Contents

1 . SCOPE 4

2 . DOCUMENT STATUS 5

2.1 COPYRIGHT NOTICE 5
2.2 ERRATA 5
2.3 COMMENTS 5

3 . REFERENCES 6

3.1 NORMATIVE REFERENCES 6
3.2 INFORMATIVE REFERENCES 6

4 . DEFINITIONS AND ABBREVIATIONS 7

4.1 DEFINITIONS 7
4.2 ABBREVIATIONS 8

5 . NOTATIONAL CONVENTIONS 9

6 . WMLSCRIPT COMPLIANCE 10

6.1 SUPPORTED DATA TYPE 10
6.2 DATA TYPE CONVERSIONS 10
6.3 ERROR HANDLING 10
6.4 SUPPORT FOR INTEGER-ONLY DEVICES 10

7 . LANG 11

7.1 ABS 11
7.2 MIN 11
7.3 MAX 12
7.4 PARSEINT 12
7.5 PARSEFLOAT 12
7.6 ISINT 13
7.7 ISFLOAT 13
7.8 MAXINT 14
7.9 MININT 14
7.10 FLOAT 14
7.11 EXIT 14
7.12 ABORT 15
7.13 RANDOM 15
7.14 SEED 16

8 . FLOAT 17

8.1 INT 17
8.2 FLOOR 17
8.3 CEIL 17
8.4 POW 18
8.5 ROUND 18
8.6 SQRT 18
8.7 MAXFLOAT 19
8.8 MINFLOAT 19

9 . STRING 20

9.1 LENGTH 20

9.2 ISEMPTY 20
9.3 CHARAT 21
9.4 SUBSTRING 21
9.5 FIND 21
9.6 REPLACE 22
9.7 ELEMENTS 22
9.8 ELEMENTAT 23
9.9 REMOVEAT 23
9.10 REPLACEAT 24
9.11 INSERTAT 24
9.12 SQUEEZE 25
9.13 TRIM 25
9.14 COMPARE 25
9.15 TOSTRING 26
9.16 FORMAT 27

10 . URL 29

10.1 ISVALID 29
10.2 GETSCHEME 29
10.3 GETHOST 30
10.4 GETPORT 30
10.5 GETPATH 30
10.6 GETPARAMETERS 31
10.7 GETQUERY 31
10.8 GETFRAGMENT 31
10.9 GETBASE 32
10.10 GETREFERER 32
10.11 RESOLVE 32
10.12 ESCAPE 33
10.13 UNESCAPE 33
10.14 ESCAPESTRING 33
10.15 UNESCAPESTRING 34
10.16 LOADSTRING 34

11 . WMLBROWSER 36

11.1 GETVAR 36
11.2 SETVAR 36
11.3 GO 36
11.4 PREV 37
11.5 NEWCONTEXT 37
11.6 GETCURRENTCARD 38
11.7 REFRESH 38

12 . DIALOGS 39

12.1 PROMPT 39
12.2 CONFIRM 39
12.3 ALERT 39

APPENDIX A. LIBRARY SUMMARY 41

1. Scope

Wireless Application Protocol (WAP) is a result of continuous work to define an industry-wide specification for developing applications that operate over wireless communication networks. The scope for the WAP Forum is to define a set of standards to be used by service applications. The wireless market is growing very quickly and reaching new customers and services. To enable operators and manufacturers to meet the challenges in advanced services, differentiation and fast/flexible service creation, WAP defines a set of protocols in transport, session and application layers. For additional information on the WAP architecture, refer to *Wireless Application Protocol Architecture Specification* [WAP].

This document specifies the library interfaces for the standard set of libraries supported by WMLScript [WMLScript] to provide access to the core functionality of a WAP client. WMLScript is a language that can be used to provide programmed functionality to WAP based applications. It is part of the WAP platform and it can be used to add script support also to the client.

One of the main differences between ECMAScript [ECMA262] and WMLScript is the fact that WMLScript is compiled into bytecode *before* it is being sent to the client. This way the narrowband communication channels available today can be optimally utilised and the memory requirements for the client kept to the minimum. For the same reasons, many of the advanced features of the JavaScript language have been removed to make the language both optimal, easier to compile into bytecode and easier to learn.

Library support has been added to the WMLScript to replace some of the functionality that has been removed from ECMAScript in accordance to make the WMLScript more efficient. This feature provides access to built-in functionality and a means for future expansion without unnecessary overhead.

The following chapters describe the set of libraries defined to provide access to core functionality of a WAP client. This means that all libraries, except *Float*, are present in the client's scripting environment. Float library is optional and only supported with clients that can support floating-point arithmetic operations.

2. Document Status

This document is available online in the following formats:

- PDF format at http://www.wapforum.org/.

2.1 Copyright Notice

2.2 Errata

Known problems associated with this document are published at http://www.wapforum.org/.

2.3 Comments

Comments regarding this document can be submitted to the WAP Forum in the manner published at http://www.wapforum.org/.

3. References

3.1 Normative references

[ECMA262] Standard ECMA-262: "ECMAScript Language Specification", ECMA, June 1997

[IEEE754] ANSI/IEEE Std 754-1985: "IEEE Standard for Binary Floating-Point Arithmetic". Institute of Electrical and Electronics Engineers, New York (1985).

[RFC1738] "Uniform Resource Locators (URL)", T. Berners-Lee, et al., December 1994. URL: ftp://ftp.isi.edu/in-notes/rfc1738.txt

[RFC1808] "Relative Uniform Resource Locators", R. Fielding, June 1995. URL: ftp://ftp.isi.edu/in-notes/rfc1808.txt

[RFC2119] "Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997. URL: ftp://ftp.isi.edu/in-notes/rfc2119.tx

[UNICODE] "The Unicode Standard: Version 2.0", The Unicode Consortium, Addison-Wesley Developers Press, 1996. URL: http://www.unicode.org/

[WAP] "Wireless Application Protocol Architecture Specification", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/

[WML] "Wireless Markup Language Specification", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/

[WMLScript] "WMLScript Language Specification", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/

3.2 Informative References

[JavaScript] "JavaScript: The Definitive Guide", David Flanagan. O'Reilly & Associates, Inc. 1997

[RFC2068] "Hypertext Transfer Protocol - HTTP/1.1", R. Fielding, et al., January 1997. URL: ftp://ftp.isi.edu/in-notes/rfc2068.txt

[WAE] "Wireless Application Environment Specification", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/

[WSP] "Wireless Session Protocol", WAP Forum, 1998. URL: http://www.wapforum.org/

[XML] "Extensible Markup Language (XML), W3C Proposed Recommendation 10-February-1998, REC-xml-19980210", T. Bray, et al, February 10, 1998. URL: http://www.w3.org/TR/REC-xml

4. Definitions and abbreviations

4.1 Definitions

The following are terms and conventions used throughout this specification.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Bytecode - content encoding where the content is typically a set of low-level opcodes (ie, instructions) and operands for a targeted hardware (or virtual) machine.

