# Learn to Brogram

At its most basic mode of operation, Bro only provides an impartial view/understanding of network protocols that it parses. But if you want to understand and detect higher-level patterns/behaviors/anomalies in network traffic, then make systematic, consistent, and reproducible decisions about it, a programming language that lends itself to expressing network fundamentals/primitives (e.g. IP addresses, subnets) will help. And Bro has just such a helpful programming language, which is likely what you will be interacting with if you ever want to customize how Bro operates.

## A) Basic Concepts

Basic concepts of Bro's language are covered here. If you're already familiar enough, feeling adventurous, or just itching to get your hands dirty with more interesting programming tasks, feel free to skip ahead and come back for reference or if you get stuck. Example code for much of the language is provided for the sake of thoroughness and so there's enough syntax shown to be able to complete the exercises. Unless you think you need to make modifications to test your understanding, feel free to just read examples and move on instead of executing each one with Bro.

### Hello, World

Fine, get it out of the way:

```
# This line is a comment.
# The next line prints exactly what you expect to stdout.
print "Hello, world.";
```

### Variables

You can assign arbitrary data to a variable in order to store it for later use.

```
global x = "Hi";
print x;
x = "Bye";
print x;

const y = "Hello";
# Changing value of 'y' is not allowed.
#y = "Nope";

local z = "Hmm";
print z;
```

A 'local' variable differs from a 'global' in that its scope is restricted to the body of a function if its declared within one and will be assigned an initial value each time the function body is executed (functions are covered later). You typically won't see locals declared outside the body of functions unless it's example code or possibly work-in-progress code that may later be moved in to a function, but is currently freely floating inside some small script just for quick testing purposes (like some examples you're about to see).

## Modules (Namespaces)

Bro implements namespacing with the `module` and `export` keywords.
Every script implicitly starts out in a module named "GLOBAL". Identifiers declared within that module are accessible by any other module. If a script changes the current module, any identifiers that are subsequently declared are only accessible by other modules if they occur within an `export` block.

```
module MyModule;

export {
    # Accessible by other modules.
    global my_public_var = "my_public";
}

# Only accessible when current module is MyModule.
global my_private_var = "my_private";

module AnotherModule;

# Only accessible when current module is AnotherModule.
global my_private_var = "another_private";

# Access something in another module requires the scoping.
print MyModule::my_public_var;

# Attempting to access an unexported identifier
# from another module is
# a parse-time error.
#print MyModule::my_private_var;

# Accessing something within the current
# module does not require scoping.
print my_private_var;
print AnotherModule::my_private_var;
```

## Operators

To manipulate, inspect or compare data, you use operators.

### Arithmetic Operators

| **Name** | **Syntax** | **Example Usage** |
|---|---|---|
| Addition | `a + b` | `print 2 + 2;    # 4` |
| Subtraction | `a - b` | `print 2 - 2;    # 0` |
| Multiplication | `a * b` | `print 4 * 4;    # 16` |
| Division | `a / b` | `print 15 / 3;   # 5` |
| Modulo | `a % b` | `print 18 % 15; # 3` |
| Unary Plus | `+a` | `local a = +1;   # Force use of a signed integer` |
| Unary Minus | `-a` | `local a = 5; print -a; # -5` |
| Increment | `++a` | `local a = 1; print ++a, a; # 2, 2` |
| Decrement | `--a` | `local a = 2; print --a, a; # 1, 1` |

### Assignment Operators

| **Name** | **Syntax** | **Example Usage** |
|---|---|---|
| Assignment | `a = b` | `local a = 7;` |
| Addition assignment | `a += b` | `local a = 7; a += 2;` |
| Subtraction assignment | `a -= b` | `local a = 7; a -= 2;` |

### Relational Operators

| **Name** | **Syntax** | **Example Usage** |
|---|---|---|
| Equality | `a == b` | `print 2 == 2; # T` |
| Inequality | `a != b` | `print 2 != 2; # F` |
| Less | `a < b` | `print 2 < 3;  # T` |
| Less or Equal | `a <= b` | `print 2 <= 2; # T` |
| Greater | `a > b` | `print 2 > 3;  # F` |
| Greater or Equal | `a >= b` | `print 2 >= 2; # T` |

