WebLang Language Reference Manual

Ryan Bernstein (rb3234) — Brendan Burke (btb2121) — Christophe Rimann (cjr2185)
Julian Serra (jjs2269) — Jordan Vega (jmv2177)

October 17, 2017

Contents

1 Introduction

Programming is becoming increasingly tied to the web. Most recent, relevant, and useful applications that we use in our everyday lives have some sort of web interaction through HTTP protocol. Much of this interaction is being done through representational state transfer and RESTful APIs. WebLang aims to provide programmers with a simple tool that consolidates interaction and facilitates consumption of RESTful applications. Our language is designed to make structures for specific applications, with callable endpoints, eliminating the hassle of authentication and identification. Moreover, it should ease the combination of information gathered from multiple APIs, allowing for exciting possibilities for programs using interaction with multiple application programming interfaces. This language is designed specifically to handle conventional return types from these interfaces (primarily JSON), and allows developers to process data and program with it efficiently.

WebLang utilizes C libraries to interact with servers using HTTP protocol, targeting the LLVM compiler. It is the goal of the language to allow programmers to easily and efficiently interact with RESTful applications. The following language reference manual provides detailed explanations on WebLangs syntax and functionality.

2 Lexical Conventions

2.1 Identifiers

Identifiers are used to name functions, types and variables. These use ASCII letters [A-Z, a-z], the underscore character, and decimals, but they must start with an ASCII letter. Identifiers are case sensitive, which means that an identifier such as random_api is treated as a different identifier from Random_api. Moreover, identifiers must not be equivalent to any of our reserved keywords (listed in the following section) as, naturally, the use of these keywords would result in errors.

2.2 Reserved Keywords

WebLang has a set of specific identifiers and functions which cannot be used for by the programmer for any other purpose (such as functions or variable names). These keywords are enumerated below, and will be explored in further depth throughout the manual.

1. `helper`: A function type
2. `type`: A function type
3. `if`: control flow
4. `else`: control flow

5. `foreach`: control flow
6. `true`: boolean
7. `false`: boolean
8. `import`: utilized to import other WebLang definitions

In addition, each of the names of the primitive types in section 3.1 are reserved.

2.3 Comments

In WebLang, one can make single-line comments, as well as multi-line comments. In a single line everything after // is a single line comment, as in the following two examples:

```
// This is a single line comment
post_joke joke_dest : String -> Nothing //this is also a comment
```

To make multiple line comments, enclose content between /* and */:

```
/* This is a comment
that spans several lines
because it looks better
like this sometimes*/
```

2.4 Literals

WebLang Literals are type defined values that are interpreted exactly as they are defined. There are literals for each primitive type in WebLang:

- JSON Literals
- Auth Literals
- Nothing Literals

See detailed descriptions of these primitives (section 3.1) outlining the composition of types.

2.5 Operators

WebLang uses the following operators that are reserved elements in the language. For more information on the function of each operator, see the operators section below.

$$= \texttt{->} - +$$
$$* \; / \; \% > \geq$$
$$< \leq \texttt{==} \; \texttt{!=}$$

2.6 Separators and Punctuation

Separators define scope and relations between variables, as well as start and end points for function declarations.

- {} - curly braces organize scope and start/end of function declarations. They are also utilized to define JSON objects (see Types section for more information)
- Whitespace - whitespace separates variable and function declarations. We consider the ASCII SP, ASCII FF, and the ASCII HT characters to be whitespace.
- New line - the new line character separates two statements, as in python. We consider ASCII LF and ASCII CR to be new line characters. An important caveat: due to the fact that JSON can often be very long and unwieldy because of large data transfers, it does not make sense to obey new line syntax with JSON. As such, the compiler will ignore all new lines and whitespace within JSON types, utilizing the separators built in to the types (for example braces/colons/commas for objects, and brackets/commas for arrays). This way, users can structure JSON in ways that make clear and intuitive sense to them.

3 Types

Types are primitive types paired with a predicate on that primitive type. A type with name A could be written either as A, or as A[p(val)], where p is a predicate on a value with primitive type the same as A's primitive type, and val is a locally bound name representing a constituent of A. A[p(val)] is a type with the same primitive type as A, but with an additional predicate p.

3.1 Primitives

The primitive type hierarchy is as follows:

Any: the most general, least descriptive type of WebLang. Every other type is an example of Any.