Client - a device (or application) that initiates a request for connection with a server.

Content - subject matter (data) stored or generated at an origin server. Content is typically displayed or interpreted by a user agent in response to a user request.

Content Encoding - when used as a verb, content encoding indicates the act of converting a data object from one format to another. Typically the resulting format requires less physical space than the original, is easier to process or store and/or is encrypted. When used as a noun, content encoding specifies a particular format or encoding standard or process.

Content Format – actual representation of content.

Device - a network entity that is capable of sending and receiving packets of information and has a unique device address. A device can act as both a client or a server within a given context or across multiple contexts. For example, a device can service a number of clients (as a server) while being a client to another server.

JavaScript - a *de facto* standard language that can be used to add dynamic behaviour to HTML documents. JavaScript is one of the originating technologies of ECMAScript.

Origin Server - the server on which a given resource resides or is to be created. Often referred to as a web server or an HTTP server.

Resource - a network data object or service that can be identified by a URL. Resources may be available in multiple representations (e.g. multiple languages, data formats, size and resolutions) or vary in other ways.

Server - a device (or application) that passively waits for connection requests from one or more clients. A server may accept or reject a connection request from a client.

User - a user is a person who interacts with a user agent to view, hear or otherwise use a rendered content.

User Agent - a user agent (or content interpreter) is any software or device that interprets WML, WMLScript or resources. This may include textual browsers, voice browsers, search engines, etc.

Web Server - a network host that acts as an HTTP server.

WML - the Wireless Markup Language is a hypertext markup language used to represent information for delivery to a narrowband device, e.g. a phone.

WMLScript - a scripting language used to program the mobile device. WMLScript is an extended subset of the JavaScript™ scripting language.

4.2 Abbreviations

For the purposes of this specification, the following abbreviations apply:

API	Application Programming Interface
ECMA	European Computer Manufacturer Association
HTTP	HyperText Transfer Protocol [RFC2068]
LSB	Least Significant Bits
MSB	Most Significant Bits
RFC	Request For Comments
UI	User Interface
URL	Uniform Resource Locator [RFC1738]
W3C	World Wide Web Consortium
WWW	World Wide Web
WSP	Wireless Session Protocol
WTP	Wireless Transport Protocol
WAP	Wireless Application Protocol
WAE	Wireless Application Environment
WTA	Wireless Telephony Applications
WTAI	Wireless Telephony Applications Interface
WBMP	Wireless BitMaP

5. Notational Conventions

The libraries in this document are represented by providing the following information:

Name: Library name. The syntax of the library name follows the syntax specified in the [WMLScript] specification. Library names are case sensitive.

Examples: `Lang, String`

Library ID: The numeric identifier reserved for the library to be used by the WMLScript Compiler. The range of values reserved for this identifier is divided into the following two categories:

0 .. 32767	Reserved for standard libraries.
32768 .. 65535	Reserved for future use.

Description: A short description of the library and used conventions.

Each function in the library is represented by providing the following information:

Function: Specifies the function name and the number of function parameters. The syntax of the function name follows the syntax specified in the [WMLScript] specification. Function names are case sensitive.

Example: abs(*value*)
Usage: `var a = 3*Lang.abs(length);`

Function ID: The numeric identifier reserved for the function to be used by the WMLScript Compiler. The range of values reserved for this identifier is: **0..255**.

Description: Describes the function behaviour and its parameters.

Parameters: Specifies the function parameter types.

Example: *value* = Number

Return value: Specifies the type(s) of the return value.

Example: String or invalid.

Exceptions: Describes the possible special exceptions and error codes and the corresponding return values. Standard errors, common to all functions, are not described here (see 6.3 for more information about error handling).

Example: If the *value1* <= 0 and *value2* < 0 and not an integer then `invalid` is returned.

Example: Gives a few examples of how the function could be used.

```
var a = -3;
var b = Lang.abs(a);  // b = 3
```

6. WMLScript Compliance

WMLScript standard library functions provide a mechanism to extend the WMLScript language. Thus, the specified library functions must follow the WMLScript conventions and rules.

6.1 Supported Data Type

The following WMLScript types [WMLScript] are used in the function definitions to denote the type of both the function parameters and return values:

- *Boolean*, *Integer*, *Float*, *String* and *Invalid*

In addition to these, *number* can be used to denote a parameter type when both integer and floating-point parameter value types are accepted. *Any* can be used when the type can be any of the supported types.

6.2 Data Type Conversions

Since WMLScript is a weakly typed language, the conversions between the data types are done automatically if necessary (see [WMLScript] for more details about data type conversion rules). The library functions follow WMLScript operator data type conversion rules except where explicitly stated otherwise.

6.3 Error Handling

Error cases are handled in the same way as in the WMLScript language (see [WMLScript] for more details):

- An `invalid` function argument results in an `invalid` return value with no other side effects unless explicitly stated otherwise.
- A function argument that cannot be converted to the required parameter type results in an `invalid` return value with no side effects. See 6.2 for more information about data type conversions.
- Function dependent error cases are handled by returning a suitable error code specified in each function definition. These errors are documented as part of the function specification (exceptions).

6.4 Support for Integer-Only Devices

The WMLScript language has been designed to run also on devices that do not support floating-point operations. The WMLScript standard libraries have operations that require floating-point support. Thus, the following rules apply when the libraries are implemented for an integer-only device:

- Library functions accept arguments of the following type only: *boolean*, *integer*, *string* and *invalid*.
- All conversion rules related to floating-point data are ignored.
- *Lang.float()* function returns `false`.
- *Lang.parseFloat()* function returns `invalid`.
- All *Float* (see chapter 8) library functions return `invalid`.

7. Lang

Name: Lang

Library ID: 0

Description: This library contains a set of functions that are closely related to the WMLScript language core.

7.1 abs

Function: abs(*value*)

Function ID: 0

Description: Returns the absolute value of the given number. If the given number is of type integer then an integer value is returned. If the given number is of type floating-point then a floating-point value is returned.

Parameters: *value* = Number

Return value: Number or invalid.

Exceptions: -

Example:

```
var a = -3;
var b = Lang.abs(a);  // b = 3
```

7.2 min

Function: min(*value1*, *value2*)

Function ID: 1

Description: Returns the minimum value of the given two numbers. The value and type returned is the same as the value and type of the selected number. The selection is done in the following way:

- WMLScript operator data type conversion rules for *integers and floating-points* (see [WMLScript]) must be used to specify the data type (integer or floating-point) for comparison.
- Compare the numbers to select the smaller one.
- If the values are equal then the first value is selected.

Parameters: *value1* = Number
value2 = Number

Return value: Number or invalid.

