RAPID Language Reference Manual

Ben Edelstein, Brian Shin, Brendon Fish, Dan Schlosser, Nate Brennand

Table of Contents

10.1 Flags

1. Introduction

With increased demand in the public and private sector for cloud-connected mobile and web applications has come a rising need for web servers to maintain state across multiple devices and users. Development of web servers is complex, however. Building a web server using modern web server packages requires learning a server-side programming language, and then integrating a web server package and implementing required methods. Furthermore, local testing and development of these servers is excessively complex, as they have numerous dependencies and are difficult to build.

RAPID is a programming language intended specifically for the rapid development of modern web APIs. Using RAPID, developers can quickly build a database-backed REST API server that guarantees JSON shapes in responses. RAPID is object oriented and database-backed, meaning that classes represent an SQL table, and upon instantiation objects are automatically saved in the database. This abstracts away much of the boiler plate code that developers typically write when building an API server.

1.1 Why RAPID?

The name RAPID represents the goal of the language: making API server development quick. Also, it's a recursive acronym for *RAPID Application Programmer Interface Dialect*.

1.2 RAPID Programs

There are two types of RAPID programs, servers and scripts. If a program contains an HTTP method, it is a server, otherwise it is a script. (See more in later sections).

2. Types

2.1 Static Typing

RAPID is a statically typed language; variables must be explicitly typed upon declaration. Variables can be cast to other types (see Casting).

2.2 Primitive Types

null

In RAPID, the `null` keyword represents an uninitialized value. Any type in rapid may take on `null` if it hasn't been initialized, or otherwise doesn't exist.

Booleans

Boolean values are defined by the `true` and `false` keywords. Because they are their own type, non-boolean values must be cast to `boolean` in order to be used in logical expressions.

For example:

```
!(3+5)? // valid RAPID

!(3+5) // not valid RAPID
```

The `?` is a an operator on all primitive types that evaluates to the "truthiness" of that value.

Integers

Integers are preceded by the type `int`, and represent an 8 byte, signed integer. Integers can be declared and initialized later, or initialized inline. Uninitialized integers are null.

```
int i     // null
int i = 5 // 5
```

Integers are copied by value.

```
int a = 1
int b = a
a = 2
printf("%d, %d", a, b) // 2 1
```

Floating Point Numbers

Floating point numbers are preceded by the type `float`, and represent IEEE-754 64-bit floating-point numbers.. They can be declared and initialized later, or initialized inline.

```
float i        // null
float j = 3.14 // 3.14
```

Strings

Strings in RAPID are mutable, and declared with the `string` type, and have the default value of the empty string. String literals are declared using double quotes, and special characters may be escaped using the `\` character. Strings may be indexed using square brackets. Because there is no Character type in RAPID, single characters are strings of length 1. Multiline strings may be declared using triple double quotes. Newlines are preserved and quotes do not need to be escaped, but they may not be nested. Strings are pass by value.

```
string s                      // null
string character = "c"        // c
string s = "He is \"Batman\"" // He called himself "Batman"
string c = s[0]               // H
string multi = """
Did you hear?

He calls himself "Batman".
"""                           // multi[0] => "\n"
```

2.3 Non-Primitive Types

List

The `list` type is a zero-indexed array that expands to fit it's contents. The type of the contents must be provided within angle brackets in the type signature. RAPID `list` literals may be declared using square brackets, and values may be accessed or set using square brackets. Uninitialized lists default to the empty list. Lists are pass by reference.

```
/* List declaration */

list< /* type */ > /* id */ = [
    /* expression */,
    /* expression */,
    ...
    /* expression */
]
```

```
list<int> empty // []
list<int> numbers = [1,2,3,42]
numbers[3]      // 42
numbers[1] = 5  // [1,5,3,42]
```

Dictionary

The `dict` type is similar to an object, but it's key set is mutable. The type of the key and value must be provided within angle brackets in the type signature. Only primitive types may be used as keys. Keys may be added, set, and accessed using square brackets. RAPID `dict` literals may be declared as comma-separated `key:value` pairs surrounded by braces. Uninitialized dictionaries default to the empty dictionary. Dictionaries are pass by reference.

```
/* Dictionary declaration */

dict< /* type */ , /* type */ > /* id */ = {
    /* expression:string */ : /* expression */,
    /* expression:string */ : /* expression */,
    ...
    /* expression:string */ : /* expression */
}
```

```
dict<string, int> empty // {}
dict<string, int> ages = {"Alice":3, "Bob":5}
ages["Alice"]           // 3
ages["Bob"] = 4         // {"Alice":3, "Bob":4}
ages["Caroline"] = 7    // {"Alice":3, "Bob":4, "Caroline":7}
```