### Logical Operators

| **Name** | **Syntax** | **Example Usage** |
|---|---|---|
| Logical NOT | `! a` | `print !T;      # F` |
| Logical AND | `a && b` | `print T && F; # F` |
| Logical OR | `a \|\| b` | `print F \|\| T; # T` |

### Other Operators

| Name | Syntax | Example Usage |
|---|---|---|
| Member Inclusion | `a in b` | `print "z" in "test"; # F` |
| Member Exclusion | `a !in b` | `print "z" !in "test"; # T` |
| Size/Length | `\|a\|` | `print \|"test"\|; # 4` |
| Absolute Value | `\|a\|` | `print \|-5\|; # 5` |
| Index | `a[i]` | `print "test"[2]; # s` |
| String Slicing | `a[i:j], a[i:], a[:j]` | `print "testing"[2:4]; # st` |

## Logic/Control

You're going to want conditional branching and control flow to accomplish more complicated tasks.

### For Loop

Bro uses a "foreach" style loop.

```
for ( c in "abc" )
    print c;
```

### If Statement

If statements conditionally execute another statement or block of statements.

```
local x = "3";

for ( c in "12345" )
    {
    if ( c == x )
        {
        print "Found it.";
        # A preview of functions: fmt() does substitutions,
            outputs result.
        print fmt("And by 'it', I mean %s.", x);
        }
    else
        # A quick way to print multiple things on one line.
        print "I'm looking for", x, "not", c;
    }
```

There's enough blueprints about the language by this point to start solving more "interesting" problems, so we'll start practicing.

## Exercise 1

**Level beginner**
Write a program to remove every letter "e" from an arbitrary string of your choosing (does not have to be done in-place).

### Switch Statement

Sometimes a switch statement is a more convenient way to organize code. For example, consider a switch instead of large chains of "else if" blocks, or possibly if there's a large chain of OR'd conditions. The syntax looks like:

```
local x = 4;

switch ( x ) {
case 0:
    # This block only executes if x is 0.
    print "case 0";
    break;
case 1, 2, 3:
    # This block executes if any of the case labels match.
    print "case 1, 2, 3";
    break;
case 4:
    print "case 4 and ...";
    # Block ending in the "fallthrough" also execute subsequent
         case.
    fallthrough;
case 5:
    # This block may execute if x is 4 or 5.
    print "case 5";
    break;
default:
    # This block executed if no other case matches.
    print "default case";
    break;
}
```

## Exercise 2

**Level beginner**
Write a program (that relies on a switch statement) to count the number of vowels (a, e, i, o, u) in an arbitrary string of your choosing.

## Data Types

Bro has a static type system (the type of data a variable holds is fixed) with type inference (e.g. `local x = 0` is equivalent to `local x: count = 0`) and implicit type promotion/coercion (limited to numeric types or records with optional/default fields).

### Primitive Types

- `bool` - a value that's either true (`T`) or false (`F`).
- `double` - a double-precision floating-point value.
- `int` - a signed 64-bit integer. May be automatically promoted to a `double` when needed.
- `count` - an unsigned 64-bit integer. May be automatically promoted to an `int` or `double` when needed.
- `time` - an absolute point in time (note the only way to create an arbitrary time value is via the `double_to_time(d)`, with `d` being a variable of type `double`).
- `interval` - a relative unit of time. Known units are `usec`, `msec`, `sec`, `min`, `hr`, or `day` (any may be pluralized by adding "s" to the end). Examples: `3secs`, `-1min`.
- `port` - a transport-level port number. Examples: `80/tcp`, `53/udp`.
- `addr` - an IP address. Examples: `1.2.3.4`, `[2001:db8::1]`.
- `subnet` - a set of IP addresses with a common prefix. Example: `192.168.0.0/16`. Note that the `/` operator used on an address as the left operand produces a subnet mask of bit-width equal to the value of the right operand.
- `enum` - a user-defined type specifying a set of related values.

  ```
  type Color: enum { Red, Green, Blue, };
  ```

- `string` - character-string values.
- `pattern` - a regular expression using flex's syntax. Some examples:

  ```
  print /one|two|three/ == "two";  # T
  print /one|two|three/ == "ones"; # F (exact
      matching)
  print /one|two|three/ in "ones"; # T (embedded
      matching)
  print /[123].*/ == "2 two";  # T
  print /[123].*/ == "4 four"; # F
  ```

## Exercise 3

### Level beginner

Write a program (that relies on pattern matching) to count the number of vowels (a, e, i, o, u) in an arbitrary string of your choosing.

