WAP-193-WMLScript Language Specification

June-2000

Wireless Application Protocol
WMLScript Language Specification
Version 1.2

Disclaimer:

This document is subject to change without notice.

Contents

1. SCOPE

Wireless Application Protocol (WAP) is a result of continuous work to define an industry-wide specification for developing applications that operate over wireless communication networks. The scope for the WAP Forum is to define a set of standards to be used by service applications. The wireless market is growing very quickly and reaching new customers and services. To enable operators and manufacturers to meet the challenges in advanced services, differentiation and

fast/flexible service creation, WAP defines a set of protocols in transport, session and application layers. For additional information on the WAP architecture, refer to *Wireless Application Protocol Architecture Specification* [WAP].

This paper is a specification of the WMLScript language. It is part of the WAP application layer and it can be used to add client side procedural logic. The language is based on ECMAScript [ECMA262] but it has been modified to better support low bandwidth communication and thin clients. WMLScript can be used together with Wireless Markup Language [WML] to provide intelligence to the clients but it has also been designed so that it can be used as a standalone tool.

One of the main differences between ECMAScript and WMLScript is the fact that WMLScript has a defined bytecode and an interpreter reference architecture. This way the narrowband communication channels available today can be optimally utilised and the memory requirements for the client kept to the minimum. Many of the advanced features of the ECMAScript language have been dropped to make the language smaller, easier to compile into bytecode and easier to learn. For example, WMLScript is a procedural language and it supports locally installed standard libraries.

2. DOCUMENT STATUS

This document is available online in the following formats:

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

2.1 Copyright Notice

2.2 Errata

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

2.3 Comments

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

2.4 Document Changes

2.4.1 WAP-193 24-Mar-2000

Change Request	Title	Comments
WAG-IBM-20000209-MustDefinition	RFC2119 Conformance	Section 5.1 – definition of SHOULD and MUST

2.5 Document History

Document Name	Date of Release
SPEC-WMLScript-v1.1	17-Jun-1999
WAP-193.WMLScript-Proposed	24-March-2000
WAP-193.WMLScript (Approved)	June-2000

3. REFERENCES

3.1 Normative references

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

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

[ISO10646] "Information Technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane", ISO/IEC 10646-1:1993.

[RFC2279] "UTF-8, a transformation format of Unicode and ISO 10646", F. Yergeau, January 1998. URL: ftp://ftp.isi.edu/in-notes/rfc2279.txt

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

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

[RFC2396] "Uniform Resource Identifiers (URI): Generic Syntax", T. Berners-Lee, et al., August 1998. URL: http://info.internet.isi.edu/in-notes/rfc/files/rfc2396.txt

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

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

[WML] "Wireless Markup Language Specification", WAP Forum, 04-November-1999. URL: http://www.wapforum.org/

[WMLSLibs] "WAP-194-WMLScript Standard Libraries Specification", WAP Forum, 15-May-2000. URL: http://www.wapforum.org/

[WSP] "Wireless Session Protocol", WAP Forum, 05-November-1999. URL: http://www.wapforum.org/

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

3.2 Informative References

[HTML4] "HTML 4.0 Specification, W3C Recommendation 18-December-1997, REC-HTML40-971218", D. Raggett, et al., September 17, 1997. URL: http://www.w3.org/TR/REC-html40

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

[WAE] "Wireless Application Environment Specification", WAP Forum, 04-November-1999. URL: http://www.wapforum.org/

4. DEFINITIONS AND ABBREVIATIONS

4.1 Definitions

The following are terms and conventions used throughout this specification.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]. In the absence of any such terms, the specification should be interpreted as "MUST".

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

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

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

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

Content Format – actual representation of content.

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

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

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

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

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

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

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

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

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

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

4.2 Abbreviations

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

API	Application Programming Interface
BNF	Backus-Naur Form
ECMA	European Computer Manufacturer Association
HTML	HyperText Markup Language [HTML4]
HTTP	HyperText Transfer Protocol [RFC2068]
IANA	Internet Assigned Number Authority
LSB	Least Significant Bits
MSB	Most Significant Bits
RFC	Request For Comments
UI	User Interface
URL	Uniform Resource Locator [RFC2396]
UTF	UCS Transformation Format
UCS	Universal Multiple-Octet Coded Character Set
W3C	World Wide Web Consortium
WWW	World Wide Web
WSP	Wireless Session Protocol
WTP	Wireless Transport Protocol
WAP	Wireless Application Protocol
WAE	Wireless Application Environment
WTA	Wireless Telephony Applications
WTAI	Wireless Telephony Applications Interface
WBMP	Wireless BitMaP

5. OVERVIEW

5.1 Why Scripting?

WMLScript is designed to provide general scripting capabilities to the WAP architecture. Specifically, WMLScript can be used to complement the Wireless Markup Language [WML]. WML is a markup language based on Extensible Markup Language [XML]. It is designed to be used to specify application content for narrowband devices like cellular phones and pagers. This content can be represented with text, images, selection lists etc. Simple formatting can be used to make the user interfaces more readable as long as the client device used to display the content can support it. However, all this content is *static* and there is no way to extend the language without modifying WML itself. The following list contains some capabilities that are not supported by WML:

- Check the validity of user input (validity checks for the user input)
- Access to facilities of the device. For example, on a phone, allow the programmer to make phone calls, send messages, add phone numbers to the address book, access the SIM card etc.
- Generate messages and dialogs locally thus reducing the need for expensive round-trip to show alerts, error messages, confirmations etc.
- Allow extensions to the device software and configuring a device 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.

5.2 Benefits of using WMLScript

Many of the services that can be used with thin mobile clients can be implemented with WML. Scripting enhances the standard browsing and presentation facilities of WML with behavioural capabilities. They can be used to supports more advanced UI functions, add intelligence to the client, provide access to the device and its peripheral functionality and reduces the amount of bandwidth needed to send data between the server and the client.

WMLScript is loosely based on ECMAScript [ECMA262] and does not require the developers to learn new concepts to be able to generate advanced mobile services.

6. WMLSCRIPT CORE

One objective for the WMLScript language is to be close to the core of the ECMAScript Language specification [ECMA262]. The part in the ECMAScript Language specification that defines basic types, variables, expressions and statements is called *core* and can almost be used "as is" for the WMLScript specification. This section gives an overview of the core parts of WMLScript.

See section *WMLScript Grammar* (7) for syntax conventions and precise language grammar.

6.1 Lexical Structure

This section describes the set of elementary rules that specify how you write programs in WMLScript.

6.1.1 Case Sensitivity

WMLScript is a case-sensitive language. All language keywords, variables and function names must use the proper capitalisation of letters.

6.1.2 Whitespace and Line Breaks

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

Syntax:

WhiteSpace **::**
<TAB>
<VT>
<FF>
<SP>
<LF>
<CR>

LineTerminator **::**
<LF>
<CR>
<CR><LF>

6.1.3 Usage of Semicolons

The following statements in WMLScript have to be followed by a semicolon:[1]

- Empty statement (see section 6.5.1)
- Expression statement (see section 6.5.2)
- Variable statement (see section 6.5.4)
- Break statement (see section 6.5.8)
- Continue statement (see section 6.5.9)

[1] Compatibility note: ECMAScript supports optional semicolons.

- Return statement (see section 6.5.10)

6.1.4 Comments

The language defines two comment constructs: *line comments* (ie, start with // and end in the end of the line) and *block comments* (ie, consisting of multiple lines starting with /* and ending with */). It is illegal to have nested block comments.[2]

Syntax:

Comment **::**
MultiLineComment
SingleLineComment

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

SingleLineComment **::**
// *SingleLineCommentChars*$_{opt}$

6.1.5 Literals

6.1.5.1 Integer Literals

Integer literals can be represented in three different ways: 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

[2] Compatibility note: ECMAScript also supports HTML comments.

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 6.2.7.1. An integer literal that is not within the specified value range must result in a compile time error.

6.1.5.2 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 6.2.7.2. A floating-point literal that is not within the specified value range must result in a compile time error. A floating-point literal underflow results in a floating-point literal zero (`0.0`).

6.1.5.3 String Literals

Strings are any sequence of zero or more characters enclosed within double (") or single quotes (').

Syntax:

StringLiteral **::**
" $DoubleStringCharacters_{opt}$ **"**
' $SingleStringCharacters_{opt}$ **'**

Examples of valid strings are:

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

Since some characters are not representable within strings, WMLScript supports special escape sequences by which these characters can be represented:

Sequence	**Character represented**[3]	**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*hhhh*	The Unicode character with the encoding specified by the four hexadecimal digits *hhhh*.		

