Shell Scripts

Sistemas Operativos I

Scripts

- Text files, usually identified using “.sh” extension

- To run them (2 options):
  - *bash myscript.sh* or *./myscript*
  - Make the script runnable:
    - chmod +x myscript.sh
    - ./myscript (The “.” means “execute”, and the “/” is to indicate the current folder).

Scripts

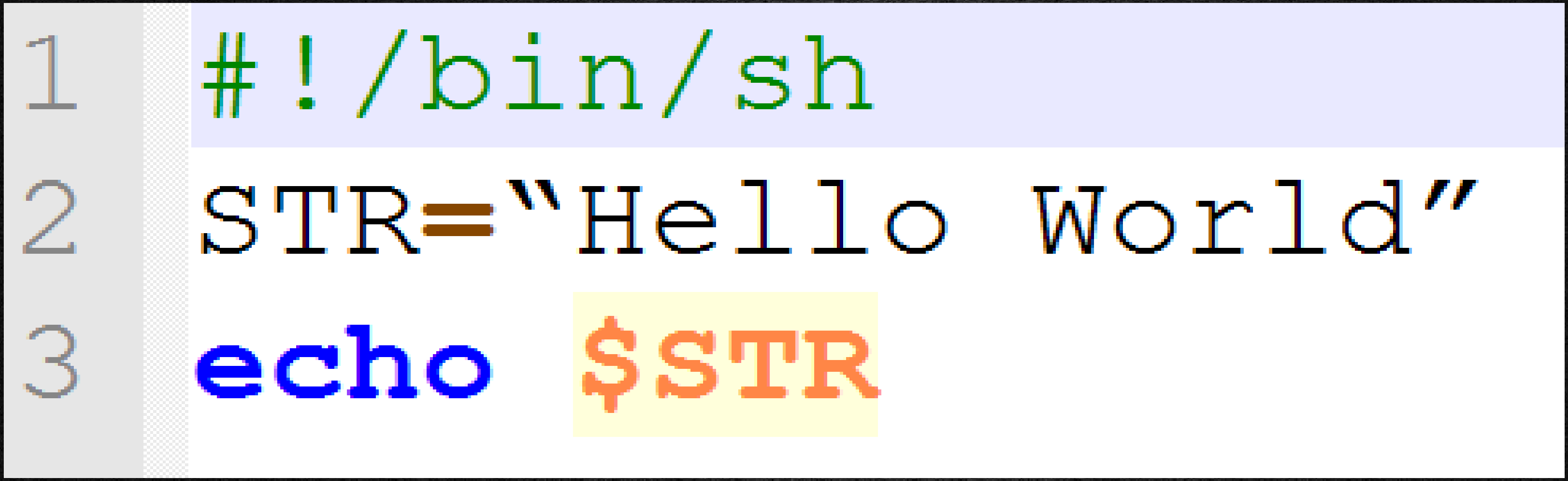
- Usually, an special comment is added to indicate the shell used to run the script. E.g.:
  - #!/bin/bash
- Each line is executed one after the other
- Things that can be done in the shell can also be done in a script, and viceversa.

Variables

- You can use variables as in any programming languages
- There are no data types. A variable in bash can contain a number, a character, a string of characters. Sample script:

```
#put "something" into myvar
myvar=”something”
 #put 4 into MY_VAR2
MY_VAR2=4
 #Concat myvar,MY_VAR2, add “extra” and save it as concat_var
concat_var="$myvar $MY_VAR2 extra”
# Prints something 4 extra
echo $concat_var
```

Hello World! using variables

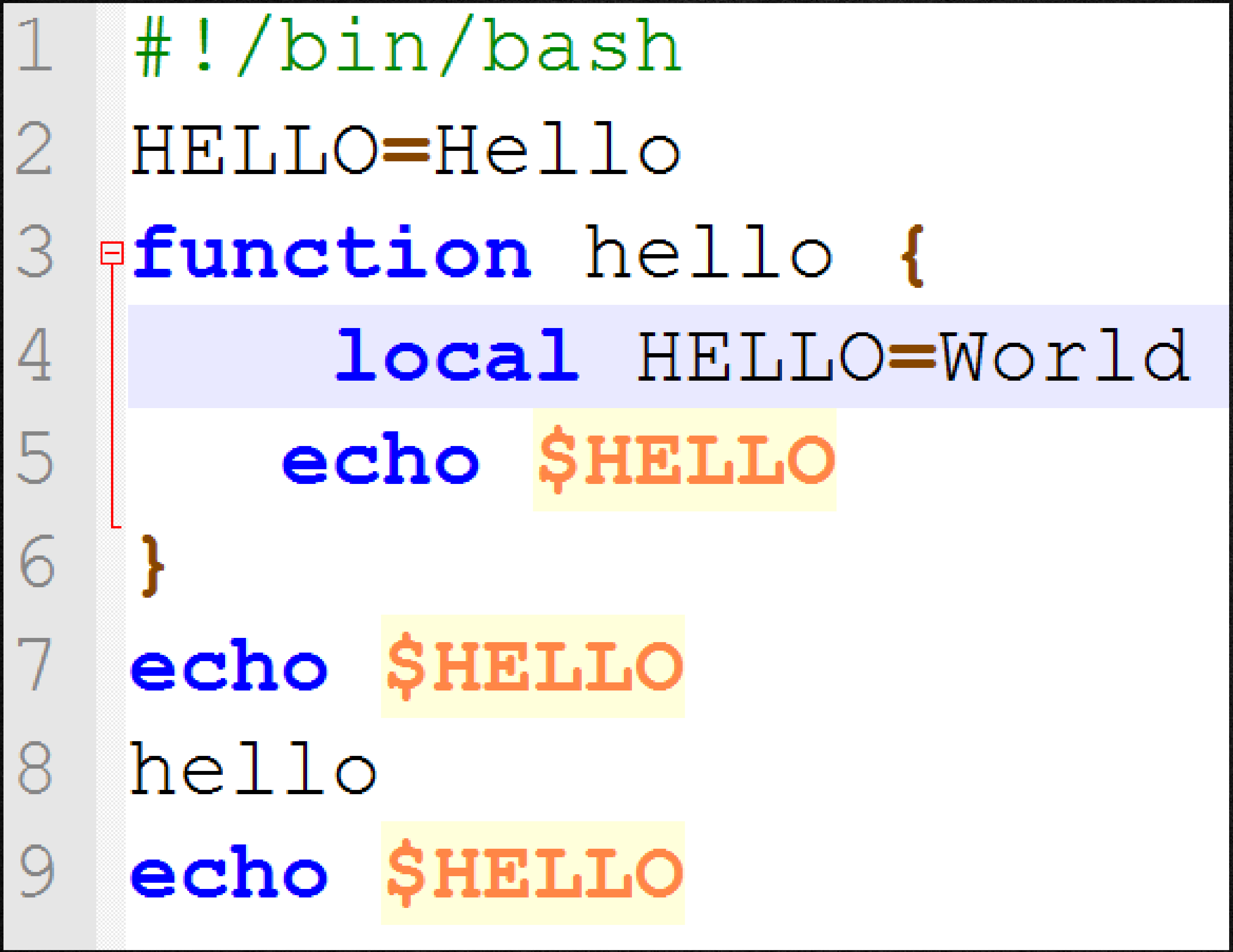


```
#!/bin/sh
STR=“Hello World”
echo $STR
```