> `JSON`: The majority of our primitives implement JSON, which is based on the JavaScript ECMA 262 specification.
>
> > `String`: Weblang utilizes ASCII strings to represent textual data. Like Javascript, a single character is treated as a single character String and we do not support a type for chars.
> >
> > `Number`: A primitive corresponding to a doubleprecision 64bit binary format value

`Object`: A key-value pair container, with the key as a string and value as anything. Keys and values are separated with an equals sign, and key-value pairs are separated with commas.

`Array`: A traditional array structure.

`Bool`: A boolean representing true or false

`Null`: Absence of a value within a JSON object or array. Distinct from Nothing, which is absence of a value for overarching Any type.

`Auth`: An authentication type to pass to APIs that may require it. Currently, we only support OAuth2.

`OAuth2`: An authentication type in the form of a JSON object with predefined keys. All keys are optional; keys include: api_key, api_pub, api_secret.

`Nothing`: Absence of value within WebLang. Different from Null, which is absence of value within a JSON object or array

`Type`: A value representing a Type

Parent-children relationships in the hierarchy are is-a relationships, so a descendant can be used transparently as one of its ancestors. For example, a String can be used as a JSON value, and any value can be used as an Any.
All values in WebLang are of a primitive type, except for functions, which are of type A `->` B, where A and B are primitive types.

3.2 Using Types

Any type with name A can be used as A or as A[p(val)]. Additionally, some primitive container types have additional, more descriptive representations. The additional information from these representations will sometimes be checked at compile-time, and are always checked at run-time.

Array types also can be represented as:

```
[Type1, Type2, Type3]
[Type1, Type2, Type3, ...]
[Type1, Type2, Type3...]
```

The first instance represents an array with three elements, which are of Type1, Type2, and Type3 in that order.
The second instance represents an array that starts with three elements of those types, but can contain anything afterward.
The third instance represents an array that contains an element of Type1 and then of

Type2, and then zero or more elements of Type3.

Object types also can be represented as:

```
{key1 : Type1, key2 : Type2, key3 : Type3}
```

This represents a JSON object that contains at least pairs with keys key1, key2, and key3 with types Type1, Type2 and Type3 respectively.

3.3 Function Types

When functions are declared, type annotations of the form A `->` B are required. A function is guaranteed at compile-time to be called with a value that is the same primitive type as A, and to return the same primitive type as B. In addition, at runtime, the function's input is checked against A's predicate, and the function's output is checked against B's predicate. If either predicate fails, the program will exit with an appropriate error message.

3.4 Types as Values

Types can be assigned to variables as values, which is to say you can represent a type through the use of an identifier.

3.5 Type checking

The built-in function : can be used to check if a value matches both the primitive type and the predicate of a given type, with the form [value] : [type].

For example, `123 : Object` would evaluate to false, `123 : Number[val < 10]` would evaluate to false, and `123 : Number` would evaluate to true.

3.6 User Defined Types

Users can define types that are derived from primitive types. There are effectively aliases on other types. Users may use the following syntax:

```
type [name-of-type] [value-name] : [already-existing-type]
  [predicate on value-name]
```

as in:

```
type A a : B
  p(a)
```

This will create a type called A and will behave as B, but with the additional predicate p(a).
For example:

```
type Integer i : Number
  integral i
```

This will create a new type Integer that is represented as a number, but will additionally check that the number is integral.

4 Imports and Namespaces

`import` is a built-in function that takes an API specification and authorization information, and returns a namespace with the API's endpoints available.
import has the following type:

```
import : { address : String,
           port : Number[integral(val)],
           auth_required : Bool,
           auth_accepted : [String...],
           endpoints : Object[keys(val) : [Type...]],
         }[!val.auth_required || val.auth : Auth]
         -> Nothing
```

- `address` is the server address, for example "http://google.com/" or "127.0.0.1"
- `port` is a number representing the port
- `auth_required` indicates whether the user needs to specify authentication
- `auth_accepted` is a list of names of Auth subtypes that are accepted by the API. Strings that don't match supported Auth subtypes are ignored.
- `endpoints` is a JSON object whose keys are endpoint names, and whose values are Function Types that describe the type of the corresponding endpoint.

