WebAppQA Language Reference Manual

Adrian Frei (af2364)
Roy H. Han (rhh2109)
Peter Lu (yl2505)
Ankit Malhotra (am2994)

October 21, 2008

1 Introduction

WebAppQA is a language that simplifies the task of writing unit tests for web applications. It is a mixture between a regular scripting language like Perl or Python, unit-testing frameworks like jUnit and tools like BadBoy. With WebAppQA, users write tests to check page format, functionality and security. The interpreter runs each test and produces a report.

- Existence – Do all the links on a page work?
- Content – Does a page match a regular expression?
- Mechanics – Are links and cookies well formed?
- Security – Is the page safe against cross-site scripting attacks?

2 Usage Scenarios

- A web shop wants to automatically check a number of its product pages. Several salespeople constantly modify these pages and sometimes links are broken and images and offers are missing.
- A company develops software in small iterations in an agile manner. This means that the production system is updated frequently several times a day. Unfortunately this also means that the chance of errors on some pages is quite high.
- Wikis and other user-defined content management systems are volatile and prone to error. Tests written with WebAppQA can provide immediate feedback to the contributor before the page is released to the public.

3 Language Overview

The language is dynamically typed and runs in an interpreter. It supports loops, conditions, variables, methods with a syntax similar to Ruby. However there are no user-defined classes, iterators or blocks. WebAppQA code files end with the extension **wqa**.

4 Lexical conventions

4.1 Comments

A comment begins with a hash character # and ends with a newline EOL.

```
# This is a comment
```

4.2 Identifiers

An identifier is a sequence of letters, digits and underscore characters; the first character cannot be a digit. Uppercase letters are different from lowercase letters.

- Class identifiers must start with an uppercase character as in **Site** or **List**.
- Variables, methods and attributes must start with a lowercase character.
- Variables with the @ prefix are global variables (see Scope).

```
identifier12
identiFIER
_identifier
ClassIdentifier
@globalIdentifier
```

4.3 Keywords

The following keywords are reserved.

Conditionals

```
if
else
```

Methods

```
def
test
setup
return
```

Loops

```
for
in
while
```

Booleans

```
true
false
and
or
not
```

Miscellaneous

```
end
with
nil
```

Assertions

```
assert
fail
warn
```

4.4 Constants

4.4.1 Integer constants

An integer constant is a sequence of digits.

4.4.2 Strings

A string is a sequence of characters surrounded by double quotes. Precede special characters with a backslash.

\n represents the newline character EOL

\t represents the tab character

\" represents the double quote character

\\ represents the backslash character

5 Data types and structures

Constants are objects and variables are references to objects. An object is an instance of a class; every data type has a corresponding class.

WebAppQA supports two simple types.

Integers range from -2^{30} to $2^{30} - 1$.

Strings are sequences of characters from the ASCII set.

WebAppQA supports two compound types.

Lists of objects.

Dictionaries of objects. A dictionary is a mapping of objects to objects.

WebAppQA also supports library-defined classes precompiled with the interpreter. Each library-defined class has a **new()** constructor, methods and attributes. An example of a library-defined class is the **Site** class (see Standard Library). Class names begin with an uppercase letter.

```
anObject = AClass.new()  # Constructor
anObject.aMethod()       # Method
anObject.anAttribute = 1 # Attribute
```

6 Expressions

We list expressions in their order of precedence.

6.1 Primary expressions

Primary expressions group from left to right.

6.1.1 identifier

The identifier is a primary expression.

6.1.2 constant

An integer constant is a primary expression.

6.1.3 string

A string is a primary expression.

6.1.4 boolean

A boolean is a primary expression that is either **true** or **false**.

6.1.5 nil

The **nil** is a primary expression.

6.1.6 (expression)

An expression in parentheses is a primary expression.

6.1.7 primary-expression[expression]

List-lookup If the primary-expression on the left evaluates to a list, then the entire expression returns the object corresponding to the index specified between the brackets; the expression between the brackets must evaluate to an integer. The first index of a list is zero.

Dictionary-lookup If the primary-expression on the left evaluates to a dictionary, then the entire expression returns the object corresponding to the key specified between the brackets.

6.1.8 primary-expression(arguments$_{optional}$)

The primary-expression on the left must evaluate to a method and the arguments within parentheses may either be empty or be a list of expressions corresponding to the arguments of the method.

```
arguments:
    expression
    arguments, expression
```

6.1.9 primary-expression.identifier

The primary-expression on the left must evaluate to an object and the identifier must be an attribute of that object.