Functions

- A function is a sub-script. It receives arguments and to return values it must print to standard output
- WARNING! The “return” reserved word returns an integer value indicating the error code, it's not used for function value returning
- Functions may have their own variables, if locality needs to be enforced, the “local” word can be used to indicate local variables

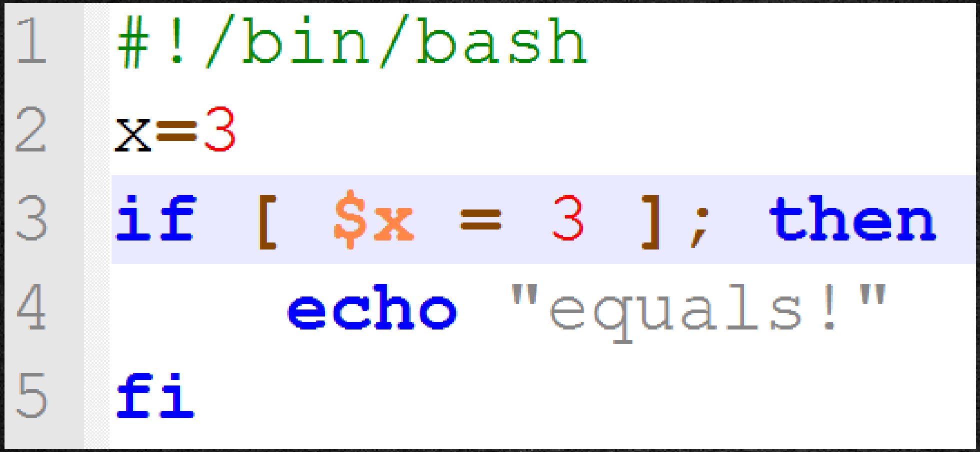
Local variables



```
#!/bin/bash
HELLO=Hello
function hello {
        local HELLO=World
        echo $HELLO
}
echo $HELLO
hello
echo $HELLO
```

Conditions and Tests

- First, we need to know the different ways of expressing conditions:
  - Brackets: Oldest way of testing



```
#!/bin/bash
x=3
if [ $x = 3 ]; then
      echo "equals!"
fi
```

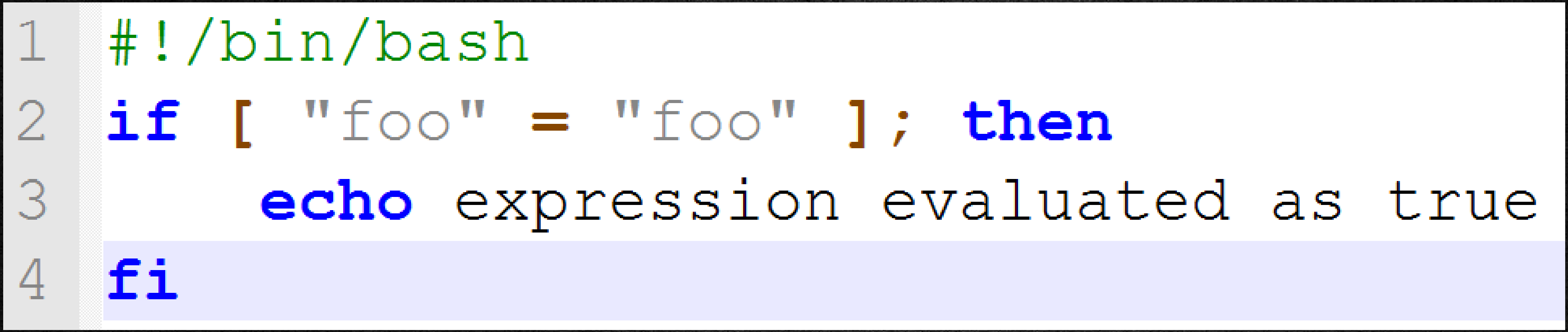
It's the same as: if test $x = 3; then

That's why you must use spaces before brackets.