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 quote mark that might terminate the string literal.

6.1.5.4 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`

6.1.5.5 Invalid Literal

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

[3] Compatibility note: ECMAScript supports also non-escape characters preceded by a backslash.

Syntax:

InvalidLiteral **::**
`invalid`

6.1.6 Identifiers

Identifiers are used to name and refer to three different elements of WMLScript: variables (see section 6.2), functions (see section 6.4) and pragmas (see section 6.7). Identifiers[4] 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 are:

```
timeOfDay  speed  quality  HOME_ADDRESS  var0  _myName  ____
```

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

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

Uppercase and lowercase letters are distinct which means that the identifiers `speed` and `Speed` are different.

6.1.7 Reserved Words

WMLScript specifies a set of reserved words that have a special meaning in programs and they cannot be used as identifiers. Examples of such words are (full list can be found from the WMLScript grammar specification, see section 7):

---

[4] Compatibility note: ECMAScript supports the usage of $ character in any position of the name, too.

```
break  continue  false  true  while
```

6.1.8 Name Spaces

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

- Function names (see section 6.4.1)
- Function parameters (see section 6.4.2) and variables (see section 6.2)
- Pragmas (see section 6.7)

Thus, the same identifiers can be used to specify a function name, variable/parameter name or a name for a pragma within the same compilation unit:

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

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

6.2 Variables and Data Types

This section describes the two important concepts of WMLScript language: variables and internal data types. A variable is a name associated with a data value. Variables can be used to store and manipulate program data. WMLScript supports local variables[5] only declared inside functions or passed as function parameters (see section 6.4).

6.2.1 Variable Declaration

Variable declaration is compulsory[6] in WMLScript. Variable declaration is done simply by using the *var* keyword and a variable name (see section 6.5.4 for information about variable statements). Variable names follow the syntax defined for all identifiers (see section 6.1.6):

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

Variables must be declared before they can be used. Initialisation of variables is optional. Uninitialised variables are automatically initialised to contain an empty string ("").

6.2.2 Variable Scope and Lifetime

The scope of WMLScript variables is the remainder of the function (see section 6.4) in which they have been declared. All variable names within a function must be unique. Block statements (see section 6.5.3) are not used for scoping.

---

[5] Compatibility note: ECMAScript supports global variables, too.

[6] Compatibility note: ECMAScript supports automatic declaration, too.

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

The lifetime of a variable is the time between the variable declaration and the end of the function.

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

6.2.3 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 name:

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

6.2.4 Variable Type

WMLScript is a weakly typed language. The variables are not typed but 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 an invalid data type is needed to separate it from the other internal data types. Since these data types are supported only internally, the programmer does not have to specify variable types and any variable can contain any type of data at any given time. WMLScript will attempt automatically 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
```

6.2.5 L-Values

Some operators (see section 6.3.1 for more information about assignment operators) require that the left operand is 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
```

6.2.6 Type Equivalency

WMLScript supports operations on different data types. All operators (see section 6.3) specify the accepted data types for their operands. Automatic data type conversions (see section 6.8) are used to convert operand values to required data types.

6.2.7 Numeric Values

WMLScript supports two different numeric variable values: *integer* and *floating-point* values[7]. Variables can be initialised with integer and floating-point literals and several operators can be used to modify their values during the run-time. Conversion rules between integer and floating-point values are specified in chapter 6.8

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

6.2.7.1 Integer Size

The size of the integer is 32 bits (two's complement). This means that the supported value range[8] for integer values is: -2147483648 and 2147483647. *Lang* [WMLSLibs] library functions can be used to get these values during the run-time:

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

6.2.7.2 Floating-point Size

The minimum/maximum values[9] and precision for floating-point values are specified by [IEEE754]. WMLScript supports 32-bit single precision floating-point format:

- Maximum value: 3.40282347E+38
- Minimum positive nonzero value (at least the normalised precision must be supported): 1.17549435E-38 or smaller

The *Float* [WMLSLibs] library can be used to get these values during the run-time:

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

---

[7] Convention: In cases where the value can be either an integer or a floating-point, a more generic term *number* is used instead.

[8] Compatibility note: ECMAScript does not specify maximum and minimum values for integers. All numbers are represented as floating-point values.

[9] Compatibility note: ECMAScript uses double-precision 64-bit format [IEEE754] floating-point values for all numbers.

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 part of the set of finite real numbers (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.

6.2.8 String Values

WMLScript supports *strings* that can contain letters, digits, special characters etc. Variables can be initialised with string literals and string values can be manipulated both with WMLScript operators and functions specified in the standard *String* library [WMLSLibs].

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

6.2.9 Boolean Values

Boolean values can be used to initialise or assign a value to a variable or in statements which require a boolean value as one of the parameters. Boolean value can be a literal or the result of a logical expression evaluation (see section 6.3.3 for more information).

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

6.3 Operators and Expressions

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

6.3.1 Assignment Operators

WMLScript supports several ways to assign a value to a variable. The simplest one is the regular assignment (=) but assignments with operation 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

Operator	Operation
>>>=	bitwise right shift zero fill and assign
&=	bitwise AND and assign
^=	bitwise XOR and assign
|=	bitwise OR and assign

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"
```

6.3.2 Arithmetic Operators

WMLScript supports all the basic binary arithmetic operations:

Operator	Operation
+	add (numbers)/concatenation (strings)
-	subtract
*	multiply
/	divide
div	integer division

In addition to these, a set of more complex binary operations are supported, too:

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

The basic unary operations supported are:

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

Examples:

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

6.3.3 Logical Operators

WMLScript supports the basic logical operations:

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

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 value of boolean type for logical operations. Automatic conversions from other types to boolean type and *vice versa* are supported (see section 6.8).

Notice: If the value of the first operand for logical 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
```

6.3.4 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[10] are supported by a standard *String* library (see [WMLSLibs]).

[10] Compatibility note: ECMAScript supports String objects and a length attribute for each string. WMLScript does not support objects. However, similar functionality is provided by WMLScript libraries.

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

6.3.5 Comparison Operators

WMLScript supports all the basic comparison operations:

Operator	Operation
<	less than
<=	less than or equal
==	equal
>=	greater 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 given string 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:

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

6.3.6 Array Operators

WMLScript does not support arrays[11] as such. However, the standard *String* library (see [WMLSLibs]) supports functions by which array like behaviour can be implemented by using strings. A string can contain elements that are separated by a separator specified by the application programmer. For this purpose, the *String* library contains functions by which creation and management of string arrays can be done.

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

6.3.7 Comma Operator

WMLScript supports the comma (`,`) operator by which multiple evaluations can be combined into one expression. The result of the comma operator is the value of the second operand:

[11] Compatibility note: ECMAScript supports arrays.

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

Commas used in the function call to separate parameters and in the variable declarations to separate multiple variable declarations are 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
```

6.3.8 Conditional Operator

WMLScript 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.

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

Notice: This operator behaves like an *if* statement (see section 6.5.5). The third operand is evaluated if the evaluation of the condition results in `false` or `invalid`.

6.3.9 typeof Operator

Although WMLScript is a weakly typed language, internally the following basic data types are supported: *boolean*, *integer*, *floating-point*, *string* and *invalid*. Typeof (*typeof*) operator returns an integer value[12] that describes the type of the given expression. The possible results are:

Type	Code
Integer:	0
Floating-point:	1
String:	2
Boolean:	3
Invalid:	4

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.

[12] Compatibility note: ECMAScript specifies that the *typeof* operator returns a string representing the variable type.

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

6.3.10 isvalid Operator

This operator can be used to check the type of the given expression. It returns a boolean value `false` if the type of the expression is invalid, otherwise `true` is returned. *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
```