6.2 Definition expressions

6.2.1 List definition

Lists are defined using the following syntax.

```
list:
    []
    [ objects ]
objects:
    expression
    objects, expression
```

6.2.2 Dictionary definition

Dictionaries are defined using the following syntax.

```
dictionary:
    {}
    { mappings }
mappings:
    expression => expression
    mappings, expression => expression
```

6.3 Unary operators

6.3.1 - expression

The negation operator is only defined for integers and returns the negated integer.

6.3.2 not expression

The **not** operator is only defined for boolean expressions and returns **true** if the boolean expression evaluates to **false** and vice versa.

6.4 Multiplicative operators

6.4.1 expression * expression

Multiplication is defined for integers, strings and lists.

integer * integer If both expressions are integers, then the result is the product of the two integers.

string * integer, integer * string If one expression is an integer and the other expression is a string, then the result is multiplicative string concatenation. For example, $3 *$ "one" = "oneoneone".

list * integer, integer * list If one expression is an integer and the other expression is a list, then the result is multiplicative list concatenation. For example, $3 * [1] = [1, 1, 1]$.

Other permutations are undefined.

6.4.2 expression / expression

Division is only defined for integers and returns the result rounded toward zero. Division by zero ends the interpreter with a runtime error.

```
assert 10 / 3 == 3
assert -10 / 3 == -2
```

6.4.3 expression % expression

The modulo operator is only defined for integers and returns the integer remainder after division. Modulo by zero ends the interpreter with a runtime error.

```
assert 10 % 3 == 1
assert -10 % 3 == -1
```

6.5 Additive operators

Additive operators are defined for integers, strings and lists.

6.5.1 expression + expression

integer + integer If both expressions are integers, then the result is the sum of the two integers.

string + string If both expressions are strings, then the result is the concatenation of the two strings.

list + list If both expressions are lists, then the result is the concatenation of the two lists.

Other combinations are undefined. Note that additive operators are not defined for dictionaries.

6.5.2 expression - expression

integer - integer If both expressions are integers, then the result is the subtraction of the second integer from the first integer.

Other combinations are undefined.

6.6 Relational operators

Relational operators are only defined for integers.

6.6.1 expression < expression

The < operator returns **true** if the left operand is less than the right operand, and **false** otherwise.

6.6.2 expression > expression

The > operator returns **true** if the left operand is greater than the right operand, and **false** otherwise.

6.6.3 expression <= expression

The <= operator returns **true** if the left operand is less than or equal to the right operand, and **false** otherwise.

6.6.4 expression >= expression

The >= operator returns **true** if the left operand is greater than or equal to the right operand, and **false** otherwise.

6.7 Equality operators

Equality operators are defined for booleans, integers, strings, sites, lists, dictionaries and **nil**. Equality is not defined for methods.

6.7.1 expression == expression

The == operator returns **true** if the operands are equal and **false** if the operands are not equal. Equality is defined differently depending on the data type of both expressions.

- Integers, booleans, strings and **nil** are compared by value.
- Lists, dictionaries and other objects are compared by reference. The == operator returns **true** if both expression refer to the same object and **false** if they refer to different objects.

```
>> a = [1,2]
>> b = [1,2]
>> a == b
false
>> c = a
>> c == a
true
```

6.7.2 expression != expression

The **!=** operator returns **false** if the operands are equal and **true** if the operands are not equal. Equality is defined differently for different data types.

6.7.3 expression ~= regularExpression

The ~= operator returns **true** if the left expression matches the regularExpression and **false** if it does not. The interpreter evaluates regular expressions according to the OCaml **Str** module.

6.7.4 expression and expression

The **and** operator returns **true** if both of its operands are **true**, and **false** otherwise.

6.7.5 expression or expression

The **or** operator returns **true** if either of its operands is **true**, and **false** otherwise.

6.8 Assignment operators

6.8.1 expression = expression

The value of the expression on the right replaces the value of the expression on the left. The expression on the left must evaluate to either a variable, list-lookup, dictionary-lookup or class attribute.

7 Statements

7.1 Expression statement

Expression statements have the following form:

```
statement:
    expression EOL
```

Examples of expression statements are assignments or method calls.