Exceptions: -

Example:

```
var a = -3;
var b = Lang.abs(a);
var c = Lang.min(a,b);      // c = -3
var d = Lang.min(45, 76.3); // d = 45 (integer)
var e = Lang.min(45, 45.0); // e = 45 (integer)
```

7.3 max

Function: max(*value1, value2*)

Function ID: 2

Description: Returns the maximum value of the given two numbers. The value and type returned is the same as the value and type of the selected number. The selection is done in the following way:

- WMLScript operator data type conversion rules for *integers and floating-points* (see [WMLScript]) must be used to specify the data type (integer or floating-point) for comparison.
- Compare the numbers to select the larger one.
- If the values are equal then the first value is selected.

Parameters: *value1* = Number
value2 = Number

Return value: Number or invalid.

Exceptions: -

Example:

```
var a = -3;
var b = Lang.abs(a);
var c = Lang.max(a,b);      // c = 3
var d = Lang.max(45.5, 76); // d = 76 (integer)
var e = Lang.max(45.0, 45); // e = 45.0 (float)
```

7.4 parseInt

Function: parseInt(*value*)

Function ID: 3

Description: Returns an integer value defined by the string *value*. The legal integer syntax is specified by the WMLScript (see [WMLScript]) numeric string grammar for *decimal integer literals* with the following additional parsing rule:

- Parsing ends when the first character is encountered that is not a leading `'+'` or `'-'` or a decimal digit.

The result is the parsed string converted to an integer value.

Parameters: *value* = String

Return value: Integer or invalid.

Exceptions: In case of a parsing error an `invalid` value is returned.

Example:

```
var i = Lang.parseInt("1234");      // i = 1234
var j = Lang.parseInt(" 100 m/s");  // j = 100
```

7.5 parseFloat

Function: parseFloat(*value*)

Function ID: 4

Description: Returns a floating-point value defined by the string *value*. The legal floating-point syntax is specified by the WMLScript (see [WMLScript]) numeric string grammar for *decimal floating-point literals* with the following additional parsing rule:

- Parsing ends when the first character is encountered that cannot be parsed as being part of the floating-point representation.

The result is the parsed string converted to a floating-point value.

Parameters: *value* = String

Return value: Floating-point or invalid.

Exceptions: In case of a parsing error an `invalid` value is returned.

If the system does not support floating-point operations then an `invalid` value is returned.

Example:

```
var a = Lang.parseFloat("123.7");         // a = 123.7
var b = Lang.parseFloat("  +7.34e2 Hz");  // b = 7.34e2
var c = Lang.parseFloat(" 70e-2 F");      // c = 70.0e-2
var d = Lang.parseFloat("-.1 C");         // d = -0.1
var e = Lang.parseFloat(" 100 ");         // e = 100.0
var f = Lang.parseFloat("Number: 5.5");   // f = invalid
var g = Lang.parseFloat("7.3e meters");   // g = invalid
var h = Lang.parseFloat("7.3e- m/s");     // h = invalid
```

7.6 isInt

Function: isInt(*value*)

Function ID: 5

Description: Returns a boolean value that is `true` if the given *value* can be converted into an integer number by using parseInt(*value*). Otherwise `false` is returned.

Parameters: *value* = Any

Return value: Boolean or invalid.

Exceptions: -

Example:

```
var a = Lang.isInt(" -123");   // true
var b = Lang.isInt(" 123.33"); // true
var c = Lang.isInt("string");  // false
var d = Lang.isInt("#123");    // false
var e = Lang.isInt(invalid);   // invalid
```

7.7 isFloat

Function: isFloat(*value*)

Function ID: 6

Description: Returns a boolean value that is `true` if the given *value* can be converted into a floating-point number using parseFloat(*value*). Otherwise `false` is returned.

Parameters: *value* = Any

Return value: Boolean or invalid.

Exceptions: If the system does not support floating-point operations then an `invalid` value is returned.

Example:

```
var a = Lang.isFloat(" -123");   // true
var b = Lang.isFloat(" 123.33"); // true
var c = Lang.isFloat("string");  // false
var d = Lang.isFloat("#123.33"); // false
var e = Lang.isFloat(invalid);   // invalid
```

7.8 maxInt

Function: maxInt()

Function ID: 7

Description: Returns the maximum integer value.

Parameters: -

Return value: Integer 2147483647.

Exceptions: -

Example:

```
var a = Lang.maxInt();
```

7.9 minInt

Function: minInt()

Function ID: 8

Description: Returns the minimum integer value.

Parameters: -

Return value: Integer –2147483648.

Exceptions: -

Example:

```
var a = Lang.minInt();
```

7.10 float

Function: float()

Function ID: 9

Description: Returns `true` if floating-points are supported and `false` if not.

Parameters: -

Return value: Boolean.

Exceptions: -

Example:

```
var floatsSupported = Lang.float();
```

7.11 exit

Function: exit(*value*)

Function ID: 10

Description: Ends the interpretation of the WMLScript bytecode and returns the control back to the caller of the WMLScript interpreter with the given return *value*. This function can be used to perform a normal exit from a function in cases where the execution of the WMLScript bytecode should be discontinued.

Parameters: *value* = Any

Return value: None (this function ends the interpretation).

Exceptions: -

Example:

```
Lang.exit("Value: " + myVal); // Returns a string
Lang.exit(invalid);           // Returns invalid
```

7.12 abort

Function: abort(*errorDescription*)

Function ID: 11

Description: Aborts the interpretation of the WMLScript bytecode and returns the control back to the caller of the WMLScript interpreter with the return *errorDescription*. This function can be used to perform an abnormal exit in cases where the execution of the WMLScript should be discontinued due to serious errors detected by the calling function. If the type of the *errorDescription* is `invalid`, string `"invalid"` is used as the *errorDescription* instead.

Parameters: *errorDescription* = String

Return value: None (this function aborts the interpretation).

Exceptions: -

Example:

```
Lang.abort("Error: " + errVal);  // Error value is a string
```

7.13 random

Function: random(*value*)

Function ID: 12

Description: Returns an integer value with positive sign that is greater than or equal to 0 but less than or equal to the given *value*. The return value is chosen randomly or pseudo-randomly with approximately uniform distribution over that range, using an implementation-dependent algorithm or strategy.

If the *value* is of type floating-point, *Float.int()* is first used to calculate the actual integer *value*.

Parameters: *value* = Integer

Return value: Integer or invalid.

Exceptions: If *value* is equal to zero (0), the function returns zero.

If *value* is less than zero (0), the function returns `invalid`.

Example:

```
var a = 10;
var b = Lang.random(5.1)*a;    // b = 0..50
var c = Lang.random("string"); // c = invalid
```

7.14 seed

Function: seed(*value*)

Function ID: 13

Description: Initialises the pseudo-random number sequence and returns an empty string. If the *value* is zero or a positive integer then the given *value* is used for initialisation, otherwise a random, system dependent initialisation value is used.

If the *value* is of type floating-point, *Float.int()* is first used to calculate the actual integer *value*.

Parameters: *value* = Integer

Return value: String or invalid.