`import` brings the endpoints into the current namespace. If a name is bound to the result of import, as in `name = import` , then the endpoints will be addressed as `name.[endpoint]`. If there are conflicting names in the namespace, a warning is shown at compile time, and the existing names are overwritten.
The endpoints brought into the namespace by import behave as functions, with the corresponding function type from their API specification.
When an endpoint is used, a network call is made to the server at the port with the given authentication to that endpoint, with the endpoints arguments sent. Essentially, there are two ways to use API calls in WebLang: user-defined or built-in. Since API calls in of and themselves are a type, users may define custom API calls in their code. Alternatively, users can include one of the premade API calls. To do this, the user places an import statement

at the beginning of the program follow.
An example of importing the reddit API with OAuth2 authentication:

```
reddit_api = {
  address="reddit.com/api/json/",
  port=8080,
  auth_required = true,
  auth_accepted = [OAuth2],
  endpoints = {
    get_user = String -> Object { user_name : String,
      date_registered : String, user_id : Number },
    get_top_post = String -> Object { post_title : String,
      date_posted : String, upvotes : Number,
      comments : Array (Object { content : String }) },
    write_post = Object { subreddit : String,
      content : String, title : String } -> Nothing
  }
}

reddit = import (reddit_api + {
  auth = oAuth2 {
    api_key = "my api key",
    api_secret = "my api secret"
  }
})
```

5 Functions

Functions in WebLang take one argument and return at most one value. Operators, all of which are built-in, however, may take two inputs. The function header consists of a function name, a single argument, a colon, and the input and output types separated by an arrow. The function body consists of a variable number of declarations and function calls. An example function declaration foo that takes a String argument x and returns nothing would be written as follows:

```
foo x : String -> Nothing
        // statements here
```

WebLang functions can be called by stating the function name followed by its argument. An example function call for foo defined above would be written as follows:

```
foo "hello"
```

5.1 Endpoint Functions

Endpoint functions are the default function type in WebLang. Any function without the helper reserved word at the beginning of the declaration is an endpoint function. When a WebLang program is compiled, it generates a server binary and provides an endpoint for each endpoint function. For example, having an endpoint function foo and a port defined as 8000 will expose the /foo/ endpoint locally at `127.0.0.1:8000/foo`.

5.2 Helper Functions

Helper functions in WebLang are user-defined functions that do not result in a new endpoint upon compilation. These functions follow the same declaration syntax described above, but are initiated with the reserved word `helper`. For example, the following is the declaration for a helper function bar with String parameter x:

```
helper Bar x : String -> Nothing
        // statements here
```

5.3 Function calls

Functions are called by calling their identifier followed by an argument. As mentioned in the import section, functions may be called from other namespaces when imported using dot notation on the identifier we assigned their namespace.

```
                                foo "hello"
                  reddit.foo "hello" //if we had imported "reddit" as in the impor
```

5.4 Variable Assignment from Function

A variable may be assigned the return value of a function by separating the two with a single = sign. For example, the following sets the value of a variable exampleVar to the return value of the function foo with argument "hello":

```
                                exampleVar = foo "hello"
```

5.5 Arguments

Each endpoint or helper function takes one argument of any WebLang type. Each operator takes two arguments. All arguments passed to functions and operators are passed by value. Note that because generally we expect functions to require more than one argument, we expect this behavior to typically be passing in a JSON dictionary or array (as is almost always the case with RESTful requests).

5.6 Recursion

Recursive function calls are formatted identically to traditional function calls. Both Helper and Endpoint functions may be called recursively.

5.7 log

Log is a built in function that takes one argument as string. It prints whatever argument it is given and print it to stdout. If the argument is not a string (i.e. a Number, JSON object, etc.), it will attempt to cast it to string; if it is unable to do so, it will an error at compile time. As such, it is highly recommended that user defined types have a way to cast to string.

6 Control Statements

Weblang executes statements from top to bottom and left to right. But when using control statements, this breaks up the flow of the execution by integrating logical execution of code by using loops or branching with if/else statements.