7.2 Statement list

A group of statements is executed in sequence.

```
statementList:
    statement
    statement statementList
```

7.3 Conditional statement

Conditional statements execute different parts of code under different conditions.

```
conditional-statement:
    if expression EOL statementList end EOL
    if expression EOL statementList else EOL statementList end EOL
```

The interpreter executes the first statementList when the **if** expression evaluates to **true**; if the expression evaluates to **false**, then the interpreter executes the **else** statementList.

7.4 While statement

The **while** statement looks like the following.

```
while-statement:
    while expression EOL statementList end EOL
```

In a **while** statement, the interpreter first evaluates the expression; if the expression is **true**, the interpreter executes statementList. The interpreter repeats expression evaluation and statementList execution until the expression is **false**.

7.5 For statement

The **for** statement has the following form.

```
for-statement:
    for variable in range EOL statementList end EOL
range:
    expression .. expression
```

The **for** statement executes the statementList once for each integer in the range, starting from the smallest. At each iteration, the interpreter sets the variable to the next integer in the range. The two expressions in the range are evaluated once, before the **for** loop. Each expression in the range must evaluate to an integer.

7.6 Return statement

There are two **return** options.

```
returnStatement:
    return EOL
    return expression EOL
```

In both cases, the **return** statement lets the interpreter return from a method. When an expression is not provided, the **return** statement returns **nil**; when an expression is provided, it returns the result of the expression. If a method is missing a **return** statement, the interpreter returns the last statement evaluation.

7.7 Assert statement

The **assert** statement has one of four forms.

```
assertStatement:
    assert expression fail EOL
    assert expression warn EOL
    assert expression fail expression EOL
    assert expression warn expression EOL
```

In all cases, **assert** evaluates the first expression. If the expression is **true**, nothing happens. If the expression is **false** and is followed by the keyword **fail**, the interpreter stops immediately and reports failure. If the expression is **false** and is followed by **warn**, the interpreter continues and reports a warning. The optional second expression customizes the reported error or warning; it must evaluate to a string that is then inserted into the report.

7.8 With statement

The **with** statement looks like the following.

```
withStatement:
    with object EOL statementList end EOL
```

The **with** statement shadows local variables and methods with the attributes and methods of the object. For example,

```
object.method1()
object.attribute1 = 1
object.method2()
```

can be written as

```
with object
    method1()
    attribute1 = 1
    method2()
end
```

However, global variables are still accessible with the @ prefix.

8 Definitions

Every WebAppQA program is a collection of test and method definitions.

8.1 Method definitions

Methods are defined using the **def** keyword:

```
methodDefinition:
    def identifier (parameterList) EOL statementList end EOL

parameterList:
    identifier
    identifier , parameterList
```

8.2 Test definitions

Tests are the centerpiece of a WebAppQA program. The interpreter executes each user-defined test. There are two different kinds of tests.

```
test-definition:
    test identifier () EOL statementList end EOL

setup-definition:
    setup identifier () EOL statementList end EOL
```

The difference between the two is that the program guarantees that all **setup** methods are executed before any **test** method, regardless of their position in the source code.

9 Scope

9.1 Dynamic Declaration

Variables do not require declaration before use.

Only code defined in a **def**, **test** or **setup** method is executed. Statements that appear outside of a method are not allowed.

9.2 Global scope

Variables that begin with the prefix @ have global scope. As soon as a value is assigned to a global variable, it is available for the entire program.

9.3 Method scope

Variables without a prefix have local scope. A local variable that appears in a method is discarded at the end of the method.

9.4 Blocks

Blocks such as those defined by **for** and **while** do not introduce new scopes. Variables inside a block refer to the local variables of the surrounding method. Assignments to undefined variables within a block define new local variables for the surrounding method.

9.5 Overloading and namespaces

Method overloading is not allowed. If a variable and a method have the same name, then the name refers to the variable whenever the variable is in scope.

Global and local variables have different namespaces.

```
>> @g = 1
>> g = 2
>> @g
1
>> g
2
```

9.6 Argument to methods

If an argument is reassigned inside the method, then the original reference is not modified.

```
setup f()
    x = 1
    g(x)
    assert x == 1 fail    # (x == 1) == true
end

def g(x)
    x = 2
end
```

However, if the value of an argument is changed inside the method, then the original object is modified.

```
setup f()
    x = [1,2]
    g(x)
    assert x == [1,2,3] fail   # (x == [1,2,3]) == true
end

def g(x)
    x.append(3)
end
```

10 Standard Library

The Standard Library is a collection of classes that are precompiled with the interpreter. Each class has a **new()** constructor.