Exceptions: -

Example:

```
var a = Lang.seed(123);    // a = ""
var b = Lang.random(20);   // b = 0..20
var c = Lang.seed("seed"); // c = invalid (random seed left
                           //              unchanged)
```

8. Float

Name: Float

Library ID: 1

Description: This library contains a set of typical arithmetic floating-point functions that are frequently used by applications.

The implementation of these library functions is *optional* and implemented only by devices that can support floating-point operations (see 6.4). If floating-point operations are not supported, all functions in this library must return `invalid`.

8.1 int

Function: int(*value*)

Function ID: 0

Description: Returns the integer part of the given value. If the *value* is already an integer, the result is the *value* itself.

Parameters: *value* = Number

Return value: Integer or invalid.

Exceptions: -

Example:

```
var a = 3.14;
var b = Float.int(a);    // b = 3
var c = Float.int(-2.8); // c = -2
```

8.2 floor

Function: floor(*value*)

Function ID: 1

Description: Returns the greatest integer value that is not greater than the given *value*. If the *value* is already an integer, the result is the *value* itself.

Parameters: *value* = Number

Return value: Integer or invalid.

Exceptions: -

Example:

```
var a = 3.14;
var b = Float.floor(a);    // b = 3
var c = Float.floor(-2.8); // c = -3
```

8.3 ceil

Function: ceil(*value*)

Function ID: 2

Description: Returns the smallest integer value that is not less than the given *value*. If the *value* is already an integer, the result is the *value* itself.

Parameters: *value* = Number

Return value: Integer or invalid.

Exceptions: -

Example:
```
var a = 3.14;
var b = Float.ceil(a);    // b = 4
var c = Float.ceil(-2.8); // c = -2
```

8.4 pow

Function: pow(*value1, value2*)

Function ID: 3

Description: Returns an implementation-dependent approximation to the result of raising *value1* to the power of *value2*. If *value1* is a negative number then *value2* must be an integer.

Parameters: *value1* = Number
value2 = Number

Return value: Floating-point or invalid.

Exceptions: If *value1* == 0 and *value2* < 0 then `invalid` is returned.

If *value1* < 0 and *value2* is not an integer then `invalid` is returned.

Example:
```
var a = 3;
var b = Float.pow(a,2); // b = 9
```

8.5 round

Function: round(*value*)

Function ID: 4

Description: Returns the number value that is closest to the given *value* and is equal to a mathematical integer. If two integer number values are equally close to the *value*, the result is the larger number value. If the *value* is already an integer, the result is the *value* itself.

Parameters: *value* = Number

Return value: Integer or invalid.

Exceptions: -

Example:
```
var a = Float.round(3.5);  // a = 4
var b = Float.round(-3.5); // b = -3
var c = Float.round(0.5);  // c = 1
var d = Float.round(-0.5); // b = 0
```

8.6 sqrt

Function: sqrt(*value*)

Function ID: 5

Description: Returns an implementation-dependent approximation to the square root of the given *value*.

Parameters: *value* = Floating-point

Return value: Floating-point or invalid.

Exceptions: If *value* is a negative number then `invalid` is returned.

Example:
```
var a = 4;
var b = Float.sqrt(a); // b = 2.0
var c = Float.sqrt(5); // c = 2.2360679775
```

8.7 maxFloat

Function: maxFloat()

Function ID: 6

Description: Returns the maximum floating-point value supported by [IEEE754] single precision floating-point format.

Parameters: -

Return value: Floating-point 3.40282347E+38.

Exceptions: -

Example:
```
var a = Float.maxFloat();
```

8.8 minFloat

Function: minFloat()

Function ID: 7

Description: Returns the smallest nonzero floating-point value supported by [IEEE754] single precision floating-point format.

Parameters: -

Return value: Floating-point. Smaller than or equal to the normalised minimum single precision floating-point value: 1.17549435E-38.

Exceptions: -

Example:
```
var a = Float.minFloat();
```

9. String

Name: String

Library ID: 2

Description: This library contains a set of string functions. A string is an array of Unicode characters. Each of the characters has an index. The first character in a string has an index zero (0). The length of the string is the number of characters in the array.

The user of the String library can specify a special *separator* by which *elements* in a string can be separated. These elements can be accessed by specifying the separator and the element index. The first element in a string has an index zero (0). Each occurrence of the separator in the string separates two elements (no escaping of separators is allowed).

A *White space character* is any character with a Unicode 2.0 character code that is less than or equal to 32 (decimal).

9.1 length

Function: length(*string*)

Function ID: 0

Description: Returns the length (number of Unicode characters) of the given *string*.

Parameters: *string* = String

Return value: Integer or invalid.

Exceptions: -

Example:

```
var a = "ABC";
var b = String.length(a);    // b = 3
var c = String.length("");   // c = 0
var d = String.length(342);  // d = 3
```

9.2 isEmpty

Function: isEmpty(*string*)

Function ID: 1

Description: Returns a boolean `true` if the string length is zero and boolean `false` otherwise.

Parameters: *string* = String

Return value: Boolean or invalid.

Exceptions: -

Example:

```
var a = "Hello";
var b = "";
var c = String.isEmpty(a);    // c = false;
var d = String.isEmpty(b);    // d = true
var e = String.isEmpty(true); // e = false
```

9.3 charAt

Function: charAt(*string*, *index*)

Function ID: 2

Description: Returns a new string of length one containing the character at the specified *index* of the given *string*.

If the *index* is of type floating-point, *Float.int()* is first used to calculate the actual integer *index*.

Parameters: *string* = String
index = Number (the index of the character to be returned)

Return value: String or invalid.

Exceptions: If *index* is out of range then an empty string ("") is returned.

Example:

```
var a = "My name is Joe";
var b = String.charAt(a, 0);       // b = "M"
var c = String.charAt(a, 100);     // c = ""
var d = String.charAt(34, 0);      // d = "3"
var d = String.charAt(a, "first"); // e = invalid
```

9.4 subString

Function: subString(*string*, *startIndex*, *length*)

Function ID: 3

Description: Returns a new string that is a substring of the given *string*. The substring begins at the specified *startIndex* and its length (number of characters) is the given *length*. If the *startIndex* is less than 0 then 0 is used for the *startIndex*. If the *length* is larger than the remaining number of characters in the string, the *length* is replaced with the number of remaining characters.

If the *startIndex* or the *length* is of type floating-point, *Float.int()* is first used to calculate the actual integer value.

Parameters: *string* = String
startIndex = Number (the beginning index, inclusive)
length = Number (the length of the substring)

Return value: String or invalid.

Exceptions: If *startIndex* is larger than the last index an empty string ("") is returned.

If *length* <= 0 an empty string ("") is returned.

Example:

```
var a = "ABCD";
var b = String.subString(a, 1, 2);    // b = "BC"
var c = String.subString(a, 2, 5);    // c = "CD"
var d = String.subString(1234, 0, 2); // d = "12"
```

9.5 find

Function: find(*string*, *subString*)

Function ID: 4

Description: Returns the index of the first character in the *string* that matches the requested *subString*. If no match is found integer value `-1` is returned.

