WMLScript Reference

Version 1.1

http://www.forum.nokia.com
Product number: SDK-01-000-003
September 1999

WMLScript Reference

Version 1.1

Product number: SDK-01-000-003

We welcome and consider all comments and suggestions. Please send them to:

Nokia Group Finland
P.O. Box 226,
FIN-00045 NOKIA GROUP

Tel. +358 9 180 71
Fax. +358 9 656 388

Internet mail address:
wap.sw.developer@nokia.com

http://www.forum.nokia.com

This document is part of the Nokia Wireless Application Protocol Toolkit. The contents of this guide are based on the Wireless Application Protocol WMLScript Specification Version 1.1 (WMLScript Specification Version 16-June-1999) and on the Wireless Application Protocol WMLScript Standard Libraries Specification Version 1.1 (WMLScript Standard Libraries Specification Version 16-June-1999).

Contents

Introduction

This guide introduces the Wireless Markup Language Script (WMLScript) and its standard libraries. WMLScript is part of the Wireless Application Prototocol (WAP) application layer, and you can use it to add client side procedural logic to WML cards and decks. The language is based on ECMAScript, but it has been modified to better support low bandwidth devices such as mobile phones. You can use WMLScript with Wireless Markup Language (WML) to provide intelligence to the clients, or you can use it as a stand-alone tool.

WMLScript has a defined bytecode and an interpreter reference architecture. In addition, all WMLScript data is transmitted in binary format over wireless networks. This allows you to use the narrowband communication channels to the full and keep the memory needed by the client to a minimum. Many advanced features of the ECMAScript language have been dropped to make the language smaller and easier to compile into bytecode, and easier to learn. For example, WMLScript is a procedural language supporting locally installed standard libraries.

Benefits of using WMLScript

WMLScript was designed to provide general scripting capabilities to the WAP architecture. Specifically, you can use WMLScript to complement WML, which is e based on Extensible Markup Language (XML). It was designed for specifying application content for narrowband devices like mobile phones. This content can include text, images, selection lists, and so on. In addition, you can use simple formatting to make the user interfaces more attractive and readable. However, all the content is static and there is no way to extend the language without modifying the WML itself. The following list contains some features that are not supported by WML:

Checking the validity of user input.

Accessing facilities of the user agent. For example, on a mobile phone, allowing the programmer to make phone calls, send messages, and add phone numbers to the address book or access the SIM card.

Generating messages and dialogs locally, thus allowing alerts, error messages, confirmations etc to be seen faster by the user.

Allowing extensions to the user agent software and configuring a user agent after it has been deployed.

WMLScript was designed to overcome these limitations and to provide programmable functionality that can be used over narrowband communication links in clients with limited capabilities.

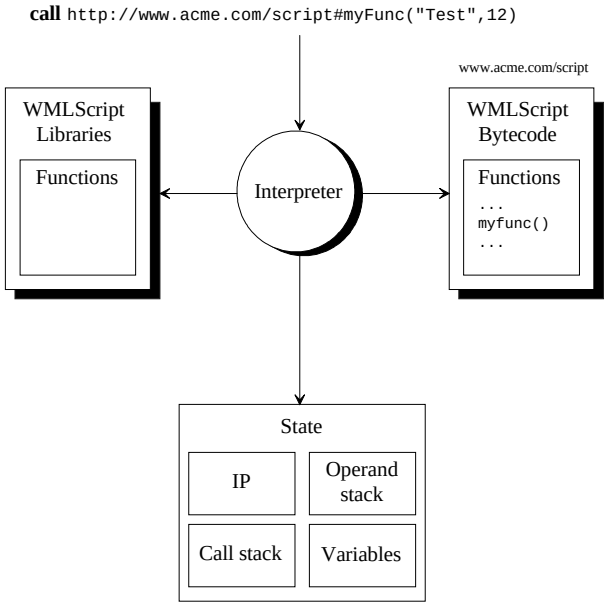
Many of the services that can be used with small mobile clients can be implemented withWML. However, the human behavioural compatibility of scripting improves the standard browsing and presentation facilities of WML. You can use scripting to support more advanced user interface functions, add intelligence to the client, provide access to the user agent and its peripheral functionality, and reduce the bandwidth needed to send data between the server and client.

WMLScript bytecode interpreter

The textual format of WMLScript language is compiled into a binary format before it can be interpreted by the WMLScript bytecode interpreter. The WMLScript compiler encodes one WMLScript compilation unit into WMLScript bytecode. A WMLScript compilation unit contains pragmas and any number of WMLScript functions. The WMLScript compiler takes one compilation unit as input and generates the WMLScript bytecode as its output.

Interpreter Architecture

The WMLScript interpreter takes WMLScript bytecode as its input and executes encoded functions as they are called. The following figure contains the main parts involved in WMLScript bytecode interpretation:



General architecture of the WMLScript interpreter.

You can use the WMLScript interpreter to call and execute functions in a compilation unit encoded as WMLScript bytecode. Each function specifies the number of parameters it accepts and the instructions used to express its behaviour. Thus, a call to a WMLScript function must specify the function, the function call arguments and the compilation unit in which the function is declared. Once the execution completes normally, the WMLScript interpreter returns the control and return values to the caller.

Execution of a WMLScript function involves interpreting the instructions residing in the WMLScript bytecode. While a function is being interpreted, the WMLScript interpreter maintains the following state information:

IP (Instruction Pointer): This points to an instruction in the bytecode that is being interpreted.

Variables: Maintenance of function parameters and variables.

Operand stack: Is used for expression evaluation and passing arguments between the called functions and the caller.

Function call stack: The WMLScript function can call other functions in the current or separate compilation unit or make calls to library functions. The function call stack holds the information on the functions and their return addresses.

Access control

WMLScript provides two mechanisms for controlling access to the functions in the WMLScript compilation unit: an external keyword and a specific access control pragma. Thus, the WMLScript interpreter must support the following behaviour:

External functions: Only functions specified as external can be called from other compilation units.

Access control: Access to the external functions defined inside a compilation unit is allowed from other compilation units that match the given access domain and access path definitions.

Character Set

The WMLScript Interpreter must use only one character set (*native character set*) for all of its string opreations. Transcoding between different character sets and their encodings is allowed as long as the WMLScript string operations are performed using only the native character set. The native character set can be requested by using the Lang library function *Lang.characterSet().*

Typographical conventions

The following conventions are used in this guide.

Notation	Explanation
`Courier`	Text that appears onscreen, program code, file and directory names.
`Courier Bold`	WML tag syntax, Uniform Resource Locators and other types of specialized language.
Italic	References to other guides and documents, new terminology.

Related documents

The following documents contain additional information on the Nokia WAP Toolkit and the Wireless Application Protocol. The web address provided after each document specifies the Internet location where the document can be obtained.

Documents included in the Nokia WAP Toolkit

Nokia WAP Toolkit Getting Started

This guide provides basic information on the Nokia WAP Toolkit and WML, and provides instructions on installing and using the product.

Nokia WAP Toolkit Developer's Guide

This guide provides information on the Nokia WAP Toolkit andWML for developers who want to create their own wireless services on the WAP platform.

WML Reference

This guide provides reference information onWML. It introduces the WML syntax and provides code examples.

Other references

Wireless Markup Language Specification.
WAP Forum, 16-June-1999.
http://www.wapforum.org/

WMLScript Specification.
WAP Forum, 16-June-1999.
http://www.wapforum.org/

WMLScript Standard Libraries Specification.
WAP Forum, 16-June-1999.
http://www.wapforum.org/

Wireless Application Protocol Architecture Specification.
WAP Forum, 16-June-1999.
http://www.wapforum.org/

Wireless Session Protocol Specification.
WAP Forum, 16-June-1999.
http://www.wapforum.org/

ISO 10646: Information Technology - Universal Multiple Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane.

The Unicode Standard: Version 2.0.
`http://www.unicode.org`

Extensible Markup Language (XML).
W3C Proposed Recommendation, 10-February-1998, REC-xml-19980210.
`http://www.w3.org/TR/REC-xml`

RFC2068: Hypertext Transfer Protocol - HTTP/1.1.
`http://www.w3.org/Protocols/`

RFC2119: Key words for use in RFCs to Indicate Requirement Levels.
`http://info.internet.isi.edu/in-notes/rfc/files/rfc2119.txt`

RFC2279: UTF-8, a transformation format of Unicode and ISO 10646.
`http://info.internet.isi.edu/in-notes/rfc/files/rfc2279.txt`

RFC2396: Uniform Resource Identifiers (URI): Generic Syntax

WMLScript core

This chapter provides an overview of the basic types, variables, expressions and statements of WMLScript.

WMLScript and URLs

The World Wide Web is a network of databases and devices where three areas of specification ensure widespread interoperability:

A unified naming model. Naming is implemented with Uniform Resource Locators (URLs), which provide standard ways of naming any network resource.

Standard protocols to transport information, for example, HTTP.

Standard content types, for example, HTML, WMLScript.

WMLScript assumes the same reference architecture as HTML and the World Wide Web. The WMLScript compilation unit is named using URLs and can be retrieved over standard protocols using HTTP semantics, such as Wireless Session Protocol (WSP). URLs and the character set used to specify URLs are defined in *RFC2396*.

In WMLScript, URLs are used in the following situations:

When a user agent wants to make a WMLScript call. For more information, see "URL calls and parameter passing" on page 8.

When specifying external compilation units. For more information, see "External compilation units" on page 38.

When specifying access control information. For more information, see "Access control" on page 39.

For detailed information on URL syntax, see "URL call syntax" on page 123.

Fragment anchors

WMLScript has adopted the HTML method of naming locations within a resource. A WMLScript fragment anchor is specified by the document URL, followed by a hash mark (#), followed by a fragment identifier. WMLScript uses fragment anchors to identify individual WMLScript functions within a WMLScript

compilation unit. The syntax of the fragment anchor is specified in the following section.

URL calls and parameter passing

A user agent can make a call to an external WMLScript function by providing the following information using URLs and fragment anchors:

URL of the compilation unit. For example,
`http://www.acme.com/myScripts.scr`

Function name and parameters as the fragment anchor. For example,
`testFunc('Test%20argument', -8)`

The final URL with the fragment is:

```
http://www.acme.com/myScripts.scr#testFunc('Test%20argument', -8)
```

If the given URL denotes a valid WMLScript compilation unit then:

1. Access control checks are performed. The call fails if the caller does not have the right to call the compilation unit.

2. The function name specified in the fragment anchor is matched against the external functions in the compilation unit. The call fails if no match is found.

3. The parameter list in the fragment anchor is parsed and the given arguments with their appropriate types (string literals as string data types, integer literals as integer data types, and so on) are passed to the function. The call fails if the parameter list has an invalid syntax.

Character escaping

URL calls can use both URL escaping to specify the URL and WMLScript string escaping for any Unicode characters inside string literals. An URL is unescaped by first applying the URL escaping rules and then WMLScript string literal escaping rules for each string literal that is passed as a function parameter.

Relative URLs

WMLScript has adopted the use of relative URLs, as specified in *RFC1808*. The base URL of a WMLScript compilation unit is the URL that identifies the compilation unit.

Lexical structure

This section describes the set of elementary rules for writing programs in WMLScript.

Content types

The content types specified for the WMLScript compilation unit and its textual and binary encoding are:

Textual form: text/vnd.wap.wmlscript.

Binary form: application/vnd.wap.wmlscriptc.

Case sensitivity

WMLScript is a case-sensitive language. Therefore you should capitalize the letters correctly in all the language keywords, variables and function names.

White space and line breaks

WMLScript ignores spaces, tabs, newlines etc. that appear between tokens in programs, except those that are part of string constants.

Syntax

WhiteSpaces :

WhiteSpace
WhiteSpaces WhiteSpace

***WhiteSpace* : : :**

<TAB>
<VT>
<FF>
<SP>
<LF>
<CR>

***LineTerminator* : :**

<LF>
<CR>
<CR><LF>

Use of semicolons

The following statements in WMLScript must be followed by a semicolon:

Empty statement. For details, see “Empty statement” on page 31.

Expression statement. For details, see “Expression statement” on page 31.

Variable statement. For details, see “Variable statement” on page 32.

Break statement. For details, see “Break statement” on page 36.

Continue statement. For details, see “Continue statement” on page 36.

Return statement. For details, see “Return statement” on page 37.

Comments

WMLScript includes two comment constructions:

Line comments start with // and end at the end of the line.

Block comments consist of multiple lines starting with /* and ending with */. Note that it is a WMLScript syntax error to have nested block comments.

Syntax

Comment **: :**

MultiLineComment
SingleLineComment

MultiLineComment **: :**

/* *MultiLineCommentChars*$_{opt}$ ***/**

SingleLineComment **: :**

// *SingleLineCommentChars*$_{opt}$

Literals

WMLScript supports four types of literals: integer, floating-point, string and boolean. In addition, invalid literal is used to denote an invalid value.

Integer literals

Integer literals can be represented in three different forms: decimal, octal and hexadecimal integers.

Syntax

DecimalIntegerLiteral **: :**

`0`
NonZeroDigit DecimalDigits$_{opt}$

NonZeroDigit **: :**

One of
`1 2 3 4 5 6 7 8 9`

DecimalDigits **: :**

DecimalDigit
DecimalDigits DecimalDigit

DecimalDigit **: :**

One of
`0 1 2 3 4 5 6 7 8 9`

HexIntegerLiteral **: :**

`0x` *HexDigit*
`0X` *HexDigit*
HexIntegerLiteral HexDigit

HexDigit **: :**

One of
`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

OctalIntegerLiteral **: :**

`0` *OctalDigit*
OctalIntegerLiteral OctalDigit

OctalDigit **: :**

One of
`0 1 2 3 4 5 6 7`

The minimum and maximum sizes for integer literals and values are specified in the section “Integer size” on page 18. Note that an integer literal that is not within the specified value range results in a compile time error.

Floating-point literals

Floating-point literals can contain a decimal point as well as an exponent.

Syntax

DecimalFloatLiteral **: :**

DecimalIntegerLiteral **.** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
. *DecimalDigits ExponentPart*$_{opt}$
DecimalIntegerLiteral ExponentPart

DecimalDigits **: :**

DecimalDigit
DecimalDigits DecimalDigit

ExponentPart **: :**

ExponentIndicator SignedInteger

ExponentIndicator **: :**

One of
e E

SignedInteger **: :**

DecimalDigits
+ *DecimalDigits*
– *DecimalDigits*

The minimum and maximum sizes for floating-point literals and values are specified in the section “Floating-point size” on page 18. Note that a floating-point literal that is not within the specified value range results in a compile time error and that a floating-point literal underflow results in a floating-point literal zero.

