Manchester Metropolitan University

Coffee Tutorial (Linux only!)

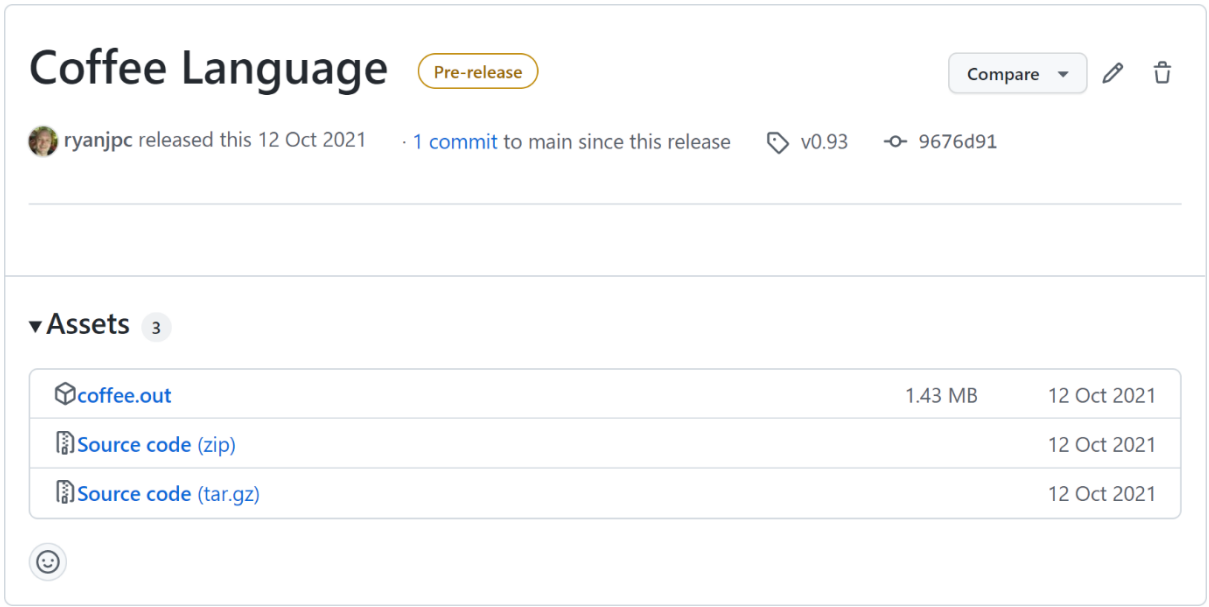
Contents

Installing Coffee

1) In a terminal, type the following (one line at a time):

```
cd ~/Downloads
wget "https://github.com/ryanjpc/CoffeeRelease/releases/download/v0.93/coffee.out"
chmod +x ./coffee.out
```

Alternatively, visit https://github.com/ryanjpc/CoffeeRelease/releases and download the latest release (see below: coffee.out), then copy to ~/Downloads/. Then make the file executable with `chmod +x ./coffee.out`



2) Now, in a terminal, type:

```
nano ~/.bashrc
```

Enter and save (ctrl+x then Y then Enter) the following text:

```
alias coffee='~/Downloads/coffee.out'
```