Two strings are defined to match when they are *identical*. Characters with multiple possible representations match only if they have the same representation in both strings. No case folding is performed.

Parameters: *string* = String
subString = String

Return value: Integer or invalid.

Exceptions: -

Example:
```
var a = "abcde";
var b = String.find(a, "cd");    // b = 2
var c = String.find(34.2, "de"); // c = -1
var d = String.find(a, "qz");    // d = -1
var e = String.find(34, "3");    // e = 0
```

9.6 replace

Function: replace(*string*, *oldSubString*, *newSubString*)

Function ID: 5

Description: Returns a new string resulting from replacing all occurrences of *oldSubString* in this string with *newSubString*.

Two strings are defined to match when they are *identical*. Characters with multiple possible representations match only if they have the same representation in both strings. No case folding is performed.

Parameters: *string* = String
oldSubString = String
newSubString = String

Return value: String or invalid.

Exceptions: -

Example:
```
var a = "Hello Joe.  What is up Joe?";
var newName = "Don";
var oldName = "Joe";
var c = String.replace(a, oldName, newName);
//  c = "Hello Don.  What is up Don?";
var d = String.replace(a, newName, oldName);
//  d = "Hello Joe.  What is up Joe?"
```

9.7 elements

Function: elements(*string*, *separator*)

Function ID: 6

Description: Returns the number of elements in the given *string* separated by the given *separator*.

Parameters: *string* = String
separator = String (the first character of the string used as separator)

Return value: Integer or invalid.

Exceptions: Returns `invalid` if the *separator* is an empty string.

Example:

```
var a = "My name is Joe; Age 50;";
var b = String.elements(a, " ");       // b = 6
var c = String.elements(a, ";");       // c = 3
var d = String.elements("", ";");      // d = 0
var e = String.elements("a", ";");     // e = 1
var f = String.elements(";", ";");     // f = 2
var g = String.elements(";;;;", ";;"); // g = 4 separator = ;
```

9.8 elementAt

Function: elementAt(*string*, *index, separator*)

Function ID: 7

Description: Search *string* for *index*'th element, elements being separated by *separator* and return the corresponding element. If the *index* is less than 0 then the first element is returned. If the *index* is larger than the number of elements then the last element is returned. If the *string* is an empty string then an empty string is returned.

If the *index* is of type floating-point, *Float.int()* is first used to calculate the actual *index* value.

Parameters: *string* = String
index = Number (the index of the element to be returned)
separator = String (the first character of the string used as separator)

Return value: String or invalid.

Exceptions: Returns `invalid` if the *separator* is an empty string.

Example:

```
var a = "My name is Joe; Age 50;";
var b = String.elementAt(a, 0, " ");  // b = "My"
var c = String.elementAt(a, 14, ";"); // c = ""
var d = String.elementAt(a, 1, ";");  // d = " Age 50"
```

9.9 removeAt

Function: removeAt(*string*, *index*, *separator*)

Function ID: 8

Description: Returns a new string where the element and the corresponding *separator* (if existing) with the given *index* are removed from the given *string*. If the *index* is less than 0 then the first element is removed. If the *index* is larger than the number of elements then the last element is removed. If the *string* is empty, the function returns a new empty string.

If the *index* is of type floating-point, *Float.int()* is first used to calculate the actual *index* value.

Parameters: *string* = String
index = Number (the index of the element to be deleted)
separator = String (the first character of the string used as separator)

Return value: String or invalid.

Exceptions: Returns `invalid` if the *separator* is an empty string.

Example:

```
var a = "A A;  B C D";
var s = " ";
var b = String.removeAt(a, 1, s);
//  b = "A  B C D"
var c = String.removeAt(a, 0, ";");
//  c = "  B C D"
var d = String.removeAt(a, 14, ";");
//  d = "A A"
```

9.10 replaceAt

Function: replaceAt(*string*, *element*, *index*, *separator*)

Function ID: 9

Description: Returns a string with the current element at the specified *index* replaced with the given *element*. If the *index* is less than 0 then the first element is replaced. If the *index* is larger than the number of elements then the last element is replaced. If the *string* is empty, the function returns a new string with the given *element*.

If the *index* is of type floating-point, *Float.int()* is first used to calculate the actual *index* value.

Parameters: *string* = String
element = String
index = Number (the index of the element to be replaced)
separator = String (the first character of the string used as separator)

Return value: String or invalid.

Exceptions: Returns `invalid` if the *separator* is an empty string.

Example:

```
var a = "B C; E";
var s = " ";
var b = String.replaceAt(a, "A", 0, s);
//  b = "A C; E"
var c = String.replaceAt(a, "F", 5, ";");
//  c = "B C;F"
```

9.11 insertAt

Function: insertAt(*string*, *element*, *index*, *separator*)

Function ID: 10

Description: Returns a string with the *element* and the corresponding *separator* (if needed) inserted at the specified element *index* of the original *string*. If the *index* is less than 0 then 0 is used as the *index*. If the *index* is larger than the number of elements then the element is appended at the end of the *string*. If the *string* is empty, the function returns a new string with the given *element*.

If the *index* is of type floating-point, *Float.int()* is first used to calculate the actual *index* value.

Parameters: *string* = String (original string)
element = String (element to be inserted)
index = Number (the index of the element to be added)
separator = String (the first character of the string used as separator)

Return value: String or invalid.

Exceptions: Returns `invalid` if the *separator* is an empty string.

Example:

```
var a = "B C; E";
var s = " ";
var b = String.insertAt(a, "A", 0, s);
//  b = "A B C; E"
var c = String.insertAt(a, "X", 3, s);
//  c = "B C; E X"
var d = String.insertAt(a, "D", 1, ";");
//  d = "B C;D; E"
var e = String.insertAt(a, "F", 5, ";");
//  e = "B C; E;F"
```

9.12 squeeze

Function: squeeze(*string*)

Function ID: 11

Description: Returns a string where all consecutive series of white spaces within the *string* are reduced to one.

Parameters: *String* = String

Return value: String or invalid.

Exceptions: -

Example:

```
var a = "Hello";
var b = " Bye    Jon .  See you!  ";
var c = String.squeeze(a); // c = "Hello";
var d = String.squeeze(b); // d = " Bye Jon . See you! ";
```

9.13 trim

Function: trim(*string*)

Function ID: 12

Description: Returns a string where all trailing and leading white spaces in the given *string* have been trimmed.

Parameters: *String* = String

Return value: String or invalid.

Exceptions: -

Example:

```
var a = "Hello";
var b = " Bye    Jon .  See you!  ";
var c = String.trim(a); // c = "Hello"
var d = String.trim(b); // d = "Bye    Jon .  See you!"
```

9.14 compare

Function: compare(*string1*, *string2*)

Function ID: 13

Description: The return value indicates the lexicographic relation of *string1* to *string2*. The relation is based on the relation of the Unicode character codes. The return value is –1 if *string1* is less than *string2*, 0 if *string1* is identical to *string2* or 1 if *string1* is greater than *string2*.