String literals

A string is any sequence of zero or more characters enclosed by double (") or single (') quotes.

Syntax

StringLiteral **: :**

" *DoubleStringCharacters*$_{opt}$ "
' *SingleStringCharacters*$_{opt}$ '

Since some characters cannot be represented within strings, WMLScript offers special escape sequences which represent these characters:

Sequence	Character represented	Unicode	Symbol
\'	Apostrophe or single quote	\u0027	'
\"	Double quote	\u0022	"
\\	Backslash	\u005C	\
\/	Slash	\u002F	/
\b	Backspace	\u0008	
\f	Form feed	\u000C	
\n	Newline	\u000A	
\r	Carriage return	\u000D	
\t	Horizontal tab	\u0009	
\x*hh*	The character with the encoding specified by two hexadecimal digits *hh* (Latin-1 ISO8859-1)		
ooo	The character with the encoding specified by the three octal digits *ooo* (Latin-1 ISO8859-1)		
\u*hhh*	The Unicode character with the encoding specified by the four hexadecimal digits *hhhh*.		

Note that an escape sequence occurring within a string literal always contributes a character to the string value of the literal, and is never interpreted as a line terminator or as a quotation mark that might terminate the string literal.

Examples of valid strings

```
"Example"
'Specials: \x00 \' \b'
"Quote: \""
```

Boolean literals

A "truth value" in WMLScript is represented by a boolean literal. The two boolean literals are `true` and `false`.

Syntax

BooleanLiteral **::**

```
true
false
```

Invalid literal

WMLScript supports a special invalid literal to denote an invalid value.

Syntax

InvalidLiteral **::**

```
invalid
```

Identifiers

You can use identifiers to name and refer to three different elements of WMLScript: variables, functions and pragmas. Note that identifier names cannot start with a digit but can start with an underscore (_).

Syntax

Identifier **::**

IdentifierName **but not** *ReservedWord*

IdentifierName **::**

IdentifierLetter
IdentifierName IdentifierLetter
IdentifierName DecimalDigit

IdentifierLetter **::**

One of
`a b c d e f g h i j k l m n o p q r s t u v w x y z`
`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

DecimalDigit : :

One of
`0 1 2 3 4 5 6 7 8 9`

Examples of legal identifiers

```
timeOfDay
speed
quality
HOME_ADDRESS
var0
_myName
___
```

! **Note**: Uppercase and lowercase letters are distinct, which means that the identifiers `speed` and `Speed` are different.

Examples of illegal identifiers

The compiler looks for the longest string of characters that make up a valid identifier. Identifiers cannot contain any special characters except the underscore (_). WMLScript keywords and reserved words cannot be used as identifiers.

```
while
for
if
my~name
$sys
123
3pieces
take.this
```

Reserved words

WMLScript contains a set of reserved words that have special meanings and cannot be used as identifiers. Examples of such words are

```
break
continue
false
true
while
```

You can find the full list of reserved words in “WMLScript grammar” on page 103.

Name spaces

WMLScript supports name spaces for identifiers that are used for different purposes. The following name spaces are supported:

Function names. For details, see “Functions” on page 27.

Function parameters and variables. For details, see “Functions” on page 27 and “Variables and data types” on page 16.

Pragmas. For details, see “Pragmas” on page 38.

Note that you can use the same identifiers to specify a function name, variable/parameter name or a name for a pragma within the same compilation unit:

```
use url myTest "http://www.acme.com/script";

function myTest(myTest) {
   var value = myTest#myTest(myTest);
   return value;
};
```

Variables and data types

This section discusses two important concepts of WMLScript language: variables and internal data types. A variable is a name associated with a data value. You can use variables to store and manipulate program data. WMLScript supports local variables only when declared inside functions or passed as function parameters.

Variable declaration

Variable declaration is compulsory in WMLScript. It is done simply by using the *var* keyword and a variable name. Variable names follow the syntax defined for all the identifiers above in the section “Identifiers” on page 14. For example, the following are legal variable declarations:

```
var x;
var price;
var x,y;
var size = 3;
```

Note that variables must be declared before you can use them. Initialization of variables is optional. Uninitialized variables are automatically initialized to contain an empty string ("").

Variable scope and lifetime

The scope of a WMLScript variable is the remainder of the function in which it has been declared. Note that all variable names within a function must be unique and that block statements are not used for scoping.

```
function priceCheck(givenPrice) {
   if (givenPrice > 100) {
      var newPrice = givenPrice;
   }  else  {
      newPrice = 100;
   };
   return newPrice;
};
```

The lifetime of a variable is the time between when the variable is declared and when the function ends.

```
function foo() {
   x = 1;         // Error: usage before declaration
   var x,y;
   if (x) {
      var (y);    // Error: redeclaration
   };
};
```

Variable access

Variables are accessible only within the function in which they have been declared. Accessing the content of a variable is done by using the variable's name:

```
var myAge   = 37;
var yourAge = 63;
var ourAge  = myAge + yourAge;
```

Variable type

WMLScript is a weakly typed language: the variables are not typed. Internally, the following basic data types are supported: *boolean*, *integer*, *floating-point* and *string*. In addition to these, a fifth data type *invalid* is specified to be used in cases where an invalid data type is needed to distinguish it from the other internal data types. Since these data types are supported only internally, you do not have to specify variable types, and any variable can contain any type of data at any given time. WMLScript will automatically try to convert between the different types as needed.

```
var flag         = true;     // Boolean
var number       = 12;       // Integer
var temperature  = 37.7;     // Float
number           = "XII";    // String
var except       = invalid;  // Invalid
```

L-values

Some operators require that the left operand be a reference to a variable (L-value) and not the variable value. Thus, in addition to the five data types supported by WMLScript, a sixth type *variable* is used to specify that a variable name must be provided.

```
result += 111;    // += operator requires a variable
```

Type equivalency

WMLScript supports operations on different data types. All operators specify the accepted data types for their operands. Automatic data type conversions are used to convert operand values to the required data types.

For detailed information on operators, see "Operators and expressions" on page 19. For detailed information on data type conversions, see "Automatic data type conversion rules" on page 43.

Numeric values

WMLScript supports two different numeric variable values: *integer* and *floating-point*. Note that in cases where the value can be either an integer or a floating-point, a more generic term *number* is used instead.

You can initialize variables with integer and floating-point literals, and you can use several operators to modify the variable values during the runtime. Conversion rules between integer and floating-point values are specified in the chapter "Automatic data type conversion rules" on page 43.

```
var pi      = 3.14;
var length  = 0;
var radius  = 2.5;
length      = 2*pi*radius
```

Integer size

The size of the integer is 32 bits (complement of two), meaning that the value range supported for integer values goes from –2147483648 to 2147483647. You can get these values during the runtime by using the following *Lang* library functions.

`Lang.maxInt ()`	Maximum representable integer value.
`Lang.minInt ()`	Minimum representable integer value.

Floating-point size

WMLScript supports 32-bit single precision floating-point format:

Maximum value: 3.40282347E+38.

Minimum positive non-zero value: 1.17549435E–38 or smaller (the normalized precision at least must be supported).

You can get these values during the runtime by using the following *Float* library functions:

`Float.maxFloat ()`	Maximum representable floating-point value supported.
`Float.minFloat ()`	Smallest positive non-zero floating-point value supported.

The special floating-point number types are handled by using the following rules:

If an operation results in a floating-point number that is not a finite real number (not a number, positive infinity etc.) supported by the single precision floating-point format, then the result is an `invalid` value.

If an operation results in a floating-point *underflow* the result is zero (`0.0`).

Negative and *positive zero* are equal and undistinguishable.

String values

WMLScript supports *strings* that can contain letters, digits, special character, and so on. You can initialize variables with string literals, and you can manipulate string values with WMLScript operators and functions specified in the standard *String* library described on page 73 of this guide.

```
var msg = "Hello";
var len = String.length(msg);
msg     = msg + 'World!';
```

Boolean values

You can use *Boolean* values to initialize or assign a value to a variable or in statements which require a boolean value as one of the parameters. A Boolean value can be a literal or the result of a logical expression.

```
var truth = true;
var lie   = !truth;
```

Operators and expressions

The following sections describe the operators supported by WMLScript and how they can be used to form complex expressions.

Assignment operators

WMLScript offers several ways to assign a value to a variable. The simplest is the regular assignment (=), but assignments with operations are also supported:

Operator	Operation
=	assign
+=	add (numbers)/concatenate (strings) and assign
-=	subtract and assign
*=	multiply and assign
/=	divide and assign
div=	divide (integer division) and assign
%=	remainder (the sign of the result equals the sign of the dividend) and assign
<<=	bitwise left shift and assign
>>=	bitwise right shift with sign and assign
>>>=	bitwise right shift zero fill and assign
&=	bitwise AND and assign
^=	bitwise XOR and assign
|=	bitwise OR assign

Note that assignment does not necessarily imply sharing of structure, nor does assignment of one variable change the binding of any other variable.

```
var a = "abc";
var b = a;
b     = "def";    // Value of a is "abc"
```

Arithmetical operators

WMLScript supports all the basic binary arithmetical operations:

Operator	Operation
+	add (numbers)/concatenation (strings)
-	subtract
*	multiply

Operator	Operation
/	divide
div	integer division

In addition to these, the following set of more complex binary operations are supported:

Operator	Operation
%	remainder, the sign of the result equals the sign of the dividend
<<	bitwise left shift
>>	bitwise right shift with sign
>>>	bitwise right shift with zero fill
&	bitwise AND
|	bitwise OR
^	bitwise XOR

The basic unary operations are:

Operator	Operation
+	plus
-	minus
--	pre-or-post decrement
++	pre-or-post increment
~	bitwise NOT

Examples of arithmetical operators:

```
var y = 1/3;
var x = y*3+(++b);
```

Logical operators

WMLScript supports the following basic logical operations:

Operator	Operation
`&&`	logical AND
`||`	logical OR
`!`	logical NOT (unary)

The logical AND operator evaluates the first operand and tests the result:

If the result is `false`, the result of the operation is `false` and the second operand is not evaluated.

If the first operand evaluates to `true`, the result of the operation is the result of the evaluation of the second operand.

If the first operand evaluates to `invalid`, the second operand is not evaluated and the result of the operation is `invalid`.

Similarly, the logical OR evaluates the first operand and tests the result:

If the result is `true`, the result of the operation is `true` and the second operand is not evaluated.

If the first operand evaluates to `false`, the result of the operation is the result of the evaluation of the second operand.

If the first operand evaluates to `invalid`, the second operand is not evaluated and the result of the operation is `invalid`.

```
weAgree = (iAmRight && youAreRight) ||
          (!iAmRight && !youAreRight);
```

WMLScript requires a boolean value for logical operations. Therefore, automatic conversions from other types to boolean type and vice versa are performed.

Note that if the value of the first operand AND or OR is `invalid`, the second operand is not evaluated and the result of the operand is `invalid`:

```
var a = (1/0) || foo();    // result: invalid, no call to foo()
var b = true  || (1/0);    // true
var c = false || (1/0);    // invalid
```

String operators

WMLScript supports string concatenation as a built-in operation. The + and += operators used with strings perform a concatenation on the strings. Other string operations are supported by the standard *String* library, described on page 73 of this guide.

```
var str = "Beginning" + "End";
var chr = String.charAt(str,10);    // chr = "E"
```

Comparison operators

WMLScript supports all the basic comparison operations:

Operator	Operation
<	less than
<=	less than or equal
==	equal
>=	greater than or equal
>	greater than
!=	inequality

Comparison operators use the following rules:

Boolean: `true` is larger than `false`.

Integer: Comparison is based on the given integer values.

Floating-point: Comparison is based on the given floating-point values.

String: Comparison is based on the order of character codes of the givenstring values. Character codes are defined by the character set supported by the WMLScript Interpreter.

Invalid: If at least one of the operands is `invalid` then the result of the comparison is `invalid`.

Examples of comparison operators:

```
var res = (myAmount > yourAmount);
var val = ((1/0) == invalid);          // val = invalid
```

Array operators

WMLScript does not support arrays as such. However, the standard *String* library supports functions by which array-like behaviour can be implemented using strings. A string can contain elements separated by a specified separator. For this purpose, the *String* library contains functions which allow you to create and manage string arrays. The following is an example of an array operator:

```
function dummy() {
   var str  = "Mary had a little lamb";
   var word = String.elementAt(str,4,"");
};
```

Comma operator

WMLScript supports the comma (`,`) operator which allows you to combine multiple evaluations in one expression. The result of the comma operator is the value of the second operand:

```
for (a=1, b=100; a < 10; a++,b++) {
   … other functions …
};
```

Note that the commas used in the function call to separate parameters, and in the variable declarations to separate multiple variable declarations, and are therefore **not** comma operators. In these cases, the comma operator must be placed inside the parenthesis:

```
var a=2;
var b =3, c=(a,3);
myFunction("Name", 3*(b*a,c));   // Two parameters: "Name",9
```

Conditional operator

WML supports the conditional (`?:`) operator which takes three operands:

The operator selectively evaluates one of the given two operands based on the boolean value of the first operand.

If the value of the first operand (condition) is `true` then the result of the operation is the result of the evaluation of the second operand.

If the value of the first operand is `false` or `invalid` then the result of the operation is the result of the evaluation of the third operand.