Object

The `object` type is a generic, non-primitive, dictionary-backed type that has attributes for instance variables and functions. Accessing instance variables or functions can be done with dot notation. Objects may not be declared anonymously; they must be declared as instances of classes. Objects have immutable key sets, so variables and functions may not be added or removed, although their values may be changed. For more on classes and instantiation, see Classes.

json

The `json` type is shares qualities of a dictionary and of an Object. Every `json` type is directly connected to an Object class that is user-defined. They have keys and values like dictionaries, but have the strict requirements of shape like objects do. Every property of a class is a mandatory key on the corresponding `json` object, and properties that have default values on objects have default values in `json`. Unlike objects, however, `json` objects do not have methods associated with them, and instances do not represent rows in the database. Each `class` declaration defines an Object type and a `json` object type, and only `json` objects that are associated with classes may be instantiated.

For example, if we previously defined a `User` object with three instance variables `username`, `full_name`, and `password` (all strings), then we may declare a `json` User like so:

```
/* JSON Object initialization */

json< /* id:classname */ > /* id */ = json< /* id:classname */ >(
    key = /* expression */,
    key = /* expression */,
    ...
    key = /* expression */
)
```

```
json<User> steve = json<User>(
    username="sedwards",
    full_name="Stephen Edwards",
    password="easypeasy"
)
```

Errors

Errors in RAPID are not thrown and caught, rather they are returned directly by unsafe functions (see Functions). Errors contain a string message, which can be dot-accessed, an integer error code that conforms with the HTTP/1.1 standard, and an optional string name.

For example, to declare a custom error:

```
error e = error(message="There was an error with that Request.",
                code=400,
                name="RequestError")
```

Unsafe operations return an error as the last return value:

```
dict<string, int> d = {"foo": 4, "bar": 5}
```

```
int val, error e = d["baz"]
if (!e?) {
    printf("%d\n", e.code)    // 500
    printf("%s\n", e.message) // Key error when accessing "baz" on `d`.
    printf("%s\n", e.name)    // KeyError
}
```

Many standard library classes and builtin objects define errors pertinent to their functions, to which an error instance may be compared.

```
dict<string, int> d = {"foo": 4, "bar": 5}
int val, error e = d["baz"]
if (!e?) {
    printf("%s", e == dict.KeyError) // true
}
```

Stacking

Unsafe functions (like list and dictionary access) may exist in the same expression. If unsafe functions return successfully, the error that is returned is consumed (ignored), and the return value is taken. If an unsafe function returns an error, the expression evaluation short-circuits, and the value of the expression is null and the error that is returned by the failed function call.

```
dict<string, list<int>> d = {"foo": [4,5], "bar": [1,2,3]}
int val, error e = d["foo"][2]     // List index out of bounds...
printf("%d", val)                  // null
printf("%s", e.name)               // IndexError
printf("%t", e == list.IndexError) // true

val, e = d["baz"][0]             // No such key, short circuit
printf("%d", val)                // null
printf("%s", e.name)             // KeyError
printf("%t", e == dict.KeyError) // true
```

More generally, if a subexpression of an expression is unsafe, it is presumed to be successful and the return value of the subexpression is used in the evaluation of the larger expression, unless the unsafe expression evaluates to an error, in which case evaluation of the large expression short-circuits, and the value of the large expression is `null, /* sub-expression's error */`.

Predefined Responses

Unsafe functions may also choose to return an predefined response, which is an predefined literal that will be cast to a generic error object at compile time.
See Functions for more details.

All predefined errors exist in the root scope and are named according to their status code, `e<code>`.

`e404` is the error, `error("Not Found", 404, "NotFound")` (message, code, name).
All the below errors are predefined as such.

Response	Message	Code	Name
`e100`	Continue	100	`Continue`
`e200`	OK	200	`OK`
`e201`	Created	201	`Created`
`e301`	Moved Permanently	301	`MovedPermanently`
`e302`	Found	302	`Found`
`e304`	Not Modified	304	`NotModified`
`e400`	Bad Request	400	`BadRequest`
`e401`	Unauthorized	401	`Unauthorized`
`e403`	Forbidden	403	`Forbidden`
`e404`	Not Found	404	`NotFound`
`e405`	Method Not Allowed	405	`MethodNotAllowed`
`e410`	Gone	410	`Gone`
`e413`	Request Entity Too Large	413	`RequestEntityTooLarge`
`e414`	Request-URI Too Long	414	`RequestURITooLong`
`e417`	Expectation Failed	417	`ExpectationFailed`
`e500`	Internal Server Error	500	`InternalServerError`
`e501`	Not Implemented	501	`NotImplemented`
`e502`	Bad Gateway	502	`BadGateway`
`e503`	Service Unavailable	503	`ServiceUnavailable`
`e504`	Gateway Timeout	504	`GatewayTimeout`

Functions

Functions are first class objects, and may be passed around as variables (see Functions)

2.4 Casting

Integers and Floats

Casting between float and int can be done using the `float()` and `int()` keyword functions. Floats are floored when they are cast to int. Additionally, integers are cast to floats if floats and integers are used together in a binary operator.

```
float f = 7.5
int i = 3
float f = float(i) // f == 3.0
int i = int(f)     // i == 7
```

Booleans

Any value may be cast to boolean using the `?` operator.