You can invert a condition using ! (after first bracket)

-a and -o to make “and” and “or” conditions.

Basic conditional example if .. then



```
#!/bin/bash
if [ "foo" = "foo" ]; then
    echo expression evaluated as true
fi
```

Conditions and Tests

- With brackets you can test:
  - numbers using equal(-eq), greater than(-gt), greater or equal than (-ge), and so (-lt,-le,-ne):

    `if [ $x -eq 3 ]; then`

  - Strings using ==,!=,> and < (> and < must be escaped using \, because they are also used for redirection)

    `if [ “$x” == “my string” ]; then`

  - Files:
    - Exist: -a myfile.tar.gz, true if file (or dir) exists
    - Dir: -d mydir, true if exists and is dir.
    - Size: -s myfile, file exists and its size>0
  - String tests:
    - Non-empty: -n “$myvar”
    - Empty: -z “$myvar”

Conditions and Tests

- Double brackets: if [[condition]];

You can test the same things than using single brackets and:

- Use simple regular expressions to compare Strings:

```
If [[ “$var1” == *[aA]le* ]];
```

- Omit quotes for strings

```
If [[ $var1 == algo ]];
```

- When testing files, it does not use globbing. Example: check if the file “*.txt” exists

```
if [[ -a *.txt ]]; then
```

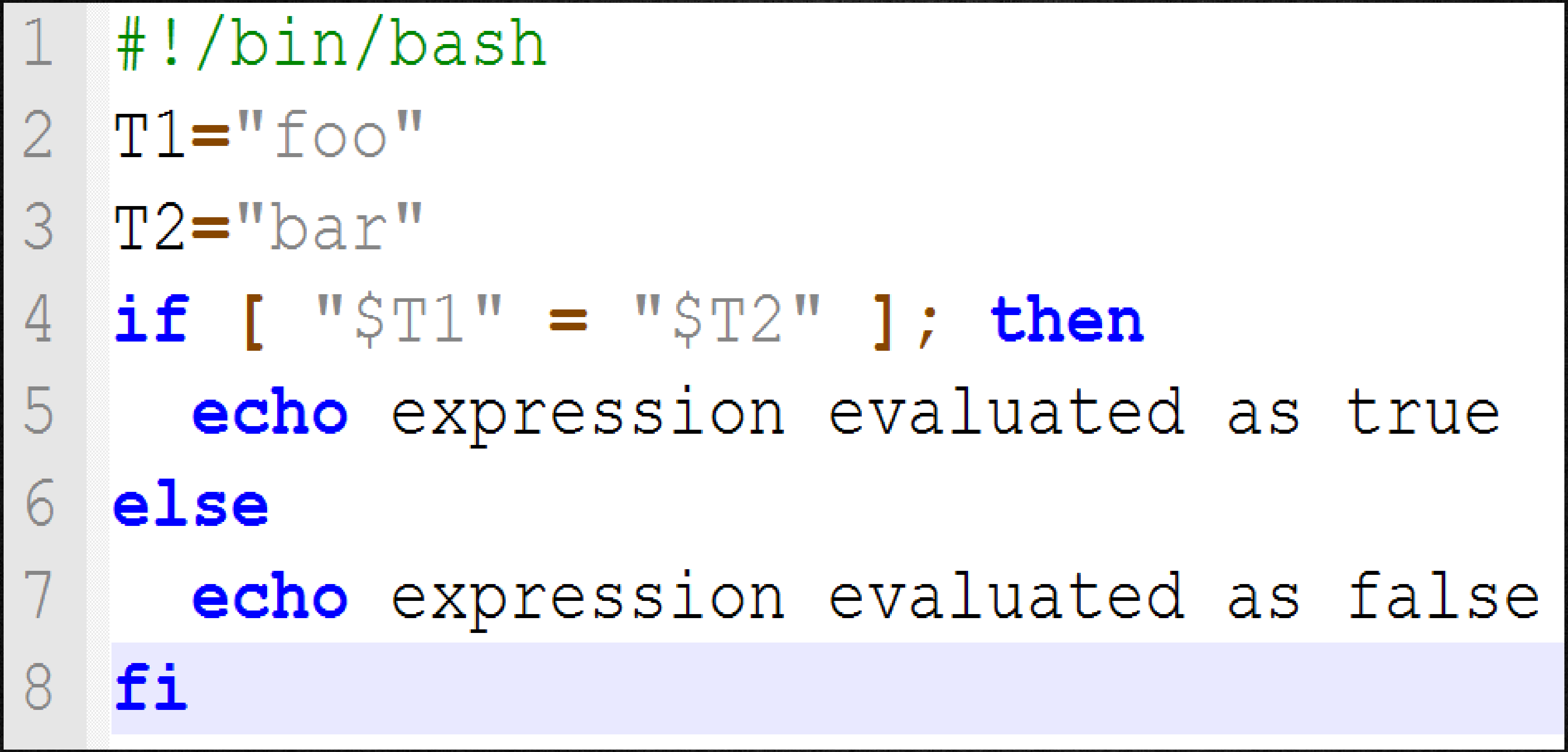
- Use &&, || to chain conditions (also -a and -o).

Conditions and Tests

- Double parenthesis:
  - Check number conditions using <,>,<=,>=,!=, == && and ||.