The following is an example of a conditional operator:

```
myResult = flag ? "Off" : "On (value=" + level + ")";
```

Note that the conditional operator behaves like an *if* statement. The third operand is evaluated if the evaluation of the condition results in `false` or `invalid`.

typeof operator

As stated before, although WMLScript is a weakly typed language, internally the following basic data types are supported: *boolean*, *integer*, *floating-point*, *string* and *invalid*. The *typeof* operator returns an integer value that describes the type of the given expression. The possible results are:

Type	Code
Integer	0
Floating-point	1

Type	Code
String	2
Boolean	3
Invalid	4

The *typeof* operator does not try to convert the result from one type to another, but returns the type as it is after the evaluation of the expression.

```
var str    = "123";
var myType = typeof str;   // myType = 2
```

isvalid operator

You can use this operator to check the type of the given expression. It returns the boolean value `false` if the type of the expression is invalid; otherwise `true` is returned. The *isvalid* operator does not try to convert the result from one type to another, but returns the type as it is after the evaluation of the expression.

```
var str = "123";
var ok  = isvalid str;     // true
var tst = isvalid (1/0);   // false
```

Expressions

WMLScript supports most of the expressions supported by other programming languages. The simplest expressions are constants and variable names, which simply evaluate to either the value of the constant or the variable.

```
567
66.77
"This is too simple"
'This works too'
true
myAccount
```

You can define more complex expressions by using simple expressions with operators and function calls.

```
myAccount + 3
(a + b)/3
initialValue + nextValue(myValues);
```

Expression bindings

The following table contains all the operators supported by WMLScript. It also contains information on operator precedence and the operator associativity (left-to-right (L) or right-to-left (R)):

Precedence	Associativity	Operator	Operand types	Result type	Operation performed
1	R	++	number	number*	pre- or post-increment (unary)
1	R	--	number	number*	pre- or post-decrement (unary)
1	R	+	number	number*	unary plus
1	R	-	number	number*	unary minus (negation)
1	R	~	integer	integer*	bitwise NOT (unary)
1	R	!	boolean	boolean*	logical NOT (unary)
1	R	typeof	any	integer	return internal data type
1	R	isvalid	any	boolean	check for validity (unary)
2	L	*	numbers	number*	multiplication
2	L	/	numbers	floating-point*	division
2	L	div	integers	integer*	integer division
2	L	%	integers	integer*	remainder
3	L	-	numbers	number*	subtraction
3	L	+	numbers or strings	number or string*	addition (numbers) or string concatenation
4	L	<<	integers	integer*	bitwise left shift
4	L	>>	integers	integer*	bitwise right shift with sign
4	L	>>>	integers	integer*	bitwise right shift with zero fill
5	L	<, <=	numbers or strings	boolean*	less than, less than or equal
5	L	>, >=	numbers or strings	boolean*	greater than, greater than or equal
6	L	==	numbers or strings	boolean*	equal (identical values)
6	L	!=	numbers or strings	boolean*	not equal (different values)
7	L	&	integers	integer*	bitwise AND
8	L	^	integers	integer*	bitwise XOR
9	L	|	integers	integer*	bitwise OR
10	L	&&	booleans	boolean*	logical AND

Precedence	Associativity	Operator	Operand types	Result type	Operation performed
11	L	||	booleans	boolean*	logical OR
12	R	?:	boolean, any	any*	conditional expression
13	R	=	variable, any	any*	assignment
13	R	*=, -=	variable, number	number*	assignment with numeric operation
13	R	/=	variable, number	floating-point*	assignment with numeric operation
13	R	%=, div=	variable, integer	integer*	assignment with integer operation
13	R	+=	variable, number or string	number or string*	assignment with addition or concatenation
13	R	<<=, >>=, >>>=, &=, ^=, |=	variable, integer	integer*	assignment with bitwise operation
14	L	,	any	any	multiple evaluation

* The operator can return an `invalid` value if a data type conversion fails or one of the operands is `invalid`.

Functions

A WMLScript function is a named part of the WMLScript compilation unit that you can call to perform a specific set of statements and to return a value. The following sections describe how you can declare and use WMLScript functions.

Declaration

You can use the function declaration to declare a WMLScript function name (*Identifier*) with the optional parameters (*FormalParameterList*) and a block statement that is executed when the function is called. All functions have the following characteristics:

Function declarations cannot be nested.

Function names must be unique within one compilation unit.

All parameters to functions are passed by value.

Function calls must pass exactly the same number of arguments to the called function as specified in the function declaration.

Function parameters behave like local variables that have been initialized before the function body (block of statements) is executed.

A function always returns a value. By default, it is an empty string (""). However, you can use a return statement to specify other return values.

Note that the functions in WMLScript are not data types but a syntactical feature of the language.

Syntax

FunctionDeclaration **:**

`extern`$_{opt}$ **`function`** *Identifier (FormalParameterList*$_{opt}$ *) Block* **`;`**$_{opt}$

FormalParameterList **:**

Identifier
FormalParameterList **`,`** *Identifier*

You can use the optional `extern` keyword to specify that a function is to be externally accessible. External functions can be called from outside the compilation unit in which they are defined.

Identifier is the name specified for the function.

The optional *FormalParameterList* is a list of argument names separated by commas.

Block is the body of the function that is executed when the function is called and the parameters have been initialized by the passed arguments.

Examples of function declaration

```
function currencyConverter (currency, exchangeRate) {
   return currency*exchangeRate;
};

extern function testIt() {
   var USD = 10;
   var FIM = currencyConverter(USD, 5.3);
};
```

Function calls

The way you call a function depends on where the called (target) function is declared. The following sections describe the three function calls available in

WMLScript: local script function call, external function call and library function call.

Local script functions

Local script functions are defined inside the same compilation unit. You can call them simply by providing the function name and a list of arguments separated by commas. Note that the number of arguments must match the number of parameters accepted by the function.

Syntax

LocalScriptFunctionCall **:**

FunctionName Arguments

FunctionName **:**

Identifier

Arguments **:**

( )
(*ArgumentList* **)**

ArgumentList **:**

AssignmentExpression
ArgumentList **,** *AssignmentExpression*

Note that you can call functions inside the same compilation unit even before the function has been declared:

```
function test2(param) {
   return test1(param+1);
};

function test1(val) {
   return val*val;
};
```

External functions

External function calls must be used when the called function is declared in an external compilation unit. The external function call is similar to a local function call but it must be prefixed with the name of the external compilation unit.

Syntax

ExternalScriptFunctionCall **:**

ExternalScriptName **#** *FunctionName Arguments*

ExternalScriptName **:**

Identifier

The pragma `use url` is used to specify the external compilation unit. It defines the mapping between the external unit and a name that you can use within function declarations. This name and the hash mark (#) are used to prefix the standard function call syntax:

```
use url OtherScript "http://www.acme.com/script";

function test3(param) {
   return OtherScript#test2(param+1);
};
```

Library functions

Library function calls must be used when the called function is a WMLScript standard library function. For more information on the standard libraries, see “WMLScript standard libraries” on page 61.

Syntax

LibraryFunctionCall **:**

LibraryName **.** *FunctionName Arguments*

LibraryName **:**

Identifier

You can call a library function by prefixing the function name with the name of the library and the period character (.):

```
function test4(param) {
   return Float.sqrt(Lang.abs(param)+1);
};
```

Default return value

The default return value for a function is an empty string (""). Return values of functions can be ignored, that is, the function call can be a statement:

```
function test5() {
   test4(4);
};
```

Statements

WMLScript statements consist of expressions and keywords used with the appropriate syntax. A single statement can span multiple lines, and multiple statements can occur on a single line.

The following sections describe the statements available in WMLScript: empty statement, expression statement, block statement, break, continue, for, if...else, return, var, while.

Empty statement

Empty statement is a statement used in cases where a statement is needed but no operation is required.

Syntax

EmptyStatement **:**

`;`

Example of empty statement

```
while (!poll(device)) ; // Wait until poll() is true
```

Expression statement

Expression statements are used to assign values to variables, calculate mathematical expressions, make function calls, and so on.

Syntax

ExpressionStatement **:**

Expression `;`

Expression **:**

AssignmentExpression
Expression **,** *AssignmentExpression*

Examples of expression statement

```
str  = "Hey " + yourName;
val3 = prevVal + 4;
counter++;
myValue1 = counter, myValue2 = val3;
alert("Watch out!");
retVal = 16*Lang.max(val3,counter);
```

Block statement

A block statement is a set of statements enclosed in brackets. You can use it anywhere a single statement is needed.

Syntax

Block **:**

{ *StatementList*$_{opt}$ **}**

StatementList **:**

Statement
StatementList Statement

Example of block statement

```
{
   var i = 0;
   var x = Lang.abs(b);
   popUp("Remember!");
}
```

Variable statement

The variable statement declares variables with an initialization. The initialization is optional: variables are initialized to an empty string ("") by default. The scope of the declared variable is the rest of the current function.

Syntax

VariableStatement **:**

var *VariableDeclarationList* **;**

VariableDeclarationList **:**

VariableDeclaration
VariableDeclarationList **,** *VariableDeclaration*

VariableDeclaration **:**

Identifier VariableInitializer$_{opt}$

VariableInitializer **:**

= *ConditionalExpression*

Identifier is the variable name and can be any legal identifier.

ConditionalExpression is the initial value of the variable and can be any legal expression. This expression, or the default initialization to an empty string is evaluated every time the variable statement is executed.

! **Note**: Variable names must be unique within a single function.

Examples of variable statements

```
function count(str)  {
   var result = 0;         // Initialized once
   while (str != "") {
      var ind = 0;         // Initialized every time
      // modify string
   };
   return result
};

function example(param) {
   var a = 0;
   if (param > a) {
      var b = a+1;         // Variables a and b can be used
   }  else  {
      var c = a+2;         // Variables a, b and c can be used
   };
   return a;               // Variables a, b and c are accessible
};
```

If statement

The *if* statement specifies conditional execution of statements. It consists of a condition and one or two statements, and executes the first statement if the specified condition is true. If the condition is false, the second (optional) statement is executed.

Syntax

IfStatement **:**

`if (` *Expression* **`)`** *Statement* **`else`** *Statement*
`if (` *Expression* **`)`** *Statement*

Expression (condition) can be any WMLScript expression that evaluates directly or after conversion to a boolean or invalid value.

— If the condition evaluates to `true`, the first statement is executed.

— If the condition evaluates to `false` or `invalid`, the second (optional) `else` statement is executed.

Statement can be any WMLScript statement, including another (nested) `if` statement. Note that `else` is always tied to the closest `if`.

Example of if statement

```
if (sunShines) {
   myDay = "Good";
   goodDays++;
}  else
   myDay = "Oh well...";
```

While statement

The *while* statement creates a loop that evaluates an expression and, if it is `true`, executes a statement. The loop repeats as long as the specified condition is `true`.

Syntax

WhileStatement **:**

`while (` *Expression* **`)`** *Statement*

Expression (condition) can be any WMLScript expression that evaluates directly or after conversion to a boolean or invalid value. The condition is evaluated before each execution of the loop statement.

— If the condition evaluates to `true`, the *Statement* is performed.

— If the condition evaluates to `false` or `invalid`, execution continues with the statement following *Statement*.

Statement is executed as long as the condition evaluates to `true`.

Example of while statement

```
var counter = 0;
var total   = 0;
while (counter < 3) {
   counter++;
   total += c;
};
```

For statement

You can use the *for* statement to create loops. It consists of three optional expressions enclosed in parentheses and separated by semicolons followed by a statement executed in the loop.

Syntax

ForStatement **:**

for (*Expression*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*
for (var *VariableDeclarationList* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

The first *Expression* or *VariableDeclarationList* (initializer) is typically used to initialize a counter variable. This expression may optionally declare new variables with the *var* keyword. The scope of the defined variables is the rest of the function.

The second *Expression* (condition) can be any WMLScript expression that evaluates directly or after conversion to a boolean or invalid value. The condition is evaluated on each pass through the loop. If the condition evaluates to `true`, the *Statement* is performed. This conditional test is optional. If omitted, the condition always evaluates to `true`.

The third *Expression* (increment expression) is generally used to update or increment the counter variable. *Statement* is executed as long as the condition evaluates to `true`.

Example of for statement

```
for (var index = 0; index < 100; index++) {
   count += index;
   myFunc(count);
};
```

Break statement

The *break* statement terminates the current *while* or *for* loop and continues the program execution from the statement following the terminated loop. Note that it is a WMLScript syntax error to use the *break* statement outside a *while* or a *for* statement.

Syntax

BreakStatement **:**

```
break ;
```

Example of break statement

```
function testBreak(x) {
   var index = 0;
   while (index < 6) {
      if (index == 3) break;
      index++;
   };
   return index*x;
};
```

Continue statement

The *continue* statement terminates execution of a block of statements in a *while* or *for* loop and continues execution of the loop with the next iteration. Note that the *continue* statement does not terminate the execution of the loop:

In a *while* loop, it jumps back to the condition.

In a *for* loop, it jumps to the update expression.

Note that it is a WMLScript syntax error to use the *continue* statement outside a *while* or a *for* statement.

Syntax

ContinueStatement **:**

```
continue;
```

Example of continue statement

```
var index = 0;
var count = 0;
```

```
while (index < 5) {
   index++;
   if (index == 3)
      continue;
   count += index;
};
```

Return statement

You can use the *return* statement inside the function body to specify the function return value. If no *return* statement is specified or none of the function return statements are executed, the function returns an empty string by default.

Syntax

ReturnStatement **:**

`return` *Expression*$_{opt}$**`;`**

Example of return statement

```
function square( x ) {
   if (!(Lang.isFloat(x))) return invalid;
   return x * x;
};
```

Libraries

WMLScript supports the use of libraries. Libraries are named collections of functions that belong together logically. You can call these functions by using a dot (`.`) separator between the library name and the function name with parameters.

The following example illustrates a library function call:

```
function dummy(str) {
   var i = String.elementAt(str,3," ");
};
```

Standard libraries

The WMLScript standard libraries are described in more detail in “WMLScript standard libraries” on page 61.

Pragmas

WMLScript supports the use of pragmas that specify compilation unit level information. Pragmas are specified at the beginning of the compilation unit before any function declaration. Note that all pragmas start with the keyword `use` and are followed by pragma-specific attributes.

Syntax

CompilationUnit **:**

Pragmas$_{opt}$ *FunctionDeclarations*

Pragmas **:**

Pragma
Pragmas Pragma

Pragma **:**

use *PragmaDeclaration* **;**

PragmaDeclaration **:**

ExternalCompilationUnitPragma
AccessControlPragma
MetaPragma

The following sections contain more information on the pragmas supported.

External compilation units

You can access WMLScript compilation units by using an URL. Thus, you can access each WMLScript function by specifying the URL of the WMLScript resource and its name. Note that the `use url` pragma must be used when calling a function in an external compilation unit.

Syntax

ExternalCompilationUnitPragma **:**

url *Identifier StringLiteral*

The `use url` pragma specifies the location (URL) of the external WMLScript resource and gives it a local *name*. You can then use this name inside the function declarations to make external function calls.

```
use url OtherScript "http://www.acme.com/app/script";

function test(par1, par2) {
   return OtherScript#check(par1-par2);
};
```

The behavior of this example is the following:

1. The pragma specifies an URL for a WMLScript compilation unit.
2. The function call loads the compilation unit by using the given URL: `http://www.acme.com/app/script`.
3. The content of the compilation unit is verified and the specified function `check` is executed.

The `use url` pragma has its own name space for local names. However, the local names must be unique within one compilation unit. The following URLs are supported:

Uniform Resource Locators without a hash mark (#) or a fragment identifier. The schemes supported are specified in the Wireless Application Environment Specification. For detailed information on URLs, refer to *RFC2396*.