Then in a terminal, type:

```
source ~/.bashrc
```

Invoking the Coffee Compiler

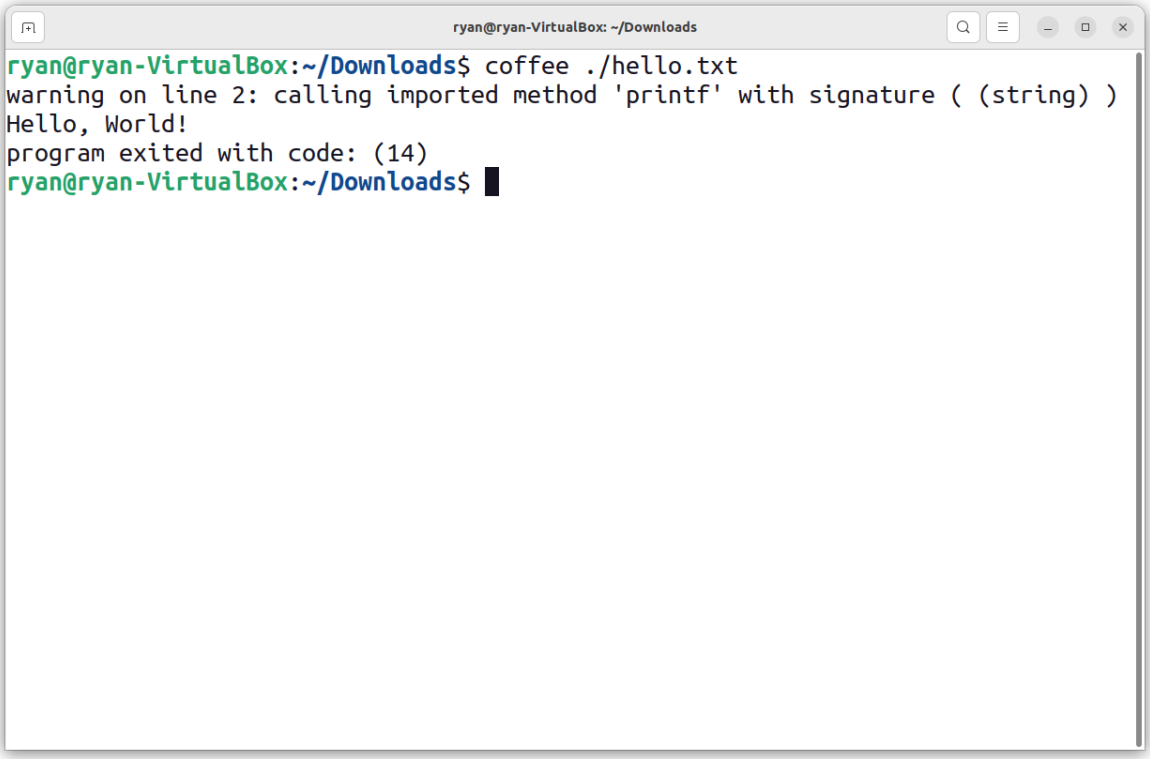
1) Create a new text document with the following contents:

```
import printf;
printf("Hello, World!\n");
```

Save the document with the filename "hello.txt", then, in a terminal, navigate to your working directory (where you saved "hello.txt"), then type:

```
coffee ./hello.txt
```

Verify that your terminal responds as follows:



Debugging Coffee Programs

It is possible to return an 8-bit code from the main program. The return value is the lower 8 bytes of a 64-bit integer, meaning that it is possible to test expressions resulting in values up to 2^8-1=255.

1) Compile and run the following program:

```
return 1;
```

Expected response:

```
program exited with code: (1)
```

2) The program code is the return value. It can be useful in reporting error codes, or just evaluating and testing expressions, without requiring function calls to print to terminal (i.e. printf).

Here is an example using a binary expression:

```
return 1 + 2;
```

Expected response:

```
program exited with code: (3)
```

3) The C-library printf function can be imported and used to print messages and results to the terminal during program execution:

```
import printf;
printf("integer: %d, float: %f", 3, 3.0);
```

Expected response:

```
integer: 3, float: 3.000000
program exited with code: (28)
```

Please see this C programming guide (https://www.cprogramming.com/tutorial/printf-format-strings.html) for explanation of printf and other C functions (note: the C-library is not fully compatible with Coffee, thus some features will not work).

