GNU poke Manual

for version 2.4, 25 July 2022

by Jose E. Marchesi et al.

This manual describes GNU poke (version 2.4, 25 July 2022).

Table of Contents

1 Introduction

1.1 Motivation

The main purpose of GNU poke is to manipulate structured binary data in terms of abstractions provided by the user. The Poke type definitions can be seen as a sort of declarative specifications for decoding and encoding procedures. The user specifies the structure of the data to be manipulated, and poke uses that information to automagically decode and encode the data. Under this perspective, struct types correspond to sequences of instructions, array types to repetitions or loops, union types to alternatives or conditionals, and so on.

1.1.1 Decode-Compute-Encode

Computing with data whose form is not the most convenient way to be manipulated, like is often the case in unstructured binary data, requires performing a preliminary step that transforms the data into a more convenient representation, usually featuring a higher level of abstraction. This step is known in computer jargon as *unmarshalling*, when the data is fetch from some storage or transmission media or, more generally, *decoding*.

Once the computation has been performed, the result should be transformed back to the low-level representation to be stored or transmitted. This is performed in a closing step known as *marshalling* or, more generally, *encoding*.

Consider the following C program whose purpose is to read a 32-bit signed integer from a byte-oriented storage media at a given offset, multiply it by two, and store the result at the same offset.

```
void double_number (int fd, off_t offset, int endian)
{
   int number, i;
   unsigned char b[4];

   /* Decode.  */
   lseek (fd, offset, SEEK_SET);
   for (i = 0; i < 4; ++i)
      read (fd, &b[i], 1);

   if (endian == BIG)
     number = b[0] << 24 | b[1] << 16 | b[2] << 8 | b[3];
   else
     number = b[3] << 24 | b[2] << 16 | b[1] << 8 | b[0];

   /* Compute.  */
   number = number * 2;

   /* Encode.  */
   if (endian == BIG)
   {
     b[0] = (number >> 24) & 0xff;
     b[1] = (number >> 16) & 0xff;
     b[2] = (number >> 8) & 0xff;
     b[3] = number & 0xff;
   }
   else
   {
```

```
      b[3] = (number >> 24) & 0xff;
      b[2] = (number >> 16) & 0xff;
      b[1] = (number >> 8) & 0xff;
      b[0] = number & 0xff;
    }

    lseek (fd, offset, SEEK_SET);
    for (i = 0; i < 4; ++i)
       write (fd, &b[i], 1);
}
```

As we can see, decoding takes care of fetching the data from the storage in simple units, bytes. Then it mounts the more abstract entity on which the computation will be performed, in this case a signed 32-bit integer. Considerations like endianness, negative encoding (which is assumed to be two's complement in this example and handled automatically by C) and error conditions (omitted in this example for clarity) should be handled properly.

Conversely, encoding turns the signed 32-bit integer into a sequence of bytes and then writes them out to the storage at the desired offset. Again, this requires taking endianness into account and handling error conditions.

This example may look simplistic and artificial, and it is, but too often the computation proper (like multiplying the integer by two) is way more straightforward than the decoding and encoding of the data used for the computation.

Generally speaking, decoding and encoding binary data is laborious and error prone. Think about sequences of elements, variable-length and clever compact encodings, elements not aligned to byte boundaries, the always bug-prone endianness, and a long *etc.* Dirty business, sometimes risky, and always *boring*.

1.1.2 Describe-Compute

This is where poke comes into play. Basically, it allows you to describe the characteristics of the data you want to compute on, and then decodes and encodes it for you, taking care of the gory details. That way you can concentrate your energy on the *fun* part: computing on the data at your pleasure.

Of course, you are still required to provide a description of the data. In the Poke language, these descriptions take the form of *type definitions*, which are *declarative*: you specify *what* you want, and poke extracts the *how* from that.

For example, consider the following Poke type definition:

```
type Packet =
  struct
  {
    uint<16> magic = 0xef;
    uint<32> size;
    byte[size] data @ 8#B;
  };
```

This tells poke that, in order to decode a `Packet`, it should perform the following procedure (a similar procedure is implied for encoding):

– Read two bytes from the IO space, mount them into an unsigned 16-bit integer using whatever current endianness, and put it in `magic`. If this unsigned 16-bit integer doesn't equal to `0xef`, then stop and emit a "data integrity" error.

– Read four bytes, mount them into an unsigned 32-bit integer using the same endianness, and put it in `size`.

- Seek the IO space to advance 16 bits.
- Do `size` times:
  - Read one byte and mount it into an unsigned 8-bit integer.
  - Put the integer in the proper place in the `data` array.

If during this procedure an end of file is encountered, or some other erroneous condition happens, an appropriate error is raised.

In the procedure sketched above we find a sequence of operations, implied by the struct type, and a loop, implied by the array type. As we shall see later in this book, it is also possible to decode conditionally. Union types are used for that purpose.

1.2 Nomenclature

GNU poke is a new program and it introduces many a new concept. It is a good idea to clarify how we call things in the poke community. Unless everyone uses the same nomenclature to refer to pokish thingies, it is gonna get very confusing very soon!

First of all we have `poke`, the program. Since "poke" is a very common English word, when the context is not clear we either use the full denomination `GNU poke`, or quote the word using some other notation.

Then we have *Poke*, with upper case P, which is the name of the domain-specific programming language implemented by `poke`, the program.

This distinction is important. For example, when people talk about "poke programmers" they refer to the group of people hacking GNU poke. When they talk about "Poke programmers" they refer to the people who write programs using the Poke programming language.

Finally, a *pickle* is a Poke source file containing definitions of types, variables, functions, *etc*, that conceptually apply to some definite domain. For example, `elf.pk` is a pickle that provides facilities to poke ELF object files. Pickles are not necessarily related to file formats: a set of functions to work with bit patterns, for example, could be implemented in a pickle named `bitpatterns.pk`.

We hope this helps to clarify things.

1.3 Invoking poke

Synopsis:

```
poke [option...] [file]
```

The following options are available.

'`-l`'
'`--load=`*`file`*'
: Load the given file as a Poke program. Any number of '`-l`' options can be specified, and they are loaded in the given order.

'`-L` *`file`*'
: Load the given file as a Poke program and exit. The rest of the command-line is not processed by poke, and is available to the Poke script in the `argv` variable. This is commonly used along with a shebang (see Section 1.4.5 [Scripts], page 7) to implement Poke scripts.

Commanding poke from the command line:

'`-c`'
'`--command=`*`cmd`*'
: Execute the given command. Any number of '`-c`' options can be specified, and they are executed in the given order.

‘`-s`’
‘`--source=`*file*’
: Load *file* as a command file. Any number of ‘`-s`’ options may be specified, and they are loaded in the given order. See Section 1.4.4 [Command Files], page 6.

Styling text output:

‘`--color=`*how*’
: Whether to use styled output, and how. Valid options for *how* are ‘`yes`’, ‘`no`’, ‘`auto`’, ‘`html`’ and ‘`test`’.

‘`--style-dark`’
: Use the default style that works better for dark backgrounds. This is the default.

‘`--style-bright`’
: Use the default style that works better for bright backgrounds.

‘`--style=`*file*’
: Use *file* as the CSS to use for styling poke, instead of the default style.

Machine interface:

`--mi`
: Run poke in MI mode. In this mode, poke communicates with a client using JSON messages in the standard input and output.

Other options:

`-q`
`--no-init-file`
: Do not load the `~/.pokerc` init file.

`--no-auto-map`
: Do not load map-files automatically when poke opens IO spaces.

`--no-hserver`
: Do not run the terminal hyperlinks server.

`--quiet`
: Be as terse as possible.

`--help`
: Print a help message and exit.

`--version`
: Show version and exit.

1.4 Commanding poke

GNU poke is primarily an interactive editor that works in the command line. However, it is also possible to use it in a non-interactive way. This chapter documents both possibilities.

1.4.1 The REPL

If poke is invoked with an interactive TTY connected to the standard input, it greets you with a welcome message, licensing information and such, and finally a prompt that looks like:

```
(poke)
```

At this point, the program is ready to be commanded. You are expected to introduce a line and press `enter`. At that point poke will examine the command, notify you if there is some error condition, process the line and maybe displaying something in the terminal.

Repeatedly typing complex commands can be tiresome. To help you, poke uses the readline library See *GNU Readline Library*. This provides shortcuts and simple keystrokes to repeat previous commands with or without modification, fast selection of file names and entries from

other multiple choice contexts, and navigation within a command and among previous commands. When the REPL starts, the history of your previous sessions are loaded from the file `.poke_history` located in your home directory (if it exists).

There are several kinds of lines that can be provided in the REPL:

- A *dot-command* invocation, that starts with a dot character (`.`).
- A command invocation.
- A Poke statement.
- A Poke expression.

These are explained in the following sections.

1.4.2 Evaluation

You can evaluate a Poke statement by typing it at the REPL's prompt. Only a single statement, including expression statements (see Section 18.11 [Expression Statements], page 146) and compound statements (see Section 18.8 [Compound Statements], page 145), can be evaluted this way. It needs not be terminated by a semicolon.

When an expression is evaluated, the result of the evaluation is printed back to you. For example:

```
(poke) 23
23
(poke) [1,2,3]
[1,2,3]
(poke) Packet @ 0#B
Packet {i=1179403647,j=65794L}
```

When a statement other than an expression statement is executed in the REPL no result is printed, but of course the statement can print on its own:

```
(poke) fun do_foo = void: {}
(poke) do_foo
(poke) for (i in [1,2,3]) printf "elem %i32d\n", i;
elem 1
elem 2
elem 3
```

If there is an error compiling the line, you are notified with a nice error message, showing the location of the error. For example:

```
(poke) [1,2,3 + "foo"]
<stdin>:1:6: error: invalid operands in expression
[1,2,3 + "foo"];
     ^~~~~~~~~
```

1.4.3 Commands and Dot-Commands

There are two kinds of commands in poke: the *dot-commands*, which are written in C and have their own conventions for handling sub-commands and passing arguments and flags, and normal commands, which are written in Poke.

1.4.3.1 Dot-Commands

Dot-commands are so called because their names start with the dot character (`.`). They can feature subcommands. Example:

```
(poke) .vm disassemble mapper int[] @ 0#B
(poke) .vm disassemble writer int[] @ 0#B
```

When there is no ambiguity, the command name and the subcommands can be shortened to prefixes. The commands above can also be written as:

```
(poke) .vm dis m int[] @ 0#B
(poke) .vm dis w int[] @ 0#B
```

Some commands also get flags, which are one-letter indicators that can be appended to the command name (including subcommands) after a slash character (`/`). For example, the `.vm` disassembler commands accept a `n` flag to indicate we want a native disassemble. We can pass it as follows:

```
(poke) .vm disassemble mapper/n int[] @ 0#B
(poke) .vm disassemble writer/n int[] @ 0#B
```

If a dot-command accepts more than one argument, they are separated using comma characters (`,`). Spaces are generally ignored.

1.4.3.2 Commands

Regular poke commands are written in Poke and use different conventions. The name of commands follow the same rules as normal Poke identifiers, and do not start with a dot character.

An example is the `dump` command:

```
(poke) dump
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 f700 0100 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 8001 0000 0000 0000  ................
00000030: 0000 0000 4000 0000 0000 4000 0800 0700  ....@.....@.....
00000040: 1800 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 7900 0000 0000 0000 b701 0000 9a02 0000  y...............
00000060: 7b10 0000 0000 0000 1800 0000 0000 0000  {...............
00000070: 0000 0000 0000 0000 7900 0000 0000 0000  ........y.......
```

After the name of the command, arguments can be specified by name, like this:

```
(poke) dump :from 0#B :size 8#B
(poke) dump :from 0#B :size 8#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 7f45 4c46 0201 0100                      .ELF....
```

The `dump` command is discussed in greater detail below (see Section 17.1 [dump], page 110). The order of arguments is irrelevant in principle:

```
(poke) dump :from 0#B :size 8#B :ascii 0 :ruler 0
00000000: 7f45 4c46 0201 0100
(poke) dump :ruler 0 :from 0#B :size 8#B :ascii 0
00000000: 7f45 4c46 0201 0100
```

However, beware side effects while computing the values you pass as the arguments! The expressions themselves are evaluated from left to right.

Which arguments are accepted, and their kind, depend on the specific command.

Note that the idea is to restrict the number of dot-commands to the absolutely minimum. Most of the command-like functionality provided in poke shall be implemented as regular commands.

1.4.4 Command Files

Command files contain poke commands. A poke command may be a dot command, a Poke statement or a Poke expression. Lines starting with `#` are comments will be ignored. However a comment must start at the beginning of a line. Here is an example of a script:

```
# The following two lines are dot commands
.load my-pickle.pk
.set obase 16

# The following line is a Poke statement
dump :size 0x100#B :from 0x10#B

# The following line is a Poke expression statement without any side effect.
# Consequently it is valid, but rather useless.
4 == 4
```

A command file contains commands, not Poke code. This means it gets read line by line and commands cannot occupy more than one line. Hence the following is a valid command file:

```
type foo = struct {int this; int that;}
```

but this is not valid as a command file (although it is a valid Poke statement) and will provoke an error:

```
type foo = struct
{
 int this;
 int that;
}
```

Command files can be loaded at startup using the `-s` command line option (see Section 1.3 [Invoking poke], page 3). The `~/.pokerc` startup file is also an example of a poke command file (see Section 8.1 [pokerc], page 93).

1.4.5 Scripts

Following the example of Guile Scheme, the Poke syntax includes support for multi-line comments using the `#!` and `!#` delimiters. This, along with the `-L` command line option, allows to write Poke scripts and execute them in the command line like if they were normal programs. Example of a script:

```
#!/usr/bin/poke -L
!#

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

The resulting script can process command-line options by accessing the `argv` array. The following Poke script prints its arguments:

```
#!/usr/bin/poke -L
!#

for (arg in argv)
  printf ("Argument: %s\n", arg);
```

If you want to pass additional flags to the poke command, you need to use a slightly different kind of shebang:

```
#!/usr/bin/env sh
exec poke -L "$0" "$@"
!#

load elf;
printf ("%v\n", Elf64_Ehdr @ 0#B);
```

2 Setting Up

2.1 Setting up Hyperlinks

GNU poke uses *terminal hyperlinks* in order to improve the interactive usage of the tool: clicking on terminal hyperlinks requests poke to execute certain actions. This is used to implement buttons and other interactive goodies.

Many terminal emulators support terminal hyperlinks. However, we are using a very simple protocol called `app://` that is not supported (yet) on GNU/Linux distros. Fortunately, it is very easy to set your system to use this protocol, and this chapter shows you how.

2.1.1 Make sure your poke speaks hyperlinks

The first step in having an hyperlinks-capable poke is to make sure to have a recent enough version of libtextstyle when building poke. If your poke can emit hyperlinks you will see a message like this when running it on the terminal:

```
hserver listening in port 43713.
```

2.1.2 Use a terminal emulator that supports hyperlinks

Gnome Terminal has support for displaying hyperlinks as do many other emulators that rely on VTE. Check the list at `https://gist.github.com/egmontkob/eb114294efbcd5adb1944c9f3cb5feda#supporting-apps` for a mostly up-to-date, non-exhaustive list of emulators that support printing hyperlinks.

2.1.3 Get and install the app-client utility

Since `app://` is a new URI protocol that we designed, common terminal emulators don't know what to do when they encounter such a URI. To work around this problem we use the XDG Desktop Specification and a little C utility called `app-client`, which can be found at `https://gitlab.com/darnir/hyperlink-app-client`.

By setting `app-client` as the default handler for `app://` URIs, the terminal emulator does not need to understand the syntax or semantics of the `app://` protocol. It offloads the handling of the URI entirely to `app-client`. In order to use this, first download and install `app-client`:

```
$ git clone https://gitlab.com/darnir/hyperlink-app-client
$ cd hyperlink-app-client
$ make
$ cp app-client /location/in/$PATH/variable
```

Next step is to copy the `app-client.desktop` file in the git repository to `$HOME/.local/share/applications`. This is a XDG Desktop Entry for the app-client. And let's most applications on your system know that this should be used to handle `app://` URIs. (See the `MimeType` field)

This is enough for any utility (like terminals) that use `xdg-open` to do the right thing with hyperlinks. However, certain terminals require additional setup. See below if that is your case.

2.1.4 GNOME Terminal

Gnome Terminal doesn't use `xdg-open` to start the applications. Instead, it parses the `mimeapps.list` file manually to find the right application.

Edit your `mimeapps.list`, it is usually located at `$HOME/.local/share/applications/mimeapps.list`, but it might also be at `$XDG_CONFIG_DIR/mimeapps.list`, and add the following line to it:

```
x-scheme-handler/app=app-client.desktop
```

This let's Gnome Terminal know how to open `app://` links.

2.2 Simple Init File

GNU poke is a spartan program that tries to be as simple as possible by default, without fancy displays. Therefore, before exploring poke you may want to configure it minimally. This section contains a few recommendations in that respect.

First we must say that poke reads a per-user configuration from the `~/.pokerc` file. See Section 8.1 [pokerc], page 93.

We recommend new users to set the following options:

```
.set endian little
.set omode tree
.set oacutoff 5
.set pretty-print yes
```

These options are explained later in this manual. See Section 16.13 [set command], page 107.

3 Basic Editing

In this chapter you will learn how to shuffle binary data around with poke, in terms of fundamental predefined entities: bits, bytes, integers, and the like.

3.1 Binary Files

GNU poke is an editor for *binary files*. Right, so what is a binary file? Strictly speaking, every file in a computer's file system is binary. This is because, in a very fundamental level, files are just sequences of bytes.

Colloquially, however, it is very common to talk about "binary files" as opposed to "text files". In this informal meaning, a text file is basically a file composed, mostly, of bytes (and byte sequences) that can be translated into printable characters in some character set, such as ASCII, EBCDIC or Unicode. It follows that binary files would then be files composed, mostly, of bytes not intended to be interpreted as encoded characters.

Some text files contain non-printable characters, such as form feed characters, and many binary files contain printable strings, such as a string table in an ELF object file. That is why we used the word "mostly" in the definitions above. In practice, however, the distinction is almost always clear and there is common consensus on whether a given file format can be considered as a binary format, or not.

GNU poke can edit any file, and as we shall see, it provides some nice features to manipulate sequences of bytes interpreted as character strings. However, it is called a "binary editor" because it is especially designed to be particularly useful editing binary files, in the sense of the term defined above.

In this chapter, we will be using ELF object files as the experiment subject in most of the examples. ELF files are good for this purpose, because they are eminently binary, highly structured, and still strings play a role in them, encoding names of entities like sections and symbols. You don't need to have a perfect knowledge of the ELF format in order to follow the examples, but being familiarized with the concept of object file formats should surely help.

Obtaining a simple ELF object file is easy, if you have a C compiler installed:

```
$ echo 'int foo () { return 0; }' | gcc -c -xc -o foo.o -
```

The command above compiles a very simple ELF object file that contains the compiled form of a little dummy function. This object file will be our companion for a while, and will be the subject of much analysis and abuse, as we poke it.

3.2 Files as IO Spaces

Now that we have a binary file (`foo.o`) it is time to open it with poke. There are two ways to do that.

One way is to pass the name of the file in the poke invocation. The program will start, open the file, and present you with the REPL, like in:

```
$ poke foo.o
[...]
(poke)
```

The other way is to fire up poke without arguments, and then use the `.file` dot-command to open the file:

```
$ poke
[...]
(poke) .file foo.o
The current IOS is now `./foo.o'.
```

```
(poke)
```

Note how poke replies to the dot-command, stating that the *current IOS* is now the file we opened.

You may be wondering, what is this IOS thing? It is an acronym for Input/Output Space, often written IO Space. This is the denomination used to refer to the entities being edited with poke. In this case the IO space being edited is a file, but we will see that is not always the case: poke can also edit other entities such as memory buffers and remote block-oriented devices over the network. For now, let's keep in mind that IOS, or IO space, refers to the file being edited.

And why "current"? GNU poke is capable of editing several files (more generally, several IO spaces) simultaneously. At any time, one of these files is the "current one". In our case, the current IO space is the file `foo.o`, since it is the only file poke knows about:

```
(poke) .info ios
  Id Type Mode Bias Size Name
* #0 FILE rw 0x00000000#B 0x00000398#B ./foo.o
```

The command `.info ios` gives us information about all the IO spaces that are currently open. The first column tells us a *tag* that identifies the IOS. In this example, the tag corresponding to `foo.o` is `#0`. The second column tells us the type of IO space. The third column tells us that `foo.o` allows both reading and writing. The fourth column tells us the size of the file, in hexadecimal.

You may wonder what is that weird suffix `#B`. It is a unit, and tells us that the size `0x398` is measured in bytes, *i.e.* the size of `foo.o` is `0x398` bytes (or, in decimal, `920` bytes.)

Finally, the asterisk character at the left of the entry for `foo.o` identifies it as the current IO space. To see this more clearly, let's open another file:

```
(poke) .file bar.o
The current IOS is now ‘./bar.o’.
(poke) .info ios
  Id Type Mode Bias Size Name
* #1 FILE rw 0x00000000#B 0x00000398#B ./bar.o
  #0 FILE rw 0x00000000#B 0x00000398#B ./foo.o
```

Ah, there we have both `foo.o` and `bar.o`. Now the current IO space (the entry featuring the asterisk at the left) is the file that we just opened, `bar.o`. This is because poke always sets the most recently open file as the current one. We can switch back to `foo.o` using yet another dot-command, `.ios`, which gets an IO space tag as an argument:

```
(poke) .ios #0
The current IOS is now ‘./foo.o’.
(poke) .info ios
  Id Type Mode Bias Size Name
  #1 FILE rw 0x00000000#B 0x00000398#B ./bar.o
* #0 FILE rw 0x00000000#B 0x00000398#B ./foo.o
```

We are back to `foo.o`. Since we are not really interested in `bar.o`, let's close it:

```
(poke) .close #1
(poke) .info ios
  Id Type Mode Bias Size Name
* #0 FILE rw 0x00000000#B 0x00000398#B ./foo.o
```

Awesome. Now we can focus on `foo.o`'s contents...

3.3 Dumping File Contents

Data stored in modern computers, in both volatile memory and persistent files, is fundamentally a sequence of entities called *bytes*. The bytes can be addressed by its position in the sequence, starting with zero:

```
+--------+--------+--------+ ... +--------+
| byte 0 | byte 1 | byte 2 |     | byte N |
+--------+--------+--------+ ... +--------+
```

Each byte has capacity to store a little unsigned integer in the range `0..255`. Therefore, the IO spaces that we edit with poke (like the file `foo.o`) can be seen as a sequence of little numbers, like depicted in the figure above.

GNU poke provides a command whose purpose is to display the values of these bytes: `dump`[1] . It is called like that because it dumps ranges of bytes to the terminal, allowing the user to inspect them.

So let's use our first poke command! Fire up poke, open the file `foo.o` as explained above, and execute the `dump` command:

```
(poke) dump
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 f700 0102 0000 0000 0000 0000 0000  ................
00000020: 0102 0000 0000 0000 9801 0000 0000 0000  ................
00000030: 0000 0000 4000 0000 0000 4000 0800 0700  ..............
00000040: 2564 0a00 0000 0000 0000 0000 0000 0000  %d..............
00000050: b702 0000 0100 0000 1801 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 8510 0000 ffff ffff  ................
00000070: b700 0000 0000 0000 9500 0000 0000 0000  ................
(poke)
```

What are we looking at?

The first line of the output, starting with `76543210`, is a *ruler*. It is there to help us to visually determine the location (or offset) of the data.

The rest of the lines show the values of the bytes that are stored in the file, 16 bytes per line. The first column in these data lines shows the offset, in hexadecimal and measured in number of bytes, from which the row of data starts. For example, the offset of the first byte shown in the third data line has offset 0x20 in the file, the second byte has offset 0x21, and so on. Note how the data rows show the values of the individual bytes, in hexadecimal. Generally speaking, when dealing with bytes (and binary data in general) it is useful to manipulate magnitudes in hexadecimal, or octal. This is because it is easy to group digits in these bases to little groups of bits (four and three respectively) in the equivalent binary representation. In this case, each couple of hexadecimal digits denote the value of a single byte[2]. For example, the value of the first byte in the third data row is 0x01, the value of the second byte 0x02, and so on.

Using the ruler and the column of offsets, locating bytes in the data is very easy. Let's say for example we are interested in the byte at offset 0x68: we use the first column to quickly find the row starting at 0x60, and the ruler to find the column marked with `88`. Cross column and row and... voila! The byte in question has the value 0x85. The reverse process is just as easy. What is the offset of the first 0x40 in the file? Try it!

---

[1] Note that this is not a dot-command like `.file`, `.ios` or `.close`: `dump` does not start with a dot! We will see later how dot-commands differ from "normal commands" like `dump`, but for now, let's ignore the distinction.

[2] Do not be fooled by the fact `dump` shows the hexadecimal digits in groups of four: this is just a visual aid and, as we shall see, it is possible to change the grouping by passing arguments to `dump`.

The section at the right of the output is the ASCII output. It shows the row of bytes at the left interpreted as ASCII characters. Non-printable characters are shown as `.` to avoid scrambling the terminal, and yes, there is actually way to customize what character to use, so they are not confused from real ASCII dot characters (`0x2e`) :P In this particular dump we can see that near the beginning of the file there are three bytes whose value, if interpreted as ASCII characters, conform the string "ELF". As we shall see, this is part of the ELF magic number. Again, the ruler is very useful to locate the byte corresponding to some character in the ASCII section, or the other way around. What is the value of the byte corresponding to the `F` in `ELF`? Try it!

Something to notice in the `dump` output above is that these are not, by any mean, the complete contents of the file `foo.o`. The `.info ios` dot-command informed us in the last section that `foo.o` contains 920 bytes, of which the `dump` command only showed us... `0x80` bytes, or `128` bytes in decimal.

`dump` is certainly capable of showing more (and less) than `128` bytes. We can ask `dump` to display some given amount of data by specifying its size using a *command argument*. For example:

```
(poke) dump :size 64#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 f700 0102 0000 0000 0000 0000 0000  ................
00000020: 0102 0000 0000 0000 9801 0000 0000 0000  ................
00000030: 0000 0000 4000 0000 0000 4000 0800 0700  ..............
```

The command above asks poke to "dump 64 bytes". In this example `:size` is the name of the argument, and `64#B` is the argument's value. Again, the suffix `#B` tells poke we want to dump 64 bytes, not 64 kilobits nor 64 potatoes.

Another interesting aspect of our first dump (ahem) is that the dumped bytes start from the beginning of the file, *i.e.* the offset of the first byte is 0x0. Certainly there should be other areas of the file with interesting contents for us to inspect. To that purpose, we can use yet another option, `:from`:

```
(poke) dump :size 64#B :from 128#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000080: 1400 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000a0: 0000 0000 0300 0100 0000 0000 0000 0000  ................
000000b0: 0000 0000 0000 0000 0000 0000 0300 0300  ................
```

The command above asks poke to "dump 64 bytes starting at 128 bytes from the beginning of the file". Note how the first row of bytes start at offset `0x80`, *i.e.* `128` in decimal.

Passing options to commands is easy and natural, but we may find ourselves passing the same values again and again to certain command options. For example, if the default size of `dump` of 128 bytes is not what you prefer, because you have a particularly tall monitor, or you are one of these people using sub-atomic sized fonts, it can be tiresome and error-prone to pass `:size` to `dump` every time you use it. Fortunately, the default size can be customized by setting a *global variable*:

```
(poke) pk_dump_size = 160#B
```

This tells poke to set `160` bytes as the new value for the `pk_dump_size` variable. This is a global variable that the `dump` command uses to determine how much data to show if the user doesn't specify an explicit value with the `:size` option. Many other commands use the same strategy in order to alter their default behavior, not just `dump`.

And now that we are talking about that, it is also cumbersome to have to set the default size used by `dump` every time we run poke. But no problem, just set the variable in a file called `.pokerc` in your home directory, like this:

```
pk_dump_size = 160#B
```

Every time poke starts, it reads `~/.pokerc` and executes the commands contained in it. See Section 8.1 [pokerc], page 93.

The `dump` command is very flexible, and accepts a lot of options and customization variables that we won't be covering in this chapter. For a complete description of the command, see Section 17.1 [dump], page 110.

3.4 Poking Bytes

Let's look again at the first bytes of the file `foo.o`:

```
(poke) dump :size 64#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 f700 0102 0000 0000 0000 0000 0000  ................
00000020: 0102 0000 0000 0000 9801 0000 0000 0000  ................
00000030: 0000 0000 4000 0000 0000 4000 0800 0700  ............
```

At this point we know how to use the ruler to localize specific bytes just by looking at the displayed data. If we wanted to operate on the values of some given bytes, we could look at the dump and type the values in the REPL. For example, if we wanted to add the values of the bytes at offsets 0x2 and 0x4, we could look at the dump and then type:

```
(poke) 0x4c + 0x02
0x4e
```

GNU poke supports many operators that take integers as arguments, to perform arithmetic, relational, logical and bit-wise operations on them (see Section 18.1 [Integers], page 114). Since bytes are no more (and no less) than little unsigned integers, we can use these operators to perform calculations on bytes.

For example, this is how we would calculate whether the highest bit in the second byte in `foo.o` is set:

```
(poke) 0x45 & 0x80
0
```

Note how booleans are encoded in Poke as integers, 0 meaning false, any other value meaning true.

Looking at the output of `dump` and writing the desired byte value in the prompt is cumbersome. Fortunately, there is a much more convenient way to access the value of a byte, given its offset in the file: it is called *mapping* a byte value. This operation is implemented by a binary operator, called the map operator.

This is how it works. Assuming we were interested in the byte at `64` bytes from the beginning of the file, this is how we would refer to it (or "map" it):

```
(poke) byte @ 64#B
37UB
```

This application of the map operator tells poke to map a byte at the offset 64 bytes. It can be read as "byte at 64 bytes". Note how poke replies with the value 37UB. The suffix `UB` means "unsigned byte", and is an indication for the user about the nature of the preceding number: it is unsigned, and it occupies a byte when stored.

As we can see in this example, poke uses decimal by default when showing values in the REPL. We already noted how it is usually better to work in hexadecimal when dealing with byte values.

Fortunately, we can change the numeration base used by poke when printing numbers, using the `.set obase` ("set output base") dot-command as this:

```
(poke) .set obase 16
```

After this, we can map the byte again, this time getting the result expressed in hexadecimal:

```
(poke) byte @ 64#B
0x25UB
```

Again, you may find it useful to add the `.set obase 16` command to your `.pokerc` file, if you want the customization to be persistent between poke invocations.

Going back to the example of calculating whether the highest bit in the second byte in `foo.o` is set, this is how we would do it with a map:

```
(poke) (byte @ 2#B) & 0x80
0
```

Turns out the answer is no.

The map operator can also be used at the left side of an assignment operator:

```
(poke) byte @ 0x28#B = 0xff
```

Which reads "assign 0xff to the byte at offset 0x4a bytes". Dumping again, we can verify that the byte actually changed:

```
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 f700 0102 0000 0000 0000 0000 0000  ................
00000020: 0102 0000 0000 0000 ff01 0000 0000 0000  ................
00000030: 0000 0000 4000 0000 0000 4000 0800 0700  ..............
```

Does this mean that `foo.o` changed accordingly, in disk? The answer is yes. poke always commits changes immediately to the file being edited. This, that is an useful feature, can also be a bit tricky if you forget about it, leading to data corruption, so please be careful.

Incidentally, altering the byte at offset 0x28 most probably have caused `foo.o` to stop being a valid ELF file, but since we are just editing bytes (and not ELF structures) we actually don't care much.

3.5 Values and Variables

Up to now we have worked with byte values, either writing them in the REPL or mapping them at the current IO space. Often it is useful to save values under meaningful names, and access to them by name. In poke we do that by storing the values in *variables*.

Before being used, variables shall be defined using the `var` construction. Let's get the byte at offset `64` bytes and save it in a variable called `foo`:

```
(poke) var foo = byte @ 64#B
```

This defines a new variable (`foo`) and initializes it to the value of the byte at offset 64 bytes. This results on `foo` to hold the value 37.

Once defined, we can get the value of a variable by just giving its name to poke:

```
(poke) foo
37UB
```

Several variables can be defined in a single `var` declaration. Each definition is separated by a comma. This is equivalent of issuing several `var`s:

```
(poke) var a = 10, b = 20
```

In general, a variable containing a byte value can be used in any context where the contained value would be expected. If we wanted to check the highest bit in the byte value stored in `foo` we would do:

```
(poke) foo & 0x80
0x0
```

Assigning a value to a variable makes the value the new contents of the variable. For example, we can increase the value of `foo` by one like this:

```
(poke) foo = foo + 1
```

At this point, an important question is: when we change the value of the variable `foo`, are we also changing the value of the byte stored in `foo.o` at offset 64 bytes? The answer is no. This is because when we do the mapping:

```
(poke) var foo = byte @ 64#B
```

The value stored in `foo` is a *copy* of the value returned by the map operator `@`. You can imagine the variable as a storage cell located somewhere in poke's memory. After the assignment above is executed there are two copies of the byte value 0x25: one in `foo.o` at offset 64 bytes, and the other in the variable `foo`.

It follows that if we wanted to increase the byte in the file, we would need to do something like:

```
(poke) var foo = byte @ 64#B
(poke) foo = foo + 1
(poke) byte @ 64#B = foo
```

Or, more succinctly, omitting the usage of a variable:

```
(poke) byte @ 64#B = (byte @ 64#B) + 1
```

Or, even *more* succinctly:

```
(poke) (byte @ 64#B) += 1
```

Note how we have to use parenthesis around the map at the right hand side, because the map operator `@` has less precedence than the plus operator `+`.

3.6 From Bytes to Integers

The bytes we have been working with are unsigned whole numbers (or integers) in the range `0..255`. We saw how poke sees the contents of the files as a sequence of bytes, and how each byte can be addressed using an offset. Mapping bytes using the map operator `@` gives us these values, which are denoted in poke with literals like `10UB` or `0x0aUB`.

This very limited range of values have consequences when it comes to do arithmetic with bytes. Suppose for example we wanted to calculate the average of the first byte values stored in `foo.o`. We could do something like:

```
(poke) a0 = byte @ 0#B
(poke) a1 = byte @ 1#B
(poke) a2 = byte @ 2#B
(poke) a0
0x7fUB
(poke) a1
0x45UB
(poke) a2
0x4cUB
(poke) (a0 + a1 + a2) / 3UB
5UB
```

That is obviously the wrong answer. What happened? Let's do it step by step. First, we add the first two bytes:

```
(poke) a0 + a1
0xc4UB
```

Which is all right. 0xc4 is 0x7f plus 0x45. But, let's add now the third byte:

```
(poke) a0 + a1 + a2
0x10UB
```

That's no good. Adding the value of the third byte (0x4c) we overflow the range of valid values for a byte value. The calculation went banana at this point.

Another obvious problem is that we surely will want to store integers bigger than 255 in our files. Clearly we need a way to encode them somehow, and since all we have in a file are bytes, the integers will have to be composed of them.

Integers bigger than 255 can be encoded by interpreting consecutive byte values in a certain way. First, let's consider a single byte. If we print a byte value using binary rather than decimal or hexadecimal, we will observe that eight bits are what it takes to encode the numbers between 0 and 0xff (255) using a *natural binary encoding*:

```
(poke) .set obase 2
(poke) 0UB
0b00000000UB
(poke) 0xFFUB
0b11111111UB
```

This is the reason why people say bytes are "composed" of eight bits, or that the width of a byte is eight bits. But this way of talking doesn't really reflect the view that the operating system has of devices like files or memory buffers: both disk and memory controllers provide and consume bytes, *i.e.* little unsigned numbers in the range `0..255`. At that level, bytes are indivisible. We will see later that poke provides ways to work on the "sub-byte" level, but that is just really an artifact to make our life easier: underneath, all that goes in and out are bytes.

Anyhow, if we were to "concatenate" the binary representation of two consecutive bytes, we would end with a much bigger range of possible numbers, in the range `0b00000000_00000000..0b11111111_11111111`[3], or `0x0000..0xffff` in hexadecimal. poke provides a bit-concatenation operator `:::` that does exactly that:

```
(poke) 0x1UB
0b00000001UB
(poke) 0x1UB ::: 0x1UB
0b0000000100000001UH
```

Note how the suffix of the resulting number is now `UH`. This indicates that the number is no longer a byte value: it is too big for that. The `H` in this new suffix means "half", and it is a traditional way to call an integer that is encoded using two bytes, or 16 bits.

So, using our method of encoding bigger numbers concatenating bytes, what would be the "half" integer composed of two bytes at the beginning of `foo.o`?

```
(poke) .set obase 16
(poke) (byte @ 0#B):::(byte @ 1#B)
0x7f45UH
```

Now, let's go back to the syntax we used to map a byte value. In the invocation of the map operator `byte @ 0#B` the operand at the left (in this case `byte`) tells the operator what kind of

[3] poke allows to insert underscore characters `_` anywhere in number literals. The only purpose of these characters is to improve readability, and they are totally ignored by poke, *i.e.* they do not alter the value of the number.

value to map. This is called a *type specifier*; `byte` is the type specifier for a single byte value, and `byte[3]` is the type specifier for a group of three byte values arranged in an array.

As it happens, `byte` is a synonym for another slightly more interesting type specifier: `uint<8>`. You can probably infer the meaning already: a byte is an unsigned integer which is 8 bits big. We can of course use this alternate specifier in a mapping operation, achieving exactly the same result than if we were using `byte`:

```
(poke) uint<8> @ 0#B
0x7fUB
```

You may be wondering: is it possible to use a similar type specifier for mapping bigger integers, like these "halves" that are composed of two bytes? Yeah, it is indeed possible:

```
(poke) uint<16> @ 0#B
0x7f45UH
```

Mapping an unsigned integer of 16-bits at the offset 0 gives us an unsigned "half" value, as expected.

You can easily build bigger and bigger numbers concatenating more and more bytes. Three bytes? sure:

```
(poke) uint<24> @ 0#B
(uint<24>) 0x7f454c
```

Note that in this case poke uses a prefix instead of a suffix to indicate that the given value is 24-bits long. Four bytes?

```
(poke) uint<32> @ 0#B
0x7f454c46U
```

Certain integer widths are so often used that easier-to-type synonyms for their type specifiers are provided. We already know `byte` for `uint<8>`. Similarly, `ushort` is a synonym for `uint<16>`, `uint` is a synonym for `uint<32>` and `ulong` is a synonym for `uint<64>`. Try them!

GNU poke supports integers up to eight bytes, *i.e.* up to 64-bits. This may change in the future, as we are planning to support arbitrarily large integers.

3.7 Big and Little Endians

When talking about whole numbers (integers) we should distinguish between their value (such as 123) and their *written form* that we would use when writing the number on a piece of paper, such as `123`.

The written form of a number is composed of digits, arranged in certain order. We all know that the ordering of the digits in the written form of a number is important: if we write `123` we are referring to a different value than if we write `321`. The mathematical reason for this is that depending on the position they occupy in the written form, each digit contributes with a different "weight" to the total value of the number. This is always the case, regardless of the numerical base used to denote the number.

For example, the value of the number 123 (whose written form is `123`) is calculated as `1*10^2+2*10^1+3*10^0`. If we swap the last two digits in the written form of the number, we have `1*10^2+3*10^1+2*10^0`, which results in a different value: `132`. When we consider other numerical bases, the bases in the polynomial change accordingly, but the correspondence between written form and value stands: for example, the value of 0x123 is calculated as `1*16^2+2*16^1+3*16^0`.

The "higher" a digit is in the polynomial, the *more significant* it is, *i.e.* the more weight it has on the value of the number where it appears. In the written number `123`, for example, the digit 1 is the *most significant* digit of the number, and the digit 3 is the *least significant* digit.

This distinction between the written form of a number and its value is very important. Just like in certain languages letters are read right-to-left (Arabic) or even down-to-up (Japanese) we could certainly conceive a language in which the digits of numbers were arranged from right-to-left instead of left-to-right. In such a language the written representation of 123 would be `321`, not `123`. In other words: the least significant digit would come first, not last, in the written form of the number.

Now when it comes to store numbers in computers, rather than writing them on a paper, the role of the paper is played by the computer's memory, be it ephemeral (like RAM) or persistent (like a spinning hard disk or a Flash memory), which is organized as a sequence of bytes. Since we are composing numbers with bytes, it makes sense to have each byte to play the role of a digit in the written form of the bigger number. Since bytes can have values from 0 to 255, the base is 256. But what is the "written form" for our byte-composed numbers?

In the last section we tried to compose bigger integers by concatenating bytes together and interpreting the result. In doing so, we assumed (quite naturally) that in the written form of the resulting integer the bytes are ordered in the same order than they appear in the file, *i.e.* we assume that the written form of the number `b1*256^2+b2*256^1+b3*256^0` would be `b1b2b3`, where `b1`, `b2` and `b3` are bytes. In other words, given a written form `b1b2b3`, `b1` would be the most significant byte (digit) and `b3` would be the least significant byte (digit). In our world of IO spaces, the "written form" is the disposition of the bytes in the IO space (file, memory buffer, *etc*) being edited.

That interpretation of the written form is exactly what the bit-concatenation operator implements:

```
(poke) dump :from 0#B :size 3#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 7f45 4c                                  .EL
(poke) var b1 = byte @ 0#B
(poke) var b2 = byte @ 1#B
(poke) var b3 = byte @ 2#B
(poke) b1:::b2:::b3
(uint<24>) 0x7f454c
```

However, much like in certain human languages the written form is read from right to left, some computers also read numbers from right to left in their "written form". Actually, turns out that *most* modern computers do it like that. This means that, in these computers, given the written form `b1b2b3` (*i.e.* given a file where `b1` comes first, followed by `b2` and then `b3`) the most significant byte is `b3` and the least significant byte is `b1`. Therefore, the value of the number would be `b3*256^2+b2*256^1+b3*256^0`.

So, given the written form of a bigger number `b1b2b3` (*i.e.* some ordering of bytes implied by the file they are stored in) there are at least two ways to interpret them to calculate the value of the number. When the written form is read from left to right, we talk about a *big endian* interpretation. When the written form is read from right to left, we talk about a *little endian* interpretation.

Given the first three bytes in `foo.o`, we can determine the value of the integer composed of these three bytes in both interpretations:

```
(poke) b1:::b2:::b3
(uint<24>) 0x7f454c
(poke) b3:::b2:::b1
(uint<24>) 0x4c457f
```

Remember how the type specifier `byte` is just a synonym of `uint<8>`, and how we can use type specifiers like `uint<24>` and `uint<32>` to map bigger integers? When we do that, like in:

```
(poke) uint<24> @ 0#B
```

```
(uint<24>) 0x7f454c
```

Poke should somehow decide what kind of interpretation to use, *i.e.* how to read the "written form" of the number. As you can see from the example, poke uses the left-to-right interpretation, or big-endian, by default. But you can change it using a new dot-command: `.set endian`:

```
(poke) .set endian little
(poke) uint<24> @ 0#B
(uint<24>) 0x4c457f
```

The currently used interpretation (also called *endianness*) is shown if you invoke the dot-command without an argument[4]:

```
(poke) .set endian
little
```

Different systems use different endianness. Into a given system, it is to be expected that most files will be encoded following the same conventions. Therefore poke provides you a way to set the endianness to whatever endianness is in the system. You do it this way:

```
(poke) .set endian host
```

3.8 Negative Integers

3.8.1 Negative Encodings

Up to this point we have worked with unsigned integers, *i.e.* whole numbers which are zero or bigger than zero. Much like it happened with endianness, the interpretation of the value of several bytes as a negative number depends on the specific interpretation.

In computing there are two main ways to interpret the values of a group of bytes as a negative number: *one's complement* and *two's complement*.

At the moment GNU poke supports the two complement interpretation, which is really ubiquitous and is the negative encoding used by the vast majority of modern computers and operating systems.

We may consider adding support for one's complement in the future, but only if there are real needs that would justify the effort (which wouldn't be a small one ;)).

3.8.2 Signed Integers

Unsigned values are never negative. For example:

```
(poke) 0UB - 1UB
0xffUB
```

Instead of getting a -1, we get the result of an unsigned underflow, which is the biggest possible value for an unsigned integer of size 8 bits: 0xff.

When using type specifiers like `uint<8>` or `uint<16>` in a map, we get unsigned values such as 0UB. It follows that we need other type specifiers to map signed values. These look like `int<8>` and `int<16>`.

For example, let's map a signed 16-bit value from `foo.o`:

```
(poke) .set obase 10
(poke) int<16> @ 0#B
28515H
```

Note how the suffix of the value is now `H` and not `UH`. This means that the value is signed! But in this case it is still positive, so let's try to get an actual negative value:

```
(poke) var h = int<16> @ 0#B
```

[4] This also applies to the other `.set` commands

```
(poke) h - h - 1H
-1H
```

3.8.3 Mixing Signed and Unsigned Integers

Adding two signed integers gives you a signed integer:

```
(poke) 1 + 2
3
```

Likewise, adding two unsigned integers results in an unsigned integer:

```
(poke) 1U + 2U
3U
```

But, what happens if we mix signed and unsigned values in an expression? Is the result signed, or unsigned? Let's find out:

```
(poke) 1U + 2
3U
```

Looks like combining an unsigned value with a signed value gives us an unsigned value. This actually applies to all the operators that work on integer values: multiplication, division, exponentiation, *etc.*

What actually happens is that the signed operand is converted to an unsigned value before executing the expression. You can also convert signed values into unsigned values (and vice-versa) using *cast constructions*:

```
(poke) 2 as uint<32>
2U
```

Therefore, the expression `1U + 2` is equivalent to `1U + 2 as uint<32>`:

```
(poke) 1U + 2 as uint<32>
3U
```

You may be wondering: why not doing it the other way around? Why not converting the unsigned operand into a signed value and then operate? The reason is that, given an integer of some particular size, the positive range that you can store in it is bigger when interpreted as an unsigned integer than when interpreted as a signed integer. Therefore, converting signed into unsigned before operating reduces the risk of positive overflow. This of course assumes that we, as users, will be working with positive numbers more often than with negative numbers, but that is a reasonable assumption to do, as it is often the case!

3.9 Weird Integers

Up to this point we have been playing with integers that are built using a whole number of bytes. However, we have seen that the type specifier for an integer has the form `int<N>` or `uint<N>` for signed and unsigned variants, where `N` is the width of the integer, in bits. We have used bit-sizes that are multiple of 8, which is the size of a byte. So, why is this so? Why is `N` not measured in bytes instead?

The reason is that poke is not limited to integers composed of a whole number of bytes. You can actually have integers having *any* number of bits, between 1 and 64. So yes, `int<3>` is a type specifier for signed 3-bit integers, and `uint<17>` is a type specifier for unsigned 17-bit integers.

We call integers like this *weird integers*.

The vast majority of programming languages do not provide any support for weird integers. In the few cases they do, it is often in a very limited and specific way, like bitmap fields in C structs. Such constructions are often vague, obscure, and often their semantics depend on the

specific implementation of the language, and/or the characteristics of the system where you run your program.

In poke, on the contrary, weird numbers are first class citizens, and they don't differ in any way from "normal" integers composed of a whole number of bytes. Their interpretation is also well defined, and they keep the same semantics regardless of the characteristics of the computer on which poke is running.

3.9.1 Incomplete Bytes

Let's consider first weird numbers that span for more than one byte. For example, an unsigned integer of 12 bits. Let's visualize the written form of this number, *i.e.* the sequence of its constituent bytes as they appear in the underlying IO space:

```
  byte 0  |  byte 1
+---------+----+----+
|::::::::::::::|    |
+---------+----+----+
|   uint<12>   |
```

All right, the first byte is used in its entirely, but only half of the second byte is used to conform the value of the number. The other half of the second byte has no influence of the value of the 12 bits number.

Now, we talk about the "second half of the byte", but what do that means exactly? We know that bytes in memory and files (bytes in IO spaces) are indivisible at the system level: bytes are read and written one at a time, as little integers in the range `0..255`. However, we can create the useful fiction that each byte is composed by *bits*, which are the digits in the binary representation of the byte value.

So, we can look at a byte as composed of a group of eight bits, like this:

```
            byte
+-------------------------+
| b7 b6 b5 b4 b3 b2 b1 b0 |
+-------------------------+
```

Note how we decided to number the bits in descending order from left to right. This is because these bits correspond to the base of the polynomial equivalent to the binary value of the byte, *i.e.* the value of the byte is `b7*2^7+b6*2^6+b5*2^5+b4*2^4+b3*2^3+b2*2^2+b1*2^1+b0*2^0`. In other words: at the bit level poke always uses a big endian interpretation, and the bit that "comes first" in this imaginary stream of bits is the most significant bit in the binary representation of the number. Please note that this is just a convention, set by the poke authors: the opposite could have been chosen, but it would have been a bit confusing, as we would have to picture binary numbers in reverse order!

With this new way of looking at bytes, we can now visualize what we mean exactly with the "first half" and "second half" of the trailing byte, in our 12 bits unsigned number:

```
            byte 0          |           byte 1
+-------------------------+-------------+-------------+
| a7 a6 a5 a4 a3 a2 a1 a0   b7 b6 b5 b4 :             |
+-------------------------+-------------+-------------+
|                uint<12>                |
```

Thus the first half of `byte 1` is the sequence of bits `b7 b6 b5 b4`. The second half, which is not pictured since it doesn't contribute to the value of the number, would be `b3 b2 b1 b0`.

So what would be the value of the 12-bit integer? Exactly like with non-weird numbers, this depends on the current selected endianness, which determines the ordering of bytes.

If the current endianness is big, then `byte 0` provides the most significant bits of the result number, and the used portion of `byte 1` provides the least significant bits of the result number:

```
0b a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4
```

However, if the current selected endianness is little, then the used portion of `byte 1` provides the most significant bits of the result number, and `byte 0` provides the least significant bits of the result number:

```
0b b7 b6 b5 b4 a7 a6 a5 a4 a3 a2 a1 a0
```

Let's see this in action. Let's take a look to the value of the first two bytes in `foo.o`, in binary:

```
(poke) .set obase 2
(poke) byte @ 0#B
0b01111111UB
(poke) byte @ 1#B
0b01000101UB
```

Looking at these bytes as sequences of bits, we have:

```
         byte @ 0#B       |         byte @ 1#B
+-------------------------+-------------+-------------+
|  0  1  1  1  1  1  1  1    0  1  0  0 :  0  1  0  1
+-------------------------+-------------+-------------+
|                uint<12>                |
```

Let's map our weird number at offset 0 bytes, using big endian:

```
(poke) .set endian big
(poke) uint<12> @ 0#B
(uint<12>) 0b011111110100
```

That matches what we explained before: the most significant bits of the unsigned 12 bits number come from the byte at offset 0, *i.e.* `01111111`, whereas the least significant bits come from the byte at offset 1, *i.e.* `0100`.

Now let's map it using little endian:

```
(poke) uint<12> @ 0#B
(uint<12>) 0b010001111111
```

This time the most significant bits of the unsigned 12 bits number come from the byte at offset 1, *i.e.* `0100`, whereas the least significant bits come from the byte at offset 0, *i.e.* `01111111`.

An important thing to note is that non-weird numbers, *i.e.* numbers built with a whole number of bytes, are basically a particular case of weird numbers where the last byte in the written form (in the IO space) provides all its bits. The rules are exactly the same in all cases, which makes it easy to obtain predictable and natural results when building integers using poke.

3.9.2 Quantum Bytenics

The second kind of weird numbers are integers using less than 8 bits. These "sub-byte" numbers do not use all the bits of their containing byte. Consider for example the written form of an unsigned integer of size 5 bits:

```
    byte
+-----+----+
|:::::|    |
+-----+----+
uint<5>
```

Now let's view the byte as a sequence of bits:

```
                byte
```

```
+----------------+----------+
| b7 b6 b5 b4 b3 |          |
+----------------+----------+
|    uint<5>     |
```

What is the value of this number? Applying the general rules for building integers from bytes, we can easily see that regardless of the current endianness the value, in binary, is:

```
0b b7 b6 b5 b4 b3
```

Let's see this in poke:

```
(poke) .set obase 2
(poke) .set endian big
(poke) byte @ 0#B
0b01111111UB
(poke) uint<5> @ 0#B
(uint<5>) 0b01111
(poke) .set endian little
(poke) uint<5> @ 0#B
(uint<5>) 0b01111
```

3.9.3 Signed Weird Numbers

In the section discussing negative integers, we saw how the difference between a signed number and an unsigned number is basically a different interpretation of the most significant byte. Exactly the same applies to weird numbers.

Let's summon our unsigned 12-bit integer at the beginning of the file `foo.o`:

```
(poke) .set endian big
(poke) uint<12> @ 0#B
(uint<12>) 0b011111110100
```

The most significant byte of the resulting value (not of its written form) indicates that this number would be positive if we were mapping the corresponding signed value. Let's see:

```
(poke) int<12> @ 0#B
(int<12>) 0b010001111111
(poke) .set obase 10
(poke) int<12> @ 0#B
(int<12>) 1151
```

Let's make it a bit more interesting, and change the value of the first byte in the file so we get a negative number:

```
(poke) .set obase 2
(poke) byte @ 0#B = 0b1111_1111
(poke) int<12> @ 0#B
(int<12>) 0b111111110100
(poke) .set obase 10
(poke) int<12> @ 0#B
(int<12>) -12
```

Now, let's switch to little endian:

```
(poke) .set endian little
(poke) .set obase 2
(poke) int<12> @ 0#B
(int<12>) 0b010011111111
(poke) .set obase 10
(poke) int<12>  0#B
(int<12>) 1279
```

3.10 Unaligned Integers

We have mentioned above that the data stored in computers, that we edit with poke, is arranged as a sequence of bytes. The entities we edit with poke (that we call *IO devices*) are presented to us as IO spaces. Up to now, we have accessed this IO space in terms of bytes, in commands like `dump :from 32#B` and in expressions like `2UB + byte @ 0#B`. We said that mapped integers are built from bytes read from the IO space.

However, the IO space that poke offers to us is actually a space of bits, not a space of bytes, and the poke values are mapped on this space of bits. The following figure shows this:

```
poke values      |        uint<16> @ 2#b         |
-----------      |                               |
IO space     |b|b|b|b|b|b|b|b|b|b|b|b|b|b|b|b|b|b|b|b|
-----------  |               |               |               |
IO device    |     byte0     |     byte1     |     byte2     |
```

The main consequence of this, that you can see in the figure above, is that we can use offsets in mapping operations that are not *aligned to bytes*. You can specify an offset in bits, instead of bytes, using the `#b` suffix instead of `#B`. Little `b` means bits, and big `B` means bytes.

Let's map an unaligned 16 bit unsigned integer in `foo.o`:

```
(poke) dump :from 0#B :size 3#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 7f45 4c                                 .EL
(poke) .set obase 2
(poke) byte @ 0#B
0b01111111UB
(poke) byte @ 1#B
0b01000101UB
(poke) byte @ 2#B
0b01001100UB
(poke) .set endian big
(poke) uint<16> @ 2#b
0b1111110100010101UH
```

Graphically:

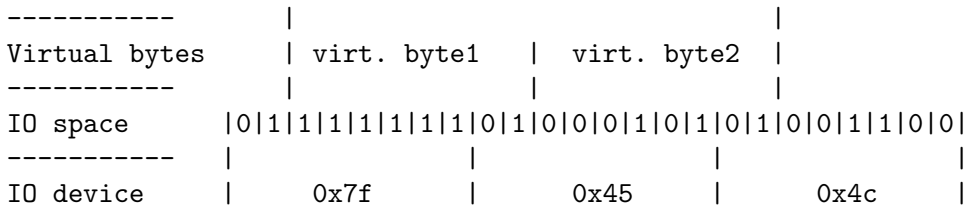
```
poke values      |        uint<16> @ 2#b         |
-----------      |                               |
IO space     |0|1|1|1|1|1|1|1|0|1|0|0|0|1|0|1|0|1|0|0|1|1|0|0|
-----------  |               |               |               |
IO device    |     0x7f      |      0x45     |      0x4c     |
```

These three levels of abstractions make it very easy and natural to work with unaligned data. Imagine for example that you are poking packages in a network protocol that is bit-oriented. This means that the packages will generally not be aligned to byte boundaries, but still the payload stored in the packages contains integers of several sizes. poke allows you to directly map these integers as if they were aligned to byte boundaries, and work with them.

However, when one tries to determine the correspondence between a given poke value and the underlying bytes in the IO device, things can get complicated. This is particularly true when we map what we called "weird numbers", *i.e.* numbers with partial bytes. As we saw, the rules to build these numbers were expressed in terms of bytes.

In order to ease the visualization of the process used to build integer values (especially if they are weird numbers, *i.e.* integers with partial bytes) one can imagine an additional layer of "virtual bytes" above the space of bits provided by the IO space. Graphically:

```
poke values       |        uint<16> @ 2#b         |
```



It is very important to understand that the IO space is an abstraction provided by poke. The underlying file, or memory buffer, or whatever, is actually a sequence of bytes; poke translates the operations on integers, bits, bytes, *etc* into the corresponding byte operations, which are far from trivial. Fortunately, you can let poke to do that dirty job for you, and abstract yourself from that complexity.

3.11 Integers of Different Sizes

When integer values of different sizes are passed to an arithmetic or relational operator, the "smaller" operand gets converted into the size of the "bigger" operand. For example:

```
(poke) 1H + 2
3
```

The operands are of size 16-bit and 32-bit respectively, and the result is a 32-bit integer. This is equivalent to:

```
(poke) 1H as int<32> + 2
3
```

3.12 Offsets and Sizes

Early in the design of what is becoming GNU poke I was struck by a problem that, to my surprise, would prove not easy to fix in a satisfactory way: would I make a byte-oriented program, or a bit-oriented program? Considering that the program in question was nothing less than an editor for binary data, this was no petty dilemma.

Since the very beginning I had a pretty clear idea of what I wanted to achieve: a binary editor that would be capable of editing user defined data structures, besides bytes and bits. I also knew I needed some sort of domain specific language to describe these structures and operate on them. How that language would look like, and what kind of abstractions it would provide, however, was not clear to me. Not at all.

So once I sketched an initial language design, barely something very similar to C structs, I was determined to not continue with the poke implementation until I had described as many as binary formats in my language as possible. That, I reckoned, was the only way to make sure the implemented language would be expressive, complete and useful enough to fulfil my requirements.

The first formats I implemented using my immature little language included ELF, FLV, MP3, BSON... of them describing structures based on whole bytes. Even when they try to be compact, it is always by packing bit-fields in elements that are, invariably, sized as a multiple of bytes. Consequently, the language I was evolving became byte oriented as well. No doubt also influenced by my C inheritance, I would think of bit-fields either as a sort of second class citizen, or as mere results of shifting and masking.

This worked well. The language evolved to be able to express many different aspects of these formats in a very nice way, like variable-length data and holes in structures. Consider the following definition, which is **not** valid in today's Poke:

```
type Data =
  struct
  {
```

```
    byte magic;
    byte count;
    byte dstart;

    byte[count] data @ dstart;
  };
```

The data starts with a byte that is a magic number. Then the size of the data stored, in bytes, and then the data itself. This data, however, doesn't start right after `dstart`: it starts at `dstart`, which is expressed as an offset, in bytes, since the beginning of the Data. I conceived struct field labels to be any expression evaluating to an integer, which would be..., bytes obviously.

Then, one day, it was the turn for IETF RFC1951, which is the specification of the DEFLATE algorithm and associated file format. Oh dear. Near the beginning of the spec document it can be read:

> This document does not address the issue of the order in which bits of a byte are transmitted on a bit-sequential medium, since the final data format described here is byte- rather than bit-oriented. However, we describe the compressed block format in below, as a sequence of data elements of various bit lengths, not a sequence of bytes.

Then it goes on describing rules to pack the DEFLATE elements into bytes. I was appalled, and certainly sort of deflated as well. The purpose of my program was precisely to edit binary in terms of the data elements described by a format. And in this case, these data elements came in all sort of bit lengths and alignments. This can be seen in the following RFC1951 excerpt, that describes the header of a compressed block:

> Each block of compressed data begins with 3 header bits containing the following data:
>
> ```
> first bit       BFINAL
> next 2 bits     BTYPE
> ```
>
> Note that the header bits do not necessarily begin on a byte boundary, since a block does not necessarily occupy an integral number of bytes.

At this point I understood that my little language on the works would never be capable to describe the DEFLATE structures naturally: C-like bit-fields, masking and shifting, all based on byte-oriented containers and boundaries, would never provide the slickness I wanted for my editor. I mean, just use C and get done with it.

This pissed me off. Undoubtedly other formats and protocols would be impacted in a similar way. Even when most formats are byte oriented, what am I supposed to tell to the people hacking bit-oriented stuff? "Sorry pal, this is not supported, this program is not for you"? No way, I thought, not on my watch.

The obvious solution for the problem, is to be general. In this case, to express every offset and every memory size in bits instead of bytes. While this obviously gives the language maximum expressiveness power, and is ideal for expressing the few bit-oriented formats, it has the disadvantage of being very inconvenient for most situations.

To see how annoying this is, let's revisit the little Data element we saw above. In a bit-oriented description language, we would need to write something like:

```
type BitData =
  struct
  {
    byte magic;
    byte count;
```

```
  byte dstart;

  byte[count] data @ dstart * 8;
};
```

Yeah... exactly. The `*` and `8` keys in the keyboards of the poke users would wear out very fast, not to mention their patience as well. Also, should I provide both `sizeof` and `bitsizeof` operators? Nasty.

I am very fond of the maxim "Never write a program you would never use yourself"[5], so I resigned myself to make GNU poke byte oriented, and to provide as many facilities for operating on bit-fields as possible.

Fortunately, I have smart friends...

During one of the Rabbit Herd's Hacking Weekends[6] I shared my frustration and struggle with the other rabbits, and we came to realize that offsets and data sizes in Poke should not be pure magnitudes or mere integer values: they should be united. They should have units.

It makes full sense when you come to think about it. For a program like poke, it is only natural to talk about different memory units, like bits, bytes, kilobytes, megabits, and so on. Bits and bytes are just two common units. Apart from allowing me to express values in different units, this approach also has other benefits as we will see shortly.

I'm really grateful to Bruno Haible, Luca Saiu and Nacho Gonzalez for putting me on the right track.

3.13 Buffers as IO Spaces

We have mentioned already that files are not the only entities that can be edited using poke. Remember the dot-command `.file` that opened a file as an IO space?

```
(poke) .file foo.o
The current IOS is now ‘./foo.o’.
(poke) .info ios
  Id Type Mode Bias Size Name
* #0 FILE rw 0x00000000#B 0x000004c8#B ./foo.o
```

Memory buffers can be created using a similar dot-command, `.mem`:

```
(poke) .mem foo
The current IOS is now ‘*foo*’.
(poke) .info ios
  Id Type Mode Bias Size Name
* #1 MEMORY 0x00000000#B 0x00001000#B    *foo*
  #0 FILE rw 0x00000000#B 0x000004c8#B    ./foo.o
```

Note how the name of the buffer is built by prepending and appending asterisks. Therefore, the name of the buffer created by the command `.mem foo` is `*foo*`. Note also that the new buffer is created with a default size of 0x1000 bytes, or 4096 bytes. The contents of the buffer are zeroed:

```
(poke) dump
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

[5] Actually it is Lord Vetinari's "Never build a dungeon you can't get out of." but the point is the same.

[6] `http://www.jemarch.net/rhhw`

```
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Memory buffer IO spaces grow automatically when a value is mapped beyond their current size. This is very useful when populating newly created buffers. However, for security reasons, there is a limit: the IO spaces are only allow to grow 4096 bytes at a time.

When it comes to map values, there is absolutely no difference between an IO space backed by a file and an IO space backed by a memory buffer. Exactly the same rules apply in both cases.

3.14 Copying Bytes

Memory buffer IO spaces are cheap, and they are often used as "scratch" areas.

Suppose for example we want to experiment with the ELF header of `foo.o`. We could open it in poke:

```
(poke) .file foo.o
The current IOS is now ‘./foo.o’.
```

The header of an ELF file comprises the first 64 bytes in the file:

```
(poke) dump :size 64#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  ..>.............
00000020: 0000 0000 0000 0000 0802 0000 0000 0000  ................
00000030: 0000 0000 4000 0000 0000 4000 0b00 0a00  ............
```

We know that as soon as we poke something on an IO space, the underlying file is immediately modified. So if we start playing with `foo.o`'s header, we may corrupt the file. We could of course make a copy of `foo.o` and work on the copy, but it is much better to create a memory IO space and copy the ELF header there:

```
(poke) .mem scratch
The current IOS is now ‘*scratch*’.
(poke) copy :from_ios 0 :from 0#B :size 64#B
(poke) dump :size 64#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  ..>.............
00000020: 0000 0000 0000 0000 0802 0000 0000 0000  ................
00000030: 0000 0000 4000 0000 0000 4000 0b00 0a00  ............
```

The command `copy` above copies 64 bytes starting at byte 0 from the IO with id 0 (the file `foo.o`) to the current IO space (the buffer `*scratch*`).

Once we are done working with the copy of the ELF header, and satisfied, we can copy it back to the file and close the memory IO space:

```
(poke) copy :from 0#B :size 64#B :to_ios 0
(poke) .close
The current IOS is now ‘./foo.o’.
```

Note how the command arguments `:from_ios` and `:to_ios` are assumed to be the current IO space if they are not explicitly specified in the command invocation. For detailed information on the `copy` command, see Section 17.2 [copy], page 111.

3.15 Saving Buffers in Files

Another useful command when working with buffer IO spaces (and in general, with any IO space) is `save`. Let's say we want to save a copy of the header of an ELF file in another file. We could do it the pokeish way:

```
$ poke foo.o
[...]
(poke) save :from 0#B :size 64#B :file "header.dat"
```

The command above saves the first 64 bytes in the current IO space (which is backed by the file `foo.o`) into a new file `header.dat` in the current working directory.

Let's now consider again the scenario where we are using a memory IO space as a scratch area. It is late in the night and we are tired, so we would like to save the contents of the scratch buffer to a file, so we can continue our work tomorrow. This is how we would do that:

```
(poke) .info ios
  Id Type Mode Bias Size Name
* #1 MEMORY 0x00000000#B 0x00001000#B    *scratch*
  #0 FILE rw 0x00000000#B 0x000f4241#B    ./foo.o
(poke) save :from 0#B :size iosize (1) :file "scratch.dat"
```

Here we used the built-in function `iosize`, that given an IO space identifier returns its size.

3.16 Character Sets

Computers understand text as a sequence of *codes*, which are numbers identifying some particular *character*. A character can represent things like letters, digits, ideograms, word separators, religious symbols, *etc.* Collections of character codes are called *character sets*.

Some character sets try to cover one or a few similar written languages. This is the case of ASCII and ISO Latin-1, for example. These character sets are small, *i.e.* just a few hundred codes.

Other character sets are much more ambitious. This is the case of Unicode, that tries to cover the entire totality of human languages in the globe, including the fictitious ones, like klingon. Unicode is a really big character set.

In order to store character codes in a computer's memory, or a file, we need to *encode* each character code in one or more bytes. The number of bytes needed to encode a given character code depends on the range of codes in the containing set.

ASCII, for example, defines 128 character codes: a single byte is enough to encode every possible ASCII character. It is very easy to encode ASCII.

Unicode, on the contrary, defines many thousand of character codes, and has room for many more: we would need 31 bits in order to encode any conceivable Unicode character code. However, it would be wasteful to use that many bits per character: most used character codes tend to be in lower regions of the code space. For example, the code corresponding to the Latin letter `'a'` is a fairly small number, whereas the codes corresponding to the Klingon alphabet are really big numbers. Consequently, some systems opt to just encode a subset of Unicode, like the first 16 bits of the Unicode space, which is called the Basic Multilingual Plane, and contains all the characters that most people will ever need. There are also variable-length encodings of Unicode, that try to use as less bytes as possible to encode any given code. A particularly clever encoding of Unicode, designed by Rob Pike, is backwards compatible with the ASCII encoding, *i.e.* it encodes all the ASCII codes in one byte each, and the values of these byte are the same than in ASCII. This clever encoding is called UTF-8.

3.17 From Bytes to Characters

3.17.1 Character Literals

poke has built-in support for ASCII, and its simple encoding: each ASCII code is encoded using one byte. Let's see:

```
(poke) 'a'
0x61UB
```

We presented poke with the character `a`, and it answered with its corresponding code in the ASCII character set, which is 0x61. In fact, 'a' and 0x61UB are just two ways to write exactly the same byte value in poke:

```
(poke) 'a' == 0x61UB
1
(poke) 'a' + 1
0x62U
```

In order to make this more explicit, poke provides yet another synonym for the type specifier `uint<8>`: `char`.

3.17.2 Classifying Characters

When working with characters it is very useful to have some acquaintance of the ASCII character set, which is designed in a very clever way with the goal of easing certain code calculations. See Appendix A [Table of ASCII Codes], page 227, for a table of ASCII codes in different numeration bases.

Consider for example the problem of determining whether a byte we map from an IO space is a digit. Looking at the ASCI table, we observe that digits all have consecutive codes, so we can do:

```
(poke) var b = byte @ 23#B
(poke) b >= '0' && b <= '9'
1
```

Now that we know that `b` is a digit, how could we calculate its digit value? If we look at the ASCII table again, we will find that the character codes for digits are not only consecutive: they are also ordered in ascending order `0`, `1`, ... Therefore, we can do:

```
(poke) b
0x37UB
(poke) b - '0'
7UB
```

`b` contains the ASCII code 0x37UB, which corresponds to the character `7`, which is a digit.

How would we check whether `b` is a letter? Looking at the ASCII table, we find that lower-case letters are encoded consecutively, and the same applies to upper-case letters. This leads to repeat the trick again:

```
(poke) (b >= 'a' && b <= 'z') || (b >= 'A' && b <= 'Z')
0
```

3.17.3 Non-printable Characters

Not all ASCII code are printable using the glyph that are usually supported in terminals. If you look at the table in Appendix A [Table of ASCII Codes], page 227, you will find codes for characters described as "start of text", "vertical tab", and so on.

These character codes, which are commonly known as *non-printable characters*, can be represented in poke using its octal code:

```
(poke) '\002'
```

```
0x2UB
```

This is of course no different than using `2UB` directly, but in some contexts the "character like" notation may be desirable, to stress the fact that the byte value is used as an ASCII character.

Some of the non-printable characters also have alternative notations. This includes new-line and horizontal tab:

```
(poke) '\n'
0xaUB
(poke) '\t'
0x9UB
```

These \ constructions in character literals are called *escape sequences*. See Section 18.1.2 [Characters], page 115, for a complete list of allowed escapes in character literals.

3.18 ASCII Strings

3.18.1 String Values

Now that we know how to manipulate ASCII codes in poke, we may wonder how can we combine them to conform words or, more generally, *strings* of ASCII characters.

GNU poke has support for native ASCII string values. The characters conforming the string are written between `"` and `"` characters, like in:

```
(poke) "word"
"word"
```

Note, and this is important, that string values are as atomic as integer values: they are not really composite values. The fact that `"word"` contains an `r` at position 3 is like the fact that the value `123` contains a digit `2` at position 2.

Like in character literals, poke strings support several escape sequences that help to denote non-printed characters, such as new lines and horizontal tabs. See Section 18.3.1 [String Literals], page 122.

3.18.2 Poking Strings

Let's start with a fresh memory buffer IOS `*scratch*`:

```
(poke) .mem scratch
The current IOS is now `*scratch*'.
(poke) dump :size 48#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

If we wanted to, somehow, store the word `word` in this IO space, encoded in ASCII, we could proceed as:

```
(poke) char @ 0x12#B = 'w'
(poke) char @ 0x13#B = 'o'
(poke) char @ 0x14#B = 'r'
(poke) char @ 0x15#B = 'd'
(poke) dump :size 48#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 776f 7264 0000 0000 0000 0000 0000  ..word..........
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

This worked. The ASCII part of the `dump` output, which interprets the bytes as ASCII, clearly shows the word `word` at the offset where we poked the character values. However, we can do better: string values can be mapped themselves.

String values use the type specifier `string`. As any other kind of value in poke, they can be mapped from an IO space:

```
(poke) string @ 0x12#B
"word"
```

Clearly that is the string resulting from the concatenation of the character values that we poked before.

The question now is: how did poke know that the last character of the string was the `d` at offset 0x15#B? The fact the character code 0 (also known as the *NULL character*) at offset 0x16#B is non-printable, doesn't imply it is not part of the ASCII character set. Clearly, we have to pick an ASCII code and reserve it to mark the end of strings. Like the C programming language, and countless formats and systems do, poke uses the NULL character to delimit strings.

Now consider this:

```
(poke) "word"'length
4UL
(poke) "word"'size
40UL#b
```

Using the `length` and `size` attributes, poke tells us that the length of the string `"word"` is 4 characters, but the size of the string value is 40 bits, or 5 bytes. Why this discrepancy? The `size` value attribute tells how much storage space a given value required once mapped to an IO space, and in the case of strings it should count the space occupied by the terminating NULL character.

Poking string values on the IO space is as straightforward as poking integers:

```
(poke) string @ 0x22#B = "WORD"
(poke) dump :size 48#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 776f 7264 0000 0000 0000 0000 0000  ..word..........
00000020: 0000 574f 5244 0000 0000 0000 0000 0000  ..WORD..........
```

3.18.3 From Characters to Strings

Strings can be concatenated using the string-concatenation `+` operators:

```
(poke) "foo" + "bar"
"foobar"
```

The string resulting from the operation above has length 6 characters, and size 7 bytes. The terminating NULL character of `"foo"` is lost in the operation. This is easily seen:

```
(poke) "foo"'size + "bar"'size
0x40UL#b
(poke) ("foo" + "bar")'size
0x38UL#b
```

The string concatenation operator requires two strings. Therefore, if we wanted to append a character to a string, we would get an error:

```
(poke) "Putin" + 'a'
<stdin>:1:1: error: invalid operands in expression
"Putin" + 'a';
~~~~~~~~~~~~~
```

It is possible to transform a character value (*i.e.* a byte value) into a string composed of that character using a cast:

```
(poke) 'a' as string
"a"
```

Using that cast, we can now append:

```
(poke) "Putin" + 'a' as string
"Putina"
```

3.19 From Strings to Characters

Despite being atomic values, poke strings can be indexed in order to retrieve individual characters:

```
(poke) "word"[2]
0x72UB
```

Note how the indexing is zero-based, *i.e.* the first position in the string is referred as `[0]`, the second position with `[1]`, and so on.

If you specify a negative index, or an index that is too big, you will get an error:

```
(poke) "word"[-1]
<stdin>:1:8: error: index is out of bounds of string
"word"[-1];
       ^
(poke) "word"[5]
<stdin>:1:8: error: index is out of bounds of string
"word"[5];
       ^
```

3.20 Strings are not Arrays

Arrays are collections of homogeneous data organized in a sequence. We have already seen (briefly) arrays of bytes, which use the type specifier `byte[]`.

Similarly, it is possible to arrange characters (which are basically little numbers) in an array, like in:

```
(poke) ['a','b','c']
[0x61UB,0x62UB,0x63UB]
```

However, the array above is *not* equivalent to the string `"abc"`. The later is a simple value, whereas an array is a composite value, and also is implicitly terminated with a NULL character, *i.e.* a 0 byte.

The Poke standard library provides a couple of utility functions to convert between string values and character arrays: `catos` and `stoca`.

`catos` gets an array of characters and returns an equivalent string. For example:

```
(poke) catos (['a','b','c'])
"abc"
```

`stoca` gets a string and an array and sets the element of the array to the characters composing the string. For example:

```
(poke) var a = char[3]()
(poke) stoca ("abc", a)
(poke) a
[0x61UB,0x62UB,0x63UB]
```

4 Structuring Data

In the previous chapter we learned how to manipulate pre-defined entities like bytes, integers and strings. One of the big points of poke, however, is that it allows you to define your own data abstractions, and to operate in term of them. This is achieved by defining data structures.

4.1 The SBM Format

4.1.1 Images as Maps of Pixels

There are two main ways to store two-dimensional images in computers.

One is to explicitly store the different *pixels* that compose the image. In these *bitmaps*, the pixels are arranged sequentially and implicitly organized into *lines*. A header typically provides information to determine how many pixels fit in each line:

```
 <--- line_width  --->
| pixel | pixel | ... | pixel | pixel | ... | ...
|        line 1       |         line 2      |
```

Several properties have to be encoded for each pixel, depending on the sophistication of the image: for monochrome images each pixel can be just either switched on or off, so a single bit could be used to encode each pixel (this is the origin of the term "bitmap"). When color is added, a bit is no longer sufficient: the color of the pixel shall be encoded in some way, typically using a color schema such as RGB, that requires triplet of little integers to be encoded. If transparency is to be supported, the degree of transparency of the pixel shall also be encoded somehow.

The other way to store an image is to store a functional description of the "painted" parts of the image. This functional description usually contains instructions like "paint a line of thickness 1 and color red from the coordinate 10,20 to coordinate 10,40". Once executed with certain parameters (like the desired resolution) the functional description generates a bitmap. Image formats using this approach are commonly known as *vectorial formats*.

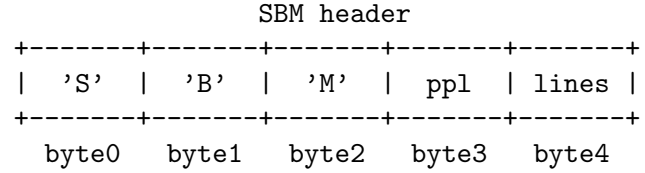
When it comes to bitmaps, there are a plethora of different formats out there (bmp, jpg, png) competing in terms of capabilities such as higher color depths, better resolution, support for transparency (alpha channels), higher compression level, and the like. These capabilities greatly complicate these formats, but ultimately any bitmap can be reduced to a sequence of pixels, which can be further structured in terms of *lines*.

We don't know enough poke yet to handle the complications of these real-life bitmap formats, so in subsequent sections we introduce a very simple format for bitmaps, the Stupid BitMap format, or simply SBM, that we will use for learning purposes.

But please do not feel disappointed: later in this book, when we become more proficient pokers, we will be messing with these shiny complex formats as well :)

4.1.2 SBM header

A SBM file starts with a header that contains a "magic number" composed of the three bytes 'S' (0x53UB), 'B' (0x42UB) and 'M' (0x4dUB). The next two bytes indicate the number of pixels per line, and the number of lines, respectively. In summary:



Here `ppl` stands for pixels-per-line. The encoding of these fields implies that the bigger image that can be encoded in SBM has dimensions 255x255.

4.1.3 SBM data

Albeit stupid, SBM is a colorful format. It supports more than a million colors, encoding each color in what is known as *RGB24*. In RGB24, each color is encoded using three little integers, specifying the amount of red, green and blue that, added together, compose the color.

The name RGB24 comes from the fact that each color is encoded using three bytes, or 24 bits. Therefore, each pixel in a SBM image is encoded using three consecutive bytes:

```
         SBM pixel
+-------+-------+-------+
|  Red  | Green | Blue  |
+-------+-------+-------+
  byte0   byte1   byte2
```

Each byte in this encoding determines the amount of the base color (red, green or blue) that compose the color. We will talk more about these components later.

4.2 Poking a SBM Image

4.2.1 P is for poke

Let's compose our first SBM image, using poke. The image we want to encode is the very simple rendering of the letter `P` shown in the figure below.

```
  | 0 | 1 | 2 | 3 | 4 |
  +---+---+---+---+---+
0 |   | * | * |   |   |
1 |   | * |   | * |   |
2 |   | * |   | * |   |
3 |   | * | * |   |   |
4 |   | * |   |   |   |
5 |   | * |   |   |   |
6 |   | * |   |   |   |
```

The image has seven lines, and there are five pixels per line, *i.e.* the dimension of the image in pixels is 5x7. Also, the pixels denoted by asterisks are red, whereas the pixels denoted with empty spaces are white. In other words, our image uses a red foreground over a white background. The "painted" pixels are called foreground pixels, the non painted pixels are called background pixels.

4.2.2 Preparing the Canvas

The first thing we need is some suitable IO space where to encode the image. Let's fire up poke and create a memory buffer:

```
$ poke
[...]
(poke) .mem image
The current IOS is now ‘*image*’.
(poke) dump
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Freshly created memory IO spaces are 4096 bytes long, and that's big enough for our little image. If we wanted to work with more data, remember that memory IO spaces will grow automagically when poked past their size.

4.2.3 Poking the Header

The first three bytes of the header of a SBM file contains the magic number that identifies the file as a SBM bitmap. We can poke these bytes very easily:

```
(poke) byte @ 0#B = 'S'
(poke) byte @ 1#B = 'B'
(poke) byte @ 2#B = 'M'
```

The next couple of bytes encode the dimensions of the bitmap, in this case 5x7:

```
(poke) byte @ 3#B = 5
(poke) byte @ 4#B = 7
```

There is something worth noting in this last mapping. Even tough we were poking bytes (passing the `byte` type specifier to the map operators) we specified the 32-bit signed integers `5` and `7` instead of `5UB` and `7UB`. When poke finds a situation like this, where certain kind of integers are expected but other kind are provided, it converts the value from the provided type to the expected type. This conversion may result in truncation (think about converting, say 0xfff to an unsigned byte, whose maximum possible value is 0xff) but certainly not in the case at hands.

The final header looks like:

```
(poke) dump :size 16#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 5342 4d05 0800 0000 0000 0000 0000 0000  SBM.............
```

4.2.4 Poking the Pixels

Now that we have written a SBM header, we have to encode the sequence of pixels composing the image.

Recall that every pixel is encoded using three bytes, that conform a RGB24 color. We have two kinds of pixels in our image: white pixels, and red pixels. In RGB24 white is encoded as `(255,255,255)`. Pure red is encoded as `(255,0,0)`, but to make things more interesting we will be using a nicer tomato-like red `(255,99,71)`.

Therefore, poking a white pixel at some offset *offset* would involve the following operations:

```
(poke) byte @ offset = 255
(poke) byte @ offset+1#B = 255
(poke) byte @ offset+2#B = 255
```

Likewise, the operations to poke a tomato pixel would look like:

```
(poke) byte @ offset = 255
(poke) byte @ offset+1#B = 99
(poke) byte @ offset+2#B = 71
```

To ease things a bit, we can define variables with the color components for both foreground and background pixels:

```
(poke) var bg1 = 255
(poke) var bg2 = 255
(poke) var bg3 = 255
(poke) var fg1 = 255
(poke) var fg2 = 99
```

```
(poke) var fg3 = 71
```

Then to poke a foreground pixel would involve doing:

```
(poke) byte @ offset = fg1
(poke) byte @ offset+1#B = fg2
(poke) byte @ offset+2#B = fg3
```

At this point, you may feel that the perspective of mapping the pixels of our image is not very appealing, considering we have 5x7 = 35 pixels in our image. We will need to poke 35 * 3 = 105 bytes. We may feel tempted to, somehow, use a bigger integer to "encapsulate" the bytes. Using the bit-concatenation operator, we could do something like:

```
(poke) var bg = 255UB::255UB::255UB
(poke) var fg = 255UB::99UB::71UB
(poke) bg
(uint<24>) 0xffffff
(poke) fg
(uint<24>) 0xff6347
```

This encodes each color with a 24-bit unsigned integer. When looking at the hexadecimal values of `bg` and `fg` above, note that 0xff = 255, 0x63 = 99 and 0x47 = 71. Each byte seems to be in the right position in the 24-bit containing number. Now, poking a pixel at some given offset should be as easy as issuing just one map operation, right? Let's see, using some arbitrary offset 10#B:

```
(poke) uint<24> @ 10#B = fg
(poke) dump :from 10#B :size 4#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
0000000a: 4763 ff00                                Gc..
```

If your current endianness is little (*i.e.* you are running on a x86 system or similar) you will get the dump above. The bytes are reversed, and consequently the resulting pixel has the wrong color. Our little trick didn't work :(

So are we doomed to poke three bytes for each pixel we want to poke in our image? No, not really. The Poke language provides a construction oriented to alleviate cases like this, where several similar elements are to be "encapsulated" in a container. These constructions are called *arrays*.

Using array values, we can define the foreground and background colors like this:

```
(poke) var bga = [255UB, 255UB, 255UB]
(poke) var fga = [255UB, 99UB, 71UB]
```

All the elements on an array should be of the same kind, *i.e.* of the same type. Therefore, this is not allowed:

```
(poke) [1,"foo"]
<stdin>:1:1: error: array initializers should be of the same type
[1,"foo"];
^~~~~~~~~
```

Given an array value, it is possible to query for the number of values contained in it (called *elements*) by using the `'length` value attribute. For example:

```
(poke) bga'length
3UL
```

Tells us that the array value stored in the variable `bga` has three elements.

How can we poke an array value? We know that the map operator accepts two operands: a type specifier and the value to map. The type specifier of an array of three bytes is denoted

as `byte[3]`. Therefore, we can again try to poke a foreground pixel at offset 10#B, this time using `fga`:

```
(poke) byte[3] @ 10#B = fga
(poke) dump :from 10#B :size 4#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
0000000a: ff63 4700                                .cG.
```

This time, the bytes were written in the right order. This is because array elements are always written using their "written" ordering, with no mind to endianness. We can also map a pixel from a given offset:

```
(poke) byte[3] @ 10#B
[255UB,99UB,71UB]
```

4.2.5 Poking Lines

At this point, we could encode the 40 pixels composing the image, by issuing the same number of pokes of `byte[3]` arrays. However, we can simplify the task even further.

Our pixels are arrays of bytes, denoted by the type specifier `byte[3]`. Similarly, we could conceive arrays of 32-bit signed integers, denoted by `int[3]`, or arrays of bits, denoted by `uint<1>[3]`. But, is it possible to have arrays of other arrays? Yes, it is:

```
(poke) [[1,2],[3,4]]
```

The value above is an array of two arrays of two integers each. If we wanted to map such an array, what would be the type specifier we would need to use? It would be `int[2][2]`, which should be read from right-to-left as "array of two arrays of two integers". Let's map one from an arbitrary offset in our IO space:

```
(poke) int[2][2] @ 100#B
[[0,0],[0,0]]
```

Consider again the sequence of pixels composing the image. Using the information we have in the SBM header, we can group the pixels in the sequence into "lines". In our example image, each line contains 5 pixels. It would be natural to express each line as a sequence of pixels. The first line in our image would be:

```
(poke) var l0 = [bga,fga,fga,bga,bga]
(poke) l0
[[255UB,255UB,255UB],[255UB,99UB,71UB],...]
```

Let's complete the image lines:

```
(poke) var l0 = [bga,fga,bga,fga,bga]
(poke) var l1 = [bga,fga,bga,fga,bga]
(poke) var l2 = [bga,fga,fga,bga,bga]
(poke) var l3 = [bga,fga,bga,bga,bga]
(poke) var l4 = l3
(poke) var l5 = l4
```

Note how we exploited the fact that the three last lines of our image are identical, to avoid to write the same array thrice. Array values can be assigned, and in general manipulated, like any other kind of value, such as integers or strings.

At this point, we could poke the pixels line-by-line. What would be the type specifier for a line? A line is an array of five arrays of 3 bytes each, so the type specifier would be `byte[3][5]`. Let's do that:

```
(poke) byte[3][5] @ 5#B = l0
(poke) byte[3][5] @ 10#B = l1
(poke) byte[3][5] @ 15#B = l2
(poke) byte[3][5] @ 20#B = l3
```

```
(poke) byte[3][5] @ 25#B = l4
(poke) byte[3][5] @ 30#B = l5
(poke) byte[3][5] @ 35#B = l6
```

Not bad, we went from poking 105 bytes in the IO space to poking six lines. But we can still do better...

4.2.6 Poking Images

When we poked the lines at the end of the previous section, we had to increase the offset in every map operation. This is inconvenient.

In the same way that a sequence of bytes can be abstracted in a line, a sequence of lines can be abstracted in an image. It follows that we can look at the image data as an array of lines. But lines are themselves arrays of arrays... no matter, there is no limit on the number of arrays-of levels that you can nest.

So, let's define our image as an array of the lines defined above:

```
(poke) var image_data = [l0,l1,l2,l3,l4,l5]
(poke) image_data
[[[255UB,255UB,255UB],[255UB,99UB,71UB],[255UB,99UB,71UB]...]...]
```

What would be the type specifier for an image? It would be an array of seven arrays of five arrays of three bytes each, in other words `byte[3][5][7]`. Let's poke the pixels:

```
(poke) byte[3][5][6] @ 5#B = image_data
```

This is an example of how abstraction can simplify the handling of binary data: we switched from manipulating bytes to manipulate higher abstractions such as colors, lines and images. We achieved that by structuring the data in a way that reflects these abstractions. That's the way of the Poker.

4.2.7 Saving the Image

Now that we have completed the SBM image in our buffer `*image*`, it is time to save it to disk. For that, we can use the `save` command we are already familiar with.

We know that the SBM image starts at offset 0#B, but what is the size of its entire binary representation? The header is easy: it spans for 5 bytes. The size of the sequence of pixels can be derived from the pixels per line byte, and the number of lines byte. We know that each pixel occupies 3 bytes, so calculating...

```
(poke) var ppl = byte @ 3#B
(poke) var lines = byte @ 4#B
(poke) save :from 0#B :size 5#B + ppl#B * lines#B :file "p.sbm"
```

Note how we expressed "ppl bytes" as `ppl#B`, and "lines bytes" as `lines#B`. This is the same than expressing "10 bytes" as `10#B`. We will talk more about these united values later.

There is another way of getting the size of the stream of pixels. Recall that we have the entire set of pixels, structured as lines, stored in the variable `image_data`. Given an array, it is possible to query for its size using the `'size` *attribute*:

```
(poke) .set obase 10
(poke) [1,2,3]'size
96UL#b
```

The above indicates that the size of the array of the three integers 1, 2 and 3 is 96 bits. Using that attribute, we can also obtain the size of the pixels in the image:

```
(poke) image_data'size
720UL#b
```

And we can use it in the save command:

```
(poke) save :from 0#B :size 5#B + image_data'size :file "p.sbm"
```

Using either strategy, at this point a file named `p.sbm` should have been written in the current working directory, containing our "P is for poke" image. Keep that file around, because we will be poking it further!

4.3 Modifying SBM Images

4.3.1 Reading a SBM File

Let's open with poke the cute image we created in the last section, `p.sbm`:

```
$ poke p.sbm
[...]
(poke) dump
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 5342 4d05 07ff ffff ff63 47ff 6347 ffff  SBM......cG.cG..
00000010: ffff ffff ffff ffff 6347 ffff ffff 6347  ........cG....cG
00000020: ffff ffff ffff ff63 47ff ffff ff63 47ff  .......cG....cG.
00000030: ffff ffff ffff 6347 ff63 47ff ffff ffff  ......cG.cG.....
00000040: ffff ffff ff63 47ff ffff ffff ffff ffff  .....cG.........
00000050: ffff ffff 6347 ffff ffff ffff ffff ffff  ....cG..........
00000060: ffff ff63 47ff ffff ffff ffff ffff       ...cG.........
```

You can see the `P` in the ASCII column, right? If it wasn't for the header, it would be pictured almost straight. This is because dump shows 16 bytes per row, and our image has lines that are 15 bytes long. This is a happy coincidence: you definitely shouldn't expect to see ASCII art in the dump output of SBM files in general! :)

Now let's read the image's metadata from the header: pixels per line and how many lines are contained in the image:

```
(poke) var ppl = byte @ 3#B
(poke) ppl
5UB
(poke) var lines = byte @ 4#B
(poke) lines
7UB
```

All right, our image is 7x5. Knowing that each pixel occupies three bytes, and that each line contains `ppl` pixels, and that we have `lines` lines, we can map the entire image data using an array type specifier:

```
(poke) var image_data = byte[3][ppl][lines] @ 5#B
(poke) image_data
[[[255UB,255UB,255UB],[255UB,99UB,71UB], ...]...]
```

4.3.2 Painting Pixels

The "P is for poke" slogan was so successful and widely appraised that the recutils[1] chaps wanted to do a similar campaign "R is for recutils". For that purpose, they asked us for a tomato-colored SBM image with an `R` in it.

Our creative department got at it, and after a lot of work they came with the following design:

```
    | 0 | 1 | 2 | 3 | 4 |
```

[1] http://www.gnu.org/s/recutils

```
  +---+---+---+---+---+---+
0 |   | * | * |   |   |   |
1 |   | * |   | * |   |   |
2 |   | * |   | * |   |   |
3 |   | * | * |   |   |   |
4 |   | * | * |   |   |   |
5 |   | * |   | * |   |   |
6 |   | * |   | * |   |   |
```

Observe that this design really looks like our `P` (so much for a creative department). The bitmap has exactly the same dimensions, and difference are just three pixels, that pass from being background pixels to foreground pixels.

Therefore, it makes sense to read our `p.sbm` and use it as a base, completing the missing pixels. We saw in the last section how to read a SBM image. This time, however, we will copy the image first to a memory IO space to avoid overwriting `p.sbm`:

```
$ poke p.sbm
[...]
(poke) .mem scratch
The current IOS is now ‘*scratch*’.
(poke) .info ios
  Id Type Mode Bias Size Name
* #1 MEMORY 0x00000000#B 0x00001000#B *scratch*
  #0 FILE rw 0x00000000#B 0x0000006e#B ./p.sbm
(poke) copy :from_ios 0 :from 0#B :to 0#B :size iosize (0)
(poke) dump
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 5342 4d05 07ff ffff ff63 47ff 6347 ffff  SBM......cG.cG..
00000010: ffff ffff ffff ffff 6347 ffff ffff 6347  ........cG....cG
00000020: ffff ffff ffff ff63 47ff ffff ff63 47ff  .......cG....cG.
00000030: ffff ffff ffff 6347 ff63 47ff ffff ffff  ......cG.cG.....
00000040: ffff ffff ff63 47ff ffff ffff ffff ffff  .....cG.........
00000050: ffff ffff 6347 ffff ffff ffff ffff ffff  ....cG..........
00000060: ffff ff63 47ff ffff ffff ffff ffff 0000  ...cG...........
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Good. Now let's map the contents of the image, both header information and the sequence of pixels:

```
(poke) var ppl = byte @ 3#B
(poke) var lines = byte @ 4#B
(poke) var image_data = byte[3][ppl][lines] @ 5#B
```

Let's modify the image. Since the dimensions of the new image are exactly the same, the header remains the same. It is the pixel sequence that is different. We basically need to turn the pixels at coordinates `(4,2)`, `(5,3)` and `(6,3)` from background pixels to foreground pixels.

Remember how we would change the value of some integer in the IO space? First, we would map it into a variable, change the value, and then poke it back to the IO space. Something like this:

```
(poke) var n = int @ offset
(poke) n = n + 1
(poke) int @ offset = n
```

This three-steps process is necessary because in the `n = n + 1` above we are modifying the value of the variable `n`, not the integer actually stored at offset *offset* in the current IO space. Therefore we have to explicitly poke it back if we want the IO space to be updated as well.

Array values (and, as we shall see, other "composited" values) are different: when they are the result of the application of the map operator, the resulting values are *mapped* themselves.

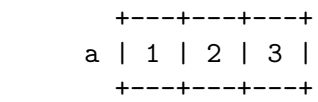
When a Poke value is mapped, updating their elements have a side effect: the area corresponding to the updated element, in whatever IO space it is mapped on, is updated as well!

Why is this? The map operator, regardless of the kind of value it is mapping, always returns a *copy* of the value found stored in the IO space. We already saw how this worked with integers. However, in Poke values are copied around using a mechanism called "shared value". This means that when a composite value like an array is copied, its elements are shared by both the original value and the new value.

We can depict this graphically for better understanding. A Poke array like:

```
(poke) var a = [1,2,3]
```

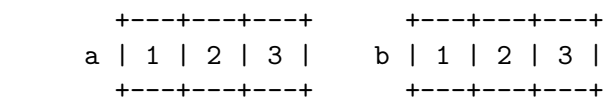
is stored in memory like this:



If we make a copy of the array in another variable `b`:

```
(poke) var b = a
```

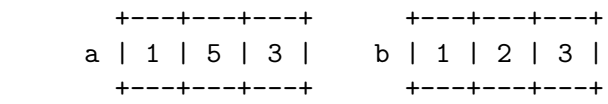
we get



Note how each of the integer elements has been copied to the new value. The resulting two arrays can then be modified independently:

```
(poke) a[1] = 5
```

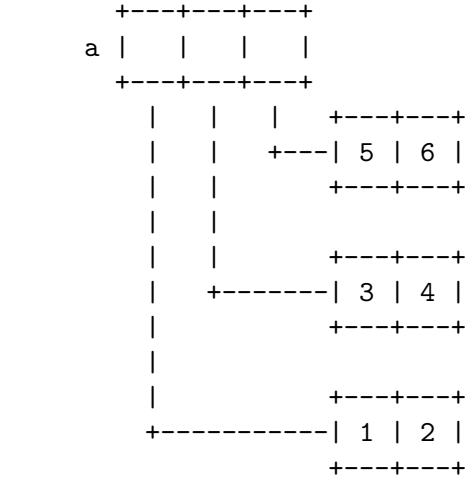
resulting in:



However, consider the following array whose elements are also arrays:

```
(poke) var a = [[1,2],[3,4],[5,6]]
```

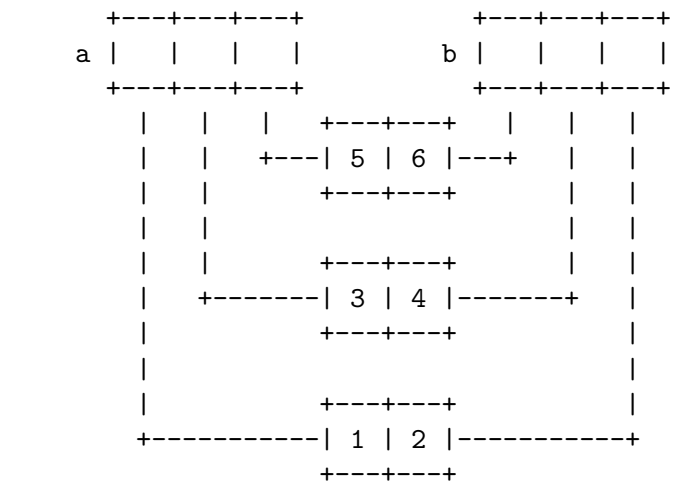
This array is stored in memory like this:



If now we make a copy of the same array into another variable `b`:

```
(poke) var b = a
```

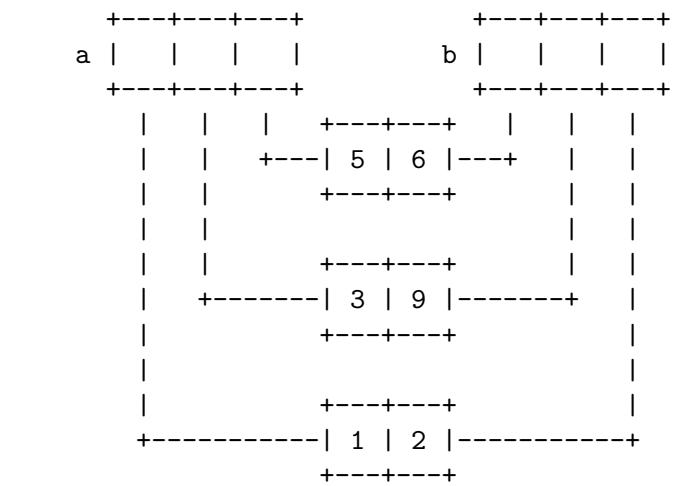
The elements of the array are copied "by shared value", *i.e.*



The elements are indeed shared between the original array and the copy! If now we modify any of the shared values, this will be reflected in both containing values:

```
(poke) a[1][1] = 9
(poke) a
[[1,2],[3,9],[5,6]]
(poke) b
[[1,2],[3,9],[5,6]]
```

or graphically:



Back to our image, it follows that if we wanted to change the color of some SBM pixel stored at offset *offset*, we would do this:

```
(poke) var a = byte[3] @ offset
(poke) a[1] = 10
```

There is no need to poke the array back explicitly: the side effect of assigning 10 to a[1] is that the byte at offset `offset+1` is poked.

Generally speaking, mapped values can be handled in exactly the same way than non-mapped values. This is actually a very central concept in poke. However, it is possible to check whether a given value is mapped or not using the `'mapped` attribute.

As we said, *simple values* such as integers and strings are never mapped, regardless of where they come from. Both `ppl` and `lines` are integers, therefore:

```
(poke) ppl'mapped
```

```
0
(poke) lines'mapped
0
```

However, `image_data` is an array that was the result of the application of a map operator, so:

```
(poke) image_data'mapped
1
```

When a value is mapped, you can ask for the offset where it is mapped, and the IO space where it is mapped, using the attributes `'offset` and `'ios` respectively. Therefore:

```
(poke) image_data'ios
1
(poke) image_data'offset
40UL#b
```

In other words, `image_data` is mapped in the IO space with id 1 (the `*scratch*` buffer) at offset 40 bits, or 5 bytes. We already knew that, because we mapped the image data ourselves, but in other situations these attributes are most useful. We shall see how later.