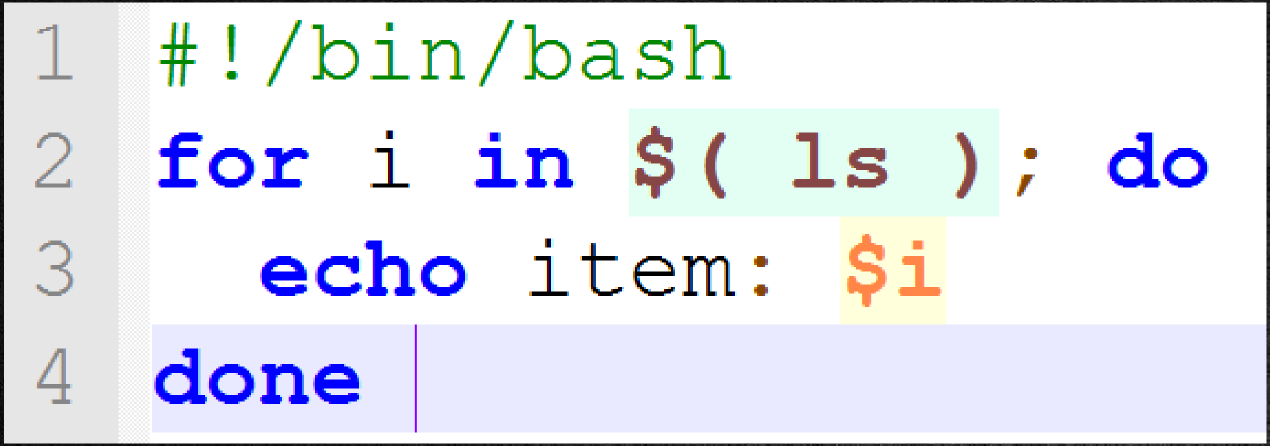
```
if (( $num >= 5 )); then
```

Conditionals



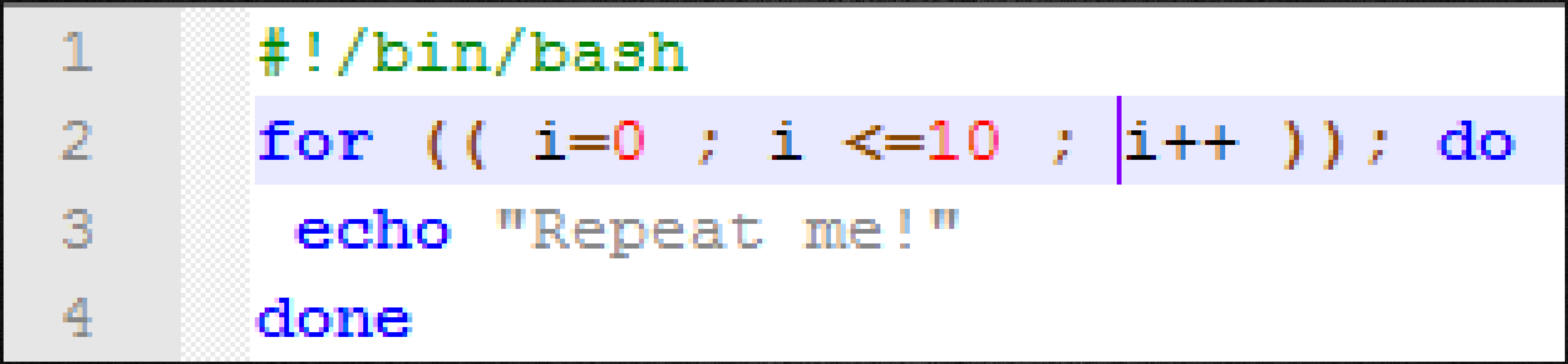
```bash
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ]; then
  echo expression evaluated as true
else
  echo expression evaluated as false
fi
```

ForEach Loop



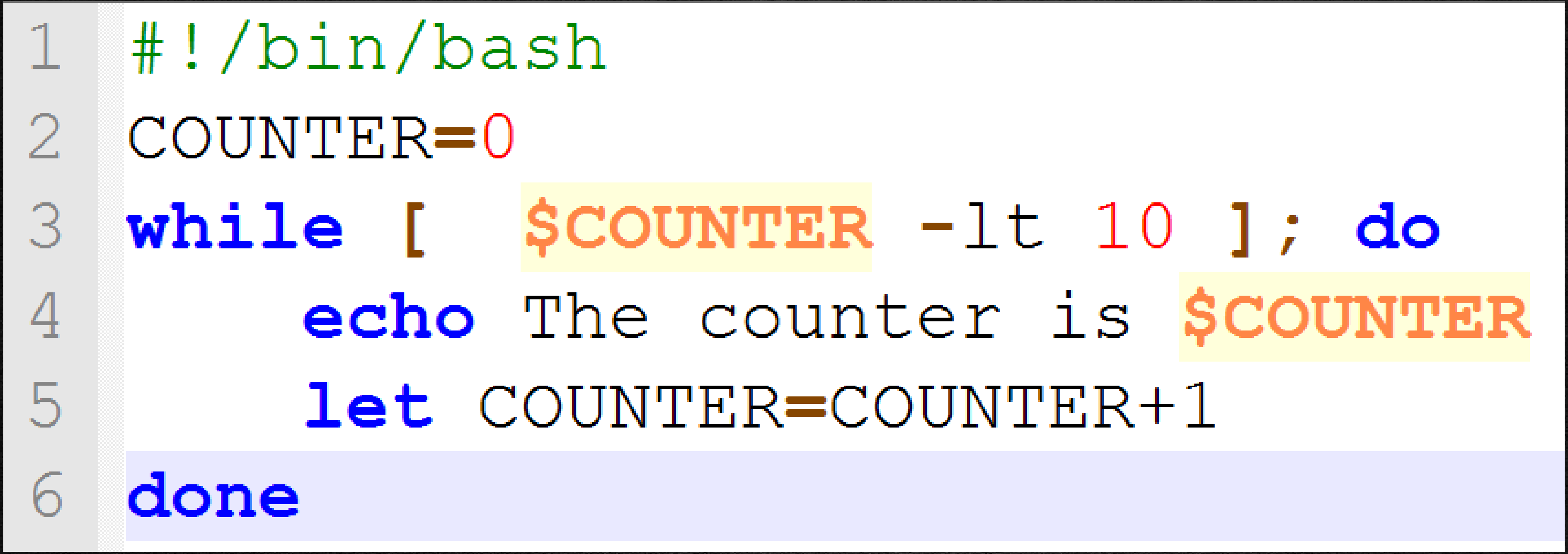
```
#!/bin/bash
for i in $( ls ); do
  echo item: $i
done
```