Variables & Types

1) Variables can have types bool, int and float (bool type is just a 64-bit integer):

```
bool a;
int b;
float c;
```

2) Variables can be assigned during declaration:

```
bool a=true;
int b=1;
float c=1.0;
```

3) Variables can be declared in comma-separated list notation and assignment is optional:

```
bool a=true, a1=false;
int b=1, x, y;
float c=1.0;
```

4) Coffee allows strings and chars in expressions, which are most commonly used in printing to screen:

```
import printf;
printf("string: %s, char: %c\n", "my string", 'c');
```

5) There are no string or char variables, although (bit hacky, but...) they can be assigned to integer variables via type casting and used as an argument in a print function for example:

```
import printf;
int a = (int)"string test!\n";
printf("%s", a);
```

Global / Local Scopes

The scope of an identifier (variable, method, etc) is the parts of the code where that identifier can be referenced.

1) Variables and methods can be declared in the global scope, which is the main program body. Variables can be declared in local scope, which is anywhere within curly braces:

```
int global_variable;
{
  int local_variable;
}
```

2) Method arguments exist within the local scope of the method:

```
void my_method(int local_method_var_1) {
  int local_method_var_2;
}
```

3) The loop variable in a for loop exists in the local scope of the loop:

```
for (local_loop_var_1 in [0:10]) {
  int local_loop_var_2;
}
```

4) Globals exist in the static address space of the program thus can be referenced in any scope, while locals exist in the stack address space of the program. Local variables do not exist outside of their scope and thus cannot be referenced in a higher scope.

```
int a = 1;
{
  int a = 2;
  {
    int a = 3;
  }
  printf("%d\n", a);
}
printf("%d\n", a);
```

Expected response:

```
2
1
```

Arrays

1) Arrays are compile-time fixed length and can have any type a variable can have. They can exist in global and local scope, but cannot be method arguments:

```
int a[10], b[10];
{
  int a[10], b[10];
}
```

2) Arrays cannot be assigned during declaration, but can be assigned afterwards using indexing:

```
int a[10], b[10];
{
  a[0] = 5;
}
b[2] = 3;
```

3) Arrays can be indexed with an expression, which includes literals, variables and arithmetic expressions:

```
int a[10], b = 1;
{
  a[b] = 5;
}
a[b + 1] = 3;
```

Expressions (Arithmetic)

Coffee has binary and unary arithmetic operations that look like many other programming languages. The order of operations depends upon the relative precedence (see language doc) of each operator. Round brackets are used to group sub-expressions.

Here are some examples of arithmetic expressions and their results:

1) Addition & multiplication

```
return 1 + 2 * 3;
program exited with code: (7)
```

2) Grouping with brackets:

```
return (1 + 2) * 3;
program exited with code: (9)
```

3) Modulo (remainder):

```
return 10 % 3;
program exited with code: (1)
```

4) Unary minus (negation):

```
return -3;
program exited with code: (253)
```

Note: the return code is unsigned, so it displays 253 in accordance with two's complement (i.e. (binary not x) + 1 applied to 11111101 (253) becomes 00000011 (3)).

Expressions (type coercion)

Coffee uses a system of type coercion (promotion) to implicitly convert expressions with different types, enforcing compatibility and evaluating to the type with the highest precedence (see language doc). Note: bool must be explicitly cast (see type casting).

Here are some examples of type coercion:

1) An int variable 'a' can be promoted to float (notice printf is expecting float %f):

```
import printf;
int a = 2;
printf("result: %f", 3. * a);
```

Expected response:

```
result: 6.000000
```

2) The expression 2 + 3 is promoted to float for assignment to float variable 'a':

```
import printf;
float a = 2 + 3;
printf("result: %f\n", a);
```

Expected response:

```
result: 5.000000
```

Expressions (type casting)

Type coercion does not apply in every situation (to give room for teaching other concepts like reporting incompatibility errors and type casting). In most situations, type casting can be specified to otherwise enforce type compatibility – particularly useful for method arguments and return types:

Here are some examples of type casting:

1) A bool literal 'true' can be cast to int:

```
import printf;
float a = 2 + (int)true;
printf("result: %f", a);
```