See the following table for the result of using the `?` operator on various types:

Type	**`true`**	**`false`**	**Comments**
boolean	`true?`	`false?`	Booleans retain their value
int	`1?`, `-1?`	`0?`	0 is `false`, other ints are `true`
float	`1.0?`, `-1.0?`	`0.0?`	0.0 is `false`, other floats are `true`
null	-	`null?`	`null` is `false`
list	`[0]?`, `[false]?`	`[]?`	Empty lists are `false`
dict	`{"key":false}?`	`{}?`	Empty dicts are `false`
json	`json<Obj>()?`	-	JSON objects are `true`
object	`Obj()?`	-	Objects are `true`

3. Lexical Conventions

3.1 Identifiers

Identifiers must start with a letter or an underscore, followed by any combination of letters, numbers, and underscores.

> **Valid Identifiers:**
>
> `abc`, `abc_def`, `a___1`, `__a__`, `_1`, `ABC`
>
> **Invalid Identifiers:**
>
> `123`, `abc-def`, `1abc`, `ab\ cd`

3.2 Keywords

The following identifiers are keywords in RAPID, and are reserved. They can not be used for any other purpose.

`if`, `else`, `for`, `in`, `while`, `switch`, `case`, `default`, `fallthrough`, `http`, `func`, `json`, `class`, `namespace`, `param`, `true`, `false`, `new`, `optional`, `unsafe`, `instance`

3.3 Literals

Integer literals

Integer literals may be declared using digits.

```
int x = 5
```

Float literals

Float literals are declared as an integer part, a decimal point, and a fraction part, all of which are mandatory. The integer part may not start with a zero, unless it is only a zero (for floats less than `1.0`), in which case it is still

required. There may not be any whitespace between these three parts.

```
// Valid float literals:
float x = 15.0
float y = 0.25

// Invalid float literals:
float z = .5
float w = 10.
float v = 1 . 4
```

String literals

String literals are declared using double quotes. Special characters may be declared using the `\` escape character.

```
string a = "hello"
string b = " \"world\"\n"
```

Boolean literals

Boolean literals are declared using the `true` and `false` keywords.

```
boolean t = true
boolean f = false
```

List literals

List literals may be declared between square brackets, with comma-separated values.

```
list<int> a = [1,2,3,4]
list<string> b = ["hello", "world"]
```

Dictionary literals

Dictionary literals may be declared as comma-separated key value pairs between braces, with a colon separating the key and value. Whitespace is ignored.

```
dict<string, int> a = {"hello": 4, "world": 5}
dict<string, ing> b = {
```

```
    "a": 42,
    "b": 27
}
```

3.4 Comments

There are two types of comments in RAPID: single line comments, and block comments. Single line comments are preceded by `//` and block comments begin with `/*` and end with `*/`

```
// This is a single line comment

/*
This is a multi-line
comment
/* They may be nested */
*/
```

3.5 Operators

Operator	Use	Associativity
`+`	Addition	left
`*`	Multiplication	left
`/`	Division	left
`-`	Subtraction	left
`%`	Modulus	left
`=`	Assignment	non-associative
`==`	Equal to	non-associative
`!=`	Not equal to	non-associative
`>`	Greater than	non-associative
`<`	Less than	non-associative
`>=`	Greater than or equal to	non-associative

Operator	Use	Associativity
`<=`	Less than or equal to	non-associative

4. Database Backing

4.1 Classes

RAPID classes are backed by a PostgreSQL database.
Classes are defined using the `class` keyword, and represent an SQL table.
Instance variables (variables declared directly within the `class` block) represent columns for the table.
Instances of a class represent rows in SQL.
By default, columns are not nullable, but this may be overwritten using the `optional` keyword.
If the assignment syntax is used, a default value will be given.

```
class /* id */ {
    /* declaration */
    /* declaration */
    ...
    /* declaration */
}
```

Take the following example:

```
class User {
    string username
    optional string full_name
    int age = 18
    string password
}
```

In this example, the “User” table has four columns: `username`, `full_name`, `age`, and `password`. The `full_name` column may be omitted in the instantiation, and if `age` is omitted, it will take the value `18`.