Relative URLs without a hash mark (#) or a fragment identifier. The base URL is the one identifying the current compilation unit. For detailed information on relative URLs, refer to *RFC2396*.

The given URL is escaped according to the URL escaping rules. No compile time automatic escaping, URL syntax or URL validity checking is performed.

Access control

A WMLScript compilation unit can protect its content by using an access control pragma. Access control is performed before any external functions are called. Note that it is a WMLScript syntax error for a compilation unit to contain more than one access control pragma.

Syntax

AccessControlPragma **:**

`access` *AccessControlSpecifier*

AccessControlSpecifier **:**

`domain` *StringLiteral*
`path` *StringLiteral*
`domain` *StringLiteral path StringLiteral*

Every time an external function is invoked, an access control check is performed to determine whether the destination compilation unit allows access from the caller. The access control pragma specifies the *domain* and *path* attributes against which the access control checks are performed. If a compilation unit has a domain and/or path attribute, the referring compilation unit's URL must match the values of the attributes. Matching is done as follows: the access domain is suffix-matched against the domain name portion of the referring URL, and the access path is prefix-matched against the path portion of the referring URL. Domain and path attributes follow the URL capitalization rules.

Domain suffix matching is done using the entire element of each sub-domain and must match each element exactly. For example, `www.acmecomp.com` matches `acmecomp.com`, but does not match `comp.com`.

Path prefix matching is done using entire path elements and each element must match exactly. For example, `/X/Y` matches `/X`, but does not match `/XZ`.

The domain attribute defaults to the current compilation unit's domain. The path attribute defaults to the value "`/`".

To simplify the development of applications that may not know the absolute path to the current compilation unit, the path attribute accepts relative URLs. The user agent converts the relative path to an absolute path and then performs prefix matching against the path attribute.

By default, access control is disabled.

Example of access control

Given the following access control attributes for a compilation unit:

```
use access domain "acme.com" path "/docs";
```

The following referring URLs would be allowed to call the external functions specified in this compilation unit:

```
acme.com/docs/technical.cgi
www.acme.com/docs/marketing.cgi
www.acme.com/docs/demos/packages.cgi?x=123&y=456
```

The following referring URLs would not be allowed to call the external functions:

```
www.test.net/docs
www.acme.com/internal/foo.wml
```

Meta information

You can use also pragmas to specify compilation unit-specific meta information. Meta information is specified with property names and values. Note that user agents are not required for the meta data to be acted upon.

Syntax

MetaPragma **:**

`meta` *MetaSpecifier*

MetaSpecifier **:**

MetaName
MetaHttpEquiv
MetaUserAgent

MetaName **:**

`name` *MetaBody*

MetaHttpEquiv **:**

`http equiv` *MetaBody*

MetaUserAgent **:**

`user agent` *MetaBody*

MetaBody **:**

MetaPropertyName MetaContent $MetaScheme_{opt}$

Meta pragmas have three attributes:

Property name specifies a name for the meta element.

Content specifies the value of the property.

The optional *scheme* specifies a form or structure that may be used to interpret the property value. The values vary depending on the type of meta data.

The attribute values are string literals.

Name

The name meta pragma specifies meta information intended for use by the web servers. The user agent ignores any meta data named with this attribute.

```
use meta name "Created" "18-March-1998";
```

HTTP equiv

The HTTP equiv meta pragma specifies meta information indicating that the property should be interpreted as an HTTP header. Meta data named with this attribute is converted to a WSP or HTTP response header if the compilation unit is compiled before it arrives at the user agent.

```
use meta http equiv "Keywords" "Script,Language";
```

User agent

The user agent meta pragma specifies meta information intended for use by the user agents.

```
use meta user agent "Type" "Test";
```

Automatic data type conversion rules

In some cases, WMLScript operators require specific data types as their operands. WMLScript supports automatic data type conversions to meet the requirements of these operators. The following sections describe the different conversions in detail.

General conversion rules

As stated previously, WMLScript is a weakly typed language, and the variable declarations do not specify a type. However, internally the language handles the following data types:

Boolean: represents a boolean value true or false.

Integer: represents an integer value.

Floating-point: represents a floating-point value.

String: represents a sequence of characters.

Invalid: represents a type with a single value `invalid`.

A variable can contain one of these types of value. WMLScript provides an operator *typeof*, which can be used to determine the current type of a variable or any expression. The typeof operator performs no conversions.

Each WMLScript operator accepts a predefined set of operand types. If the operands provided are not of the right data type, an automatic conversion takes place. The following sections discuss the legal automatic conversions between two data types.

! **Note**: Data type conversions may reduce precision.

Conversions to string

Legal conversions of other data types to string are:

An integer value is converted to a string of decimal digits that follows the numeric string grammar rules for decimal integer literals. For more information, see “Numeric string grammar” on page 104.

A floating-point value is converted to a string representation that follows the numeric string grammar rules for decimal floating-point literals. The resulting string representation is equal to the original value. For example, `.5` can be represented as `0.5` or `.5e0`.

The boolean value `true` is converted to the string `"true"` and the value `false` to the string `"false"`.

`Invalid` cannot be converted to a string value.

Conversions to integer

Legal conversions of other data types to integer are:

A string can be converted into an integer value only if it contains a decimal representation of an integer number. For more information, see “Numeric string grammar” on page 104.

A floating-point value cannot be converted to an integer value.

The boolean value `true` is converted to the integer value 1, `false` to 0.

`Invalid` cannot be converted to an integer value.

Conversions to floating-point

Legal conversions of other data types to floating-point are:

A string can be converted into a floating-point value only if it contains a valid representation of a floating-point number. For more information, see “Numeric string grammar” on page 104.

An integer value is converted to a corresponding floating-point value.

The boolean value `true` is converted to a floating-point value 1.0, `false` to 0.0.

`Invalid` cannot be converted to a floating-point value.

The conversions between a string and a floating-point value must be transitive according to the ability of the data types to accurately represent the value. Note that a conversion can reduce precision.

Conversions to boolean

Legal conversions of other data types to boolean are:

The empty string ("") is converted to `false`. All other strings are converted to `true`.

The integer value `0` is converted to `false`. All other integer numbers are converted to `true`.

The floating-point value `0.0` is converted to `false`. All other floating-point numbers are converted to `true`.

`Invalid` cannot be converted to a boolean value.

Conversions to invalid

There are no legal conversion rules for converting any of the other data types to an invalid type. `Invalid` is either a result of an operation error or a literal value. In most cases, an operator that has an `invalid` value as an operand evaluates to `invalid`. For the exceptions to this rule, see “Conditional operator” on page 24, “typeof operator” on page 24 and “isvalid operator” on page 25.

Summary

The following table summarizes the legal conversions between data types.

Given/Used as	Boolean	Integer	Floating-point	String
Boolean true	–	1	1.0	"true"
Boolean false	–	0	0.0	"false"
Integer 0	false	–	0.0	"0"
Any other integer	true	–	floating-point value of number	string representation of a decimal integer
Floating-point 0.0	false	Illegal	–	implementation-dependent string representation of a floating-point value, e.g. "0.0"
Any other floating-point	true	Illegal	–	implementation-dependent string representation of a floating-point value
Empty string	false	Illegal	Illegal	–
Non-empty string	true	integer value of its string representation or illegal	floating-point value of its string representation or illegal	–
Invalid	Illegal	Illegal	Illegal	Illegal

Operator data type conversion rules

The conversion rules just discussed specified when a legal conversion is possible between two data types. WMLScript operators use these rules, the operand data type and values to select the operation to be performed (in cases where the type is used to specify the operation), and to perform the data type conversions needed for the selected operation. The rules are specified as follows:

The additional conversion rules are specified in steps. Each step is performed in the given order until the operation and the data types for its operands are specified and the return value defined.

If the type of the operand value matches the required type, the value is used as such.

If the operand value does not match the required type, a conversion from the current data type to the one required is attempted:

— *Legal conversion*: A conversion can be done only if the general conversion rules specify a legal conversion from the current operator data type to the one required.

— *Illegal conversion*: Conversion cannot be done if the general conversion rules do not specify a legal conversion from the current type to the type required.

If a legal conversion rule is specified for the operand (unary) or for all the operands then the conversion is performed, the operation is performed on the converted values, and the result returned is the value of the operation. If a legal conversion results in an `invalid` value, then the operation returns an `invalid` value.

If no legal conversion is specified for one or more of the operands, no conversion is performed, and the next step in the additional conversion rules is performed.

The following table contains the operator data type conversion rules based on the operand data types given.

Operand types	Additional conversion rules	Examples

Boolean(s)	If the operand is boolean or can be converted into a boolean value, then perform a boolean operation and return its value, otherwise return `invalid`.	`true  3.4 => boolean`
		`1 && 0 => boolean`
		`"A" || "" => boolean`
		`!42 => boolean`
		`!invalid => invalid`
		`3 && invalid => invalid`

Integer(s)	If the operand is an integer or can be converted into an integer value, then perform an integer operation and return its value, otherwise return `invalid`.	`"7" << 2 => integer`
		`true << 2 => integer`
		`7.2 >> 3 => invalid`
		`2.1 div 4 => invalid`

Operand types	Additional conversion rules	Examples
Floating-point(s)	If the operand is floating-point or can be converted into a floating-point value, then perform a floating-point operation and return its value, otherwise return `invalid`.	–
String(s)	If the operand is a string or can be converted into a string value, then perform a string operation and return its value, otherwise return `invalid`.	–

Integer or floating-point (unary)	If the operand is an integer or can be converted into an integer value, then perform an integer operation and return its value.
	
	If the operand is a floating-point or can be converted into a floating-point value, then perform a floating-point operation and return its value, otherwise return `invalid`.
		`+10 => integer`
		`-10.3 => float`
		`-"33" => integer`
		`+"47.3" => float`
		`+true => integer 1`
		`-false => integer 0`
		`-"ABC" => invalid`
		`-"9e9999" => invalid`

Integers or floating-points	If at least one of the operands is a floating-point, then convert the remaining operand into a floating-point value, perform a floating-point operation, and return its value.
	
	If the operands are integers or can be converted into integer values, then perform an integer operation and return its value.
	
	If the operands can be converted into floating-point values then perform a floating-point operation and return its value, otherwise return `invalid`.
		`100/10.3 => float`
		`33*44 => integer`
		`"10"*3 => integer`
		`3.4*"3.4" => float`
		`"10"-"2" => integer`
		`"2.3"*"3" => float`
		`3.2*"A" => invalid`
		`.9*"9e999" => invalid`
		`invalid*1 => invalid`

Integers, floating-points or strings	If at least one of the operands is a string then convert the remaining operand into a string value, perform a string operation, and return its value.
	
	If at least one of the operands is a floating-point, then convert the remaining operand into a floating-point value, perform a floating-point operation, and return its value.
	
	If the operands are integers or can be converted into integer values, then perform an integer operation and return its value, otherwise return `invalid`.
		`12+3 => integer`
		`32.4+65 => float`
		`"12"+5.4 => string`
		`43.2<77 => float`
		`"Hey"<56 => string`
		`2.7+"4.2" => string`
		`9.9+true => float`
		`3<false => integer`
		`"A"+invalid => invalid`

Any	Any type is accepted.	`a = 37.3 => float` `b = typeof "s" => string`

Summary of operators and conversions

The following sections summarize how the conversion rules are applied to WMLScript operators, and what their possible return value types are.

Single-typed operators

Operators that accept operands of one specific type use the general conversion rules directly. The following lists all the single-typed WMLScript operators.

Operator	Operand types	Result type	Operation performed
!	boolean	boolean	logical NOT (unary)
&&	booleans	boolean	logical AND
||	booleans	boolean	logical OR
~	integer	integer	bitwise NOT (unary)
<<	integers	integer	bitwise left shift
>>	integers	integer	bitwise right shift with sign
>>>	integers	integer	bitwise right shift with zero fill
&	integers	integer	bitwise AND
^	integers	integer	bitwise XOR
|	integers	integer	bitwise OR
%	integers	integer	remainder
div	integers	integer	integer division
<<=, >>=, >>>=, &=, ^=, |=	first operand: variable second operand: integer	integer	assignment with bitwise operation
%=, div=	first operand: variable second operand: integer	integer	assignment with numeric operation

! **Note**: All operators may also have an invalid result type.

Multi-typed operators

The following lists the operators that accept multi-typed operands.

Operator	Operand types	Result type	Operation performed
++	integer or floating-point	integer/ floating-point	pre- or post-increment (unary)
--	integer or floating-point	integer/ floating-point	pre- or post-decrement (unary)
+	integer or floating-point	integer/ floating-point	unary plus
-	integer or floating-point	integer/ floating-point	unary minus (negation)
*	integers or floating-points	integer/ floating-point	multiplication
/	integers or floating points	floating-point	division
-	integers or floating points	integer/ floating-point	subtraction
+	integers, floating points or strings	integer/floating-point/string	addition or string concatenation
<. <=	integers, floating points or strings	boolean	less than, less than or equal
>, >=	integers, floating points or strings	boolean	greater than, greater or equal
==	integers, floating points or strings	boolean	equal (identical values)
!=	integers, floating points or strings	boolean	not equal (different values)
*=, -=	first operand: variable second operand: integer or floating point	integer/ floating point	Assignment with numeric operation
/=	first operand: variable second operand: integer or floating-point	floating-point	assignment with division
+=	first operand: variable second operand: integer, floating-point or string	integer/floating-point/string	assignment with addition or concatenation
typeof	any	integer	return internal data type (unary)
isvalid	any	boolean	check for validity (unary)

Operator	Operand types	Result type	Operation performed
?:	first operand: boolean second operand: any third operand: any	any	conditional expression
=	first operand: variable second operand: any	any	assignment
,	first operand: any second operand: any	any	multiple evaluation

! **Note**: All operators may also have an invalid result type.

Runtime error detection and handling

Since WMLScript functions are used to implement services for users that expect the user agents to work properly in all situations, error handling is of the utmost importance. This means that while the language might not provide an exception mechanism, for example, it should provide tools which prevent errors from occurring or which notice them and take appropriate action. Aborting a program execution is the last resort, to be used only in cases where nothing else is possible.

The following sections list errors that can occur when bytecode is downloaded and executed. Programming errors such as infinite loops are not discussed. For such cases a user-controlled abortion mechanism is needed.

Error detection

The error detection tools allow you to detect errors (whenever possible) that would disrupt system performance. Since WMLScript is a weakly typed language, special functionality has been provided to detect errors caused by invalid data types :

Check that the given variable contains the *right value*: WMLScript includes type validation library functions such as *Lang.isInt()*, *Lang.isFloat()*, *Lang.parseInt()* and *Lang.parseFloat()*.