Well, at this point it should be clear how to paint pixels. First, let's define our background and foreground pixels:

```
(poke) var bga = [255UB, 255UB, 255UB]
(poke) var fga = [255UB, 99UB, 71UB]
```

Then, we just update the pixels in the image data using the right coordinates:

```
(poke) image_data[4][2] = fga
(poke) image_data[5][3] = fga
(poke) image_data[6][3] = fga
(poke) dump
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 5342 4d05 07ff ffff ff63 47ff 6347 ffff  SBM......cG.cG..
00000010: ffff ffff ffff ffff 6347 ffff ffff 6347  ........cG....cG
00000020: ffff ffff ffff ff63 47ff ffff ff63 47ff  .......cG....cG.
00000030: ffff ffff ffff 6347 ff63 47ff ffff ffff  ......cG.cG.....
00000040: ffff ffff ff63 47ff 6347 ffff ffff ffff  .....cG.cG......
00000050: ffff ffff 6347 ffff ffff 6347 ffff ffff  ....cG....cG....
00000060: ffff ff63 47ff ffff ff63 47ff ffff 0000  ...cG....cG.....
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

4.3.3 Cropping the R

Looking at our new image, we realize that the first and the last column are all background pixels. We are aware that the recutils project is always short of resources, so we would like to modify the image to remove these columns, *cropping* it so it looks like this:

```
  | 0 | 1 | 2 |
  +---+---+---+
0 | * | * |   |
1 | * |   | * |
2 | * |   | * |
3 | * | * |   |
4 | * | * |   |
5 | * |   | * |
6 | * |   | * |
```

In order to perform this operation we need to rework the stream of pixels to reflect the desired result, and then to update the header metadata accordingly.

4.3.4 Shortening and Shifting Lines

Let's think in term of lines. In the original image, each line has 5 pixels, that we can enumerate as:

```
     +----+----+----+----+----+
line | p0 | p1 | p2 | p3 | p4 |
     +----+----+----+----+----+
```

What we want is to crop out the first and the last column, so the resulting line would look like:

```
     +----+----+----+
line | p1 | p2 | p3 |
     +----+----+----+
```

Let's get the first line from the original `image_data`:

```
(poke) var l0 = image_data[0]
(poke) l0
[[255UB,255UB,255UB],[255UB,99UB,71UB],...]
```

We could create the corresponding cropped line, by doing something like this:

```
(poke) var cl0 = [l0[1],l0[2],l0[3]]
```

And the same for the other lines. However, Poke provides a better way to easily obtain sub portions of arrays. It is called *trimming*. Given an array like the line `l0`, we can obtain the desired portion of it by issuing:

```
(poke) l0[1:4]
[[255UB,99UB,71UB],[255UB,99UB,71UB]]
```

Note how the limits of the semi-open interval specified in the trim reflect array indexes, and hence they are 0 based. Also, the left limit is closed and the right limit is open. The result of an array trimming is always another array, which may be empty:

```
(poke) l0[1:3]
[[255UB,99UB,71UB]]
```

Armed with this new operation, we can very easily mount the sequence of pixels for our cropped image:

```
(poke) var l0 = image_data[0]
(poke) var l1 = image_data[1]
(poke) var l2 = image_data[2]
(poke) var l3 = image_data[3]
(poke) var l4 = image_data[4]
(poke) var l5 = image_data[5]
```

And then update the lines in the mapped image data:

```
(poke) image_data[0] = l0[1:4]
(poke) image_data[0] = l1[1:4]
(poke) image_data[1] = l2[1:4]
(poke) image_data[2] = l3[1:4]
(poke) image_data[3] = l4[1:4]
(poke) image_data[4] = l5[1:4]
```

4.3.5 Updating the Header

The last step is to update the header to reflect the new dimensions of the image:

```
(poke) byte[] @ 0#B = ['S','B','M']
(poke) byte @ 3#B = 3
(poke) byte @ 4#B = 7
```

And we are done. Note how this time we wrote the magic bytes as an array, to save some typing and silly manual offset arithmetic. You may have noticed that the type specifier we used this time in the map is `byte[]` instead of `byte[3]`. This type specifier denotes an array of any number of bytes, which certainly includes arrays of three bytes, like in the example.

4.3.6 Saving the Result

And finally, let's write out the new file as `r.sbm`:

```
(poke) save :from 0#B :size 5#B + image_data'size :file "r.sbm"
```

4.4 Defining Types

4.4.1 Naming your Own Abstractions

During the process of creating and manipulating SBM files we soon started talking about things like lines, pixel sequences, pixels, colors, and so on. What is more, very naturally we started *thinking* in terms of these entities: let's drop this or that line, or let's change the green component of this pixel.

Consider for example RGB colors. We know that each color is defined by three levels of light: red, green and blue. These components are also called color beams. Since each color beam has a range of 0 to 255, many formats, like SBM, use bytes to encode them.

Therefore, in the previous sections we used the type specifier `byte` when we needed to map RGB color beams, like in:

```
(poke) byte[3] @ 5#B
```

Recall that the mapping operation above means "map three bytes at the offset 5 bytes in the current IO space". But what we really want is to map color beams, not bytes!

Poke provides a way to assign names to type specifiers:

```
(poke) type RGB_Color_Beam = byte
```

The definition above tells poke that a RGB color beam is composed of a byte, *i.e.* an unsigned 8-bit integer. Any type specifier can be used at the right side of the assignment sign, and also names of already defined types. From this point on, we can map in terms of color beams:

```
(poke) RGB_Color_Beam[3] @ 5#B
```

Meaning "map three RGB color beams at the offset 5 in the current IO space".

Once a type is defined, the name can be used anywhere where a type specifier is expected.

By the way, we mentioned many times how `byte` is a synonym for `uint<8>`, `int` is a synonym for `int<32>` and so on. These synonyms are actually result of type definitions that are in the poke *standard library*. This library is loaded by poke at startup time.

4.4.2 Abstracting the Structure of Entities

Since we didn't know better, during our work with SBM images we had to remember how these entities were constructed from more simple entities such as bits and bytes, every time we needed to map them, or to poke them. For example, if we wanted to map a pixel at some particular offset, we had to issue the following command:

```
(poke) var pixel = byte[3] @ 5#B
```

Now that we made poke aware of what a RGB color beam is, we can rewrite the above as:

```
(poke) var pixel = RGB_Color_Beam[3] @ 5#B
```

This is better, but still adoleces from a big problem: what if at some point the SBM pixels get expanded to also have a transparency index, stored in a fourth byte? If that ever happens (and

will happen later in this book) then we would need to remember it before issuing commands like:

```
(poke) var image_data = RGB_Color_Beam[4][ppl][lines] @ 5#B
```

To avoid this problem, we define yet another type, this time describing the structure of a SBM pixel:

```
(poke) type SBM_Pixel = RGB_Color_Beam[3]
```

And then we can define `image_data` as a table of SBM pixels, instead of as a table of triplets of RGB color beams:

```
(poke) var image_data = SBM_Pixel[ppl][lines] @ 5#B
```

This later definition doesn't need to be changed if we change the definition of what a SBM pixel is. This is called *encapsulation* and is a very useful abstraction in computing.

4.5 Pickles

4.5.1 poke Commands versus Poke Constructions

In Section 1.2 [Nomenclature], page 3, we mentioned that poke, the program, implements a domain specific programming language called Poke, with a big p. In the examples so far we have already used the Poke language, somewhat extensively, while interacting with the program using the REPL.

For example, in:

```
(poke) 10 + 2
12
```

We are giving poke a *Poke expression* `10 + 2` to be evaluated. Once the expression is evaluated, the REPL prints the resulting value for us.

Similarly, when we define a variable or a type with `var` and `type` respectively, we are providing poke definitions to be evaluated. When we assign a value to a variable we are actually providing a Poke statement to be executed.

So the REPL accepts *poke commands*, some of which happen to be Poke expressions, definitions or statements. But we also have used dot-commands like `.file` or `.info`. These dot-commands are not part of the Poke programming language.

Every time we insert a line in the REPL and hit `enter`, poke recognizes the nature of the line, and then does the right thing. If the line is recognized as a Poke expression, for example, the Poke compiler is used to compile the statement into a routine, that is executed by the Poke Virtual Machine. The resulting value is then printed for the benefit of the user.

4.5.2 Poke Files

Poke programs are basically a collection of definitions and statements, which most often than not are stored in files, which avoids the need to type them again and again. By convention, we use the `.pk` file extension when naming files containing Poke programs.

Remember how we defined the foreground and background pixels for `p.sbm` in the REPL?

```
(poke) var bga = [255UB, 255UB, 255UB]
(poke) var fga = [255UB, 99UB, 71UB]
```

Where `bga` is a white pixel and `fga` is a tomato colored pixel. We could write these definitions in a file `colors.pk` like this:

```
var white = [255UB, 255UB, 255UB];
var tomato = [255UB, 99UB, 71UB];
```

Note that variable definitions in Poke are terminated by a semicolon (`;`) character, but we didn't need to specify them when we issued the definitions in the REPL. This is because poke adds

the trailing semicolon for us when it detects a Poke construction requiring it is introduced in the REPL.

Another difference is that Poke constructions can span for multiple lines, like in most programming languages. For example, we could have the following variable definition in a file:

```
var matrix = [[10, 20, 30],
              [40, 50, 60],
              [70, 80, 90]];
```

Once we have written our `colors.pk` file, how can we make poke aware of it? A possibility is to use the `load` construction:

```
(poke) load colors
```

Assuming a file named `colors.pk` exists in the current working directory, poke will load it and evaluate its contents. After this, we can use the colors:

```
(poke) tomato
[0xffUB,0x63UB,0x47UB]
```

Before, we would need to define these colors every time we would like to poke SBM files or, in fact, any RGB24 data. Now we just have to load the file `colors.pk` and use the variables defined there.

If you try to load a file whose name contains a dash character (`-`) you will get an error message:

```
(poke) load my-colors
<stdin>:1:8: error: syntax error, unexpected '-', expecting ';'
load my-colors;
       ^
```

This is because the argument to `load` is interpreted as a Poke identifier, and dash characters are not allowed in identifiers. To alleviate this problem, you can also specify the name of the file to load encoded in a string, line in:

```
(poke) load "my-colors.pk"
```

If you use this form of `load`, however, you are required to specify the complete name of the file, including the `.pk` file extension.

Since loading files is such a common operation, poke provides a dot-command `.load` that does auto-completion:

```
(poke) .load my-colors.pk
```

Which is equivalent to `load "my-colors.pk"`.

Since `load` is part of the Poke language, it can also be used in Poke programs stored in files. We will explore this later.

4.5.3 Pickling Abstractions

In the last section we defined a couple of RGB colors `white` and `tomato` in a file called `colors.pk`. If we keep adding colors to the file, we may end with a nice collection of colors that are available to us at any time, by just loading the file.

Since there are many ways to understand the notion of "color", and also many ways to implement these many notions, it would be better to be more precise and call our file `rgb24.pk`, since the notion of color we are using is of that RGB24. While are at it, let's also rename the variables to reflect the fact they denote not just any kind of colors, but RGB24 colors:

```
var rgb24_white = [255UB, 255UB, 255UB];
var rgb24_tomato = [255UB, 99UB, 71UB];
```

At this point, we can also add the definitions of a couple of types to our `rgb24.pk`:

```
type RGB_Color_Beam = byte;
```

```
type RGB24_Color = RGB_Color_Beam[3];

var rgb24_white = [255UB, 255UB, 255UB];
var rgb24_tomato = [255UB, 99UB, 71UB];
```

Any time we want to manipulate RGB24 colors we can just load the file `rgb24.pk` and use these types and variables.

In poke parlance we call files like the above, that contain definitions of conceptually related entities, *pickles*. Pickles can be very simple, like the `rgb24.pk` sketched above, or fairly complicated like `dwarf.pk`.

It is common for pickles to `load` other pickles. For example, if we were to write a SBM pickle, we would load the RGB24 pickle from it:

```
load rgb24;

[... SBM definitions ...]
```

This way, when we `load sbm` from the repl, the dependencies get loaded as well.

GNU poke includes several already written pickles for commonly used file formats, and other domains. The `load` construction knows where these pickles are installed, so in order to load the pickle to manipulate ELF files, for example, all you have to do is to:

```
(poke) load elf
```

4.5.4 Exploring Pickles

As we have seen, a pickle provides Poke variables, types and functions related to some definite domain. Let's say we are interested in editing an ELF file. We know that GNU poke comes with a pre-installed pickle named `elf.pk`. We can load it like this:

```
(poke) load elf
```

By convention, the entities provided by a pickle `foo.pk` are usually prefixed like:

- Variables, units and functions are prefixed with `foo_`.
- Types are prefixed with `Foo_`.

Therefore, once the pickle is loaded we can use the `.info` dot-command in order to get a glimpse of the functionality provided by the pickle. See Section 16.12 [info command], page 106. Example:

```
(poke) .info variables elf
Name            Declared at
elf_stb_names   elf-common.pk:53
elf_stt_names   elf-common.pk:72
elf_stv_names   elf-common.pk:84
```

The above tells us that the `elf.pk` pickle provides these three variables. Types are way more interesting:

```
(poke) .info types Elf64
Name                Declared at
Elf64_Ehdr          elf-64.pk:184
Elf64_Off           elf-64.pk:26
Elf64_SectionFlags  elf-64.pk:130
Elf64_Shdr          elf-64.pk:152
Elf64_RelInfo       elf-64.pk:36
Elf64_Chdr          elf-64.pk:85
Elf64_Rela          elf-64.pk:61
Elf64_Phdr          elf-64.pk:169
```

```
Elf64_Sxword         elf-64.pk:24
Elf64_Sym            elf-64.pk:71
Elf64_Rel            elf-64.pk:45
Elf64_File           elf-64.pk:207
Elf64_Addr           elf-64.pk:25
Elf64_Group          elf-64.pk:121
Elf64_Dyn            elf-64.pk:98
Elf64_Xword          elf-64.pk:23
```

We immediately notice the `Elf64_File`. Looks like what we would need to map in order to access the entire ELF data in some given IO space (like a file.) We can ask for more details about it:

```
(poke) .info type Elf64_File
Class:      struct
Name:       Elf64_File
Complete:   no
Fields:
  Name     Type
  ehdr     Elf64_Ehdr
  shdr     Elf64_Shdr[]
  phdr     Elf64_Phdr[]
Methods:
  Name                    Type
  get_section_name        (offset<Elf_Word,8>)string
  get_symbol_name         (Elf64_Shdr,,offset<Elf_Word,8>)string
  get_sections_by_name    (string)Elf64_Shdr[]
  get_sections_by_type    (Elf_Word)Elf64_Shdr[]
  section_name_p          (string)int
  get_string              (offset<Elf_Word,8>)string
  get_group_signature     (Elf64_Shdr)string
  get_group_signatures    ()string[]
  get_section_group       (string)Elf64_Shdr[]
```

We see that an `Elf64_File` has three fields, and the nature of these files: a header `ehdr`, an array of section headers `shdr` and an array of program headers `phdr`.

The type description above also lists the methods defined in the type. Looking at the method name and its type is very usually revealing enough to use it. For example, we see that `get_sections_by_name` returns an array of section headers, characterized by some given section name:

```
(poke) var elf = Elf64_File @ 0#B
(poke) elf.get_sections_by_name (".text")
[Elf64_Shdr {
  sh_name=0x1bU#B,
  sh_type=0x1U,
  sh_flags=#<ALLOC,EXECINSTR>,
  sh_addr=0x0UL#B,
  sh_offset=0x40UL#B,
  sh_size=0xbUL#B,
  sh_link=0x0U,
  sh_info=0x0U,
  sh_addralign=0x1UL,
  sh_entsize=0x0UL#b
}]
```

Ultimately, of course the best way to see what a pickle provides is to go read its source code.

4.5.5 Startup

When poke starts it loads the file `.pokerc` located in our home directory, if it exists. This initialization file contains poke commands, one per line.

If we wanted to get some Poke file loaded at startup, we could do it by adding a load command to our `.pokerc`. For example:

```
# My poke configuration - jemarch
[...]
.load ~/.poke.d/mydefs.pk
```

4.6 Poking Structs

4.6.1 Heterogeneous Related Data

Let's recap the structure of the header of a Stupid BitMap:

```
                SBM header
+-------+-------+-------+-------+-------+
|  'S'  |  'B'  |  'M'  |  ppl  | lines |
+-------+-------+-------+-------+-------+
  byte0   byte1   byte2   byte3   byte4
```

The header is composed of five fields, which actually compose three different logical *fields*: a magic number, the number of pixels per line, and the number of lines.

We could of course abstract the header using an array of five bytes, like this:

```
type SBM_Header = byte[5];
```

However, this would not capture the properties of the fields themselves, which would need to be remembered by the user: which of these five bytes correspond to the magic number? Is the pixels per line number signed or unsigned? *etc.*

Poke provides a much better way to abstract collections of heterogeneous data: *struct types*. Using a struct type we can abstract the SBM header like this:

```
type SBM_Header =
 struct
 {
   byte[3] magic;
   uint<8> ppl;
   uint<8> lines;
 }
```

Note how the struct has three named fields: `magic`, `ppl` and `lines`. `magic` is an array of three bytes, while `ppl` and `lines` are both unsigned integers.

4.6.2 Mapping Structs

Once defined, struct types can be referred by name. For example, we can map the SBM header at the beginning of our file `p.sbm` like this:

```
(poke) SBM_Header @ 0#B
SBM_Header {
  magic=[0x53UB,0x42UB,0x4dUB],
  ppl=0x5UB,
  lines=0x7UB
}
```

The value resulting from the mapping is a struct value. The fields of struct values are accessed using the familiar dot-notation:

```
(poke) var header = SBM_Header @ 0#B
(poke) header.ppl * header.lines
35UB
```

The total number of pixels in the image is 35. Note how both `header.ppl` and `header.lines` are indeed unsigned byte values, and thus the result of the multiplication is also an unsigned byte. This could be problematic if the image contained more than 255 pixels, but this can be prevented by using a cast:

```
(poke) header.ppl as uint * header.lines
35U
```

Now the second operand `header.lines` is promoted to a 32-bit unsigned value before the multiplication is performed. Consequently, the result of the operation is also 32-bit wide (note the suffix of the result.)

4.6.3 Modifying Mapped Structs

Remember when we wanted to crop a SBM image by removing the first and last row? We updated the header in a byte by byte manner, like this:

```
(poke) byte @ 3#B = 3
(poke) byte @ 4#B = 7
```

Now that we have the header mapped in a variable, updating it is much more easy and convenient. The dot-notation is used to update the contents of a struct field, by placing it at the left hand side of an assignment:

```
(poke) header.ppl = 3
(poke) header.lines = 7
```

This updates the pixel per line and the number of lines, in the IO space:

```
(poke) dump :size 5#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 5342 4d03 07                             SBM..
```

4.7 How Structs are Built

First we need to define some structure to use as an example. Let's say we are interesting in poking Packets, as defined by the Packet Specification 1.2 published by the Packet Foundation (none less).

In a nutshell, each Packet starts with a byte whose value is always `0xab`, followed by a byte that defines the size of the payload. A stream of bytes conforming the payload follows, themselves followed by another stream of the same number of bytes with "control" values.

We could translate this description into the following Poke struct type definition:

```
type Packet =
  struct
  {
    byte magic = 0xab;
    byte size;
    byte[size] payload;
    byte[size] control;
  };
```

There are some details described by the fictitious Packet Specification 1.2 that are not covered in this simple definition, but we will be attending to that later in this manual.

So, given the definition of a struct type like Packet, there are only two ways to build a struct value in Poke.

One is to map it from some IO space. This is achieved using the map operator:

```
(poke) Packet @ 12#B
Packet {
  magic = 0xab,
  size = 2,
  payload = [0x12UB,0x30UB],
  control = [0x1UB,0x1UB]
}
```

The expression above maps a Packet starting at offset 12 bytes, in the current IO space. See the Poke manual for more details on using the map operator.

The second way to build a struct value is to _construct_ one, specifying the value to some, all or none of its fields. It looks like this:

```
(poke) Packet {size = 2, payload = [1UB,2UB]}
Packet {
  magic = 0xab,
  size = 2,
  payload = [0x1UB,0x2UB],
  control = [0x0UB,0x0UB]
}
```

In either case, building a struct involves to determine the value of all the fields of the struct, one by one. The order in which the struct fields are built is determined by the order of appearance of the fields in the type description.

In our example, the value of `magic` is determined first, then `size`, `payload` and finally `control`. This is the reason why we can refer to the values of previous fields when defining fields, such as in the size of the `payload` array above, but not the other way around: by the time `payload` is mapped or constructed, the value of `size`, has already been mapped or constructed.

What happens behind the curtains is that when poke finds the definition of a struct type, like Packet, it compiles two functions from it: a mapper function, and a constructor function. The mapper function gets as arguments the IO space and the offset from which to map the struct value, whereas the constructor function gets the template specifying the initial values for some, or all of the fields; reasonable default values (like zeroes) are used for fields for which no initial values have been specified.

These functions, mapper and constructor, are invoked to create fresh values when a map operator @ or a struct constructor is used in a Poke program, or at the poke prompt.

4.8 Variables in Structs

Fields are not the only entity that can appear in the definition of a struct type.

Suppose that after reading more carefully the Packet Specification 1.2 (that spans for several thousand of pages) we realize that the field `size` doesn't really store the number of bytes of the payload and control arrays, like we thought initially. Or not exactly: the Packet Foundation says that if `size` has the special value 0xff, then the size is zero.

We could of course do something like this:

```
type Packet =
  struct
  {
    byte magic = 0xab;
```

```
    byte size;

    byte[size == 0xff ? 0 : size] payload;
    byte[size == 0xff ? 0 : size] control;
  };
```

However, we can avoid replicating code by using a variable instead:

```
type Packet =
  struct
  {
    byte magic = 0xab;
    byte size;

    var real_size = (size == 0xff ? 0 : size);

    byte[real_size] payload;
    byte[real_size] control;
  };
```

Note how the variable can be used after it gets defined. In the underlying process of mapping or constructing the struct, the variable is incorporated into the lexical environment. Once defined, it can be used in constraint expressions, array sizes, and the like. We will see more about this later.

Incidentally, it is of course possible to use global variables as well. For example:

```
var packet_special = 0xff;
type Packet =
  struct
  {
    byte magic = 0xab;
    byte size;

    var real_size = (size == packet_special ? 0 : size);

    byte[real_size] payload;
    byte[real_size] control;
  };
```

In this case, the global `packet_special` gets captured in the lexical environment of the struct type (in reality in the lexical environment of the implicitly created mapper and constructor functions) in a way that if you later modify `packet_special` the new value will be used when mapping/constructing *new* values of type Packet. Which is really cool, but let's not get distracted from the main topic... :)

4.9 Functions in Structs

Further reading of the Packet Specification 1.2 reveals that each Packet has an additional `crc` field. The content of this field is derived from both the payload bytes and the control bytes.

But this is no vulgar CRC we are talking about. On the contrary, it is a special function developed by the CRC Foundation in partnership with the Packet Foundation, called superCRC (patented, TM).

Fortunately, the CRC Foundation distributes a pickle `supercrc.pk`, that provides a `calculate_crc` function with the following spec:

```
fun calculate_crc = (byte[] data, byte[] control) int:
```

So let's use the function like this in our type, after loading the supercrc pickle:

```
load supercrc;

type Packet =
  struct
  {
    byte magic = 0xab;
    byte size;

    var real_size = (size == 0xff ? 0 : size);

    byte[real_size] payload;
    byte[real_size] control;

    int crc = calculate_crc (payload, control);
  };
```

However, there is a caveat: it happens that the calculation of the CRC may involve arithmetic and division, so the CRC Foundation warns us that the `calculate_crc` function may raise E_div_by_zero. However, the Packet 1.2 Specification tells us that in these situations, the `crc` field of the packet should contain zero. If we used the type above, any exception raised by `calculate_crc` would be propagated by the mapper/constructor:

```
(poke) Packet @ 12#B
unhandled division by zero exception
```

A solution is to use a function that takes care of the extra needed logic, wrapping calculate_crc:

```
load supercrc;

type Packet =
  struct
  {
    byte magic = 0xab;
    byte size;

    var real_size = (size == 0xff ? 0 : size);

    byte[real_size] payload;
    byte[real_size] control;

    fun corrected_crc = int:
    {
      try return calculate_crc (payload, control);
      catch if E_div_by_zero { return 0; }
    }

    int crc = corrected_crc;
  };
```

Again, note how the function is accessible after its definition. Note as well how both fields and variables and other functions can be used in the function body. There is no difference to define variables and functions in struct types than to define them inside other functions or in the top-level environment: the same lexical rules apply.

4.10 Struct Methods

At this point you may be thinking something on the line of "hey, since variables and functions are also members of the struct, I should be able to access them the same way than fields, right?".

So you will want to do:

```
(poke) var p = Packet  12#B
(poke) p.real_size
(poke) p.corrected_crc
```

But sorry, this won't work.

To understand why, think about the struct building process we sketched above. The mapper and constructor functions are derived/compiled from the struct type. You can imagine them to have prototypes like:

```
Packet_mapper (IOspace, offset) -> Packet value
Packet_constructor (template)   -> Packet value
```

You can also picture the fields, variables and functions in the struct type specification as being defined inside the bodies of Packet_mapper and Packet_constructor, as their contents get mapped/constructed. For example, let's see what the mapper does:

```
Packet_mapper:

  . Map a byte, put it in a local 'magic'.
  . Map a byte, put it in a local 'size'.
  . Calculate the real size, put it in a local 'real_size'.
  . Map an array of real_size bytes, put it in a local 'payload'.
  . Map an array of real_size bytes, put it in a local 'control'.
  . Compile a function, put it in a local 'corrected_crc'.
  . map a byte, call the function in the local 'corrected_crc',
    complain if the values are not the same, otherwise put the
    mapped byte in a local 'crc'.
  . Build a struct value with the values from the locals 'magic',
    'size', 'payload', 'control' and 'crc', and return it.
```

The pseudo-code for the constructor would be almost identical. Just replace "map a byte" with "construct a byte".

So you see, both the values for the mapped fields and the values for the variables and functions defined inside the struct type end as locals of the mapping process, but only the values of the fields are actually put in the struct value that is returned in the last step.

This is where methods come in the picture. A method looks very similar to a function, but it is not quite the same thing. Let me show you an example:

```
load supercrc;

type Packet =
  struct
  {
    byte magic = 0xab;
    byte size;

    var real_size = (size == 0xff ? 0 : size);

    byte[real_size] payload;
    byte[real_size] control;
```

```
    fun corrected_crc = int:
    {
      try return calculate_crc (payload, control);
      catch if E_div_by_zero { return 0; }
    }

    int crc = corrected_crc;

    method c_crc = int:
    {
      return corrected_crc;
    }
  };
```

We have added a method `c_crc` to our Packet struct type, that just returns the corrected superCRC (patented, TM) of a packet. This can be invoked using dot-notation, once a Packet value is mapped/constructed:

```
(poke) var p = Packet  12#B
(poke) p.c_crc
0xdeadbeef
```

Now, the important bit here is that the method returns the corrected crc *of a* `Packet`. That's it, it actually operates on a Packet value. This Packet value gets implicitly passed as an argument whenever a method is invoked.

We can visualize this with the following "pseudo Poke":

```
method c_crc = (Packet SELF) int:
{
   return SELF.corrected_crc;
}
```

Fortunately, poke takes care to recognize when you are referring to fields of this implicit struct value, and does The Right Thing(TM) for you. This includes calling other methods:

```
method foo = void: { ... }
method bar = void:
{
 [...]
 foo;
}
```

The corresponding "pseudo-poke" being:

```
method bar = (Packet SELF) void:
{
 [...]
 SELF.foo ();
}
```

It is also possible to define methods that modify the contents of struct fields, no problem:

```
var packet_special = 0xff;

type Packet =
  struct
  {
    byte magic = 0xab;
    byte size;
```

```
  [...]

  method set_size = (byte s) void:
  {
    if (s == 0)
      size = packet_special;
    else
      size = s;
  }
};
```

This is what is commonly known as a *setter*. Note, incidentally, how a method can also use regular variables. The Poke compiler knows when to generate a store in a normal variable such as `packet_special`, and when to generate a set to a `SELF` field.

Given the different nature of the variables, functions and methods, there are a couple of restrictions:

- Functions can't set fields defined in the struct type.

  This will be rejected by the compiler:

  ```
  type Foo =
    struct
    {
       int field;
       fun wrong = void: { field = 10; }
    };
  ```

  Remember the construction/mapping process. When a function accesses a field of the struct type like in the example above, it is not doing one of these pseudo `SELF.field = 10`. Instead, it is simply updating the value of the local created in this step in Foo_mapper:

  ```
  Foo_mapper:

   . Map an int, put it in a local 'field'.
   . [...]
  ```

  Setting that local would impact the mapping of the subsequent fields if they refer to `field` (for example, in their constraint expression) but it wouldn't actually alter the value of the field `field` in the struct value that is created and returned from the mapper!

  This is very confusing, so we just disallow this with a compiler error "invalid assignment to struct field", for your own sanity 8-)

- Methods can't be used in field constraint expressions, nor in variables or functions defined in a struct type.

  How could they be? The field constraint expressions, the initialization expressions of variables, and the functions defined in struct types are all executed as part of the mapper/constructor and, at that time, there is no struct value yet to pass to the method.

  If you try to do this, the compiler will greet you with an "invalid reference to struct method" message.

Something to keep in mind about methods is that they can destroy the integrity of the data stored in a struct. Consider for example the following struct type:

```
type Packet =
  struct
  {
    byte magic = 0xab;
    byte size : size <= 4096;
```

```
  [...]

  method set_size = (byte s) void:
  {
    if (s == 0)
      size = packet_special;
    else
      size = s;
  }
};
```

Observe how this new version of `Packet` has an additional constraint that specifies `size` should not exceed 4096. However, when the method `set_size` is executed the constraints are not checked again. This is useful at times, but can also lead to unintended data corruption.

A solution for this problem is to make methods aware of the restrictions. Like in this case:

```
type Packet =
  struct
  {
    var MAXSIZE = 4096;

    byte magic = 0xab;
    byte size : size <= MAXSIZE;
    [...]

    method set_size = (byte s) void:
    {
      if (s == 0)
        size = packet_special;
      else if (s > MAXSIZE)
        raise E_inval;
      else
        size = s;
    }
  };
```

Note how we use a variable `MAXSIZE` in order to avoid hard-coding 4096 twice in the struct definition.

4.11 Padding and Alignment

It is often the case in binary formats that certain elements are separated by some data that is not really used for any meaningful purpose other than occupy that space. The reason for keeping that space varies from case to case; sometimes to reserve it for future use, sometimes to make sure that the following data is aligned to some particular alignment. This is known as *padding*. There are several ways to implement padding in GNU poke. This article shows these techniques and discusses their advantages and disadvantages.

4.11.1 Esoteric and exoteric padding

Padding is the technique of keeping some amount of space between two different elements in some data stream. GNU poke provides two different ways to express sequences of data elements: the fields of a struct type, which are defined one after the other, and elements in an array.

We call adding space between two struct fields *esoteric* (or internal) padding.

We call adding space between two array elements *exoteric* (or external) padding.

The following sections contain examples of the two kinds of padding and how to better handle them in Poke.

4.11.2 Reserved fields

People designing binary encoded formats tend to be cautious and try to avoid future backward incompatibilities by keeping some unused fields that are reserved for future use. This is the first kind of padding we will be looking at, and is particularly common in structures like headers.

See for example the header used to characterize compressed section contents in ELF files:

```
type Elf64_Chdr =
  struct
  {
    Elf_Word ch_type;
    Elf_Word ch_reserved;
    offset<Elf64_Xword,B> ch_size;
    offset<Elf64_Xword,B> ch_addralign;
  };
```

where the `ch_reserved` field is reserved for future use. When the time comes the space occupied by that field (32 bits in this case) will be used to hold additional data in the form of one or more fields. The idea is that implementations of the older formats will still work.

The most obvious way to handle this in Poke is using a named field like `ch_reserved` above. This field will be decoded/encoded by poke when constructing/mapping/writing struct values of this type, and will be available to the user as `chdr.ch_reserved`.

Sometimes reserved space is required to be filled with certain data values, such as zeroes. This may be to simplify things, or to force data producers to initialize the memory in order to avoid potential leaking of sensible information. In these cases we can use Poke initial values:

```
type Elf64_Chdr =
  struct
  {
    Elf_Word ch_type;
    Elf_Word ch_reserved = 0;
    offset<Elf64_Xword,B> ch_size;
    offset<Elf64_Xword,B> ch_addralign;
  };
```

This will make poke to check that `ch_reserved` is zero when constructing or mapping headers for compressed sections raising a constraint violation exception otherwise. It will also make poke to make sure `ch_reserved` to 0 when constructing `Elf64_Chdr` struct values:

```
(poke) Elf64_Chdr { ch_reserved = 23 }
unhandled constraint violation exception
```

An alternative way to characterize reserved space in Poke is to use anonymous fields. For example:

```
type Elf64_Chdr =
  struct
  {
    Elf_Word ch_type;
    Elf_Word;
    offset<Elf64_Xword,B> ch_size;
    offset<Elf64_Xword,B> ch_addralign;
  };
```

Using Poke anonymous fields to implement reserved fields has at least two advantages. First, the user cannot anymore temper with the data in the reserved space in an easy way, *i.e.* `chdr.ch_reserved = 666` is no longer valid. Second, the printed representation of anonymous struct fields is more compact and denotes better than the involved space is not to be messed with:

```
(poke) Elf64_Chdr {}
Elf64_Chdr {
  ch_type=0x0U,
  0x0U,
  ch_size=0x0UL#B,
  ch_addralign=0x0UL#B
}
```

A disadvantage of using anonymous fields is that you cannot specify constraint expressions for them, nor initial values. At some point we will probably add syntax to declare certain struct fields as read-only.

At this point, it is important to note that anonymous fields are still encoded/decoded by poke every time the struct value is mapped or written, exactly like regular fields. Therefore using them doesn't pose any advantage in terms of performance.

4.11.3 Payloads

The reserved fields discussed in the previous section are most often discrete units like words, double-words, and the like, they are usually of some fixed size, and they are used to delimit some space that is not to be used.

Another kind of padding happens when an entity contains space to be used to store some kind of payload whose contents are not determined. This would be such an example:

```
type Packet =
  struct
  {
    offset<uint<32>,B> payload_size;
    byte[payload_size] payload;
    int flags;
  };
```

In this example we are using a `payload` field which is an array of bytes. The size of the payload is determined by the packet header, and the contents are not determined. Of course this assumes that the payload sizes are divisible in whole bytes; a bit-oriented format may need to use an array of bits instead.

This approach of using a byte (or bit) array like in the example above has the advantage of providing a field with the bytes (or bits) to the user, for inspection and modification:

```
(poke) packet.payload
[23UB, ...]
(poke) packet.payload[0] = 0
```

The user can still map whatever payload structure in that space using the attributes of a mapped `Packet`. For example, if the packet contains an array of ULEB128 numbers, we could do:

```
(poke) var numbers = ULEB128[packet.payload'size] @ packet.payload'offset
```

But this approach has a disadvantage: every time the packet structure is mapped or written the entire payload array gets decoded and encoded. If the payloads are big enough (think about the data blocks of a file described by a file system i-node for example) this can be a big problem in terms of performance.

Another problem of using byte (or bit) arrays for payloads is that the printed representation of the struct values include the contents of the arrays, and most often the user won't be interested in seeing that:

```
(poke) Packet { payload_size = 23#B }
Packet {
  payload_size=0x17U#B,
  payload=[0x0UB,0x0UB,0x0UB,0x0UB,0x0UB,...],
  flags=0x0
}
```

Another alternative is to implement the padding implied by a payload using field labels:

```
type Packet =
  struct
  {
    offset<uint<32>,B> payload_size;:
    int flags @ OFFSET + payload_size;
  };
```

Note how a `payload` field no longer exists in the struct type, and the field `flags` is defined to start at offset `OFFSET + payload_size`. This way no explicit array is encoded/decoded when manipulating `Packet` values:

```
(poke) .set omaps yes
(poke) Packet { payload_size = 500#Mb }
Packet {
  payload_size=62500000U#B @ 0UL#b,
  flags=0 @ 4000000032UL#b
} @ 0UL#b
```

In this example we used the `omaps` option, which asks poke to print the offsets of the fields. The offset of `flags` is 4000000032 bits, or 500 megabytes:

```
(poke) 4000000032UL #b/#MB
500UL
```

Mapping this new `Packet` involves reading and decoding five bytes, for the `payload_size` and `flags` only. This is clearly much faster and avoids unneeded IO.

However you may be wondering, if there is no explicit `payload` field, how to access the payload space? A way is to define a method to the struct to provide the payload attributes:

```
type Packet =
  struct
  {
    offset<uint<32>,B> payload_size;:
    var payload_offset = OFFSET;
    int flags @ OFFSET + payload_size;

    method get_payload_offset = off64:
    {
      return payload_offset;
    }
  };
```

Note how we captured the offset of the payload using a variable in the strict type definition. Returning `OFFSET` in `get_payload_offset` wouldn't work for obvious reasons: in the body of the method `OFFSET` evaluates to the end of `flags` in this case.

Using this method you can easily access the payload (again as an array of ULEB128 numbers) like this:

```
var numbers = ULEB128[packet.payload_size @ packet.get_payload_offset
```

Finally, using labels for this purpose makes the printed representation of the struct values more readable by not including the payload bytes in it:

```
(poke) Packet {}
Packet {
  payload_size=0x0U#B,
  flags=0x0
}
```

4.11.4 Aligning struct fields

Another kind of esoteric padding happens when certain fields in entities are required to be aligned to some particular alignment. For example, suppose that the `flags` field in the packets used in the previous sections is required to always be aligned to 4 bytes regardless of the size of the payload. This would be a common requirement if the format is intended to be implemented in systems where data is to be accessed using its "natural" alignment.

Using explicit fields for both the payload and the additional padding, we could come with:

```
type Packet =
  struct
  {
    offset<uint<32>,B> payload_size;
    byte[payload_size] payload;
    byte[alignto (OFFSET, 4#B)] padding;
    int flags;
  };
```

Where `alignto` is a little function defined in the Poke standard library, like this:

```
fun alignto = (uoff64 offset, uoff64 to) uoff64:
{
  return (to - (offset % to)) % to;
}
```

Alternatively, using the labels approach (which is generally better as we discussed in the last section) the definition would become:

```
type Packet =
  struct
  {
    offset<uint<32>,B> payload_size;:
    var payload_offset = OFFSET;
    int flags @ OFFSET + payload_size + alignto (payload_size, 4#B);

    method get_payload_offset = off64:
    {
      return payload_offset;
    }
  };
```

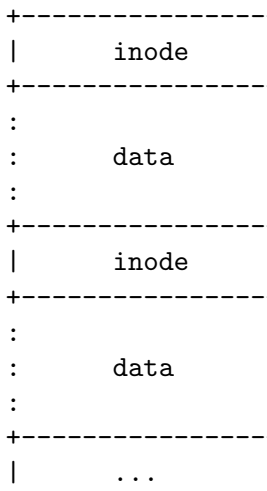
In this case, the payload space is still completely characterized by the `payload_size` field and the `get_payload_offset` method.

4.11.5 Padding array elements

Up to now all the examples of padding we have shown are in the category of esoteric or internal padding, *i.e.* it was intended to add space between fields of some particular entity.

However, sometimes we want to specify some padding between the elements of a sequence of entities. In Poke this basically means an array.

Suppose we have a simple file system that is conformed by a sequence of inodes. The contents of the file system have the following form:



That's it, each inode describes a block of data of variable size that immediately follows. Then more pairs of inode-data follow until the end of the device. However, a requirement is that each inode has to be aligned to 128 bytes.

Let's start by writing a simple type definition for the inodes:

```
type Inode =
  struct
  {
    string filename;
    int perms;
    offset<uint<32>,B> data_size;
  };
```

This definition is simple enough, but it doesn't allow us to just map an array of inodes like this:

```
(poke) Inode[]  0#B
```

We could of course add the data and padding explicitly to the inode structure:

```
type Inode =
  struct
  {
    string filename;
    int perms;
    offset<uint<32>,B> data_size;
    byte[data_size] data;
    byte[alignto (data_size, 128#B) padding;
  };
```

Then we could just map `Inode[] @ 0#B` and we would the get expected result.

But this is not a good idea. On one hand because, as we know, this would imply mapping the full file system data byte by byte, and that would be very very slow. On the other hand, because the data is not part of the inode, conceptually speaking.

A better solution is to use this idiom:

```
type Inode =
  struct
  {
    string filename;
    int perms;
    offset<uint<32>,B> data_size;

    byte[0] @ OFFSET + data_size + alignto (data_size, 128#B);
  };
```

This uses an anonymous field at the end of the struct type, of size zero, located at exactly the offset where the data plus padding would end in the version with explicit fields.

This later solution is fast and still allows us to get an array of inodes reflecting the whole file system with:

```
(poke) var inodes = Inode[] @ 0#B
```

Like in the previous sections, a method =get_data_offset= can be added to the struct type in order to allow accessing the data blocks corresponding to a given inode.

4.12 Dealing with Alternatives

4.12.1 BSON

BSON[2] is a binary encoding for JSON documents. The top-level entity described by the spec is the `document`, which contains the total size of the document, a sequence of elements, and a trailing null byte.

We can describe the above in Poke with the following type definition:

```
type BSON_Doc =
  struct
  {
    offset<int<32>,B> size;
    BSON_Elem[size - size'size - 1#B] elements;
    byte endmark : endmark == 0x0;
  };
```

BSON elements come in different kinds, which correspond to the different types of JSON entities: 32-bit integers, 64-bit integers, strings, arrays, timestamps, and so on. Every element starts with a tag, which is a 8-bit unsigned integer that identifies it's kind, and a name encoded as a NULL-terminated string. What comes next depends on the kind of element.

The following Poke type definition describes a subset of BSON elements, namely integers, big integers and strings:

```
type BSON_Elem =
  struct
  {
    byte tag;
    string name;

    union
    {
      int32 integer32 : tag == 0x10;
```

[2] http://bsonspec.org/spec.html

```
      int64 integer64 : tag == 0x12;
      BSON_String str : tag == 0x02;
    } value;
};
```

The union in `BSON_Elem` corresponds to the variable part. When poke decodes an union, it tries to decode each alternative (union field) in turn. The first alternative that is successfully decoded without raising a constraint violation exception is the chosen one. If no alternative can be decoded, a constraint violation exception is raised.

To see this process in action, let's use the BSON corresponding to the following little JSON document:

```
{
    "name" : "Jose E. Marchesi",
    "age" : 40,
    "big" : 1076543210012345
}
```

Let's take a look to the different elements:

```
(poke) .load bson.pk
(poke) var d = BSON_Doc  0#B
(poke) d.elements'length
0x3UL
(poke) d.elements[0]
BSON_Elem {
  tag=0x2UB,
  name="name",
  value=struct {
    str=BSON_String {
      size=0x11,
      value="Jose E. Marchesi",
      chars=[0x4aUB,0x6fUB,0x73UB,0x65UB...]
    }
  }
}
(poke) d.elements[1]
BSON_Elem {
  tag=0x10UB,
  name="age",
  value=struct {
    integer32=0x27
  }
}
(poke) d.elements[2]
BSON_Elem {
  tag=0x12UB,
  name="big",
  value=struct {
    integer64=0x3d31c3f9e3eb9L
  }
}
```

Note how unions decode into structs featuring different fields. What field is available depends on the alternative chosen while decoding.

In the session above, `d.elements[1].value` contains an `integer32` field, whereas `d.elements[2].value` contains an `integer64` field. Let's see what happens if we try to access the wrong field:

```
(poke) d.elements[1].value.integer64
unhandled invalid element exception
```

We get a run-time exception. This kind of errors cannot be catched at compile time, since both `integer32` and `integer64` are potentially valid fields in the union value.

4.12.2 Unions are Tagged

Unlike in C, Poke unions are tagged. Unions have their own lexical closures, and it is the capured values that determine what field is chosen at every time. Wherever the union goes, its tag accompanies it.

To see this more clearly, consider the following alternative Poke description of the BSON elements:

```
type BSON_Elem =
  union
  {
    struct
    {
      byte tag : tag == 0x10;
      string name;
      int32 value;
    } integer32;

    struct
    {
      byte tag : tag == 0x12;
      string name;
      int64 value;
    } integer64;

    struct
    {
      byte tag : tag == 0x12;
      string name;
      BSON_String value;
    } str;
  };
```

This description is way more verbose than the one used in previous sections, but it shows a few important properties of Poke unions.

First, the constraints guiding the decoding are not required to appear in the union itself: it is a recursive process. In this example, `BSON_String` could have constraints on it's own, and these constraints will also impact the decoding of the union.

Second, there are generally many different ways to express the same plain binary using different type structures. This is no different than getting different parse trees from the same sequence of tokens using different grammars denoting the same language.

See how different a BSON element looks (and feels) using this alternative description:

```
(poke) d.elements[0]
BSON_Elem {
  str=struct {
```

```
    tag=0x2UB,
    name="name",
    value=BSON_String {
      size=0x11,
      value="Jose E. Marchesi",
      chars=[0x4aUB,0x6fUB,0x73UB,0x65UB...]
    }
  }
}
(poke) d.elements[1]
BSON_Elem {
  integer32=struct {
    tag=0x10UB,
    name="age",
    value=0x27
  }
}
(poke) d.elements[2]
BSON_Elem {
  integer64=struct {
    tag=0x12UB,
    name="big",
    value=0x3d31c3f9e3eb9L
  }
}
```

What is the best way? It certainly depends on the kind of data you want to manipulate, and the level of abstraction you want to achieve. Ultimately, it is up to you.

Generally speaking, the best structuring is the one that allows you to manipulate the data in terms of the structured abstractions as naturally as possible. That's the art and craft of writing good pickles.

4.13 Structured Integers

When we structure data using Poke structs, arrays and the like, we often use the same structure than a C programmer would use. For example, to model ELF RELA structures, which are defined in C like:

```
typedef struct
{
  Elf64_Addr   r_offset;  /* Address */
  Elf64_Xword  r_info;    /* Relocation type and symbol index */
  Elf64_Sxword r_addend;  /* Addend */
} Elf64_Rela;
```

we could use something like this in Poke:

```
type Elf64_Rela =
  struct
  {
    Elf64_Addr r_offset;
    Elf64_Xword r_info;
    Elf64_Sxword r_addend;
  };
```

Here the Poke struct type is pretty equivalent to the C incarnation. In both cases the fields are always stored in the given order, regardless of endianness or any other consideration.

However, there are situations where stored integral values are to be interpreted as composite data. This is the case of the `r_info` field above, which is a 64-bit unsigned integer (`Elf64_Xword`) which is itself composed by several fields, depicted here:

```
 63                                                0
+----------------------+----------------------+
|       r_sym          |      r_type          |
+----------------------+----------------------+
MSB                                            LSB
```

In order to support this kind of composition of integers, C programmers usually resort to either bit masking (most often) or to the often obscure and undefined behaviour-prone C bit fields. In the case of ELF, the GNU implementations define a few macros to access these "sub-fields":

```
#define ELF64_R_SYM(i)          ((i) >> 32)
#define ELF64_R_TYPE(i)         ((i) & 0xffffffff)
#define ELF64_R_INFO(sym,type) ((((Elf64_Xword) (sym)) << 32) + (type))
```

Where `ELF64_R_SYM` and `ELF64_R_TYPE` are used to extract the fields from an `r_info`, and `ELF64_R_INFO` is used to compose it. This is typical of C data structures.

We could of course mimic the C implementation in Poke:

```
fun Elf64_R_Sym = (Elf64_Xword i) uint<32>:
   { return i .>> 32; }
fun Elf64_R_Type = (Elf64_Xword i) uint<32>:
   { return i & 0xffff_ffff; }
fun Elf64_R_Info = (uint<32> sym, uint<32> type) Elf64_Xword:
   { return sym as Elf64_Xword <<. 32 + type; }
```

However, this approach has a huge disadvantage: since we are not able to encode the logic of these "sub-fields" in proper Poke fields, they become second class citizens, with all that implies: no constraints on their own, can't be auto-completed, can't be assigned individually, *etc.*

But we can use the so-called *integral structs*! These are structs that are defined exactly like your garden variety Poke structs, with a small addition:

```
type Elf64_RelInfo =
  struct uint<64>
  {
    uint<32> r_sym;
    uint<32> r_type;
  };
```

Note the `uint<64>` addition after `struct`. This can be any integer type (signed or unsigned). The fields of an integral struct should be integral themselves (this includes both integers and offsets) and the total size occupied by the fields should be the same size than the one declared in the struct's integer type. This is checked and enforced by the compiler.

The Elf64 RELA in Poke can then be encoded like:

```
type Elf64_Rela =
  struct
  {
    Elf64_Addr r_offset;
    struct Elf64_Xword
    {
      uint<32> r_sym;
```

```
      uint<32> r_type;
    } r_info;
    Elf64_Sxword r_addend;
  };
```

When an integral struct is mapped from some IO space, the total number of bytes occupied by the struct is read as a single integer value, and then the values of the fields are extracted from it. A similar process is using when writing. That is what makes it different with respect a normal Poke struct.

It is possible to obtain the integral value corresponding to an integral struct using a cast to an integral type:

```
(poke) type Foo = struct int<32> { int<16> hi; uint<16> lo; };
(poke) Foo { hi = 1 } as int<32>
0x10000
```

An useful idiom, that doesn't require to specify an explicit integral type, is this:

```
(poke) type Foo = struct int<32> { int<16> hi; uint<16> lo; };
(poke) var x = Foo @ 0#B;
(poke) +x
0xfe00aa
```

These casts allow to "integrate" struct values explicitly, but the compiler also implicitly promotes integral struct values to integers in all contexts where an integer is expected:

- Operator to an addition, subtraction, multiplication, division, ceil division, bit left shift, bit right shift, bit-wise not, bit-wise and, bit-wise or, bit-wise xor, logical and, logical or, logical not, logical implication, positive, negative, bit-concatenation, condition in conditional expression.
- Argument to a function whose formal expects an integer.
- Value returned from a function that returns an integer.
- Condition in loop statement.
- Constraint expression in a struct field.
- Expression in a conditional struct field.

Note that the above list doesn't include relational operators, since these operators work with struct values.

4.14 Working with Incorrect Data

We have seen how Poke type definitions very often include constraint expressions reflecting the integrity of the data stored in these types.

Consider for example the following abbreviated description of the header of an ELF file:

```
type Elf64_Ehdr =
  struct
  {
    struct
    {
      byte[4] ei_mag = [0x7fUB, 'E', 'L', 'F'];
      byte ei_class;
      [...]
    } e_ident;

    Elf_Half e_type;
    Elf_Half e_machine;
```

```
  Elf_Word e_version = EV_CURRENT;
  [...]
  Elf_Half e_shstrndx : e_shnum == 0 || e_shstrndx < e_shnum;
};
```

There are three constraint expressions in the definition above:

- The constraint expression in the field `ei_mag` makes sure that a right magic number begins the header.
- The constraint expression in `e_version` checks that the ELF version is the current version.
- The constraint expression in `e_shstrndx` checks that the index stored in the field doesn't overflow the section header table of the file.

Every time we construct or map an `Elf64_Ehdr` value the constraint expressions are checked. In case any of the constraints are not satisfied we get a "constraint violation exception". This is useful to realize that we are facing invalid data (when mapping) or that we are trying to build an invalid ELF header (when constructing.)

However, the exception avoids the completion of the mapping or construction operation. Sometimes this is inconvenient:

- When we are given invalid data that we want to explore or fix. Imagine for example a programmer investigating a possible assembler or linker bug that results in an invalid ELF file.
- When we are discovering or inferring the structure of some data: we suspect at some point there may be something that is similar to an ELF header.
- When we want to create invalid data. Suppose for example a programmer who is building a test case for the linker, that requires a corrupted ELF file.

The way poke provides support for these situation is using the concept of *map strictness*. Suppose we map an ELF header in some IO space:

```
(poke) load elf
(poke) var ehdr = Elf64_Ehdr @ 0#B
```

The mapping operator `@` performs a *strict mapping*. This means that constraint expressions are evaluated and the operation is interrupted if any constraint fails, as described above. Also, this means the integrity will be checked when the data is modified:

```
(poke) ehdr.e_version = 666
unhandled constraint violation exception
```

So the mapping operator generates "strict" values. We can check the strictness of a mapped value using the `'strict` attribute:

```
(poke) ehdr'strict
1
```

If we know that we are facing corrupted data, or if we want to corrupt the ELF header, we perform a *non-strict mapping* using a variant of the mapping operator:

```
(poke) var ehdr_ns = Elf64_Ehdr @! 0#B
(poke) ehdr_ns'strict
0
```

We can corrupt the header using this non-strict value:

```
(poke) ehdr_ns.e_version = 666
```

5 Maps and Map-files

Editing data with GNU poke mainly involves creating mapped values and storing them in Poke variables. However, this may not be that convenient when poking several files simultaneously, and when the complexity of the data increases. poke provides a convenient mechanism for this: maps and map files.

5.1 Editing using Variables

Editing data with GNU poke mainly involves creating mapped values and storing them in Poke variables. For example, if we were interested in altering the fields of the header in an ELF file, we would map an `Elf64_Ehdr` struct at the beginning of the underlying IO space (the file), like in:

```
(poke) .file foo.o
(poke) load elf
(poke) var ehdr = Elf64_Ehdr @ 0#B
```

At this point the variable `ehdr` holds an `Elf64_Ehdr` structure, which is mapped. As such, altering any of the fields of the struct will update the corresponding bytes in `foo.o`. For example:

```
(poke) ehdr.e_entry = 0#B
```

A Poke value has three mapping related attributes: whether it is mapped, the offset at which it is mapped in an IO space, and in which IO space. This information is accessible for both the user and Poke programs using the following attributes:

```
(poke) ehdr'mapped
1
(poke) ehdr'offset
0UL#b
(poke) ehdr'ios
0
```

Thats it, `ehdr` is mapped at offset zero byte in the IO space `#0`, which corresponds to `foo.o`:

```
(poke) .info ios
  Id Type Mode Bias Size Name
* #0 FILE rw 0x00000000#B 0x000004c8#B   ./foo.o
```

Now that we have the ELF header, we may use it to get access to the ELF section header table in the file, that we will reference using another variable `shdr`:

```
(poke) var shdr = Elf64_Shdr[ehdr.e_shnum] @ ehdr.e_shoff
(poke) shdr[1]
Elf64_Shdr {
  sh_name=0x1bU#B,
  sh_type=0x1U,
  sh_flags=#<ALLOC,EXECINSTR>,
  sh_addr=0x0UL#B,
  sh_offset=0x40UL#B,
  sh_size=0xbUL#B,
  sh_link=0x0U,
  sh_info=0x0U,
  sh_addralign=0x1UL,
  sh_entsize=0x0UL#b
}
```

Variables are convenient entities to manipulate in Poke. Let's suppose that the file has a lot of sections and we want to do some transformation in every section. It is a time consuming

operation, and we may forget which sections we have already processed and which not. We could create an empty array to hold the sections already processed:

```
(poke) var processed = Elf64_Shdr[] ()
```

And then, once we have processed some given section, add it to the array:

```
... edit shdr[23] ...
(poke) processed += [shdr[23]]
```

Note how the array =processed= is not mapped, but the sections contained in it are mapped: Poke uses copy by shared value. So, after we spend the day carefully poking our ELF file, we can ask poke, are we done with all the sections in the file?