6.3.11 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
```

Expressions that are more complex can be defined by using simple expressions together with operators and function calls.

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

6.3.12 Expression Bindings

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

Precedence[13]	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)

13 Binding: 0 binds tightest

Precedence[13]	Associativity	Operator	Operand types	Result type	Operation performed
1	R	!	boolean	boolean*	logical NOT (unary)
1	R	typeof	any	integer	return internal data type (unary)
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 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
11	L	||	booleans	boolean*	logical OR
12	R	? :	boolean, any, 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 in case the data type conversions fail (see section 6.8 for more information about conversion rules) or one of the operands is `invalid`.

6.4 Functions

A WMLScript function is a named part of the WMLScript compilation unit that can be called to perform a specific set of statements and to return a value. The following sections describe how WMLScript functions can be declared and used.

6.4.1 Declaration

Function declaration can be used 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 initialised before the function body (block of statements) is executed.
- A function *always* returns a value. By default it is an empty string (""). However, a *return* statement can be used to specify other return values.

Functions in WMLScript are not data types[14] but a syntactical feature of the language.

Syntax:

FunctionDeclaration **:**

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

FormalParameterList **:**

Identifier
FormalParameterList **,** *Identifier*

Arguments: The optional **extern** keyword can be used to specify a function to be externally accessible. Such functions can be called from outside the compilation unit in which they are defined. There must be at least one externally accessible function in a compilation unit. *Identifier* is the name specified for the function. *FormalParameterList* (optional) is a comma-separated list of argument names. *Block* is the body of the function that is executed when the function is called and the parameters have been initialised by the passed arguments.

[14] Compatibility note: Functions in ECMAScript are actual data types.

Examples:

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

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

6.4.2 Function Calls

The way a function is called depends on where the called (target) function is declared. The following sections describe the three function calls supported by WMLScript: local script function call, external function call and library function call.

6.4.2.1 Local Script Functions

Local script functions (defined inside the same compilation unit) can be called simply by providing the function name and a comma separated list of arguments (number of arguments must match the number of parameters[15] accepted by the function).

Syntax:

LocalScriptFunctionCall **:**
FunctionName Arguments

FunctionName **:**
Identifier

Arguments **:**
( )
(*ArgumentList* **)**

ArgumentList **:**
AssignmentExpression
ArgumentList **,** *AssignmentExpression*

Functions inside the same compilation unit can be called before the function has been declared:

[15] Compatibility note: ECMAScript supports a variable number of arguments in a function call.

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

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

6.4.2.2 External Functions

External function calls must be used when the called function is declared in an external compilation unit. The 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

Pragma `use url` (see section 6.7) must be used to specify the external compilation unit. It defines the mapping between the external unit and a name that can be used within function declarations. This name and the hash symbol (#) are used to prefix the standard function call syntax:

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

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

6.4.2.3 Library Functions

Library function calls must be used when the called function is a WMLScript standard library function [WMLSLibs].

Syntax:

LibraryFunctionCall **:**

LibraryName **.** *FunctionName Arguments*

LibraryName **:**

Identifier

A library function can be called by prefixing the function name with the name of the library (see section 6.6 for more information) and the dot symbol (.):

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

6.4.3 Default Return Value

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

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

6.5 Statements

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

The following sections define the statements available in WMLScript[16]: empty statement, expression statement, block statement, break, continue, for, if...else, return, var, while.

6.5.1 Empty Statement

Empty statement is a statement that can be used where a statement is needed but no operation is required.

Syntax:

EmptyStatement **:**
`;`

Examples:

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

6.5.2 Expression Statement

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

Syntax:

ExpressionStatement **:**
Expression `;`

Expression **:**
AssignmentExpression
Expression `,` *AssignmentExpression*

[16] Compatibility note: ECMAScript supports also *for..in* and *with* statements.

Examples:

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

6.5.3 Block Statement

A set of statements enclosed in the curly brackets is a block statement. It can be used anywhere a single statement is needed.

Syntax:

Block **:**

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

StatementList **:**

Statement
StatementList Statement

Example:

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

6.5.4 Variable Statement

This statement declares variables with initialisation (optional, variables are initialised to empty string ("") by default). The scope of the declared variable is the rest of the current function (see section 6.2.2 for more information about variable scoping).

Syntax:

VariableStatement **:**

var *VariableDeclarationList* **;**

VariableDeclarationList **:**

VariableDeclaration
VariableDeclarationList **,** *VariableDeclaration*

VariableDeclaration **:**

Identifier VariableInitializer$_{opt}$

VariableInitializer **:**

= *ConditionalExpression*

Arguments: *Identifier* is the variable name. It can be any legal identifier. *ConditionalExpression* is the initial value of the variable and can be any legal expression. This expression (or the default initialisation to an empty string) is evaluated every time the variable statement is executed.

Variable names must be unique within a single function.

Examples:

```
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;            // Variable a, b and c are accessible
};
```

6.5.5 If Statement

This statement is used to specify 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*

Arguments: *Expression* (condition) can be any WMLScript expression that evaluates (directly or after conversion) to a *boolean* or an *invalid* value. If condition evaluates to `true`, the first statement is executed. If condition evaluates to `false` or `invalid`, the second (optional) `else` statement is executed. *Statement* can be any WMLScript statement, including another (nested) `if` statement. `else` is always tied to the closest `if`.

Example:

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

6.5.6 While Statement

This statement is used to create a loop that evaluates an expression and, if it is `true`, execute a statement. The loop repeats as long as the specified condition is `true`.

Syntax:

WhileStatement **:**

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

Arguments: *Expression* (condition) can be any WMLScript expression that evaluates (directly or after the conversion) to a *boolean* or an *invalid* value. The condition is evaluated before each execution of the loop statement. If this condition evaluates to `true`, the *Statement* is performed. When 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:

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

6.5.7 For Statement

This statement is used to create loops. The statement 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*

Arguments: The first *Expression* or *VariableDeclarationList* (initialiser) is typically used to initialise 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 (see section 6.2.2 for more information about variable scoping).

The second *Expression* (condition) can be any WMLScript expression that evaluates (directly or after the conversion) to a *boolean* or an *invalid* value. The condition is evaluated on each pass through the loop. If this 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:

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

6.5.8 Break Statement

This statement is used to terminate the current *while* or *for* loop and continue the program execution from the statement following the terminated loop. It is an error to use break statement outside a *while* or a *for* statement.

Syntax:

BreakStatement **:**

`break ;`

Example:

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

6.5.9 Continue Statement

This statement is used to terminate execution of a block of statements in a *while* or *for* loop and continue execution of the loop with the next iteration. 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.

It is an error to use continue statement outside a *while* or a *for* statement.

Syntax:

ContinueStatement **:**

`continue ;`

Example:

```
var index = 0;
var count = 0;
while (index < 5) {
   index++;
   if (index == 3)
      continue;
   count += index;
};
```

6.5.10 Return Statement

This statement can be used inside the function body to specify the function return value. If no return statement is specified or none of the function return statements is executed, the function returns an empty string by default.

Syntax:

ReturnStatement **:**

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

Example:

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

6.6 Libraries

WMLScript supports the usage of libraries[17]. Libraries are named collections of functions that belong logically together. These functions can be called by using a dot (‘.’) separtor with the library name and the function name with parameters:

An example of a library function call:

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

6.6.1 Standard Libraries

Standard libraries are specified in more detail in the *WAP-194-WMLScript Standard Libraries Specification* [WMLSLibs].

[17] Compatibility note: ECMAScript does not support libraries. It supports a set of predefined objects with attributes. WMLScript uses libraries to support similar functionality.

6.7 Pragmas

WMLScript supports the usage of *pragmas* that specify compilation unit level information. Pragmas are specified at the beginning of the compilation unit before any function declaration. 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 about the supported pragmas.

6.7.1 External Compilation Units

WMLScript compilation units can be accessed by using a URL. Thus, each WMLScript function can be accessed by specifying the URL of the WMLScript resource and its name. A `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*. This name can then be used inside the function declarations to make external function calls (see section 6.4.2.2).

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

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

The behaviour of the previous example is the following:

- The pragma specifies a URL to a WMLScript compilation unit.
- The function call loads the compilation unit by using the given URL (`http://www.host.com/app/script`)
- 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 [RFC2396] without a hash mark (#) or a fragment identifier. The schemes supported are specified in [WAE].
- Relative URLs [RFC2396] without a hash mark (#) or a fragment identifier: The base URL is the URL that identifies the current compilation unit.

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

6.7.2 Access Control

A WMLScript compilation unit can protect its content by using an *access control* pragma. Access control must be performed before calling external functions. It is an 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. Access control pragma is used to specify *domain* and *path* attributes against which these 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 capitalisation rules.

Domain suffix matching is done using the entire element of each sub-domain and must match each element exactly (e.g. `www.wapforum.org` shall match `wapforum.org`, but shall not match `forum.org`).

Path prefix matching is done using entire path elements and must match each element exactly (e.g. `/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 [RFC2396]. The user agent converts the relative path to an absolute path and then performs prefix matching against the path attribute.

Given the following access control attributes for a compilation unit:

```
use access domain "wapforum.org" path "/finance";
```

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

```
http://wapforum.org/finance/money.cgi
https://www.wapforum.org/finance/markets.cgi
http://www.wapforum.org/finance/demos/packages.cgi?x=123&y=456
```

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

```
http://www.test.net/finance
http://www.wapforum.org/internal/foo.wml
```

By default, access control is disabled (ie, all external functions have public access).

6.7.3 Meta-Information

Pragmas can also be used to specify compilation unit specific meta-information. Meta-information is specified with property names and values. This specification does not define any properties, nor does it define how user agents must interpret meta-data. User agents are not required to act on the meta-data.

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*, *content* (the value of the property) and 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.

6.7.3.1 Name

Name meta-pragma is used to specify meta-information intended to be used by the origin servers. The user agent should ignore any meta-data named with this attribute. Network servers should not emit WMLScript content containing meta-name pragmas.

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

6.7.3.2 HTTP Equiv

HTTP equiv meta-pragma is used to specify meta-information that indicates that the property should be interpreted as an HTTP header (see [RFC2068]). Meta-data named with this attribute should be 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";
```

6.7.3.3 User Agent

User agent meta-pragma is used to specify meta-information intended to be used by the user agents. This meta-data must be delivered to the user agent and must not be removed by any network intermediary.

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

6.8 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.

6.8.1 General Conversion Rules

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 at any given time can contain a value of one of these types. WMLScript provides an operator *typeof*, which can be used to determine what is the current type of a variable or any expression (no conversions are performed).

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

6.8.2 Conversions to String

Legal conversions from other data types to string are:

- Integer value must be converted to a string of decimal digits that follows the numeric string grammar rules for decimal integer literals. See section 7.4 for more information about the numeric string grammar.
- Floating-point value must be converted to an implementation-dependent string representation that follows the numeric string grammar rules for decimal floating-point literals (see section 7.1.4 for more information about the numeric string grammar). The resulting string representation must be equal to the original value (ie `.5` can be represented as `"0.5"`, `".5e0"`, etc.).
- The boolean value `true` is converted to string `"true"` and the value `false` is converted to string `"false"`.
- `Invalid` can not be converted to a string value.

6.8.3 Conversions to Integer

Legal conversions from 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 (see section 7.4 for the numeric string grammar rules for a decimal integer literal).
- Floating-point value cannot be converted to an integer value.

- The boolean value `true` is converted to integer value 1, `false` to 0.
- `Invalid` can not be converted to an integer value.

6.8.4 Conversions to Floating-Point

Legal conversions from 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 (see section 7.4 for the numeric string grammar rules for a decimal floating-point literal).
- 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` can not be converted to a floating-point value.

The conversions between a string and a floating-point type must be transitive within the ability of the data types to accurately represent the value. A conversion could result in loss of precision.

6.8.5 Conversions to Boolean

Legal conversions from other data types to boolean are:

- The empty string ("") is converted to `false`. All other strings are converted to `true`.
- An integer value 0 is converted to `false`. All other integer numbers are converted to `true`.
- A floating-point value 0.0 is converted to `false`. All other floating-point numbers are converted to `true`.
- `Invalid` can not be converted to a boolean value.

6.8.6 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` (see the operators in sections 6.3.8, 6.3.9 and 6.3.10 for the exceptions to this rule).

6.8.7 Summary

The following table contains a summary of 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

Given \ Used as:	Boolean	Integer	Floating-point	String
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 (if valid – see section 7.4 for numeric string grammar for decimal integer literals) or illegal	floating-point value of its string representation (if valid – see section 7.4 for numeric string grammar for decimal floating-point literals) or illegal	-
invalid	Illegal	Illegal	Illegal	Illegal

6.9 Operator Data Type Conversion Rules

The previous conversion rules specify 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 case the type is used to specify the operation) and to perform the data type conversions needed for the selected operation. The rules are specified in the following way:

- 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 then the value is used as such.
- If the operand value does not match the required type then a conversion from the current data type to the required one is attempted:
  - *Legal conversion*: Conversion can be done only if the general conversion rules (see section 6.9) specify a *legal* conversion from the current operator data type to the required one.
  - *Illegal conversion*: Conversion can not be done if the general conversion rules (see section 6.9) do not specify a *legal* conversion from the current type to the required type.
- If a legal conversion rule is specified for the operand (unary) or for all operands then the conversion is performed, the operation performed on the converted values and the result returned as 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 then 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 given operand data types:

Operand types	**Additional conversion rules**	**Examples**

Boolean(s)	▪ If the operand is of type boolean or can be converted into a boolean value[18] 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 of type integer or can be converted into an integer value[18] 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`

Floating-point(s)	▪ If the operand is of type floating-point or can be converted into a floating-point value[18] then perform a floating-point operation and return its value, otherwise ▪ return `invalid`	-
String(s)	▪ If the operand is of type string or can be converted into a string value[18] then perform a string operation and return its value, otherwise ▪ return `invalid`	-

Integer or floating-point (unary)	▪ If the operand is of type integer or can be converted into an integer value then perform an integer operation and return its value, otherwise
	▪ if the operand is of type floating-point or can be converted into a floating-point value[18] 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`

[18] Conversion can be done if the general conversion rules (see section **Error! Reference source not found.**) specify a legal conversion from the current type to the required type.

Operand types	**Additional conversion rules**	**Examples**

Integers or floating-points	▪ If at least one of the operands is of type floating-point then convert the remaining operand to a floating-point value, perform a floating-point operation and return its value, otherwise
	▪ if the operands are of type integer or can be converted into integer values[18] then perform an integer operation and return its value, otherwise
	▪ if the operands can be converted into floating-point values[18] 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*"4.3" =>  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 of type string then convert the remaining operand to a string value, perform a string operation and return its value, otherwise
	▪ if at least one of the operands is of type floating-point then convert the remaining operand to a floating-point value, perform a floating-point operation and return its value, otherwise
	▪ if the operands are of type integer or can be converted into integer values[18] 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`

6.10 Summary of Operators and Conversions

The following sections contain a summary on how the conversion rules are applied to WMLScript operators and what are their possible return value types.

6.10.1 Single-Typed Operators

Operators that accept operands of one specific type use the general conversion rules directly. The following list contains all single type WMLScript operators:

Operator	**Operand types**	**Result type**[19]	**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

6.10.2 Multi-Typed Operators

The following sections contain the operators that accept multi-typed operands:

Operator	**Operand types**	**Result type**[20]	**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

19 All operators may have an invalid result type.

20 All operators (unless otherwise stated) may have an invalid result type.

Operator	Operand types	Result type[20]	Operation performed
<, <=	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[21]	return internal data type (unary)
isvalid	any	boolean[21]	check for validity (unary)
? :	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

[21] Operator does not generate an invalid result type.

7. WMLSCRIPT GRAMMAR

The grammars used in this specification are based on [ECMA262]. Since WMLScript is not compliant with ECMAScript, the standard has been used only as the basis for defining WMLScript language.

7.1 Context-Free Grammars

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

7.1.1 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.

7.1.2 Lexical Grammar

A *lexical grammar* for WMLScript is given in section 7.2. This grammar has as its terminal symbols the characters of the Universal Character set of ISO/IEC-10646 ([ISO10646]). 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.

Input elements other than white space and comments form the terminal symbols for the syntactic grammar for WMLScript and are called WMLScript *tokens*. These tokens are the reserved words, identifiers, literals and punctuators of the WMLScript language. Simple white space and single-line comments are simply 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; but 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.

7.1.3 Syntactic Grammar

The *syntactic grammar* for WMLScript is given in section 7.3. This grammar has 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.

When a stream of Unicode characters is to be parsed as a WMLScript, it is first converted to a stream of input elements by repeated application of the lexical grammar; this stream of input

elements is then parsed by a single application of the syntax grammar. The program is syntactically in error if the tokens in the stream of input elements cannot be parsed as a single instance of the goal nonterminal *CompilationUnit*, with no tokens left over.

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

7.1.4 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 having to do with numeric literals and has as its terminal symbols the characters of the Unicode character set. This grammar appears in section 7.4.

Productions of the numeric string grammar are distinguished by having three colons "**:::**" as punctuation.

