Functions

A feature of `bash` is that it allows you to define *functions* within scripts. (In fact it allows you to define functions on the command line as well, but we'll only look at functions for scripting purposes.) A function is simply like a named script within a script.

A function declaration is as follows:

```
function name () {
  cmd
  ...
  cmd
}
```

For example, consider the following function declaration, that defines a function `lsn` that reports the name of the current directory before reporting its content:

```
function lsn () {
  echo "$(pwd)"
  ls
}
```

Invoking a function is done just like you invoke other commands. Functions can be redirected just like other commands, so that `lsn > f` will send the output of executing `lsn` to the file `f`.

Functions are also like commands in that they return an exit code. By default, the exit code is 0. If you want to return a different exit code, you call `return` *n* in the function. This will terminate the function execution and return an exit code of *n*. (Note that if you use `exit` *n* in a function, you'll actually get out of the script that is executing the function, which may or may not be what you want.) Functions returning exit codes mean that you can use functions as controlling commands in conditionals and loops.

Functions can also be recursive, i.e., they can invoke themselves. This is useful when doing for example recursive walks over directory structures (what you did in Homework 1, for instance). As with all recursive functions, you need to make sure that you do not end up with an infinitely

loop. Everything that you have ever learned (and will ever learn) about recursion in programming languages applies here.

A function has access to all the variables defined by the script. It can access them, and change them. Moreover, variables defined by the function are also available to the script. Consider the following simple example:

```
function test () {
  foo=after
  bar=new
}

foo=before
test
echo "$foo $bar"
```

If you run this script, you get an output of `after new`, showing that the function `test` changed the value of `foo`, and also that the variable `bar` is available after the function executes.

To prevent this kind of behavior, a function can declare a variable as local. A local variable does not survive the invocation of the function. Moreover, if a variable of the same name is used in the script, that variable is not affected by changes to the local variable of the same name. Hence, local variables behave like local variables in standard programming languages. A variable `foo` is declared local by using the expression `local foo`. Hence:

```
function test () {
  local foo
  local bar
  foo=after
  bar=new
}

foo=before
test
echo "$foo $bar"
```

will output `before`, showing that the script variable `foo` was unaffected by the update of the local `foo` in `test`, and moreover that the local variable `bar` did not survive past the invocation of `test`.

Something interesting happens when one function invokes another. The invoked function has access to all the local variables of the invoking function. This can lead to surprising behavior, and is typically very hard to debug. To see why, notice that if a function `f` accesses a non-local

variable `foo`, determining what variable `foo` gets updated depends on who exactly is invoking `f`. (If you're curious, this means that shell script functions use dynamic scope to manage variable access. Most programming languages you are used to, i.e. C, Java, SML, use lexical scope.)

Finally, since functions behave as commands, they can take arguments. The arguments to a function are passed in the standard argument variables `$1`, `$2`, etc. These argument variables are considered local to the function, and hence they will not overwrite the the argument variables of the script itself (or any other function for that matter). Note that all the argument variables are reset in this way when a function is called, not just those corresponding to actual arguments. Hence, if a function is called with two arguments, `$1` and `$2` inside the function will refer to those arguments, while `$3` and above will be null. The argument variable `$#` containing the argument count is also updaet. Consider the following example:

```
function test () {
  echo "first arg of function:  $1"
  echo "second arg of function: $2"
}

echo "first arg of script (before):  $1"
echo "second arg of script (before): $2:
test some arg
echo "first arg of script (after):  $1"
echo "second arg of script (after): $2:
```

If this is part of script invoked with arguments `one` and `two`, we get the following outputs:

```
first arg of script (before):  one
second arg of script (before): two
first arg of function:  some
second arg of function: arg
first arg of script (after):  one
second arg of script (after): two
```

A final word about how the shell executes commands. If a command to be executed contains a slash, the file system is used to find the command. If the command does not contain a slash (i.e. it's just a command name), then the shell first checks if a function of that name exists, and if so executes it. If none exists, it checks if it's a builtin command (such as `cd`, or `echo`), and if so executes. If it's not a builtin, then the directories in the `PATH` environment variables are scanned to find an executable with that name.