6.0.1 Looping: foreach

The `foreach` statement allows the user to iterate over an array or a JSON object as in python. If iterating over a JSON object, the loop will loop over the outermost keys in the object.
Examples:

```
\\Statement writes to std out all the values in array
foreach [1,2,"hello"] val {
        log val
}

\\Statement writes to std out all the keys in array
obj = {"foo": "bar"}

foreach arrKeys key {
        log key //will just print key to stdout
    //In this case, will only print foo
}
```

Due to the fact that arrays and objects can contain multiple types, a common pattern in WebLang is to check the type utilizing the : built in function within each loop and execute based on that. For example:

```
foreach [1,2,"hello"] val {
        if(val : Number)
                log val+1 //only triggers if array contains Number
    else
        log val
}
```

Because foreach operates on arrays or objects only, for loops as expected in Java or C must be approximated. This can easily be done by creating an Array of Numbers and iterating over that.

6.0.2 if else

The `if` statement tells the program to execute a certain block of code when a test evaluates to true; there is an optional `else` clause that can follow an if should the if fail. Elses are greedy and will latch on to the nearest if. Brackets can be used to encompass an if statement, but if the body is only one line they are unnecessary.

```
\\Statement writes to std out all the values in array
if(true){
        foreach [1,2,"hello"] val {
        log val
        }
}

\\Statement writes bar to stdout
if(false)
        log "foo"
else
        log "bar"
```

7 Expressions

An expression is composed of one of the following:

- An operand followed by an operator followed by an operand
- Initializing an object
- Accessing a:
  - Object
  - JSON Object

    - Array
- An expression between ()
- Any of the subsections below

7.1 Arithmetic

An arithmetic expression consists of an operand followed by one or more operators. Operands can be variables, constants, and expressions.

```
"15" //Expression evaluates to String
2 + 2 //Expression evaluates to Number 4
10.1 + 1.0 //Expression evaluates to Number 11.1
{"foo":"bar"} + {"bar":"foo"}
//Expression evaluates to {"foo":"bar", "bar": "foo"}
```

7.2 Function Call

A call to a function that returns a value is considered an expression.

```
post_dad_joke "What time did the man go to the dentist? Tooth hurt-y."
//Evaluates to Nothing
```

7.3 Object, Array

Values of an object can be accessed via dot notation as in Python: accessing the "color" key of Object car is car.color.
Array values can be accessed utilizing bracket notation as in C, Python, or Java; accessing the 5th element of array a is a[5].
Examples of the above are shown here:

```
obj = { "array": [1,2,3] }
obj //Evaluates to Object {  array  :[1,2,3]}
Obj.array //Evaluates to JSON Array [1,2,3]
obj.array[0] //Evaluates to 1
```

7.4 Operators

An operator specifies a built in operation to be performed on operands. An operator can have one or two operands depending on what purpose it serves.

7.4.1 Assignment Operator

The assignment operator is used to store values into variables. As with most well used languages (Java, C, Python, etc.) Weblang uses “=” to store the value of the right side to the variable specified by the left side. The left side of the assignment operator may not be a literal or constant values.

```
obj = { "array": [1,2,3] }
```

7.4.2 Arithmetic Operators

The standard arithmetic operations addition, subtraction, multiplication, division, and modulo are included in WebLang.

1. Addition: Addition is performed on two values of type number. Examples are provided:

   ```
   5 + 5 // Evaluates to 10
   5.4 + 3.1 // Evaluates to 8.5
   ```

2. Subtraction: Subtraction is performed on two values of type number. Examples are provided:

   ```
   5 - 5 // Evaluates to 0
   4.2 - 1.3 // Evaluates to 2.9
   ```

3. Multiplication: Multiplication is performed on two values of type number. Examples are provided:

   ```
   5 * 5 // Evaluates to 25
   4.2 * 3.1 // Evaluates to 13.02
   ```

4. Division: Division is performed on two values of type number. Examples are provided:

   ```
   5/5 // Evaluates to 1
   6.4/2 // Evaluates to 3.1
   ```

5. Modulo: Modulo is performed on two values of type number, but the numbers must be whole integers. Examples are provided:

   ```
   6 % 2 // Evaluates to 0
   5 % 2 // Evaluates to 1
   ```

7.4.3 Conditional Operators

Conditional operators are used to determine how two operands relate to each other. As such, they will always take two values as inputs. The result of an operator is either `true` or `false`.