Instance Methods

Instances of objects may have methods that operate on their instances variables. Using the `instance` keyword, a block may be created in which instance methods may be defined:

```
class /* id:classname */ {
    instance /* id:selfname */ {
        /* declaration */
        /* declaration */
        ...
        /* declaration */
```

```
    }
}
```

The identifier specified after the `instance` keyword will represent the instance in all functions defined inside the `instance` block.

Instance methods may not be `http` routes. The `.` is syntactic sugar for calling instance methods.

For example:

```
class User {
    string first_name
    string last_name

    instance my {
        func get_full_name() string {
            return my.first_name + " " + my.last_name
        }
    }
}

User obama = User(first_name="Barrak", last_name="Obama")
printf("%s", obama.get_full_name()) // Barrak Obama
```

Instantiation

New instances of a class may be declared using the `new` keyword.
The `new` keyword is followed by the name of the class and a pair of parenthesis, in which a JSON User literal (described more in-depth in the next section) may be passed to declare instance variables.
Once a user defined object is created, it will be database backed.
Any changes to the object will trigger an update to the database backed copy.
Every object will have an `ID` attribute generated (this is a reserved attribute that cannot be used).
This is a unique identifier to the object.

```
User bob = new User(
    username="burgerbob",
    full_name="Bob Belcher",
    password="burgersrock",
    age=42
)
```

4.2 Deletion

All objects have an instance method, `delete()`, defined that will delete the database record.
There is no return value for the `delete` call.

json

Defining the “User” class defines a `User` type, as well as a `json<User>` type. The `json<User>` type has the same keys and value types as the User class, and may be declared in dictionary literal syntax.

```
json<User> bob_json = json<User>(
    username="burgerbob",
    full_name="Bob Belcher",
    password="burgersrock",
    age=42
)
```

This `json<User>` object does not represent a row in the database, and will be deallocated when it leaves scope.

It may be passed into an instantiation statement for a User object, to be persisted:

```
User bob, error e = new User(bob_json)
```

4.3 Querying

Objects may be queried from the database using the `get` function, which is automatically defined on all classes.
`get` is an unsafe function which will return a list of objects as well as an error.

The following example queries all User objects from the database:

```
Tweet[] tweets = Tweet.get()
```

A optional `filter` parameter can be set to limit the responses returned by Get().
The filter value shoudld be a dictionary of attributes and values.
Any non-existant attributes in the dictionary will be logged as an warning and ignored.

```
// returns all tweets by burgerbob.
Tweet[] tweets, error e = Tweet.get(filter={
  username="burgerbob"
})
```

A optional `ID` parameter can be set to return a single Object.
This can be combined with filter if desired.

In the case that the object is not found, the returned object will be `null` and the error will be a non-null value.

```
// returns all tweets by burgerbob.
Tweet t, error e = Tweet.get(ID="123abc")
```

5. Functions

5.1 Declaration

Functions in RAPID are first-class objects, but may not be declared anonymously. Functions are declared using the `func` keyword. The arguments (within parenthesis), return type (after the parenthesis, but before the braces), and the body of the function (within the braces) must be declared explicitly. Return types may include multiple types separated by commas, or may be omitted for void functions.

Return values are specified using the `return` keyword, which must be followed by an expression to be returned for functions that have declared return types. If the return type is omitted, the function is void, an the result of calling it may not be assigned to a value. Void functions may use the `return` keyword by itself to exit prematurely from the function.

Unsafe functions may not be void, because they must return errors.

```
return /* expression */
```

The arguments must be in order `namespace` arguments, then formal arguments.

```
[unsafe] func /* id */ ( /* namespace args */ /* formal args */ ) {
    // statements
}
```

For example:

```
func sum(int a, int b) int {
    return a + b
}
```

Or:

```
func printInt(int a) {
    printf("%d", a)
}
```

5.2 Unsafe Functions

If a function performs actions that may be unsafe, it must be preceded by the keyword `unsafe`. Unsafe functions

return unsafe expressions, which is denoted by the presence of an `error`-typed second value that is returned.

```
unsafe func access(dict<string, int> d, string key) int {
    int val, error error = d[key]
    return val, error
}
```

Notice that the return type remains `int`, although an error is also returned. For more on unsafe expressions, see Expressions.

Unsafe functions may also return a error, which are integer literals that will be cast to a generic error object at compile time. See Status Code Definitions for a complete list of error codes that may be declared as anonymous errors.

```
/* Default dict accessing:
 *    If there is a KeyError, return 0 with a 400 Not Found error
 */
unsafe func access(dict<string, int> d, string key) int {
    int val, error error = d[key]
    if (error == dict.KeyError) {
        return 0, e400
    }
    return val, e200
}
```

6. Routing

One of the core features of RAPID is it's ability to easily define routes for a REST API server.