For



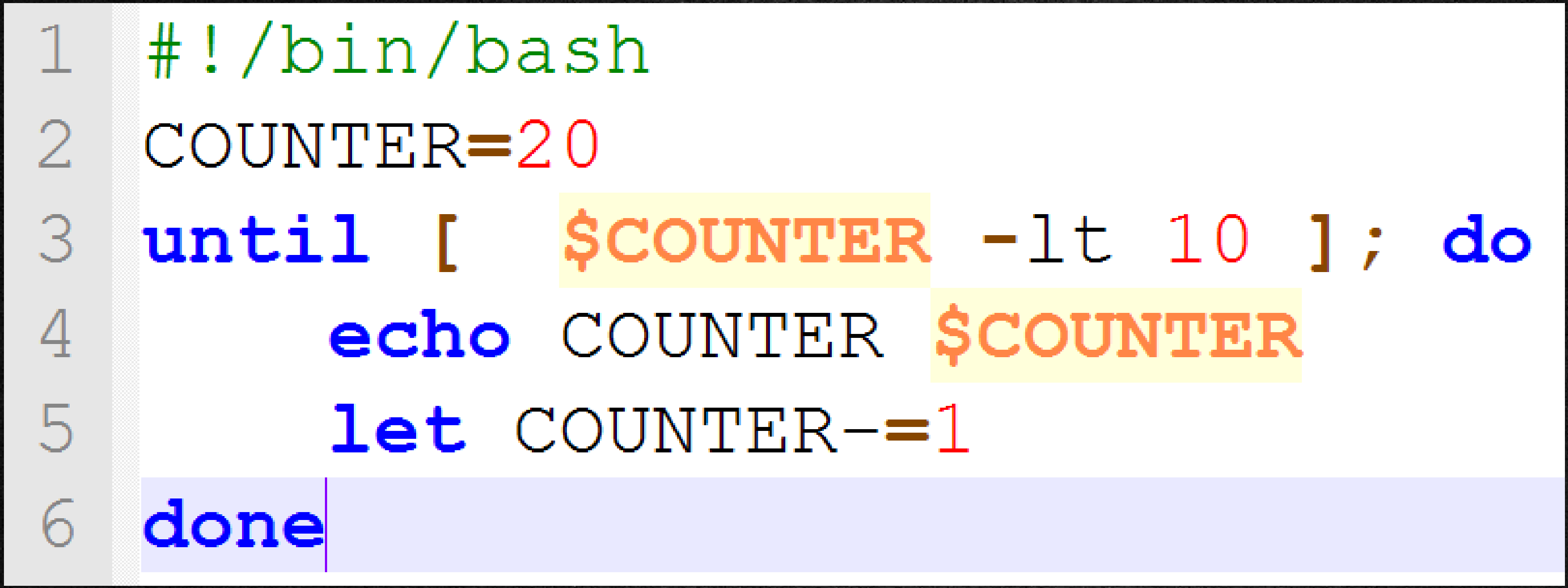
```
#!/bin/bash
for (( i=0 ; i <=10 ; i++ )); do
 echo "Repeat me!"
done
```