7.1.5 Grammar Notation

Terminal symbols of the lexical and string grammars and some of the terminal symbols of the syntactic grammar, are shown in **`fixed width`** font, both in the productions of the grammars and throughout this specification whenever the text directly refers to such a terminal symbol. These are to appear in a program exactly as written.

Nonterminal symbols are shown in *italic* type. 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 to which grammar the production belongs.) 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 to say, 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 suffix "*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 to be excluded. 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

7.1.6 Source Text

WMLScript source text is represented as a sequence of characters representable using the Universal Character set of ISO/IEC-10646 ([ISO10646]). Currently, this character set is identical to Unicode 2.0 ([UNICODE]). Within this document, the terms ISO10646 and Unicode are used interchangeably and will indicate the same document character set.

SourceCharacter **::**

any Unicode character

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

Every WMLScript program can be represented using only 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 effectively ignored as part of the comment. Within a string literal, the Unicode escape sequence contributes one character to the string value of the literal.

7.1.7 Character Set Resolution

When a WMLScript document is accompanied by external information (e.g. HTTP or MIME) there may be multiple sources of information available to determine the character encoding. In this case, their relative priority and the preferred method of handling conflict should be specified as part of the higher-level protocol. See, for example, the documentation of the "`text/vnd.wap.wmlscript`" and "`application/vnd.wap.wmlscriptc`" MIME media types.

The pragma `meta http equiv` (see section 6.7.3.2), if present in the document, is never used to determine the character encoding.

If a WMLScript document is transformed into a different format - for example, into the WMLScript bytecode (see section 9.2) - then the rules relevant for that format are used to determine the character encoding.

7.2 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
Identifier
Punctuator
Literal

ReservedWord **::**
Keyword
KeywordNotUsedByWMLScript
FutureReservedWord
BooleanLiteral
InvalidLiteral

Keyword **:: one of**

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

KeywordNotUsedByWMLScript **:: one of**

delete	**null**
in	**this**
lib	**void**
new	**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**[22]

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

Punctuator **:: one of**[23]

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

Literal **::**[24]

InvalidLiteral
BooleanLiteral
NumericLiteral
StringLiteral

InvalidLiteral **::**[25]

invalid

BooleanLiteral **::**[26]

true
false

NumericLiteral **::**

DecimalIntegerLiteral
HexIntegerLiteral
OctalIntegerLiteral
DecimalFloatLiteral

DecimalIntegerLiteral **::**

0
NonZeroDigit DecimalDigits$_{opt}$

NonZeroDigit **:: one of**

1 2 3 4 5 6 7 8 9

---

22 Compatibility note: ECMAScript supports the usage of dollar sign ($) in identifier names, too.

23 Compatibility note: ECMAScript supports arrays and square brackets ([ ]), too.

24 Compatibility note: ECMAScript supports *Null* literal, too.

25 Compatibility note: ECMAScript does not support *invalid*.

26 Compatibility note: ECMAScript supports both lower and upper case boolean literals.

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

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*

7.3 WMLScript Syntactic Grammar

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

PrimaryExpression **:**[27]
Identifier
Literal
(*Expression* **)**

CallExpression **:** [28]
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 **--**

---

27 Compatibility note: ECMAScript supports objects and *this*, too.

28 Compatibility note: ECMAScript support for arrays ([]) and object allocation (*new*) removed. *MemberExpression* is used for specifying library functions, e.g. `String.length("abc")`, not for accessing members of an object.

UnaryExpression **:**[29]

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

MultiplicativeExpression **:**[30]

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*

---

29 Compatibility note: ECMAScript operators *delete* and *void* are not supported. *parseInt* and *parseFloat* are supported as library functions. ECMAScipt does not support operator *isvalid*.

30 Compatibility note: Integer division (*div*) is not supported by ECMAScript.

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 **:**[31]
Block
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement

Block **:**
{ $StatementList_{opt}$ **}**

StatementList **:**
Statement
StatementList Statement

31 Compatibility note: ECMAScript *with* statement is not supported.

VariableStatement **:**
var *VariableDeclarationList* **;**

VariableDeclarationList **:**
VariableDeclaration
VariableDeclarationList **,** *VariableDeclaration*

VariableDeclaration **:**
Identifier VariableInitializer$_{opt}$

VariableInitializer **:**
= *ConditionalExpression*

EmptyStatement **:**
;

ExpressionStatement **:**
Expression **;**

IfStatement **:**[32]
if (*Expression* **)** *Statement* **else** *Statement*
if (*Expression* **)** *Statement*

IterationStatement **:**[33]
WhileStatement
ForStatement

WhileStatement **:**
while (*Expression* **)** *Statement*

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

ContinueStatement **:**[34]
continue ;

BreakStatement **:**[35]
break ;

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

---

32 *else* is always tied to the closest *if.*

33 Compatibility note: ECMAScript *for in* statement is not supported.

34 Continue statement can only be used inside a *while* or a *for* statement.

35 Break statement can only be used inside a *while* or a *for* statement.

FunctionDeclaration **:**[36]
extern$_{opt}$ **function** *Identifier* **(** *FormalParameterList*$_{opt}$ **)** *Block* **;**$_{opt}$

FormalParameterList **:**
Identifier
FormalParameterList **,** *Identifier*

CompilationUnit **:**
Pragmas$_{opt}$ *FunctionDeclarations*

Pragmas **:**[37]
Pragma
Pragmas Pragma

Pragma **:**
use *PragmaDeclaration* **;**

PragmaDeclaration **:**
ExternalCompilationUnitPragma
AccessControlPragma
MetaPragma

ExternalCompilationUnitPragma **:**
url *Identifier StringLiteral*

AccessControlPragma **:**[38]
access *AccessControlSpecifier*

AccessControlSpecifier **:**
domain *StringLiteral*
path *StringLiteral*
domain *StringLiteral* **path** *StringLiteral*

MetaPragma **:**
meta *MetaSpecifier*

MetaSpecifier **:**
MetaName
MetaHttpEquiv
MetaUserAgent

MetaName **:**
name *MetaBody*

---

36 Compatibility note: ECMAScript does not support keyword *extern*.

37 Compatibility note: ECMAScript does not support *pragmas*.

38 Compilation unit can contain only one *access control* pragma.

MetaHttpEquiv **:**
`http equiv` *MetaBody*

MetaUserAgent **:**
`user agent` *MetaBody*

MetaBody **:**
MetaPropertyName MetaContent $MetaScheme_{opt}$

MetaPropertyName **:**
StringLiteral

MetaContent **:**
StringLiteral

MetaScheme **:**
StringLiteral

FunctionDeclarations **:**
FunctionDeclaration
FunctionDeclarations FunctionDeclaration

7.4 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. This grammar is similar to the part of the lexical grammar having to do with numeric literals and has as its terminal symbols the characters of the US-ASCII character set.

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 *StringDecimalFloatingPointLiteral*.

StringDecimalIntegerLiteral **:::**
$StrWhiteSpace_{opt}$ *StrSignedDecimalIntegerLiteral* $StrWhiteSpace_{opt}$

StringDecimalFloatingPointLiteral **:::**
$StrWhiteSpace_{opt}$ *StrSignedDecimalIntegerLiteral* $StrWhiteSpace_{opt}$
$StrWhiteSpace_{opt}$ *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*

8. WMLSCRIPT BYTECODE INTERPRETER

The textual format of WMLScript language must be compiled into a binary format before it can be interpreted by the WMLScript bytecode interpreter. *WMLScript compiler* encodes one WMLScript compilation unit into WMLScript bytecode using the encoding format presented in the chapter 9. A WMLScript *compilation unit* (see section 7.1.3) is a unit containing pragmas and any number of WMLScript functions. WMLScript compiler takes one compilation unit as input and generates the WMLScript bytecode as its output.