6.1 Declaring Routes

Routes may be declared like functions, but substituting the `http` keyword for the `func` keyword. Routes specify a REST API endpoint, it's parameters, it's response, and any side effects.

Like functions, routes take namespace arguments, and then other formal arguments. Unlike functions, however, routes may also take a single request body argument that of a `json<Obj>` type. It will be read from the request body and interpreted as JSON.

```
http /* id */ ( /* namespace args */ /* formal args */ /* request body args */) {
    // statements
}
```

Routes are unsafe by default, and therefore must include `error` in their return types. This may be an anonymous error (see Functions).

For example, the following route echos the URL parameter that it is passed.

```
http echo(string foo) string, error {
    return foo, e200
}
```

The name of the function will be the name of the route. Therefore, in the preceding example, a `GET` request to `/echo?foo=Dog` will return `"Dog"`.

6.2 Path context

The endpoint associated with each route is determined by the combination of one or more blocks associated with it and the name of the route itself. There is a one-to-one mapping from any route to a series of accessors on class instances.

Classes

Classes provide path context. Class names are put to lowercase, and appended to path context. The following

example defines a route named `add` inside a class called `Math`.

```
class Math {
    http add(int a, int b) int {
        return a + b, e200
    }
}
```

A `GET` request to `/math/add?a=3&b=4` will return `7`.

Similarly, the following code will print `7`:

```
math = Math()
int sum, error _ = math.add(3,4)
printf("%d", sum)
```

Namespaces

Sometimes, functions or routes should be grouped together for organization purposes, rather than any functional purpose. The `namespace` keyword defines a named block of functions that has the namespace name appended to the path context for those functions.

```
/* Namespace declaration */

namespace /* id */ {
    // statements
}
```

```
class Math {
    namespace ops {
        http add(int a, int b) int { return a + b, e200 }
        http sub(int a, int b) int { return a - b, e200 }
    }
    namespace convert {
        func ft_to_in(float feet) float { return feet*12, e200 }
    }
}
```

This defines routes at `/math/ops/add` and `/math/ops/sub`, and functions at `Math.ops.add`, `Math.ops.sub`, and `Math.convert.ft_to_in`.

A `GET` request to `/math/ops/add?a=3&b=4` will return `7`.

Parameters

Variable URL parameters may be defined similar to namespaces, using a named block with the `param` keyword. The `param` keyword is followed by a type and an identifier.

Any function or route defined within a `param` block must take the parameters defined by the `param` blocks in order from inside to out.

```
param /* type */ /* id */ {
    // statements
}
```

For example:

```
class Math {
    param int a {
        param int b {
            http add(int a, int b) int { return a + b, e200 }
        }
        http square(int a) int { return a*a, e200 }
    }
}
```

A `GET` request to `/math/5/7/add` will return `12`, and a `GET` request to `/math/5/square` will return `25`. A `GET` request to `/math/5/7/add?a=4` will return return a 400 HTTP error. The following code snipped will print `12` then `25`:

```
math = Math()
int sum, error _ = math.add(5,7)
printf("%d", sum)
int sqr, error _ = math.square(5)
printf("%d", sqr)
```

7. Syntax

7.1 Program Structure

A valid RAPID program is a series of valid statements. If the program contains any `http` blocks, it will be interpreted as a restful web API, and will run a HTTP web server on `localhost:5000`.

7.2 Expressions

Expressions are series of operators and operands that may be evaluated to a value and type. Any subexpressions are evaluated from left to right, and side effects of evaluations occur by the time the evaluation is complete. Type checking on operations occur in compile time.

Constants

Constants may be string, integer, float, or boolean, dict, list, or JSON object literals. See Lexical Conventions for more information.

Identifiers

Identifiers could be primitive types, lists, dictionaries, objects, JSON objects, functions, classes, or errors. Identifiers can be modified, and reused throughout a program.

For example, in the following example, the variable `a` changes value three times.

```
class a {} // `a` is a class
func a() void {} // `a` is a function, the class no longer exists.
int a = 5 // `a` is an int, the function no longer exists.
```

Identifiers are tied to the scope that they are declared in. The following example prints `3`, then `5`, then `3`:

```
int a = 3
if (true) {
    printf("%d", a) // `a` is from the parent scope.
    int a = 5
    printf("%d", a) // `a` is from the local scope.
}
```

```
printf("%d", a) // the `a` from within the block does not leave it
```

Binary Operators

Binary operators have two operands, one on the left side, and one on the right.

```
/* expression */ /* bin-op */ /* expression */
```

In the case of multiple consecutive binary operations without parenthesis, the association of the binary operator is followed (see Operators).