Check that the given variable contains a value that is of the *right type*: WMLScript includes the operators *typeof* and *isvalid* that you can use for this purpose.

Error handling

Error handling takes place after an error has occurred. This is the case when the error could not have been prevented by error detection (because of memory limits, external signals etc) or because it would have been too difficult to do so (overflow, underflow etc). Such cases can be divided into two classes:

Fatal errors: These are errors that cause the program to abort. Since WMLScript functions are always called from some other user agent, program abortion should always be signaled to the calling user agent. The user agent then informs the user about the error.

Non-fatal errors: These are errors that can be signaled back to the program as special return values and the program then decides on the appropriate action.

The following errors are grouped according to their degree of severity.

Fatal errors

The following sections describe the fatal errors of WMLScript.

Bytecode errors

These errors are related to the bytecode and the instructions that are executed by the WMLScript bytecode interpreter. They are indications of erroneous constant pool elements, invalid instructions, invalid arguments to instructions, or instructions that cannot be completed.

Verification failed

Description: Reports that the specified bytecode for the called compilation unit did not pass the verification.

Generated: Every time a program attempts to call an external function.

Example:
```
var a = 3*OtherScript#doThis(param);
```

Severity: Fatal.

Predictability: Detected during the bytecode verification.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

Fatal library function error

Description: Reports that a call to a library function resulted in a fatal error.

Generated: Every time a call is made to a library function. Typically, this is an unexpected error in the library function implementation.

Example:
```
var a = String.format(param);
```

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

Invalid function arguments

Description: Reports that the number of arguments specified for a function call do not match the number of arguments specified in the called function.

Generated: Every time a call is made to an external function.

Example: Compiler generates an invalid parameter to an instruction, or the number of parameters in the called function has changed.

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

External function not found

Description: Reports that a call to an external function could not be found in the specified compilation unit.

Generated: Every time a program attempts to call an external function.

Example:
```
var a = 3*OtherScript#doThis(param);
```

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

Unable to load compilation unit

Description: Reports that the specified compilation unit could not be loaded due to unrecoverable errors in accessing the compilation unit in the network server, or that the specified compilation unit does not exist in the network server.

Generated: Every time a program attempts to call an external function.

Example:
```
var a = 3*OtherScript#doThis(param);
```

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

Access violation

Description: Reports an access violation. The called external function resides in a protected compilation unit.

Generated: Every time a program attempts to call an external function.

Example:
```
var a = 3*OtherScript#doThis(param);
```

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

Stack underflow

Description: Indicates a stack underflow because of a program error (compiler generated bad code).

Generated: Every time a program attempts to pop an empty stack.

Example: Only generated if compiler generates bad code.

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

Programmed abort

This error is generated when a WMLScript function calls the library function *Lang.abort()*. See "WMLScript standard libraries" on page 61 to abort the execution.

Description: Reports that the execution of the bytecode was aborted by a call to the *Lang.abort()* function.

Generated: Every time a program makes a call to the *Lang.abort()* function.

Example: `Lang.abort("Unrecoverable error");`

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

Memory exhaustion errors

These errors are related to the dynamic behaviour of the WMLScript interpreter and its use of memory.

Stack overflow

Description: Indicates a stack overflow.

Generated: Every time a program recourses too deeply or attempts to push too many variables onto the operand stack.

Example: `function f(x) { f(x+1); };`

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

Out of memory

Description: Indicates that no more memory resources are available to the interpreter.

Generated: Every time the operating system fails to allocate more space for the interpreter.

Example:
```
function f(x) {
   x=x+"abcdefghijklmnopqrstuvxyz";
   f(x);
};
```

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

External exceptions

The following exceptions are initiated outside the WMLScript bytecode interpreter.

Initiated by the user

Description: Indicates that the user wants to abort the execution of the program (for example, by pushing reset button).

Generated: At any time.

Example: User presses reset button while an application is running.

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

Initiated by the system

Description: Indicates that an external fatal error occurred while a program is running and it must be aborted. Exceptions can be caused by a low battery, power off, and so on.

Generated: At any time.

Example: The system is automatically switched off due to a low battery.

Severity: Fatal.

Predictability: None.

Solution: Abort program and signal an error to the caller of the WMLScript interpreter.

Non-fatal errors

The following sections describe the non-fatal errors of WMLScript.

Computational errors

These errors are related to arithmetical operations supported by the WMLScript.

Divide by zero

Description: Indicates a division by zero.

Generated: Every time a program attempts to divide by `0` (integer or floating-point division or remainder).

Example:

```
var a = 10;
var b = 0;
var x = a / b;
var y = a div b;
var z = a % b;
a /= b;
```

Severity: Non-fatal.

Predictability: High.

Solution: The result is an `invalid` value.

Integer overflow

Description: Reports an arithmetical integer overflow.

Generated: Every time a program attempts to execute an integer operation.

Example:

```
var a = Lang.maxInt();
var b = Lang.maxInt();
var c = a + b;
```

Severity: Non-fatal.

Predictability: High, although difficult in some cases.

Solution: The result is an `invalid` value.

Floating-point overflow

Description: Reports an arithmetical floating-point overflow.

Generated: Every time a program attempts to execute a floating-point operation.

Example:
```
var a = 1.6e308;
var b = 1.6e308;
var c = a * b;
```

Severity: Non-fatal.

Predictability: High, although difficult in some cases.

Solution: The result is an `invalid` value.

Floating-point underflow

Description: Reports an arithmetical underflow.

Generated: Every time the result of a floating-point operation is smaller than what can be represented.

Example:
```
var a = Float.precision();
var b = Float.precision();
var c = a * b;
```

Severity: Non-fatal.

Predictability: High, although difficult in some cases.

Solution: The result is a floating-point value `0.0`.

Constant reference errors

These errors are related to runtime references and to constants in the constant pool.

Not a number floating-point constant

Description: Reports a reference to a floating-point literal in the constant pool that is *not a number*, as defined in IEEE754.

Generated: Every time a program attempts to access a floating-point literal and the compiler has generated a *not a number* as a floating-point constant.

Example: A reference to a floating-point literal.

Severity: Non-fatal.

Predictability: High.

Solution: The result is an `invalid` value.

Infinite floating-point constant

Description: Reports a reference to a floating-point literal in the constant pool that is either positive or negative infinity.

Generated: Every time a program attempts to access a floating-point literal and the compiler has generated a floating-point constant with a value of positive or negative infinity.

Example: A reference to a floating-point literal.

Severity: Non-fatal.

Predictability: High.

Solution: The result is an `invalid` value.

Illegal floating-point reference

Description: Reports an erroneous reference to a floating-point value in the constant pool.

Generated: Every time a program attempts to use floating-point values and the environment supports only integer values.

Example:
```
var a = 3.14;
```

Severity: Non-fatal.

Predictability: High, can be detected during the runtime.

Solution: The result is an `invalid` value.

Conversion errors

These errors are related to automatic conversions supported by the WMLScript.

Integer too large

Description: Indicates a conversion to an integer value where the integer value is too large (positive or negative).

Generated: Every time an application attempts to make an automatic conversion to an integer value.

Example:
```
var a = -"99999999999999999999999999999999999999";
```

Severity: Non-fatal.

Predictability: None.

Solution: The result is an `invalid` value.

Floating-point too large

Description: Indicates a conversion to a floating-point value where the floating-point value is too large (positive or negative).

Generated: Every time an application attempts to make an automatic conversion to a floating-point value.

Example:
```
var a = -"99999999.999999999999e99999";
```

Severity: Non-fatal.

Predictability: None.

Solution: The result is an `invalid` value.

Floating-point too small

Description: Indicates a conversion to a floating-point value where the floating-point value is too small (positive or negative).

Generated: Every time an application attempts to make an automatic conversion to a floating-point value.

Example:
```
var a = -"0.01e-99";
```

Severity: Non-fatal.

Predictability: None.

Solution: The result is a floating-point value `0.0`.

Appendix A

WMLScript standard libraries

This appendix discusses the library interfaces for the standard set of libraries supported by WMLScript.

Typographical conventions

The libraries are described by the following information:

Name: Library name. Note that library names are case-sensitive.

Examples: `Lang`, `String`

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

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

Function: Specifies the function name and the number of function parameters. Note that function names are case-sensitive.

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

Description: Describes the function's behaviour and its parameters.

Parameters: Specifies the function's 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.

Example: If the *value1* <= 0 and the *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
```

WMLScript compliance

These standard library functions provide a mechanism for extending the WMLScript language. Thus, the specified library functions must follow the WMLScript conventions and rules.

Supported data types

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

Boolean
Integer
Float
String
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.

Data type conversions

Since WMLScript is a weakly typed language, the conversions between the data types are done automatically if necessary. The library functions follow WMLScript operator data type conversion rules except where explicitly stated otherwise.

Error handling

Error cases are handled in the same way as in the WMLScript language:

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.

Function-dependent error cases are handled by returning a suitable error code specified in each function definition.

Lang library

Name: Lang

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

abs

Function: abs(*value*)

Description: Returns the absolute value of the given number.

If the given number is an integer, an integer value is returned.

If the given number is a floating-point, 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
```

min

Function: min(*value1, value2*)

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

1. WMLScript operator data type conversion rules for integers and floating-points are used to specify the data type for comparison.
2. The numbers are compared and the smaller one is selected.
3. If the values are equal, 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)
```

max

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

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

1. WMLScript operator data type conversion rules for integers and floating-points are used to specify the data type.
2. The numbers are compared and the larger one is selected.
3. If the values are equal, 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)
```

parseInt

Function: parseInt(*value*)

Description: Returns an integer value defined by the string *value*. The legal integer syntax is specified by the 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: the parsed string is 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
```

parseFloat

Function: parseFloat(*value*)

Description: Returns a floating-point value defined by the string *value*. The legal floating-point syntax is specified by the 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: the parsed string is 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, 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
```

isInt

Function: isInt(*value*)

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
```

isFloat

Function: isFloat(*value*)

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, 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
```

maxInt

Function: maxInt( )

Description: Returns the maximum integer value.

Parameters: –

Return value: Integer 2147483647

Exceptions: –

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

minInt

Function: minInt( )

Description: Returns the minimum integer value.

Parameters: –

Return value: Integer –2147483647

Exceptions: –

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

float

Function: float( )

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

Parameters: –

Return value: Boolean.

Exceptions: –

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

exit

Function: exit(*value*)

Description: Ends the interpretation of the WMLScript bytecode and returns control to the caller of the WMLScript interpreter with the given return *value*. You can use this function 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
```

abort

Function: abort(*errorDescription*)

Description: Aborts the interpretation of the WMLScript bytecode and returns control to the caller of the WMLScript interpreter with the return *errorDescription*. You can use this function 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 string
```

random

Function: random(*value*)

Description: Returns an integer value with a 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 an approximately uniform distribution over that range.

If the *value* is a 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
```

seed

Function: seed(*value*)

Description: Initializes the pseudo-random number sequence and returns an empty string.

If the *value* is zero or a positive integer, the given *value* is used for initialization: otherwise a random initialization value is used.

If the *value* is a 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)
```

characterSet

Function: CharacterSet()

Description: Returns the character set supported by the WMLScript Interpreter. The return value is an integer that denotes a MIBEnum value assigned by the IANA for all character sets.

Parameters: -

Return value: Integer

Exceptions: -

Example:

```
Var charset = Lang.characterSet(); // charset = 4 for
latin1
```

Float library

Name: Float

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

int

Function: int(*value*)

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
```

floor

Function: floor(*value*)

Description: Returns the integer value that is nearest to but 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
```

ceil

Function: ceil(*value*)

Description: Returns the integer value that is nearest to but 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
```

pow

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

Description: Returns an approximation of the result of raising *value1* to the power of *value2*.

If *value1* is a negative number, *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
```

round

Function: round(*value*)

Description: Returns the number value that is closest to the given *value* and that 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);      // d=0
```

sqrt

Function: sqrt(*value*)

Description: Returns an approximation of the square root of the given *value*.

Parameters: *value*=Floating-point

Return value: Floating-point or invalid.

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

Example:

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

maxFloat

Function: maxFloat( )

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

Parameters: –

Return value: Floating-point 3.40282347E+38.

Exceptions: –

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

minFloat

Function: minFloat( )

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

Parameters: –

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

Exceptions: –

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

String library

Name: String

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

You can specify a special separator by which elements in a string can be separated, and you can then access these elements by specifying the separator and the element index. The first element in a string has an index of zero (0). Each occurrence of the separator in the string separates two elements. Note that no escaping of separators is allowed.

A *white space character* is one of the following characters:

- TAB: Horizontal Tabulation
- VT: Vertical Tabulation
- FF: Form Feed
- SP: Space
- LF: Line Feed
- CR: Carriage Return

length

Function: length(*string*)

Description: Returns the length (number of 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
```

isEmpty

Function: isEmpty(*string*)

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

charAt

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

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

If the *index* value is a 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 e=String.charAt(a, "first")      // e=invalid
```

subString

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

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, 0 is used for the *startIndex*.

If the *length* is larger than the remaining number of characters, the *length* is replaced by the number of remaining characters.

If the *startIndex* or the *length* is a 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"
```

find

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

Description: Returns the index of the first character in the *string* that matches the requested *subString*.

If no match is found, the 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, "gz");          // d=-1
var e=String.find(34, "3");          // e=0
```

replace

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

Description: Returns a new string resulting from replacing all occurrences of *oldSubString* in the given *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?"
```

elements

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

Description: Returns the number of elements in the given *string* separated by the given *separator*. An empty string ("") is a valid element, meaning that this function can never return a value that is less or equal to zero.

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

Return value: Integer or invalid.

Exceptions: Return `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=1
var e=String.elements("a", ";");     // e=1
var f=String.elements(";",";");      // f=2
var g=String.elements(";;,;",";.");  // g=4 separator=;
```

elementAt

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

Description: Searches *string* for *index*'th element, the elements being separated by a *separator*, and returns the corresponding element.

If the *index* is less than `0`, the first element is returned.

If the *index* is larger than the number of elements, the last element is returned.

If the *string* is an empty string, an empty string is returned.

If the *index* value is a 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"
```

removeAt

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

Description: Returns a new string where the element and the corresponding *separator* (if it exists) with the given *index* are removed from the given *string*.

If the *index* is less than 0, the first element is removed.

If the *index* is larger than the number of elements, the last element is removed.

If the *string* is empty, the function returns a new empty string.