While



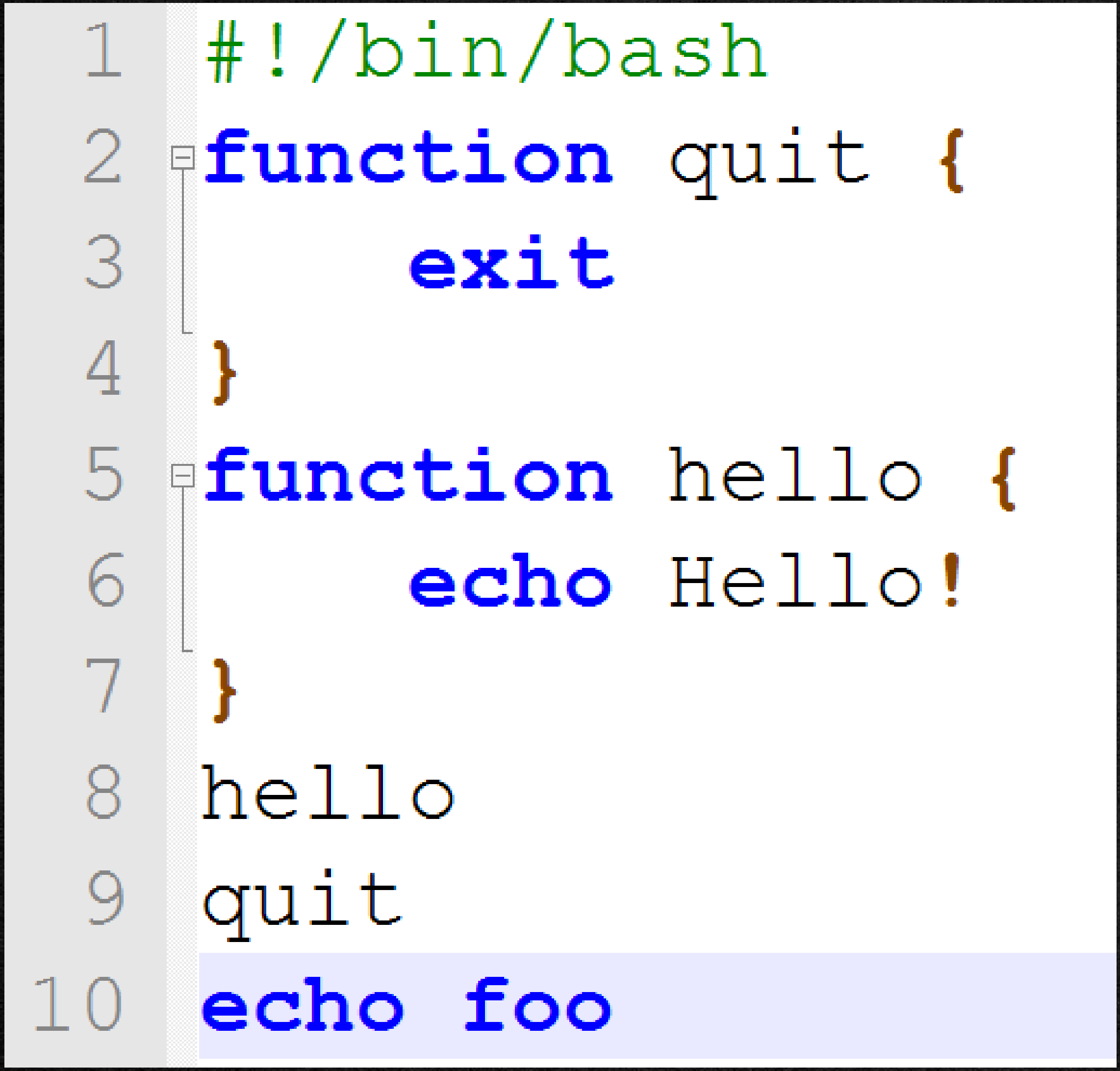
```
#!/bin/bash
COUNTER=0
while [  $COUNTER -lt 10 ]; do
     echo The counter is $COUNTER
     let COUNTER=COUNTER+1
done
```

Until



```
#!/bin/bash
COUNTER=20
until [  $COUNTER -lt 10 ]; do
     echo COUNTER $COUNTER
     let COUNTER-=1
done
```

Functions without parameters



```
#!/bin/bash
function quit {
	exit
}
function hello {
	echo Hello!
}
hello
quit
echo foo
```

Parameter Expansion

A parameter is an entity that stores values and is referenced by a:

- Name Variables! (for example, echo $MYVAR)
- Number Arguments (from 1 to 9, accesed using $1 to $9)
- Special symbol ?,*,$,#,@,_,0,!,- (for example, $$ returns the PID of the current process, or $0 returns the name of the shell or current shell script)

Parameter Expansion Access values

- Access value: $PARAMETER or ${PARAMETER}
  - Access the value of the PARAMETER ($VAR, $4, $@, another example: ${WORD}s, adds an “s” after the value of WORD).
- Indirection: ${!PARAMETER}, access the value of the parameter contained in PARAMETER
- Variable name expansion: ${!PREFIX*} or ${!PREFIX@} searches for variables that start with PREFIX

Parameter Expansion String manipulation

- Case Modification (to upper, to lower).
  - ${PARAMETER^} first char to upper
  - ${PARAMETER^^} all chars to upper
  - ${PARAMETER,} first char to lower
  - ${PARAMETER,,} all chars to lower
  - ${PARAMETER~} alternates case of first char of every words
  - ${PARAMETER~~} alternates case of all chars except the first of every word.

Parameter Expansion String manipulation

- Substring removal (also for filename manipulation!)
  - ${PARAMETER#PATTERN}
  - ${PARAMETER##PATTERN}
  - ${PARAMETER%PATTERN}
  - ${PARAMETER%%PATTERN}
- Search and replace
  - ${PARAMETER/PATTERN/STRING}
  - ${PARAMETER//PATTERN/STRING}
  - ${PARAMETER/PATTERN}
  - ${PARAMETER//PATTERN}
- String length
  - ${#PARAMETER}

Command Substitution

Replaces the invocation of a command with it's result. There are two forms:

`$(command)` or `` `command` ``

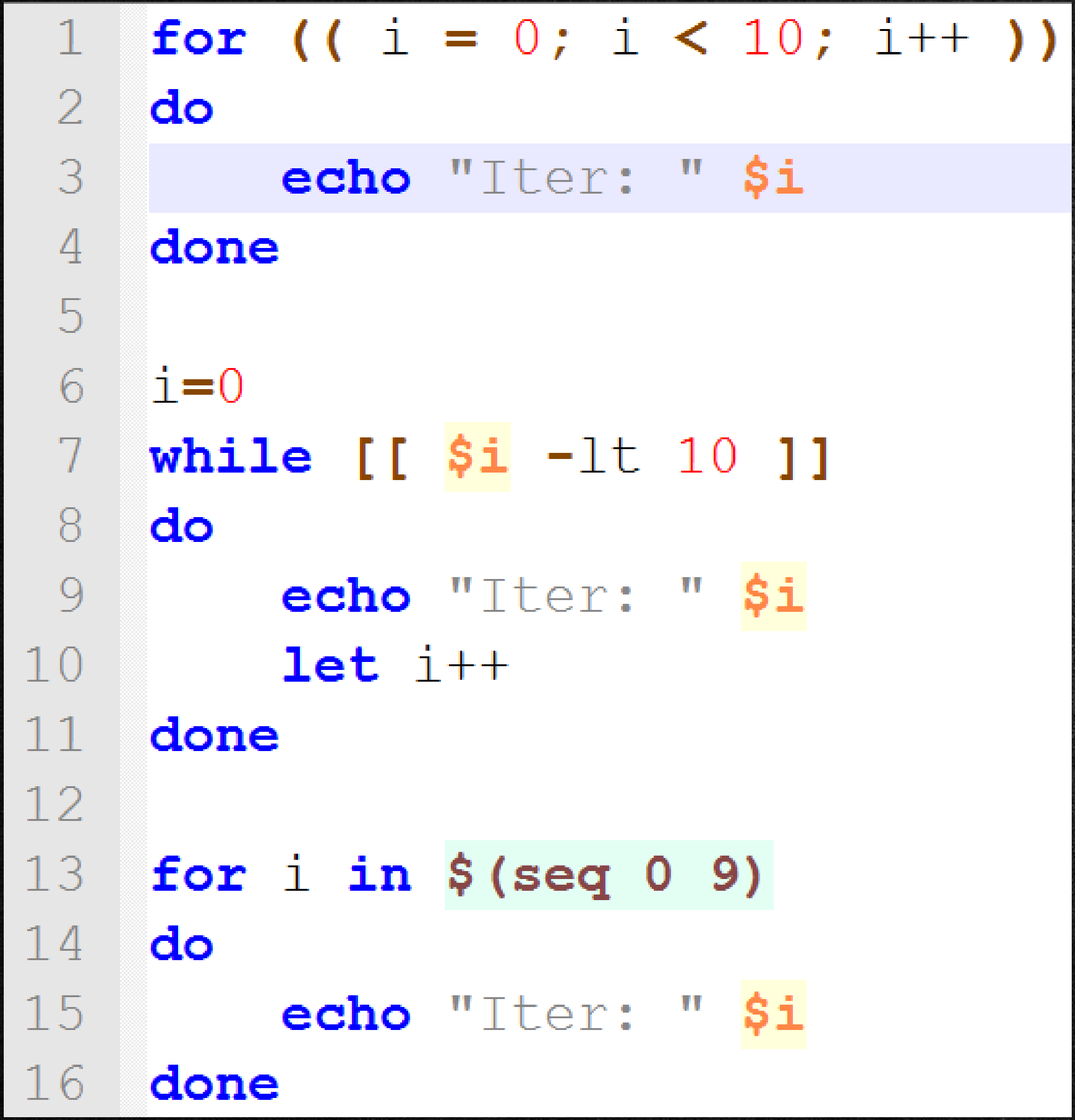
In general, is preferable to use $() as it opens a new parsing step. This means that $() can be nested and no escaping is needed (for example, you can use quotes as in salute="$(echo "hello")”).

If you happen to need this `$((command))`, be careful to use spaces $((command)), because $(()) means arithmetic expansion!.

Curly Braces Expansion

- {expansion}, without the $
- Generates sequences of characters:
  - {a..g} → a b c … g
  - {1..15} → 1 2 3 … 15
  - a{1..4}b or a{1,2,3,4}b → a1b a2b a3b a4b
  - Nesting: {a, {1,2,3}b, c} → a 1b 2b 3b c

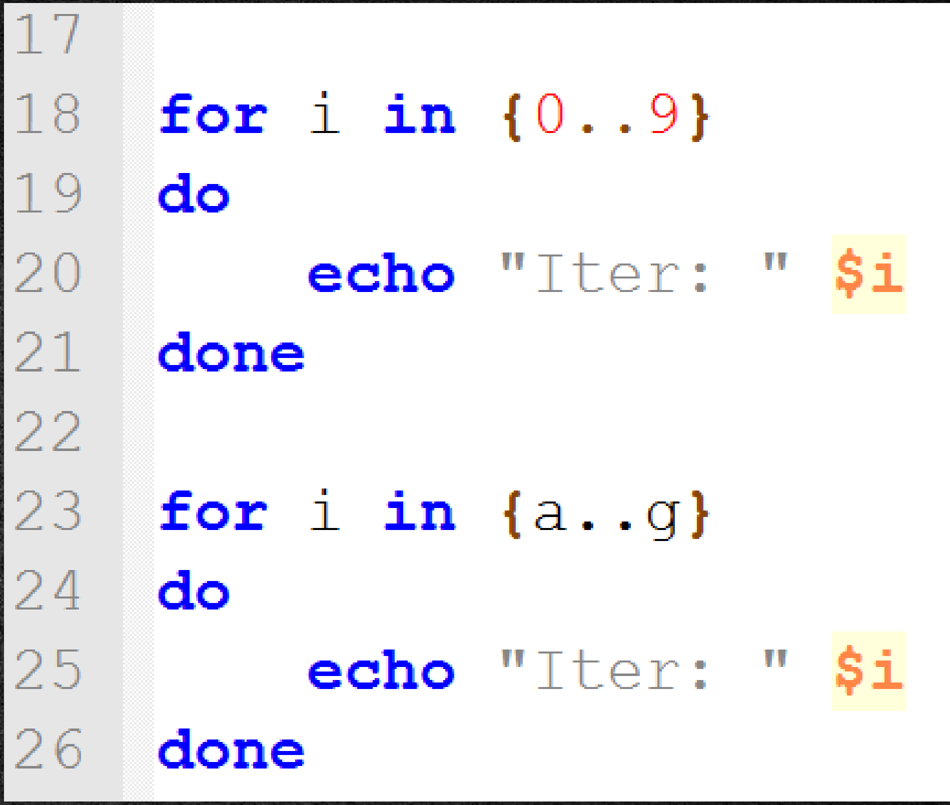
Using sequences in a loop



```bash
for (( i = 0; i < 10; i++ ))
do
	echo "Iter: " $i
done

i=0
while [[ $i -lt 10 ]]
do
	echo "Iter: " $i
	let i++
done

for i in $(seq 0 9)
do
	echo "Iter: " $i
done
```



```bash

for i in {0..9}
do
	echo "Iter: " $i
done

for i in {a..g}
do
	echo "Iter: " $i
done
```