```
(poke) shdr'length == processed'length
1
```

Yes, we are. This can be made as sophisticated as desired. We could easily write a function that saves the contents of `processed` in files, so we can continue hacking tomorrow, for example.

We can then concluding that using mapped variables to edit data structures stored in IO spaces works well in common and simple cases like the above: we make our ways mapping here and there, defining variables to hold data that interests us, and it is easy to remember that the variables `ehdr` and `shdr` are mapped, where are they mapped, and that they are mapped in the file `foo.o`.

However, GNU poke allows to edit more than one IO space simultaneously. Let's say we now want to poke the sections of another ELF file: `bar.o`. We would start by opening the file:

```
(poke) .file bar.o
(poke) .info ios
  Id Type Mode Bias Size Name
* #1 FILE rw 0x00000000#B 0x000004c8#B ./bar.o
  #0 FILE rw 0x00000000#B 0x000004c8#B ./foo.o
```

Now that `bar.o` is the current IO space, we can map its header. But now, what variable to use? We would rather not redefine `ehdr`, because that is already holding the header of `foo.o`. We could adapt our naming schema on the fly:

```
(poke) var foo_ehdr = ehdr
(poke) var bar_ehdr = Elf64_Ehdr @ 0#B
```

But then we would need to do the same for the other variables too:

```
(poke) var foo_shdr = shdr
(poke) var bar_shdr = Elf64_Shdr[bar_ehdr.e_shnum] @ bar_ehdr.e_shoff
```

However, we can easily see how this can degenerate quickly: what about `processed`, for example? In general, as the number of IO spaces being edited increases it becomes more and more difficult to manage our mapped variables, which are associated to each IO space.

5.2 poke Maps

As we have seen mapping variables is a very powerful, general and flexible mean to edit stored binary data in one or more IO spaces. However it is easy to lose track of where the variables are mapped and, ideally speaking, we would want to have a mean to refer to, say, the "ELF header", and get the header as a mapped value regardless of what specific file we are editing. Sort of a "meta variable". GNU poke provides a way to do this: *maps*.

A *map* can be conceived as a sort of "view" that can be applied to a given IO space. Maps have entries, which are values mapped at some given offset, under certain conditions. For example, we have seen an ELF file contains, among other things, a header at the beginning of the file and a table of section headers of certain size and located at certain location determined by the header. These would be two entries of a so-called ELF map.

poke maps are defined in *map files*. These files use the `.map` extension. A map file `self.map` (for sectioned/simple elf) defining the view of an ELF file as a header and a table of section header would look like this:

```
/* self.map - map file for a simplified view of an ELF file.  */

load elf;

%%

%entry
%name ehdr
%type Elf64_Ehdr
%offset 0#B

%entry
%name shdr
%type Elf64_Shdr[(Elf64_Ehdr @ 0#B).e_shnum]
%condition (Elf64_Ehdr @ 0#B).e_shnum > 0
%offset (Elf64_Ehdr @ 0#B).e_shoff
```

This map file defines a view of an ELF file as a header entry `ehdr` and an entry with a table of section headers `shdr`.

The first section of the file, which spans until the separator line containing `%%`, is arbitrary Poke code which as we shall see, gets evaluated before the map entries are processed. This is called the map "prologue". In this case, the prologue contains a comment explaining the purpose of the file, and a single statement `load` that loads the `elf.pk` pickle, since the entries below use definitions like `Elf64_Ehdr` that are defined by that pickle. The prologue is useful to define Poke functions and other entities that are then used in the definitions of the entries.

A separator line containing only `%%` separates the prologue from the next section, which is a list of entries definitions. Each entry definition starts with a line `%entry`, and has the following attributes:

`%name` Like `ehdr` and `shdr`. These names should follow the same rules than Poke variables, but as we shall see later, map entries are not Poke variables. This attribute is mandatory.

`%type` This can be any Poke expression denoting a type, like `int`, `Elf64_Ehdr` or `Elf64_Shdr[(Elf64_Ehdr @ 0#B).e_shnum]`. This attribute is mandatory.

`%condition`
If specified, will determine whether to include the entry in the map. In the example above, the map will have an entry `shdr` only if the ELF file has one or more sections. Any Poke expression evaluating to a boolean can be used as conditions. This attribute is optional: entries not having a condition will always be included in the map.

`%offset`
Offset in the IO space where the entry will be mapped. Any Poke expression evaluating to an offset can be used as entry offset. This attribute is mandatory.

5.3 Loading Maps

So we have written our =self.map=, which denotes a view or structure of ELF files we are interested on, and that resides in the current working directory. How to use it?

The first step is to fire up poke and open some object file. Let's start with `foo.o`:

```
(poke) .file foo.o
```

Now, we can load the map using the `.map load` dot-command:

```
(poke) .map load self
[self](poke)
```

The `.map load self` command makes poke to look in certain directories for a file called `self.map`, and to load it. The list of directories where poke looks for map files is encoded in the variable `map_load_path` as a string containing a maybe empty list of directories separated by `:` characters. Each directory is tried in turn. This variable is initialized with suitable defaults:

```
(poke) map_load_path
"/home/jemarch/.poke.d:.:/home/jemarch/.local/share/poke:[...]"
```

Once a map is loaded, observe how the prompt changed to contain a prefix `[self]`. This means that the map `self` is loaded for the current IO space. You can choose to not see this information in the prompt by setting the `prompt-maps` option either at the prompt or in your `.pokerc`:

```
poke) .set prompt-maps no
```

By default `prompt-maps` is `yes`. This prompt aid is intended to provide a cursory look of the "views" or maps loaded for the current IO space. If we load another IO space and switch to it, the prompt changes accordingly:

```
(poke) [self](poke) .mem foo
The current IOS is now ‘*foo*’.
(poke) .ios #0
The current IOS is now ‘./foo.o’.
[self](poke)
```

At any time the `.info maps` dot-command can be used to obtain a full list of loaded maps, with more information about them:

```
(poke) .info maps
IOS   Name   Source
#0    self   ./self.map
```

In this case, there is a map `self` loaded in the IO space `#0`, which corresponds to `foo.o`.

Once we make `foo.o` our current IO space, we can ask poke to show us the entries corresponding to this map using another dot-command:

```
: (poke) .map show self
: Offset     Entry
: 0x0UL#B    $self::ehdr
: 0x208UL#B  $self::shdr
```

This tells us there are two entries for `self` in `foo.o`: `$self::ehdr` and `$self::shdr=` Note how map entries use names that start with the `$` character, then contain the name of the map an the name of the entry we defined in the map file, separated by `::`.

We can now use these entries at the prompt like if they were regular mapped variables:

```
[self](poke) $self::ehdr
Elf64_Ehdr {
  e_ident=struct {
    ei_mag=[0x7fUB,0x45UB,0x4cUB,0x46UB],
    [...]
  },
 e_type=0x1UH,
 e_machine=0x3eUH,
```

```
 [...]
}
(poke) $self::shdr'length
11UL
```

It is important to note, however, that map entries like `$foo::bar` are *not* part of the Poke language, and are only available when using poke interactively. Poke programs and scripts can't use them.

Let's now open another ELF file, and the =self= map in it:

```
(poke) .file /usr/local/lib/libpoke.so.0.0.0
(poke) .map load self
[self](poke)
```

So now we have two ELF files loaded in poke: `foo.o` and `libpoke.so.0.0.0`, and in both IO spaces we have the `self` map loaded. We can easily see that the map entries are different depending on the current IO space:

```
[self](poke) .map show self
Offset       Entry
0UL#B        $self::ehdr
3158952UL#B  $self::shdr
[self](poke) .ios #0
The current IOS is now `./foo.o'.
[self](poke) .map show self
Offset   Entry
0UL#B    $self::ehdr
520UL#B  $self::shdr
```

`foo.o` is an object file, whereas `libpoke.so.0.0.0` is a DSO:

```
(poke) .ios #0
The current IOS is now `./foo.o'.
[self](poke) $self::ehdr.e_type
1UH
[self](poke) .ios #2
The current IOS is now `/usr/local/lib/libpoke.so.0.0.0'.
[self](poke) $self::ehdr.e_type
3UH
```

The interpretation of the map entry `$self::ehdr` is different depending on the current IO space. This makes it possible to refer to the "ELF header" of the current file.

Underneath, poke implements this by defining mapped variables and "redirecting" the entry names `$foo::bar` to the right variable depending on the IO space that is currently selected. It hides all that complexity from us.

5.4 Multiple Maps

It is perfectly possible (and useful!) to load more than one map in the same IO space. It is very natural for a single file, for example, to contain data that can be interpreted in several ways, or of different nature.

Let's for example open again an ELF file, this time compiled with `-g`:

```
(poke) .file foo.o
```

We now load our `self` map, to get a view of the file as a collection of sections:

```
(poke) .map load self
[self](poke)
```

And now we load the `dwarf` map that comes with poke, to get a view of the file as having debugging information encoded in DWARF:

```
[self(poke) .map load dwarf
[dwarf,self](poke)
```

See how the prompt now reflects the fact that the current IO space contains DWARF info! Let's take a look:

```
[dwarf,self](poke) .info maps
IOS   Name    Source
#0    dwarf   /home/jemarch/gnu/hacks/poke/maps/dwarf.map
#0    self    ./self.map
[dwarf,self](poke) .map show dwarf
Offset    Entry
0x5bUL#B  $dwarf::info
```

Now we can access entries from any of the loaded maps, *i.e.* access the file in terms of different perspectives. As an ELF file:

```
[dwarf,self](poke) $self::shdr[1]
Elf64_Shdr {
  sh_name=0xb5U#B,
  sh_type=0x11U,
  sh_flags=#<>,
  sh_addr=0x0UL#B,
  sh_offset=0x40UL#B,
  sh_size=0x8UL#B,
  sh_link=0x18U,
  sh_info=0xfU,
  sh_addralign=0x4UL,
  sh_entsize=0x4UL#b
}
```

And as a file containing DWARF info:

```
[dwarf,self](poke) $dwarf::info
Dwarf_CU_Header {
  unit_length=#<0x0000004eU#B>,
  version=0x4UH,
  debug_abbrev_offset=#<0x00000000U#B>,
  address_size=0x8UB#B
}
```

If you are curious about how the DWARF entries are defined, look at `maps/dwarf.map` in the poke source distribution, or in your installed poke (`.info maps` will tell you the file the map got loaded from.)

It is possible to unload or remove a map from a given IO space using the `.map remove` dot-command. Say we are done looking at the DWARF in `foo.o`, and we are no longer interested in it as a file containing debugging info. We can do:

```
[dwarf,self](poke) .map remove dwarf
[self](poke)
```

Note how the prompt was updated accordingly: only `self` remains as a loaded map on this file.

5.5 Auto-map

Certain maps make sense when editing certain types of data. For example, `dwarf.map` is intended to be used in ELF files. In order to ease using maps, poke provides a feature called *auto mapping*, which is disabled by default.

You can set auto mapping like this:

```
(poke) .set auto-map yes
```

When auto mapping is enabled, poke will look to the value of the pre-defined variable `auto_map`, which must contain an array of pairs of strings, associating a regular expression with a map name.

For example, you may want to initialize `auto_map` like this in your `.pokerc` file:

```
auto_map = [[".*\\.mp3$", "mp3"],
            [".*\\.o$", "elf"],
            ["a\\.out$", "elf"]];
```

This will make poke to load =mp3.map= for every file whose name ends with `.mp3`, and `elf.map` for files having names like `foo.o` and `a.out`.

Following the usual pokeish philosophy of being as less as intrusive by default as possible, the default value of `auto_map` is the empty string.

5.6 Constructing Maps

As we have seen, we can define our own maps using map files like `self.map`, which contain a prologue and a set of map entries. However, sometimes it is useful to create maps "on the fly" while we explore some data with poke.

To make this possible, poke provides a suitable set of dot-commands. Let's say we are poking some data, and we want to create a map for it. We can do that like this:

```
(poke) .map create mymap
```

This creates an empty map named `mymap`, with no entries:

```
[mymap](poke) .map show mymap
Offset   Entry
```

Adding entries is easy. First, we have to map some variable, and then use it as the base for the new entry:

```
[mymap](poke) var foo = int[3]  0#B
[mymap](poke) .map entry add mymap, foo
[mymap](poke) .map show mymap
Offset   Entry
0x0UL#B  $mymap::foo
```

Note how the entry `$mymap::foo` gets created, associated to the current IO space and mapped at the same offset than the variable `foo`.

We can remove entries from existing maps using the `.map entry remove` dot-command:

```
[mymap](poke) .map entry remove mymap, foo
[mymap](poke) .map show mymap
Offset   Entry
[mymap](poke)
```

We plan to add an additional command to save maps to map files. The idea is that you can create your maps on the fly, save them, and then load them back some other day when you are ready to continue poking. This is not implemented yet though.

5.7 Predefined Maps

GNU poke comes with a set of useful pre-written maps, which get installed in a system location. We want to expand this collection, so please send us your map files!

6 Writing Pickles

GNU poke encourages the user to write little pieces of code in order to face spontaneous needs and fix situations. Is that name in the file encoded in a fixed array of characters padded with white spaces? No problem, just write a three lines function so you can update the file using a comfortable NULL-terminated string. Better than waiting for some poke maintainer to add that function for you, isn't it?

Just save your functions in some personal `.pk` file that you load at startup, and your magic tricks bag will increase in time, making your poking more and more efficient.

However, when it comes to share the code with other people, it is important to follow certain conventions in order to achieve certain uniformity. This makes it easier for other people to discover what your hack provides, and how it works. It is this consistency, and these conventions, that makes some random `.pk` file a *pickle*.

This guide contains guidelines and recommendations for the pickle's writer.

6.1 Pretty-printers

6.1.1 Convention for pretty-printed Output

Very often the structure of the data encoded in binary is not very intelligible. This is because it is usual for binary formats to be designed with goals in mind other than being readable by humans: compactness, detail *etc*.

In our pickle we of course want to provide access to the very finer detail of the data structures. However, we also want for the user to be able to peruse the data visually, and only look at the fine detail on demand.

Consider for example an ID3V1 tag data from some MP3 file. This is the result of mapping a `ID3V1_Tag`:

```
ID3V1_Tag {
  id=[0x54UB,0x41UB,0x47UB],
  title=[0x30UB,0x31UB,0x20UB,0x2dUB,0x20UB,...],
  artist=[0x4aUB,0x6fUB,0x61UB,0x71UB,0x75UB,...],
  album=[0x4dUB,0x65UB,0x6eUB,0x74UB,0x69UB,...],
  year=[0x20UB,0x20UB,0x20UB,0x20UB],
  data=struct {
    extended=struct {
      comment=[0x20UB,0x20UB,0x20UB,0x20UB,0x20UB,...],
      zero=0x0UB,
      track=0x1UB
    }
  },
  genre=0xffUB
}
```

Not very revealing. Fortunately, poke supports pretty printers. If a struct type has a method called `_print`, it will be used by poke as a pretty printer if the `pretty-print` option is set:

```
(poke) .set pretty-print yes
```

By convention, the output of pretty-printers should always start with `#<` and end with `>`. The convention makes it explicit for the user that everything she sees between `#<` and `>` is pretty-printed, and do *not* necessarily reflect the physical structure of the data. Also some information may be missing. In order to get an exact and complete description of the data, the user should `.set pretty-print no` and evaluate the value again at the prompt.

For example, in the following BPF instructions it is obvious at first sight that the shown register values are pretty-printed:

```
BPF_Insn = {
  ...
  regs=BPF_Regs {
     src=#<%r3>,
     dst=#<%r0>
  }
  ...
}
```

If the pretty-printed representation spans for more than one line, please place the opening `#<` in its own line, then the lines with the data, and finally `>` in its own line, starting at column 0.

Example of the MP3 tag above, this time pretty-printed:

```
#<
  genre: 255
  title: 01 - Eclipse De Mar
  artist: Joaquin Sabina
  album: Mentiras Piadosas
  year:
  comment:
  track: 1
>
```

6.1.2 Pretty Printing Optional Fields

Let's say we are writing a pretty-printer method for a struct type that has an optional field. Like for example:

```
type Packet =
  struct
  {
    byte magic : magic in [MAGIC1,MAGIC2];
    byte n;
    byte[n] payload;
    PacketTrailer trailer if magic == MAGIC2;
  }
```

In this case, the struct value will have a `trailer` conditionally, which has to be tackled on the pretty-printer somehow.

An approach that often works good is to replicate the logic in the optional field condition expression, like this:

```
  struct
  {
    byte magic : magic in [MAGIC1,MAGIC2];
    [...]
    PacketTrailer trailer if packet_magic2_p (magic);

    method _print = void:
    {
      [...]
      if (magic == MAGIC2)
        pretty_print_trailer;
```

```
    }
  }
```

This works well in this simple example. In case the expression is big and complicated, we can avoid rewriting the same expression by encapsulating the logic in a function:

```
fun packet_magic2_p = int: { return magic == MAGIC2; }
type Packet =
  struct
  {
    byte magic : magic in [MAGIC1,MAGIC2];
    [...]
    PacketTrailer trailer if packet_magic2_p (magic);

    method _print = void:
    {
      [...]
      if (packet_magic2_p (magic))
        pretty_print_trailer;
    }
  }
```

However, this may feel weird, as the internal logic of the type somehow leaks its natural boundary into the external function `packet_magic2_p`.

An alternative is to use the following idiom, that checks whether the field actually exists in the struct:

```
type Packet =
  struct
  {
    byte magic : magic in [MAGIC1,MAGIC2];
    [...]
    PacketTrailer trailer if packet_magic2_p (magic);

    method _print = void:
    {
      [...]
      try pretty_print_trailer;
      catch if E_elem {}
    }
  }
```

This approach also works with unions:

```
type ID3V1_Tag =
  struct
  {
    [...]
    union
    {
      /* ID3v1.1  */
      struct
      {
        char[28] comment;
        byte zero = 0;
        byte track : track != 0;
```

```
      } extended;
      /* ID3v1  */
      char[30] comment;
    } data;
    [...]

    method _print = void:
    {
      [...]
      try print "  comment: " + catos (data.comment) + "\n";
      catch if E_elem
      {
        print "  comment: " + catos (data.extended.comment) + "\n";
        printf "  track: %u8d", data.extended.track;
      }
    }
  }
```

6.2 Setters and Getters

Given a struct value, the obvious way to access the value of a field is to just refer to it using dot-notation.

For example, for the following struct type:

```
type ID3V1_Tag =
  struct
  {
    [...]
    char[30] title;
    char[30] artist;
    char[30] album;
    char[4] year;
    [...]
  }
```

Suppose the find out the year in the tag is wrong, off by two years: the song was release in 1980, not in 1978!. Unfortunately, due to the bizarre way the year is stored in the file (as a sequence of digits encoded in ASCII, non-NULL terminated) we cannot just write:

```
(poke) tag.year = tag.year + 2
error
```

Instead, we can use facilities from the standard library and a bit of programming:

```
(poke) stoca (format ("%d", atoi (catos (tag.year)) + 2), tag.year)
```

The above line basically transforms the data we want to operate on (the tag year) from the stored representation into a more useful representation (from an array of character digits to an integer) then operates with it (adds two) then converts back to the stored representation. Let's call this "more useful" representation the "preferred representation".

A well written pickle should provide *getter* and *setter* methods for fields in struct types for which the stored representation is not the preferred representation. By convention, getter and setter methods have the following form:

```
method get_field preferred_type: { ... }
method set_field (preferred_type val) void: { ... }
```

Using the `get_` and `set_` prefixes consistently is very important, because the pokist using your pickle can easily find out the available methods for some given value using tab-completion in the REPL.

For example, let's add setter and getter methods for the field `year` in the ID3V1 tag struct above:

```
type ID3V1_Tag =
  struct
  {
    [...]
    char[4] year;

    method get_year = int: { return atoi (catos (year)); }
    method set_year = (int val) void:
    {
      var str = format "%d", val;
      stoca (str, year);
    }
    [...]
  }
```

What constitutes the preferred representation of a field is up to the criteria of the pickle writer. For the tag above, I would say the preferred representations for the title, artist, album and year are string for title, artist and album, and an integer for the year.

7 Writing Binary Utilities

GNU poke is, first and foremost, intended to be used as an interactive editor, either directly on the command line or using a graphical user interface built on it. However, since its conception poke was intended to also provide a suitable and useful foundation on which other programs, the so-called *binary utilities*, could be written. This chapter shows how to write Poke scripts and programs.

7.1 Poke Scripts

In interactive usage, there are two main ways to execute Poke code: at the interactive prompt (or REPL), and loading "pickles".

Executing Poke code at the REPL is as easy as introducing a statement or expression:

```
(poke) print "Hello\n"
Hello
```

Executing Poke code in a pickle is performed by loading the file containing the code:

```
(poke) .load say-hello.pk
Hello
```

Where `say-hello.pk` contains simply:

```
print "Hello\n";
```

However, we would like to have Poke scripts, *i.e.* to be able to execute Poke programs as standalone programs, from the shell. In other words, we want to use GNU poke as an interpreter. This is achieved by using a shebang, which should appear at the top of the script file. The poke shebang looks like this:

```
#!/usr/bin/poke -L
!#
```

The `-L` command line option tells poke that it is being used as an interpreter. Additional arguments for poke can be specified before `-L` (but not after). The `#! ... !#` is an alternative syntax for multi-line comments, which allows to have the shebang at the top of a Poke program without causing a syntax error. This nice trick has been borrowed from guile.

Therefore, we could write say-hello as a Poke script like this:

```
#!/usr/bin/poke -L
!#

print "Hello\n";
```

And then execute it like any other program or script:

```
$ ./say-hello
```

7.2 Command-Line Arguments

When a Poke script is executed, the command line arguments passed to the script become available in the array argv. Example:

```
#!/usr/bin/poke -L
!#

for (arg in argv)
  print "Argument: " + arg + "\n";
```

Executing this script results in:

```
$ ./printargs foo bar 'baz quux'
```

```
Argument: foo
Argument: bar
Argument: baz quux
```

Note how it is not needed to have an argc variable, since the number of elements stored in a Poke array can be queried using an attribute: `argv'length`.

Note also that `argv` is only defined when poke runs as an interpreter:

```
$ poke
[...]
(poke) argv
<stdin>:1:1: error: undefined variable 'argv'
argv;
^~~~
```

Accessing the `argv` is more than enough for many simple programs. However, we may need a more sophisticated handling of command-line options: support for both short and long style options, adherence to the GNU coding standards, and the like. For these cases poke provides a pickle `argp`. See Section 13.1 [argp], page 100.

7.3 Exiting from Scripts

By default a Poke script will communicate a successful status to the environment, upon exiting:

```
$ cat hello
#!/usr/bin/poke -L
!#

print "hello\n";
$ ./hello && echo $?
0
```

In order to exit with some other status code, most typically to signal an erroneous situation, the Pokeish way is to raise an `E_exit` exception with the desired exit status code:

```
raise Exception { code = EC_exit, exit_status = 1 };
```

This can be a bit cumbersome to write, so poke provides a more conventional syntax in the form of an `exit` function:

```
fun exit = (int<32> exit_code = 0) void:
{
  raise Exception { code = EC_exit, exit_status = exit_code };
}
```

Using `exit`, the above raise statement becomes the much simpler:

```
exit (1);
```

7.4 Loading pickles as Modules

Suppose we want to write a Poke program to extract the contents of sections in a given ELF object file. Extracting sections requires dealing with several data structures encoded in the ELF file, such as the header, the section header table, the string table (that contains the names of the sections) and so on. It would be of course possible to define Poke types for these structures in the script itself but, as it happens, GNU poke ships with an already written pickle that describes the ELF structures. It is called `elf.pk`.

So a script needing to mess with ELF data structures can just make use of `elf.pk` using the load construction:

```
load elf;
```

This looks for a file called `elf.pk` in a set of directories, which are predefined by poke, and loads it. The list of directories where poke looks for pickles is stored in the load_path variable as a colon separated list of directory names, and can be customized:

```
$ poke
[...]
(poke) load_path
"/home/jemarch/.poke.d:.:/home/jemarch/.local/share/poke:..."
```

The default value of `load_path` contains both user-specific directories and system-wide directories. This assures that all the pickles installed by poke are available, and that the user can load her own pickles in her scripts.

Once a pickle is loaded in a script the types, functions and variables defined in it (either directly or indirectly by loading its own pickles) become available.

7.5 elfextractor

As an example, in this section we will be hacking a very simple utility called elfextractor, that extracts the contents of the sections of an ELF file, whose name is provided as an argument in the command line, into several output files. This is the synopsis of the program:

```
elfextractor file [section_name]
```

Where *file* is the name of the ELF file from which to extract sections, and an optional *section_name* specifies the name of the section to extract.

Say we have a file `foo.o` and we would like to extract its text section. We would use elfextractor like:

```
$ elfextractor foo.o .text
```

Provided `foo.o` indeed has a section named `.text`, the utility will create a file `foo.o.text` with the section's contents. Note how the names of the output files are derived concatenating the name of the input ELF file and the name of the extracted section.

If no section name is specified, then all sections are extracted. For example:

```
$ elfextractor foo.o
$ ls foo.o*
foo.o          foo.o.eh_frame       foo.o.shstrtab  foo.o.symtab
foo.o.comment  foo.o.rela.eh_frame  foo.o.strtab    foo.o.text
```

This is a possible implementation of elfextractor:

```
#!/usr/bin/poke -L
!#

/* elfextractor - Extract sections from ELF64 files. */

load elf;

if (!(argv'length in [1,2]))
  {
    print "Usage: elfextractor FILE [SECTION_NAME]\n";
    exit (1);
  }

var file_name = argv[0];
var section_name = (argv'length > 1) ? argv[1] : "";
```

```
try
  {
    var fd = open (file_name, IOS_M_RDONLY);
    var elf = Elf64_File @ fd : 0#B;

    for (shdr in elf.shdr where shdr.sh_type != 0x0)
      {
        var sname = elf.get_string (shdr.sh_name);

        if (section_name == "" || sname == section_name)
          save :ios elf'ios :file file_name + sname
               :from shdr.sh_offset :size shdr.sh_size;
      }

    close (fd);
  }
catch (Exception e)
  {
    if (e == E_constraint)
      printf ("error: `%s' is not a valid ELF64 file\n", file_name);
    else if (e == E_io)
      printf ("error: couldn't open file `%s'\n", file_name);
    else
      raise e;

    exit (1);
  }
```

First the command line arguments are handled. The script checks whether the right number of arguments have been passed (either 1 or 2) exiting with an error code otherwise. The file name and the section name are then extracted from the `argv` array.

Once we have the file name and the optional desired section name, it is time to do the real work. The code is enclosed in a try-catch block statement, because some of the operations may result on exceptions being raised.

First, the ELF file whose name is specified in the command line is opened for reading:

```
var fd = open (file_name, IOS_M_RDONLY);
```

The built-in function `open` returns a file descriptor that can be subsequently used in mapping operations. If the provided file name doesn't identify a file, or if the file can't be read for whatever reason, an `E_io` exception is raised. Note how the exception is handled in the `catch` block, emitting an appropriate diagnostic message and exiting with an error status.

Once the ELF file is open for reading, we map an `Elf64_File` on it, at the expected offset (zero bytes from the beginning of the file):

```
var elf = Elf64_File @ fd : 0#B;
```

If the file doesn't contain valid ELF data, this map will fail and raise an `E_constraint` exception. Again, the `catch` block handles this situation.

At this point the variable `elf` contains an `Elf64_File`. Since we want to extract the sections contained in the file, we need to somehow iterate on them. The section header table is available in `elf.shdr`. A for-in-where loop is used to iterate on all the headers, skipping the "null" ELF sections which are always empty, and are characterized by a `shdr.sh_type` of 0. An inner conditional filters out sections whose name do not match the provided name in the command line, if it was specified at all.

For each matching section we then save its contents in a file named after the input ELF file, by calling a function `save`, which is provided by poke:

```
save :ios elf'ios :file file_name + sname
     :from shdr.sh_offset :size shdr.sh_size;
```

The above is exactly what we would have written at the poke REPL! (modulus trailing semicolon). How is this supposed to work? Thing is, GNU poke commands are implemented as Poke functions. Let's consider `save`, for example. It is defined as a function having the following prototype:

```
fun save = (int ios = get_ios,
               string file = "",
               off64 from = 0#B,
               off64 size = 0#B,
               int append = 0,
               int verbose = 0) void:
{ ... }
```

Once a Poke function is defined in the environment, it becomes available as such. Therefore, in a poke session we could call it like:

```
(poke) save (get_ios, "filename", 0#B, 12#B, 0, 1)
```

However, this is cumbersome and error prone. To begin with, we should remember the name, position and nature of each argument accepted by the command. What is even more annoying, we are forced to provide explicit values for them, like in the example above we have to pass the current IOS (the default), and 0 for `append` (the default) just to being able to set `verbose`. Too bad.

To ease commanding poke, the Poke language supports an alternative syntax to call functions, in which the function arguments are referred by name, can be given in any order, and can be omitted. The command above can be thus written like:

```
(poke) save :from 0#B :size 12#B :verbose 1
```

This syntax is mostly intended to be used interactively, but nothing prevents to use it in Poke programs and scripts whenever it is deemed appropriate, like we did in elfextractor. We could of course have used the more conventional syntax:

```
if (section_name == "" || sname == section_name)
  save (elf'ios, file_name + sname,
        shdr.sh_offset, shdr.sh_size, 0, 0);
```

What style to use is certainly a matter of taste.

Anyhow, once the sections have been written out, the file descriptor is closed and the program exits with the default status, which is success. Should the `save` function find any problem saving the data, such as a full disk, not enough permissions or the like, exceptions will be raised, caught and maybe handled by our `catch` block.

And this is it! The complete program is 44 lines long. This is a good example that shows how, given a pickle providing a reasonable description of some binary-oriented format (ELF in this case) poke can be leveraged to achieve a lot in a very concise way, free from the many details involved in the encoding, reading and writing of binary data.

7.6 Filters

The classic notion of *filter* in Unix-like systems and other Operating Systems is of a program that reads certain input using its standard input, realized some transformations on it, and writes the result on its standard output. The filter doesn't need to operate on all the data at once, nor to buffer it: it reads input in chunks once it becomes available and is needed, and write output as necessary.

GNU poke provides the so called *stream IO spaces* in order to implement binary utilities that act like filters.

7.6.1 Stream IO Spaces

GNU poke provides support for opening and operating on several kinds of IO spaces: files, memory buffers, process memory, *etc.* An IO space can be opened for reading, for writing or both, and they all can be accessed randomly, *i.e.* to read or write some data from/to a particular offset in the IO space.

Writing a filter would involve to peek data from the standard input and then poke the data (maybe transformed) to the standard output or the standard error output. However, we know that these devices are not random oriented devices that can be accessed at any offset, but rather stream-like devices whose special characteristics are:

– They are either read-only or write-only. You cannot write to an input stream and you cannot read from an output stream.
– Once data is read from an input stream, it can't be read again. It is not buffered.
– Once data is written to an output stream, it can't be altered again.

Given the above characteristics of streams like the standard input and output, it is clear we cannot use the File IO streams on them. It is also not clear how could we perform a map operation on a stream device: in poke these operations always require an explicit offset.

GNU poke solves this by providing three special IO spaces that the user can open using the following handlers:

`<stdin>` Read-only stream IOS that is connected to the standard input of the poke process.

`<stdout>` Write-only stream IOS that is connected to the standard output of the poke process.

`<stderr>` Write-only stream IOS that is connected to the standard error of the poke process.

Opening these IO spaces is done exactly like with any other IO space. Example:

```
var stdin = open ("<stdin>");
```

7.6.2 Reading from Streams

Unlike normal Unix file descriptors, read-only poke streams are buffered. Let's supposed we open the standard input:

```
var stdin = open ("<stdin>");
```

Then we map some data (one byte) at some particular offset:

```
var b = byte @ stdin : 28#B;
```

The mapping operation above causes poke to read 28 bytes from the standard input and to set the variable `b` to the 28-th byte. The bytes read from the stream are still available (buffered) and the offsets are still relative from the "beginning" of the standard input:

```
var c = byte @stdin : 0#B;
```

The byte in `c` is the byte that appeared in the standard input 27 bytes before the byte in `b`.

This buffering is nice, and it is what allows us to access the standard input like if it were a random oriented device. However, it is obvious we would be in trouble if we filter big amounts of data, like in a network interface: we would likely use all available memory.

To allow filtering big amount of data, poke allows to *flush* the read-only streams. Flushing means that the buffered data in the read-only stream is "forgotten", and trying to access it will result in an exception:

```
var stdin = open ("<stdin>");
```

```
var b = byte @ stdin : 28#B;
var c = byte @ stdin : 29#B;
flush (stdin, 29#B);
var d = byte @ stdin : 30#B;
var e = byte @ stdin : 20#B; /* Exception.  */
```

7.6.3 Writing to Streams

Similarly, unlike normal Unix file descriptors, write-only poke streams are also buffered and can be accesses randomly:

```
var stdout = open ("<stdout>");
```

We can write to the output stream by mapping as usual:

```
byte @ stdout : 28#B = 0xff;
```

The mapping operation above writes 27 null bytes, *i.e.* it fills until reaching the requested offset in the output stream. Then it writes the byte `0xff` at offset 28 bytes. If we then write to one of the buffered bytes:

```
byte @ stdout : 27#B = 0xab;
```

This results in updating the byte that will be written right before the `0xff` once the output stream is flushed.

Again, we cannot buffer ad-infinitum: we would exhaust all available output. Therefore, XXX

7.6.4 pk-strings

Below you can find a very simple Poke program that works like the standard Unix utility `strings`.

```
#!/usr/local/bin/poke -L
!#

/* Printable ASCII characters: 0x20..0x7e */

var stdin = open ("<stdin>");
var stdout = open ("<stdout>");

var offset = 0#B;

try
{
  flush (stdin, offset);

  var b = byte @ stdin : offset;
  if (b >= 0x20 && b <= 0x7e)
    byte  stdout : iosize (stdout) = b;

  offset = offset + 1#B;
}
until E_eof;

close (stdin);
close (stdout);
```

8 Configuration

8.1 .pokerc

Upon invocation poke will read and execute the commands of an initialization file, if it exists: the `pokerc` file. There are two ways to keep an initialization file.

The simple, traditional way is to have a file named `.pokerc` in your home directory. GNU poke will look for a file like that first.

If `.pokerc` is not found your home directory, poke looks in the locations specified by the *XDG Base Directory Specification*[1] for a file named `poke/pokerc.conf`. For example, you could use `~/.config/poke/pokerc.conf`.

Which way is better depends on your specific requirements and taste.

GNU poke can be insturcted to not read the initialization file by passing `-q` or `--no-init-file` in the command line.

Example of initialization file:

```
# My poke configuration.
.set endian host
.set obase 16
.set pretty-print yes
pk_dump_cluster_by = 4
.load ~/.poke.d/mypickles.pk
```

8.2 Load Path

The `load_path` Poke variable contains a list of directories separated by the colon character (`:`). When a module is loaded using the `load` construction, these directories are searched in sequential order for the file corresponding to the requested module.

Empty directory names and entries that do not name existing directories are ignored.

Some entries have special meanings:

`%DATADIR%`
: This is interpreted as the system-wide datadir directory, that depends on the prefix where poke is installed.

For example, say you want to maintain `.pk` files in your `~/.poke.d` directory. You will probably want to add that directory to the `load_path`, when poke initializes. A way to do that is to add a command like this to your `pokerc` file:

```
load_path = getenv ("HOME") + "/.poke.d:" + load_path
```

If the environment variable `POKE_LOAD_PATH` is defined in the environment, its value is added to `load_path` at poke startup time.

8.3 Styling

XXX

[1] https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html

9 Time

Systems encode time-stamps or dates in several ways. GNU poke provides support for some of them.

9.1 POSIX Time

In POSIX systems time is encoded as a certain number of seconds elapsed since some origin date, namely the first of January 1970. The `time` pickle provides two types for 32-bit and 64-bit POSIX time encodings:

```
POSIX_Time32
POSIX_Time64
```

A function `ptime` is also provided, that prints a human-readable representation of a POSIX date, given a number of seconds.

10 Colors

Colors are often found in binary data, encoded in several different ways. GNU poke provides several pickles that makes it easier to work with these colors.

10.1 The Color Registry

The `color` pickle provides a registry of *standard colors*, organized as a space of integers. Each integer identifies a standard color, which are accessible as variables named `color_*` after the pickle is loaded. Example:

```
(poke) load color
(poke) color_tomato
114
```

The purpose of having this register is to have a global namespace for colors that can be used in different pickles. The position of each color in the registry is totally unrelated to how the color may be encoded. Other pickles, as we shall see below, contain tables associating standard colors with their encoding, such as RGB.

If you want to add a new color that is not in the standard collection, you can use the `color_register` function, which gets no arguments:

```
(poke) var mycolor = color_register
(poke) mycolor
490
```

The total number of registered colors is recorded in the variable `color_num_colors`. You can use it to iterate on all the colors in the register, from `0` to `color_num_colors - 1`.

The index in the registry of the first user-defined color is hold in the variable `color_LAST`. For example, if you wanted to store a frob per standard color, you would do it like:

```
(poke) type Frobs = Frob[color_LAST]
```

The pickle also provides a function `color_name` that, given a color code, returns a printable name for the color, *i.e.* a string describing it. For user defined colors, this string is fixed:

```
(poke) color_name (23)
"lavender blush"
(poke) color_name (color_register)
"user-defined color"
```

If a non-existent color code is passed to `color_name` the function raises `E_out_of_bounds`:

```
(poke) color_name (color_num_colors)
unhandled out of bounds exception
```

10.2 RGB24 Encoding

The *RGB24* encoding encodes each color as a triplet of *color beans*, each beam indicating a level of red, green and blue respectively.

Types are provided for both the color beams, and the triplets:

```
type RGB24_Color_Beam = uint<8>;
type RGB24_Color = RGB24_Color_Beam[3];
```

The indexes `RGB24_RED`, `RGB24_GREEN` and `RGB24_BLUE` can be used to access to a specific beam of a given color:

```
(poke) rgb24_color[color_tomato][RGB24_GREEN]
99UB
```

The `rgb24` pickle also provides a table associating poke standard colors (see Section 10.1 [The Color Registry], page 95) with their RGB24 encodings. Once the pickle is loaded, this table is available in the variable `rgb24_color`. The table is to be indexed by color codes:

```
(poke) load rgb24
(poke) rgb24_color[color_tomato]
[255UB,99UB,71UB]
```

11 Audio

11.1 MP3

11.1.1 ID3V1 Tags

The `id3v1` pickle provides abstractions in order to edit the metadata stored in a MP3 file.

11.1.1.1 Song Genres

The ID3V1 tags support the notion of *song genre*. The space for genres is from 0 to 254. The genre code 255 is reserved to mean "no genre".

The table `id3v1_genres` can be indexed with a code in order to get the corresponding genre name:

```
(poke) id3v1_genres[14]
"rhythm and blues"
```

Conversely, the function `id3v1_search_genre` gives us the code of a genre, given its name. The prototype of this function is:

```
fun id3v1_search_genre = (string name) uint<8>:
```

For example:

```
(poke) id3v1_search_genre ("rock")
17UB
```

If `id3v1_search_genre` is given a name that doesn't correspond with any genre in the genres table, then `E_inval` is raised.

11.1.1.2 The ID3V1_Tag Type

The main data structure defined in the `id3v1` pickle is `ID3V1_Tag`, which corresponds to a ID3V1 tag (surprise!).

Tags comprise the following information:

- The title of the song, which is limited to 30 bytes.
- The artist, which is limited to 30 bytes.
- The album, which is limited to 30 bytes.
- The year, which is encoded as text in 4 bytes, each byte containing the ASCII code for the corresponding digit.
- A comment, which is limited to either 28 or 30 bytes, depending whether the tag contains track information or not.
- An optional track, which is an unsigned 8-bit number.

The specification does not mention any specific encoding for the entries that store text (such as title or artist), but it is safe to assume some ASCII-compatible encoding is used.

The text entries are stored as arrays of characters, and they are *not* finished by NULL characters. Instead the arrays of characters are filled with whitespaces (ASCII code 0x20) at the right. For example, the artist name Picasso encoded in ID3V1 would be:

```
['P','i','c','a','s','s','o',' ',' ', ..., ' ']
```

In order to ease the manipulation of the text fields, setters and getters are provided in order to handle these values as strings and not as whitespace-filled arrays of characters. Example:

```
(poke) tag.title
[48UB,49UB,32UB,45UB,32UB,...]
(poke) tag.get_title
```

```
"01 - Eclipse De Mar"
```

Also for setters:

```
(poke) tag.set_title ("Join us Now")
(poke) tag.title
[74UB,111UB,105UB,110UB,32UB,...]
```

Setters and getters are also provided in order to manipulate the year as an integer value:

```
(poke) tag.year
[0x31UB,0x39UB,0x38UB,0x30UB]
(poke) tag.get_year
1980
(poke) tag.set_year (1988)
```

12 Object Formats

This chapter is of special interest to toolchain developers, people doing reverse-engineering, and similar low-level development activities.

12.1 ELF

XXX

12.2 Dwarf

XXX

13 Programs

This chapter describes pickles whose purpose is to facilitate the writing of binary utilities in Poke.

13.1 argp

Like any other program, Poke scripts often need to handle options passed in the command line. The `argp` pickle provides a convenient and easy way to handle these arguments.

The main entry point of the pickle is the function `argp_parse`, with the following prototype:

```
fun argp_parse = (string program,
                  string version = "",
                  string summary = "",
                  Argp_Option[] opts = Argp_Option[](),
                  string[] argv = string[](),
                  int allow_unknown = 0) string[]
```

Where *program* is the name of the program providing the command-line options. *version* is either a string identifying the version of the program, or the empty string. *summary* is a short summary of what the program does. *opts* is an array of `Argp_Option` structs, each of which describe a command line option and, in particular, a handler for that option. *argv* is an array of string containing the command line elements to process. Finally, *allow_unknown* specifies whether unknown options constitute an error or not. This last option is useful in order to support sub-parsers.

Once invoked, `argp_parse` analyzes the command line provided in *argv*, recognizes options, performs checks making sure that all specified options are well formed, and that every option requiring an argument gets one, invokes the handlers registered for each option, and finally returns an array of non-option arguments in *argv*, for further processing by the user.

Each `Argp_Option` struct describes an option. They have this form:

```
type Argp_Option =
  struct
  {
    string name;
    string long_name;
    string summary;
    string arg_name;
    int arg_required;
    Argp_Option_Handler handler;
  };
```

Where *name* is a string of size one character specifying the short name for the option. *long_name* is a non-empty string of any size specifying the long name for the option. *summary* is a short paragraph with a summary of that the option does. *arg_name*, if specified, is a string to be used to describe the argument of the option in the `--help` output. If *arg_name* is not specified then `"ARG"` is used. *arg_required* is a boolean specifying whether the option accepts and expects an argument. Defaults to 0. Finally, *handler* is a function that will be invoked to process the option. The function has the following prototype:

```
type Argp_Option_Handler = (string)void
```

The `argp_parse` function provides default handlers for several standard options:

`--help` The default handler for this option prints out an usage message in the standard GNU way, and exits. In particular, this output is compatible with the `help2man` utility.

`--version`

The default handler for this option prints out the name and the version of the program, and exits.

If the user provides handlers for the options above, these take precedence wrt the default handlers.

Several short options can be accumulated this way:

```
foo -qyz
```

The special option `--`, if found in the command line, stops the processing of options. Everything found about it is considered non-option arguments.

14 Programming Emacs Modes

GNU poke ships with a bunch of Emacs major modes that eases writing Poke and PVM assembly. This chapter documents these modes.

14.1 poke-mode

`poke-mode` is a major mode for editing Poke source files, *i.e.* `.pk` files. It provides font-lock, auto-completion and indentation features.

14.2 poke-map-mode

`poke-map-mode` is a major mode for editing poke map-files. It provides font-lock. See Chapter 5 [Maps and Map-files], page 73.

14.3 poke-ras-mode

`poke-ras-mode` is a major mode for editing PVM assembly. It is used in the development of GNU poke. See the file `libpoke/ras` in the source distribution for more information about RAS, our Retarded Assembler.

15 Vim Syntax Highlighting

GNU poke ships with a vim syntax highlighter for *.pk files. Normally this should be installed as part of your existing poke installation and work out of the box. If that is not the case this chapter explains how to install it manually.

15.1 `poke.vim`

`etc/vim/syntax/poke.vim` is a syntax highlighter for Poke source *i.e.* `.pk` files. To use it, first place the file in your `~/.vimrc/syntax/` folder on Unix based systems, or in `$HOME/vimfiles/syntax/` on Windows. There are a few ways to tell Vim to use `poke.vim` for `.pk` files, but the quickest is to add the following line to your `.vimrc`:

```
au BufRead,BufNewFile *.pk set filetype=poke
```

16 Dot-Commands

16.1 `.load`

The `.load` command loads a file containing Poke code and compiles and executes it. These files usually have the extension `.pk`.

If a relative path is provided, then *`prefix`*`/share/poke` is tried first as a base directory to find the specified file. If it is not found, then the current directory is tried next.

If the environment variable `POKEDATADIR` is defined, it replaces *`prefix`*`/share/poke`. This is mainly intended to test a poke program before it gets installed in its final location.

If an absolute path is provided, it is used as-is.

16.2 `.source`

The `.source` command executes command lines from some given file. The syntax is:

```
.source path
```

where *path* is a path to the file containing the commands. See Section 1.4.4 [Command Files], page 6.

16.3 `.file`

The `.file` command opens a new IO space backed by a file, or switches to a previously opened file. The syntax is:

```
.file path
```

where *path* is a path to a file to open, which can be relative to poke's current working directory or absolute.

Tilde expansion is performed in *path*, much like it's done in the shell. This means you can include special characters like `~` (which will expand to your home directory) delimit the file name with `"` in case it includes leading or trailing blank characters, *etc.*

When a new file is opened it becomes the current IO space. From that point on, every map executed in the REPL or while loading a Poke program will operate on that IO space:

```
(poke) .file foo.o
The current file is now ‘foo.o’.
```

When `.file` opens a file, it is opened in whatever mode makes more sense: if the file allows to be written and read then the IO space is open in read/write mode, for example.

If the specified file path doesn't exist then poke emits an error. However, the flag `/c` (for "create") can be passed to the command to tell poke it should create a new file.

16.4 `.mem`

The `.mem` command opens a new IO space backed by a memory buffer. The syntax is:

```
.mem name
```

where *name* is the name of the buffer to create. Note that poke adds prefix and trailing asterisk characters, to differentiate file names from buffer names.

When a new memory buffer IOS is opened it becomes the current IO space. See Section 16.3 [file command], page 104.

16.5 .nbd

The `.nbd` command opens a new IO space backed by an external NBD server. The syntax is:

```
.nbd uri
```

where *uri* is the name of the newly created buffer, matching the NBD URI specification (`https://github.com/NetworkBlockDevice/nbd/blob/master/doc/uri.md`).

When a new NBD IOS is opened, it becomes the current IO space. See Section 16.3 [file command], page 104.

NBD support in GNU poke is optional, depending on whether poke was compiled against libnbd.

For an example of connecting to the guest-visible content of a qcow2 image, with the default export name as exposed by using qemu as an NBD server:

```
$ qemu-nbd --socket=/tmp/mysock -f qcow2 image.qcow2
$ poke
(poke) .nbd nbd+unix:///socket=?/tmp/mysock
The current file is now ‘nbd+unix:///socket=?/tmp/mysock’.
```

16.6 .proc

The `.proc` command opens a new IO space backed by the memory mapped for some live process, or switches to a previously opened process. The syntax is:

```
.proc pid
```

where *pid* is a process identifier. In most systems this is a number.

16.7 .sub

The `.sub` command opens a new IO sub-space. The syntax is:

```
.sub ios, base, size[, name]
```

where *ios* is the tag of some existing IO space, *base* and *size* are integers denoting the base and size of the range covered by the sub-space (in bytes) and *name* is an optional name.

16.8 .ios

A list of open files, and their corresponding tags, can be obtained using the `.info ios` command. Once a tag is known, you can use the `.ios` command to switch back to that file:

```
(poke) .ios #1
The current IOS is now ‘foo.o’.
```

16.9 .close

The `.close` command closes the selected IO space. The syntax is:

```
.close #tag
```

where *#tag* is a tag identifying an open IO space.

Note that closing an IO space with this dot-command implies closing any sub IO space having it as a base.

16.10 .doc

The `.doc` command is used to display this manual in poke's REPL. The syntax is:

```
.doc [node]
```

where *node* is an optional parameter which indicates the chapter or section at which the manual should be opened.

This command uses whatever documentation viewer configured using `.set doc-viewer` whose valid options are either `info` and `less`.

Unless `doc-viewer` is set to `less`, this command uses the info program (*The GNU Texinfo Manual*) to interactively present the manual. If info is not installed, then less is tried next.

When using less to display the documentation, the entry in the Table of Contents corresponding to the requested entry is highlighted. Just press `/` and then `RET` to jump to the corresponding section in the manual.

If neither info nor less are installed, the `.doc` command will fail. If poke is not running interactively then `.doc` does nothing.

16.11 .editor

The `.editor` command (usually abbreviated as `.edit`) invokes an external text editor on a temporary file. You can then put contents on that file, save it and exit the editor. At that point poke will read the file contents, turn them into a single line and execute them in the repl. If poke is not running interactively, then `.editor` does nothing.

The editor used is identified by the `EDITOR` environment variable.

16.12 .info

The `.info` command provides information about several kinds of entities. The recognized sub commands are:

`.info ios` Display a list of open files.