If the *index* value is a 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 a 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"
```

replaceAt

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

Description: Returns a string with the current element at the specified *index* replaced by the given *element*.

If the *index* is less than `0`, the first element is replaced.

If the *index* is larger than the number of elements, the last element is replaced.

If the *string* is empty, the function returns a new string with the given *element*.

If the *index* value is a 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"
```

insertAt

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

Description: Returns a new 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`, `0` is used as the *index*.

If the *index* is larger than the number of elements, 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* value is a 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 a 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"
```

squeeze

Function: squeeze(*string*)

Description: Returns a string where all the 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! "
```

trim

Function: trim(*string*)

Description: Returns a string where all the 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!"
```

compare

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

Description: The return value indicates the lexicographic relationship of *string1* to *string2*. The relation is based on the relationships between the character codes in the native character set. 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
```

toString

Function: toString(*value*)

Description: Returns a string representation of the given *value*. This function performs exactly the same conversions as WMLScript except that the `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"
```

format

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

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 takes the following form:

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

The **`width`** argument is a non-negative 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 the width is not given, all characters of the *value* are printed (subject to the precision argument).

The **`precision`** argument specifies a non-negative 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 the precision value, the output value is padded on the left with zeroes. The value is not truncated when the number of digits exceeds the precision value. The default precision value is 1. If the precision value 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 must appear before it. The value is rounded to the appropriate number of digits. The default precision is 6; if the precision is `0` or if the period (.) appears without a number following it, no decimal point is printed. When the number of digits after the decimal point in the *value* is less than the `precision`, letter `0` should be padded to fill columns (e.g., result of `String.format ("%2.3f", 1.2)` will be `" 1.200"`)

s Specifies the maximum number of characters to be printed. By default, all the characters are printed. When the `width` is larger than `precision`, the `width` should be ignored.

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 format argument required; 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 size of the number, and the number of digits after the decimal point depends on the precision requested.

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 specifies 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"
```

URL library

Name: URL

Description: This library contains a set of functions for handling absolute URLs and relative URLs. The general URL syntax supported is:

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

isValid

Function: isValid(*url*)

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://www.acme.com/script#func()");
// a=true
var b=URL.isValid("../common#test()");
// b=true
var c=URL.isValid("experimental?://www.acme.com/pub")
// c=false
```

getScheme

Function: getScheme(*url*)

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.a.com");       // a="http"
var b=URL.getScheme("w.a.com");              // b=""
```

getHost

Function: getHost(*url*)

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://www.acme.com/pub");
// a="www.acme.com"
var b=URL.getHost("path#frag");
// b=""
```

getPort

Function: getPort(*url*)

Description: Returns the port number specified in the given *url*.

If no port 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 port number, an `invalid` value is returned.

Example:
```
var a=URL.getPort("http://www.acme.com:80/path");
// a="80"
var b=URL.getPort("http://www.acme.com/path");
// b=""
```

getPath

Function: getPath(*url*)

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.

Examples:

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

getParameters

Function: getParameters(*url*)

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:

```
var a=URL.getParameters("http://w.a.c/scr;3;2?x=1&y=3");
// a="3;2"
var b=URL.getParameters("../scr;3;2?x=1&y=3");
// b="3;2"
```

getQuery

Function: getQuery(*url*)

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:

```
var a=URL.getQuery("http://w.a.c/scr;3;2?x=1&y=3");
// a="x=1&y=3"
```

getFragment

Function: getFragment(*url*)

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://www.acme.com/cont#frag");
// a="frag"
```

getBase

Function: getBase( )

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.acme.com/test.scr"
```

getReferer

Function: getReferer( )

Description: Returns the smallest relative URL (relative to the base URL of the current compilation unit) 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, an empty string is returned.

Parameters: –

Return value: String.

Exceptions: –

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

resolve

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

Description: Returns an absolute URL from the given *baseUrl* and the *embeddedUrl* according to the rules specified in *RFC2396*.

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, `invalid` value is returned.

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

escapeString

Function: escapeString(*string*)

Description: This function computes a new version of a *string* value in which special characters are replaced by a hexadecimal escape sequence (you must use a two-digit escape sequence of the form %xx). The characters to be escaped are:

Control characters: US – ASCII coded characters 00 – 1F and 7F

Space: US - ASCII coded character 20 hexadecimal

Reserved: “;” | “/” | “?” | “:” | “@” | “&” | “=” | “+” | “$” | “,”

Unwise: “{“ | “}” | “|” | “\” | “^” | “[“ | “]” | “`”

Delims: “<” | “>” | “#” | “%” | <”>

The given string is escaped 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=URL.escapeString("http://w.a.c/dck?x=\u007ef#crd");
// a="http%3a%2f%2fw.a.c%2fdck%3fx%3d%ef%23crd"
```

unescapeString

Function: unescapeString(*string*)

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

The given string is unescaped as such; no URL parsing is performed.

Parameters: *string*=String

Return value: String or invalid.

Exceptions: If the *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.a.c%2fdck%3fx%3d12%23crd";
var b=URL.unescapeString(a);
// b ="http://w.a.c/dck?x=12#crd
```

loadString

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

Description: Returns the content denoted by the given absolute *url* and the *contentType*.

The given content type is erroneous if it does not follow the following rules:

You can only specify one content yype. The whole string must match with only one content type and you cannot have any extra leading or trailing spaces.

The type must be `text` but the subtype can be anything. The type prefix must be "`text/`"

This function behaves as follows:

The content with the given *contentType* 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 *contentType*, the content is converted to a string and returned.

If the load is unsuccessful or the returned content is of the wrong content type, 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 if the load fails.

If HTTP or WSP schemes are used, HTTP error codes are returned.

If an erroneous *contentType* is given, an `invalid` value is returned.

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

WMLBrowser library

Name: WMLBrowser

Description: This library contains functions which WMLScript uses to access the associated WML context. These functions must not have any side effects and they must return `invalid` in the following cases:

The system does not support WML Browser.

The WMLScript interpreter is not invoked by the WML Browser.

getVar

Function: getVar(*name*)

Description: Returns the value of the variable with the given *name* in the current browser context.

If the given variable does not exist, returns an empty string.

The variable name must follow the syntax specified by the WML Specification.

Parameters: *name*=String

Return value: String or invalid.

Exceptions: Returns an `invalid` value if the syntax of the variable name is incorrect.

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

setVar

Function: setVar(*name, value*)

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

The variable name and its value must follow the syntax specified by the WML Specification.

The variable value must be legal XML CDATA.

Parameters: *name*=String
value=String

Return value: Boolean or invalid.

Exceptions: Returns an `invalid` value if the syntax of the variable name or its value is incorrect.

Example:

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

go

Function: go(*url)*

Description: Specifies the content denoted by the given *url* to be loaded. This function has the same semantics as the GO task in WML.

The content is loaded only after the WML browser resumes the control back from the WMLScript interpreter after the WMLScript invocation is finished.

If the given *url* is an empty string (""), no content is loaded.

The go( ) and prev( ) library functions override each other. Both can be called many times before returning control to the WML browser.

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.

This function returns an empty string.

Parameters: *url*=String

Return value: String or invalid.

Exceptions: –

Example:
```
var card="http://www.acme.com/loc/app.dck#start";

WMLBrowser.go(card);
```

prev

Function: prev()

Description: Signals the WML browser to go back to the previous WML card. This function has the same semantics as the PREV task in WML.

The previous card is loaded only after the WML browser resumes the control back from the WMLScript interpreter after the WMLScript invocationis finished.

The go( ) and prev( ) library functions override each other. Both can be called many times before returning control to the WML browser.

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.

This function returns an empty string.

Parameters: -

Return value: String or invalid.

Exceptions: –

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

newContext

Function: newContext( )

Description: Clears the current WML browser context and returns an empty string. This function has the same semantics as the NEWCONTEXT attribute in WML.

Parameters: –

Return value: String or invalid.

Exceptions: –

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

getCurrentCard

Function: getCurrentCard( )

Description: Returns the smallest relative URL specifying the card (if any) currently being processed by the WML browser. The function returns an absolute URL if 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"
```

refresh

Function: refresh( )

Description: Forces the WML browser to update its context and returns an empty string. As a result, the user interface is updated to reflect the updated context. This function has the same semantics as the REFRESH task in WML.

Parameters: –

Return value: String or invalid.

Exceptions: –

Example:

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

Dialogs library

Name: Dialogs

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

prompt

Function: prompt(*message, defaultInput*)

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);
```

confirm

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

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 text)

Return value: Boolean or invalid.

Exceptions: –

Example:
```
function onAbort() {
  return Dialogs.confirm("Are you sure?", "Yes", "No");
};
```

alert

Function: alert(*message*)

Description: Displays the given *message* to the user, waits for the user to confirm 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!";
  };
};
```

Library summary

The libraries:

Library name
Lang
Float
String
URL
WMLBrowser
Dialogs

The libraries and their functions:

Lang library
abs
min
max
parseInt
parseFloat
isInt
isFloat
maxInt

Lang library
minInt
float
exit
abort
random
seed
characterSet

Float library
int
floor
ceil
pow
round
sqrt
maxFloat
minFloat

String library
length
isEmpty
charAt
subString
find

String library
replace
elements
elementAt
removeAt
replaceAt
insertAt
squeeze
trim
compare
toString
format

URL library
isValid
getScheme
getHost
getPort
getPath
getParameters
getQuery
getFragment
getBase
getReferer
resolve

URL library
escapeString
unescapeString
loadString

WMLBrowser library
getVar
setVar
go
prev
newContext
getCurrentCard
refresh

Dialogs library
prompt
confirm
alert

Example application

The following WMLScript example shows how to calculate mortgage payments.

```
/*
 * Calculate a mortgage's payment
 *
 *@param varname the variable name to store the result
 *@param principal the principal
 *@param interest the interest rate
 *@param num_payments the number of payements
 *@return the payment
 */
```

```
extern function payment(varname, principal, interest, num_payments) {
   /*
    * Interest formulae:
    *
    * If (i != 0), then:
    *       pmt = principal * [i * (1+i)^n / ((1+i)^n - 1)]
    *
    * If (i == 0), then:
    *       pmt = principal / n
    */
    var mi = interest/1200;   // monthly interest from annual percentage
    var payment = 0;
   if (mi != 0) {
      var tmp = Float.pow((1 + mi), num_payments);
      payment = principal * (mi * tmp / (tmp - 1));
   } else {
      if (num_payments != 0)
         payment = principal / num_payments;
   }
   var s;
   if (payment != 0)
      s = String.format("$%6.2f", payment);
   else
      s = "Missing data";

   /*
    * Send the result to the browser
    */
    WMLBrowser.setVar(varname, s);

   /*
    * Make sure the browser updates its current card
    */
    WMLBrowser.refresh();
};
```

Appendix B

WMLScript non-standard library

This appendix describes Nokia's non-standard "Debug" library.

Debug Library

Name: Debug

Description: This library contains a set of functions to help debug script applications.

! **Note**: This is Nokia's proprietary extension to the WMLScript library.

openFile

Function: openFile (*fileName*, *mode*)

Description: Opens a file for reading, writing, or appending

Parameters: *fileName*=String mode=String ("r" = open fileName for reading, "w" = open fileName for writing, "a" = open fileName for appending)

Return value: An empty string or invalid.

Exceptions: If *fileName* is not successfully opened (e.g., *fileName* does not exist) or the *mode* is not valid, invalid is returned. Invalid is also returned if *fileName* or *mode* is not a String.

Example:

```
Debug.openFile("c\\tmp\\script.debug", "r");
Debug.openFile("c:\\tmp\\debug", "a"):
```

closeFile

Function: closeFile ()

Description: Closes the Debug library file

Parameters: -

Return value: An empty string.

Exceptions -

Example:

```
Debug.closeFile();
```

println

Function: println(*string*)

Description: Writes *string* to the currently open Debug output file. If openFile has not been called, *string* is written to standard output.

Paremeters: string=String

Return value: An empty string or invalid.

Exceptions: If *string* is not a String, invalid is returned.

Example:

```
Debug.println("Function f1 ENTRY ...");
Debug.println("Function f1 EXIT ...");
```

Appendix C

WMLScript grammar

This apendix discusses the standard grammar of WMLScript.

Context-free grammars

This section describes the context-free grammars used in this guide to define the lexical and syntactic structure of a WMLScript program.

General

A context-free grammar consists of a number of productions. Each production has an abstract symbol called a *nonterminal* as its left-hand side and a sequence of one or more nonterminal and *terminal* symbols as its right-hand side. For each grammar, the terminal symbols are drawn from a specified alphabet.

A given context-free grammar specifies a *language*. It begins with a production consisting of a single distinguished nonterminal called the *goal symbol* followed by a (perhaps infinite) set of possible sequences of terminal symbols. They are the result of repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

Lexical grammar

A lexical grammar for WMLScript is given below in section "WMLScript lexical grammar" on page 108. This grammar uses the characters of the Universal Character set of ISO/IEC-10646 as its terminal symbols. It defines a set of productions, starting from the goal symbol *Input* that describes how sequences of characters are translated into a sequence of input elements.

The syntactic grammar of WMLScript is composed of terminal symbols called *tokens,* which are input elements other than white space and comments. These tokens are the reserved words, identifiers, literals and punctuators of the WMLScript language. Simple white space and single line comments are discarded and do not appear in the stream of input elements for the syntactic grammar. Likewise, a multi-line comment is simply discarded if it contains no line terminator. If a multi-line comment contains one or more line terminators then it is replaced by

a single line terminator, which becomes part of the stream of input elements for the syntactic grammar.

Productions of the lexical grammar are distinguished by having two colons (**::**) as separating punctuation.

Syntactic grammar

The syntactic grammar of WMLScript is given below in "WMLScript syntactic grammar" on page 114. This grammar features WMLScript tokens defined by the lexical grammar as its terminal symbols. It defines a set of productions, starting from the goal symbol *CompilationUnit*, that describe how sequences of tokens can form syntactically correct WMLScript programs.

Productions of the syntactic grammar are distinguished by having just one colon (**:**) as punctuation.

Numeric string grammar

A third grammar is used for translating strings into numeric values. This grammar is similar to the part of the lexical grammar which deals with numeric literals, and uses the characters of the US-ASCII character set as its terminal symbols. This grammar appears below in "Numeric string grammar" on page 121.

Productions of the numeric string grammar are distinguished by three colons (**:::**) used as punctuation.

Grammar notation

Throughout this guide, the terminal symbols of the lexical and string grammars, and some of the terminal symbols of the syntactic grammar, are shown in the **`Courier bold`** font, whenever the text directly refers to such a terminal symbol. This is also true in the productions of the grammars, which are to appear in a program exactly as written here.