Parameters: *String1* = String
String2 = String

Return value: Integer or invalid.

Exceptions: -

Example:

```
var a = "Hello";
var b = "Hello";
var c = String.compare(a, b);          // c = 0
var d = String.compare("Bye", "Jon"); // d = -1
var e = String.compare("Jon", "Bye"); // e = 1
```

9.15 toString

Function: toString(*value*)

Function ID: 14

Description: Returns a string representation of the given *value*. This function performs exactly the same conversions as supported by the [WMLScript] language (automatic conversion from boolean, integer and floating-point values to strings) except that `invalid` value returns the string `"invalid"`.

Parameters: *value* = Any

Return value: String.

Exceptions: -

Example:

```
var a = String.toString(12);   // a = "12"
var b = String.toString(true); // b = "true"
```

9.16 format

Function: format(*format*, *value*)

Function ID: 15

Description: Converts the given *value* to a string by using the given formatting provided as a *format* string. The format string can contain only one format specifier, which can be located anywhere inside the string. If more than one is specified, only the first one (leftmost) is used and the remaining specifiers are replaced by an empty string. The format specifier has the following form:

```
% [width] [.precision] type
```

The **`width`** argument is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left until the minimum width is reached. The `width` argument never causes the *value* to be truncated. If the number of characters in the output value is greater than the specified width or, if width is not given, all characters of the *value* are printed (subject to the precision argument).

The **`precision`** argument specifies a nonnegative decimal integer, preceded by a period (.), that can be used to set the precision of the output value. The interpretation of this value depends on the given `type`:

- **d** Specifies the minimum number of digits to be printed. If the number of digits in the *value* is less than precision, the output value is padded on the left with zeroes. The value is not truncated when the number of digits exceeds precision. Default precision is 1. If precision is specified as 0 and the value to be converted is 0, the result is an empty string.
- **f** Specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. Default precision is 6; if precision is 0 or if the period (.) appears without a number following it, no decimal point is printed.
- **s** Specifies the maximum number of characters to be printed. By default, all characters are printed.

Unlike the `width` argument, the `precision` argument can cause either truncation of the output value or rounding of a floating-point value.

The **`type`** argument is the only required format argument; it appears after any optional format fields. The type character determines whether the given *value* is interpreted as integer, floating-point or string. The supported `type` arguments are:

- **d** *Integer*: The output value has the form [-]dddd, where dddd is one or more decimal digits.
- **f** *Floating-point*: The output value has the form [-]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number and the number of digits after the decimal point depends on the requested precision.
- **s** *String*: Characters are printed up to the end of the string or until the precision value is reached.

Percent character (%) in the format string can be presented by preceding it with another percent character (%%).

Parameters: *format* = String
value = Any

Return value: String or invalid.

Exceptions: Illegal format specifier results in an `invalid` return value.

Example:

```
var a = 45;
var b = -45;
var c = "now";
var d = 1.2345678;
var e = String.format("e: %6d", a);   // e = "e:     45"
var f = String.format("%6d", b);      // f = "   -45"
var g = String.format("%6.4d", a);    // g = "  0045"
var h = String.format("%6.4d", b);    // h = " –0045"
var i = String.format("Do it %s", c); // i = "Do it now"
var j = String.format("%3f", d);      // j = "1.234567"
var k = String.format("%10.2f%%", d); // k = "      1.23%"
var l = String.format("%3f %2f.", d); // l = "1.234567 ."
var m = String.format("%.0d", 0);     // m = ""
var n = String.format("%7d", "Int");  // n = invalid
var o = String.format("%s", true);    // o = "true"
```

10. URL

Name: URL

Library ID: 3

Description: This library contains a set of functions for handling both absolute URLs and relative URLs. The general URL syntax supported is (see [RFC1808]):

```
<scheme>://<host>:<port>/<path>;<params>?<query>#<fragment>
```

10.1 isValid

Function: isValid(*url*)

Function ID: 0

Description: Returns `true` if the given *url* has the right URL syntax, otherwise returns `false`. Both absolute and relative URLs are supported. Relative URLs are not resolved into absolute URLs.

Parameters: *url* = String

Return value: Boolean or invalid.

Exceptions: -

Example:

```
var a = URL.isValid("http://w.hst.com/script#func()");
//  a = true
var b = URL.isValid("../common#test()");
//  b = true
var c = URL.isValid("experimental?://www.host.com/cont");
//  c = false
```

10.2 getScheme

Function: getScheme(*url*)

Function ID: 1

Description: Returns the scheme used in the given *url*. Both absolute and relative URLs are supported. Relative URLs are not resolved into absolute URLs.

Parameters: *url* = String

Return value: String or invalid.

Exceptions: If an invalid URL syntax is encountered while extracting the scheme an `invalid` value is returned.

Example:

```
var a = URL.getScheme("http://w.h.com/path#frag");
//  a = "http"
var b = URL.getScheme("w.h.com/path#frag");
//  b = ""
```

10.3 getHost

Function: getHost(*url*)

Function ID: 2

Description: Returns the host specified in the given *url*. Both absolute and relative URLs are supported. Relative URLs are not resolved into absolute URLs.

Parameters: *url* = String

Return value: String or invalid.

Exceptions: If an invalid URL syntax is encountered while extracting the host part an `invalid` value is returned.

Example:

```
var a = URL.getHost("http://w.h.com/path#frag");
//  a = "w.h.com"
var b = URL.getHost("path#frag");
//  b = ""
```

10.4 getPort

Function: getPort(*url*)

Function ID: 3

Description: Returns the port number specified in the given *url*. If no port is specified then an empty string is returned. Both absolute and relative URLs are supported. Relative URLs are not resolved into absolute URLs.

Parameters: *url* = String

Return value: String or invalid.

Exceptions: If an invalid URL syntax is encountered while extracting the port number an `invalid` value is returned.

Example:

```
var a = URL.getPort("http://w.h.com:80/path#frag");
//  a = "80"
var b = URL.getPort("http://w.h.com/path#frag");
//  b = ""
```

10.5 getPath

Function: getPath(*url*)

Function ID: 4

Description: Returns the path specified in the given *url*. Both absolute and relative URLs are supported. Relative URLs are not resolved into absolute URLs.

Parameters: *url* = String

Return value: String or invalid.

Exceptions: If an invalid URL syntax is encountered while extracting the path an `invalid` value is returned.

Example:

```
a = URL.getPath("http://w.h.com/home/sub/comp#frag");
//  a = "/home/sub/comp"
b = URL.getPath("../home/sub/comp#frag");
//  b = "../home/sub/comp"
```

10.6 getParameters

Function: getParameters(*url*)

Function ID: 5

Description: Returns the parameters used in the given *url*. If no parameters are specified an empty string is returned. Both absolute and relative URLs are supported. Relative URLs are not resolved into absolute URLs.

