Michael Langford
Khai Vuong
Javier Coca
George Sirois

1. AWE Language Reference Manual

1.1 Introduction

The intent of Autonomous Web Explorer (AWE) is to allow users to create simple scripts in order to automate the browsing of web pages. The language is syntactically small and does not contain an abundance of library functions – only enough to get the very basics done. In this respect, the syntax is very much modeled after C. This allows the user to pick up the syntax very quickly yet still write powerful scripts that gather the information they need.

1.2 Structure

The structure of an AWE script is taken directly from C. The main loop of the program is denoted by:

```
void main()
{
	//CODE
}
```

Further, the programmer may declare functions in the C style with a return type, arguments inside of the parenthesis, and return types.

```
int sum(int i, int k)
{
	return i + k;
}
```

1.3 Lexical Conventions

1.3.1 Comments

Comments are in the traditional C style where a double slash '//' at the beginning of a line denotes the entire line as a comment.

1.3.2 Identifiers

A sequence of letters and numbers is considered an identifier. AWE retains C's case sensitivity. An identifier may not start with a number and may contain the underscore "_" character.

1.3.3 Keywords

There are a number of identifiers which are reserved as keywords and may not be used as user-defined identifiers:

```
int         else        return
string      in          break
hyperlink   while       if
webfile     foreach     collection
```

1.3.4 Constants

1.3.4.1 String Constants

A string constant is a sequence of characters surrounded by a double quote. There is no char type so any constant must be surrounded by double quotes. The forward slash "\" is used to escape special characters such as newlines, carriage returns, etc. A double slash "\\" inserts a single slash into the string constant. A newline is given by "\n" and a carriage return is given by "\r".

1.3.4.2 Integer Constants

An integer constant is a sequence of digits (0-9).

1.3.5 Separators

The curly brackets ({ }) denote the beginnings and endings of functions and allow the programmer to group code blocks in the same scope.

The semicolon ";" denotes the end of a statement. Multiple statements may be placed on a single line.

White space is ignored for anything other than the purpose of separating syntactic elements.

1.3.6 Operators

AWE contains a somewhat large set of operators. They are simply listed here and will be discussed in more detail in the operators section.

```
+      -      *      /      %      =
==     !=     <      <=     >      >=
&&     ||     ->     &      ++     --
~~     !~     ^
```

1.4 Types

AWE has two different types – value types and standard types.

1.4.1 Value types

Value types are integers and strings. They are essentially an area of allocated memory which stores a primitive type. Certain operators operate only on value types. Value types are:

```
int
```

```
string
```

1.4.1.1 int

An integer is 32 bits allocated in memory to store integer values.

1.4.1.2 string

A string is dynamically allocated and stores a sequence of Unicode characters.

1.4.2 Standard types

Standard types are the more complicated types built into the language. They may contain several information blocks and must be manipulated using the built in functions. Certain assignment operators may only be used on standard types. Standard types are:

```
hyperlink
webfile
collection
```

1.4.2.1 hyperlink

Contains the url of the link and the html of the page.

1.4.2.2 webfile

Stores a resource pointed to by a link. Contains MIME type information.

1.4.2.3 collection

Stores a collection of hyperlinks. Individual hyperlinks be accessed using the bracket operator as discussed in section 1.5.

1.5 Operator Definitions

Precedence and Associativity of Operators

Operators	Associativity
`() []`	Left to right
`++ -- * & -`	Right to left
`* / %`	Left to right
`+ - ^`	Left to right
`< <= > >=`	Left to right
`== != ~~ !~`	Left to right
`&&`	Left to right
`||`	Left to right
`=`	Right to left
`->`	Left to right
`,`	Left to right

1.5.1 Grouping and Indexing Operators

Grouping operators are used to modify the order of evaluation by specifying that everything within the operators is to be evaluated with the highest precedence.

1.5.1.1 (*expressions*)

Groups the contained expressions into a group of highest precedence

Indexing operators are used to traverse collections

1.5.1.2 *collection*[*int* OR *string*]

Returns the value in the collection at index *int* or the first link value matching *string*. Only hyperlinks may be placed in a collection so it is guaranteed that there will be a link to match for each item.

1.5.2 Unary Operators

Unary operators may only be used on integer types.

1.5.2.1 *int*++

Increments an integer value by 1.

1.5.2.2 *int*—

Decrements an integer value by 1.

1.5.2.3 –*int*

Invert the sign of an integer.

1.5.3 Multiplicative Operators

Multiplicative operators may only be used on integer types and result in integers.