Nonterminal symbols are shown in *italic* font. The definition of a nonterminal is introduced by the name of the nonterminal being defined, followed by one or more colons. The number of colons indicates the grammar the production belongs to. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the syntactic definition:

WhileStatement **:**

`while (` *Expression* **`)`** *Statement*

states that the nonterminal *WhileStatement* represents the token **`while`**, followed by a left parenthesis token, followed by an *Expression*, followed by a right parenthesis token, followed by a *Statement*. The occurrences of *Expression* and *Statement* are themselves nonterminals. As another example, the syntactic definition:

ArgumentList **:**

AssignmentExpression
ArgumentList **,** *AssignmentExpression*

states that an *ArgumentList* may represent either a single *AssignmentExpression* or an *ArgumentList*, followed by a comma, followed by an *AssignmentExpression*. This definition of *ArgumentList* is recursive, that is, it is defined in terms of itself. The result is that an *ArgumentList* may contain any positive number of arguments, separated by commas, where each argument expression is an *AssignmentExpression*. Such recursive definitions of nonterminals are common.

The subscripted "*opt*", which may appear after a terminal or nonterminal, indicates an optional symbol. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it. This means that:

VariableDeclaration **:**

Identifier VariableInitializer$_{opt}$

is a convenient abbreviation for:

VariableDeclaration **:**

Identifier
Identifier VariableInitializer

and that:

IterationStatement **:**

for (*Expression*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

is a convenient abbreviation for:

IterationStatement **:**

for (; *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*
for (*Expression* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

which in turn is an abbreviation for:

IterationStatement :

for (; ; *Expression*$_{opt}$ **)** *Statement*
for (; *Expression* **;** *Expression*$_{opt}$ **)** *Statement*
for (*Expression* **; ;** *Expression*$_{opt}$ **)** *Statement*
for (*Expression* **;** *Expression* **;** *Expression*$_{opt}$ **)** *Statement*

which in turn is an abbreviation for:

IterationStatement :

for (; ;) *Statement*
for (; ; *Expression* **)** *Statement*
for (; *Expression* **;)** *Statement*
for (; *Expression* **;** *Expression* **)** *Statement*
for (*Expression* **; ;)** *Statement*
for (*Expression* **; ;** *Expression* **)** *Statement*
for (*Expression* **;** *Expression* **;)** *Statement*
for (*Expression* **;** *Expression* **;** *Expression* **)** *Statement*

Therefore, the nonterminal *IterationStatement* actually has eight alternative right-hand sides.

Any number of occurrences of *LineTerminator* may appear between any two consecutive tokens in the stream of input elements without affecting the syntactic acceptability of the program.

When the words "**one of**" follow the colon(s) in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition. For example, the lexical grammar for WMLScript contains the production:

ZeroToThree **: :**

One of
0 1 2 3

which is merely a convenient abbreviation for:

ZeroToThree **: :**

0
1
2
3

When an alternative in a production of the lexical grammar or the numeric string grammar appears to be a multicharacter token, it represents the sequence of characters that would make up such a token.

The right-hand side of a production may specify that certain expansions are not permitted by using the phrase "**but not**" and then indicating the expansions that are not permitted. For example, the production:

Identifier **: :**

IdentifierName **but not** *ReservedWord*

means that the nonterminal *Identifier* may be replaced by any sequence of characters that could replace *IdentifierName* provided that the same sequence of characters could not replace *ReservedWord*.

Finally, a few nonterminal symbols are described by a descriptive phrase in Roman type in cases where it would be impractical to list all the alternatives:

SourceCharacter **:**

any Unicode character

Source text

WMLScript source text uses the Universal Character set of ISO/IEC-10646 to represent a sequence of characters. Currently, this character set is identical to Unicode 2.0. This guide uses the terms ISO 10646 and Unicode interchangeably to indicate the same document character set.

SourceCharacter **: :**

any Unicode character

There is no requirement that WMLScript documents be encoded using the full Unicode encoding, for example, UCS-4. Any character encoding ("charset") that contains an inclusive subset of the characters in Unicode may be used, for example, US-ASCII or ISO-8859-1.

WMLScript programs can only be represented using ASCII characters, which are equivalent to the first 128 Unicode characters. Non-ASCII Unicode characters may appear only within comments and string literals. In string literals, any Unicode character may also be expressed as a Unicode escape sequence consisting of six ASCII characters, namely **\u** plus four hexadecimal digits. Within a comment, such an escape sequence is an effectively ignored part of the comment. Within a string literal, the Unicode escape sequence contributes one character to the string value of the literal.

WMLScript lexical grammar

The following contains the specification of the lexical grammar for WMLScript:

***SourceCharacter* : :**

any Unicode character

***WhiteSpace* : :**

<TAB>
<VT>
<FF>
<SP>
<LF>
<CR>

***LineTerminator* : :**

<LF>
<CR>
<CR><LF>

***Comment* : :**

MultiLineComment
SingleLineComment

***MultiLineComment* : :**

/* *MultiLineCommentChars*$_{opt}$ ***/**

***MultiLineCommentChars* : :**

MultiLineNotAsteriskChar MultiLineCommentChars$_{opt}$
***** *PostAsteriskCommentChars*$_{opt}$

***PostAsteriskCommentChars* : :**

MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars$_{opt}$
***** *PostAsteriskCommentChars*$_{opt}$

***MultiLineNotAsteriskChar* : :**

SourceCharacter **but not** *asterisk* *****

MultiLineNotForwardSlashOrAsteriskChar **: :**

SourceCharacter **but not** *forward-slash* **/ or** *asterisk* *****

SingleLineComment **: :**

// *SingleLineCommentChars*$_{opt}$

SingleLineCommentChars **: :**

SingleLineCommentChar SingleLineCommentChars$_{opt}$

SingleLineCommentChar **: :**

SourceCharacter **but not** *LineTerminator*

Token **: :**

ReservedWord
Identified
Punctuator
Literal

ReservedWord **: :**

Keyword
KeywordNotUsedByWMLScript
FutureReservedWord
BooleanLiteral
InvalidLiteral

Keyword **: :**

One of

access	**equiv**	**meta**	
agent	**extern**	**name**	**while**
break	**for**	**path**	**url**
continue	**function**	**return**	
div	**header**	**typeof**	
div=	**http**	**use**	
domain	**if**	**user**	
else	**isvalid**	**var**	

KeywordNotUsedByWMLScript **::**

One of

`delete`	`lib`	`null`	`void`
`in`	`new`	`this`	`with`

FutureReservedWord **::**

One of

`case`	`default`	`finally`	`struct`
`catch`	`do`	`import`	`super`
`class`	`enum`	`private`	`switch`
`const`	`export`	`public`	`throw`
`debugger`	`extends`	`sizeof`	`try`

Identifier **::**

IdentifierName **but not** *ReservedWord*

IdentifierName **::**

IdentifierLetter
IdentifierName IdentifierLetter
IdentifierName DecimalDigit

IdentifierLetter **::**

One of
`a b c d e f g h i j k l m o p q r s t u v w x y z`
`A B C D E F G H I J K L M O P Q R S T U V W X Y Z`

DecimalDigit **::**

One of
`0 1 2 3 4 5 6 7 8 9`

Punctuator ::

One of

```
=       >       <       ==      <=      >=
!=      ,       !       ~       ?       :
.       &&      ||      ++      --      +
-       *       /       &       |       ^
%       <<      >>      >>>     +=      -=
*=      /=      &=      |=      ^=      %=
<<=     >>=     >>>=    (       )       {
}       ;       #
```

Literal ::

InvalidLiteral
BooleanLiteral
NumericLiteral
StringLiteral

InvalidLiteral ::

invalid

BooleanLiteral ::

true
false

NumericLiteral ::

DecimalIntegerLiteral
HexIntegerLiteral
OctalIntegerLiteral
DecimalFloatLiteral

DecimalIntegerLiteral ::

0
NonZeroDigit DecimalDigits$_{opt}$

NonZeroDigit : :

One of
`1 2 3 4 5 6 7 8 9`

HexIntegerLiteral : :

`0x` *HexDigit*
`0X` *HexDigit*
HexIntegerLiteral HexDigit

HexDigit : :

One of
`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

OctalIntegerLiteral : :

`0` *Octal Digit*
OctalIntegerLiteral OctalDigit

OctalDigit : :

One of
`0 1 2 3 4 5 6 7`

DecimalFloatLiteral : :

DecimalIntegerLiteral `.` $DecimalDigits_{opt}$ $ExponentPart_{opt}$
`.` *DecimalDigits* $ExponentPart_{opt}$
DecimalIntegerLiteral ExponentPart

DecimalDigits : :

DecimalDigit
DecimalDigits DecimalDigit

ExponentPart : :

ExponentIndicator SignedInteger

ExponentIndicator : :

One of
`e E`

SignedInteger **: :**

DecimalDigits
+ *DecimalDigits*
- *DecimalDigits*

StringLiteral **: :**

" *DoubleStringCharacters*$_{opt}$ **"**
' *SingleStringCharacters*$_{opt}$ **'**

DoubleStringCharacters **: :**

DoubleStringCharacter DoubleStringCharacters$_{opt}$

SingleStringCharacters **: :**

SingleStringCharacter SingleStringCharacters$_{opt}$

DoubleStringCharacter **: :**

SourceCharacter **but not** *double-quote* **"** **or** *backslash* **** **or** *LineTerminator*
EscapeSequence

SingleStringCharacter **: :**

SourceCharacter **but not** *single-quote* **'** **or** *backslash* **** **or** *LineTerminator*
EscapeSequence

EscapeSequence **: :**

CharacterEscapeSequence
OctalEscapeSequence
HexEscapeSequence
UnicodeEscapeSequence

CharacterEscapeSequence **: :**

**** *SingleEscapeCharacter*

SingleEscapeCharacter **: :**

One of

' " \ / b f n r t

***HexEscapeSequence* : :**

\x *HexDigit HexDigit*

***OctalEscapeSequence* : :**

*****OctalDigit*

*****OctalDigit OctalDigit*

**** *ZeroToThree OctalDigit OctalDigit*

***ZeroToThree* : :**

One of

`0 1 2 3`

***UnicodeEscapeSequence* : :**

\u *HexDigit HexDigit HexDigit HexDigit*

WMLScript syntactic grammar

The following contains the specification of the syntactic grammar for WMLScript.

***PrimaryExpression* :**

Identifier
Literal
(*Expression* **)**

***CallExpression* :**

PrimaryExpression
LocalScriptFunctionCall
ExternalScriptFunctionCall
LibraryFunctionCall

***LocalScriptFunctionCall* :**

FunctionName Arguments

***ExternalScriptFunctionCall* :**

ExternalScriptName **#** *FunctionName Arguments*

LibraryFunctionCall **:**

LibraryName **.** *FunctionName Arguments*

FunctionName **:**

Identifier

ExternalScriptName **:**

Identifier

LibraryName **:**

Identifier

Arguments **:**

( )
(ArgumentList **)**

ArgumentList **:**

AssignmentExpression
ArgumentList **,** *AssignmentExpression*

PostfixExpression **:**

CallExpression
Identifier **++**
Identifier **--**

UnaryExpression **:**

PostfixExpression
typeof *UnaryExpression*
isvalid *UnaryExpression*
++ *Identifier*
-- *Identifier*
+ *UnaryExpression*
- *UnaryExpression*
~ *UnaryExpression*
! *UnaryExpression*

MultiplicativeExpression **:**

UnaryExpression
MultiplicativeExpression **`*`** *UnaryExpression*
MultiplicativeExpression **`/`** *UnaryExpression*
MultiplicativeExpression **`div`** *UnaryExpression*
MultiplicativeExpression **`%`** *UnaryExpression*

AdditiveExpression **:**

MultiplicativeExpression
AdditiveExpression **`+`** *MultiplicativeExpression*
AdditiveExpression **`-`** *MultiplicativeExpression*

ShiftExpression **:**

AdditiveExpression
ShiftExpression **`<<`** *AdditiveExpression*
ShiftExpression **`>>`** *AdditiveExpression*
ShiftExpression **`>>>`** *AdditiveExpression*

RelationalExpression **:**

ShiftExpression
RelationalExpression **`<`** *ShiftExpression*
RelationalExpression **`>`** *ShiftExpression*
RelationalExpression **`<=`** *ShiftExpression*
RelationalExpression **`>=`** *ShiftExpression*

EqualityExpression **:**

RelationalExpression
EqualityExpression **`==`** *RelationalExpression*
EqualityExpression **`!=`** *RelationalExpression*

BitwiseANDExpression **:**

EqualityExpression
BitwiseANDExpression **`&`** *EqualityExpression*

BitwiseXORExpression **:**

BitwiseANDExpression
BitwiseXORExpression **`^`** *BitwiseANDExpression*

BitwiseORExpression **:**

BitwiseXORExpression
BitwiseORExpression **|** *BitwiseXORExpression*

LogicalANDExpression **:**

BitwiseORExpression
LogicalANDExpression **&&** *BitwiseORExpression*

LogicalORExpression **:**

LogicalANDExpression
LogicalORExpression **||** *LogicalANDExpression*

ConditionalExpression **:**

LogicalORExpression
LogicalORExpression **?** *AssignmentExpression* **:** *AssignmentExpression*

AssignmentExpression **:**

ConditionalExpression
Identifier AssignmentOperator AssignmentExpression

AssignmentOperator **: :**

One of
= *= /= %= += -= <<= >>= >>>= &= ^= |= div=

Expression **:**

AssignmentExpression
Expression **,** *AssignmentExpression*

Statement **:**

Block
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement

Block **:**

{ *StatementList*$_{opt}$ **}**

StatementList **:**

Statement
StatementList Statement

VariableStatement **:**

var *VariableDeclarationList* **;**

VariableDeclarationList **:**

VariableDeclaration
VariableDeclarationList **,** *VariableDeclaration*

VariableDeclaration **:**

Identifier VariableInitializer$_{opt}$

VariableInitializer **:**

= *ConditionalExpression*

EmptyStatement **:**

;

ExpressionStatement **:**

Expression **;**

IfStatement **:**

if (*Expression* **)** *Statement* **else** *Statement*
if (*Expression* **)** *Statement*

IterationStatement **:**

WhileStatement
ForStatement

WhileStatement :

while (*Expression* **)** *Statement*

ForStatement :

for (*Expression*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*
for (var *VariableDeclarationList* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

ContinueStatement :

continue ;

BreakStatement :

break ;

ReturnStatement :

return *Expression*$_{opt}$ **;**

FunctionDeclaration :

extern$_{opt}$ **function** *Identifier* **(** *FormalParameterList*$_{opt}$ **)** *Block* **;**$_{opt}$

FormalParameterList :

Identifier
FormalParameterList **,** *Identifier*

CompilationUnit :

Pragmas$_{opt}$ *FunctionDeclarations*

Pragmas :

Pragma
Pragmas Pragma

Pragma :

use *PragmaDeclaration* **;**

PragmaDeclaration **:**

ExternalCompilationUnitPragma
AccessControlPragma
MetaPragma

ExternalCompilationUnitPragma **:**

`url` *Identifier StringLiteral*

AccessControlPragma **:**

`access` *AccessControlSpecifier*

AccessControlSpecifier **:**

`domain` *StringLiteral*
`path` *StringLiteral*
`domain` *StringLiteral* **`path`** *StringLiteral*

MetaPragma **:**

`meta` *MetaSpecifier*

MetaSpecifier **:**

MetaName
MetaHttpEquiv
MetaUserAgent

MetaName **:**

`name` *MetaBody*

MetaHttpEquiv **:**

`http equiv` *MetaBody*

MetaUserAgent **:**

`user agent` *MetaBody*

MetaBody **:**

MetaPropertyName MetaContent $MetaScheme_{opt}$

MetaPropertyName **:**

StringLiteral

MetaContent **:**

StringLiteral

MetaScheme **:**

StringLiteral

FunctionDeclarations **:**

FunctionDeclaration
FunctionDeclarations FunctionDeclaration

Numeric string grammar

The following contains the specification of the numeric string grammar for WMLScript. This grammar is used for translating strings into numeric values, is similar to the part of the lexical grammar dealing with numeric literals, and uses the characters of the Unicode character set as its terminal symbols.