Parameters: *url* = String

Return value: String or invalid.

Exceptions: If an invalid URL syntax is encountered while extracting the parameters an `invalid` value is returned.

Example:

```
a = URL.getParameters("http://w.h.com/script;3;2?x=1&y=3");
//  a = "3;2"
b = URL.getParameters("../script;3;2?x=1&y=3");
//  b = "3;2"
```

10.7 getQuery

Function: getQuery(*url*)

Function ID: 6

Description: Returns the query part specified in the given *url*. If no query part is specified an empty string is returned. Both absolute and relative URLs are supported. Relative URLs are not resolved into absolute URLs.

Parameters: *url* = String

Return value: String or invalid.

Exceptions: If an invalid URL syntax is encountered while extracting the query part an `invalid` value is returned.

Example:

```
a = URL.getParameters("http://w.h.com/home;3;2?x=1&y=3");
//  a = "x=1&y=3"
```

10.8 getFragment

Function: getFragment(*url*)

Function ID: 7

Description: Returns the fragment used in the given *url*. If no fragment is specified an empty string is returned. Both absolute and relative URLs are supported. Relative URLs are not resolved into absolute URLs.

Parameters: *url* = String

Return value: String or invalid.

Exceptions: If an invalid URL syntax is encountered while extracting the fragment an `invalid` value is returned.

Example:

```
var a = URL.getFragment("http://w.h.com/cont#frag");
//  a = "frag"
```

10.9 getBase

Function: getBase()

Function ID: 8

Description: Returns an absolute URL (without the fragment) of the current WMLScript compilation unit.

Parameters: -

Return value: String.

Exceptions: -

Example:
```
var a = URL.getBase();
//  Result: "http://www.host.com/test.scr"
```

10.10 getReferer

Function: getReferer()

Function ID: 9

Description: Returns the smallest relative URL (relative to the base URL of the current compilation unit, see 10.9) to the resource that called the current compilation unit. Local function calls do not change the referer. If the current compilation unit does not have a referer then an empty string is returned.

Parameters: -

Return value: String.

Exceptions: -

Example:
```
var base    = URL.getBase();
//  base    = "http://www.host.com/current.scr"
var referer = URL.getReferer();
//  referer = "app.wml"
```

10.11 resolve

Function: resolve(*baseUrl*, *embeddedUrl*)

Function ID: 10

Description: Returns an absolute URL from the given *baseUrl* and the *embeddedUrl* according to the rules specified in [RFC1808]. If the *embeddedUrl* is already an absolute URL, the function returns it without modification.

Parameters: *baseUrl* = String
embeddedUrl = String

Return value: String or invalid.

Exceptions: If an invalid URL syntax is encountered as part of the resolution an `invalid` value is returned.

Example:
```
var a = URL.resolve("http://foo.com/","foo.vcf");
//  a = "http://foo.com/foo.vcf"
```

10.12 escape

Function: escape(*string*)

Function ID: 11

Description: This function computes a new version of a *string* value in which special characters specified by [RFC1738] have been replaced by a hexadecimal escape sequence. The result thus contains no special characters that might have special meaning within a URL.

For special characters, whose Unicode encoding is `0xFF` or less, a two-digit escape sequence of the form `%xx` is used in accordance with [RFC1738].

Both absolute and relative URLs are supported. Relative URLs are not resolved into absolute URLs.

Parameters: *string* = String

Return value: String or invalid.

Exceptions: If *string* contains characters that are not part of the US-ASCII character set an `invalid` value is returned.

If an invalid URL syntax is encountered while escaping the given URL, an `invalid` value is returned.

Example:

```
var a = URL.escape("http://w.h.com/dck?x=\u00ef#crd");
//  a = "http://w.h.com/dck?x=%ef#crd"
```

10.13 unescape

Function: unescape(*string*)

Function ID: 12

Description: The unescape function computes a new version of a *string* value in which each escape sequences of the sort that might be introduced by the escape function (see 10.12) is replaced with the character that it represents.

Both absolute and relative URLs are supported. Relative URLs are not resolved into absolute URLs.

Parameters: *string* = String

Return value: String or invalid.

Exceptions: If *string* contains characters that are not part of the US-ASCII character set, an `invalid` value is returned.

If an invalid URL syntax is encountered while unescaping the given URL, an `invalid` value is returned.

Example:

```
var a = URL.unescape("http://w.h.com/dck?x=%31%32#crd");
//  a = "http://w.h.com/dck?x=12#crd"
```

10.14 escapeString

Function: escapeString(*string*)

Function ID: 13

Description: This function computes a new version of a *string* value in which special characters (unsafe, reserved and unprintable characters) specified by [RFC1738] have been replaced by a hexadecimal escape sequence. The given string is escaped as such; no URL parsing is performed.

For special characters, whose Unicode encoding is `0xFF` or less, a two-digit escape sequence of the form `%xx` is used in accordance with [RFC1738].

Parameters: *string* = String

Return value: String or invalid.

Exceptions: If *string* contains characters that are not part of the US-ASCII character set an `invalid` value is returned.

Example:
```
var a = URL.escapeString("http://w.h.com/dck?x=\u00ef#crd");
//  a = "http%3a%2f%2fw.h.com%2fdck%3fx%3d%ef%23crd"
```

10.15 unescapeString

Function: unescapeString(*string*)

Function ID: 14

Description: The unescape function computes a new version of a *string* value in which each escape sequences of the sort that might be introduced by the *URL.escapeString()* function (see 10.14) is replaced with the character that it represents. The given string is unescaped as such; no URL parsing is performed.

Parameters: *string* = String

Return value: String or invalid.

Exceptions: If *string* contains characters that are not part of the US-ASCII character set, an `invalid` value is returned.

Example:
```
var a = "http%3a%2f%2fw.h.com%2fdck%3fx%3d12%23crd";
var b = URL.unescapeString(a);
//  b = "http://w.h.com/dck?x=12#crd"
```

10.16 loadString

Function: loadString(*url*, *contentType*)

Function ID: 15

Description: Returns the content denoted by the given absolute *url* and the *content type*. The *content type* must specify only one content type, no wildcards are allowed.

The behaviour of this function is the following:

- The content with the given *content type* and *url* is loaded. The rest of the attributes needed for the content load are specified by the default settings of the user agent.
- If the load is successful and the returned content type matches the given *content type* then the content is converted to a Unicode string and returned.
- If the load is unsuccessful or the returned content is of wrong content type then a scheme specific error code is returned.

Parameters: *url* = String
contentType = String

Return value: String, integer or invalid.

Exceptions: Returns an integer *error code* that depends on the used URL scheme in case the load fails. If HTTP [RFC2068] or WSP (see [WAE]) schemes are used, HTTP error codes are returned.