8.1 Interpreter Architecture

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

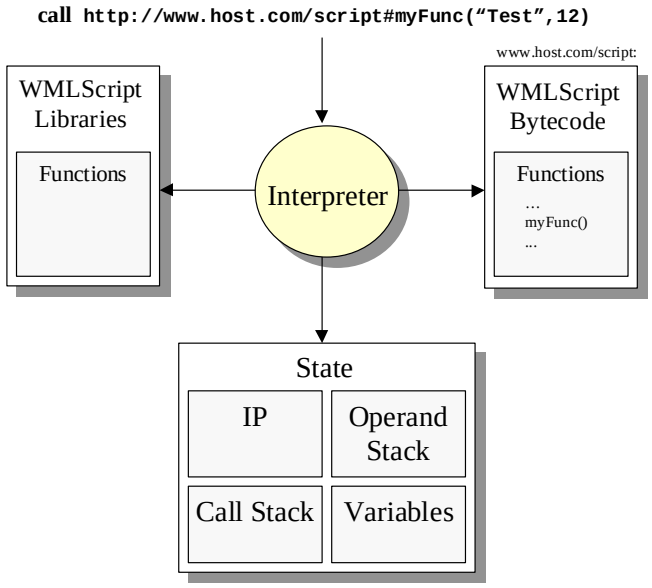

Figure 1: General architecture of the WMLScript interpreter

The WMLScript interpreter can be used 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 the *return value* back to the caller.

Execution of a WMLScript function means 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*: It is used for expression evaluation and passing arguments to called functions and back to the caller.
- *Function call stack*: WMLScript function can call other functions in the current or separate compilation unit or make calls to library functions. The function call stack maintains the information about functions and their return addresses.

8.2 Character Set

The WMLScript Interpreter must use only one character set (*native character set*) for all of its string operations. 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()* (see [WMLSLibs])

8.3 WMLScript and URLs

The World Wide Web is a network of information and devices. Three areas of specification ensure widespread interoperability:

- A *unified naming model*. Naming is implemented with Uniform Resource Locators (URLs), which provide standard way to name any network resource. See [RFC2396].
- Standard *protocols* to transport information (e.g. HTTP).
- Standard *content types* (e.g. HTML, WMLScript).

WMLScript assumes the same reference architecture as HTML and the World Wide Web. WMLScript compilation unit is named using URLs and can be fetched over standard protocols that have HTTP semantics, such as [WSP]. URLs are defined in [RFC2396]. The character set used to specify URLs is also defined in [RFC2396].

In WMLScript, URLs are used in the following situations:

- When a user agent wants to make a WMLScript call (see section 8.3.4)
- When specifying *external compilation units* (see section 6.7.1)
- When specifying *access control* information (see section 6.7.2)

8.3.1 URL Schemes

A WMLScript interpreter must implement the URL schemes specified in [WAE].

8.3.2 Fragment Anchors

WMLScript has adopted the HTML de facto standard 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.

8.3.3 URL Call Syntax

This section contains the grammar for specifying the syntactic structure of the URL call. This grammar is similar to the part of the WMLScript lexical and syntactic grammars having to do with function calls and literals and has as its terminal symbols the characters of the US-ASCII character set.

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

Only the syntax for the fragment anchor (#) is specified (see [RFC2396] for more information about URL syntax).

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 **:::**
$WhiteSpaces_{opt}$ *Literal* $WhiteSpaces_{opt}$

WhiteSpaces **:**
WhiteSpace
White Spaces WhiteSpace

WhiteSpace **:::**
<TAB>
<VT>
<FF>
<SP>
<LF>
<CR>

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* **"**

SingleStringCharacter **:::**
SourceCharacter **but not** *single-quote* **'**

8.3.4 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 (e.g. `http://www.x.com/myScripts.scr`)
- Function name and parameters as the fragment anchor (e.g. `testFunc('Test%20argument',-8)`)

The final URL with the fragment is:

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

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

- Access control checks are performed (see section 6.7.2). The call fails if the caller does not have rights to call the compilation unit.
- 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.
- The parameter list in the fragment anchor (see section 8.3.2) is parsed and the given arguments with their appropriate types (string literals as string data types, integer literals as integer data types etc.) are passed to the function. The call fails if the parameter list has an invalid syntax.

8.3.5 Character Escaping

URL calls can use URL escaping (see [RFC2396]) and any other escaping mechanism provided by the content format containing the URL call to specify the URL. However, the URL Call Syntax is applied to the URL fragment only after it has been properly unescaped.

8.3.6 Relative URLs

WMLScript has adopted the use of relative URLs, as specified in [RFC2396]. [RFC2396] specifies the method used to resolve relative URLs in the context of a WMLScript compilation unit. The base URL of a WMLScript compilation unit is the URL that identifies the compilation unit.

8.4 Bytecode Semantics

The following sections describe the general encoding rules that must be used to generate WMLScript bytecode. These rules specify what the WMLScript compiler can assume from the behaviour of the WMLScript interpreter.

8.4.1 Passing of Function Arguments

Arguments must be present in the operand stack in the same order as they are presented in a WMLScript function declaration at the time of a WMLScript or library function call. Thus, the first argument is pushed into the operand stack first, the second argument is pushed next, etc. The instruction executing the call must pop the arguments from the operand stack and use them to initialise the appropriate function variables.

8.4.2 Allocation of Variable Indexes

A WMLScript function refers to variables by using unique variable indexes. These indexes must match with the information specified for each called WMLScript function: the *number of arguments* the function accepts and *the number of local variables* used by the function. Thus, the variable index allocation must be done using the following rules:

1) *Function Arguments*: Indexes for function arguments must be allocated first. The allocation must be done in the same order as the arguments are pushed into the operand stack (0 is allocated for the first argument, 1 for the second argument, etc.). The number of indexes allocated for function arguments must match the number of arguments accepted by the function. Thus, if the function accepts *N* arguments then the last variable index must be *N-1*. If the function does not accept any arguments (*N = 0*) then no variable indexes are allocated.

2) *Local variables*: Indexes for local variables must be allocated subsequently from the first variable index (*N*) that is not used for function arguments. The number of indexes allocated for local variables must match the number of local variables used by the function.

8.4.3 Automatic Function Return Value

WMLScript function must return an empty string in case the end of the function is encountered without a return statement. The compiler can rely on the WMLScript interpreter to automatically return an empty string every time the interpreter reaches the end of the function without encountering a return instruction.

8.4.4 Initialisation of Variables

The WMLScript compiler should rely on the WMLScript interpreter to initialise all function local variables initially to an empty string. Thus, the compiler does not have to generate initialization code for variables declared without initialisation.

8.5 Access Control

WMLScript provides two mechanisms for controlling the access to the functions in the WMLScript compilation unit: *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 (see section 6.4.2.2).
- *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 (see section 6.7.2).

9. WMLSCRIPT BINARY FORMAT

The following sections contain the specifications for the WMLScript bytecode, a compact binary representation for compiled WMLScript functions. The format was designed to allow for compact transmission over narrowband channels, with no loss of functionality or semantic information.

9.1 Conventions

The following sections describe the general encoding conventions and data types used to generate WMLScript bytecode.

9.1.1 Used Data Types

The following data types are used in the specification of the WMLScript Bytecode:

Data Type	Definition
bit	1 bit of data
byte	8 bits of opaque data
int8	8 bit signed integer (two's complement encoding)
u_int8	8 bit unsigned integer
int16	16 bit signed integer (two's complement encoding)
u_int16	16 bit unsigned integer
mb_u_int16	16 bit unsigned integer, in multi-byte integer format. See section 9.1.2 for more information.
int32	32 bit signed integer (two's complement encoding)
u_int32	32 bit unsigned integer
mb_u_int32	32 bit unsigned integer, in multi-byte integer format. See section 9.1.2 for more information.
float32	32 bit signed floating-point value in ANSI/IEEE Std 754-1985 [IEEE754] format.

Network byte order for multi-byte integer values is "big-endian". In other words, the most significant byte is transmitted on the network first followed subsequently by the less significant bytes. Network bit ordering for bit fields within a byte is "big-endian". In other words, bit fields described first are placed in the most significant bits of the byte.

9.1.2 Multi-byte Integer Format

This encoding uses a multi-byte representation for integer values. A multi-byte integer consists of a series of octets, where the most significant bit is the *continuation flag* and the remaining seven bits are a scalar value. The continuation flag is used to indicate that an octet is not the end of the multi-byte sequence. A single integer value is encoded into a sequence of N octets. The first N-1 octets have the continuation flag set to a value of one (1). The final octet in the series has a continuation flag value of zero.

The remaining seven bits in each octet are encoded in a big-endian order, e.g., most significant bit first. The octets are arranged in a big-endian order, e.g. the most significant seven bits are transmitted first. In the situation where the initial octet has less than seven bits of value, all unused bits must be set to zero (0).

For example, the integer value `0xA0` would be encoded with the two-byte sequence `0x81 0x20`. The integer value `0x60` would be encoded with the one-byte sequence `0x60`.