1.5.3.1 (*integer expression*) * (*integer expression*)

Multiplies the two integer expressions.

1.5.3.2 (*integer expression*) / (*integer expression*)

Divides the two integer expressions. Returns only the quotient.

1.5.3.3 (*integer expression*) % (*integer expression*)

Takes the modulus of two integers. Returns the remainder.

1.5.4 Additive Operators

Additive operators may only be used on integer types and result in integers.

1.5.4.1 (*integer expression*) + (*integer expression*)

Adds the two integer expressions.

1.5.4.1 (*integer expression*) - (*integer expression*)

Subtracts the two integer expressions.

1.5.5 Assignment Operators

Assignment operators may be performed on all types and simply copy the value of one expression to another.

1.5.5.1 *variable = expression*

Sets the *variable* to the value of the expression provided they are of the same type.

1.5.5.2 *webfile = *hyperlink*

Similar to C's pointer notation, this points a webpage to a hyperlink.

1.5.5.3 *hyperlink = &webfile*

Similar to C's pointer notation, this points a hyperlink to a webpage.

1.5.5.4 *webfile -> string*

The assignment operator makes it easy to pipe the data contained in a *webfile* to a path on disk as specified by *string.*

As an example of the above 3 operators,
*{http://www.yahoo.com/index.htm, "} -> "C:\\downloaded"; would save the index file to
C:\downloaded\http___www_yahoo_com_index_htm
where all non alphanumeric characters are replaced by underscores.

1.5.6 Comparison Operators

Most comparison operators may only be used on integer types. There are a limited number which may be used on string types.

1.5.6.1 (*integer expression*) == (*integer expression*)

Evaluates to 0 if both integers are equal. Evaluates to 1 otherwise since there is no Boolean type.

1.5.6.2 (*integer expression*) != (*integer expression*)

Evaluates to 1 if both integers are equal. Evaluates to 0 otherwise.

1.5.6.3 (*integer expression a*) < (*integer expression b*)

Evaluates to 1 if a is less than b. Evaluates to 0 otherwise.

1.5.6.4 (*integer expression a*) <= (*integer expression b*)

Evaluates to 1 if a is less than or equal to b. Evaluates to 0 otherwise.

1.5.6.5 (*integer expression a*) > (*integer expression b*)

Evaluates to 1 if a is greater than b. Evaluates to 0 otherwise.

1.5.6.6 (*integer expression a*) >= (*integer expression b*)

Evaluates to 1 if a is greater than or equal to b. Evaluates to 0 otherwise.

1.5.6.7 (*integer expression*) && (*integer expression*)

Performs the logical AND of two integer expressions. With no Boolean type, 0 is false, and any other value is considered true. Result is given as 0 or 1.

1.5.6.8 (*integer expression*) || (*integer expression*)

Performs the logical OR of two integer expressions. With no Boolean type, 0 is false, and any other value is considered true. Result is given as 0 or 1.

1.5.6.9 (*string expression a*) ~~ (*string expression b*)

Returns 1 if expression a matches the REGEX given in expression b, or 0 otherwise.

1.5.6.10 (*string expression a*) !~ (*string expression b*)

Returns 0 if expression a matches the REGEX given in expression b, or 1 otherwise.

1.5.6.11 (*string expression*) ^ (*string expression*)

Returns the concatenation of the two strings.

1.6 Flow Control

1.6.1 The *while* statement

while (condition is true)

```
{
        //Execute this
}
```

The while loop evaluates the logical syntax of the condition in the parenthesis. If it evaluates to true, the code in the braces begins execution. When the end brace is reached and the condition is reevaluated. If there is only a single line of code to be executed, it may directly follow the while statement with no braces.

1.6.2 The *for* statement

for*(*variable = *int*; variable comparison *int2;* variable ++ or variable --

```
{
        //Execute
}
```

The for statement allows the programmer to assign an integer value to variable. Then, as long as "variable comparison *int2"* returns true, the loop executes. At the end of the loop, the variable is incremented or decremented and the evaluation is made again. The loop continues until the middle condition is no longer satisfied.

1.6.3 The *foreach* statement

foreach(*type* name in *collection*)

```
{
        //Do something with name
}
```

The foreach statement allows the programmer to easily traverse a collection. For each item in the collection of type *type*, it is assigned to the local variable "name" and the loop is run.