Expected response:

```
result: 3.000000
```

2) A method argument 'a' can be cast to int:

```
import printf;
float a = 2 + (int)true;
printf("result: %d\n", (int)a);
```

Expected response:

```
result: 3
```

3) Method arguments are not coerced – they must be explicitly cast if they are not a compatible type:

```
import printf;
float a = 2 + (int)true;
printf("result: %d\n", a);
```

Expected response:

```
result: -1252187672
```

Expressions (logic)

Coffee has binary and unary logic operations that look like many other programming languages. The order of operations depends upon the relative precedence (see language doc) of each operator in the expression, and round brackets can be used to group sub-expressions.

Here are some examples of logic expressions and their results:

1) Comparison operators:

```
import printf;
printf("1>2:%d, 1<2:%d, 3==3:%d, 5<=5:%d\n", 1>2, 1<2, 3==3, 5<=5);
```

Expected response:

```
1>2:0, 1<2:1, 3==3:1, 5<=5:1
```

2) Truth Operators (with logical not):

```
import printf;
printf("T and T:%d, T and F:%d\n", true && true, true && false);
printf("F or T:%d, F or (not T):%d\n", false || true, false || !true);
```

Expected response:

```
T and T:1, T and F:0
F or T:1, F or (not T):0
```

Expressions (ternary)

Coffee has ternary expressions just like many other languages (e.g. Python, C, Java, etc...).

1) Ternary expressions can be a convenient shorthand way to assign a variable one of multiple values, depending upon the outcome of some conditional expression e.g:

```
import printf;
int a = 3 > 1 ? 9 : 10;
int b = 3 < 1 ? 9 : 10;
printf("a:%d, b:%d\n", a, b);
```

Expected response:

```
a:9, b:10
```

2) Ternary expressions can also be used to pass method arguments in the same way:

```
import printf;
printf("a:%d, b:%d\n", 3 > 1 ? 9 : 10, 3 < 1 ? 9 : 10);
```

Expected response:

```
a:9, b:10
```

3) Or to execute a method:

```
import printf;
return 3 > 1 ? printf("true\n") : printf("false\n");
```

Expected response:

```
true
```

Branching (if statements)

1) The if statements in Coffee are just like those of Java or C:

```
bool x = true;
if (x) {
  return 1;
} else {
  return 0;
}
```

Expected response:

```
program exited with code: (1)
```

2) Just like in C or Java, the curly braces (blocks) are optional *if* the code block contains only a single statement:

```
bool x = true;
if (x)
  return 1;
else
  return 0;
```

Expected response:

```
program exited with code: (1)
```

3) Nested if statements are allowed:

```
bool x = true;
if (x) {
  if (!x)
    return 0;
  return 1;
}
```

Expected response:

```
program exited with code: (1)
```

4) Else-if statements are allowed:

```
bool x = true;
if (!x)
  return 1;
else if (x)
  return 0;
```

Expected response:

```
program exited with code: (0)
```

5) The logical component of if statements in Coffee is ***truthy*** and ***falsy***, meaning that all values greater than 0 are considered true, and all other values are considered false – just like in other languages (C, Python, etc):

```
if ((bool)5) {
  return 1;
} else {
  return 0;
}
```

Expected response:

```
program exited with code: (1)
```

Note the type cast from int to bool, which is required here (see semantics in language doc).

Loops (while)

"While loops" in Coffee are similar to those in other languages – they provide a mechanism for looping without bounds until some logic condition is met. The body of a while loop is in a block (curly braces). Just like "if statements", the braces are not required if there is only a single statement in the body of the loop.

1) While loops use truth logic, with the same bool type requirements as if statements:

```
bool cond = true;
int counter = 0;
while (cond) {
  counter = counter + 1;
  if (counter > 10)
    cond = false;
}
return counter;
```

Expected response:

```
program exited with code: (11)
```

Note: if experimenting with while loops and you encounter **infinite looping, hit ctrl+c or ctrl+z in the terminal to break execution, then fix your code, or talk to me if you think it's a nasty compiler bug 😊.