9.1.3 Character Encoding

WMLScript bytecode supports the following character encoding:

- UTF-8 (see [RFC2279])

Other character sets and their encodings are supported by a *special string type* (string with external character encoding definition, see section 9.4.1.3.3) that does not explicitly specify the used character set or its encoding but assumes that this information is provided as part of the compilation unit itself (constant pool). The following rules must be applied when defining the used character encoding for these special strings:

- If the value of the character set number in the constant pool is non-zero then this number defines the used character encoding (the number denotes the MIBEnum value assigned by the IANA for all character sets).
- If the value of the character set number in the constant pool is zero (0) then the character set is unknown.

The compiler must select one of these encodings to encode character strings in the WMLScript bytecode.

WMLScript language constructs, such as *function names* in WMLScript, are written by using only a subset of Unicode character set i.e, a subset of US-ASCII characters. Thus, function names in the WMLScript bytecode must use a fixed UTF-8 encoding.

9.1.4 Notational Conventions

WMLScript bytecode is a set of bytes that represent WMLScript functions in a binary format. It contains all the information needed by the WMLScript interpreter to execute the encoded functions as specified. The bytecode can be divided into sections and subsections each of which containing a binary representation of a logical WMLScript unit.

The WMLScript bytecode structure and content is presented using the following table based notation:

Name	**Data type and size**	**Comment**
This is a *name* of a section inside the bytecode.	This specifies a *data type* and its *size* reserved for a section in case it cannot be divided into smaller subsections. Subsection specification is given in a separate table. Reference to the table is provided.	This gives a general overview of the meaning of this section.

Name	Data type and size	Comment
The name of the next section. Any number of sections can be presented in one table.		
...		

The following conventions apply:

- Sections of bytecode are represented as rows in a table.
- Each section may be divided into subsections and represented in separate tables. In such case a reference to the subsection table is provided.
- Repetitive sections are denoted by section name followed by three dots (...).

9.2 WMLScript Bytecode

The WMLScript encoding contains two major elements: constant literals and the constructs needed to describe the behaviour of each WMLScript function. Thus, the WMLScript bytecode consists of the following sections:

Name	Data type and size	Comment
HeaderInfo	See section 9.3	Contains general information related to the bytecode.
ConstantPool	See section 9.4	Contains the information of all constants specified as part of the WMLScript compilation unit that are encoded into bytecode.
PragmaPool	See section 9.5	Contains the information related to pragmas specified as part of the WMLScript compilation unit that are encoded into bytecode.
FunctionPool	See section 9.6	Contains all the information related to the encoding of functions and their behaviour.

The following sections define the encoding of these sections and their subsections in detail.

9.3 Bytecode Header

The header of the WMLScript bytecode contains the following information:

Name	Data type and size	Comment
VersionNumber	byte	Version number of the WMLScript bytecode. The version byte contains the *major version* minus one in the upper 4 bits and the *minor version* in the lower 4 bits. The current version is 1.1. Thus, the version number must be encoded as `0x01`.
CodeSize	mb_u_int32	The size of the rest of the bytecode (not including the version number and this variable) in bytes

9.4 Constant Pool

Constant pool contains all the constants used by the WMLScript functions. Each of the constants has an index number starting from zero that is defined by its position in the list of constants. The instructions use this index to refer to specific constants.

Name	Data type and size	Comment
NumberOfConstants	mb_u_int16	Specifies how many constants are encoded in this pool.
CharacterSet	mb_u_int16	Specifies the character set used by the string constants in the constant pool. The character set is specified as an integer that denotes a MIBEnum value assigned by the IANA for all character sets (see [WSP] for more information).
Constants...	See section 9.4.1	Contains the definitions for each constant in the constant pool. The number of constants is specified by NumberOfConstants.

9.4.1 Constants

Constants are stored into the bytecode one after each other. Encoding of each constant starts with the definition of its type (integer, floating-point, string etc.). It is being followed by constant type specific data that represents the actual value of the constant:

Name	Data type and size	Comment
ConstantType	u_int8	The type of the constant.
ConstantValue	See sections 9.4.1.1, 9.4.1.2 and 9.4.1.3	Type specific value definition.

The following encoding for constant types is used:

Code	Type	Encoding
0	8 bit signed integer	9.4.1.1.1
1	16 bit signed integer	9.4.1.1.2
2	32 bit signed integer	9.4.1.1.3
3	32 bit signed floating-point	9.4.1.2
4	UTF-8 String	9.4.1.3.1
5	Empty String	9.4.1.3.2
6	String with external character encoding definition	9.4.1.3.3
7-255	Reserved for future use	

9.4.1.1 Integers

WMLScript bytecode supports 8 bit, 16 bit and 32 bit signed integer constants. The compiler can optimise the WMLScript bytecode size by selecting the smallest integer constant type that can still hold the integer constant value.

9.4.1.1.1 8 Bit Signed Integer

8 bit signed integer constants are represented in the following format:

Name	Data type and size	Comment
ConstantInteger8	int8	The value of the 8 bit signed integer constant.

9.4.1.1.2 16 Bit Signed Integer

16 bit signed integer constants are represented in the following format:

Name	Data type and size	Comment
ConstantInteger16	int16	The value of the 16 bit signed integer constant.

9.4.1.1.3 32 Bit Signed Integer

32 bit signed integer constants are represented in the following format:

Name	Data type and size	Comment
ConstantInteger32	int32	The value of the 32 bit signed integer constant.

9.4.1.2 Floats

Floating-point constants are represented in 32-bit ANSI/IEEE Std 754-1985 [IEEE754] format:

Name	Data type and size	Comment
ConstantFloat32	float32	The value of the 32 bit floating point constant.

9.4.1.3 Strings

WMLScript bytecode supports several ways to encode string constants[39] into the constant pool. The compiler can select the most suitable character encoding supported by the client and optimise the WMLScript bytecode size by selecting the smallest string constant type that can still hold the string constant value.

9.4.1.3.1 UTF-8 Strings

Strings that use UTF-8 encoding are encoded into the bytecode by first specifying their length and then the content:

Name	Data type and size	Comment
StringSizeUTF8	mb_u_int32	The size of the following string in bytes (not containing this variable).
ConstantStringUTF8	StringSizeUTF8 bytes	The value of the Unicode string (non-null terminated) constant encoded using UTF-8. See 9.1.3 for more information about transfer encoding of strings.

9.4.1.3.2 Empty Strings

Empty strings do not need any additional encoding for their value.

9.4.1.3.3 Strings with External Character Encoding Definition

Strings that use external character encoding definition are encoded into the bytecode by first specifying their length and then the content:

Name	Data type and size	Comment
StringSizeExt	mb_u_int32	The size of the following string in bytes (not containing this field).
ConstantStringExt	StringSizeExt bytes	The value of the string (non-null terminated) constant using external character encoding definition. See 9.1.3 for more information about transfer encoding of strings.

[39] Note that string constants can contain embedded null characters.

9.5 Pragma Pool

The pragma pool contains the information for pragmas defined in the compiled compilation unit.

Name	Data type and size	Comment
NumberOfPragmas	mb_u_int16	The number of pragmas.
Pragmas…	See 9.5.1	Contains the definitions for each pragma in the pragma pool. The number of pragmas is specified by NumberOfPragmas.

9.5.1 Pragmas

Pragmas are stored into the bytecode one after each other. Encoding of each pragma starts with the definition of its type. It is being followed by pragma type specific data that represents the actual value of the pragma:

Name	Data type and size	Comment
PragmaType	u_int8	The type of the pragma following pragma value.
PragmaValue	See sections 9.5.1.1 and 9.5.1.2	Pragma type specific value definition.

The following encoding for pragma types is used:

Code	Type	Encoding
0	Access Domain	9.5.1.1.1
1	Access Path	9.5.1.1.2
2	User Agent Property	9.5.1.2.1
3	User Agent Property and Scheme	9.5.1.2.2
4-255	Reserved for future use	

9.5.1.1 Access Control Pragmas

Access control information is encoded into the bytecode using two different pragma types: *access domain* and *access path*. The pragma pool can contain only one entry for each access control pragma type.

9.5.1.1.1 Access Domain

This pragma specifies the access domain to be used for the access control.

Name	Data type and size	Comment
AccessDomainIndex	mb_u_int16	Constant pool index to a string constant containing the value of the access domain. The referred constant type must be between 4 and 6.

9.5.1.1.2 Access Path

This pragma specifies the access path to be used for access control.