If *content type* specifies more than one content type or contains wildcards then `invalid` value is returned.

Example:

```
var myUrl = "http://www.host.com/vcards/myaddr.vcf";
myCard = URL.loadString(myUrl,"text/x-vcard");
```

11. WMLBrowser

Name: WMLBrowser

Library ID: 4

Description: This library contains functions by which WMLScript can access the different WML Browser variables and attributes. If the system does not support WML Browser then all following functions return `invalid` value.

11.1 getVar

Function: getVar(*name*)

Function ID: 0

Description: Returns the value of the variable with the given *name* in the current browser context. Returns an empty string if the given variable does not exist.

Parameters: *name* = String

Return value: String or invalid.

Exceptions: -

Example:

```
var a = WMLBrowser.getVar("name");
// a = "Jon" or whatever value the variable has.
```

11.2 setVar

Function: setVar(*name*, *value*)

Function ID: 1

Description: Returns `true` if the variable with the given *name* is successfully set to contain the given *value* in the current browser context, `false` otherwise.

Parameters: *name* = String
value = String

Return value: Boolean or invalid.

Exceptions: -

Example:

```
var a = WMLBrowser.setVar("name", Mary); // a = true
```

11.3 go

Function: go(*url*, *vars*)

Function ID: 2

Description: Specifies the content denoted by the given *url* and the variable mappings to be loaded. This function has the same semantics as the GO task in WML (see [WML] for more information). The content is loaded after the WMLScript interpreter returns the control back to the WML browser. No content is loaded if the given *url* is an empty string (""). This function returns an empty string.

go() and *prev()* (see 11.4) functions override each other. Both of these functions can be called multiple times before returning the control back to the WML browser. However, only the settings of the last call stay in effect. In particular, if the last call to *go()* or *prev()* set the URL to an empty string (""), all requests are effectively cancelled.

Parameters: *url* = String
vars = String

Return value: String or invalid.

Exceptions: -

Example:

```
var card = "http://www.host.com/loc/app.dck#start";
var vars = "x=3&y=2";
WMLBrowser.go(card,vars);
```

11.4 prev

Function: prev(*vars*)

Function ID: 3

Description: Signals the WML browser to go back to the previous WML card with the given variable mappings. This function has the same semantics as the PREV task in WML (see [WML] for more information). The previous card is loaded after the WMLScript interpreter returns the control back to the WML browser. This function returns an empty string.

prev() and *go()* (see 11.3) functions override each other. Both of these functions can be called multiple times before returning the control back to the WML browser. However, only the settings of the last call stay in effect. In particular, if the last call to *go()* or *prev()* set the URL to an empty string (""), all requests are effectively cancelled.

Parameters: *vars* = String

Return value: String or invalid.

Exceptions: -

Example:

```
WMLBrowser.prev("price=" + currentPrice);
```

11.5 newContext

Function: newContext()

Function ID: 4

Description: Clears the current WML Browser context and returns an empty string. This function has the same semantics as the NEWCONTEXT attribute in WML (see [WML] for more information).

Parameters: -

Return value: String or invalid.

Exceptions: -

Example:
```
WMLBrowser.newContext();
```

11.6 getCurrentCard

Function: getCurrentCard()

Function ID: 5

Description: Returns the smallest relative URL (relative to the base of the current compilation unit, see 10.9 for information about how to access the current base) specifying the card (if any) currently being processed by the WML Browser (see [WML] for more information). The function returns an absolute URL in case the WML deck containing the current card does not have the same base as the current compilation unit.

Parameters: -

Return value: String or invalid.

Exceptions: Returns `invalid` in case there is no current card.

Example:
```
var a = WMLBrowser.getCurrentCard();
//  a = "deck#input"
```

11.7 refresh

Function: refresh()

Function ID: 6

Description: Forces the WML Browser to update its context and returns an empty string. This function has the same semantics as the REFRESH task in WML (see [WML] for more information). As a result, the user interface is updated to reflect the updated context.

Parameters: -

Return value: String or invalid.

Exceptions: -

Example:
```
WMLBrowser.setVar("name","Zorro");
WMLBrowser.refresh();
```

12. Dialogs

Name: Dialogs

Library ID: 5

Description: This library contains a set of typical user interface functions.

12.1 prompt

Function: prompt(*message, defaultInput*)

Function ID: 0

Description: Displays the given *message* and prompts for user input. The *defaultInput* parameter contains the initial content for the user input. Returns the user input.

Parameters: *message* = String
defaultInput = String

Return value: String or invalid.

Exceptions: -

Example:
```
var a = "09-555 3456";
var b = Dialogs.prompt("Phone number: ",a);
```

12.2 confirm

Function: confirm(*message, ok, cancel*)

Function ID: 1

Description: Displays the given *message* and two reply alternatives: *ok* and *cancel*. Waits for the user to select one of the reply alternatives and returns `true` for *ok* and `false` for *cancel*.

Parameters: *message* = String
ok = String (text, empty string results in the default implementation-dependent text)
cancel = String (text, empty string results in the default implementation-dependent text)

Return value: Boolean or invalid.

Exceptions: -

Example:
```
function onAbort() {
  return Dialogs.confirm("Are you sure?","Yes","Well...");
};
```

12.3 alert

Function: alert(*message*)

Function ID: 2

Description: Displays the given *message* to the user, waits for the user confirmation and returns an empty string.

Parameters: *message* = String

Return value: String or invalid.

Exceptions: -

Example:

```
function testValue(textElement) {
  if (String.length(textElement) > 8) {
    Dialogs.alert("Enter name < 8 chars!");
  };
};
```

Appendix A. Library Summary

The libraries and their library identifiers:

Library name	Library ID	Page
Lang	0	11
Float	1	17
String	2	20
URL	3	29
WMLBrowser	4	36
Dialogs	5	39

The libraries and their functions:

Lang library	Function ID
abs	0
min	1
max	2
parseInt	3
parseFloat	4
isInt	5
isFloat	6
maxInt	7
minInt	8
float	9
exit	10
abort	11
random	12
seed	13

Float library	Function ID
int	0
floor	1
ceil	2
pow	3
round	4
sqrt	5
maxFloat	6
minFloat	7

String library	Function ID
length	0
isEmpty	1
charAt	2
subString	3
find	4

String library	**Function ID**
replace	5
elements	6
elementAt	7
removeAt	8
replaceAt	9
insertAt	10
squeeze	11
trim	12
compare	13
toString	14
format	15

URL library	**Function ID**
isValid	0
getScheme	1
getHost	2
getPort	3
getPath	4
getParameters	5
getQuery	6
getFragment	7
getBase	8
getReferer	9
resolve	10
escape	11
unescape	12
escapeString	13
unescapeString	14
loadString	15

WMLBrowser library	**Function ID**
getVar	0
setVar	1
go	2
prev	3
newContext	4
getCurrentCard	5
refresh	6

Dialogs library	**Function ID**
prompt	0
confirm	1
alert	2