Parenthesized Expressions

Parenthesis may be used to alter the order of operand evaluation.

7.3 Statements

Assignments

Assignments have an `lvalue`, and an expression, separated by an equal sign. Possible `lvalue`s include identifiers, accessors (either list, dict, or object), a declaration, or another assignment:

```
/* lvalue */ = /* expression */
```

Examples include:

```
a = b
int i = 7
j = square(i)
k = 5 * b
```

Declarations

A declaration may be the declaration of a variable, an assignment, or the declaration of a function, route, class, namespace, or param.

Variable Declaration

A variable declaration consists of a type and an id.

```
/* type */ /* id */
```

Function Declaration

The declaration of a function is a valid statement (see Functions).

Route Declaration

The declaration of a class is a valid statement (see Routing).

Class Declaration

The declaration of a class is a valid statement (see Classes).

Namespace or Parameter Declaration

The declaration of a namespace or parameter is a valid statement (see Path Context).

Function call

A function call is an identifier of a declared function and a set of parenthesis containing the comma-separated arguments. There may not be a space between the identifier and the open parenthesis.

```
my_func(4,5)
int x = add(2, 6, 7)
```

Control flow

If

If the expression between the parenthesis of an `if` statement evaluates to `true`, then the statements within the body are executed. Note that non-boolean values will not be cast to boolean, and will result in a compile-time error.

```
if (/* expression */) { /* statements */ }
```

If-else

An `if` statement may be immediately followed by an `else` statement, in which case the block of code within the

braces after the `else` keyword will be executed if the `if`'s expression evaluates to `false`.

```
if (/* expression */) {
    // statements
}
else {
    // statements
}
```

Else-if

An `if` statement may be followed by an `else if` statement, in which case the the second `if` statement will be evaluated if and only if the first `if` statement evaluates to `false`. The body of the `else if` is executed if the second `if` statement is evaluated, and evaluates to `true`. An `else if` statement may be followed by another `else if` statement, or an `else` statement.

```
if (/* expression */) {
    // statements
}
else if (/* expression */) {
    // statements
}
...
else if (/* expression */ ) {
    // statements
}
else {
    // statements
}
```

Switch

A `switch` statement includes an expression, which is evaluated and then compared in order to a series of one or more `case` expressions. If the expressions are equal, the body of the `case` statement that matches will be executed, and then the switch statement will short circuit. The `fallthrough` keyword may be used to avoid this short circuit, continuing to compare the `switch` expression with subsequent `case` expressions.

The `default` statement may be included after all `case` statements, and will be executed if it is reached. This can be thought of as a `case` whose expression always equals that of the `switch`. Observe the syntax below:

```
switch (/* expression */) {
    case (/* expression */) {
        // statements
        fallthrough
    }
```

```
    case (/* expression */) {
        // statements
    }
    default {
        // statements
    }
}
```

While loops

While loops contain an expression and a body. If the expression evaluates to `true`, the body will be executed. Afterwards, the expression will be evaluated again, and the process repeats. Like `if` statements, `while` statements must have expressions that evaluate to a boolean in order to compile.

```
while (/* expression */) {
    // statements
}
```

For loops

A `for` loop may be used to iterate over a `list`. The syntax is:

```
for (/* type */ /* id */ in /* list expr */) {
    // statements
}
```

For example:

```
list<int> my_list = [1,2,3,4,5]
for (int num in my_list) {
    printf("%d ", num)
}
// 1 2 3 4 5
```

The `range()` function in the standard library may be used to generate lists of sequential integers.

```
for (int num in range(1,6)) {
    printf("%d ", num)
}
// 1 2 3 4 5
```

Return statements

A `return` statement may be used to exit a function, optionally passing the value of an expression as the return value of the function.

```
return /* optional expression */
```

For example:

```
func int add(int x, int y) int {
    return x + y
}
printf("%d", add(3,4))
// 7
```

8. Built-in Functions

8.1 length()

```
func length(string s) int
func length(list<T> l) int
func length(dict<T,S> d) int
func length(json<T> j) int
```

Returns the length of the argument. For strings, this is the number of characters in the string, for lists, this is the number of elements in the list. For dictionaries, this is the number of keys, for JSON objects, this is the number of keys.

Examples:

```
length("hello")                                 // 5
length([0,1,2,3])                               // 4
length({"a":0, "b":null, "c": False, "d": ""}) // 4
```

Taking the `length` of a `null` value will return `null`

8.2 range()

```
func range(int stop) int[]
func range(int start, int stop[, int step=1]) int[]
```

Returns a list of integers `r` where `r[i] = start + step*i` where `i>=0` and while `r[i] < stop`. If start is omitted, it defaults to 0. If step is omitted, it defaults to 1.