The **Nil** class is the class underlying the nil datatype. The **Nil** class supports the following attribute.

- **to_s** returns the string 'nil'

The **Integer** class is the class underlying the integer datatype. The **Integer** class supports the following attribute.

- **to_s** returns the integer as a string constant

The **Boolean** class is the class underlying the boolean datatype. The **Boolean** class supports the following attribute.

- **to_s** returns the string 'true' or 'false'

The **String** class is the class underlying the string datatype. The **String** class supports the following attribute.

- **to_s** returns the string as a string constant

The **List** class is the class underlying the list datatype. Besides the [ ] operator for accessing elements of a list, the **List** class supports the following attributes.

- **length** returns the length of the list
- **to_s** returns the list as a string constant
- **append(object)** appends object to the end of the list

The **Dictionary** class is the class underlying the dictionary datatype. Besides the [ ] operator for accessing the values of a dictionary, the **Dictionary** class supports the following methods.

- **keys** returns the keys of the dictionary as a list
- **to_s** returns the dictionary as a string constant

The **Site** class has methods to visit, navigate and test web sites. The following methods and attributes are defined:

- **to_s** returns the site as a string constant
- **visit(url, parameters, isPOST)** visits the specified url using the given form parameters given as a dictionary of strings; if isPOST is **true** then the interpreter uses a HTTP POST request; if isPOST is **false** then the interpreter uses a HTTP GET request
- **status** returns the status of the page (e.g. 404 or 200)
- **links** returns a list of the links on the page
- **images** returns a list of the images on the page
- **contentType** returns the content type of the page (e.g. text/html)
- **content** returns the content of the page as a string constant
- **cookies** returns a dictionary of the cookies set by the page

11 Examples

11.1 Inner product

The example below computes the inner product of two vectors given as lists.

```
def computeInnerProduct(vector1, vector2)
    sum = 0
    for i in 0 .. vector1.length - 1
        sum = sum + vector1[i] * vector2[i]
    end
    return sum
end
```

11.2 Application testing

The example below tests a small web shop.

- Test that the home page exists.
- Test the login and make sure cookies are set and the user is forwarded to the welcome page.
- Test that the website verifies cookies and allows access to logged in users.
- Test a number of offer pages and make sure that each page displays the merchant's logo and that all links are working.

```
# Define a setup method (called first)
setup initialize()
    # Global string
    @startURL = "www.myshop.ch"
    # Global hash table containing string
    @loginParam = {"username" => "hans", "password" => "p8ssw0rd"}
end
# Define a test method
test home()
    # Create a new site object and visit the site
    site = Site.new()
    site.visit(@startURL, @loginParam, true)
    # Make sure the page is there
    assert site.status == 200 fail
end
```

12 Appendix – Scanner

Our scanner distinguishes between separators and operators.

A separator is one of the following:

```
. , .. () [] {} =>
```

An operator is one of the following:

```
not and + - * / % = < <= > >= == !=
```

12.1 Code

Listing 1: scanner.mll

```
{ open Parser }

let digit   = ['0'-'9']
let id      = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9']*
let space   = [' ' '\t' '\r']
let comment = '#'[^ '\n']*
let str     = '\"'[^ '\"' '\n']*'\"'

rule qalang = parse
  | digit+  as x { INT (int_of_string x) }
  | str     as x { STR x }
  | id      as x { ID x }

  | "nil"        { NIL }

  | "def"        { DEF }
  | "setup"      { SETUP }
  | "test"       { TEST }
  | "return"     { RETURN }

  | "assert"     { ASSERT }
  | "fail"       { FAIL }
  | "warn"       { WARN }

  | "if"         { IF }
  | "else"       { ELSE }

  | "for"        { FOR }
  | "in"         { IN }
  | "while"      { WHILE }
  | ".."         { RANGE }

  | "with"       { WITH }
  | "end"        { END }
```

```
| "true"          { TRUE }
| "false"         { FALSE }
| "and"           { AND }
| "or"            { OR }
| "not"           { NOT }

| "+"             { PLUS }
| "-"             { MINUS }
| "*"             { MUL }
| "/"             { DIV }
| "%"             { MOD }

| "=>"            { ARROW }
| "="             { ASN }

| "=="            { EQ }
| "~="            { REGEQ }
| ">"             { GT }
| "<"             { LT }
| "!="            { NEQ }
| ">="            { GEQ }
| "<="            { LEQ }

| "("             { LPAREN }
| ")"             { RPAREN }
| "["             { LSQBRA }
| "]"             { RSQBRA }
| "{"             { LBRACE }
| "}"             { RBRACE }

| "."             { DOT }
| ","             { COMMA }

| "@"             { GLOBAL }

| comment         { qalang lexbuf }
| space           { qalang lexbuf }
| '\n'            { EOL }
| eof             { EOF }
```