2) Break and continue statements can be used within loops:

```
bool cond = true;
int counter = 0;
while (cond) {
  counter = counter + 1;
  if (counter > 10)
    break;
}
return counter;
```

Expected response:

```
program exited with code: (11)
```

Loops (for)

"For loops" in Coffee are similar to those of Java or C, with some small differences. A "for loop" in coffee has a loop variable ('i' in code below) of type int, which is declared in the local loop scope and used to iterate over a range of values defined by the limit syntax (square brackets in examples below). The loop variable is destroyed outside the body scope of the loop.

1) Simple loop to print i from 0 to 10 (not inclusive):

```
import printf;
for (i in [0:10]) {
  printf("%d ", i);
}
```

Expected response:

```
0 1 2 3 4 5 6 7 8 9
```

2) The limit syntax is flexible – the programmer can provide:

i) a lower and upper limit to loop over a range:

```
[0:10]
```

ii) only an upper limit (lower limit of 0 is implied):

```
[:10]
```

iii) only a lower limit (upper limit of infinity is implied):

```
[0:]
```

iv) no limits (loop is infinite):

```
[:]
```

v) a step (amount to increment the loop variable):

```
[0:10:2]
```

Note: the step feature will accept negative values but is **currently not implemented (stopping condition is loop_var < upper limit, regardless of step value) and thus will not work.

3) Using the loop variable as array indices:

```
import printf;
int array[2];
array[0] = 5;
array[1] = -8;
for (i in [0:2])
  printf("%d ", array[i]);
```

Expected response:

```
5 -8
```

4) Iterating over elements of an array:

```
import printf;
int array[2];
array[0] = 5;
array[1] = -8;
for (i in array)
  printf("%d ", i);
```

Expected response:

```
5 -8
```

Methods (declaration)

Methods in Coffee can be defined anywhere in the global scope.

1) Methods can return a value:

```
int foo()
  return 5;
return foo();
```

Expected response:

```
program exited with code: (5)
```

2) There is a special syntax for a method which only returns an expression:

```
int foo() 5
return foo();
```

Expected response:

```
program exited with code: (5)
```

3) Methods can accept arguments of type bool, int or float, but cannot accept arguments of array type without indexing first:

```
int a[1];
a[0] = 1;
int foo(int a, int b) a + b
return foo(a[0], 2);
```

Expected response:

```
program exited with code: (3)
```

4) Recursion is an intrinsic feature of Coffee methods:

```
int foo(int a, int b) {
  if (b > 100) return b;
  return foo(b, a + b);
}
return foo(1, 1);
```

Expected response:

```
program exited with code: (144)
```

Methods (importing C-lib methods)

Coffee makes use of the C library functions by an import mechanism. The import statement is only syntax, used during semantic analysis to deliver warnings about the signature (number and type of arguments) of calls to imported methods.

There are many C functions that are compatible with Coffee, though there are many that are not (e.g. anything that uses pointers / dereferencing).

The most common functions you will use for the exercises while learning Coffee are:

pow	arg1: float, arg2: float, returns: arg1 raised to power arg2
sqrt	arg: float, returns: square root of arg
rand	no args, returns: random integer between 0 and 2^31-1
srand	arg: int, returns: void (sets the random seed to arg)
time	arg: pointer (zero), returns: # seconds since Jan 1st 1970
printf	arg1: string, other args: mixed type, returns: length

Example use of pow function:

```
import pow;
return (int)pow(2., 3.);
```

Expected response:

```
program exited with code: (8)
```

**Note the type cast to int as pow returns float. Also notice the arguments are floats – we do not (rather, the compiler does not) know the signature so we have to be explicit in the types, else the wrong registers will be used.