Arithmetic Expansion

- `$((expression)) or $[expression]`
  - `x=3;y=4;echo $((x+y))`
    - `Output: 7`
  - `echo $((4+4))`
    - `Output: 8`
  - `echo $[4/3]`
    - `Output: 0`

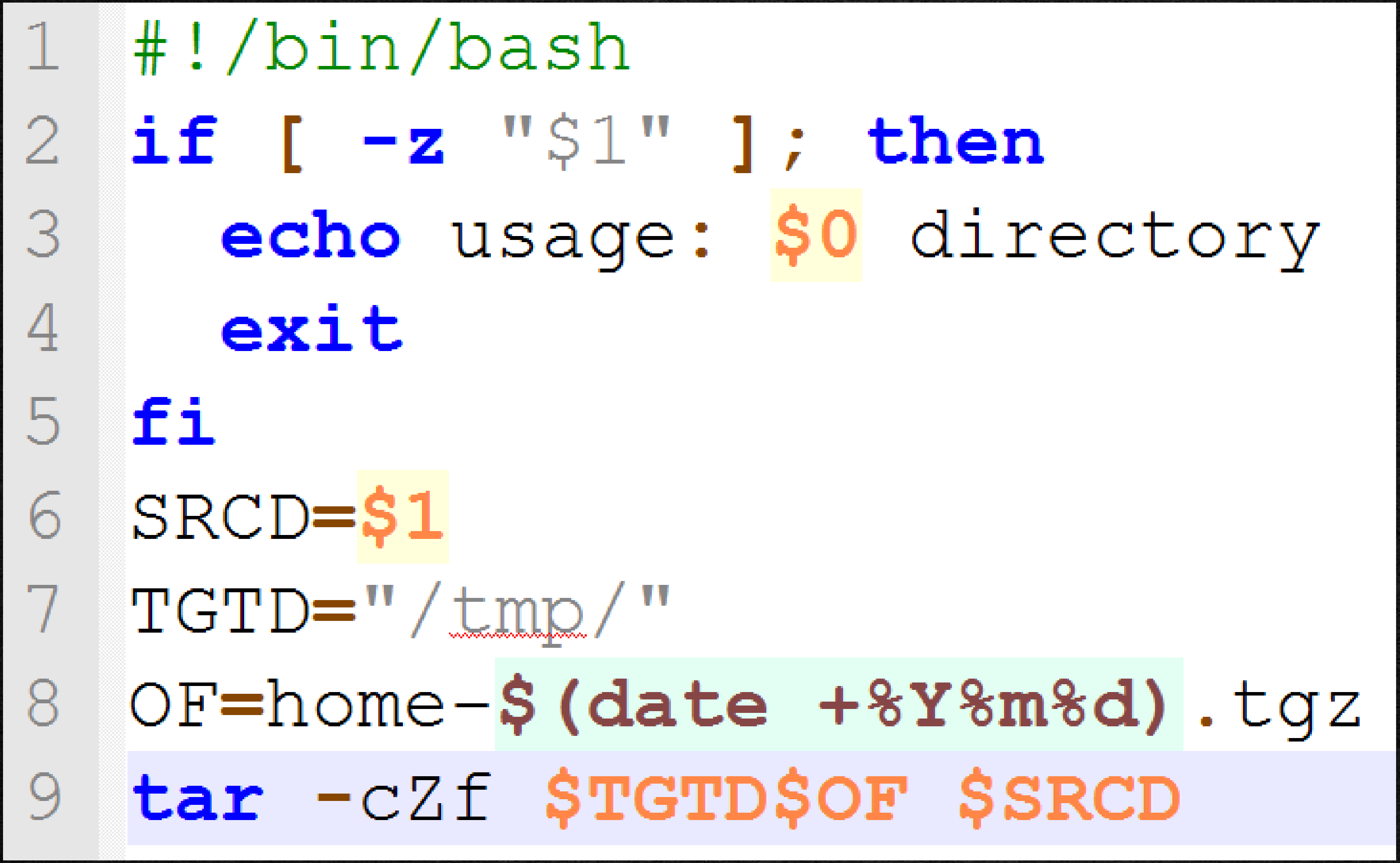
Floating Point

- Must another command, for example, bc (a calculator language).
  - echo “scale=2;4/3” | bc -l
- Meaning:
  - Set scale to 2 decimals, then define expression 4/3. “echo” that code to bc to be interpreted.

Asigning expressions

- `let` *builtin* command:
  - `let i++` or `let i=i+1`
  - `let i=4+3`
  - `let a=4+3;echo $a` → prints 7
  - `let a=2**3` → a=8
- Another alternative:
  - `i=$((i+1))`

Using command line arguments



```
#!/bin/bash
if [ -z "$1" ]; then
  echo usage: $0 directory
  exit
fi
SRCD=$1
TGTD="/tmp/"
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD
```

Debugging Scripts

- In many shells,-x (for *xtrace*) prints command execution to standard error allowing the user to debug scripts.
- Example:

```
# bash -x <(echo "echo Hello World")
+ echo Hello World
Hello World
```

- Same Example (using a pipe, instead of file descriptor)

```
# echo "echo Hello World" | bash -x
+ echo Hello World
Hello World
```

-