```
(poke) .info ios
  Id Type Mode Bias Size Name
  #1 FILE r 0x00000000#B 0x0000df78#B foo.o
* #0 FILE rw 0x00000000#B 0x00000022#B foo.bson
```

The file acting as the current IO space is marked with an asterisk character `*` at the beginning of the file. The mode in which the file is open is also specified. The `Id` field is the tag of the file that can be passed to the `.file` command in order to switch to it as the new current IO space:

```
(poke) .ios #1
The current file is now ‘foo.o’.
(poke) .info ios
  Id Type Mode Bias Size Name
* #1 FILE r 0x00000000#B 0x0000df78#B foo.o
  #0 FILE rw 0x00000000#B 0x00000022#B foo.bson
```

`.info variables [regexp]`

Shows a list of defined variables along with their current values and the location where the variables were defined. If a regular expression is provided then only the types whose name match the expression are listed. If a regular expression is provided then only the types whose name match the expression are listed.

`.info functions [`*`regexp`*`]`
: Shows a list of defined functions along with their prototypes and the location where the functions were defined. If a regular expression is provided then only the types whose name match the expression are listed. If a regular expression is provided then only the types whose name match the expression are listed.

`.info types [`*`regexp`*`]`
: Shows a list of defined types along with the locations where the types were defined. If a regular expression is provided then only the types whose name match the expression are listed.

`.info type` *`name`*
: Prints a description of the type with name *name*.

16.13 `.set`

The `.set` command allows you to inspect and set the value of global settings. The following invocations are valid:

`.set` Prints all settings along with their values.

`.set` *`setting`*
: Prints the value of the given *setting*.

`.set` *`setting value`*
: Sets the value of *setting* to *value*.

The following settings can be handled with `.set`:

`pretty-print`
: This setting determines whether poke should use pretty-printers while printing out the written representation of struct values.

  Pretty-printers are defined as methods named '_print'.

`auto-map` This setting determines whether to automatically load map files when opening IO spaces.

  The decision of what maps to load is driven by the contents of the array 'auto_map'. Each entry in the array is an array whose first element is a regular expression, and the second element is the name of a map.

  Example:

  auto_map = [["\.o", "elf"], ["\.o", "dwarf"]];

  The entry above makes poke to automatically load the maps 'elf.map' and 'dwarf.map' when opening any file whose name ends in '.o', provided 'auto-map' is set to 'yes'.

`prompt-maps`
: This setting determines whether a list of open maps is displayed before the poke prompt.

  Example of a prompt with map information and 'prompt-maps' set to 'yes':

  (poke) .file foo.o [dwarf,elf](poke)

  Which indicates that the opening of the IO space 'foo.o' resulted in the 'elf.map' and 'dwarf.map' maps to be automatically loaded.

`omaps` This sort of misnamed setting determines whether to show the mapped offsets and element/field offsets of composite values like arrays and structs.

  Example:

(poke) .set omaps yes (poke) [1,2,3] [0x1 @ 0x0UL#b,0x2 @ 0x20UL#b,0x3 @ 0x40UL#b] @ 0x0UL#b

This setting defaults to 'no'.

`obase` This setting determines the output numerical base to use when printing integral values like integers, offset magnitudes, and the like. The supported bases are 2 for binary, 8 for octal, 10 for decimal and 16 for hexadecimal.

`oacutoff` This setting determines the maximum number of elements to output when printing out array values. Once the maximum number of elements is reached, an ellipsis '...' is printed to represent the rest of the elements.

Example:

(poke) [1,2,3,4,5] [1,2,3,4,5] (poke) .set oacutoff 3 [1,2,3,...]

The default value is 0, which means no limit.

`odepth` This setting determines the maximum number of nested structs to output when printing struct and union values. Once the maximum number of nested values is reached, a collapsed form 'Type {...}' is printed.

Example:

(poke) type Foo = struct { struct { int i; } bar; } (poke) Foo {} Foo { bar=struct {...} }

The default value for 'odepth' is 0, which means infinite depth.

`oindent` This setting determines the width (in blank characters) of each indentation level when printing out struct values in 'tree' mode.

Example:

(poke) .set oindent 4 (poke) Foo {} Foo { i=0x0, l=0x0L }

This setting defaults to 2. See also ".help omode".

`omode` This setting determines the way binary struct data is displayed.

In 'flat' mode data is not formatted in any special way. Each value, be it simple or composite, is printed on one line.

In 'tree' mode composite values like structs and arrays are displayed in an hierarchical way, using newlines and indentation.

Examples:

(poke) .set omode flat (poke) Foo {} Foo {i=0x0,l=0x0L}

(poke) .set omode tree (poke) Foo {} Foo { i=0x0, l=0x0L }

The default value of 'omode' is 'flat'.

`endian` This setting determines the byte endianness used when accessing IO spaces. Valid values are 'big', 'little', 'host' and 'network'.

The meaning of 'big' and 'little' is the obvious one: the endianness is set to big-endian and little-endian respectively.

The 'host' endianness is the endianness used by the computer and operating system that is running poke in order to handle it's own memory and files.

The 'network' endianness is the endianness used by the host machine in order to communicate with the network.

`doc-viewer`

This setting determines what mechanism to use to open the online user manual.

If 'info' is selected, the GNU info standalone program is launched to show the info version of the user manual.

If 'less' is selected, the less utility is launched ot show a plain text version of the user manual.

See also ".help .doc".

16.14 .vm

The Poke Virtual Machine (PVM) executes the programs that are the result of the compilation of what you write in the REPL or the pickles you load. The `.vm` command provides sub-commands to interact with the PVM.

16.14.1 .vm disassemble

The `.vm disassemble` command provides access to the PVM disassembler. It supports the following subcommands:

`.vm disassemble expression` *expr*
: Dumps the assembler corresponding to the Poke expression *expr*.

`.vm disassemble function` *function*
: Dumps the assembler corresponding to the Poke function called *function*. The function shall be reachable from the top-level.

`.vm disassemble mapper` *expr*
: If *expr* is a mapped value, dumps the assembler corresponding to its mapper function.

`.vm disassemble writer` *expr*
: If *expr* is a mapped value, dumps the assembler corresponding to its writer function.

The disassembler will provide a PVM disassembly by default, but it can be passed the flag `/n` to do a native disassembly instead in whatever architecture running poke.

16.14.2 .vm profile

The `.vm profile` command provides access to the PVM profiler. This profiler is only available if poke has been configured with PVM profiling support. This command supports the following subcommands:

`.vm profile reset`
: Resets the profiling counts in the virtual machine.

`.vm profile show`
: Outputs a summary with both counts and sample information.

16.15 .exit

The `.exit` command exits poke. The syntax is:

```
.exit [status]
```

Poke will terminate, returning the exit status *status*. If *status* is omitted, then the exit status zero will be returned.

16.16 .quit

The `.quit` command behaves exactly like `.exit`. See Section 16.15 [exit command], page 109.

17 Commands

17.1 dump

At the most basic level, memory can be examined byte by byte. To do this, use the `dump` command.

This command has the following synopsis.

```
dump [:from offset] [:size offset] [:ios int] [:ruler bool]
     [:ascii bool] [:group_by int] [:cluster_by int]
```

All arguments are optional, which means the simplest use of the command is to simply type `dump`:

```
(poke) dump
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000000: 9b07 5a61 4783 f306 4897 f37c fe39 4cd3  ..ZaG...H..|.9L.
00000010: b6a2 a578 8d82 7b7f 2076 374c 3eab 7150  ...x..{. v7L>.qP
00000020: 31df 8ecb 3d33 ee12 429b 2e13 670d 948e  1...=3..B...g...
00000030: 86f1 2228 ae07 d95c 9884 cf0a d1a8 072e  .."(...\........
00000040: f93c 5368 9617 6c96 3d61 7b92 9038 a93b  .<Sh..l.=a{..8.;
00000050: 3b0d f8c9 efbd a959 88d0 e523 fd3b b029  ;......Y...#.;.)
00000060: e2eb 51d5 cb5b 5ba9 b890 9d7a 2746 72ad  ..Q..[[....z'Fr.
00000070: 6cbd 6e27 1c7f a554 8d2e 77f9 315a 4415  l.n'...T..w.1ZD.
```

The first row is the *ruler* which serves as a heading for each subsequent row. On the left hand side is the offset of the io space under examination. The centre block displays the hexadecimal representation of each byte, and on the right hand side is their ascii representation. If a byte is not representable in ascii, then the byte will be displayed as a dot.

By default the `dump` command reads from the currently selected IO space. However, it is possible to specify an explicit IO space using the `ios` option:

```
(poke) var myfile = open ("/path/to/file")
(poke) dump :ios myfile
```

17.1.1 Information `dump` shows

By default `dump` displays 128 bytes of memory starting at offset 0#B. You can change the quantity and starting offset by using the `size` and `from` arguments. For example:

```
(poke) dump :from 0x10#B :size 0x40#B
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789ABCDEF
00000010: b6a2 a578 8d82 7b7f 2076 374c 3eab 7150  ...x..{. v7L>.qP
00000020: 31df 8ecb 3d33 ee12 429b 2e13 670d 948e  1...=3..B...g...
00000030: 86f1 2228 ae07 d95c 9884 cf0a d1a8 072e  .."(...\........
00000040: f93c 5368 9617 6c96 3d61 7b92 9038 a93b  .<Sh..l.=a{..8.;
```

Note that both the `size` and `from` arguments are offsets. As such, both must be specified using `#` and an appropriate unit. (see Section 18.2.1 [Offset Literals], page 118).

The other arguments change the appearance of the dump. If the `ruler` argument is zero, then the ruler will be omitted:

```
(poke) dump :ruler 0 :size 0x40#B
00000000: b1fd 1608 2346 759c 46a6 aa94 6fcd 846a  ....#Fu.F...o..j
00000010: e39f 473f 3247 415f 174d a32b ed89 a435  ..G?2GA_.M.+...5
00000020: d2c6 2c52 bc82 e0a7 e767 31ea 84de 41e5  ..,R.....g1...A.
00000030: 2add 2869 e9c2 226b e222 8c74 4b94 af24  *.(i.."k.".tK..$
```

To omit the ascii representation of the memory, call `dump` with the `ascii` argument set to zero:

```
(poke) dump :ruler 0 :size 0x40#B :ascii 0
00000000: 4393 85e7 0b0c 3921 5a26 39ec 2f5f 5f15
00000010: cc46 e6f3 d50f 6ae6 8988 d50e f8c4 d1c6
00000020: 5a2f 7c3e 490b 18d8 d867 4b6f 2549 1f6c
00000030: 34a9 a0d7 24d2 e9ac 9240 8247 10cb 4ba1
```

17.1.2 Presentation options for `dump`

By default, the hexadecimal display shows two bytes grouped together, and then a space. You can alter this behaviour using the `group_by` parameter.

```
(poke) dump :ascii 0 :size 0x40#B :group_by 4#B
00000000: 68f19a63 df2a8886 c466631c a7fdd5c7  h..c.*...fc.....
00000010: 3075746a 0adb03ca f5b1ff14 6166fa07  0utj........af..
00000020: 0dd3cfbd 8eff46a2 4152a81d 471beddf  ......F.AR..G...
00000030: a0501cae 8bfcec6f 7a4f5701 45ba9fc3  .P.....ozOW.E...
```

Another parameter is the `cluster_by` argument. By setting `cluster_by` to *n*, this causes `dump` to display an additional space after the *n*th group has been displayed, and also in the corresponding position in the ascii display:

```
(poke) dump :size 0x40#B :group_by 2#B :cluster_by 4
76543210  0011 2233 4455 6677  8899 aabb ccdd eeff  01234567 89ABCDEF
00000000: 91b8 540d d4dc 49ae  3320 ba7d efd1 16ab  ..T...I. 3 .}....
00000010: b1a8 5ea0 5846 8bea  f741 3f80 42bc 201f  ..^.XF.. .A?.B. .
00000020: 6e5e fa50 23fb f16a  d380 be8c fc98 d195  n^.P#..j ........
00000030: 7bbf fa3e 3fc2 43a4  2a1e 9763 2bd6 5d24  {..>?.C. *..c+.]$
```

Another variable that the user can customize is `pk_dump_nonprintable_char`. It defaults to `'.'`, and contains the character to use in the ASCII dump to represent a non-printable character.

Not all bytes from an IO space can be read, depending on the kind of IO device. When `dump` cannot read a byte it will print `pk_dump_unknown_byte` instead, which is a string.

If the variable `pk_dump_addr_hyperlinks` is 1 then the `dump` command will emit hyperlinks in the printed bytes. Clicking on a byte value will result on the address of that byte to be inserted at the prompt.

If you have a personal preference on how memory dumps should appear, you can set the relevant `pk_dump_*` variables. These global variables serve as the defaults for `dump`, so this way, you will not need to explicitly pass them when you call the function.

17.2 `copy`

The command `copy` allows to copy regions of data inside an IO space, or between different IO spaces.

This command has the following synopsis.

```
copy [:from offset] [:to offset] [:size offset]
     [:from_ios ios] [:to_ios ios]
```

All arguments are optional. When invoked with no arguments, `copy` does nothing.

The arguments `from` and `size` determine the region to copy. By default, this region is taken from the current IO space, but this can be overwritten using the optional `from_ios` argument.

The argument `to` tells `copy` where to copy the stuff. It defaults to `from`. Again, this offset is applied to the current IO space by default, but this can be overwritten using the `to_ios` argument.

Note that it is allowed for the source and destination ranges to overlap.

17.3 save

Use the `save` command in order to write a region from an IO space into a file in your file system.

This command has the following synopsis.

```
save [:ios ios] [:from offset] [:size offset] [:file string]
     [:append bool] [:verbose bool]
```

All arguments are optional. When invoked with no arguments, `save` does nothing.

The arguments `from` and `size` are used to determine the region of the IO space to save. This is how you would extract and save the contents of an ELF section to a file `out.text`:

```
(poke) var s = elf_section_by_name (Elf64_Ehdr @ 0#B, ".text")
(poke) save :file "out.text" :from s.sh_offset :size s.sh_size
```

Both `from` and `size` are arbitrary offsets. You should keep in mind however that files are byte oriented. Therefore saving, say, nine bits to a file will actually write two bytes.

By default `save` will extract the data from the *current IO space*. However, it is possible to specify an alternative IOS by using the `ios` argument.

By default `save` will truncate the output file, if it exists, before starting writing. If the argument `append` is set as true, however, it will append to the existing contents of the file. In this case, the file should exist.

17.4 extract

Use the `extract` command in order to create a temporary memory IO space with the contents of a mapped value.

This command the has the following synopsis.

```
extract :val val :to ios_name
```

Where `:val` is a mapped value, and *:to* is the name of the memory IOS to create (or use). The contents of *value* are copied to the beginning of the memory IOS **ios_name**.

17.5 scrabble

The `scrabble` command rearranges portions of an IO space based on a transformation defined by a pair of patterns.

This command has the following synopsis:

```
scrabble [:from offset] [:size offset]
         [:from_pattern string] [:to_pattern string]
         [:ent_size offset]
         [:from_ios ios] [:to_ios ios]
```

The optional arguments are documented below.

`:from` *offset*
: Beginning of the range in the origin IO space from where to start rearranging units. Defaults to `0#B`.

`:size` *offset*
: Size of the range to scrabble in the origin IO space. Defaults to `0#B`.

`:from_pattern` *string*
: String specifying a sequence of units. Each character is an unit. Defaults to `""`.

`:to_pattern` *string*
: String specifying the scrabbled units. Each character is an unit. If a character that doesn't appear in the `from_pattern` is found in this string, it is ignored. Defaults to `from_pattern`.

`:ent_size` *offset*
: Size of the entities to scrabble around. Defaults to `1#B`.

`:from_ios` *ios*
: IO space from where to obtain the data to scrabble. Defaults to the current IO space.

`:to_ios` *ios*
: The IO space to where write the scrabbled data. Defaults to `from_ios`.

18 The Poke Language

18.1 Integers

Most of the values manipulated in Poke programs are whole numbers, also typically known as *integers* in computing parlance. This is because integers are pervasive in binary formats, often featuring unusual characteristics in terms of size and/or alignment. Single bits denoting flags or packed small integers are good examples of this. In order to ease the manipulation of such entities, and unlike most programming languages, Poke provides integer types of any number of bits and a rich set of accompanying operators.

18.1.1 Integer Literals

Integers literals can be expressed in several numeration bases.

Decimal numbers use the usual syntax `[1-9][0-9]*`. For example, `2345`.

Octal numbers are expressed using a prefix `0o` (or `0O`) followed by one or more digits in the range `[0-7]`. Examples are `0o0`, `0o100` and `0o777`.

Hexadecimal numbers are expressed using a prefix `0x` (or `0X`) followed by one or more hexadecimal digits in the range `[0-f]`. Examples are `0x0` and `0xfe00ffff`. Note that both the `x` in the prefix and the letters in the hexadecimal number are case insensitive. Thus, `0XdeadBEEF` is a valid (but ugly as hell) literal.

Binary numbers are expressed using a prefix `0b` (or `0B`) followed by one or more binary digits in the range `[0-1]`. Examples of binary literals are `0b0` and `0B010`.

Negative numbers, of any numeration base, are constructed using the minus operator as explained below. Therefore the minus symbol `-` in negative numbers is not part of the literal themselves.

18.1.1.1 The digits separator _

The character `_` can appear anywhere in a numeric literal except as the first character. It is ignored, and its purpose is to make it easier for programmers to read them:

```
0xf000_0000_0000_0000
0b0000_0001_0000_0001
```

18.1.1.2 Types of integer literals

The type of a numeric literal is the smallest signed integer capable of holding it, starting with 32 bits, in steps of powers of two and up to 64 bits.[1]

So, for example, the value `2` has type `int<32>`, but the value `0xffff_ffff` has type `int<64>`, because it is out of the range of signed 32-bit numbers.

A set of suffixes can be used to construct integer literals of certain types explicitly. `L` or `l` is for 64-bit integers. `H` or `h` is for 16-bit integers (also known as *halves*), `B` or `b` is for 8-bit integers (also known as *bytes*) and `n` or `N` is for 4-bit integers (also known as *nibbles*).

Thus, `10L` is a 64-bit integer with value `0x0000_0000_0000_000A`, `10H` is a 16-bit integer with value `0x000A` and `10b` is a 8-bit integer with value `0x0A`.

Similarly, the signed or unsigned attribute of an integer can be explicitly specified using the suffix `u` or `U` (the default are signed types). For example `0xffff_ffffU` has type `uint<32>` and `0ub` has type `uint<8>`. It is possible to combine width-indicating suffixes with signedness suffixes: `10UL` denotes the same literal as `10LU`.

[1] Rationale: the width of a C "int" is 32 bits in most currently used architectures, and binary data formats are usually modelled after C.

The above rules guarantee that it is always possible to determine the width and signedness of an integer constant just by looking at it, with no ambiguity.

18.1.2 Characters

8-bit unsigned integers can use an alternative literal notation that is useful when working with *ASCII character codes*. Printable character codes can be denoted with `'c'`.

Non-printable characters can be expressed using escape-sequences. The allowed sequences are:

`\n` New-line character (ASCII 012).

`\t` Tab character (ASCII 011).

`\\` The backslash character.

`\[0-9][0-9]?[0-9]?`
Character whose ASCII code is the specified number, in octal.

Examples:

```
'o'
'\n'
'\t'
'\\'
'\0'
```

The type of a character literal is always `char`, aka `uint<8>`.

18.1.3 Booleans

Like in C, truth values in Poke are encoded using integers. Zero (`0`) denotes the logical value "false", and any integer other than zero denotes the logical value "true".

18.1.4 Integer Types

Most general-purpose programming languages provide a small set of integer types, each featuring a range corresponding to strategic storage sizes: basically, signed and unsigned variants of 8, 16, 32, 64 bits. As we have seen in the previous sections, suffixes like `H` or `L` are used in Poke for that purpose.

However, in conventional programming languages when integers having an "odd" width (like 13 bits, for example) get into play for whatever reason, the programmer is required to use the integer arithmetic operators (and sometimes bit-wise operators) herself, in a clever way, in order to achieve the desired results.

Poke, on the contrary, provides a rich set of integer types featuring different widths, in both signed and unsigned variants. The language operators are aware of these types, and will do the right thing when operating on integer values having different widths.

Unsigned integer types are specified using the type constructor `uint<`*n*`>`, where *n* is the number of bits. *n* should be an integer literal in the range `[1,64]`. Examples:

```
uint<1>
uint<7>
uint<64>
```

Similarly, signed integer types are created using the type constructor `int<`*n*`>`, where *n* is the number of bits. *n* should be an integer literal in the range `[1,64]`. Examples:

```
int<1>
int<8>
int<64>
```

Note that expressions are not allowed in the type integral constructor parameters. Not even constant expressions. Thus, things like `int<foo>` and `uint<2+3>` are not allowed.

18.1.5 Casting Integers

The right-associative unary operator cast `as` can be used to derive a new integer value having a different type from an existing value.

For example, this is how we would create a signed 12-bit integer value holding the value 666:

```
(poke) 666 as int<12>
(int<12>) 666
```

Note that the `666` literal is originally a 32-bit signed integer. The cast performs the conversion.

Casts between integer types are always allowed, and never fail. If the new type is narrower than the existing type, truncation may be performed to accommodate the value in its new type. For example, the expression `0x8765_4321 as uint<16>` evaluates to `0x4321`. If the new type is wider than the existing type, either zero-extension or sign-extension is performed depending on the signedness of the operand.

The semantics of the sign-extension operation depends on the signedness of the value being converted, and on the currently selected encoding for negative numbers.

When using two's complement encoding, converting a signed value will always sign-extend regardless of the signedness of the target type. Thus:

```
(poke) -2H as uint<32>
0xfffffffeU
(poke) -2H as int<32>
0xfffffffe
```

Likewise, converting an unsigned value will always zero-extend regardless of the signedness of the target type. Thus:

```
(poke) 0xffffUH as uint<32>
0xffffU
(poke) 0xffffUH as int<32>
0xffff
```

18.1.6 Relational Operators

The following binary relational operators are supported on integer values, in descending precedence order:

- Equality `==` and inequality `!=`.
- Less than `<` and less or equal than `<=`.
- Greater than `>` and greater or equal than `>=`.

When applied to integer and character values, these operators implement an arithmetic ordering.

These operators resolve in boolean values encoded as 32-bit integers: `0` meaning false and `1` meaning true.

18.1.7 Arithmetic Operators

The following left-associative binary arithmetic operators are supported, in descending precedence order:

- Exponentiation `**`, multiplication `*`, integer division `/`, integer ceil-division `/^` and modulus `%`.
- Addition `+` and subtraction `-`.

In all the binary arithmetic operations automatic promotions (coercions) are performed in the operands as needed. The rules are:

- If one of the operands is unsigned and the other operand is signed, the second is converted to an unsigned value.
- If the size in bits of one of the operands is bigger than the size of the other operand, the second is converted to the same number of bits.

The following right-associative unary arithmetic operators are supported:

- Unary minus `-` and unary plus `+`.

Finally, pre-increment, pre-decrement, post-increment and post-decrement operators `++` and `--` are supported.

18.1.8 Bitwise Operators

The following left-associative bitwise binary operators are supported, in descending precedence order:

- Bitwise shift left `<<.` and bitwise shift right `.>>`.
- Bitwise AND `&`.
- Bitwise exclusive OR `^`.
- Bitwise inclusive OR `|`.
- Bitwise concatenation `:::`.

Both `<<.` and `.>>` operators perform logical shifting. Unlike in many other programming languages, arithmetic right-shifting operators are not provided. This means that right shifting always inserts zeroes at the most-significant side of the value operand, whereas left shifting always inserts zeroes at the least-significant side of the value operand.

Left shifting by a number of bits equal or bigger than the size of the value operand is an error, and will trigger either a compile-time error or a run-time `E_out_of_bounds` exception. This does not apply to right shifting.

Bitwise concatenation works with any integral type, of any bit length.

The following right-associative unary bitwise operators are supported:

- Bitwise complement `~`.

18.1.9 Boolean Operators

The following left-associative, short-circuited binary logical operators are supported, in descending precedence order:

- Logical implication: `=>`.
- Logical AND: `&&`.
- Inclusive OR: `||`.

The following right-associative unary logical operators are supported:

- Logical negation `!`.

18.1.10 Integer Attributes

The following attributes are defined for integer values.

`size` Gives an offset with the storage occupied by the string. This includes the terminating null. Examples:

```
(poke) 10'size
0x20UL#b
```

```
(poke) 10N'size
0x4UL#b
(poke) (10 as int<1>)'size
0x1UL#b
```

`signed` Gives 1 if the value is a signed integer, 0 otherwise. Examples:

```
(poke) 10'signed
1
(poke) 10UL'signed
0
```

`mapped` Always 0 for integers. (see Section 18.14 [Mapping], page 152).

`length` Always `1UL` for integers.

18.2 Offsets

Poke uses united values to handle offsets and data sizes. This is a very central concept in poke.

18.2.1 Offset Literals

Poke provides a convenient syntax to provide united values, which are called *offsets* (because in a binary editor you mostly use them to denote offsets in the file you are editing):

```
12#B
7#b
1024#KB
```

The offsets above denote twelve bytes, seven bits and one thousand twenty four kilobytes, respectively. The unit can be separated from the magnitude by blank characters, so you can write the following instead if you are so inclined:

```
12 #B
7 #b
(1024 * 1024) #Kb
```

Note how the magnitude part of an offset doesn't need to be constant. If the variable `a` contains an integer, this is how you would denote "a bytes":

```
a#B
```

In the offset syntax units are specified as `#`*`unit`*, where *unit* is the specification of an unit. See the next section for details.

18.2.2 Offset Units

There are several ways to express the unit of an offset, which is always interpreted as a multiple of the basic unit, which is the bit (one bit).

18.2.2.1 Named Units

The first way is to refer to an unit by name. For example, `2#B` for two bytes. Units are defined using the `unit` construction:

```
unit name = constant_expression [, name = constant_expression]...;
```

where *name* is the name of the unit, and *constant_expression* is a constant expression that should evaluate to an integral value. The resulting value is always coerced into an unsigned 64-bit integer.

Note that unit names live in a different namespace than variables and types. However, when a given name is both a type name and an unit name in an unit context, the named unit takes precedence:

```
(poke) type xx = int
```

```
(poke) unit xx = 2
(poke) 1#xx
1#2
```

Many useful units are defined in the standard library. See Section 19.3 [Standard Units], page 170.

18.2.2.2 Arbitrary Units

It is also possible to express units in multiples of the base unit, which is the bit. Using this syntax, it becomes possible to express offsets in any arbitrary unit, as disparate as it may seem:

```
17#3
0#12
8#1
```

That's it: 17 units of 3 bits each, zero units of 12 bits each, and eight units of 1 bit each. Note that the unit should be greater than 0.

18.2.2.3 Types as Units

But then, why stop there? Poking is all about defining data structures and operating on them... so why not use these structures as units as well? Consider the following struct:

```
type Packet = struct { int i; long j; };
```

The size of a `Packet` is known at compile time (which is not generally true for Poke structs). Wouldn't it be nice to use it as a unit in offsets? Sure it is:

```
23#Packet
```

The above is the size occupied by 23 packets. Any type whose size is known at compile time can be specified as an offset unit.

Expressing offsets as united values also relieves the programmer from doing many explicit unit conversions: poke can do them for you. Consider for example an ELF section header. One of its fields is the size of the described section, in bytes:

```
type Elf64_Shdr =
  struct
  {
   ...
   offset<Elf64_Xword,B> sh_size;
   ...
  };
```

If a given section is to contain, say, relocations with addends, we can set its size doing something like this:

```
shdr.sh_size = 10#Elf64_Rela;
```

Instead of doing the conversion to bytes explicitly.

If the magnitude of an offset is 1 then it is allowed to omit it entirely. To denote one kilobyte, for example, we can write `#KB`.

18.2.3 Offset Types

Offset types are denoted as `offset<`*`base_type`*`,`*`unit`*`>`, where *base_type* is an integer type and *unit* the specification of an unit.

The offset base type is the type of the magnitude part of the united value. It can be any integer type, signed or unsigned, of any size.

The unit specification should be one of the unit identifiers that are allowed in offset literals (see above), a constant positive integer or the name of a Poke type whose size is known at compile time.

Let's see some examples. A signed 32-bit offset expressed in bytes has type `offset<int<32>,B>`. An unsigned 12-bit offset expressed in kilobits has type `offset<uint<12>,Kb>`. The latter type can also be written using an explicit integer unit like in `offset<uint<12>,1024>`. Finally, a signed 64-bit offset in units of "packets", where a packet is denoted with a Poke type `Packet` has type `offset<uint<64>,Packet>`.

18.2.4 Casting Offsets

The right-associative unary operator cast `as` can be used to derive a new offset value having a different type from an existing value.

For example, this is how we would create a signed 12-bit offset in units bytes:

```
(poke) 1024#b as offset<int<12>,B>
(int<12>) 128#B
```

The same rules governing conversion of integers apply for the magnitude part. Depending on the unit, there can be truncation, like in:

```
(poke) 9#b as offset<int,B>
1#B
```

18.2.5 Offset Operations

Poke supports a little algebra for offsets.

18.2.5.1 Addition and subtraction

The addition or subtraction of two offsets results in another offset. Examples:

```
(poke) 1#B + 1#b
9#b
(poke) 2#KB - 1024#B
1024#B
```

The unit of the result is the greatest common divisor of the units of the operands.

The operators `++` and `--`, in their prefix and suffix versions, can be applied to offsets as well. The step used in the increment/decrement is one unit.

18.2.5.2 Multiplication by a scalar

Multiplying an offset by a magnitude gives you another offset. Examples:

```
(poke) 8#b * 2
16#b
(poke) 16#MB * 0
0#MB
```

The unit of the result is the same as the unit of the offset operand.

Note that multiplying two offsets is not supported. This makes sense, since computer memory is linear, and therefore it wouldn't make any sense to have units like $\#B^2$.

18.2.5.3 Division

Dividing two offsets gives you a magnitude. Examples:

```
(poke) 16#b / 1#B
2
(poke) 1024#MB / 512#Mb
16
```

Dividing offsets is the Pokish way of converting memory magnitudes between different units: just use units like you do when doing physics or working with units in other contexts.

For example, using the syntactic trick of omitting the magnitude (in which case it is assumed to be 1) it is very natural to write something like the following to convert from kilobits to bytes:

```
(poke) 24 #Kb/#B
3072
```

There is also a ceil-division operator for offsets, with the same semantics as the ceil-division for integers:

```
(poke) 10#B /^ 3#B
4
```

18.2.5.4 Division by an integer

Dividing an offset by an integer gives you an offset. Example:

```
(poke) 8#B / 2
4#B
```

18.2.5.5 Modulus

The modulus of two offsets gives you another offset with the expected semantics. Examples:

```
(poke) 9#b % 1#B
1#b
(poke) 1#B % 9#b
8#b
```

The unit of the result is the greatest common divisor of the units of the operands.

18.2.6 Offset Attributes

The following attributes are defined for offset values.

`size` Gives an offset with the storage occupied by the offset. Examples:

```
(poke) 10#B'size
0x20UL#b
(poke) 10N#B'size
0x4UL#b
```

`magnitude`
Gives the magnitude part of the offset. Examples:

```
(poke) 10#B'magnitude
10
(poke) 2H#b'magnitude
2H
```

`unit` Gives a number with the unit of the offset, expressed in bits. Examples:

```
(poke) 10#B'unit
8UL
(poke) 2H#b'unit
1UL
```

`mapped` Always 0 for offsets. (see Section 18.14 [Mapping], page 152).

`length` Always `1UL` for integers.

18.3 Strings

Poke supports a notion of *strings* which is very similar to the C programming language: a string value is a sequence of characters that is terminated by the so-called *null character*.

The standard library provides functions which process strings. See Section 19.6 [String Functions], page 172.

18.3.1 String Literals

NULL-terminated sequences of ASCII codes can be denoted using the following syntax:

```
"foo"
```

Poke string values are very similar to C strings. They comprise a sequence of 8-bit character codes, terminated by the value `0UB`.

The following escape sequences are supported inside string literals:

`\n` Denotes a new line character.

`\t` Denotes an horizontal tab.

`\\` Denotes a backlash `\` character.

`\"` Denotes a double-quote `"` character.

`\`*num* *num* is a number in the range 1 to 255 (inclusive) in base 8.

`\x`*num* *num* is a number in the range 1 to 255 (inclusive) in base 16.

Strings cannot contain `0UB` character, and inserting one using escape sequences (`\0` or `\x0`) leads to compilation error.

18.3.2 String Types

Every string value in Poke is of type `string`.

18.3.3 String Indexing

Poke supports accessing the characters in a string using the array indexing notation. The indexes are in the `[0,`*n*`]` range, where *n* is the length of the string minus one. Note the length doesn't include the null character, *i.e.* it is not possible to access the terminating null. Examples:

```
(poke) "foo"[0]
0x66UB
(poke) "foo"[1]
0x6fUB
```

If the passed index is less than zero or it is too big, an `E_out_of_bounds` exception is raised:

```
(poke) "foo"[-1]
unhandled out of bounds exception
(poke) "foo"[3]
unhandled out of bounds exception
```

18.3.4 String Concatenation

Strings can be concatenated using the `+` operator. This works like this:

```
(poke) "foo" + "bar"
"foobar"
```

Note how the null character terminating the first string is removed. Therefore, the length of the concatenation of two given strings of lengths `N` and `M` is always `N+M-1`.

Concatenation and indexing are useful together for building strings. A string can be created empty, and additional characters added to it by means of concatenation:

```
(poke) var bytes = "";
(poke) bytes = bytes + 'x' as string;
```

Then, we can retrieve characters from the string we built using indexing:

```
(poke) bytes[0]
0x78UB
```

Additionally, the `*` operator allows to "multiply" a string by concatenating it with itself a given number of times. This works like this:

```
(poke) "foo" * 3
"foofoofoo"
(poke) "foo" * 0
""
```

This is useful for building strings whose length is not known at compile time. For example:

```
fun make_empty_string = (int length) string:
{
   return " " * length;
}
```

18.3.5 String Attributes

The following attributes are defined for string values.

`length` Gives the number of characters composing the string, not counting the terminating null. Examples:

```
(poke) "foo"'length
3UL
(poke) ""'length
0UL
```

`size` Gives an offset with the storage occupied by the string. This includes the terminating null. Examples:

```
(poke) "foo"'size
32UL#b
(poke) ""'size
8UL#b
```

`mapped` Always 0 for strings. (see Section 18.14 [Mapping], page 152).

18.3.6 String Formatting

Poke provides a built-in function `format` that can be used in order to format, or build, a string value that is derived from a set of zero or more Poke values:

```
format (fmt[, value...])
```

Where *fmt* is a format string that interpreted verbatim but for predefined *format tags* which start with `%`. These format tags are interpreted especially. See Section 18.17.2 [printf], page 164.

Example:

```
var s = format ("Age is %i32d.", 66);
```

The value of the variable `s` is `"Age is 66."`.

18.4 Arrays

Arrays are homogeneous collections of values.

18.4.1 Array Literals

Array literals are constructed using the following syntax:

```
[exp,exp...]
```

where *exp* is an arbitrary expression.

For example, `[1,2,3]` constructs an array of three signed 32-bit integers. Likewise, `['a','b','c']` constructs an array of three unsigned 8-bit integers (characters). For convenience, a trailing comma is accepted but not required.

The type of the array literal is inferred from the type of its elements. Accordingly, all the elements in an array literal must be of the same type. Examples of invalid array literals, that will raise a compilation-time error if evaluated, are:

```
[1,2u,3]
[1,0xffff_ffff,3]
['a',"b",'c']
```

Array literals must contain at least one element. Accordingly, `[]` is not a valid array literal.

This is how a `3x3` matrix could be constructed using an array of arrays:

```
[[1,2,3],[4,5,6],[7,8,9],]
```

It is possible to refer to specific elements when constructing array literals. For example, `[1,2,3,.[3] = 4]` denotes the same array as `[1,2,3,4]`.

This allows creating arrays without having to specify all its elements; every unspecified element takes the value of the first specified element to its right. For example, `[.[2] = 2]` denotes the same array as `[2,2,2]`.

Note that an array element can be referenced more than once. When that happens, the final value of the element is the last specified. For example, `[1,2,3,.[1]=10]` denotes the array `[1,10,3]`.

18.4.2 Array Types

There are three different kind of array types in Poke.

Unbounded arrays have no explicit boundaries. Examples are `int[]` or `Elf64_Shdr[]`. Arrays can be *bounded by number of elements* specifying a Poke expression that evaluates to an integer value. For example, `int[2]`. Finally, arrays can be *bounded by size* specifying a Poke expression that evaluates to an offset value. For example, `int[8#B]`.

18.4.2.1 Writing unbounded array literals

The type of an array literal is always bounded by number of elements. For example, the type of `[1,2,3]` is `int[3]`. If what we want is an unbounded array literal we can obtain it with a case like `[1,2,3] as int[]`.

18.4.2.2 Array boundaries and closures

Poke arrays are rather peculiar. One of their seemingly bizarre characteristics is the fact that the expressions calculating their boundaries (when they are bounded) evaluate in their own lexical environment, which is captured. In other words: the expressions denoting the boundaries of Poke arrays conform closures. Also, the way they evaluate may be surprising. This is no capricious.

When an array type is bounded, be it by number of elements or by size, the expression indicating the boundary doesn't need to be constant and it can involve variables. For example, consider the following type definition:

```
var N = 2;
type List = int[N*2];
```

Let's map a `List` at some offset:

```
(poke) List @ 0#B
[0x746f6f72,0x303a783a,0x723a303a,0x3a746f6f]
```

As expected, we get an array of four integers. Very good, obviously the boundary expression `N*2` got evaluated when defining the type `List`, and the result of the evaluation was `4`, right?. Typical semantics like in my garden variety programming language. . . ? Right?!?

Well, not really. Let's modify the value of `N` and map a `List` again...

```
(poke) N = 1
(poke) List @ 0#B
[0x746f6f72,0x303a783a]
```

Yes, The boundary of the array type changed... on, this is Poke, was you **really** expecting something typical? ;)

What happens is that at type definition time the lexical environment is captured and a closure is created. The body of the closure is the expression. Every time the type is referred, the closure is re-evaluated and a new value is computed.

Consequently, if the value of a variable referred in the expression changes, like in our example, the type itself gets updated auto-magically. Very nice but, why is Poke designed like this? Just to impress the cat? Nope.

In binary formats, and also in protocols, the size of some given data is often defined in terms of some other data that should be decoded first. Consider for example the following definition of a `Packet`:

```
type Packet =
  struct
  {
    byte size;
    byte[size] payload;
  };
```

Each packet contains a 8-bit integer specifying the size of the payload transported in the packet. The payload, a sequence of `size` bytes, follows.

In struct types like the above, the boundaries of arrays depend on fields that have been decoded before and that exist, like variables, in the lexical scope captured by the struct type definition (yes, these are also closures, but that's for another article.) This absolutely depends on having the array types evaluate their bounding expressions when the type is used, and not at type definition time.

To show this property in action, let's play a bit:

```
(poke) var data = byte[4] @ 0#B
(poke) data[0] = 2
(poke) data[1] = 3
(poke) data[2] = 4
(poke) data[3] = 5
(poke) dump
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff
00000000: 0203 0405 0000 0000 0000 0000 0000 0000
00000010: 0000 0000 0000 0000 0000 0000 0000 0000
(poke) var p1 = Packet @ 0#B
(poke) var p2 = Packet @ 1#B
(poke) p1
Packet {size=0x2UB,payload=[0x3UB,0x4UB]}
(poke) p2
Packet {size=0x3UB,payload=[0x4UB,0x5UB,0x0UB]}
```

Now, let's change the data and see how the sizes of the payloads are adjusted accordingly:

```
(poke) data[0] = 1
(poke) data[1] = 0
(poke) p1
Packet {size=0x1UB,payload=[0x0UB]}
```

```
(poke) p2
Packet {size=0x0UB,payload=[]}
```

So, as we have seen, Poke's way of handling boundaries in array types allows data structures to adjust to the particular data they contain, so usual in binary formats. This is an important feature, that gives Poke part of its feel and magic.

18.4.3 Casting Arrays

There are two kinds of casts for array values:

- Cast to array types that may have a different bounding.
- Cast of array of integers or integral structs to an integral type.

For example, we have seen that the type of an array literal is always bounded by a constant number of elements. Thus the type of `[1,2,3]` is `int[3]`.

We could cast the value above to an unbounded array type by doing:

```
[1,2,3] as int[]
```

Or to an array bounded by size:

```
[1,2,3] as int[12#B]
```

Of course trying to cast an array value to a type for which the bounding restrictions will fail. This will give you a compile-time error:

```
[1,2,3] as int[4]
```

And this will result in a run-time error:

```
(poke) [1,2,3] as int[13#B]
unhandled conversion error exception
```

Now let's see how Poke reinterprets integral arrays as integers:

```
(poke) [0xdeUB,0xadUB] as uint16
0xdeadUH
(poke) [0xdeUB,0xadUB,0UB] as int
0xdead00
```

Trying to cast an array with a size larger than 64-bit will result in an error:

```
(poke) [1,2,3] as int
unhandled conversion error exception
arrays bigger than 64-bit cannot be casted to integral types
```

It also works for nested arrays and arrays of integral structs (See Section 18.5.11 [Integral Structs], page 136):

```
(poke) type Foo = struct uint { int16 hi; int16 lo; };
(poke) [[Foo{ hi=0xdeadH, lo=0xbeefH}],[Foo{}]] as ulong
0xdeadbeef00000000UL
```

The way it works is like putting all array elements in an `uint<64>` number and then casting that number to the destination integral type.

18.4.4 Array Constructors

Array literals can only specify arrays which a constant number of elements. They also require initial values for their elements, and it is also not possible to construct an array literal denoting an empty array of some given type.

Array constructors provide a suitable way to build such array values. The syntax is:

array_type ([*value*])

Where *array_type* is an array type, or the name of an array type, and *value* is an optional argument that specify the value to which the elements of the new array are initialized. If no value is provided in the constructor then default values are calculated.

Examples:

```
(poke) int[]()
[]
(poke) int[2]()
[0,0]
(poke) int[2](10)
[10,10]
(poke) offset<int,B>[2](16#b)
[2#B,2#B]
(poke) string[3]()
["","",""]
(poke) int[][3](int[2](666))
[[666,666],[666,666],[666,666]]
(poke) Packet[3]()
[Packet {...}, Packet {...}, Packet {...}]
```

18.4.5 Array Comparison

The equality operator (`==`) and the inequality operator (`!=`) can be applied to arrays. Examples:

```
(poke) [1,2,3] == [1,2,3]
1
(poke) [[1,2],[3,4]] != [[5,6],[7,8]]
1
```

Note that the array elements are compared recursively.

18.4.6 Array Indexing

Arrays are indexed using the usual notation, providing an index enclosed between square brackets with `[` and `]`:

```
(poke) [1,2,3][0]
1
(poke) [1,2,3][1]
2
```

The index should be an expression that evaluates to an integer value, and it is promoted to an unsigned 64-bit integer when needed.

The valid range for the index is `[0,`*n*`]` where *n* is the number of elements stored in the array minus one. If the passed integer is out of that range, an `E_out_of_bounds` exception is raised:

```
(poke) [1,2,3][-1]
unhandled out of bounds exception
(poke) [1,2,3][3]
unhandled out of bounds exception
```

18.4.7 Array Trimming

Indexing is used to fetch elements from arrays. Another operation, called *trimming*, allows you to extract a subset of the array, as another array.

Trims use the following notation, where a range is specified between square brackets. The left sides of the range are closed, whereas the right sides of the range are open:

```
(poke) [1,2,3][0:2]
```

```
[1,2]
(poke) [1,2,3][1:2]
[2]
(poke) [1,2,3][0:3]
[1,2,3]
```

If the minimum side of the range is omitted, it is assumed to be zero. If the maximum side of the range is omitted, it is assumed to be the length of the trimmed array:

```
(poke) [1,2,3][:2]
[1,2]
(poke) [1,2,3][1:]
[2,3]
(poke) [1,2,3][:]
[1,2,3]
```

The elements of the base array and the trimmed sub-array are copied by shared value, exactly like when passing arguments to functions. This means that for simple types, copies of the elements are done:

```
(poke) var a = [1,2,3]
(poke) var s = a[1:2]
(poke) s[0] = 66
(poke) a
[1,2,3]
```

However, for complex types like arrays and structs, the values are shared:

```
(poke) var a = Packet[] @ 0#B
(poke) var s = a[1:2]
(poke) s[0].field = 66
(poke) a[1].field
66
```

There is an alternative syntax that can be used in order to denote trims, in which the length of the trim is specified instead of the index of the second element. It has this form:

```
(poke) [1,2,3][0+:3]
[1,2,3]
(poke) [1,2,3][1+:2]
[2,3]
(poke) [1,2,3][1+:0]
[]
```

18.4.8 Array Elements

The `in` operator can be used to determine whether a given element is stored in an array. Examples:

```
(poke) 2 in [1,2,3]
1
(poke) 10#B in [2#b,10*8#b,3#b]
1
(poke) 30#B in [2#b,10*8#b,3#b]
0
```

18.4.9 Array Concatenation

The operator `+` can be used to build new arrays resulting from the concatenation of the elements of two arrays given as operands:

```
(poke) [1,2] + [3,4,5]
```

```
[1,2,3,4,5]
```

The two arrays given as operands shall have the same elements of the same type. The resulting array is always unbounded.

18.4.10 Array Attributes

The following attributes are defined for array values.

`size` Gives an offset with the storage occupied by the complete array. Example:

```
(poke) [1,2,3]'size
96UL#b
```

`length` Gives the number of elements stored in the array. Example:

```
(poke) [1,2,3]'length
3
```

`mapped` Boolean indicating whether the array is mapped. Examples:

```
(poke) var a = [1,2,3]
(poke) var b = int[3] @ 0#B
(poke) a'mapped
0
(poke) b'mapped
1
```

`strict` Boolean indicating whether the array is mapped in strict mode. Examples:

```
(poke) var a = int[3] @ 0#B
(poke) a'strict
1
(poke) var a = int[3] @! 0#B
(poke) a'strict
0
```

18.5 Structs

Structs are the main abstraction that Poke provides to structure data. They contain heterogeneous collections of values.

18.5.1 Struct Types

A simple struct type definition in Poke looks like:

```
type Packet =
  struct
  {
    byte magic;
    uint<16> length;
    byte[length] data;
  }
```

The above defines a "Packet", which consists on a magic number encoded in a byte, a length encoded in an unsigned 16-bit integer, and an array of `length` bytes, which is the payload of the packet.

Each entry in the struct type above defines a *struct field*.

Each field has a type, which is mandatory, and a name, which is optional. Fields without names are not accessible (not even within the struct itself) but they are handy for expressing esoteric padding. See Section 4.11 [Padding and Alignment], page 60. Example:

```
type Imm64 =
```

```
struct
{
  uint<32> lo;
  uint<32>;
  uint<32> hi;
}
```

It is not allowed to have several fields with the same name in the same struct. The compiler will complain if it finds such an occurrence.

18.5.2 Struct Constructors

Once a struct type gets defined, there are two ways to build struct values from it. One is *mapping*. See Section 18.14.4 [Mapping Structs], page 156. The other is using a *struct constructor*, which is explained in this section.

Struct constructors have the following form:

type_name { [*field_initializer*,]... }

where each *field_initializer* has the form:

[*field_name*=]*exp*

Note how each field has an optional name *field_name* and a value *exp*. The expression for the value can be of any type. For convenience, a trailing comma is accepted but not required.

Suppose for example that we have a type defined as:

```
type Packet =
  struct
  {
    uint<16> flags;
    byte[32] data;
  }
```

We can construct a new packet, with all its fields initialized to zero, like this:

```
(poke) Packet {}
Packet {
  flags=0x0UH,
  data=[0x0UB,0x0UB,0x0UB,0x0UB,0x0UB,...]
}
```

In the constructor we can specify initial values for some of the fields:

```
(poke) Packet { flags = 0x8 }
Packet {
  flags=0x8UH,
  data=[0x0UB,0x0UB,0x0UB,0x0UB,0x0UB,...]
}
```

It is not allowed to specify initializers that are not part of the type being constructed:

```
(poke) Packet { foo = 10 }
<stdin>:1:10: error: invalid struct field ‘foo’ in constructor
Packet { foo = 10 };
         ^~~
```

As we shall see later, many struct types define constraints on the values their fields can hold. This is a way to implement data integrity. While building struct values using constructors, these constraints are taken into account. For example, given the following struct type:

```
type BPF_Reg =
  struct
```

```
  {
   uint<4> code : code < 11;
  };
```

we will get a constraint violation exception if we try to construct an `BPF_Reg`:

```
(poke) BPF_Reg { code = 20 }
unhandled constraint violation exception
```

18.5.3 Struct Comparison

The equality operator (`==`) and the inequality operator (`!=`) can be applied to struct values. Examples:

```
(poke) E_eof == E_eof
1
(Poke) Packet { payload=0x3 } != Packet { payload=0x4 }
1
```

The struct values are compares recursively.

18.5.4 Field Endianness

By default fields are accessed in IO space using the current default endianness. However, it is possible to annotate fields with an explicit endianness, like in:

```
type Foo =
  struct
  {
    little int a;
    big int b;
    int c;
  };
```

In the example above, the field `a` will be stored using little-endian, the field `b` will be stored using big-endian, and the field `c` will be stored using whatever current endianness.

The endianness annotations can be used in any kind of fields, like offsets, arrays and structs, and will impact the storage ofintegral fields defined in them.

When endianness annotations are used in struct fields which are themselves structs, they effectively change the default endianness in the contained struct. Therefore, in the following struct type:

```
type Bar =
  struct
  {
    little struct
    {
      big int a;
      int b;
    } f;
    int c;
  }
```

`a` is big endian, but `b` is little endian.

However, note that the scope of the `big` and `little` annotations is lexical. Therefore, in these types:

```
type Foo =
  struct
  {
```

```
    int a;
    int b;
  };

type Bar =
  struct
  {
    big Foo f;
    int c;
  };
```

The endianness of the `a` and `b` fields stored in `f` is the default endianness, not `big`.

18.5.5 Accessing Fields

Poke uses the usual dot-notation to provide access to struct fields. Examples:

```
(poke) var s = struct { i = 10, l = 20L }
(poke) s.i
10
(poke) s.l
20L
```

Writing to fields is achieved by having the field reference in the left side of an assignment statement:

```
(poke) s.i = 100
(poke) s.l = 200
(poke) s
struct {i=100,l=200L}
```

18.5.6 Field Constraints

It is common for struct fields to be constrained to their values to satisfy some conditions. Obvious examples are magic numbers, and specification-derived constraints.

In Poke you can specify a field's constraint using the following syntax:

field_type field_name : *expression* ;

where *expression* is an arbitrary Poke expression, that should evaluate to an integer value. The result is interpreted as a boolean. As an example, this is how the ELF magic number is checked for:

```
type Ctf_Preamble =
  struct
  {
    uint<16> ctp_magic : ctp_magic == CTF_MAGIC;
    byte ctp_version;
    byte ctp_flags;
  };
```

The constraint expression will often include the field where it is installed, but that's not mandatory.

Field constraints play an important role in mapping. On one side, a map will fail if there is some constraint that fails. On the other, they guide the mapping of unbounded arrays. See Section 18.14.5 [Mapping Arrays], page 156.

Another common usage of constraint expressions is to alter the global state after decoding a field. For example, this is how mapping an ELF header sets the current endianness, depending on the value of `ei_data`:

```
byte ei_data : ei_data == ELFDATA2LSB       \
```

```
                ? set_endian (ENDIAN_LITTLE) \
                : set_endian (BIG);
```

Note that `set_endian` always returns `1`.

18.5.7 Field Initializers

We saw that field constraints are useful to express magic numbers, which are pretty common in binary formats. Imagine a package has a marker byte at the beginning, which should always be `0xff`. We could use a constraint like in:

```
type Packet =
  struct
  {
    byte marker : marker == 0xff;
    byte length;
    byte[length] payload;
  };
```

This works well when mapping packages. The constraint is checked and a constraint violation exception is raised if the first byte of the alleged package is not `0xff`.

However, suppose we want to construct a new `Package`, with no particular contents. We would use a constructor, but unfortunately:

```
(poke) Packet { }
unhandled constraint violation exception
```

What happened? Since we didn't specify a value for the marker in the struct constructor, a default value was used. The default value for an integral type is zero, which violates the constraint associated with the field. Therefore, we would need to remember to specify the marker, every time we construct a new packet:

```
(poke) Packet { marker = 0xff }
Packet {
  marker=0xffUB,
  length=0x0UB,
  payload=[]
}
```

Unfortunately, such markers and magic numbers are not precisely very memorable. To help with this, Poke has the notion of *type field initializers*. Let's use one in our example:

```
type Packet =
  struct
  {
    byte marker = 0xff;
    byte length;
    byte[length] payload;
  };
```

Note how the syntax is different than the one used for constraints. When a field in a struct type has an initializer, the struct constructor will use the initializer expression as the initial value for the field. For example:

```
(poke) Packet {}
Packet {
  marker=0xffUB,
  length=0x0UB,
  payload=[]
}
```

It is possible to specify both a constraint and an initializer in the same field. Suppose we want to support several kinds of packets, characterized by several markers. The supported markers are however a closed set. We could do it like this:

```
type Packet =
  struct
  {
    byte marker = 0xff : marker in [0xffUB, 0xfeUB];
    byte length;
    byte[length] payload;
  };
```

Note however that struct mappers do not make use of field initializers, since the mapped IO space provides values for all the fields in the struct type. If we mapped the `Packet` type above, we would need to add also a constraint to make sure the value of `marker` is the right one. This also applies to constructing `Packet` types where an initial value is explicitly specified. We could do it by specifying both an initial value, and a constraint expression:

```
type Packet =
  struct
  {
    byte marker = 0xff : marker == 0xff;
    byte length;
    byte[length] payload;
  };
```

This idiom is to common that Poke provides a more compact syntax to denote it, that avoids verbosity and replicated logic:

```
type Packet =
  struct
  {
    byte marker == 0xff;
    byte length;
    byte[length] payload;
  };
```

It is considered good practice to design struct types in a way that a constructor with no arguments will result in something usable.

18.5.8 Field Labels

In structs, each field is associated with an offset, which is relative to the beginning of the struct. This is the offset used when reading and writing the field from/to the IO space when mapping.

The offset is reset to zero bits at the beginning of the struct type, and it is increased by the size of the fields:

```
struct
{
                /* Offset: 0#b */
    uint<1> f1; /* Offset: 1#b */
    sint<7> f2; /* Offset: 1#B */
    int     f3; /* Offset: 5#B */
    Bar     f4; /* Offset: 23#B */
}
```

It is possible to specify an alternative offset for a field using a *field label*.

Consider for example an entry in an ELF symbol table. Each entry has a `st_info` field which is 64-bits long, that in turn can be interpreted as two fields `st_bind` and `st_type`.

The obvious solution is to encode `st_info` as a sub-struct that is integral, like this:

```
struct
{
  elf32_word st_name;
  struct uint<64>
  {
    uint<60> st_bind;
    uint<4> st_type;
  } st_info;
}
```

This makes the value of `st_info` easily accessible as an integral value. But we may prefer to use labels instead:

```
struct
{
  elf32_word st_name;
  elf64_word st_info;
  uint<60> st_bind @ 4#B;
  uint<4> st_type @ 4#B + 60#b;
}
```

The resulting struct has fields `st_info`, `st_bind` and `st_type`, with the last two overlapping the first.

18.5.9 Pinned Structs

Pinned structs is a convenient way to write struct types where the offset of all its fields is zero. They are equivalent to C unions.

For example, consider the `_u` field below in a CTF type description:

```
type Ctf_Stype_V1 =
  struct
  {
    Ctf_Name ctt_name;
    Ctf_Info_V1 ctt_info;
    pinned struct
    {
      uint32 _size; /* Size of entire type in bytes.  */
      uint32 _type; /* Reference to another type.  */
    } _u;
  };
```

Note that integral structs cannot be pinned. See Section 18.5.11 [Integral Structs], page 136. And field labels are not allowed in pinned structs.

18.5.10 The OFFSET variable

The offset of the current field being mapped or constructed, relative to the beginning of the struct, is at all times available in a variable called `OFFSET`. This is useful in situations where we need that information in a constraint expression, the boundary of an array, an initialization value, or the like.

For example, what follows is a typical usage of OFFSET in order to introduce some padding in a struct:

```
type Elf64_Note =
  struct
  {
```

```
    [...]