Step may be negative, in which
case `r[i] > stop`

Examples:

```
range(5)       // [1,2,3,4,5]
range(4,5)     // [4]
range(3,7,2)   // [3,5]
range(10,4,-2) // [10,8,6]
```

8.3 Output Functions

There are four methods of outputing from the server.
They all accept a format string as the first argument and optional additional arguments for format strings.

Print Functions

- `printf(String formatStr, [values])`

`printf` does not include a newline at the end of the output.
The output is directed to STDOUT.

8.4 Logging Functions

- `log.info(String formatStr, [values])`
- `log.warn(String formatStr, [values])`
- `log.error(String formatStr, [values])`

All output is preceded by a timestamp.
The logging functions print with a newline at the end.
The output is directed to STDERR.

The method name being called precedes the message in all caps.

```
log.info("Hello, %s", "world")
// 2009/11/10 23:00:00 INFO: Hello, world
```

9. Standard Library

9.1 string

string.is_empty()

```
func is_empty() boolean
```

Returns a boolean value of whether the string on which it is called is of length 0 or `null`.

Examples:

```
string a = "dog"
string b = ""

a.is_empty()  // false
b.is_empty()  // true
```

string.substring()

```
unsafe func substring(start, stop) string
```

Returns the substring of a string at the given indexes. The start and stop indexes are inclusive and exclusive respectively. Providing improper indexes will cause the function to throw an error (both must be 0 <= i <= length(s))

```
string a = "catdog"

string sub, error e = a.substring(1,4)   // "atd", null
string sub, error e = a.substring(3,99)  // null, error
string sub, error e = a.substring(50,99) // null, error
```

Get (c = string[i])

Strings may be indexed using brackets. Inside the brackets must be a zero-indexed integer. Getting is unsafe, and returns `string, error`.

Examples:

```
string a = "catdog"

printf("%s", a[3]) // prints d
```

Set (string[i] = s)

After indexing, an assignment may occur, to set a value of the list. Setting is `unsafe` due to the possiblity of index errors, and returns `string, error`.

Examples:

```
string a = "catdog"
a[2], error e = "p"
if (!e?) {
    printf("%s", a)
}
// prints capdog
```

Iterate (c in string)

Strings may be iterated over in for loops. Each element is returned in order.

Examples:

```
string a = "catdog"
for (string c in a) {
    printf("%s", c)
}
// prints catdog
```

Slice (string[i:j])

Strings may be sliced into substrings using slicing with brackets. Slicing is `unsafe`, and returns `string, error`. Note that unlike for `list.substring`, the second index of a slice may be larger than the length of the string.

```
string a = "catdog"

a[1:4]    // "atd"
a[3:99]   // "dog"
a[50:99]   // error
```

9.2 list

list.is_empty()

```
func is_empty() boolean
```

Returns whether the list on which it is called is empty.

```
list<int> a = []
list<int> b = [3,4]

a.is_empty()    // false
b.is_empty()    // true
```

list.append()

```
func append(T elem) list<T>
```

Appends the argument to the end of a list.
The list is returned which allows for chaining of `append` calls but the function has side effects and does not need to be used in an assignment.

```
list<int> a = []

a.append(7)     // [7]
a.append(3)     // [7,3]
```

list.pop()

```
unsafe func pop() T
```

Removes the last element in a list, and returns it. If the list is empty, an error is returned.

```
list<int> a = [3,4]

a.pop()   // 4
a.pop()   // 3
a.pop()   // error
```

list.push()

```
func push(T elem) list<T>
```

list.concat()

```
func concat(list<T> l) list<T>
```

list.reverse()

```
func reverse() list<T>
```

Reverses the list on which it is called and returns the reversed list.

```
list<int> a = [1,2,3,4,5]

a.reverse()  // [5,4,3,2,1]
```

list.copy()

Copies by the list value.

```
func copy() list<T>
```

Returns a copy by value of the list on which it is called

```
list<int> a = [1,2,3,4,5]   // [1,2,3,4,5]
```

Get (list[i])

Lists may be indexed using brackets. Inside the brackets must be a zero-indexed integer. Getting is unsafe, and for a `list<T>` returns `T, error`

Examples:

```
list<int> a, error E = [1,2,3,4]
printf("%d", a[2])
```

Set (list[i] = j)

After indexing, an assignment may occur, to set a value of the list. Setting is `unsafe`, and for a `list<T>` returns `T, error`.

