

This essentially means that whenever the shell has to execute binaries, it will first look into `/usr/bin` followed by `/bin`.

A very common task that one has to do when building a program from source and installing to a custom path is to add its `bin` directory to the `PATH` environment variable. Let's say in this case we install `myapp` to `/opt/myapp`, which has binaries in a directory called `bin` and libraries in `lib`.

## How to do it...

A way to do this is to say it as follows:

```
export PATH=/opt/myapp/bin:$PATH
export LD_LIBRARY_PATH=/opt/myapp/lib:$LD_LIBRARY_PATH
```

`PATH` and `LD_LIBRARY_PATH` should now look something like this:

```
PATH=/opt/myapp/bin:/usr/bin:/bin
LD_LIBRARY_PATH=/opt/myapp/lib:/usr/lib:/lib
```

However, we can make this easier by adding this function in `.bashrc`:

```
prepend() { [ -d "$2" ] && eval $1="\$2': '\$${1}" && export $1; }
```

This can be used in the following way:

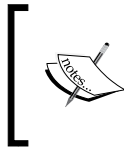
```
prepend PATH /opt/myapp/bin
prepend LD_LIBRARY_PATH /opt/myapp/lib
```

## How it works...

We define a function called `prepend()`, which first checks if the directory specified by the second parameter to the function exists. If it does, the `eval` expression sets the variable with the name in the first parameter equal to the second parameter string followed by `:` (the path separator) and then the original value for the variable.

However, there is one caveat, if the variable is empty when we try to prepend, there will be a trailing `:` at the end. To fix this, we can modify the function to look like this:

```
prepend() { [ -d "$2" ] && eval $1="\$2\${$1:+': '\$${1}}\" && export $1 ;
}
```



In this form of the function, we introduce a shell parameter expansion of the form:

`${parameter:+expression}`

This expands to `expression` if parameter is set and is not null.

With this change, we take care to try to append `:` and the old value if, and only if, the old value existed when trying to prepend.

## Math with the shell

Arithmetic operations are an essential requirement for every programming language. In this recipe, we will explore various methods for performing arithmetic operations in shell.

### Getting ready

The Bash shell environment can perform basic arithmetic operations using the commands `let`, `(( ))`, and `[]`. The two utilities `expr` and `bc` are also very helpful in performing advanced operations.

### How to do it...

1. A numeric value can be assigned as a regular variable assignment, which is stored as a string. However, we use methods to manipulate as numbers:

```
#!/bin/bash
```

```
no1=4;
```

```
no2=5;
```

2. The `let` command can be used to perform basic operations directly. While using `let`, we use variable names without the `$` prefix, for example:

```
let result=no1+no2
```

```
echo $result
```

- Increment operation:

```
$ let no1++
```

- Decrement operation:

```
$ let no1--
```

- Shorthands:

```
let no+=6
```

```
let no-=6
```