    Elf64_Xword name;
    byte[alignto (OFFSET, 4#B)] padding;

    [...]
  };
```

The value of this variable changes every time a new field is mapped or constructed, so writing to it will only have effect until the next field is processed: at that time the value will be overwritten.

18.5.11 Integral Structs

Integral structs are useful to cover cases where data is stored in composited integral containers, *i.e.* where data is structured within stored integers.

Basically, when we structure data using Poke structs, arrays and the like, we often use the same structure than a C programmer would use. For example, to model ELF RELA structures, which are defined in C like:

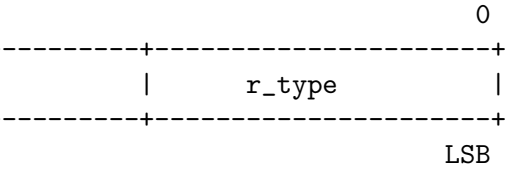
```
type struct
  {
    Elf64_Addr   r_offset;  /* Address */
    Elf64_Xword  r_info;    /* Relocation type and symbol index */
    Elf64_Sxword r_addend;  /* Addend */
  } Elf64_Rela;
```

we could use something like this in Poke:

```
type Elf64_Rela =
  struct
  {
    Elf64_Addr r_offset;
    Elf64_Xword r_info;
    Elf64_Sxword r_addend;
  };
```

Here the Poke struct type is pretty equivalent to the C incarnation. In both cases the fields are always stored in the given order, regardless of endianness or any other consideration.

However, there are situations where stored integral values are to be interpreted as composite data. This is the case of the `r_info` field above, which is a 64-bit unsigned integer (`Elf64_Xword`) which is itself composed by several fields, depicted here:



In order to support this kind of composition of integers, C programmers usually resort to either bit masking (most often) or to the often obscure and undefined behaviour-prone C bit fields. In the case of ELF, the GNU implementations define a few macros to access these *sub-fields*:

```
#define ELF64_R_SYM(i)          ((i) >> 32)
#define ELF64_R_TYPE(i)         ((i) & 0xffffffff)
#define ELF64_R_INFO(sym,type) ((((Elf64_Xword) (sym)) << 32) + (type))
```

Where `ELF64_R_SYM` and `ELF64_R_TYPE` are used to extract the fields from an `r_info`, and `ELF64_R_INFO` is used to compose it. This is typical of C data structures.

We could of course mimic the C implementation in Poke:

```
fun Elf64_R_Sym = (Elf64_Xword i) uint<32>:
   { return i .>> 32; }
fun Elf64_R_Type = (Elf64_Xword i) uint<32>:
   { return i & 0xffff_ffff; }
fun Elf64_R_Info = (uint<32> sym, uint<32> type) Elf64_Xword:
   { return sym as Elf64_Xword <<. 32 + type; }
```

However, this approach has a huge disadvantage: since we are not able to encode the logic of these "sub-fields" in proper Poke fields, they become second class citizens, with all that implies: no constraints on their own, can't be auto-completed, can't be assigned individually, *etc.*

This is where *integral structs* come to play. These are structs that are defined exactly like your garden variety Poke structs, with a small addition:

```
type Elf64_RelInfo =
  struct uint<64>
  {
    uint<32> r_sym;
    uint<32> r_type;
  };
```

Note the `uint<64>` addition after `struct`. This can be any integer type (signed or unsigned). The fields of an integral struct should be integral themselves (this includes both integers and offsets) and the total size occupied by the fields should be the same size than the one declared in the struct's integer type. This is checked and enforced by the compiler.

The Elf64 RELA in Poke can then be encoded like:

```
type Elf64_Rela =
  struct
  {
    Elf64_Addr r_offset;
    struct Elf64_Xword
    {
      uint<32> r_sym;
      uint<32> r_type;
    } r_info;
    Elf64_Sxword r_addend;
  };
```

When an integral struct is mapped from some IO space, the total number of bytes occupied by the struct is read as a single integer value, and then the values of the fields are extracted from it. A similar process is using when writing. That is what makes it different with respect a normal Poke struct.

Consider for example we have the following sequence of bytes in our IO space (like a file):

```
0x10 0x20 0x30 0x40  0x50 0x60 0x70 0x80
```

Let's see what happens when we map the integral struct above, in both big and little endian:

```
(poke) .set endian big
(poke) Elf64_RelInfo @ 0#B
Elf64_RelInfo {
  r_sym=0x10203040U,
  r_type=0x50607080U
}
(poke) .set endian little
(poke) Elf64_RelInfo @ 0#B
```

```
Elf64_RelInfo {
  r_sym=0x80706050U,
  r_type=0x40302010U
}
```

Looks good. For comparison, this is what happens when we do the same with an "equivalent" (not really) non-integral struct operating on the same data:

```
type Elf64_RelInfoBogus =
  struct
  {
    uint<32> r_sym;
    uint<32> r_type;
  };
```

We would get:

```
(poke) .set endian big
(poke) Elf64_RelInfoBogus @ 0#B
Elf64_RelInfoBogus {
  r_sym=0x10203040U,
  r_type=0x50607080U
}
(poke) .set endian little
(poke) Elf64_RelInfoBogus  0#B
Elf64_RelInfoBogus {
  r_sym=0x40302010U,
  r_type=0x80706050U
}
```

In this case, and unlike with integral structs, the endianness impacts the bytes of the individual fields, not of the whole struct.

As you can see, integral structs can be used to denote a lot of commonly found idioms in data structures and this includes a lot of what is sometimes denoted in C bit field. However, one should be cautious when "translating" C structures to Poke, especially when the C programmer has not been careful and incurres in sometimes obscure implementation-defined behavior. An integral struct is not always the right abstraction to use when we see a C bit field!

As an example of the above, consider the following C struct:

```
struct regs
{
 __u8 dst_reg:4;
 __u8 src_reg:4;
};
```

Certain virtual architecture uses that data layout to store registers in instructions (no comment.) Thing is, in bit fields like the above with sub-byte field sizes, the ordering of the fields is not clearly defined, and ultimately what order to use is up to the compiler, *i.e.* to lore and tradition. As it happens, GCC encodes `src_reg` in the most significant nibble of the byte and `dst_reg` in the least significant nibble of the byte when compiling for a little-endian target, and the other way around when compiling for a little-endian target. (I may have had that wrong, this always confuses me.)

How could we encode the C struct regs in Poke? Let's see.

A normal Poke struct clearly won't do it:

```
type RegsBogus1 =
  struct
```

```
{
  uint<4> src;
  uint<4> dst;
};
```

The reason being, the ordering of src and dst does not change when you switch endianness (since this is Poke, we can in fact talk about real ordering of bits)... remember, poke is WYPIWIG (what you poke is what you get) ;)

What about an integral struct?

```
type RegsBogus2 =
  struct uint<8>
 {
    uint<4> src;
    uint<4> dst;
 };
```

This won't work either. In fact, the net effect of the normal decoding of the normal struct type RegsBogus1 and the map-an-integer-and-extract-fields decoding of the integral struct RegsBogus2 is in this case totally equivalent.

A solution is to use a normal struct, and field labels:

```
type RegsBogus =
  struct
  {
    var little_p = (get_endian == ENDIAN_LITTLE);

    uint<8> src @ !little_p * 4#b;
    uint<8> dst @ little_p * 4#b;
  };
```

At this point, you may be wondering: is there anything particular in a field defined in an integral struct? The answer is: no, not at all. These are regular, first-class fields. Likewise, integral structs are perfectly regular structs. And of course, since this is poke, you can have integral structs of say, 11 bits, or 3 bits, map them at offsets not aligned to bytes, and all the typical poke-atrocities that we enjoy so much.

However, there exist a few restrictions, some of them fundamental, the others to be lifted eventually:

- There are no integral unions. This is a fundamental limitation and will most likely stay like that.
- Integral structs can only have integral fields. This includes offsets.
- No labels are allowed in the fields of integral structs. This is not a fundamental limitation, and may be supported at some point.
- No integral structs are supported inside other integral structs. This is purely because of lazyness on my part. This will be eventually supported.
- No optional fields are supported in integral structs. Support for this is actually partially implemented (the mapper supports them but not the writer) and most probably will be completed one of these days.

Integral structs can be converted from/to integral structs to/from integers, so we can do things like:

```
rel.r_info as uint<64>;
```

And also automatic promotions in arithmetic operators, like:

```
rel.r_info + 20 * rel.r_info.r_type
```

Integers can also be "structured" into integral structs:

```
(poke) 0xdeadbeef as Elf32_RelInfo
Elf32_RelInfo {
  r_sym=(uint<24>) 0xdeadbe,
  r_type=0xefUB
}
```

18.5.12 Unions

Union types are defined much like struct types, using a very similar syntax:

```
union
{
  [... elements ...]
}
```

Like in structs, the main kind of elements are fields. However, unlike in structs, the fields in an union type denote *alternatives*. At any given time, only one alternative is selected, and therefore union *values* have only one field.

Which of the several alternatives is selected is determined by the data integrity defined by means of constraint expressions and other means. Alternatives are considered in turn, in written order, and the first alternative that can be used without triggering a constraint violation exception is selected.

It makes little sense to mark union types as pinned, since whatever alternative is chosen will always be located at the beginning of the union. To avoid unneccesary confusion, the Poke compiler will emit an error if you try to tag an union as pinned.

Similarly, labels are not allowed in union alternatives.

18.5.13 Union Constructors

Constructing union values is very similar to constructing struct values:

type_name { [*field_initializer*] }

The main difference is that union constructors require either exactly one field initializer, or none. Specifying initialization values for more than one field doesn't make sense in the case of unions.

Suppose we have the following union type:

```
type Packet =
  union
  {
    byte b : b > 0;
    int k;
  };
```

If we try to construct a `Packet` with an invalid `b`, we get a constraint error:

```
(poke) Packet {b = 0}
unhandled constraint violation exception
```

The reason why `k` is not chosen is because we specified `b` in the union constructor: we explicitly as for a `Packet` of that kind, but the data we provide doesn't accommodate to that. If we wanted another kind of value, we could of course specify an initial value for `k` instead:

```
(poke) Packet {k = 10}
Packet {
  k=10
}
```

If no field initializer is specified in an union constructor then each alternative is tried assuming all fields have default values, which in the case of integral types is a zero:

```
(poke) Packet {}
Packet {
  k=0
}
```

Therefore the alternative `b` is considered invalid (it has to be bigger than 0) and `k` is chosen instead.

18.5.14 Optional Fields

Sometimes a field in a struct is optional, *i.e.* it exists or not depending on some criteria. A good example of this is the "extended header" in a MP3 id3v2 tag. From the specification:

```
The second bit (bit 6) indicates whether or not the header is followed
by an extended header. The extended header is described in section
3.2. A set bit indicates the presence of an extended header.
```

In order to express this in a Poke struct type, we could of course use an union, like:

```
type ID3V2_Tag =
  struct
  {
    ID3V2_Hdr hdr;
    union
    {
      ID3V2_Ext_Hdr hdr if hdr.extended_hdr_present;
      struct {};
    } ext;
    ...
  }
```

That's it, we use an union with an empty alternative. However, this is a bit cumbersome. Therefore Poke provides a more convenient way to denote this:

```
type ID3V2_Tag =
  struct
  {
     ID3V2_Hdr hdr;
     IDV2_Ext_Hdr ext_hdr if hdr.extended_hdr_present;
     ...
  }
```

If both a constraint and an optional field expression are specified, the second should follow the first.

One important characteristic of optional fields to keep in mind is that they are always constructed or mapped, even when they are omitted. This is because, generally speaking, the value of an optional field may be necessary in order to determine whether the optional field is present or not.

This may result on unexpected results for the unaware pokist, like in the example below:

```
struct
{
  Header hdr;
  byte[hdr.len - hdr.padding] data if hdr.len >= hdr.padding;
}
```

where the array index may underflow if `hdr.len` is less or equal than `hdr.padding`.

For obvious reasons, optional fields are not allowed in unions.

18.5.15 Casting Structs

It is possible to cast a struct of some particular type into another struct type. Examples:

```
(poke) type Foo = struct { int i; int j; };
(poke) type Bar = struct { int k; int j; };
(poke) Bar {j=2} as Foo {}
Foo {i=0,j=2}
```

The semantics of the cast are exactly the same than constructing a struct of the target type using the struct provided as an expression to the cast.

This means that some fields in the original struct may be "lost" when the struct is converted to the new type. Consider the following two definitions:

```
type Foo = struct { int i; long j; };
type Bar = struct { int i; }
```

Let's cast some values and see what we obtain:

```
(poke) Bar {}
Bar {
  i=0x0
}
(poke) Bar {} as Foo
Foo {
  i=0x0,
  j=0x0L
}
(poke) Foo {} as Bar
Bar {
  i=0x0
}
```

It is possible to cast an integral struct into an integral type. The value to cast is extracted from the struct fields, and then it is converted to the target type.

Given the following integral struct type:

```
type Foo =
  struct int<64>
  {
    int<32> f1;
    uint<32> f2;
  };
```

We can operate:

```
(poke) Foo { f1 = 1, f2 = 1 } as uint<64> + 2
0x100000003UL
```

Conversely, it is possible to cast an integral value to an integral struct:

```
(poke) 0x100000003UL as Foo
Foo {
  f1=0x1,
  f2=0x3U
}
```

18.5.16 Declarations in Structs

Type declarations, variable declarations and function declarations are all allowed inside struct and union types. The scope of the entity being declared spans from the declaration until the end of the struct or union type being defined.

Functions can't set fields defined in the struct type.

18.5.17 Methods

Methods in structs can be defined using the `method` keyword in definitions like this:

```
method identifier = function_body
```

where *function_body* is the body of a function. See Section 18.12.1 [Function Declarations], page 147.

When invoked, a method gets an implicit *last* argument which is the struct value whose method is being used.

Methods can refer to the fields of the containing struct type that have been defined before the method itself, by name.

18.5.18 Struct Attributes

The following attributes are defined for struct values.

`size` Gives an offset with the storage occupied by the complete struct. Example:

```
(poke) type Packet = struct { byte s; byte[s] d; }
(poke) (Packet { s = 3 })'size
32UL#b
```

`length` Gives the number of fields stored in the struct. Note that, due to absent fields, this doesn't always corresponds to the number of fields that appear in the definition of a struct type.

For example, for the following struct type:

```
type Packet =
  struct
  {
    byte magic: magic in [0xff,0xfe];
    byte control if magic == 0xfe;
    byte[128] payload;
  };
```

We get the following results:

```
(poke) (Packet { magic = 0xff })'length
2UL
(poke) (Packet { magic = 0xfe })'length
3UL
```

`mapped` Boolean indicating whether the struct is mapped.

`strict` Boolean indicating whether the struct is mapped in strict mode.

18.6 Types

18.6.1 `type`

The `type` directive allows you to declare named types. The syntax is:

```
type name = type [, name = type]...;
```

where *name* is the name of the new type, and *type* is either a type specifier or the name of some other type.

The supported type specifiers are:

`int<n>, uint<n>`
: Integral types. See Section 18.1.4 [Integer Types], page 115.

`string` The string type. See Section 18.3.2 [String Types], page 122.

type`[`*boundary*`]`
Array types. See Section 18.4.2 [Array Types], page 124.

`struct { ... }`
Struct types. See Section 18.5.1 [Struct Types], page 129.

`(`*type*`,...)`*type*`:`
Function types. See Section 18.12.5 [Function Types], page 148.

`any` The `any` type is used to implement polymorphism. See Section 18.6.2 [The any Type], page 144.

18.6.2 The `any` Type

Poke supports polymorphism with the `any` type. This type is used in contexts where a value of any type is allowed. For example, this is how you would declare a function that prints the size of any given value:

```
fun print_size = (any value) void:
{
  printf "%v\n", any'size;
}
```

The rules for handling `any` values are simple:

– Everything coerces to `any`.
– Nothing coerces from `any`.

This means that using any operator that require certain types with `any` values will fail: you have to cast them first. Example:

```
(poke) fun foo = (any v) int: { return v as int; }
```

Arrays of `any`, `any[]`, are also supported:

18.6.3 The `isa` Operator

The binary operator `isa` allows you to check for the type of a given value:

```
(poke) 10 isa int
1
(poke) "foo" isa string
1
(poke) (Packet  0#B) isa Packet
1
(poke) 2 as int<3> isa uint<4>
0
```

18.7 Assignments

The assignment statement has the form:

```
lvalue = exp;
```

where *lvalue* is either:

- A variable.
- A field reference like `foo.bar`.
- An index reference like `foo[30]`.
- A map of a type, like `int @ 0#B` or `int[3] @ 0#B` or `Packet @ 12#B`.

In all cases, the type of *exp* should match the type of the referred entity.

Examples:

```
(poke) foo = 10
(poke) packet.length = 4
(poke) packet.data = [1,2,3,4]
(poke) packet.data[2] = 666
(poke) int @ 23#B = 23
(poke) string @ str.offset = "foo"
(poke) int[2] @ 23#B = [1,2]
(poke) Packet @ 0#B = Packet { ... }
```

18.8 Compound Statements

Compound statements have the form

```
{ stmt... }
```

where *stmt...* is a list of statements, which can be themselves compound statements. Compound statements are primarily used to sequence instructions like:

```
{
  do_a;
  do_b;
  do_c;
}
```

A compound statement introduces a new lexical scope. Declarations in the compound statements are local to that statement.

Finally, compound statements can be empty: `{ }`.

18.9 Conditionals

Poke provides several conditional statements, and a ternary conditional operator, which are discussed in the sections below.

18.9.1 `if-else`

The `if-else` statement has the form

```
if (exp) if_stmt [else else_stmt]
```

where *exp* is an expression that should evaluate to a boolean value (*i.e.* to an integer), *if_stmt* is a statement that will be executed if *exp* holds true, and *else_stmt* is a statement that will be executed if *exp* holds false. The `else` part of the statement is optional.

18.9.2 Conditional Expressions

Poke supports a ternary conditional expression that has the form:

```
condition ? true_expression : false_expression
```

where *condition* is an expression that should evaluate to a boolean, and *true_expression* and *false_expression* are expressions that have exactly the same type.

18.10 Loops

Poke supports several iteration statements, which are discussed in the sections below.

18.10.1 `while`

The `while` statement has this form:

```
while (exp) stmt
```

where *exp* is an expression that should evaluate to a boolean (*i.e.* to an integer) and *stmt* is an statement that will be executed until *exp* holds false.

It is possible to leave the loop from within *stmt* using the `break` statement. Example:

```
while (1)
{
  [...]
  if (exit_loop)
    break;
}
```

It is also possible to jump to the next iteration of the loop from within *stmt* using the `continue` statement. Example:

```
while (1)
{
   [...]
   if (continue_loop)
     continue;
}
```

18.10.2 `for`

The `for` statement has this form:

```
for ([decls];[expr];[stmts]) stmt
```

where *decls* is an optional declaration or chain of declarations separated by commas, *expr* is an optional boolean expression, *stmts* is an optional comma-separated list of statements, and *stmt* is a statement.

18.10.3 `for-in`

The `for-in` statement has this form:

```
for (formal in container [where exp]) stmt
```

where, in each iteration, the name *formal* will be associated with consecutive values of *container*, which shall be an expression evaluating to an array or a string. *formal* is available in *stmt*, which is the statement executed in each iteration.

If the *where* part is specified, then only iteration in which *exp* holds true are processed. *formal* can be referred in *exp*. Note that this doesn't mean the loop will stop after processing the first "not selected" element. See the following example:

```
(poke) for (c in [1,2,3,4] where c % 2) printf " %v", c
 1 3
```

It is possible to leave the loop from within *stmt* using the `break` statement.

It is also possible to jump to the next iteration of the loop from within *stmt* using the `continue` statement.

18.11 Expression Statements

Poke is one of these languages where there is a clear separation between *expressions* and *statements*. However, it is often useful to use an expression in the place of an statement, in order to benefit from its side effects.

For that purpose Poke allows you to expressions as statements using the following syntax:

```
exp;
```

The value computed by the expression will be discarded.

18.12 Functions

18.12.1 Function Declarations

A function is declared using the following syntax:

```
fun name = [(formal,...)] ret_type:
{
   ... body ...
}
```

where *name* is the name of the function, which uses the same namespace as variables and types and *ret_type* is the type of the value returned by the function. If the function returns no value then it is `void`.

Each *formal* argument has the form:

```
type name [= exp]
```

where *type* is the type of the formal, *name* its name, and *exp* is an optional expression that will be used to initialize the argument in case it is not specified when the function is called.

The last formal argument can take the form *`name...`*, meaning the function is variadic. See Section 18.12.3 [Variadic Functions], page 148.

If the function takes no arguments, it is possible to omit the list of arguments entirely:

```
fun hello = void: { print "Hello!\n"; }
```

The `return` statement is used to return values in functions that return a value. Example:

```
fun gcd = (uint<64> a, uint<64> b) uint<64>:
  {
   if (b == 0)
     return a;
   else
     return gcd (b, a % b);
  }
```

Note that reaching the end of a non-void function will trigger a run-time error.

18.12.2 Optional Arguments

Optional function arguments are specified like:

```
fun atoi = (string s, int b = 10) long: { ... }
```

Which means that if the base argument is not specified when passed to `atoi` then it is initialized to 10.

Optional arguments should not appear before any non-optional argument in function declarations. The following is not valid Poke:

```
fun foo = (int i = 10, int j) int: { return i + j; }
```

Note that arguments declared before an optional argument can be used in its initialization expression. This is valid Poke:

```
fun foo = (int n, int[n] array = init_array (n)) void: { ... }
```

18.12.3 Variadic Functions

Functions getting an arbitrary number of arguments are denoted like this:

```
fun printf (string fmt, args...) void: { ... }
```

The variadic argument shall be the last argument in the function, and it is of type `any[]`.

18.12.4 Calling Functions

To call a function, write its name followed by the arguments in parentheses. Examples:

```
foo (1,2,3)
bar ()
```

If the function takes no arguments then it is not necessary to write the empty list of arguments. Therefore the following two calls are equivalent:

```
bar ()
bar
```

If the closure of function that takes no arguments is needed, the call can be avoided by putting the variable immediately inside parenthesis, like this:

```
(poke) (foo)
#<closure>
```

There is an alternate syntax that can be used in both expressions and expression-statements. This alternate syntax is:

```
function_name :arg1 val1...
```

where *arg1* is the name of an argument and *val1* the value to pass for that argument. This is useful when using functions as commands in the REPL:

```
(poke) dump :from 12#B :size 16#B :ascii 0
```

Note that the named arguments can appear in any order. The following two calls are equivalent:

```
dump :from 12#B :size 16#B
dump :size 16#B :from 12#B
```

If this alternate syntax for function calls is to be used in an expression, it is necessary to surround it with parenthesis:

```
2 + (gcd :a 1024 :b 8)
```

18.12.5 Function Types

Function types are denoted like:

```
(type,...)ret_type:
```

where *type* are the types of the arguments and *ret_type* is the type of the value returned by the function.

Optional arguments are marked with a `?` after the type. For example, the type of the `atoi` function with declaration:

```
fun atoi = (string s, int b = 10) long: { ... }
```

is `(string,int?)long:`.

If the function has variadic arguments, the position of the variadic argument in the function type specifier contains `...`. For example, the type of a `printf` function with declaration:

```
fun printf (string fmt, args...) void: { ... }
```

is `(string,...)void:`.

18.12.6 Lambdas

Poke support *lambda expressions* using the following syntax:

```
lambda function_specifier
```

Where *function_specifier* is any function specifier. Examples:

```
lambda void: {}
lambda (int i) int: { return i * 2; }
```

Lambdas can be manipulated exactly like any other Poke value, and can be stored in variables. Therefore, this is an alternative way of defining a named function (which can't be recursive for obvious reasons):

```
var double = lambda (int i) int: { return i * 2 };
```

18.12.7 Function Comparison

The equality operator (`==`) and the inequality operator (`!=`) can be applied to functions. They evaluate to true if the operands are the same function value. Examples:

```
(poke) lambda void: {} == lambda void: {}
0
(poke) lambda void: {} != lambda void: {}
1
(poke) var f = void: {}
(poke) f == f
1
(poke) f != f
0
```

18.12.8 Function Attributes

The following attributes are defined for function values.

`size` Gives an offset `0#B`, by convention.

18.13 Endianness

Byte endianness is an important aspect of encoding data. As a good binary editor poke provides support for both little and big endian, and will soon acquire the ability to encode exotic endianness like PDP endian. Endianness control is integrated in the Poke language, and is designed to be easily used in type descriptions.

18.13.1 `.set endian`

GNU poke maintains a global variable that holds the current endianness. This is the endianness that will be used when mapping integers whose types do not specify an explicit endianness.

Like other poke global state, this global variable can be modified using the `.set` dot-command:

```
.set endian little
.set endian big
.set endian host
```

The current endianness can be obtained like this:

```
(poke) .set endian
little
```

We can easily see how changing the current endianness indeed impacts the way integers are mapped:

```
(poke) dump :from 0#B :size 4#B :ruler 0 :ascii 0
```

```
00000000: 8845 4c46
(poke) .set endian little
(poke) int @ 0#B
0x464c4588
(poke) .set endian big
(poke) int @ 0#B
0x88454c46
```

18.13.2 Endian in Fields

It is possible to set the endianness of integral fields in struct type descriptors. See Section 18.5.4 [Field Endianness], page 131.

18.13.3 Endian built-ins

As handy as the `.set endian` dot-command may be, it is also important to be able to change the current endianness programmatically from a Poke program. For that purpose, the PKL compiler provides a couple of built-in functions: `get_endian` and `set_endian`.

Their definitions, along with the specific supported values, look like:

```
var ENDIAN_LITTLE = 0;
var ENDIAN_BIG = 1;

fun get_endian = int: { ... }
fun set_endian = (int endian) int: { ... }
```

Accessing the current endianness programmatically is especially useful in situations where the data being poked features a different structure, depending on the endianness.

A good (or bad) example of this is the way registers are encoded in eBPF instructions. eBPF is the in-kernel virtual machine of Linux, and features an ISA with ten general-purpose registers. eBPF instructions generally use two registers, namely the source register and the destination register. Each register is encoded using 4 bits, and the fields encoding registers are consecutive in the instructions.

Typical. However, for reasons we won't be discussing here the order of the source and destination register fields is switched depending on the endianness.

In big-endian systems the order is:

```
dst:4 src:4
```

Whereas in little-endian systems the order is:

```
src:4 dst:4
```

In Poke, the obvious way of representing data whose structure depends on some condition is using an union. In this case, it could read like this:

```
type BPF_Insn_Regs =
  union
  {
    struct
    {
      BPF_Reg src;
      BPF_Reg dst;
    } le : get_endian == ENDIAN_LITTLE;

    struct
    {
      BPF_Reg dst;
```

```
      BPF_Reg src;
    } be;
  };
```

Note the call to the `get_endian` function (which takes no arguments and thus can be called Algol68-style, without specifying an empty argument list) in the constraint of the union alternative. This way, the register fields will have the right order corresponding to the current endianness.

Nifty. However, there is an ever better way to denote the structure of these fields. This is it:

```
type BPF_Insn_Regs =
  struct
  {
    var little_p = (get_endian == ENDIAN_LITTLE);

    BPF_Reg src @ !little_p * 4#b;
    BPF_Reg dst @ little_p * 4#b;
  };
```

This version, where the ordering of the fields is implemented using field labels, is not only more compact, but also has the virtue of not requiring additional "intermediate" fields like `le` and `be` above. It also shows how convenient can be to declare variables inside structs.

Let's see it in action:

```
(poke) BPF_Insn_Regs @ 1#B
BPF_Insn_Regs {src=#<%r4>,dst=#<%r5>}
(poke) .set endian big
(poke) BPF_Insn_Regs @ 1#B
BPF_Insn_Regs {src=#<%r5>,dst=#<%r4>}
```

Changing the current endianness in constraint expressions is useful when dealing with binary formats that specify the endianness of the data that follows using some sort of tag. This is the case of ELF, for example. The first few bytes in an ELF header conform what is known as the `e_ident`. One of these bytes is called `ei_data` and its value specifies the endianness of the data stored in the ELF file.

This is how we handle this in Poke:

```
fun elf_endian = (int endian) byte:
 {
   if (endian == ENDIAN_LITTLE)
     return ELFDATA2LSB;
   else
     return ELFDAT2MSB;
 }

[...]

type Elf64_Ehdr =
  struct
  {
    struct
    {
      byte[4] ei_mag : ei_mag[0] == 0x7fUB
                       && ei_mag[1] == 'E'
                       && ei_mag[2] == 'L'
                       && ei_mag[3] == 'F';
```

```
      byte ei_class;
      byte ei_data : (ei_data != ELFDATANONE
                      && set_endian (elf_endian (ei_data)));
      byte ei_version;
      byte ei_osabi;
      byte ei_abiversion;
      byte[6] ei_pad;
      offset<byte,B> ei_nident;
    } e_ident;

    [...]
};
```

Note how `set_endian` returns an integer value... it is always `1`. This is to facilitate its usage in field constraint expressions.

18.14 Mapping

The purpose of poke is to edit *IO spaces*, which are the files or devices, or memory areas being edited. This is achieved by mapping values. Mapping is perhaps the most important concept in Poke.

18.14.1 IO Spaces

GNU poke supports the abstract notion of *IO space*, which is an addressable space of Poke objects: integers, strings, arrays, structs, *etc.* This underlying storage for the IO spaces (which we call *IO devices*) can be heterogeneous: from a file your file system to the memory of some process.

18.14.1.1 `open`

The `open` builtin allows you to create new IO spaces, by opening an IO device. It has the following prototype:

```
fun open = (string handler, uint<64> flags = 0) int<32>
```

where *handler* is a string identifying the IO device that will serve the IO space. This handler can be:

`*name*` An auto growing memory buffer.

`pid://[0-9]+`
: The process ID of some process.

`/path/to/file`
: An either absolute or relative path to a file.

`nbd://host:port/export`
`nbd+unix:///export?socket=/path/to/socket`
: A connection to an NBD server. See Section 16.5 [nbd command], page 105,

flags is a bitmask that specifies several aspects of the operation, including the mode in which the IO space is opened. Its value is usually built by ORing a set of flags that are provided by the compiler. These are:

`IOS_F_READ`
: The IO space is intended to be read.

`IOS_F_WRITE`
: The IO space is intended to be written.

`IOS_F_CREATE`
: If the IO device doesn't exist, then create it, usually empty.

Note that the specific behaviour of these flags depend on the nature of the IO space that is opened. For example a memory buffer is always truncated at creation by default.

In order to ease the usage of `open`, a few pre-made bitmaps are provided to specify opening *modes*:

`IOS_M_RDONLY`
: This is equivalent to `IOS_F_READ`.

`IOS_M_WRONLY`
: This is equivalent to `IOS_F_WRITE`.

`IOS_M_RDWR`
: This is equivalent to `IOS_F_READ | IOS_F_WRITE`.

The `open` builtin returns a signed 32-bit integer. This number will identify the just opened IOS until it gets closed.

If there is a problem opening the specified IO device then `open` will raise an `E_no_ios` exception.

18.14.1.2 `opensub`

The `opensub` standard function allows you to create IO spaces that show a sub-region of some other IO space. The prototype is:

```
fun opensub = (int<32> ios,
               offset<uint<64>,B> base, offset<uint<64>,B> size,
               string name = "",
               uint<64> flags = 0) int<32>
```

where *ios* is the ID of the base IOS, *base* is the offset in *ios* where the sub-range starts and *size* is the size of the range.

The argument *name* is a descriptive name of the contents of the range, and it is empty by default.

The argument *flags* is an ORed value of `IOS_F_*` flags. If no explicit *flags* are specified then the sub space inherits the flags of the underlying base IOS. If explicit *flags* are provided and they contradict the flags of the underlying IOS then `E_ios_flags` is raised.

Trying to access a sub space whose base IOS has been closed results in a `E_io` exception.

18.14.1.3 `openproc`

The `openproc` standard function allows you to create IO spaces to access the memory of some running program, given its process ID. The prototype is:

```
fun openproc = (uint<64> pid, uint<64> flags = 0) int<32>
```

where *pid* is the process ID whose memory we want to poke and *flags* is a set of open flags.

18.14.1.4 `close`

The `close` builtin allows you to destroy IO spaces, closing the underlying IO device. The prototype is:

```
fun close = (int<32> ios) void
```

where *ios* is some previously created IO space. All pending data is written to the underlying IO device.

If the IO space specified to `close` doesn't exist then an `E_no_ios` exception is raised.

If errors occured while closing the IO space, then `E_ios` exception is raised.

18.14.1.5 `flush`

The builtin `flush` performs a "flushing" operation on a given IO space. The prototype is:

```
fun flush = (int<32> ios, offset<uint<64>,1> offset) void
```

Where *ios* is the IOS identifier where to perform the flush. The semantics associated with the "flushing" operation depends on the kind of IO space:

- Read-only stream IOS will discard already mapped input up to *offset*. Any further attempt of mapping data at that area will cause an *E_eof* exception (for Early Of File ;)).
- Flushing is a no-operation for other kind of IO spaces.

18.14.1.6 `get_ios`

GNU poke maintains a *current IO space*, which is the last created IO space (this includes IO spaces opened and selected using a dot-command). The builtin `get_ios` returns this space. It has the following prototype:

```
fun get_ios = int<32>
```

If there is no IO space, `get_ios` will raise the `E_no_ios` exception.

18.14.1.7 `set_ios`

The `set_ios` builtin allows you to set a specific IO space as the new current IO space. It has the following prototype:

```
fun set_ios = (int<32> ios) int<32>
```

where *ios* is the IO space that will become the current IO space. If the IO space specified to `set_ios` doesn't exist, `E_no_ios` will be raised.

Note that `set_ios` always returns `1`. This is to ease its usage in struct fields constraint expressions.

18.14.1.8 `iosize`

The `iosize` builtin returns the size of a given IO space, as an offset. It has the following prototype:

```
fun iosize = (int<32> ios = get_ios) offset<uint<64>,1>
```

If the IO space specified to `iosize` doesn't exist, `E_no_ios` will be raised.

18.14.1.9 `ioflags`

The `ioflags` builtin returns the flags active in a given IO space, encoded in an unsigned 64-bit integer. It has the following prototype:

```
fun ioflags = (int<32> ios = get_ios) uint<64>
```

If the IO space specified to `ioflags` doesn't exist, `E_no_ios` will be raised.

18.14.2 The Map Operator

Poke values reside in memory, and their in-memory representation is not visible from Poke programs. For example, `32` is a 32-bit signed integer value, and it happens to not be boxed in the Poke Virtual Machine. Therefore, it occupies exactly 32-bit in the memory of the machine running poke. Other values, like arrays for example, are boxed, and they need to store various meta-data.

Regardless of the internal representation, we say these values live "in memory". Now, it is also possible to "map" a value to some area in some underlying IO space. This is done with the map operator `@`, which has two alternate syntax:

```
type @ offset
```

```
type @ ios : offset
```

The ternary version creates a new value using the data located at the offset *offset* in the specified IO space *ios*, which shall be an expression evaluating to a signed 32-bit integer.

The binary version uses the current IO space.

If there is no IO space, or the specified IO space doesn't exist, an `E_no_ios` exception is raised:

```
(poke) int @ 0#B
unhandled no IOS exception
```

The value created in a map can be either mapped or not mapped. Mapping simple types produces not mapped values, whereas mapping non-simple types create mapped values.

The value attributes `mapped` and `offset` can be used to check whether a value is mapped or not, and in that case the offset where it is mapped:

```
(poke) var p = Packet @ 0#B
(poke) p'mapped
0x1
(poke) p'offset
0x0UL#b
```

Using the `offset` attribute in a not mapped value results in the `E_no_map` exception being raised:

```
(poke) [1,2,3]'mapped
0x0
(poke) [1,2,3]'offset
unhandled no map exception
```

If the type specified in the map is not a simple type, like an array or a struct, the resulting value is said to be mapped in the IO space:

```
(poke) type Packet = struct { int i; long l; }
(poke) Packet @ 0#B
Packet {i=0x464c457f,l=0x10102L}
(poke) uint<8>[2] @ 0#B
[0x7fUB,0x45UB]
```

A very important idea on Poke mapping is that it should be possible to manipulate mapped and non-mapped values in a transparent way. For example, consider the quick sort implementation in poke's standard library. The prototype is:

```
fun qsort = (any[] array, Comparator cmp_f,
             long left = 0, long right = array'length - 1) void
```

`qsort` works with both mapped and not-mapped arrays:

```
(poke) var a = [2,3,1]
(poke) var b = int[3] @ 0#B
(poke) b
[1179403647,65794,0]
(poke) qsort (a, IntComparator)
(poke) a
[1,2,3]
(poke) qsort (b, IntComparator)
(poke) b
[0,33620224,1179403647]
(poke) dump :from b'offset :size b'size :ascii 0
76543210  0011 2233 4455 6677 8899 aabb ccdd eeff
```

```
00000000: 0000 0000 0001 0102 7f45 4c46
```

Similarly, you can write functions that operate on abstract entities and data structures such as ELF relocations and sections, DWARF DIEs, *etc*, and the same code will work with non mapped and mapped values.

18.14.3 Mapping Simple Types

Simple values (*i.e.* integers, offsets, strings) cannot be mapped. Therefore, if the type specified in the map is a simple type, the resulting value will be a regular non-mapped value. Examples:

```
(poke) uint<8> @ 0#B
0x7fUB
(poke) string @ 0#B
"ELF"
```

18.14.4 Mapping Structs

Struct and union types can be referred by name in a mapping (`@`) operation. For example, given a struct type `Foo` defined as:

```
type Foo =
  struct
  {
    int i;
    long l;
  }
```

we could get a mapped struct value of that type like this:

```
var f = Foo @ 128#B;
```

18.14.5 Mapping Arrays

Arrays can be mapped in IO space in three different ways, depending on the characteristics of the type provided to the mapping operator.

18.14.5.1 Array maps bounded by number of elements

When an array type bounded by number of elements is used in a mapping operation, the resulting mapped array is also bounded by number of elements.

For example, this is how we would map an array of four 32-bit signed integers in the current IO space:

```
(poke) int[4] @ 0#B
[10,20,30,40]
```

Since you can also provide a dynamic array type to the map operator, the number of elements doesn't need to be constant. For example, given the variable `nelems` has a value of `2`:

```
(poke) var nelems = 2
(poke) int[nelems + 1] @ 0#B
[100,222,333]
```

If an end-of-file condition happens while mapping the array, because the number of elements specified in the array type, at the given offset, exceeds the capacity of the underlying IO device, an exception is raised and the mapping is not completed:

```
(poke) int[99999999999] @ 0#B
unhandled EOF exception
```

Likewise, if a constraint fails while performing the mapping (while mapping an array of structs, for example) an exception is raised and the map is aborted.

18.14.5.2 Array maps bounded by size

While dealing with binary formats, it often happens that the number of entities in a collection is given by the space they occupy, rather than the count itself.

For example, consider ELF sections holding relocations. These sections contain a collection of zero or more relocations. The layout of each relocation is specified by the following type:

```
type Elf64_Rela =
  struct
  {
    offset<Elf64_Addr,B> r_offset;
    Elf64_Xword r_info;
    Elf64_Sxword r_addend;
  };
```

The section is described by an entry in the ELF sections header table:

```
type Elf64_Shdr =
  struct
  {
    Elf_Word sh_name;
    Elf_Word sh_type;
    Elf64_Xword sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off sh_offset;
    offset<Elf64_Xword,B> sh_size;
    Elf_Word sh_link;
    Elf_Word sh_info;
    Elf64_Xword sh_addralign;
    offset<Elf64_Xword,b> sh_entsize;
  };
```

The relevant elements of `Elf64_Shdr` are `sh_offset` and `sh_size`, which indicate the offset of the beginning of the section's contents, and its size, respectively. At this point, if we wanted to get an array with all the relocations in the section, we could map an array bounded by number of elements like we saw in the previous section, like this:

```
(poke) Elf64_Rela[sh_size / 1#Elf64_Rela]
[... relocs ...]
```

However, this approach adoleces from two problems. First, it doesn't work with any entity type. For an offset like `1#Elf64_Rela` to work, it is required to know the size of the type specified as the unit at compile time. In the particular case of `Elf64_Rela`, that condition is satisfied, but too often that's not the case. For example, think about a section containing `NULL` terminated strings: you can't know the number of strings contained in the section until you actually read it.

Another problem is when the data in the header is corrupt. Using the mapping bounded by number of elements, we wouldn't realize it. It would be good if the tool would tell us whether the specified size actually holds an exact number of the requested elements.

A mapping bound by size is what we need. Fortunately, as we saw when discussing array types, Poke allows you to specify an offset instead of an integral value, in the array type specification. The right amount of entities (in this case relocations) to strictly satisfy the provided size will be mapped in the IO space. So, in order to obtain an array containing all the relocations in the section, we simply write:

```
(poke) Elf64_Rela[ehdr.sh_size] @ ehdr.sh_offset
[... relocs ...]
```

The strictness mentioned above is important. GNU poke will complain (and abort the mapping) if it is not possible to map an exact number of elements. Thus the following mapping would not be successful:

```
(poke) int[33#b] @ 0#B
unhandled out of map bounds exception
```

Like in mappings bounded by number of elements, if a constraint fails while performing the mapping, an exception is raised and the map is aborted.

18.14.5.3 Unbounded array maps

We mentioned above that if an end-of-file condition happens while performing a mapping (be it bounded by number of elements or bounded by size) an EOF exception is raised, and the mapping operation is aborted.

Unbounded array mappings are performed by using an unbounded array type in the mapping operation, like in:

```
type[] @ 0#B
```

The above construction will map values of type *type* in the IO space until there is an end-of-file condition, or a constraint fails, whatever happens first. When it is a constraint expression that fails, that last element is not included in the mapped array.

Let's assume a binary file contains a series of blocks, located one after the other, of a kind described by the following struct type:

```
type Block =
  struct
  {
     byte magic[2] : magic[0] == 'B' && magic[1] == 'K';
     ... other data ...
  };
```

we can map the blocks using an unbounded array map:

```
(poke) Block[] @ 0#B
[ ... blocks ... ]
```

If the blocks extend up to the end of the IO space, that many blocks will be mapped. If there is some other content in the file following the blocks, the constraint in the `magic` field will fail and will delimit the map that way (provided the binary format is well designed.)

18.14.5.4 Mapped bounds in bounded arrays

When an array map is bounded, be it by number of elements or by size, the bounding value can be mapped itself. To illustrate how this works, let's go back to our ELF file and the section containing relocations. First, we map an `Elf64_Shdr` to get the section header:

```
(poke) var shdr = Elf64_Shdr @ offset
(poke) shdr.sh_offset
120#B
(poke) shdr.sh_size
24#B
```

Now we map an array with the relocations themselves, using a map bounded by size, as we learned in the last section:

```
(poke) var relocs = ELF64_Rela[shdr.sh_size] @ shdr.sh_offset
(poke) relocs'length
3
```

Now, observe that `shdr.sh_size` is mapped itself! This means that, should the section size be modified (to accommodate an extra relocation, for example) the mapping of `relocs` will reflect that automatically:

```
(poke) shdr.sh_size = shdr.sh_size + 1#Elf64_Rela
(poke) relocs'length
4
```

This is certainly an useful idiom, that is often used while poking around. However, sometimes this is *not* what we want. If we don't want the mapping bounds of `relocs` to be tied to `shdr`, we can just use a temporary for the size:

```
(poke) var s = shdr.sh_size
(poke) var relocs = Elf64_Rela[s] @ shdr.sh_offset
```

Since simple values (such as the size above) are not mapped, this trick works as intended.

18.14.6 Mapping Functions

Struct fields can be of any type, and that includes function types or closures. This is especially useful to build interfaces.

For orthogonality, it is possible to map function types. The result of the mapping is not very useful though: mapping a function type results in a function with an empty body. If the function returns a value, a default value (zero for integral types, empty arrays for array types, *etc*) is returned.

Example:

```
(poke) type Fun = (int) int
(poke) Fun @ 0#B
#<closure>
(poke) (Fun @ 0#B) (10)
0x0
```

18.14.7 Non-strict Mapping

The map operator `@` performs what we call a *strict mapping*. This means that the data integrity of the mapped data is checked for. Often this is what we want.

However, sometimes we have to work with incorrect or incomplete data. In these cases, we want to inspect that data and complete it using poke.

For this purpose poke provides an alternate map operator `@!` that behaves exactly like the normal operator, but inhibits the control of data integrity.

The attribute `'strict` can be used to query whether a given value is strict or non-strict. Non-mapped values are strict by definition.

18.14.8 Unmapping

The unary `unmap` operator has the form:

```
unmap value
```

It gets any value and produces the same value, making it not mapped in case it is a mapped value.

This is useful when we want to read a data structure from the IO space (say, an array of integers) and then use it for storage without changing the underlying IO space. We would do something like:

```
(poke) var a = unmap (int[3] @ 10#B)
(poke) a[2] = 100
```

18.15 Exception Handling

Sometimes an error or some other unexpected situation arises. Poke provides an exceptions mechanism to deal with these situations.

18.15.1 Exceptions

Exceptions in Poke are values of type `Exception`, which is a struct defined like this:

```
type Exception =
  struct
  {
    int<32> code;
    string msg;
  };
```

where `code` identifies the type of the exception, and `msg` is supposed to be a textual description of the exceptional situation.

You can use codes `255` and higher for your own exceptions. For example:

```
raise Exception { code = 255; msg = "double upset event" };
```

User-defined exceptions should be registered using `exception_code` function. For example:

```
var E_my_exception = Exception {
  code = exception_code,
  msg = "double upset event",
};
```

where the `E_my_exception.code` is a unique number greater than or equal to `255`.

Exception codes in the range `0..254` are reserved for poke. These are used in predefined exceptions which are standard, and have specific meanings:

`E_generic`
: Generic error.

`E_out_of_bounds`
: Out of bounds exception. This can be raised when accessing containers, like arrays and strings.

`E_eof`
: End of file exception. This can be raised when mapping in the IO space.

`E_elem`
: Invalid element exception. This is raised when attempting to access union fields that do not exist.

`E_constraint`
: Constraint violation exception. This is raised when a constraint exception fails while mapping or constructing a struct.

`E_conv`
: Conversion exception. This can be raised while casting values.

`E_map_bounds`
: Out of map bounds exception. This can be raised while modifying a mapped value in a way it would violate its declared boundary (like the size of a mapped array.)

`E_map`
: No map exception. This is raised when trying to map a not mapped value.

`E_div_by_zero`
: Division by zero exception.

`E_no_ios`
: No IOS exception. This is raised when the IO space is accessed but there is no IO space.

`E_no_return`
: No return exception. This is raised when the end of a void function is reached.

`E_io` Generic IO exception.

`E_io_flags`
: Invalid flags were tried while opening an IO device.

`E_assert` Assertion failure exception. This is raised by `assert` statement.

`E_overflow`
: This exception is raised when signed overflow is detected in an arithmetic operation.

`E_perm` This exception is raised when some operation can't be performed due to incorrect (lack of) permissions or capabilities. An example is writing to a read-only IO space.

The exception codes of the standard exceptions are available in the form of `EC_*` variables. For example, this how you would raise an IO error with a particular message:

```
raise Exception { code = EC_io,
                  msg = "fluzo capacitator overheat impedes IO" }
```

18.15.2 `try-catch`

The `try-catch` statement provides a way to catch exceptions and handle them.

The simplest form of the statement is:

```
try stmt catch compound_stmt
```

where *stmt* is any statement and *compound_stmt* is a compound statement. First, *stmt* is executed. If during its execution an exception is raised, then *compound_stmt* is executed.

The second form of the statement allows you to catch just one type of exception:

```
try stmt catch if exp compound_stmt
```

where *exp* is an expression that should evaluate to an `Exception`. The handler *compound_stmt* will only be executed if that specific exception is caught. Any other exception will be re-raised.

The third form of the statement is the most generic:

```
try stmt catch (Exception formal) compound_stmt
```

where *formal* is a formal argument that contains the exception when *compound_stmt* is executed.

18.15.3 `try-until`

The `try-until` statement allows you to execute instructions until some exception is caught. The syntax is:

```
try stmt until exp
```

where *stmt* is the statement that will be executed repeatedly until some exception is raised. If the raised exception has type *exp* then execution continues normally. *exp* should be an expression that evaluates to an `Exception`.

This statement is particularly useful for mapping IO spaces until an `eof` condition occurs. For example, this is how we would compute with every integer in the current IO space:

```
var o = 0#B;
try
{
  compute (int @ o);
  o = o + 1#B;
} until E_eof;
```

It is possible to leave the loop from within *stmt* using the `break` statement.

18.15.4 `raise`

In previous sections we saw how exceptions are usually the side-product of performing certain operations. For example, a division by zero.

However, it is also useful to explicitly raise exceptions. The `raise` statement can be used for that purpose. Its syntax is:

```
raise exception;
```

where *exception* is an integer. This integer can be any number, but most often is one of the `E_*` codes defined in Poke.

18.15.5 exception-predicate

It is often needed to determine whether the execution of some given code raised some particular exception. A possibility is to use a `try-catch` statement like in the following example, a pretty-printer in an union type:

```
method _print = void:
{
  try print "#<%u32d>", simple_value;
  catch if E_elem { print "%v\n", complex_value; }
}
```

This works as intended, but it is somewhat verbose and not that easy to read for anyone not accustomed to the idiom. It is better to use the *exception predicate* operator, `?!`, that has two forms:

```
exp ?! exception
{ [statements...] } ?! exception
```

In the first form, the expression *exp* is executed. If the execution raises the exception *exception* then the result of the predicate is `0` (false), otherwise it is `1` (true). The value of the expression is discarded.

In the second form, the compound statement passed as the first operand is executed. If the exception *exception* is raised, then the result of the predicate is `0` (false), otherwise it is `1` (true).

Using exception predicates the pretty-printer above can be written in a much more readable way:

```
method _print = void:
{
  if (simple_value ?! E_elem)
    print "#<%u32d>", simple_value;
  else
    print "%v\n", complex_value;
}
```

Or, alternatively (and slightly more efficiently):

```
method _print = void:
{
  ({ print "<%u32d>", simple_value; } ?! E_elem)
    || print "%v\n", complex_value;
}
```

18.15.6 `assert`

The `assert` statement allows you test if a condition is true, if not, the program will raise an exception with code `EC_assert`.