The conditional operators are:

Less than	$<$
Less than or equal to	$\leq$
Equality	==
Greater than	$>$
Greater than or equal to	$\geq$
Not Equal	!=

Examples are as follows:

```
a = 1
b = 2
d = {   arr   : [1,2,3]}
c = (a<b) //Evaluates to true
c = (<=x) //Evaluates to true
c = (w>x) //Evaluates to false
c = (w>=x) //Evaluates to true
c = d == Null //Evaluates to false
c = d != Null //Evaluates to true
```

Furthermore, conditional operators can be chained together using && and || operators. The && operator behaves like logical and, while the || operator behaves like logical or.

```
c = true && false //evaluates to false
c = true && true //evaluates to true
c = true || true //evaluates to true
c = false || false //evaluates to true
```

7.4.4 Object Set Operators

Objects operators are used to either union, intersect or differentiate two objects, resulting in a new one. The operators are `un`, `in`, and `diff`. These operators behave just like set operators, where the operations are done on the keys of the object.

1. Union Operator: If keys match, then the union is done on the values.

```
A = {"foo": "bar"}
B = {"foo": "one"}
A un B //Evaluates to Object {"foo" :[ "bar", "one" ]}

A = {"foo": "bar"}
B = {"foo": ["bar"]}
A un B //Evaluates to Object {"foo" :[ "bar" ]}, removes duplicates

A = {"foo": "bar", "more": "test"}
B = {"foo": 1}
A un B //Evaluates to Object {"foo" :[ "bar", 1 ]}

A = {"foo": "bar", "more": "test"}
B = {"foo": 1, "more": "in"}
A un B //Evaluates to Object {"foo" :[ "bar", 1 ], "test": ["test", "in"]}
```

2. Intersection Operator: If keys match, then the intersection is done on the values

```
A = {"foo": "bar"}
B = {"foo": "one"}
A in B //Evaluates to Object {"foo" :[ ]}

A = {"foo": "bar"}
B = {"foo": "bar"}
A in B //Evaluates to Object {"foo" :[ "bar" ]}

A = {"foo": "bar", "more": "one" }
B = {"foo": "two", "more": ["one"]}
A in B //Evaluates to Object {"foo" :[], "more": ["one"]}
```

3. Difference Operator: If keys match, then the difference is done on the values

```
A = {"foo": ["1", "2", "3"]}
B = {"foo": ["1", "2", "4", "5"]}
A diff B //Evaluates to Object {"foo" :["3"]}

A = {"foo": ["1", "2", "3"]}
B = {"foo": ["4", "5"]}
A diff B //Evaluates to Object {"foo" :["1", "2", "3"]}
```

It is possible to perform these operators on the keys rather than the values by first nesting the two objects to be compared within "wrapper" objects. This behavior may be included in the standard library, depending on whether or not it is deemed useful enough.

7.5 Operator Precedence

When multiple operators are used, the operations are grouped based on rules of precedence. Below is a list of precedence, if two or more operators have equal precedence, the operators are applied from left to right. Parentheses can be used to manually overwrite WebLang's precedence rules.

1. Method or helper calls, object or array access
2. Object set operations
3. Multiplication, division
4. Addition, subtraction
5. Expressions
6. Assignment expressions

8 Compiler Output

The program outputted upon successful compilation will always be a RESTful server, which is to say it will listen at a certain port and respond to HTTP requests at that port. As such, WebLang programs can often be referred to as "servers". Optionally, WebLang programs can be run as scripts if they include a main endpoint and the appropriate command line flag is invoked. The actual server that is created is written in C and linked. This C server accepts and routes requests to the correct endpoint.

8.1 Running a Server

A server is run by navigating to the directory containing a compiled server, and running ./out (the default server name). Other names can also be supplied during compile time with the -o flag.

8.2 Options when running a server

In addition to being able to specify command line arguments from within a server, all servers have the following two options built in:

- -c: set to false by default. When this flag is set, the server will run as script, meaning that the server will start up and automatically send itself a request to the /main/ endpoint. Prior to listening for requests, the server checks to see if the /main/ endpoint exists; if it does not it will throw a Runtime error. After successfully running the /main/ endpoint, the server will terminate. If this flag is not utilized, the main function will function the same way as any other endpoint (i.e. users can call to /main/ from the web browser etc.)

- -p [number argument]: set by default to 8080. The port that the server should listen at. The possible values for this are: 80, 443, and 1024-65535. Any other values will result in the server not starting.