13 Appendix – Parser

Listing 2: parser.mly

```
%{ open Ast %}

%token ID GLOBAL
%token EQ REGEQ NEQ GEQ LEQ GT LT
%token ASN COMMA DOT RANGE ARROW
%token LPAREN RPAREN LSQBRA RSQBRA LBRACE RBRACE
%token PLUS MINUS MUL DIV MOD
%token EOL EOF
%token IF ELSE
%token DEF TEST SETUP RETURN
%token FOR IN WHILE
%token AND OR NOT TRUE FALSE
%token END WITH NIL
%token ASSERT WARN FAIL

%token <string> STR
%token <string> ID
%token <string> CLASSID
%token <int> INT

%left COMMA
%right ASN
%left AND OR
%left EQ REGEQ NEQ GT LT LEQ GEQ
%left PLUS MINUS
%left MUL DIV MOD
%left NOT
%left NEG

%start program
%type <Ast.prog> program

%%

program:
  {} /* Empty */
| testMethod program {}
| setupMethod program {}
| defMethod program {}
| EOL {}


testMethod:
  TEST ID LPAREN RPAREN EOL statementList END EOL {}

setupMethod:
  SETUP ID LPAREN RPAREN EOL statementList END EOL {}
```

```
defMethod:
  DEF ID LPAREN RPAREN EOL statementList END EOL {}

statementList:
  statement {}
| statement statementList {}

statement:
  EOL {}
| expression EOL {}
| conditionalStatement {}
| whileStatement {}
| forStatement {}
| assertStatement {}
| withStatement {}
| returnStatement {}

conditionalStatement:
  IF expression EOL statementList END EOL {}
| IF expression EOL statementList ELSE EOL statementList END EOL {}

whileStatement:
  WHILE expression EOL statementList END EOL {}

forStatement:
  FOR ID IN range EOL statementList END EOL {}

returnStatement:
  RETURN EOL {}
| RETURN expression EOL {}

assertStatement:
  ASSERT expression FAIL EOL {}
| ASSERT expression WARN EOL {}
| ASSERT expression FAIL expression EOL {}
| ASSERT expression WARN expression EOL {}

withStatement:
  WITH ID EOL statementList END EOL {}

range:
  expression RANGE expression {}

primaryExpression:
  ID {}
| GLOBAL ID {}
| STR {}
| INT {}
| TRUE {}
| FALSE {}
| NIL {}
```

```
| LPAREN expression RPAREN {}
| dictionaryDefinition {}
| listDefinition {}

postfixExpression:
  primaryExpression {}
| postfixExpression LPAREN argumentList RPAREN {}
| postfixExpression LPAREN RPAREN {}
| postfixExpression LSQBRA expression RSQBRA {}
| postfixExpression DOT ID {}

argumentList:
  expression {}
| argumentList COMMA expression {}

listDefinition:
  LSQBRA RSQBRA {}
| LSQBRA argumentList RSQBRA {}

dictionaryDefinition:
  LBRACE RBRACE {}
| LBRACE dictionaryDefinitionList RBRACE {}

dictionaryDefinitionList:
  expression ARROW expression {}
| dictionaryDefinitionList COMMA expression ARROW expression {}

expression:
  postfixExpression {}
| expression PLUS expression {}
| expression MINUS expression {}
| expression MUL expression {}
| expression DIV expression {}
| expression MOD expression {}
| expression EQ expression {}
| expression REGEQ expression {}
| expression NEQ expression {}
| expression GEQ expression {}
| expression LEQ expression {}
| expression GT expression {}
| expression LT expression {}
| expression AND expression {}
| expression OR expression {}
| NOT expression {}
| MINUS expression %prec NEG {}
```