) print
        OP (0x8114268) pushmark
        BINOP (0x811d920) add
            UNOP (0x8115840) null
                SVOP (0x8143158) gvsv
            UNOP (0x811d900) null
                SVOP (0x8115860) gvsv
```

All that's missing is that `B::Terse` provides slightly more information based on each different type of op, and that can be easily done by putting methods in the individual op classes: `B::LISTOP`, `B::UNOP` and so on.

Let's finish off our little compiler - let's call it `B::Simple` - by turning it into a module that can be used from the `O` front-end. This is easy enough to do in our case, once we remember that `compile` has to return a callback subroutine reference:

```
package B::Simple;
use B qw(main_root peekop walkoptree_slow);

sub B::OP::print_it {
    my ($self,$level)=@_;
    print "    "x$level, peekop($self);
}

sub compile {
    return sub { walkoptree_slow(main_root, "print_it", 0); }
}
```

```
1;
```

If we save the above code as `B/Simple.pm`, we can run it on our own programs with **`perl -MO=Simple ...`**. We have a backend compiler module!

7.5. Summary

In this chapter, we've examined the basics of the Perl compiler: its front-end `O`, the nuts-and-bolts module `B`, and how to write both backend modules using these. Writing compiler modules is really an excellent way to learn about how the Perl op tree fits together and what the operations signify, so you are encouraged to complete at least some of the following exercises.

7.6. Exercises

1. Examine the code to `B::Bblock`. As the documentation says, " A basic block is a series of operations which is known to execute from start to finish, with no possiblity of branching or halting."

   If there are any ops following a `return` operation inside a basic block, they will never be executed and can be considered dead code. Write a module which detects and reports this type of dead code.

2. Add line numbers to your dead code reporting module by examining the nearest CV.
3. Extend the module to report unreachable code after unconditional `next` and `last` statements.

1. Write a module which tracks variable access and reports dead code based on it. For instance, Perl can optimize

```
#!/usr/bin/perl
if (0) {
    ...
}
```

   away to nothing, but cannot currently remove

```
#!/usr/bin/perl
$foo=0;
if ($foo) {
    ...
}
```

2. Write a module which describes other optimizations that can be made. For instance, given

```
$a = $x * $y;
$b = $x * $y;
```

   optimize to

```
$b = $a = $x * $y;
```

Appendix A. Unix cheat sheet

A brief run-down for those whose Unix skills are rusty:

Table A-1. Simple Unix commands

Action	Command
Change to home directory	**cd**
Change to *directory*	**cd *directory***
Change to directory above current directory	**cd ..**
Show current directory	**pwd**
Directory listing	**ls**
Wide directory listing, showing hidden files	**ls -al**
Showing file permissions	**ls -al**
Making a file executable	**chmod +x *filename***
Printing a long file a screenful at a time	**more *filename*** or **less *filename***
Getting help for *command*	**man *command***

Appendix B. Editor cheat sheet

This summary is laid out as follows:

Table B-1. Layout of editor cheat sheets

Running	Recommended command line for starting it.
Using	Really basic howto. This is not even an attempt at a detailed howto.
Exiting	How to quit.
Gotchas	Oddities to watch for.

B.1. vi

B.1.1. Running

```
% vi filename
```

B.1.2. Using

- `i` to enter insert mode, then type text, press **ESC** to leave insert mode.
- `x` to delete character below cursor.
- `dd` to delete the current line
- Cursor keys should move the cursor while *not* in insert mode.
- If not, try `hjkl`, `h` = left, `l` = right, `j` = down, `k` = up.

- `/`, then a string, then **ENTER** to search for text.
- `:w` then **ENTER** to save.

B.1.3. Exiting

- Press **ESC** if necessary to leave insert mode.
- `:q` then **ENTER** to exit.
- `:q!` **ENTER** to exit without saving.
- `:wq` to exit with save.

B.1.4. Gotchas

vi has an insert mode and a command mode. Text entry only works in insert mode, and cursor motion only works in command mode. If you get confused about what mode you are in, pressing **ESC** twice is guaranteed to get you back to command mode (from where you press i to insert text, etc).

B.1.5. Help

`:help` **ENTER** might work. If not, then see the manpage.

B.2. pico

B.2.1. Running

```
% pico -w filename
```

B.2.2. Using

- Cursor keys should work to move the cursor.
- Type to insert text under the cursor.
- The menu bar has `^X` commands listed. This means hold down **CTRL** and press the letter involved, eg **CTRL-W** to search for text.
- **CTRL-O**to save.

B.2.3. Exiting

Follow the menu bar, if you are in the midst of a command. Use **CTRL-X** from the main menu.

B.2.4. Gotchas

Line wraps are automatically inserted unless the -w flag is given on the command line. This often causes problems when strings are wrapped in the middle of code and similar. \\ \hline

B.2.5. Help

CTRL-G from the main menu, or just read the menu bar.

B.3. joe

B.3.1. Running

```
% joe filename
```

B.3.2. Using

- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-K** then **S** to save.

B.3.3. Exiting

- **CTRL-C** to exit without save.
- **CTRL-K** then **X** to save and exit.

B.3.4. Gotchas

Nothing in particular.

B.3.5. Help

CTRL-K then **H**.

B.4. jed

B.4.1. Running

```
% jed
```

B.4.2. Using

- Defaults to the emacs emulation mode.
- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-X** then **S** to save.

B.4.3. Exiting

CTRL-X then **CTRL-C** to exit.

B.4.4. Gotchas

Nothing in particular.

B.4.5. Help

- Read the menu bar at the top.
- Press **ESC** then **?** then **H** from the main menu.

Appendix C. ASCII Pronunciation Guide

Table C-1. ASCII Pronunciation Guide

Character	Pronunciation
!	bang, exlamation
*	star, asterisk
$	dollar
@	at
%	percent
&	ampersand
"	double-quote
'	single-quote, tick
( )	open/close bracket, parentheses
<	less than
>	greater than
-	dash, hyphen
.	dot
,	comma
/	slash, forward-slash
\	backslash, slosh
:	colon
;	semi-colon
=	equals
?	question-mark
^	caret (pron. carrot)
_	underscore
[ ]	open/close square bracket
{ }	open/close curly brackets, open/close brace

Character	Pronunciation
|	pipe, or vertical bar
~	tilde (pron. “til-duh”, wiggle, squiggle)
‘	backtick