1.6.4 The *if else* construct

if (condition 1)

```
{
        //statement 1
}
```

else if (condition 2)

```
{
        //statement 2
}
```

else

```
{
        //statement 3
}
```

The *if else* construct first evaluates condition 1 for truth. If it is true, statement 1 is executed. If not, it checks condition 2. If condition 2 is true,

statement 2 is executed. Finally, if neither condition is met, statement 3 is executed. Note that the *else if* and *else* are optional, and the programmer may chain together as many *else if*s as he or she wishes.

1.6.5 The *return [argument]* statement

Calling *return* in a function immediately breaks execution and returns the value of *argument* to the point where the function was called. For void functions, return may be called with no argument.

1.6.6 The *break* statement

Calling *break* automatically jumps execution out of the lowest-level loop that the program is executing.

1.7 Declaration Syntax

1.7.1 Value type variables

Value type variables are declared as *type identifier*; or *type identifier = value*; Where value can be an expression which returns the correct value type. *Value* may also be a constant (*string* or *int*).

1.7.2 Standard type variables

Standard type variables are declared as *type identifier*; or *type identifier = value*; If they are assigned to a value, its type must match the identifier type.

While there are no constants for standard type variables, they can be created on the fly.

For example:

hyperlink link1 = *{“http://www.yahoo.com/index.htm”,””};

This is a link to Yahoo’s index file. The * denotes the creation of a link from a webpage, the first string is the link to the page, and the second string is the optional title the programmer may assign.

collection collection1 = {};

This is an empty collection. Collections may only contain hyperlinks. To add links, place them between the braces separated by commas.

1.8 Standard Library

The following functions are built in to the language:

1.8.1 *string* url (*hyperlink*)

Returns the url of the given hyperlink as a *string.*

1.8.2 *string* text (*hyperlink*)

Returns the text of the given hyperlink as a *string.*

1.8.3 *int* count (*collection*)

Returns the length of the given *collection* as an *int.*

1.8.4 *void* add(*collection*, *hyperlink*)

Adds a *hyperlink* to a *collection*.

1.8.5 *void* remove (*collection*, *hyperlink*)

Removess a *hyperlink* from a *collection*.

1.8.6 *void* clear(*collection)*

Removes all hyperlinks from a collection.

1.8.7 *int* status(*webfile)*

Returns the status of the http request of the given file (200, 404, etc.) as an *int.*

1.8.8 *int* isSuccess (*webfile)*

Returns *int* 1 if the status of the given file is 200. Returns *int* 0 otherwise.

1.8.9 *string* type (*webfile*)

Returns the MIME type of the webfile as a *string.*

1.8.10 *int* isHtml (*webfile)*

Returns *int* 1 if the type of the given file is text/html. Returns *int* 0 otherwise.

1.8.11 *int* isImage (*webfile)*

Returns *int* 1 if the type of the given file is image. Returns *int* 0 otherwise.

1.8.12 *string* html (*webfile*)

Returns the html source code of the given webfile as a *string.*

1.8.13 *string* title (*webfile*)

Returns the title of the given webfile as a *string.*

1.8.14 *string* text (*webfile*)

Returns only the displayed text of the given webfile as a *string.*

1.8.15 *collection* links (*webfile*)

Returns a collection of all the links on the given webfile as a *collection.*

1.8.16 *collection* images (*webfile*)

Returns a collection of all the links which lead to images on the given webfile as a *collection.*

1.8.17 *void* print (*string*)

Prints the given *string* to the console.

1.8.18 *void* save (*string, string* filename)

Saves the given *string* to path filename.

1.8.19 *int* indexOf (*string* string1, *string* string2, *int* start, *int* length)

Returns the index of string2 in string1 in the given bounds as an *int*.

1.8.20 *int* length (*string*)

Returns the length of the given string as an *int*.

1.8.21 *string* replace (*string* string1, *string* string2, *string* string3)

Returns a copy of string1 in which all instances of string2 have been replaced by string3.

1.8.22 *string* delete (*string* string1, *int* start, *int* length)

Returns a copy of the given string without the given bounds.

1.8.23 *string* subString (*string* string1, *int* start, *int* length)

Returns a substring from the given string within the given bounds.

1.9 Regular Expressions

1.9.1 Regular Expression Syntax

As seen in section 1.5.6, there is a need for a regular expression syntax. The syntax for a full regular expression parser would probably be longer than this document, so we have decided to use the basics of the Javascript RegEx syntax found here:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/js56jsgrpRegExpSyntax.asp
We will not be using backreferences or pattern capture/lookahead, but the basics are the same.