The following grammar can be used to convert strings into the following numeric literal values:

Decimal Integer Literal: Use the following productions starting from the goal symbol *StringDecimalIntegerLiteral*.

Decimal Floating-Point Literal: Use the following productions starting from the goal symbol *StringFloatingPointLiteral*.

StringDecimalIntegerLiteral **: : :**

StrWhiteSpace$_{opt}$ *StrSignedDecimalIntegerLiteral StrWhiteSpace*$_{opt}$

StringDecimalFloatingPointLiteral **: : :**

StrWhiteSpace$_{opt}$ *StSignedrDecimalIntegerLiteral StrWhiteSpace*$_{opt}$
StrWhiteSpace$_{opt}$ *StrSignedDecimalFloatingPointLiteral StrWhiteSpace*$_{opt}$

StrWhiteSpace **: : :**

StrWhiteSpaceChar StrWhiteSpace$_{opt}$

StrWhiteSpaceChar **: : :**

<TAB>
<VT>
<FF>
<SP>
<LF>
<CR>

StrSignedDecimalIntegerLiteral **: : :**

StrDecimalDigits
+ *StrDecimalDigits*
- *StrDecimalDigits*

StrSignedDecimalFloatingPointLiteral **: : :**

StrDecimalFloatingPointLiteral
+ *StrDecimalFloatingPointLiteral*
- *StrDecimalFloatingPointLiteral*

StrDecimalFloatingPointLiteral **: : :**

StrDecimalDigits **.** *StrDecimalDigits*$_{opt}$ *StrExponentPart*$_{opt}$
. *StrDecimalDigits StrExponentPart*$_{opt}$
StrDecimalDigits StrExponentPart

StrDecimalDigits **: : :**

StrDecimalDigit
StrDecimalDigits StrDecimalDigit

StrDecimalDigit **: : :**

One of
0 1 2 3 4 5 6 7 8 9

StrExponentPart **: : :**

StrExponentIndicator StrSignedInteger

StrExponentIndicator **: : :**

One of
e E

StrSignedInteger **: : :**

StrDecimalDigits
+ *StrDecimalDigits*
- *StrDecimalDigits*

URL call syntax

This section contains the grammar for specifying the syntactic structure of the URL call. It uses the characters of the US-ASCII character set as its terminal symbols.

```
http://www.acme.com/scr#foo(1, -3, 'hello')       // OK
http://www.acme.com/scr#bar(1, -3+1, 'good')      // Error
http://www.acme.com/scr#test(foo(1, -3, 'hello')) // Error
```

URLCallFragmentAnchor **: : :**

FunctionName **( )**
FunctionName **(** *ArgumentList* **)**

FunctionName **: : :**

FunctionNameLetter
FunctionName FunctionNameLetter
FunctionName DecimalDigit

FunctionNameLetter **: : :**

One of
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
_

DecimalDigit **: : :**

One of
0 1 2 3 4 5 6 7 8 9

ArgumentList **: : :**

Argument
ArgumentList **,** *Argument*

Argument **:::**

WhiteSpace$_{opt}$ *Literal WhiteSpace*$_{opt}$

WhiteSpace **:::**

Any US-ASCII character with a character code less than or equal to 32.

Literal **:::**

InvalidLiteral
BooleanLiteral
NumericLiteral
StringLiteral

InvalidLiteral **:::**

`invalid`

BooleanLiteral **:::**

`true`
`false`

NumericLiteral **:::**

SignedDecimalIntegerLiteral
SignedDecimalFloatLiteral

SignedDecimalIntegerLiteral **:::**

DecimalIntegerLiteral
+ *DecimalIntegerLiteral*
- *DecimalIntegerLiteral*

DecimalIntegerLiteral **:::**

DecimalDigit DecimalDigits$_{opt}$

SignedDecimalFloatLiteral **:::**

DecimalFloatLiteral
+ *DecimalFloatLiteral*
- *DecimalFloatLiteral*

DecimalFloatLiteral **: : :**

DecimalIntegerLiteral **.** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
. *DecimalDigits ExponentPart*$_{opt}$
DecimalIntegerLiteral ExponentPart

DecimalDigits **: : :**

DecimalDigit
DecimalDigits DecimalDigit

ExponentPart **: : :**

ExponentIndicator SignedInteger

ExponentIndicator **: : :**

One of
e E

SignedInteger **: : :**

DecimalDigits
+ *DecimalDigits*
- *DecimalDigits*

StringLiteral **: : :**

" *DoubleStringCharacters*$_{opt}$ **"**
' *SingleStringCharacters*$_{opt}$ **'**

DoubleStringCharacters **: : :**

DoubleStringCharacter DoubleStringCharacters$_{opt}$

SingleStringCharacters **: : :**

SingleStringCharacter SingleStringCharacters$_{opt}$

DoubleStringCharacter **: : :**

SourceCharacter **but not** *double-quote* **"** **or** *backslash* ****
EscapeSequence

SingleStringCharacter **: : :**

SourceCharacter **but not** *single-quote* **'or** *backslash* ****
EscapeSequence

EscapeSequence **: : :**

CharacterEscapeSequence
OctalEscapeSequence
HexEscapeSequence
UnicodeEscapeSequence

CharacterEscapeSequence **: : :**

**** *SingleEscapeCharacter*

SingleEscapeCharacter **: : :**

One of
' " \ / b f n r t

HexEscapeSequence **: : :**

\x *HexDigit HexDigit*

OctalEscapeSequence **: : :**

**** *OctalDigit*
**** *OctalDigit OctalDigit*
**** *ZeroToThree OctalDigit OctalDigit*

ZeroToThree **: : :**

One of
0 1 2 3

UnicodeEscapeSequence **: : :**

\u *HexDigit HexDigit HexDigit HexDigit*

Glossary

The following terms and conventions are used throughout this document.

American Standard Code for Information Interchange (ASCII)

ASCII is a standard developed by the American National Standards Institute (ANSI) to define computer-intelligible values for characters used in text. The ASCII set of 128 characters includes upper-case and lower-case letters of the English alphabet, numbers, punctuation, and 33 control codes (such as tab, bell , carriage return). ASCII uses 7 bits to represent each character. You may see ASCII characters identified by a decimal number from 0 to 127.

The standard ASCII character set uses just 7 bits for each character, consequently one bit of each octet is not used. Larger character sets, known as extended ASCII or high ASCII, use all 8 bits, allowing as many as 128 additional characters to be defined. Numerous extensions to ASCII have been devised and quite a few have become national or international standards. Notable among them is a family of international standards, ISO-8859, that defines extensions appropriate to certain language groups which ASCII alone cannot support. The most important member of this group is ISO-8859-1, known as ISO Latin-1, which provides for the languages of western Europe.

Attribute

A syntactical component of a WML element which is often used to specify a characteristic quality of an element, other than type or content.

Author

An author is a person or program that writes or generates WML, WMLScript or other content.

Bandwidth

Bandwidth is the capacity that a telecommunications medium has for carrying data. For analog or voice communication, bandwidth is measured in the difference between the upper and lower transmission frequencies expressed in cycles per second, or hertz (Hz). For digital communication, bandwidth and transmission speed are usually treated as synonyms and measured in bits per

second. The actual speed or transmission time of any message or file from origin to destination depends on a number of factors. Most Internet transmissions travel at very high speed on fiber optic lines most of the way. Switching en route, lower bandwidths on local loops at both ends, and server processing time add to the overall transmission time.

Byte

A sequence of consecutive bits treated as a unit. On almost all modern computers, a byte is comprised of 8 bits, though other numbers were formerly encountered. To avoid ambiguity, the term octet is used in the language of international standards to refer to an 8-bit unit.

Large amounts of memory are indicated in terms of kilobytes (1,024 bytes), megabytes (1,048,576 bytes), and gigabytes (approximately 1 billion bytes). A disk that can hold 1.44 megabytes, for example, is capable of storing approximately 1.4 million ASCII characters, or about 3,000 pages of information.

Bytecode

Content encoding where the content is typically a set of low-level opcodes, that is, instructions, and operands for a targeted piece of hardware or virtual machine.

Card

A single WML navigational and user interface unit. A card may contain information to present to the user or instructions for gathering user input, for example.

Character Encoding

When used as a verb, character encoding refers to a conversion between sequence of characters and a sequence of bytes. When used as a noun, character encoding refers to a method for converting a sequence of bytes to a a sequence of characters. Typically, WML document character encoding is captured in transport headers attributes, meta information placed within a document, or the XML declaration defined by the XML specification.

Client

A device or application that initiates a request for connection with a server.

Common Gateway Interface (CGI)

A programming language that enables you to use forms on your web site.

Concatenation

Concatenating two strings means sticking them together, one after another, to make a new string. For example, the string “foo” concatenated with the string “bar” gives the string “foobar”.

Content

Subject matter stored or generated at a web 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.

Deck

A collection of WML cards. A WML deck is also an XML document.

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.

Extensible Markup Language (XML)

The Extensible Markup Language is a World Wide Web Consortium (W3C) standard for Internet markup languages, of which WML is one such language. XML is a restricted subset of SGML.

Hypertext transfer protocol (HTTP)

HTTP is the underlying protocol used by the World Wide Web. HTTP defines how messages are formatted and transmitted, and what actions web servers and browsers should take in response to various commands. For example, when you enter an URL in your browser an HTTP command is sent to the web server directing it to retrieve and transmit the requested web page.

JavaScript™

A de facto standard language that can be used to add dynamic behaviour to HTML documents.

Resource

A network data object or service that can be identified by an URL. Resources may be available in multiple representations (for example, 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.

Standardized Generalized Markup Language (SGML)

The Standardized Generalized Markup Language is a general-purpose language for domain-specific markup languages. SGML is defined in the *ISO 8879* standard.

Terminal

A device providing the user with user agent capabilities, including the ability to request and receive information. Also called a mobile terminal or mobile station.

Transcode

The act of converting from one character set to another, for example, conversion from UCS-2 to UTF-8.

Unicode

An encoding scheme for written characters and text. Unlike ASCII, which uses 7 bits for each character, Unicode uses 16 bits, which means that it can

represent more than 65,000 unique characters, a huge increase over ASCII's code capacity of 128 characters. Unicode was authored and is maintained by the Unicode Consortium, a group comprised of major corporations and institutions involved in international computing. The character repertory and the codes assigned in Unicode are identical to those specified by *ISO 10646*, the international Universal Character Set (UCS) standard.

The Unicode Standard, Version 2.0 defines codes for characters used in every major language written today. In all, the Unicode standard currently defines codes for nearly 39,000 characters from the world's alphabetic, ideographic and syllabic scripts and symbol collections. The Unicode repertory was derived from many pre-existing character set standards to which previously unstandardized characters have been added. In particular, the first 256 code values are identical to those of ISO 8859-1 extended to 16 bits. Unicode values are displayed as four hex digits preceded by U+. For example, U+0041 is Latin upper-case A.

Uniform Resouce Identifier (URI)

Uniform Resource Identifiers (URI) identify resources in the web: documents, images, downloadable files, services, electronic mailboxes, and other resources. A URI can refer to an Uniform Resource Locator (URL) or an Uniform Resource Name (URN).

Uniform Resource Locator (URL)

URL stands for Uniform Resource Locator and is an address referring to a document on the Internet. The syntax of an URL consists of three elements:

— The protocol, or the communication language, that the URL uses.

— The domain name, or the exclusive name that identifies a web site.

— The pathname of the file to be retrieved.

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 is any piece of software or physical device that interprets WML, WMLScript, WTAI or other resources. They may include textual browsers, voice browsers and search engines, for example.

Web server

The server on which a given resource resides or is to be created. Often referred to as an origin server or an HTTP server.

Wireless Application Environment (WAE)

The Wireless Application Environment specifies a general-purpose application environment based fundamentally on World Wide Web technologies and philosophies. WAE specifies an environment that allows operators and service providers to build applications and services that can reach a wide variety of different platforms. WAE is part of the Wireless Application Protocol.

Wireless Application Protocol (WAP)

The Wireless Application Protocol specifies an application framework and network protocols for wireless devices such as mobile phones, pagers, and personal digital assistants (PDAs). The WAP specifications extend mobile networking technologies (such as digital data networking standards) and Internet technologies (such as XML, URLs, scripting, and various content formats).

Wireless Markup Language (WML)

The Wireless Markup Language is a hypertext markup language used to represent information for delivery to a narrowband device such as a mobile phone.

Wireless Markup Language Script (WMLScript)

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

Wireless Session Protocol (WSP)

The Wireless Session Protocol provides the upper-level application layer of WAP with a consistent interface for two session services. The first is connection-mode service that operates above a transaction layer protocol, and the second is a connectionless service that operates above a secure or non-secure datagram transport service.

Index

D

E

F

G

H

I

L

S

T

U

V