```
assert (condition)
```

```
assert (condition, message)
```

The optional *message* will be part of the `msg` field of raised `Exception` to explain the situation.

```
assert (1 == 1);
assert (0 == 0, ``Zero is equal to zero'');
```

`assert` is useful for writing unit tests.

18.16 Terminal

Poke programs have access to a text-oriented terminal, that can be used to communicate with the user. The system provides several primitive functions to manipulate the terminal, which are described in the sections below.

18.16.1 Terminal Colors

At every moment the text terminal is configured to use certain foreground and background colors. Initially, these are the *default colors*, which depend on the underlying terminal capabilities, and maybe also on some system wide configuration setting.

Terminal colors are encoded in triplets of integers signifying RGB colors, *i.e.* levels of beams for red, green and blue. These triplets are implemented as values of type `int<32>[3]`.

The special value `[-1, -1, -1]` means the default color, for both foreground and background. There are two predefined variables for this:

```
var term_default_color = [-1,-1,-1];
var term_default_bgcolor = [-1,-1,-1];
```

Getting and setting the foreground and background colors is performed using the `term_get_color`, `term_set_color`, `term_get_fgcolor` and `term_set_fgcolor` functions, with signatures:

```
fun term_get_color = int<32>[3]
fun term_set_color = (int<32>[3] color) void
fun term_get_bgcolor = int<32>[3]
fun term_set_bgcolor = (int<32>[3] color) void
```

18.16.2 Terminal Styling

Styling of the output sent to the terminal is supported in the form of *styling classes*. Each class is characterized by a name, a string.

A class is activated by using the function `term_begin_class`, and deactivated by using the function `term_end_class`. Several classes can be activated simultaneously, but they should be deactivated in reverse order. It is down to the implementation to decide how to honor some given class, *i.e.* to decide how to characterize it physically (visually, or audibly, or using an electrical discharge on the keyboard, or whatever) The class may very well be ignored all together. Poke programs should never rely on how a specific class is implemented.

The `term_begin_class` and `term_end_class` functions have the following signatures:

```
fun term_begin_class = (string class) void
fun term_end_class = (string class) void
```

Styling classes can overlap, but trying to deactivate them in the wrong order will cause an error in the form of an `E_inval` exception.

18.16.3 Terminal Hyperlinks

The poke terminal supports terminal hyperlinks. Each hyperlink is characterized by an URL, and an ID. The `term_begin_hyperlink` and `term_end_hyperlink` functions have the following signatures:

```
fun term_begin_hyperlink = (string url, string id) void
```

```
fun term_end_hyperlink = void
```

Hyperlinks can overlap, but trying to end an hyperlink when there is no t a currently open one will result in an `E_generic` exception raised.

18.17 Printing

Poke programs can print text to the standard output in two ways: simple unformatted output, and formatted output.

18.17.1 `print`

The `print` statement prints the given string to the standard output. `print` outputs text strings verbatim. It can be invoked using two alternative syntaxes, which are equivalent:

```
print (str);
print str;
```

`print` is simple, but fast. It is good to use it in simple cases where the information to print out doesn't require any kind of formatting and styling.

18.17.2 `printf`

The `printf` statement gets a format string and, optionally, a list of values to print. It can be invoked using two alternative syntaxes, which are equivalent:

```
printf (fmt[, value...])
printf fmt[, value...]
```

The format string *fmt* is printed verbatim to the standard output, but for *format tags* which start with `%`. These format tags are interpreted especially. Most of the format tags "consume" one of the specified values. Every value in the list shall be described by a tag, or the compiler will signal an error. Likewise, values without a corresponding describing tag is an error. These tags are:

`%s` Print the argument as a string.

`%i`*bits*`(d|x|o|b|c)`
: Print the argument as a signed integer of size *bits*. The last letter determines how the argument is printed.

  `d` Print the integer in decimal.

  `x` Print the integer in hexadecimal.

  `o` Print the integer in octal.

  `b` Print the integer in binary.

  `c` Print the integer as an ASCII character. This only works with 8 bit integers.

`%u` Same as `%i`, but for unsigned integers.

`%c` A shorter way to write `%u8c`.

`%v` Print the value printed representation of the argument, which can be of any type including complex types like arrays and structs. This is similar to the `write` operation available in many Lisp systems.

This tag supports an optional numerical argument (restricted to a single digit) that specifies the maximum depth level when printing structures. Example:

```
(poke) printf ("%1v\n", struct { s = struct { i = 10 },  l = 20L });
struct {s=struct {...},l=0x14L}
```

By default, the depth level is `0`, which means no limit.

This tag also support an optional flag that specifies how the value is printed. The supported tags are `F` (for *flat*) and `T` (for *tree*). The default is flat. Example:

```
(poke) printf ("%Tv\n", struct { s = struct { i = 10 },  l = 20L });
struct {
  s=struct {
    i=0xa
  },
  l=0x14L
}
(poke) printf ("%Fv\n", struct { s = struct { i = 10 },  l = 20L });
struct {s=struct {i=0xa},l=0x14L}
```

This tag is mainly intended to be used in pretty-printers.

The following format tags do not consume arguments. They support emitting styled text using the libtextstyle approach of having styling classes that you can customize in a `.css` file.

`%<`*classname*`:`
: Start the styling class with name *classname*. The class name cannot be empty.

`%>`
: End the last opened styling class. All styling classes should be ended before finishing the format string.

`%%`
: Print `%` character.

Note that styling classes can be nested, but all classes should be ended before finishing the format string.

If you use a *name* class, you can define how to style it in the `.css` file (poke installs and uses poke-default.css but you can set the `POKE_STYLE` environment variable to point to another css) like this:

```
.NAME { text-decoration: blink; color : pink; }
```

Examples:

```
(poke) printf "This is a NAME: %<NAME:xxx%>"
This is a NAME: xxx
(poke) printf "Name: %<string:%s%> Age: %<integer:%i32d%>, "Jose", 39
Name: Jose Age: 39
```

18.18 Comments

There are several ways to document your Poke programs: comments of several types and support for separator characters.

18.18.1 Multi-line comments

Poke supports C-like multi-line, comments, which is text enclosed between `/*` and `*/` sequences. These comments cannot be nested.

18.18.2 Single line comments

C++-like single line comments are supported. Everything after the `//` sequence is interpreted as a comment, until the end of the line or the end of the file, whatever comes first.

18.18.3 Vertical separator

Poke ignores form feed characters (ASCII code 12, often visualized as ^L). In GNU software, this character is traditionally used to separate conceptually different entities in source files.

18.19 Modules

It is common for pickles to depend on stuff defined in other pickles. In such cases, the `load` language construction can be used to load pickles from your Poke program. The syntax is:

```
load module;
```

where *module* is the name of the pickle to load.

For example, your Poke program may want to access some ELF data structures. In that case, we can just do:

```
/* Pickle to poke the contents of some ELF section.  */

load elf;

/* ... code  ... */
```

When asked to open a module, poke assumes it is implemented in a file named *`module`*`.pk`. In the example above, it will try to load `elf.pk`.

The pickles are searched first in the current working directory. If not found, then *`prefix`*`/share/poke/`*`module`*`.pk` is tried next.

If the environment variable `POKEDATADIR` is defined, it replaces *`prefix`*`/share/poke`. This is mainly intended to test a poke program before it gets installed in its final location.

Nothing prevents you to load the same pickle twice. This will work if the pickle doesn't include definitions, but just executes statements. Otherwise, you will likely get an error due to trying to define stuff twice.

There is an alternate syntax of the `load` construction that is useful when the module is implemented in a file whose name doesn't conform to a Poke identifier. This happens, for example, when the file name contains hyphens. Example:

```
load "my-pickle.pk";
```

Note that if you use this variant of `load`, you must specify the full file name, including whatever extension it uses (usually `.pk`).

18.20 System

Poke provides several constructions to interact with the operating system. These are described in the subsections below.

18.20.1 getenv

The `getenv` built-in function has the form:

```
fun getenv (string name) string:
```

Given the name of an environment variable, it returns a string with its value. If the variable is not defined in the environment, then the `E_inval` exception is raised.

18.20.2 rand

The `rand` built-in function has the form:

```
fun rand = (uint<32> seed = 0) int<32>:
```

It returns a pseudo-random number in the range of the unsigned 32-bit integers. If a *seed* different to zero is provided, it is used to initialize a new sequence of pseudo-random numbers (which includes the one returned.)

The `get_time` builtin function can be used in order to seed the pseudo-random number generator very easily:

```
srand (get_time[1])
```

18.21 VM

The Poke language provides a set of pre-defined functions which can be used to access and modify properties of the underlying implementation running the Poke programs.

18.21.1 vm_obase

The pre-defined function `vm_obase` returns the current output base in use in the underlying VM. It has the following prototype:

```
fun vm_obase = int<32>:
```

18.21.2 vm_set_obase

The pre-defined function `vm_set_obase` sets the output base in the underlying VM. It has the following prototype:

```
fun vm_set_obase = (int<32> obase) void:
```

Where *obase* is the output base to set. The base must be one of 2 (for binary), 8 (for octal), 10 (for decimal) or 16 (for hexadecimal). If the provided base is not contained in that set, `vm_set_obase` raises `E_inval`.

18.21.3 vm_opprint

The pre-defined function `vm_opprint` returns a boolean indicating whether pretty-printers are used when printing struct values, or not. It has the following prototype:

```
fun vm_opprint = int<32>:
```

18.21.4 vm_set_opprint

The pre-defined function `vm_set_opprint` sets a flag in the underlying VM, indicating whether pretty-printers are used when printing struct values. It has the following prototype:

```
fun vm_set_opprint = (int<32> pretty_print_p) void:
```

18.21.5 vm_oacutoff

The pre-defined function `vm_oacutoff` returns an integer indicating the current cutoff limit used when printing array values. It has the following prototype:

```
fun vm_oacutoff = int<32>:
```

18.21.6 vm_set_oacutoff

The pre-defined function `vm_set_oacutoff` sets the current cutoff limit used when printing array values. It has the following prototype:

```
fun vm_set_oacutoff = (int<32> cutoff) void:
```

where *cutoff* is an integer specifying the maximum number of array elements that get printed.

18.21.7 vm_odepth

The pre-defined function `vm_odepth` returns an integer indicating the maximum depth level used when printing nested structures. It has the following prototype:

```
fun vm_odepth = int<32>:
```

18.21.8 vm_set_odepth

The pre-defined function `vm_set_odepth` sets the maximum depth level to be used when printing nested structures. It has the following prototype:

```
fun vm_set_odepth = (int<32> depth_level) void:
```

where *depth_level* specifies the number of depth levels in nested structs to consider when printing nested structures.

18.21.9 vm_oindent

The pre-defined function `vm_oindent` returns an integer indicating the indentation step. It has the following prototype:

```
fun vm_oindent = int<32>:
```

18.21.10 vm_set_oindent

The pre-defined function `vm_set_oindent` sets the indentation level used when printing nested structures. It has the following prototype:

```
fun vm_set_oindent = (int<32> step) void:
```

where *step* specifies how much to indent when printing nested structures. This value typically translates into the number of white spaces (or tabs) used in terminals, but this must not be relied on since other printing media may interpret it differently.

18.21.11 vm_omaps

The pre-defined function `vm_omaps` returns a boolean indicating whether offset information is included when printing out values. It has the following prototype:

```
fun vm_omaps = int<32>:
```

18.21.12 vm_set_omaps

The pre-defined function `vm_set_omaps` sets whether offset information is included when printing out values. It has the following prototype:

```
fun vm_set_omaps = (int<32> omaps_p) void:
```

18.21.13 vm_omode

The pre-defined function `vm_omode` returns an integer indicating the currently enabled output mode. It has the following prototype:

```
fun vm_omode = int<32>:
```

There are currently two supported output modes:

```
var VM_OMODE_PLAIN = 0;
var VM_OMODE_TREE = 1;
```

18.21.14 vm_set_omode

The pre-defined function `vm_set_omode` sets the output mode to use by default when printing values. It has the following prototype:

```
fun vm_set_omode = (int<32> omode) void:
```

Where *omode* is one of the `VM_OMODE_*` values documented above.

18.22 Debugging

18.22.1 __LINE__ and __FILE__

When printing traces it is often useful to include a description of the location of the trace. The poke compiler provides two builtins for this purpose.

`__LINE__` Expands to an unsigned 64-bit integer containing the current line of the program being compiled.

`__FILE__` Expands to a string with the name of the file currently being compiled. If the program is read from the standard input (like in the REPL) then this is `"<stdin>"`.

18.22.2 strace

The `strace` built-in function has the prototype:

```
fun strace = void:
```

It prints out the current contents of the PVM stack. This is especially useful to debug the code generated by the compiler.

19 The Standard Library

GNU poke ships with a standard library for Poke programs that is available to Poke programs. The following sections detail the facilities provided by the library.

19.1 Standard Integral Types

The Poke standard library provides the following standard integral types.

`bit` 1-bit unsigned integer.

`nibble` 4-bit unsigned integer.

`uint8`
`byte`
`char` 8-bit unsigned integer.

`uint16`
`ushort` 16-bit unsigned integer.

`uint32`
`uint` 32-bit unsigned integer.

`uint64`
`ulong` 64-bit unsigned integer.

`int8` 8-bit signed integer.

`int16`
`short` 16-bit signed integer.

`int32`

`int` 32-bit signed integer.

`int64`

`long` 64-bit signed integer.

19.2 Standard Offset Types

The Poke standard library provides the following standard offset types.

`off64` 64-bit signed offset in bits.

`uoff64` 64-bit unsigned offset in bits.

19.3 Standard Units

The following list of units are defined in the standard library.

Base units:

`b` bits.

`N` nibbles.

`B` bytes.

Base 10 units:

`Kb` kilo bits (1000 bits.)

`KB` Kilo bytes (1000 bytes.)

`Mb` Mega bits.

`MB` Mega bytes.

`Gb` Giga bits.

`GB` Giga bytes.

Base 2 units:

`Kib` kilo bits (1000 bits.)

`KiB` Kilo bytes (1000 bytes.)

`Mib` Mega bits.

`MiB` Mega bytes.

`Gib` Giga bits.

`GiB` Giga bytes.

19.4 Conversion Functions

The Poke standard library provides the following functions to do useful conversions.

19.4.1 `catos`

It is often useful to convert arrays of characters into strings. The standard function `catos` provides the following interface:

```
fun catos = (char[] chars) string: { ... }
```

It builds a string containing the characters in *chars*, and returns it. Examples:

```
(poke) catos (['a','b','c'])
"abc"
(poke) catos (['\0'])
""
```

Note that if the passed array contains a `NULL` character `'\0'` then no further characters are processed. For example:

```
(poke) catos (['f','o','o','\0','b','a','r'])
"foo"
```

19.4.2 `stoca`

Sometimes we want to store strings in character arrays. The standard function `stoca` provides the following interface:

```
fun stoca = (string s, char[] ca, char fill = 0) void: { ... }
```

It fills the given array *ca* with the contents of the string *s*. If the string doesn't fit in the array, then `E_out_of_bounds` is raised. If the length of the string is less than the length of the array, the extra characters are set to the *fill* character, which defaults to `\0`.

19.4.3 `atoi`

The standard function `atoi` provides the following interface:

```
fun atoi = (string str, int base = 10) long: { ... }
```

It parses a signed integral number in the given *base* in the string *str* and returns it as a signed 64-bit integer.

The accepted values for *base* are `2`, `8`, `10` (the default) and `16`. If any other base is requested an `E_inval` exception is raised.

Note that atoi allows for extra information to be stored in *str* after the parsed integer. Thus, this works:

```
(poke) atoi ("10foo")
10L
```

19.4.4 ltos

The `ltos` standard function converts a given long integer into its printed representation in some given base. It has the following prototype:

```
fun ltos = (long i, uint base = 10) string:
```

where *i* is the number for which to calculate the printed representation, and *base* is a number between 0 and 16.

19.5 Array Functions

The Poke standard library provides the following functions to do work on arrays:

19.5.1 reverse

The standard function `reverse` provides the following interface:

```
fun reverse = (any[] a) void:
```

It reverses the elements of the given array.

19.6 String Functions

The Poke standard library provides the following functions to do work on strings:

19.6.1 ltrim

The standard function `ltrim` provides the following interface:

```
fun ltrim = (string s, string cs = " \t") string: { ... }
```

It returns a copy of the input string `s` with any leading character that also appears in *cs* removed.

19.6.2 rtrim

The standard function `rtrim` provides the following interface:

```
fun rtrim = (string s, string cs = " \t") string: { ... }
```

It returns a copy of the input string `s` with any trailing character that also appears in *cs* removed.

19.6.3 strchr

The standard function `strchr` provides the following interface:

```
fun strchr = (string s, uint<8> c) int<32>: { ... }
```

It returns the index of the first occurrence of the character *c* in the string *s*. If the character is not found in the string, this function returns the length of the string.

19.7 Sorting Functions

19.7.1 qsort

The standard function `qsort` has the following prototype:

```
fun qsort = (any[] array, Comparator cmp_f,
             long left = 0,
             long right = array'length - 1) void
```

where *array* is the array to sort, *cmp_f* is a comparator function, *left* is the index of the first array element to include in the sorting, and *right* is the index of the last array element to include in the sorting. Both *left* and *right* are optional, and the default is to cover the whole array.

The comparator function *cmp_f* should have the following prototype:

```
type Comparator = (any,any)int<32>;
```

for example:

```
fun CompareInts = (any a, any b) int<32>:
{
  var ai = a as int<32>;
  var bi = b as int<32>;

  if (ai == bi)
    return 0;
  else if (ai > bi)
    return 1;
  else
    return -1;
}
```

19.8 CRC Functions

Many file formats use checksums of one sort or another. Therefore you may want to write such a checksum or verify a checksum in a constrained field. See Section 18.5.6 [Field Constraints], page 132.

Some formats use simple additive checksums. Another common checksum is the Cyclic Redundancy Checksum (CRC). The standard function `crc32` calculates the 32 bit CRC defined by ISO-3309.

```
fun crc32 = (byte[] buf) uint<32>: { ... }
```

This function returns the 32 bit CRC for the data contained in the array *buf*.

19.9 Dates and Times

Often a format encodes a date and time expressed as the number of seconds since midnight, January 1st 1970. You could map these simply as integers. However, the standard library provides two types `POSIX_Time32` and `POSIX_Time64` which include pretty-printers to display the date in a human readable format. The definition is:

```
type POSIX_Timesize = struct
{
  uint<size>  seconds;

  method _print = void:
  { ... }
}
```

where *size* is either 32 or 64. When pretty printing is enabled, a mapped value of these types will display similar to

```
#<2019-Dec-12 8:54:56>
```

whereas when pretty printing is not enabled, this example would be displayed as:

```
POSIX_Time32 {seconds=1576140896U}
```

Note that timestamps of this type do not account for leap seconds and are agnostic towards timezone.

Occasionally you might wish to print an unmapped timestamp value in a human readable format. To do this, you can use the `ptime` function, which is defined as follows:

```
fun ptime = (uint<64> seconds) void:
{ ... }
```

This pickle also provides a type `Timespec` that can store Unix times to nanosecond precision, and has the form:

```
type Timespec = struct
{
  int<64> sec;
  int<64> nsec;
};
```

The function `gettimeofday` uses the `get_time` builtin in order to construct an instance of `Timespec`. For example:

```
(poke) gettimeofday
Timespec {
  sec=1604918423L,
  nsec=753492546L
}
(poke) ptime (gettimeofday.sec)
2020-Nov-9 10:40:36
```

19.10 Offset Functions

19.10.1 alignto

The `alignto` function has the prototype:

```
fun alignto = (uoff64 offset, uoff64 to) uoff64:
{ ... }
```

It returns an offset that is the result of aligning the given *offset* to the given alignment *to*.

20 The Machine-Interface

GNU poke can be executed in a special mode in which it communicates with a client using a machine-friendly interface. This makes it possible to write programs like graphical user interfaces, testing programs and the like. This section describes this operation mode and provides a full description of the interface, for client application writers.

20.1 MI overview

A *client application* can communicate with poke through the machine-interface:

```
+--------+       MI     +--------+
| client |------ O------|  poke  |
+--------+              +--------+
```

There are two ways in which the communication can be performed: through pipes and through TCP network connections.

When using the pipe operation mode, the client application runs poke as a sub process. Data is obtained from poke reading from its standard output, and data is fed to poke writing to its standard input:

```
+--------+        stdin +--------+
| client |------------->|  poke  |
+--------+              +--------+
    ^                        | stdout
    |                        |
    +------------------------+
```

When using the TCP network operation mode the client uses a bidirectional socket to communicate with a running poke process. This has the advantage of allowing having poke and the client application running on different machines:

```
+--------+ socket    (----)__     socket +--------+
| client |<-------->( network )<-------->|  poke  |
+--------+           (-------)           +--------+
```

20.2 Running poke in MI mode

The following poke command-line options are relevant to the machine-interface.

‘`--mi`’ Run poke in Machine-Interface mode.

‘`--mi-socket=`*port*’
: Use a network TCP socket to communicate with the client. *port* is the port to use, or zero. If zero, then poke chooses a port on its own. In both cases, poke prints the port used when it starts.

If not using network sockets, poke starts talking MI immediately using the standard input and standard output:

```
$ poke --mi
^@^@^@G{"poke_mi":0,"type":2,"data":{"type":0,"args":{"version":"0.1-beta"}}}
```

When using sockets, poke prints the number of the socket where it is listening for requests in the standard output:

```
$ poke --mi --mi-socket=0
1234
```

20.3 MI transport

At the lowest level the communication is performed in terms of *frame messages*.

The layout of each message is:

```
type PMI_FrameMessage =
 struct
 {
    big uint<4> size : size <= 2048;
    byte[size] payload;
 }
```

Where *size* is the length of the payload, measured in bytes. The maximum length of a frame message payload is two kilobytes.

20.4 MI protocol

Requests are initiated by the client. Once a request is sent, it will trigger a response. The response is paired with the triggering request by the request's sequence number.

Responses are initiated by poke, in response to a request received from the client.

Events are initiated by poke.

20.4.1 MI Requests

20.4.1.1 Request EXIT

This request asks poke to exit in an orderly way. It has no arguments.

20.4.1.2 Request PRINTV

This request is used to get the printed representation of a given Poke value.

Arguments:

value The Poke value to get the printed representation of.

20.4.2 MI Responses

20.4.2.1 Response EXIT

This is the response to the EXIT request.

Attributes:

success_p If `true`, poke will exit. If `false`, something prevents poke to exit.

errmsg If *success_p* is `false`, this attribute contains a string indicating the reason why poke refuses to exit.

20.4.2.2 Response PRINTV

This is the response to the PRINTV request.

Attributes:

success_p If `true`, poke was able to get the printed representation of the value. If `false`, an error occurred that prevented poke to handle the request.

string If *success_p* is `true`, this attribute contains the printed representation of the Poke value that was passed to the request.

errmsg If *success_p* is `false`, this attribute contains a string indicating the reason why poke could not get the printed representation of the value.

20.4.3 MI Events

20.4.3.1 Event INITIALIZE

This event is sent by poke when it finishes initializing and is ready to receive requests.

Arguments:

mi_version An integer with the version of the MI protocol that this poke instance speaks.

version A string with the printable representation of the version of the poke instance. This is intended to be used for the client to show to the users.

21 The Poke Virtual Machine

21.1 PVM Instructions

21.1.1 VM instructions

21.1.1.1 Instruction canary

Synopsys:

```
canary
```

This instruction installs a canary to mark the bottom of the stack, which is cheched by the 'exit' instruction. To be most effective this should be executed before the stack is used for the first time.

Stack: `( -- )`

21.1.1.2 Instruction exit

Synopsys:

```
exit
```

Do some cleanup and finish execution of a PVM program. This checks the stack sentinel installed by the 'canary' instruction.

Stack: `( -- )`

21.1.1.3 Instruction pushend

Synopsys:

```
pushend
```

Push the current endianness on the stack. This endianness is part of the global state of the PVM.

Stack: `( -- INT )`

21.1.1.4 Instruction popend

Synopsys:

```
popend
```

Pop a signed integer from the stack and make it the current endianness in the PVM. The possible values for endianness are IOS_ENDIAN_LSB and IOS_ENDIAN_MSB.

Stack: `( INT -- )`

21.1.1.5 Instruction pushob

Synopsys:

```
pushob
```

Push output base.

This instruction pushes a signed integer value with the output base that is used when printing PVM values. Valid values are 2, 8, 10 and 16.

Stack: `( -- INT )`

21.1.1.6 Instruction popob

Synopsys:

```
popob
```

Pop and set output base.

This instructions pops a signed integer from the stack and uses it to set the new output base to be used when printing PVM values. Valid values are 2, 8, 10 and 16.

If an invalid obase is specified then this instruction raises PVM_E_INVAL.

Stack: `( INT -- )`

Exceptions: `PVM_E_INVAL`

21.1.1.7 Instruction pushom

Synopsys:

```
pushom
```

Push output mode.

This instruction pushes a signed integer value with the output mode that is used when printing PVM values. Valid values are 0 for flat mode, and 1 for tree mode.

Stack: `( -- INT )`

21.1.1.8 Instruction popom

Synopsys:

```
popom
```

Pop and set output mode.

This instructions pops a signed integer from the stack and uses it to set the new output mode to be used when printing PVM values. Valid values are 0 for flat mode and 1 for tree mode.

If an invalid omode is specified then this instruction raises PVM_E_INVAL.

Stack: `( INT -- )`

Exceptions: `PVM_E_INVAL`

21.1.1.9 Instruction pushoo

Synopsys:

```
pushoo
```

Push output offsets mode.

This instruction pushes a boolean encoded in a signed integer value indicating whether to show offsets when printing PVM values.

Stack: `( -- INT )`

21.1.1.10 Instruction popoo

Synopsys:

```
popoo
```

Pop and set output offsets mode.

This instructions pops a boolean encoded in a signed integer from the stack and uses it to set the new output offset mode to be used when printing PVM values.

Stack: `( INT -- )`

21.1.1.11 Instruction pushoi

Synopsys:

```
pushoi
```

Push output indentation mode.

This instructions pushes an integer to the stack with the current indentation step configured in the VM. The indentation step determines how many white characters to use in each indentation level when printing output.

Stack: `( -- INT )`

21.1.1.12 Instruction popoi

Synopsys:

```
popoi
```

Pop and set output indentation step mode.

This instructions pops an integer from the stack and uses it to set the current indentation step in the VM. The indentation step determines how many white characters to use in each indentation level when printing output.

Stack: `( INT -- )`

21.1.1.13 Instruction pushod

Synopsys:

```
pushod
```

Push output depth.

This instruction pushes a signed integer indicating the depth to use when printing PVM values.

Stack: `( -- INT )`

21.1.1.14 Instruction popod

Synopsys:

```
popod
```

Pop and set output depth.

This instructions pops a signed integer indicating the maximum depth included by the VM when printing values.

Stack: `( INT -- )`

21.1.1.15 Instruction pushoac

Synopsys:

```
pushoac
```

Push output array cutoff.

This instruction pushes a signed integer indicating the number of elements that the VM includes in the printed representation of PVM array values.

Stack: `( -- INT )`

21.1.1.16 Instruction popoac

Synopsys:

```
popoac
```

Pop and set output array cutoff.

This instructions pops a signed integer indicating the number of elements that the VM includes in the printed representation of PVM array values.

Stack: `( INT -- )`

21.1.1.17 Instruction pushopp

Synopsys:

```
pushopp
```

Push pretty-print usage.

This instruction pushes a signed integer indicating whether the VM is configured to use pretty-printers.

Stack: `( -- INT )`

21.1.1.18 Instruction popopp

Synopsys:

```
popopp
```

Pop and set usage of pretty-printers.

This instructions pops a signed integer indicating whether to use pretty-printers when printing values and sets it in the VM.

Stack: `( INT -- )`

21.1.1.19 Instruction pushoc

Synopsys:

```
pushoc
```

Push the current output color to the stack, encoded as a RGB triplet.

Stack: `( -- INT INT INT )`

21.1.1.20 Instruction popoc

Synopsys:

```
popoc
```

Pop the RGB triplet at the top of the stack and use it to set the new terminal output color.

Stack: `( INT INT INT -- )`

21.1.1.21 Instruction pushobc

Synopsys:

```
pushobc
```

Push the current output background color to the stack, encoded as a RGB triplet.

Stack: `( -- INT INT INT )`

21.1.1.22 Instruction popobc

Synopsys:

```
popobc
```

Pop the RGB triplet at the top of the stack and use it to set the new terminal output background color.

Stack: `( INT INT INT -- )`

21.1.1.23 Instruction sync

Synopsys:

```
sync
```

Handle pending signals, and raise exceptions accordingly. This instruction should be emitted in strategic places, such as before backwards jumps and at function prolog, to assure signals are eventually attended to.

Stack: `( -- )`

Exceptions: `PVM_E_SIGNAL`

21.1.2 IOS related instructions

21.1.2.1 Instruction open

Synopsys:

```
open
```

Open a new IO space. The handler string and a set of flags are passed on the stack. The descriptor of the opened IOS is pushed to the stack as a signed integer.

If there is no other IO space opened when this instruction is executed, then the just opened space becomes the current IO space.

If it is not possible to open the IO space according to the provided flags, the PVM_E_IOFLAGS exception is raised. If there is any other error performing the operation, PVM_E_IO is raised.

Stack: `( STR ULONG -- INT )`

Exceptions: `PVM_E_IOFLAGS, PVM_E_IO`

21.1.2.2 Instruction close

Synopsys:

```
close
```

Close an IO space. The descriptor of the space to close is provided on the stack as a signed integer.

If the specified IO space doesn't exist, this instruction raises PVM_E_NO_IOS. If the operation fails, it raises PVM_E_IO.

Stack: `( INT -- )`

Exceptions: `PVM_E_NO_IOS, PVM_E_IO`

21.1.2.3 Instruction flush

Synopsys:

```
flush
```

Flush an IO space. The descriptor of the space to flush, and the bit-offset up to which perform the flushing are provided on the stack.

If the specified IO space doesn't exist, this instruction raises PVM_E_IO.

Stack: `( INT ULONG -- )`

Exceptions: `PVM_E_IO`

21.1.2.4 Instruction pushios

Synopsys:

```
pushios
```

Push the descriptor of the current IO space on the stack, as a signed integer. If no IO space is currently opened, raise PVM_E_NO_IOS.

Stack: `( -- INT )`

Exceptions: `PVM_E_NO_IOS`

21.1.2.5 Instruction popios

Synopsys:

```
popios
```

Pop an IO space descriptor from the stack and set it as the current IO space. If the specified descriptor doesn't identify an IO space, raise PVM_E_NO_IOS.

Stack: `( INT -- )`

Exceptions: `PVM_E_NO_IOS`

21.1.2.6 Instruction ioflags

Synopsys:

```
ioflags
```

Push an unsigned 64-bit integer with the flags of the given IO space on the stack. The IO space is identified by a descriptor, which is a signed integer. If the given IO space doesn't exist, raise PVM_E_NO_IOS.

Stack: `( INT -- INT ULONG )`

Exceptions: `PVM_E_NO_IOS`

21.1.2.7 Instruction iosize

Synopsys:

```
iosize
```

Push the size of the given IO space on the stack, as an offset. The IO space is identified by a descriptor, which is a signed integer. If the given IO space doesn't exist, raise PVM_E_NO_IOS.

Stack: `( INT -- INT OFF )`

Exceptions: `PVM_E_NO_IOS`

21.1.2.8 Instruction iogetb

Synopsys:

```
iogetb
```

Each IO space has a bias associated with it, which by default is 0 bits. This bias is applied to the offset given to every read/write operation.

This instruction pushes the bias associated to the given IO space to the stack, as an offset. If the given IO space doesn't exist then the exception PVM_E_NO_IOS is raised.

Stack: `( INT - INT OFF )`

Exceptions: `PVM_E_NO_IOS`

21.1.2.9 Instruction iosetb

Synopsys:

```
iosetb
```

Each IO space has a bias associated with it, which by default is 0 bits. This bias is applied to the offset given to every read/write operation.

This instruction sets the bias associated to the given IO space. The bias is specified as an offset. If the given IO space doesn't exist, the exception PVM_E_NO_IOS is raised.

Stack: `( OFF INT -- OFF )`

Exceptions: `PVM_E_NO_IOS`

21.1.3 Function management instructions

21.1.3.1 Instruction call

Synopsys:

```
call
```

Call a closure on the stack, passing the specified arguments. After the execution of the closure, control is transferred to the instruction immediately following the call instruction.

Stack: `( ARG1 ... ARGN CLOSURE -- RETVAL )`

21.1.3.2 Instruction prolog

Synopsys:

```
prolog
```

Prepare the PVM for the execution of a function. This instruction shall be the target of every 'call' instruction and shall be the first instruction in every function body.

Stack: `( -- )`

21.1.3.3 Instruction return

Synopsys:

```
return
```

Return from a function. A function can have many 'return' instructions.

Stack: `( -- )`

21.1.4 Environment instructions

21.1.4.1 Instruction pushf

Synopsys:

```
pushf N
```

Push a new lexical frame. If the argument N is bigger than zero, it indicates the number of entries in the frame. If N is 0, it means we don't know how many entries will be stored in the frame.

Stack: `( -- )`

21.1.4.2 Instruction popf

Synopsys:

```
popf N
```

Pop N lexical frames.

Stack: `( -- )`

21.1.4.3 Instruction pushvar

Synopsys:

```
pushvar BACK, OVER
```

Retrieve the value of a variable from the lexical environment and push it on the stack. The lexical address of the variable is specified as arguments to the instruction.

Stack: `( -- VAL )`

21.1.4.4 Instruction pushtopvar

Synopsys:

```
pushtopvar OVER
```

Retrive the value of a variable from the top-level frame of the lexical environment and put it on the stack. The OVER part of the lexical address of the variable is specified as an argument to the instruction.

If the variable is not found then raise E_INVAL.

Stack: `( -- VAL )`

21.1.4.5 Instruction popvar

Synopsys:

```
popvar BACK, OVER
```

Pop a value from the stack and set it as the value of a variable having the lexical address specified in the arguments, in the current lexical environment.

Stack: `( VAL -- )`

21.1.4.6 Instruction regvar

Synopsys:

```
regvar
```

Pop a value from the stack and use it as the value for a new variable in the current lexical environment.

Stack: `( VAL -- )`

21.1.4.7 Instruction duc

Synopsys:

```
duc
```

Make a copy of the closure at the top of the stack, and replace it.

Stack: `( CLS -- CLS )`

21.1.4.8 Instruction pec

Synopsys:

```
pec
```

Put the current lexical environment to the closure at the top of the stack.

Stack: `( CLS -- CLS )`

21.1.5 Printing Instructions

In the following instructions the meaning of the argument BASE is the following:

2 - print the number in binary. 8 - print the number in octal. 16 - print the number in hexadecimal. Any other value - print the number in decimal.

21.1.5.1 Instruction indent

Synopsys:

```
indent
```

Indent the output in the terminal for LVL levels of indentation, using STEP white chars in each indentation level.

LVL is an integer in the under top stack. STEP is an integer in the top of the stack.

Stack: `( INT INT -- )`

21.1.5.2 Instruction printi

Synopsys:

```
printi BITS
```

Given a signed integer and a numeration base in the stack, print the integer to the terminal.

Stack: `( INT INT -- )`

21.1.5.3 Instruction printiu

Synopsys:

```
printiu BITS
```

Given an unsigned integer and a numeration base in the stack, print the integer to the terminal.

Stack: `( UINT INT -- )`

21.1.5.4 Instruction printl

Synopsys:

```
printl BITS
```

Given a long and a numeration base in the stack, print the integer to the terminal.

Stack: `( LONG INT -- )`

21.1.5.5 Instruction printlu

Synopsys:

```
printlu BITS
```

Given an unsigned long and a numeration base in the stack, print the integer to the terminal.

Stack: `( ULONG INT -- )`

21.1.5.6 Instruction prints

Synopsys:

```
prints
```

Print the string at the top of the stack.

Stack: `( STR -- )`

21.1.5.7 Instruction beghl

Synopsys:

```
beghl
```

Begin an hyperlink, using the URL and ID on the stack.

Stack: `( STR STR -- )`

21.1.5.8 Instruction endhl

Synopsys:

```
endhl
```

End the current hyperlink.

If no hyperlink is currently being generated, this instruction raises an exception.

Stack: `( -- )`

Exceptions: `PVM_E_GENERIC`

21.1.5.9 Instruction begsc

Synopsys:

```
begsc
```

Begin the styling class whose name is found on the stack. This class will be in effect in subsequent print operations until it is explicitly ended by a 'endsc' instruction.

Stack: `( STR -- )`

21.1.5.10 Instruction endsc

Synopsys:

```
endsc
```

End the styling class whose name is found on the stack. This class should have been previously began by a 'begsc' instruction.

Stack: `( STR -- )`

21.1.6 Format Instructions

In the following instructions the meaning of the argument BASE is the following:

2 - format the number in binary. 8 - format the number in octal. 16 - format the number in hexadecimal. Any other value - format the number in decimal.

21.1.6.1 Instruction formati

Synopsys:

```
formati BITS
```

Given a signed integer and a numeration base in the stack, push the string representation to the stack.

Stack: `( INT INT -- STR )`

21.1.6.2 Instruction formatiu

Synopsys:

```
formatiu BITS
```

Given an unsigned integer and a numeration base in the stack, push the string representation to the stack.

Stack: `( UINT INT -- STR )`

21.1.6.3 Instruction formatl

Synopsys:

```
formatl BITS
```

Given a long and a numeration base in the stack, push the string representation to the stack.

Stack: `( LONG INT -- STR )`

21.1.6.4 Instruction formatlu

Synopsys:

```
formatlu BITS
```

Given an unsigned long and a numeration base in the stack, push the string representation to the stack.

Stack: `( ULONG INT -- STR )`

21.1.7 Main stack manipulation instructions

21.1.7.1 Instruction push

Synopsys:

```
push VAL
```

Push the value given as an argument to the main stack.

Stack: `( -- VAL )`

21.1.7.2 Instruction drop

Synopsys:

```
drop
```

Pop the value at the top of the main stack, and discard it.

Stack: `( VAL -- )`

21.1.7.3 Instruction drop2

Synopsys:

```
drop2
```

Pop the two values at the top of the main stack, and discard them.

Stack: `( VAL VAL -- )`

21.1.7.4 Instruction drop3

Synopsys:

```
drop3
```

Pop the three values at the top of the main stack, and discard them.

Stack: `( VAL VAL VAL -- )`

21.1.7.5 Instruction drop4

Synopsys:

```
drop4
```

Pop the four values at the top of the stack, and discard them.

Stack: `( VAL VAL VAL VAL -- )`

21.1.7.6 Instruction swap

Synopsys:

```
swap
```

Exchange the two elements at the top of the stack.

Stack: `( A B -- B A)`

21.1.7.7 Instruction nip

Synopsys:

```
nip
```

Discard the element at the under top of the main stack.

Stack: `( A B -- B )`

21.1.7.8 Instruction nip2

Synopsys:

```
nip2
```

Discard the two elements at the under top of the main stack.

Stack: `( A B C -- C )`

21.1.7.9 Instruction nip3

Synopsys:

```
nip3
```

Discard the three elements at the under top of the main stack.

Stack: `( A B C D -- D )`

21.1.7.10 Instruction dup

Synopsys:

```
dup
```

Push a copy of the element at the top of the main stack.

Stack: `( A -- A A )`

21.1.7.11 Instruction over

Synopsys:

```
over
```

Push a copy of the element at the under top of the main stack.

Stack: `( A B -- A B A )`

21.1.7.12 Instruction rot

Synopsys:

```
rot
```

Rotate the three elements at the top of the main stack, clock-wise.

Stack: `( A B C -- B C A )`

21.1.7.13 Instruction nrot

Synopsys:

```
nrot
```

Rotate the three elements at the top of the stack, counter clock-wise.

Stack: `( A B C -- C A B )`

21.1.7.14 Instruction tuck

Synopsys:

```
tuck
```

Tuck a copy of the element at the top of the stack down two positions.

Stack: `( A B -- B A B )`

21.1.7.15 Instruction quake

Synopsys:

```
quake
```

Swap the two elements at the under top of the stack.

Stack: `( A B C - B A C )`

21.1.7.16 Instruction revn

Synopsys:

```
revn N
```

Reverse the N elements at the top of the stack.

Stack: `( VAL... -- VAL... )`

21.1.7.17 Instruction pushhi

Synopsys:

```
pushhi VAL
```

Push the high 32 bits of the value passed as an argument to the main stack. This instruction shall be completed with a 'pushlo'.

This instruction is a workaround to a limitation of Jitter.

Stack: `( -- HI32(VAL) )`

21.1.7.18 Instruction pushlo

Synopsys:

```
pushlo VAL
```

Push the low 32 bits of the value passed as an argument to the main stack. This instruction shall be preceded by a 'pushhi'.

This instruction is a workaround to a limitation of Jitter.

Stack: `( -- LO32(VAL) )`

21.1.7.19 Instruction push32

Synopsys:

```
push32 VAL
```

Push the value passed as an argument on the stack. This assumes that the internal representation of VAL doesn't require more than 32-bit.

This instruction is a workaround to a limitation of Jitter.

Stack: `( -- VAL )`

21.1.8 Registers manipulation instructions

21.1.8.1 Instruction pushr

Synopsys:

```
pushr REGNO
```

Push the contents of the register REGNO on the stack.

Stack: `( -- VAL )`

21.1.8.2 Instruction popr

Synopsys:

```
popr REGNO
```

Pop the element at the top of the stack and put it in the register REGNO.

Stack: `( VAL -- )`

21.1.8.3 Instruction setr

Synopsys:

```
setr REGNO
```

Set the element at the top of the stack to the value of the register REGNO.

Stack: `( -- )`

21.1.9 Return stack manipulation instructions

21.1.9.1 Instruction saver

Synopsys:

```
saver REGNO
```

Push the contents of the register REGNO to the return stack.

Stack: `( -- )` ReturnStack: (– VAL)

21.1.9.2 Instruction restorer

Synopsys:

```
restorer REGNO
```

Pop the element at the top of the return stack and put it in the register REGNO.

Stack: `( -- )` ReturnStack: (VAL –)

21.1.9.3 Instruction tor

Synopsys:

```
tor
```

Pop an element from the stack and push it in the return stack.

Stack: `( VAL -- )` ReturnStack: (– VAL)

21.1.9.4 Instruction fromr

Synopsys:

```
fromr
```

Pop an element from the return stack and push it on the stack.

Stack: `( -- VAL)` ReturnStack: (VAL –)

21.1.9.5 Instruction atr

Synopsys:

```
atr
```

Push a copy of the element at the top of the return stack into the stack.

Stack: `( -- VAL )`

21.1.10 Arithmetic instructions

The following instructions assume that both operands have the same size in bits.

21.1.10.1 Instruction addi

Synopsys:

```
addi
```

Push the result of adding the two integers at the top of the stack. If the operation would result in overflow, raise PVM_E_OVERFLOW.

Stack: `( INT INT -- INT INT INT )`

Exceptions: `PVM_E_OVERFLOW`

21.1.10.2 Instruction addiu

Synopsys:

```
addiu
```

Push the result of adding the two unsigned integers at the top of the stack.

Stack: `( UINT UINT -- UINT UINT UINT )` Instruction addl

Push the result of adding the two longs at the top of the stack. If the operation would result in overflow, raise PVM_E_OVERFLOW.

Stack: `( LONG LONG -- LONG LONG LONG )`

Exceptions: `PVM_E_OVERFLOW`

21.1.10.3 Instruction addlu

Synopsys:

```
addlu
```

Push the result of adding the two unsigned longs at the top of the stack.

Stack: `( ULONG ULONG -- ULONG ULONG ULONG )`

21.1.10.4 Instruction subi

Synopsys:

```
subi
```

Push the result of subtracting the two integers at the top of the stack.

Stack: `( INT INT -- INT INT INT )`

21.1.10.5 Instruction subiu

Synopsys:

```
subiu
```

Push the result of subtracting the two unsigned integers at the top of the stack.

Stack: `( UINT UINT -- UINT UINT UINT )`

21.1.10.6 Instruction subl

Synopsys:

```
subl
```

Push the result of subtracting the two longs at the top of the stack.

Stack: `( LONG LONG -- LONG LONG LONG )`

21.1.10.7 Instruction sublu

Synopsys:

```
sublu
```

Push the result of subtracting the two unsigned longs at the top of the stack.

Stack: `( ULONG ULONG -- ULONG ULONG ULONG )`

21.1.10.8 Instruction muli

Synopsys:

```
muli
```

Push the result of multiplying the two integers at the top of the stack.

Stack: `( INT INT -- INT INT INT )`

21.1.10.9 Instruction muliu

Synopsys:

```
muliu
```

Push the result of multiplying the two unsigned integers at the top of the stack.

Stack: `( UINT UINT -- UINT UINT UINT )`

21.1.10.10 Instruction mull

Synopsys:

```
mull
```

Push the result of multiplying the two longs at the top of the stack.

Stack: `( LONG LONG -- LONG LONG LONG )`

21.1.10.11 Instruction mullu

Synopsys:

```
mullu
```

Push the result of multiplying the two unsigned longs at the top of the stack.

Stack: `( ULONG ULONG -- ULONG ULONG ULONG )`

21.1.10.12 Instruction divi

Synopsys:

```
divi
```

Push the result of the integer division of the two integers at the top of the stack. If the denominator is zero, raise PVM_E_DIV_BY_ZERO.

Stack: `( INT INT -- INT INT INT )`

Exceptions: `PVM_E_DIV_BY_ZERO`

21.1.10.13 Instruction diviu

Synopsys:

```
diviu
```

Push the result of the integer division of the two unsigned integers at the top of the stack. If the denominator is zero, raise PVM_E_DIV_BY_ZERO.

Stack: `( UINT UINT -- UINT UINT UINT )`

Exceptions: `PVM_E_DIV_BY_ZERO`

21.1.10.14 Instruction divl

Synopsys:

```
divl
```

Push the result of the integer division of the two longs at the top of the stack. If the denominator is zero, raise PVM_E_DIV_BY_ZERO.

Stack: `( LONG LONG -- LONG LONG LONG )`

Exceptions: `PVM_E_DIV_BY_ZERO`

21.1.10.15 Instruction divlu

Synopsys:

```
divlu
```

Push the result of the integer division of the two unsigned longs at the top of the stack. If the denominator is zero, raise PVM_E_DIV_BY_ZERO.

Stack: `( ULONG ULONG -- ULONG ULONG ULONG )`

Exceptions: `PVM_E_DIV_BY_ZERO`

21.1.10.16 Instruction modi

Synopsys:

```
modi
```

Push the result of the modulus of the two integers at the top of the stack. If the denominator is zero, raise PVM_E_DIV_BY_ZERO.

Stack: `( INT INT -- INT INT INT )`

Exceptions: `PVM_E_DIV_BY_ZERO`

21.1.10.17 Instruction modiu

Synopsys:

```
modiu
```

Push the result of the modulus of the two unsigned integers at the top of the stack. If the denominator is zero, raise PVM_E_DIV_BY_ZERO.

Stack: `( UINT UINT -- UINT UINT UINT )`

Exceptions: `PVM_E_DIV_BY_ZERO`

21.1.10.18 Instruction modl

Synopsys:

```
modl
```

Push the result of the modulus of the two longs at the top of the stack. If the denominator is zero, raise PVM_E_DIV_BY_ZERO.

Stack: `( LONG LONG -- LONG LONG LONG )`

Exceptions: `PVM_E_DIV_BY_ZERO`

21.1.10.19 Instruction modlu

Synopsys:

```
modlu
```

Push the result of the modulus of the two unsigned longs at the top of the stack. If the denominator is zero, raise PVM_E_DIV_BY_ZERO.

Stack: `( ULONG ULONG -- ULONG ULONG ULONG )`

Exceptions: `PVM_E_DIV_BY_ZERO`

21.1.10.20 Instruction negi

Synopsys:

```
negi
```

Push the result of the negation of the integer at the top of the stack.

Stack: `( INT -- INT INT )`

21.1.10.21 Instruction negiu

Synopsys:

```
negiu
```

Push the result of the negation of the unsigned integer at the top of the stack.

Stack: `( UINT -- UINT UINT UINT )`

21.1.10.22 Instruction negl

Synopsys:

```
negl
```

Push the result of the negation of the long at the top of the stack.

Stack: `( LONG -- LONG LONG )`

21.1.10.23 Instruction neglu

Synopsys:

```
neglu
```

Push the result of the negation of the unsigned long at the top of the stack.

Stack: `( ULONG -- ULONG ULONG )`

21.1.10.24 Instruction powi

Synopsys:

```
powi
```

Perform the exponentiation of the integer at the under top of the stack. The exponent is the unsigned integer at the top of the stack. If the exponent is 0, the result is 1.

Stack: `( INT UINT -- INT UINT INT )`

21.1.10.25 Instruction powiu

Synopsys:

```
powiu
```

Perform the exponentiation of the unsigned integer at the under top of the stack. The exponent is the unsigned integer at the top of the stack. If the exponent is 0, the result is 1.

Stack: `( UINT UINT -- UINT UINT UINT )`

21.1.10.26 Instruction powl

Synopsys:

```
powl
```

Perform the exponentiation of the long at the under top of the stack. The exponent is the unsigned integer at the top of the stack. If the exponent is 0, the result is 1.

Stack: `( LONG UINT -- LONG UINT LONG )`

21.1.10.27 Instruction powlu

Synopsys:

```
powlu
```

Perform the exponentiation of the unsigned long at the under top of the stack. The exponent is the unsigned integer at the top of the stack. If the exponent is 0, the result is 1.

Stack: `( ULONG UINT -- ULONG UINT ULONG )`

21.1.11 Relational instructions

21.1.11.1 Instruction eqi

Synopsys:

```
eqi
```

Push 1 on the stack if the two integers at the top of the stack are equal. Otherwise push 0.

Stack: `( INT INT -- INT INT INT )`

21.1.11.2 Instruction eqiu

Synopsys:

```
eqiu
```

Push 1 on the stack if the two unsigned integers at the top of the stack are equal. Otherwise push 0.

Stack: `( UINT UINT -- UINT UINT UINT )`

21.1.11.3 Instruction eql

Synopsys:

```
eql
```

Push 1 on the stack if the two longs at the top of the stack are equal. Otherwise push 0.

Stack: `( LONG LONG -- LONG LONG INT )`

21.1.11.4 Instruction eqlu

Synopsys:

```
eqlu
```

Push 1 on the stack if the two unsigned longs at the top of the stack are equal. Otherwise push 0.

Stack: `( ULONG ULONG -- ULONG ULONG INT )`

21.1.11.5 Instruction eqs

Synopsys:

```
eqs
```

Push 1 on the stack if the two strings at the top of the stack are equal. Otherwise push 0.

Stack: `( STR STR -- STR STR INT )`

21.1.11.6 Instruction nei

Synopsys:

```
nei
```

Push 1 on the stack if the two integers at the top of the stack are not equal. Otherwise push 0.

Stack: `( INT INT -- INT INT INT )`

21.1.11.7 Instruction neiu

Synopsys:

```
neiu
```

Push 1 on the stack if the two unsigned integers at the top of the stack are not equal. Otherwise push 0.

Stack: `( UINT UINT -- UINT UINT INT )`

21.1.11.8 Instruction nel

Synopsys:

```
nel
```

Push 1 on the stack if the two longs at the top of the stack are not equal. Otherwise push 0.

Stack: `( LONG LONG -- LONG LONG INT )`

21.1.11.9 Instruction nelu

Synopsys:

```
nelu
```

Push 1 on the stack if the two unsigned longs at the top of the stack are not equal. Otherwise push 0.

Stack: `( ULONG ULONG -- ULONG ULONG INT )`

21.1.11.10 Instruction nes

Synopsys:

```
nes
```

Push 1 on the stack if the two strings at the top of the stack are not equal. Otherwise push 0.

Stack: `( STR STR -- STR STR INT )`

21.1.11.11 Instruction nn

Synopsys:

```
nn
```

Push 0 on the stack if the value at the top of the stack equals PVM_NULL. Otherwise push 1.

Stack: `( VAL -- VAL INT )`

21.1.11.12 Instruction nnn

Synopsys:

```
nnn
```

Push 1 on the stack if the value at the top of the stack equals PVM_NULL. Otherwise push 0.

Stack: `( VAL -- VAL INT )`

21.1.11.13 Instruction lti

Synopsys:

```
lti
```

Push 1 on the stack if the integer at the under top is less that the integer at the top. Otherwise push 0.

Stack: `( INT INT -- INT INT INT )`

21.1.11.14 Instruction ltiu

Synopsys:

```
ltiu
```

Push 1 on the stack if the unsigned integer at the under top is less that the unsigned integer at the top. Otherwise push 0.

Stack: `( UINT INT -- UINT UINT INT )`

21.1.11.15 Instruction ltl

Synopsys:

```
ltl
```

Push 1 on the stack if the long at the under top is less that the long at the top. Otherwise push 0.

Stack: `( LONG LONG -- LONG LONG INT )`

21.1.11.16 Instruction ltlu

Synopsys:

```
ltlu
```

Push 1 on the stack if the unsigned long at the under top is less that the unsigned long at the top. Otherwise push 0.

Stack: `( ULONG ULONG -- ULONG ULONG INT )`

21.1.11.17 Instruction lei

Synopsys:

```
lei
```

Push 1 on the stack if the integer at the under top is less or equal that the integer at the top. Otherwise push 0.

Stack: `( INT INT -- INT INT INT )`

21.1.11.18 Instruction leiu

Synopsys:

```
leiu
```

Push 1 on the stack if the unsigned integer at the under top is less or equal that the unsigned integer at the top. Otherwise push 0.

Stack: `( UINT UINT -- UINT UINT INT )`

21.1.11.19 Instruction lel

Synopsys:

```
lel
```

Push 1 on the stack if the long at the under top is less or equal that the long at the top. Otherwise push 0.

Stack: `( LONG LONG -- LONG LONG INT )`

21.1.11.20 Instruction lelu

Synopsys:

```
lelu
```

Push 1 on the stack if the unsigned long at the under top is less or equal that the unsigned long at the top. Otherwise push 0.

Stack: `( ULONG ULONG -- ULONG ULONG INT )`

21.1.11.21 Instruction gti

Synopsys:

```
gti
```

Push 1 on the stack if the integer at the under top is greater than the integer at the top. Otherwise push 0.

Stack: `( INT INT -- INT INT INT )`

21.1.11.22 Instruction gtiu

Synopsys:

```
gtiu
```

Push 1 on the stack if the unsigned integer at the under top is greater than the unsigned integer at the top. Otherwise push 0.

Stack: `( UINT UINT -- UINT UINT INT )`

21.1.11.23 Instruction gtl

Synopsys:

```
gtl
```

Push 1 on the stack if the long at the under top is greater than the long at the top. Otherwise push 0.

Stack: `( LONG LONG -- LONG LONG INT )`

21.1.11.24 Instruction gtlu

Synopsys:

```
gtlu
```

Push 1 on the stack if the unsigned long at the under top is greater than the unsigned long at the top. Otherwise push 0.

Stack: `( LONG LONG -- LONG LONG INT )`

21.1.11.25 Instruction gei

Synopsys:

```
gei
```

Push 1 on the stack if the integer at the under top is greater or equal than the integer at the top. Otherwise push 0.

Stack: `( INT INT -- INT INT INT )`

21.1.11.26 Instruction geiu

Synopsys:

```
geiu
```

Push 1 on the stack if the unsigned integer at the under top is greater or equal than the unsigned integer at the top. Otherwise push 0.

Stack: `( UINT UINT -- UINT UINT INT )`

21.1.11.27 Instruction gel

Synopsys:

```
gel
```

Push 1 on the stack if the long at the under top is greater or equal than the long at the top. Otherwise push 0.

Stack: `( LONG LONG -- LONG LONG INT )`

21.1.11.28 Instruction gelu

Synopsys:

```
gelu
```

Push 1 on the stack if the unsigned long at the under top is greater or equal than the unsigned long at the top. Otherwise push 0.

Stack: `( ULONG ULONG -- ULONG ULONG INT )`

21.1.11.29 Instruction lts

Synopsys:

```
lts
```

Push 1 on the stack if the string at the under top is less than the string at the top, in lexicographic order. Otherwise push 0.

Stack: `( STR STR -- STR STR INT )`

21.1.11.30 Instruction gts

Synopsys:

```
gts
```

Push 1 on the stack if the string at the under top is greater than the string at the top, in lexicographic order. Otherwise push 0.

Stack: `( STR STR -- STR STR INT )`

21.1.11.31 Instruction ges

Synopsys:

```
ges
```

Push 1 on the stack if the string at the under top is greater or equal than the string at the top, in lexicographic order. Otherwise push 0.

Stack: `( STR STR -- STR STR INT )`

21.1.11.32 Instruction les

Synopsys:

```
les
```

Push 1 on the stack if the string at the under top is less or equal than the string at the top, in lexicographic order. Otherwise push 0.

Stack: `( STR STR -- STR STR INT )`

21.1.11.33 Instruction eqc

Synopsys:

```
eqc
```

Push 1 on the stack if the two closures at the top of the stack are equal. Otherwise push 0.

Stack: `( CLS CLS -- CLS CLS INT )`

21.1.11.34 Instruction nec

Synopsys:

```
nec
```

Push 1 on the stack if the two closures at the top of the stack are not equal. Otherwise push 0.

Stack: `( CLS CLS -- CLS CLS INT )`

21.1.12 Concatenation instructions

21.1.12.1 Instruction sconc

Synopsys:

```
sconc
```

Push the concatenation of the two strings at the top of the stack.

Stack: `( STR STR -- STR STR STR )`

21.1.13 Logical instructions

21.1.13.1 Instruction and

Synopsys:

```
and
```

Push the logical and of the two elements at the top of the stack.

Stack: `( INT INT -- INT INT INT )`

21.1.13.2 Instruction or

Synopsys:

```
or
```

Push the logical or of the two elements at the top of the stack.

Stack: `( INT INT -- INT INT INT )`

21.1.13.3 Instruction not

Synopsys:

```
not
```

Push the logical not of the element at the top of the stack.

Stack: `( INT -- INT INT )`

21.1.14 Bitwise instructions

21.1.14.1 Instruction bxori

Synopsys:

```
bxori
```

Push the bitwise exclusive or of the two integers at the top of the stack.

Stack: `( INT INT -- INT INT INT )`

21.1.14.2 Instruction bxoriu

Synopsys:

```
bxoriu
```

Push the bitwise exclusive or of the two unsigned integers at the top of the stack.

Stack: `( UINT UINT -- UINT UINT UINT )`

21.1.14.3 Instruction bxorl

Synopsys:

```
bxorl
```

Push the bitwise exclusive or of the two longs at the top of the stack.

Stack: `( LONG LONG -- LONG LONG LONG )`

21.1.14.4 Instruction bxorlu

Synopsys:

```
bxorlu
```

Push the bitwise exclusive or of the two unsigned longs at the top of the stack.

Stack: `( ULONG ULONG -- ULONG ULONG ULONG )`

21.1.14.5 Instruction bori

Synopsys:

```
bori
```

Push the bitwise or of the two integers at the top of the stack.

Stack: `( INT INT -- INT INT INT )`

21.1.14.6 Instruction boriu

Synopsys:

```
boriu
```

Push the bitwise or of the two unsigned integers at the top of the stack.

Stack: `( UINT UINT -- UINT UINT UINT )`

21.1.14.7 Instruction borl

Synopsys:

```
borl
```

Push the bitwise or of the two longs at the top of the stack.

Stack: `( LONG LONG -- LONG LONG LONG )`

21.1.14.8 Instruction borlu

Synopsys:

```
borlu
```

Push the bitwise or of the two longs at the top of the stack.

Stack: `( ULONG ULONG -- ULONG ULONG ULONG )`

21.1.14.9 Instruction bandi

Synopsys:

```
bandi
```

Push the bitwise and of the two integers at the top of the stack.

Stack: `( INT INT -- INT INT INT )`

21.1.14.10 Instruction bandiu

Synopsys:

```
bandiu
```

Push the bitwise and of the two unsigned integers at the top of the stack.

Stack: `( UINT UINT -- UINT UINT UINT )`

21.1.14.11 Instruction bandl

Synopsys:

```
bandl
```

Push the bitwise and of the two longs at the top of the stack.

Stack: `( LONG LONG -- LONG LONG LONG )`

21.1.14.12 Instruction bandlu

Synopsys:

```
bandlu
```

Push the bitwise and of the two unsigned longs at the top of the stack.

Stack: `( ULONG ULONG -- ULONG ULONG ULONG )`

21.1.14.13 Instruction bnoti

Synopsys:

```
bnoti
```

Push the bitwise not of the integer at the top of the stack.

Stack: `( INT -- INT INT INT )`

21.1.14.14 Instruction bnotiu

Synopsys:

```
bnotiu
```

Push the bitwise not of the unsigned integer at the top of the stack.

Stack: `( UINT -- UINT UINT )`

21.1.14.15 Instruction bnotl

Synopsys:

```
bnotl
```

Push the bitwise not of the long at the top of the stack.

Stack: `( LONG -- LONG LONG )`

21.1.14.16 Instruction bnotlu

Synopsys:

```
bnotlu
```

Push the bitwise not of the unsigned long at the top of the stack.

Stack: `( ULONG -- ULONG ULONG )`

21.1.15 Shift instructions

21.1.15.1 Instruction bsli

Synopsys:

```
bsli
```

Left-shift the integer at the under top of the stack the number of bits indicated by the unsigned int at the top of the stack.

If the bit count is equal or bigger than the size of the left operand, then raise PVM_E_OUT_OF_BOUNDS.

Stack: `( INT UINT -- INT UINT INT )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.15.2 Instruction bsliu

Synopsys:

```
bsliu
```

Left-shift the unsigned integer at the under top of the stack the number of bits indicated by the unsigned int at the top of the stack.

If the bit count is equal or bigger than the size of the left operand, then raise PVM_E_OUT_OF_BOUNDS.

Stack: `( UINT UINT -- UINT UINT UINT )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.15.3 Instruction bsll

Synopsys:

```
bsll
```

Left-shift the long at the under top of the stack the number of bits indicated by the unsigned int at the top of the stack.

If the bit count is equal or bigger than the size of the left operand, then raise PVM_E_OUT_OF_BOUNDS.

Stack: `( LONG UINT -- LONG UINT LONG )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.15.4 Instruction bsllu

Synopsys:

```
bsllu
```

Left-shift the unsigned long at the under top of the stack the number of bits indicated by the unsigned int at the top of the stack.

If the bit count is equal or bigger than the size of the left operand, then raise PVM_E_OUT_OF_BOUNDS.

Stack: `( ULONG UINT -- ULONG UINT ULONG )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.15.5 Instruction bsri

Synopsys:

```
bsri
```

Right-shift the integer at the under top of the stack the number of tis indicated by the unsigned int at the top of the stack.

Stack: `( INT UINT -- INT UINT INT )`

21.1.15.6 Instruction bsriu

Synopsys:

```
bsriu
```

Right-shift the unsigned integer at the under top of the stack the number of tis indicated by the unsigned int at the top of the stack.

Stack: `( UINT UINT -- UINT UINT UINT )`

21.1.15.7 Instruction bsrl

Synopsys:

```
bsrl
```

Right-shift the long at the under top of the stack the number of tis indicated by the unsigned int at the top of the stack.

Stack: `( LONG UINT -- LONG UINT LONG )`

21.1.15.8 Instruction bsrlu

Synopsys:

```
bsrlu
```

Right-shift the unsigned long at the under top of the stack the number of tis indicated by the unsigned int at the top of the stack.

Stack: `( ULONG UINT -- ULONG UINT ULONG )`

21.1.16 Compare-and-swap instructions

21.1.16.1 Instruction swapgti

Synopsys:

```
swapgti
```

Swap the two integers at the top of the stack if the element at the under-top is greater than the element at the top.

Stack: `( INT INT -- INT INT )`

21.1.16.2 Instruction swapgtiu

Synopsys:

```
swapgtiu
```

Swap the two unsigned integers at the top of the stack if the element at the under-top is greater than the element at the top.

Stack: `( UINT UINT -- UINT UINT )`

21.1.16.3 Instruction swapgtl

Synopsys:

```
swapgtl
```

Swap the two longs at the top of the stack if the element at the under-top is greater than the element at the top.

Stack: `( LONG LONG -- LONG LONG )`

21.1.16.4 Instruction swapgtlu

Synopsys:

```
swapgtlu
```

Swap the two unsigned longs at the top of the stack if the element at the under-top is greater than the element at the top.

Stack: `( ULONG ULONG -- ULONG ULONG )`

21.1.17 Branch instructions

21.1.17.1 Instruction ba

Synopsys:

```
ba LABEL
```

Branch unconditionally to the given LABEL.

Stack: `( -- )`

21.1.17.2 Instruction bn

Synopsys:

```
bn LABEL
```

Branch to the given LABEL if the value at the top of the stack is PVM_NULL.

Stack: `( VAL -- VAL )`

21.1.17.3 Instruction bnn

Synopsys:

```
bnn LABEL
```

Branch to the given LABEL if the value at the top of the stack is not PVM_NULL.
Stack: `( VAL -- VAL )`

21.1.17.4 Instruction bzi

Synopsys:

```
bzi LABEL
```

Branch to the given LABEL if the integer at the top of the stack is zero.
Stack: `( INT -- INT )`

21.1.17.5 Instruction bziu

Synopsys:

```
bziu LABEL
```

Branch to the given LABEL if the unsigned integer at the top of the stack is zero.
Stack: `( UINT -- UINT )`

21.1.17.6 Instruction bzl

Synopsys:

```
bzl LABEL
```

Branch to the given LABEL if the long at the top of the stack is zero.
Stack: `( LONG -- LONG )`

21.1.17.7 Instruction bzlu

Synopsys:

```
bzlu LABEL
```

Branch to the given LABEL if the unsigned long at the top of the stack is zero.
Stack: `( ULONG -- ULONG )`

21.1.17.8 Instruction bnzi

Synopsys:

```
bnzi LABEL
```

Branch to the given LABEL if the integer at the top of the stack is nonzero.
Stack: `( INT -- INT )`

21.1.17.9 Instruction bnziu

Synopsys:

```
bnziu LABEL
```

Branch to the given LABEL if the unsigned integer at the top of the stack is nonzero.
Stack: `( UINT -- UINT )`

21.1.17.10 Instruction bnzl

Synopsys:

```
bnzl LABEL
```

Branch to the given LABEL if the long at the top of the stack is nonzero.
Stack: `( LONG -- LONG )`

21.1.17.11 Instruction bnzlu

Synopsys:

```
bnzlu LABEL
```

Branch to the given LABEL if the unsigned long at the top of the stack is nonzero.

Stack: `( ULONG -- ULONG )`

21.1.18 Conversion instructions

21.1.18.1 Instruction ctos

Synopsys:

```
ctos
```

Convert the character encoded as an unsigned integer at the top of the stack to a string that contains just that character.

Stack: `( UINT -- UINT STR )`

21.1.18.2 Instruction itoi

Synopsys:

```
itoi NBITS
```

Convert the integer at the top of the stack to an integer featuring NBITS bits.

NBITS can be any number from 1 to 32.

Stack: `( INT -- INT INT )`

21.1.18.3 Instruction itoiu

Synopsys:

```
itoiu NBITS
```

Convert the integer at the top of the stack to an unsigned integer featuring NBITS bits.

NBITS can be any number from 1 to 32.

Stack: `( INT -- INT UINT )`

21.1.18.4 Instruction itol

Synopsys:

```
itol NBITS
```

Convert the integer at the top of the stack to a long featuring NBITS bits.

NBITS can be any number from 1 to 64.

Stack: `( INT -- INT LONG )`

21.1.18.5 Instruction itolu

Synopsys:

```
itolu NBITS
```

Convert the integer at the top of the stack to an unsigned long featuring NBITS bits.

NBITS can be any number from 1 to 64.

Stack: `( INT -- INT ULONG )`

21.1.18.6 Instruction iutoi

Synopsys:

```
iutoi NBITS
```

Convert the unsigned integer at the top of the stack to an integer featuring NBITS bits.

NBITS can be any number from 1 to 32.

Stack: `( UINT -- UINT INT )`

21.1.18.7 Instruction iutoiu

Synopsys:

```
iutoiu NBITS
```

Convert the unsigned integer at the top of the stack to an unsigned integer featuring NBITS bits.

NBITS can be any number from 1 to 32.

Stack: `( UINT -- UINT UINT )`

21.1.18.8 Instruction iutol

Synopsys:

```
iutol NBITS
```

Convert the unsigned integer at the top of the stack to a long featuring NBITS bits.

NBITS can be any number from 1 to 64.

Stack: `( UINT -- UINT LONG )`

21.1.18.9 Instruction iutolu

Synopsys:

```
iutolu NBITS
```

Convert the unsigned integer at the top of the stack to an unsigned long featuring NBITS bits.

NBITS can be any number from 1 to 64.

Stack: `( UINT -- UINT ULONG )`

21.1.18.10 Instruction ltoi

Synopsys:

```
ltoi NBITS
```

Convert the long at the top of the stack to an integer featuring NBITS bits.

NBITS can be any number from 1 to 32.

Stack: `( LONG -- LONG INT )`

21.1.18.11 Instruction ltoiu

Synopsys:

```
ltoiu NBITS
```

Convert the long at the top of the stack to an unsigned integer featuring NBITS bits.

NBITS can be any number from 1 to 32.

Stack: `( LONG -- LONG UINT )`

21.1.18.12 Instruction ltol

Synopsys:

```
ltol NBITS
```

Convert the long at the top of the stack to a long featuring NBITS bits.

NBITS can be any number from 1 to 64.

Stack: `( LONG -- LONG LONG )`

21.1.18.13 Instruction ltolu

Synopsys:

```
ltolu NBITS
```

Convert the long at the top of the stack to an unsigned long featuring NBITS bits.

NBITS can be any number from 1 to 64.

Stack: `( LONG -- LONG ULONG )`

21.1.18.14 Instruction lutoi

Synopsys:

```
lutoi NBITS
```

Convert the unsigned long at the top of the stack to an integer featuring NBITS bits.

NBITS can be any number from 1 to 32.

Stack: `( ULONG -- ULONG INT )`

21.1.18.15 Instruction lutoiu

Synopsys:

```
lutoiu NBITS
```

Convert the unsigned long at the top of the stack to an unsigned integer featuring NBITS bits.

NBITS can be any number from 1 to 32.

Stack: `( ULONG -- ULONG UINT )`

21.1.18.16 Instruction lutol

Synopsys:

```
lutol NBITS
```

Convert the unsigned long at the top of the stack to a long featuring NBITS bits.

Stack: `( ULONG -- ULONG LONG )`

21.1.18.17 Instruction lutolu

Synopsys:

```
lutolu NBITS
```

Convert the unsigned long at the top of the stack to an unsigned long featuring NBITS bits.

Stack: `( ULONG -- ULONG ULONG )`

21.1.19 String instructions

21.1.19.1 Instruction strref

Synopsys:

```
strref
```

Given a string and an unsigned long at the top of the stack, push an unsigned integer with the code of the character that occupies that position in the string, on the stack.

The index is zero-based. If it is less than 0 or exceeds the length of the string, then PVM_E_OUT_OF_BOUNDS is raised.

Stack: `( STR ULONG -- STR ULONG UINT )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.19.2 Instruction strset

Synopsys:

```
strset
```

Given a string STR, an index FROM and a string NEWSTR, copy the content of NEWSTR to STR at index FROM.

Index is zero-based.

If FROM >= the size of the string, or if FROM+len(NEWSTR) > the size of the string, raise the PVM_E_OUT_OF_BOUNDS exception.

Stack: `( STR ULONG NEWSTR -- STR )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.19.3 Instruction substr

Synopsys:

```
substr
```

Given a string and two indices FROM and TO conforming a semi-open interval [FROM,TO), push the substring enclosed by that interval.

Both indexes are zero-based.

If FROM >= the size of the string, or if TO > the size of the string, or if FROM >= TO, raise the PVM_E_OUT_OF_BOUNDS exception.

Stack: `( STR ULONG(from) ULONG(to) -- STR ULONG(from) ULONG(to) STR )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.19.4 Instruction muls

Synopsys:

```
muls
```

Given a string and an unsigned long on the stack, push a new string value whose value is the concatenation of the argument string applied to itself as many times as the unsigned long. If the second argument to muls is 0 then the result of the operation is the empty string.

Stack: `( STR ULONG -- STR ULONG STR )`

21.1.19.5 Instruction sprops

Synopsys:

```
sprops
```

Given a string STR, set the styling class of the substring with length LEN from index IDX to CLASS.

Stack: `( STR IDX LEN CLASS -- STR )`

21.1.19.6 Instruction sproph

Synopsys:

```
sproph
```

Given a string STR, set the hyperlink property (which characterized by a URL and an ID) of the substring with length LEN from index IDX.

Stack: `( STR IDX LEN URL ID -- STR )`

21.1.19.7 Instruction spropc

Synopsys:

```
spropc
```

Given a string on the stack, push the copy of the string with all properties cleared.

Stack: `( STR -- STR )`

21.1.20 Array instructions

21.1.20.1 Instruction mka

Synopsys:

```
mka
```

Make a new empty array value.

TYP is the type of the new array.

NELEM is a hint on how many elements to use to initialize the array value. This is to avoid allocating memory that will never be used. Use 0UL when the number of elements in the array are not known in advance; this will make the PVM to choose a reasonable default.

Stack: `( TYP ULONG(nelem) -- ARR )`

21.1.20.2 Instruction ains

Synopsys:

```
ains
```

Insert a new element VAL, at the end of the array ARR, making it grow.

If IDX is less than the current size of the array, the value is stored in the referred argument.

If IDX is equal or bigger than the current size of the array, the same element is replicated in the previous elements.

Examples:

a = [a1, a2, a3]

[a1, a2, a3] 2 VAL ains -> INVAL exception [a1, a2, a3] 3 VAL ains -> [a1, a2, a3, VAL] [a1, a2, a3] 5 VAL ains -> [a1, a2, a3, VAL, VAL, VAL]

Stack: `( ARR IDX VAL -- ARR )`

Exceptions: `PVM_E_INVAL`

21.1.20.3 Instruction arem

Synopsys:

```
arem
```

Remove an element from an array at the specified index, making it schrink.

If IDX doesn't correspond to an element in the array, raise PVM_E_OUT_OF_BOUNDS. This always happens if the array is empty.

Stack: `( ARR IDX -- ARR )` Exception: PVM_E_OUT_OF_BOUNDS

21.1.20.4 Instruction aset

Synopsys:

```
aset
```

Set the value with index ULONG in the array ARR to have the value VAL.

If the specified index exceeds the capability of the array, then PVM_E_OUT_OF_BOUNDS is raised. If the array is bounded by size and the new value makes the total size of the array to change, then PVM_E_CONV is raised.

Stack: `( ARR ULONG VAL -- ARR )`

Exceptions: `PVM_E_CONV, PVM_E_OUT_OF_BOUNDS`

21.1.20.5 Instruction aref

Synopsys:

```
aref
```

Given an array ARR and an index ULONG, push the element of the array occupying that position on the stack.

If the provided index is out of bounds, then raise PVM_E_OUT_OF_BOUNDS.

Stack: `( ARR ULONG -- ARR ULONG VAL )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.20.6 Instruction arefo

Synopsys:

```
arefo
```

Given an array ARR and an index ULONG, push the offset of the element occupying that position in the array.

If the provided index is out of bounds, then raise PVM_E_OUT_OF_BOUNDS.

Stack: `( ARR ULONG -- ARR ULONG OFF )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.20.7 Instruction asettb

Synopsys:

```
asettb
```

Given an array ARR and a closure BOUND, set the later as the array's bounder function. This is a function that, once executed with no arguments, returns the size of the array.

Stack: `( ARR BOUND -- ARR )`

21.1.21 Struct instructions

21.1.21.1 Instruction mksct

Synopsys:

```
mksct
```

Given an offset, a list of fields, a list of methods and a struct type, create a struct value and push it on the stack.

Each field is specified as a triplet [OFF STR VAL] where OFF is the offset of field, STR the name of the field or PVM_NULL if the field is anonymous, and VAL is a value.

Each method is specified as a tuple [STR VAL] where STR is the name of the method and VAL is the closure value corresponding to the method.

Stack: `( OFF [OFF STR VAL]... [STR VAL]... ULONG ULONG TYP -- SCT )`

21.1.21.2 Instruction sset

Synopsys:

```
sset
```

Given a struct, a field name and a value, replace the value of the referred struct field with the given value. If the struct does not have a field with the given name, then raise PVM_E_ELEM.

Stack: `( SCT STR VAL -- SCT )`

Exceptions: `PVM_E_ELEM`

21.1.21.3 Instruction sseti

Synopsys:

```
sseti
```

Given a struct, a field index and a value, replace the value of the referred struct field with the given value. If the given index does not refer to a struct field, then raise PVM_E_ELEM.

Stack: `(SCT IDX VAL -- SCT)`

Exceptions: `PVM_E_ELEM`

21.1.21.4 Instruction sref

Synopsys:

```
sref
```

Given a struct and a field name, push the value contained in the referred struct field on the stack. If the struct does not have a field with the given name, or if the field is absent from the struct value then raise PVM_E_ELEM.

Stack: `( SCT STR -- SCT STR VAL )`

Exceptions: `PVM_E_ELEM`

21.1.21.5 Instruction srefo

Synopsys:

```
srefo
```

Given a struct and a field name, push the bit-offset of the referred field on the stack. If the struct does not have a field with the given name, or if the field is absent from the struct value then raise PVM_E_ELEM.

Stack: `( SCT STR -- SCT STR BOFF )`

Exceptions: `PVM_E_ELEM`

21.1.21.6 Instruction srefmnt

Synopsys:

```
srefmnt
```

Given a struct and a method name, push the closure value corresponding to that method on the stack. If the struct does not have a method with the given name then push PVM_NULL.

Stack: `( SCT STR - SCT STR CLS )`

21.1.21.7 Instruction srefnt

Synopsys:

```
srefnt
```

Given a struct and a field name, push the value contained in the struct field on the stack. If the struct does not have a field with the given name, or if the field is absent from the struct value then push PVM_NULL.

Stack: `( SCT STR -- SCT STR VAL )`

21.1.21.8 Instruction srefi

Synopsys:

```
srefi
```

Given a struct and an index, push the value of the field occupying the position specified by the index in the given struct. If the struct doesn't have that many fields, raise PVM_E_OUT_OF_BOUNDS.

Stack: `( SCT ULONG -- SCT ULONG VAL )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.21.9 Instruction srefia

Synopsys:

```
srefia
```

Given a struct and an index, push 1 if the field occupying the position specified by the index in the given struct is absent. Push 0 otherwise. If the struct doesn't have that many fields, raise PVM_E_OUT_OF_BOUNDS.

Stack: `( SCT ULONG -- SCT ULONG INT )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.21.10 Instruction srefio

Synopsys:

```
srefio
```

Given a struct and an index, push the offset of the field occupying the position specified by the index in the given struct. If the struct doesn't have that many fields, raise PVM_E_OUT_OF_BOUNDS.

Stack: `( SCT ULONG -- SCT ULONG BOFF )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.21.11 Instruction smodi

Synopsys:

```
smodi
```

Given a struct and an index, push the modified flags of the field occupying the position specified by the index in the given struct. If the struct doesn't have that many fields, raise PVM_E_OUT_OF_BOUNDS.

Stack: `( SCT ULONG -- SCT ULONG BOOL )`

Exceptions: `PVM_E_OUT_OF_BOUNDS`

21.1.22 Offset Instructions

21.1.22.1 Instruction mko

Synopsys:

```
mko
```

Given an integral magnitude VAL and an unit expressed in an ULONG, make an offset value and push it on the stack.

Stack: `( VAL ULONG -- OFF )`

21.1.22.2 Instruction ogetm

Synopsys:

```
ogetm
```

Given an offset OFF, push its magnitude on the stack.

Stack: `( OFF -- OFF VAL )`

21.1.22.3 Instruction osetm

Synopsys:

```
osetm
```

Given an offset OFF and an integral value VAL, make it the offset's magnitude.

Stack: `( OFF VAL -- OFF )`

21.1.22.4 Instruction ogetu

Synopsys:

```
ogetu
```

Given an offset OFF, push its unit on the stack.

Stack: `( OFF -- OFF ULONG )`

21.1.22.5 Instruction ogetbt

Synopsys:

```
ogetbt
```

Given an offset OFF, push its base type on the stack.

Stack: `( OFF -- OFF TYP )`

21.1.23 Instructions to handle mapped values

21.1.23.1 Instruction mm

Synopsys:

```
mm
```

Given a value, push 1 on the stack if the value is mapped. Push 0 otherwise.

Stack: `( VAL -- VAL INT )`

21.1.23.2 Instruction map

Synopsys:

```
map
```

Given a value, mark it as as mapped. If the value can't be mapped then PVM_E_INVAL is raised.

Stack: `( VAL -- VAL )`

Exceptions: `PVM_E_INVAL`

21.1.23.3 Instruction unmap

Synopsys:

```
unmap
```

Given a value, mark it as as not mapped. If the value can't be mapped then this is a no-operation.

Stack: `( VAL -- VAL )`

21.1.23.4 Instruction reloc

Synopsys:

```
reloc
```

Given a value, a IO space expressed in an ulong, and a bit-offset expressed in an ulong, relocate the value to the given bit-offset at the given IO space.

If the given value is not map-able then raise PVM_E_INVAL.

Stack: `( VAL ULONG ULONG -- VAL ULONG ULONG )`

Exceptions: `PVM_E_INVAL`

21.1.23.5 Instruction ureloc

Synopsys:

```
ureloc
```

Given a value, undo the last reloc performed on the value.

If the given value is not map-able then raise PVM_E_INVAL.

Stack: `( VAL -- VAL )`

Exceptions: `PVM_E_INVAL`

21.1.23.6 Instruction mgets

Synopsys:

```
mgets
```

Given a value, push a boolean indicating whether the value is strict. If the given value is not map-able then push false, i.e. 0.

Stack: `( VAL -- VAL INT )`

21.1.23.7 Instruction msets

Synopsys:

```
msets
```

Given a value and a boolean, set the strictness of the value to the given boolean. If the value is not map-able this is a no-operation.

Stack: `( VAL INT -- VAL )`

21.1.23.8 Instruction mgeto

Synopsys:

```
mgeto
```

Given a map-able value, push its bit-offset on the stack as an unsigned long. If the given value is not map-able then push PVM_NULL.

Stack: `( VAL -- VAL ULONG )`

21.1.23.9 Instruction mseto

Synopsys:

```
mseto
```

Given a map-able value an a bit-offset, set its offset to the value. If the given value is not map-able, then the offset is ignored.

Stack: `( VAL ULONG -- VAL )`

21.1.23.10 Instruction mgetios

Synopsys:

```
mgetios
```

Given a map-able value, push its associated IO space on the stack. If the given value is not map-able, then push PVM_NULL.

Stack: `( VAL -- VAL INT )`

21.1.23.11 Instruction msetios

Synopsys:

```
msetios
```

Given a map-able value and an IOS descriptor, set it as its associated IO space. If the IOS descriptor is PVM_NULL then it uses the current IO space. If the given value is not map-able then the IO space is ignored.

Stack: `( VAL INT -- VAL )`

21.1.23.12 Instruction mgetm

Synopsys:

```
mgetm
```

Given a map-able value, push its mapper closure on the stack. If the given value is not map-able, then push PVM_NULL.

Stack: `( VAL -- VAL CLS )`

21.1.23.13 Instruction msetm

Synopsys:

```
msetm
```

Given a map-able value and a closure, set it as its mapper. If the given value is not map-able then the closure is ignored.

Stack: `( VAL CLS -- VAL )`

21.1.23.14 Instruction mgetw

Synopsys:

```
mgetw
```

Given a map-able value, push its writer closure on the stack. If the given value is not map-able, then push PVM_NULL.

Stack: `( VAL -- VAL CLS )`

21.1.23.15 Instruction msetw

Synopsys:

```
msetw
```

Given a map-able value and a closure, set it as its writer. If the given value is not map-able then the closure is ignored.

Stack: `( VAL CLS -- VAL )`

21.1.23.16 Instruction mgetsel

Synopsys:

```
mgetsel
```

Given a map-able value in the TOS, push the number of elements to which its mapping is bounded to. If the value is not mapped, or if it is not bounded by number of elements, push PVM_NULL.

Note that only array values can have mappings bounded by number of elements.

Stack: `( VAL -- VAL ULONG )`

21.1.23.17 Instruction msetsel

Synopsys:

```
msetsel
```

Given a map-able value and an unsigned long, set it as the mapping bound by number of elements. If the value is not map-able the unsigned long is ignored.

Note that only array values can have mappings bounded by number of elements.

Stack: `( VAL ULONG -- VAL )`

21.1.23.18 Instruction mgetsiz

Synopsys:

```
mgetsiz
```

Given a map-able value in the TOS, push its mapping size-bound as a bit-offset. If the value is not map-able, or if it is not bounded by size, push PVM_NULL.

Note that only array values can have mappings bounded by size.

Stack: `( VAL -- VAL ULONG )`

21.1.23.19 Instruction msetsiz

Synopsys:

```
msetsiz
```

Given a map-able value and a bit-offset, set it as the mapping size-bound. If the value is not map-able, the bit-offset is ignored.

Note that only array values can have mappings bounded by size.

Stack: `( VAL ULONG -- VAL )`

21.1.24 Type related instructions

21.1.24.1 Instruction isa

Synopsys:

```
isa
```

Given a value and a type, push 1 on the stack if the value is of the given type. Push 0 otherwise.

Stack: `( VAL TYPE -- TYPE VAL INT )`

21.1.24.2 Instruction typof

Synopsys:

```
typof
```

Given a value that is not itself a type, push its type on the stack. Given a value that is a type, push the value on the stack.

Stack: `( VAL -- VAL TYPE )`

21.1.24.3 Instruction tyisc

Synopsys:

```
tyisc
```

Given a value, push 1 on the stack if it is a closure. Push 0 otherwise.

Stack: `( VAL -- VAL INT )`

21.1.24.4 Instruction tyissct

Synopsys:

```
tyissct
```

Given a value, push 1 on the stack if it is a struct. Push 0 otherwise.

Stack: `( VAL -- VAL INT )`

21.1.24.5 Instruction mktyv

Synopsys:

```
mktyv
```

Build a "void" type and push it on the stack.

Stack: `( -- TYPE )`

21.1.24.6 Instruction mktyany

Synopsys:

```
mktyany
```

Build an "any" type and push it on the stack.

Stack: `( -- TYPE )`

21.1.24.7 Instruction mktyi

Synopsys:

```
mktyi
```

Given an unsigned long denoting a bit width, and an unsigned int denoting signedness (0 is unsigned, 1 is signed), build a an integral type with these features and push it on the stack.

Stack: `( ULONG UINT -- TYPE )`

21.1.24.8 Instruction mktys

Synopsys:

```
mktys
```

Push a string type on the stack.

Stack: `( -- TYPE )`

21.1.24.9 Instruction mktyo

Synopsys:

```
mktyo
```

Given a base integral type and an integer denoting an offset unit (multiple of the base unit) construct an offset type having these features, and push it on the stack.

Stack: `( TYPE INT -- TYPE )` define F(res, a, b) \ undef F

21.1.24.10 Instruction mktya

Synopsys:

```
mktya
```

Given an elements type and an unsigned long denoting a length, build an array type having these features and push it on the stack. If the type array is unbounded then length is PVM_NULL.

Stack: `( TYPE (ULONG|NULL) -- TYPE )`

21.1.24.11 Instruction tyagett

Synopsys:

```
tyagett
```

Given an array type, push the type of its elements on the stack.

Stack: `( TYPE -- TYPE TYPE )`

21.1.24.12 Instruction tyagetb

Synopsys:

```
tyagetb
```

Given an array type, push its bound on the stack.

Stack: `( TYPE -- TYPE (ULONG|NULL) )`

21.1.24.13 Instruction mktyc

Synopsys:

```
mktyc
```

Given a list of argument types, a return type and a number of arguments, build a closure type and push it on the stack.

Stack: `( TYPE... TYPE ULONG -- TYPE )`

21.1.24.14 Instruction mktysct

Synopsys:

```
mktysct
```

Given a list of field descriptors, a number of fields and a struct type name, build a struct type and push it on the stack.

Each field descriptor has the form [STRING TYPE] and contains the name of the field and its type.

Stack: `( [STRING TYPE]... ULONG STR -- TYPE )`

21.1.24.15 Instruction tysctn

Synopsys:

```
tysctn
```

Given a struct type, push its name to the stack. If the struct type is not named push PVM_NULL.

Stack: `( SCT -- SCT STR )`

21.1.25 IO instructions

21.1.25.1 Instruction write

Synopsys:

```
write
```

If the value at the TOS is mapped, then write it to its associated IO space. Otherwise, this is a no-op.

Stack: `( VAL -- VAL )`

Exceptions: `PVM_E_IOS_FULL, PVM_E_CONSTRAINT_ERROR`

21.1.25.2 Instruction peeki

Synopsys:

```
peeki NENC,ENDIAN,BITS
```

Given an IOS descriptor and a bit-offset, peek an integer value of width BITS bits. The negative encoding and endianness to be used are specified in the instruction arguments.

Stack: `( INT ULONG -- INT )`

21.1.25.3 Instruction peekiu

Synopsys:

```
peekiu ENDIAN,BITS
```

Given an IOS descriptor and a bit-offset, peek an unsigned integer value of width BITS bits. The endianness to be used is specified in the instruction arguments.

Stack: `( INT ULONG -- INT )`

21.1.25.4 Instruction peekl

Synopsys:

```
peekl NENC,ENDIAN,BITS
```

Given an IOS descriptor and a bit-offset, peek a long value of width BITS bits. The negative encoding and endianness to be used are specified in the instruction arguments.

Stack: `( INT ULONG -- LONG )`

21.1.25.5 Instruction peeklu

Synopsys:

```
peeklu ENDIAN,BITS
```

Given an IOS descriptor and a bit-offset, peek an unsigned long value of width BITS bits. The endianness to be used is specified in the instruction arguments.

Stack: `( INT ULONG -- ULONG )`

21.1.25.6 Instruction peekdi

Synopsys:

```
peekdi BITS
```

Given an IOS descriptor and a bit-offset, peek an integer value of width BITS bits. Use the default endianness and negative encoding.

Stack: `( INT ULONG -- INT )`

21.1.25.7 Instruction peekdiu

Synopsys:

```
peekdiu BITS
```

Given an IOS descriptor and a bit-offset, peek an unsigned integer value of width BITS bits. Use the default endianness.

Stack: `( INT ULONG -- UINT )`

21.1.25.8 Instruction peekdl

Synopsys:

```
peekdl BITS
```

Given an IOS descriptor and a bit-offset, peek a long value of width BITS bits. Use the default endianness and negative encoding.

Stack: `( INT ULONG -- LONG )`

21.1.25.9 Instruction peekdlu

Synopsys:

```
peekdlu BITS
```

Given an IOS descriptor and a bit-offset, peek an unsigned long value of width BITS bits. Use the default endianness.

Stack: `( INT ULONG -- ULONG )`

21.1.25.10 Instruction pokei

Synopsys:

```
pokei NENC,ENDIAN,BITS
```

Given an IOS descriptor, a bit-offset and an integer value of BITS bits, poke it. Use the negative encoding and endianness specified in the instruction arguments.

Stack: `( INT ULONG INT -- )`

21.1.25.11 Instruction pokeiu

Synopsys:

```
pokeiu ENDIAN,BITS
```

Given an IOS descriptor, a bit-offset and an unsigned integer value of BITS bits, poke it. Use the endianness specified in the instruction arguments.

Stack: `( INT ULONG INT -- )`

21.1.25.12 Instruction pokel

Synopsys:

```
pokel NENC,ENDIAN,BITS
```

Given an IOS descriptor, a bit-offset and a long value of BITS bits, poke it. Use the negative encoding and endianness specified in the instruction arguments.

Stack: `( INT ULONG LONG -- )`

21.1.25.13 Instruction pokelu

Synopsys:

```
pokelu ENDIAN,BITS
```

Given an IOS descriptor, a bit-offset and an unsigned long value of BITS bits, poke it. Use the endianness specified in the instruction arguments.

Stack: `( INT ULONG ULONG -- )`

21.1.25.14 Instruction pokedi

Synopsys:

```
pokedi BITS
```

Given an IOS descriptor, a bit-offset and an integer of BITS bits, poke it. Use the default negative encoding and endianness.

Stack: `( INT ULONG INT -- )`

21.1.25.15 Instruction pokediu

Synopsys:

```
pokediu BITS
```

Given an IOS descriptor, a bit-offset and an unsigned integer of BITS bits, poke it. Use the default endianness.

Stack: `( INT ULONG UINT -- )`

21.1.25.16 Instruction pokedl

Synopsys:

```
pokedl BITS
```

Given an IOS descriptor, a bit-offset and a long of BITS bits, poke it. Use the default negative encoding and endianness.

Stack: `( INT ULONG LONG -- )`

21.1.25.17 Instruction pokedlu

Synopsys:

```
pokedlu BITS
```

Given an IOS descriptor, a bit-offset and an unsigned long of BITS bits, poke it. Use the default endianness.

Stack: `( INT ULONG ULONG -- )`

21.1.25.18 Instruction peeks

Synopsys:

```
peeks
```

Given an IOS descriptor and a bit-offset, peek a string.

Stack: `( INT ULONG -- STR )`

21.1.25.19 Instruction pokes

Synopsys:

```
pokes
```

Given an IOS descriptor, a bit-offset and a string, poke it.

Stack: `( INT ULONG STR -- )`

21.1.26 Exceptions handling instructions

21.1.26.1 Instruction pushe

Synopsys:

```
pushe LABEL
```

Given an Exception struct in the stack, push a handler for it on the exceptions stack. An exception code 0 means any exception.

Stack: `( INT -- )`

Exception Stack: `( -- EXCEPTION_HANDLER )`

21.1.26.2 Instruction pope

Synopsys:

```
pope
```

Pop an exception handler from the exceptions stack.

Stack: `( -- )`

Exception Stack: `( EXCEPTION_HANDLER -- )`

21.1.26.3 Instruction raise

Synopsys:

```
raise
```

Raise the given exception.

Stack: `( EXCEPTION -- )`

Exception Stack: `( -- )`

21.1.26.4 Instruction popexite

Synopsys:

```
popexite
```

Pops the exception on the stack and sets it in the VM.

Stack: `( EXCEPTION -- )`

Exception Stack: `( -- )`

21.1.27 Debugging Instructions

21.1.27.1 Instruction strace

Synopsys:

```
strace DEPTH
```

Print a debugging trace with the elements of the top of the stack. The number of elements to print is specified in DEPTH. A depth of zero means to print the whole stack.

Stack: `( -- )`

21.1.27.2 Instruction disas

Synopsys:

```
disas
```

Print out the disassembling of the program executed by the closure in the top of the stack.

Stack: `( CLS -- CLS )`

21.1.27.3 Instruction note

Synopsys:

```
note VALUE
```

This instruction is intended to be used to insert annotations that help to understand disassemblies. Most of the times VALUE is a string.

Semantically, this instruction does nothing.

Stack: `( -- )`

21.1.28 System Interaction Instructions

21.1.28.1 Instruction getenv

Synopsys:

```
getenv
```

This instruction gets the name of an environment variable on the stack and pushes the value of the corresponding environment variable. If no variable with the given name is defined on the environment, then push PVM_NULL.

Stack: `( STR -- STR STR )`

21.1.29 Miscellaneous Instructions

21.1.29.1 Instruction nop

Synopsys:

```
nop
```

Do nothing.

Stack: `( -- )`

21.1.29.2 Instruction rand

Synopsys:

```
rand
```

Push a pseudo-random integer to the stack.

If the argument is 0U then it is ignored. Otherwise it is used to set the seed for a new sequence of pseudo-random numbers.

Stack: `( UINT -- INT )`

21.1.29.3 Instruction time

Synopsys:

```
time
```

Push the current system time to the stack in the form of an array of two long elements containing the number of seconds and nanoseconds since the epoch.

Stack: `( -- ARR )`

21.1.29.4 Instruction sleep

Synopsys:

```
sleep
```

Sleep for a given number of seconds and nanoseconds.

If the provided number of nanoseconds are not in the range 0 to 999999999 or the number of provided seconds is negative, raise PVM_E_INVAL.

If there is any other error performing the operation then raise PVM_E_GENERIC.

Stack: `( LONG LONG -- LONG LONG )`

Exceptions: `PVM_E_INVAL, PVM_E_GENERIC`

21.1.29.5 Instruction siz

Synopsys:

```
siz
```

Given a value, push its size as a bit-offset.

Stack: `( VAL -- VAL ULONG )`

21.1.29.6 Instruction sel

Synopsys:

```
sel
```

Given a value, push its length as an unsigned long.

The length of an array is the number of values contained in it. The lenght of a struct is the number of fields contained in it. The lenght of a string is the number of characters contained in it. The length of any other value is 1.

Stack: `( VAL -- VAL ULONG )`

Appendix A Table of ASCII Codes

Oct	Dec	Hex	Char
000	0	00	NUL '\0' (null character)
001	1	01	SOH (start of heading)
002	2	02	STX (start of text)
003	3	03	ETX (end of text)
004	4	04	EOT (end of transmission)
005	5	05	ENQ (enquiry)
006	6	06	ACK (acknowledge)
007	7	07	BEL '\a' (bell)
010	8	08	BS '\b' (backspace)
011	9	09	HT '\t' (horizontal tab)
012	10	0A	LF '\n' (new line)
013	11	0B	VT '\v' (vertical tab)
014	12	0C	FF '\f' (form feed)
015	13	0D	CR '\r' (carriage ret)
016	14	0E	SO (shift out)
017	15	0F	SI (shift in)
020	16	10	DLE (data link escape)
021	17	11	DC1 (device control 1)
022	18	12	DC2 (device control 2)
023	19	13	DC3 (device control 3)
024	20	14	DC4 (device control 4)
025	21	15	NAK (negative ack.)
026	22	16	SYN (synchronous idle)
027	23	17	ETB (end of trans. blk)
030	24	18	CAN (cancel)
031	25	19	EM (end of medium)
032	26	1A	SUB (substitute)
033	27	1B	ESC (escape)
034	28	1C	FS (file separator)
035	29	1D	GS (group separator)
036	30	1E	RS (record separator)
037	31	1F	US (unit separator)
040	32	20	SPACE
041	33	21	!
042	34	22	"
043	35	23	#
044	36	24	$
045	37	25	%
046	38	26	&
047	39	27	'
050	40	28	(
051	41	29	)
052	42	2A	*
053	43	2B	+
054	44	2C	,
055	45	2D	-
056	46	2E	.
057	47	2F	/

060	48	30	0
061	49	31	1
062	50	32	2
063	51	33	3
064	52	34	4
065	53	35	5
066	54	36	6
067	55	37	7
070	56	38	8
071	57	39	9
072	58	3A	:
073	59	3B	;
074	60	3C	<
075	61	3D	=
076	62	3E	>
077	63	3F	?
100	64	40	@
101	65	41	A
102	66	42	B
103	67	43	C
104	68	44	D
105	69	45	E
106	70	46	F
107	71	47	G
110	72	48	H
111	73	49	I
112	74	4A	J
113	75	4B	K
114	76	4C	L
115	77	4D	M
116	78	4E	N
117	79	4F	O
120	80	50	P
121	81	51	Q
122	82	52	R
123	83	53	S
124	84	54	T
125	85	55	U
126	86	56	V
127	87	57	W
130	88	58	X
131	89	59	Y
132	90	5A	Z
133	91	5B	[
134	92	5C	\ '\\'
135	93	5D	]
136	94	5E	^
137	95	5F	_
140	96	60	‘
141	97	61	a
142	98	62	b
143	99	63	c

144	100	64	d
145	101	65	e
146	102	66	f
147	103	67	g
150	104	68	h
151	105	69	i
152	106	6A	j
153	107	6B	k
154	108	6C	l
155	109	6D	m
156	110	6E	n
157	111	6F	o
160	112	70	p
161	113	71	q
162	114	72	r
163	115	73	s
164	116	74	t
165	117	75	u
166	118	76	v
167	119	77	w
170	120	78	x
171	121	79	y
172	122	7A	z
173	123	7B	{
174	124	7C	|
175	125	7D	}
176	126	7E	~
177	127	7F	DEL

Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover

Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given

on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or

publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first

time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W