**Composite Types**

- `set` - a collection of unique values. Set uses the `add` and `delete` operators to add and remove elements itself and the `in` operator for querying membership:

```
local x: set[string] = { "one", "two", "three" };
add x["four"];
print "four" in x; # T
delete x["two"];
print "two" !in x; # T
add x["one"]; # x is unmodified since 1 is already
    a member.
for ( e in x ) print e;
```

- `table` - an associative collection that maps a set of unique indices to other values. Like sets, the `delete` operator is used to remove elements, however, adding elements is done just by assigning to an index:

```
local x: table[count] of string = { [1] = "one",
    [3] = "three",
                                    [5] = "five" };
x[7] = "seven";
print 7 in x; # T
print x[7]; # seven
delete x[3];
print 3 !in x; # T
x[1] = "1"; # changed the value at index 1
for ( key in x ) print key;
```

- `vector` - a collection of values with 0-based indexing.

```
local x: vector of string = { "one", "two", "three"
     };
print x; # [one, two, three]
print x[1]; # two
x[|x|] = "one";
print x; # [one, two, three, one]
for ( i in x ) print i;  # Iterates over indices.
```

- `record` - a user-defined collection of named values of heterogeneous types. Fields are dereferenced via the `$` operator (`.`, as used in other languages is ambiguous in Bro because of IPv4 address literals). Optional field existence is checked via the `?$` operator.

```
type MyRecord: record {
    a: string;
    b: count;
    c: bool &default = T;
    d: int &optional;
};

local x = MyRecord($a = "vvvvvv", $b = 6, $c = F
    , $d = -13);
if ( x?$d ) print x$d;
x = MyRecord($a = "abc", $b = 3);
print x$c;  # T (default value of the field)
print x?$d; # F (optional field was not set)
```

- `function`

  Example of function syntax/usage:

  ```
  # Optional function declaration.
  # Takes one required string argument
  # and another optional string argument
  # and returns a string value.
  global emphasize: function(s: string, p: string &default =
      "*"): string;

  # Function implementation.
  function emphasize(s: string, p: string &default = "*"):
      string
      {
      return p + s + p;
      }

  # Function calls.
  print emphasize("yes");
  print emphasize("no", "_");
  ```

- `event`

  Events are a special flavor of functions that Bro frequently uses. They differ from functions in the following ways:

  - They may be scheduled and executed at a later time, so that their effects may not be realized directly after they are invoked.
  - They return no value -- they can't since they're not called directly but rather scheduled for later execution.
  - Multiple bodies can be defined for the same event, each one is deemed an "event handler", and when it comes time to execute an event, all handler bodies for that event are executed in order of &priority.

  ```
  # Optional event declaration.
  global myevent: event(s: string);

  global n = 0;

  # Event handler implementation.
  # &priority attribute is optional and
  # may be used to influence the order in which
  # event handler bodies execute.
  # If omitted, &priority is implicitly 0.
  event myevent(s: string) &priority = -10
      {
      ++n;
      }
  ```

```
# Another event handler.  &priority can be used
# to influence the order
# in which event handler bodies are executed.
event myevent(s: string) &priority = 10
    {
    print "myevent", s, n;
    }

# "bro_init" is a special event that's executed once when
    Bro starts.
event bro_init()
    {
    print "bro_init()";
    # Schedule an event to execute as soon as possible.
    event myevent("hi");
    # Schedule an event to execute 5 seconds in
    # the future (or upon
    # Bro shutting down, whichever is sooner).
    schedule 5 sec { myevent("bye") };
    }

# Another special event executed when Bro is shutting down.
event bro_done()
    {
    print "bro_done()";
    }
```

- `hook`

  Hooks are yet another flavor of function. They are most like events, but with a few main differences:

  - They do execute immediately when invoked (i.e. they're not scheduled like events).
  - It matters how the body of a hook handler terminates. If the end of the body or a `return` statement is reached, remaining hook handlers will be executed. If a hook handler body terminates due to a `break` statement being reached, no remaining hook handlers are executed.

```
# Optional hook declaration.
global myhook: hook(s: string);

# Hook handler definition.
# &priority is optional, and implicitly 0
# if omitted.
hook myhook(s: string) &priority = 10
    {
    print "priority 10 myhook handler", s;
    # Arguments may be modified and is visible
    # to remaining handlers.
    s = "bye";
    }
hook myhook(s: string)
    {
    print "break out of myhook handling", s;
    break;
    }

hook myhook(s: string) &priority = -5
    {
    print "not going to happen", s;
    }

# Hook invocation.  Return value is implicitly
# a boolean with a value of
# true if all defined hook handlers were executed
# (i.e. no body exited
# as a result of a break statement).
local ret: bool = hook myhook("hi");

if ( ret )
    print "all handlers ran";
```

## Redefinitions (and Attributes)

Bro supports redefining constants, but only at parse-time, not at run-time. This feature may not be that useful when writing your own scripts for private usage, but it's the suggested way for script authors to advertise "knobs and switches" that one may choose to configure. These are usually values that one doesn't want to accidentally modify while Bro is running, but that the author either can't know ahead of time (e.g. local IP addresses of interest), may differ across environments (e.g. trusted SSL certificates), or may vary/evolve over time (e.g. a list of known cipher suites).

```
const TWO = 2;
const PI = 3.14 &redef;
redef PI = 3.1415;
#redef TWO = 1; # not allowed
#PI = 5.5;      # not allowed
print PI;
print TWO;
```

Normally, the declaration and the `redef` would live in different scripts (e.g. the declaration in a script from the "standard library" that comes with Bro and the `redef` in the script you write), but this is just an example.
Also, the `&redef` is something called an attribute. It simply marks the identifier as one that can be altered via a `redef` statement. There are other types of attributes, but their behavior/function ranges from fairly obvious to rather advanced, so they're not covered extensively here. To see all possible attributes please see the script reference.
One last thing: `redef` not only works with values, but also certain *types*. Namely `record` and `enum` may be extended:

```
type MyRecord: record {
    a: string &default="hi";
    b: count  &default=7;
} &redef;
```

```
redef record MyRecord += {
    c: bool &optional;
    d: bool &default=F;
    #e: bool; # Not allowed, must be &optional or &default.
};

print MyRecord();
print MyRecord($c=T)

type Color: enum { Red, Green, Blue } &redef;
redef enum Color += { Brorange };
```

## B) FizzBuzz

Oh, you thought you'd start right off programming something useful/applicable? Nope, but if it pops up at a job interview you'll be able to provide a Bro-style answer.

### Exercise 4

**Level intermediate**
Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

## C) More FizzBuzz

You may think that re-using the premise of an exercise seems lazy, and you're right. But it's also true that a programmer often has to revisit previous work in light of new considerations/requirements. Or is that just a rationalization? Either way...

### Exercise 5

**Level intermediate**
Reimplement the FizzBuzz algorithm, but do it in a different way than you did before. Hint: there's recursive (function or event based), looping, lookup table, or `switch` based approaches that are all significantly different.

## D) Frequency Analysis Module

For a change of pace, let's try writing code that we can imagine someone actually finding useful.

### Exercise 6

**Level advanced**
Write a script that implements a Bro module for frequency analysis that can track how often certain letters occur within some arbitrary input text. At a minimum it should

be able to track how many times any given letter appears in the input text compared to the total amount of letters in the input text, i.e. the "frequencies". Having code to sort frequency statistics would also be nice. Feel free to come up with your own additional requirements. As a hint to get you started, you will probably want to make use of `record` and `table` types to keep track of frequency statistics, but the main portion of your code will likely involve a `for` loop that examines every letter of an input `string`.

## Exercise 7

**Level advanced**
Can you generalize your code to arbitrary n-gram frequency analysis? An n-gram being a contiguous sequence of 'n' bytes/characters.

## Exercise 8

**Level advanced+**
Can you come up with a method for detecting whether a given frequency distribution compares "closely" with some representative letter frequency distribution of the English language?