Examples:

```
list<int> a = [1,2,3,4]
a[2], error e = 5
if (!e?) {
    for (int i in a) {
        printf("%d", i)
    }
}
// prints 1254
```

Iterate (j in list)

Lists may be iterated over in for loops. Each element is returned in order.

Examples:

```
list<int> a = [1,2,3,4]
for (int i in a) {
    printf("%d", i)
}
// prints 1234
```

Slice (list[i:j])

Lists may be sliced into sub-lists using slicing with brackets. Slicing is `unsafe`, and for a `list<T>` returns `list<T>, error`.

```
list<int> a = [1,2,3,4]
list<int> b, error e = a[2:4]
if (!e?) {
    for (int i in a) {
        printf("%d", i)
    }
}
// prints 234
```

9.3 dict

dict.is_empty()

```
func is_empty() boolean
```

Returns a boolean value of whether the dictionary on which it is called is empty.

```
dict<string, string> d = {"Dog" : "cat"}
dict<string, string> e = {}

d.is_empty()    // false
e.is_empty()    // true
```

dict.has_key()

```
func has_key(T key) boolean
```

Returns a boolean value corresponding to whether the dictionary on which it is called contains argument as a key.

```
dict<string, string> d = {"Dog" : "cat"}

d.has_key("Dog")  // true
d.has_key("Cow")  // false
```

dict.insert()

```
func insert(T key, S value)
```

Inserts the arguments as a key, value pair in the dictionary on which it is called.

```
dict<string, string> d = {"Dog" : "cat"}

d.insert("Cow" : "Pig")  // {"Dog" : "cat", "Cow" : "Pig"}
```

dict.remove()

```
unsafe func remove(T key)
```

Removes the value for the key given in the argument from the dictionary on which the function is called.

```
dict<string, string> d = {"Dog" : "cat", "Cow" : "Pig"}

d.remove("Dog")    // {"Cow" : "Pig"}
```

dict.keys()

```
func keys() list<T>
```

Returns a list of all keys in the dictionary on which it is called. The type of the returned list is that of the type of the keys in the dictionary.

```
dict<string, string> d = {"Dog" : "cat", "Cow" : "Pig"}

d.keys()    // ["Dog", "Cow"]
```

dict.values()

```
func values() list<S>
```

Returns a list of all values for the keyset in the dictonary on which it is called. The type of the returned list is that of the type of the values in the dictionary.

```
dict<string, string> d = {"Dog" : "cat", "Cow" : "Pig"}

d.values()      // ["Cat", "Pig"]
```

Get (dict[k])

Lists may be indexed using brackets. Inside the brackets must be a key in the dictionary. Getting is `unsafe`, and for a `dict<S,T>` returns `T, error`.

Examples:

```
dict<string, int> d = {"a":1, "b":2}
Int v, error e = d["a"]
```

```
if (!e?) {
    printf("%d", v)
}
// prints 1
```

Set (dict[k] = v)

After indexing, an assignment may occur, to set a value of the list. Setting is `unsafe`, and for a `dict<S,T>` returns `T , error`..

Examples:

```
dict<string, int> d = {"a":1, "b":2}
d["a"], error e = 5
if (!e?) {
    printf("%d", d["a"])
}
// prints 5
```

Iterate (j in dict)

Lists may be iterated over in for loops. Each element is returned in order.

Examples:

```
dict<string, int> d = {"a":1, "b":2}
for (int k, v in d) {
    printf("%s:%s, ", k, v)
}
// prints a:1, b:2,
```

9.4 error

error.message

```
string message
```

Returns the value of the error message on for the error object on which it is called.

```
error e = error(message="There was an error with that Request.",
                code=400,
                name="RequestError")

e.message       // "There was an error with that Request."
```

error.code

```
int code
```

Returns the value of the error message on for the error object on which it is called.

```
error e = error(message="There was an error with that Request.",
                code=400,
                name="RequestError")

e.code      // 400
```

error.name

```
string name
```

Returns the value of the error message on for the error object on which it is called.

```
error e = error(message="There was an error with that Request.",
                code=400,
                name="RequestError")

e.name      // "RequestError"
```

10. Program Execution

RAPID programs compile to a Go executable which is a platform specific binary.
Statements will be executed in order of their declaration.
If a RAPID program contains an `http` route, then running the executable will start a server on `localhost:5000` after all statements are executed.

10.1 Flags

There are several flags to customize the runtime of the app.

- `-D <pg_url>` : a string of ":@/". This is used to connect to the postgres server
- `-L <filename>` : log output will be appended to the specified file, defaults to `server.log`
- `-P <port>` : alters the port the service will run on, defaults to 80
- `-H` : prints all the options available
- `-V` : verbosely logs every HTTP request and the return values.