Name	Data type and size	Comment
AccessPathIndex	mb_u_int16	Constant pool index to a string constant containing the value of the access path. The referred constant type must be between 4 and 6.

9.5.1.2 Meta-Information Pragmas

These pragmas contain meta-information that is mean for the WMLScript interpreter. Meta-information contains following entities: name, content and scheme (optional)

9.5.1.2.1 User Agent Property

User agent properties are encoded by first specifying their name and then their value as indexes to the constant pool:

Name	Data type and size	Comment
PropertyNameIndex	mb_u_int16	Constant pool index to a string constant (constant types 4 to 6) containing the property name.
ContentIndex	mb_u_int16	Constant pool index to a string constant (constant types 4 to 6) containing the property value.

9.5.1.2.2 User Agent Property and Scheme

This pragma is encoded by specifying the property name, the value and the additional scheme:

Name	Data type and size	Comment
PropertyNameIndex	mb_u_int16	Constant pool index to a string constant (constant types 4 to 6) containing the property name.
ContentIndex	mb_u_int16	Constant pool index to a string constant (constant types 4 to 6) containing the property value.
SchemeIndex	mb_u_int16	Constant pool index to a string constant (constant types 4 to 6) containing the property schema.

9.6 Function Pool

The function pool contains the function definitions. Each of the functions has an index number starting from zero that is defined by its position in the list of functions. The instructions use this index to refer to specific functions.

Name	Data type and size	Comment
NumberOfFunctions	u_int8	The number of functions specified in this function pool.
FunctionNameTable	See section 9.6.1	Function name table contains the names of all external functions present in the bytecode.
Functions...	See section 9.6.2	Contains the bytecode for each function.

9.6.1 Function Name Table

The names of the functions that are specified as *external* (`extern`) are stored into a function name table. The names must be presented in the same order as the functions are represented in the function pool. Functions that are not specified as external are not represented in the function name table. The format of the table is the following:

Name	Data type and size	Comment
NumberOfFunctionNames	u_int8	The number of function names stored into the following table.
FunctionNames...	See section 9.6.1.1	Each external function name represented in the same order as the functions are stored into the function pool.

9.6.1.1 Function Names

Function name is provided only for functions that are specified as *external* in WMLScript. Each name is represented in the following manner:

Name	Data type and size	Comment
FunctionIndex	u_int8	The index of the function for which the following name is provided.
FunctionNameSize	u_int8	The size of the following function name in bytes (not including this variable).
FunctionName	FunctionNameSize bytes	The characters of the function name encoded by using UTF-8. See section 9.1.3 for more information about function name encoding.

9.6.2 Functions

Each function is defined by its prologue and code array:

Name	**Data type and size**	**Comment**
NumberOfArguments	u_int8	The number of arguments accepted by the function.
NumberOfLocalVariables	u_int8	The number of local variables used by the function (not including arguments).
FunctionSize	mb_u_int32	Size of the following CodeArray (not including this variable) in bytes.
CodeArray	See section 9.6.2.1	Contains the code of the function.

9.6.2.1 Code Array

Code array contains all instructions that are needed to implement the behaviour of a WMLScript function. See 10 for more information about WMLScript instruction set.

Name	**Data type and size**	**Comment**
Instructions…	See section 10	The encoded instructions.

9.7 Limitations

The following table contains the limitations inherent in the selected bytecode format and instructions:

Maximum size of the bytecode	4294967295 bytes
Maximum number of constants in the constant pool	65535
Maximum number of different constant types	256
Maximum size of a constant string	4294967295 bytes
Maximum size of a constant URL	4294967295 bytes
Maximum length of function name	255
Maximum number of different pragma types	256
Maximum number of pragmas in the pragma pool	65536
Maximum number of functions in the function pool	255
Maximum number of function parameters	255
Maximum number of local variables / function	255
Maximum number of local variables and function parameters	256
Maximum number of libraries	65536
Maximum number of functions / library	256

10. WMLSCRIPT INSTRUCTION SET

The WMLScript instruction set specifies a set of assembly level instructions that must be used to encode all WMLScript language constructs and operations. These instructions are defined in such a way that they are easy to implement efficiently on a variety of platforms.

10.1 Conversion Rules

The following table contains a summary of the conversion rules specified for the WMLScript interpreter:

Rule – Operand type(s)	**Conversions**
1 – Boolean(s)	See the conversion rules for *Boolean(s)* in section *Operator Data Type* Conversion Rules (6.9)
2 – Integer(s)	See the conversion rules for *Integer(s)* in section *Operator Data Type* Conversion Rules (6.9)
3 – Floating-point(s)	See the conversion rules for *Floating-point(s)* in section *Operator Data Type* Conversion Rules (6.9)
4 – String(s)	See the conversion rules for *String(s)* in section *Operator Data Type* Conversion Rules (6.9)
5 – Integer or floating-point (unary)	See the conversion rules for *Integer or floating-point (unary)* in section *Operator Data Type* Conversion Rules (6.9)
6 – Integers or floating-points	See the conversion rules for *Integers or floating-points* in section *Operator Data Type* Conversion Rules (6.9)
7 – Integers, floating-points or strings	See the conversion rules for *Integers, floating-points or strings* in section *Operator Data Type* Conversion Rules (6.9)
8 - Any	See the conversion rules for *Any* in section *Operator Data Type* Conversion Rules (6.9)

10.2 Fatal Errors

The following table contains a summary of the fatal errors specified for the WMLScript interpreter:

Error code:	**Fatal Error:**
1 (Verification Failed)	See section *Verification Failed* (12.3.1.1) for details
2 (Fatal Library Function Error)	See section *Fatal Library Function Error* (12.3.1.2) for details
3 (Invalid Function Arguments)	See section *Invalid Function Arguments* (12.3.1.3) for details
4 (External Function Not Found)	See section *External Function Not Found* (12.3.1.4) for details
5 (Unable to Load Compilation Unit)	See section *Unable to Load Compilation Unit* (12.3.1.5) for details
6 (Access Violation)	See section *Access Violation* (12.3.1.6) for details
7 (Stack Underflow)	See section *Stack Underflow* (12.3.1.7) for details
8 (Programmed Abort)	See section *Programmed Abort* (12.3.2.1) for details
9 (Stack Overflow)	See section *Stack Overflow* (12.3.3.1) for details*
10 (Out of Memory)	See section *Out of Memory* (12.3.3.2) for details*
11 (User Initiated)	See section *User Initiated* (12.3.4.1) for details*
12 (System Initiated)	See section *System Initiated* (12.3.4.2) for details*

* These fatal errors are not related to computation but can be generated as a result of memory exhaustion or external signals.

10.3 Optimisations

WMLScript instruction set has been defined so that it provide at least the minimal set of instructions by which WMLScript language operations can be presented. Since the WMLScript bytecode is being transferred from the gateway to the client through a narrowband connection, the selected instructions have been optimised so that the compilers can generate code of minimal size. In some

cases, this has meant that several instructions with different parameters have been introduced to perform the same operation. The compiler should use the one that generates optimal code.

Inline parameters have been used to optimally pack information into as few bytes as possible. The following inline parameter optimisations have been introduced:

Signature	**Available instructions**	**Used for**
`1XXPPPP`	4	JUMP_FW_S, JUMP_BW_S, TJUMP_FW_S, LOAD_VAR_S
`010XPPP`	2	STORE_VAR_S, LOAD_CONST_S
`011XXPP`	4	CALL_S, CALL_LIB_S, INCR_VAR_S
`00XXXXX`	63	The rest of the instructions

10.4 Notational Conventions

The following sections contain the definitions of instructions in the WMLScript instruction set. For each instruction, the following information is provided:

- *Instruction*: A symbolic name given to the instruction and its parameters.
- *Opcode*: The 8-bit encoding of the instruction.
- *Parameters*: Parameter description specifying their ranges and semantics. Some instructions are optimised and can contain an *implicit parameter* as part of the encoding, ie, a set of bits from the 8 bit encoding is reserved for a parameter value.
- *Operation*: Description of the operation of the instruction, its parameters and the effects they have on the execution and the operand stack.
- *Operands*: Specifies the number of operands required by the instruction and all acceptable operand types.
- *Conversion*: Specifies the used conversion rule (see section 10.1).
- *Result*: Specifies the result and its type.
- *Operand stack*: Specifies the effect on the operand stack. It is described by using notation where the part before the arrow (=>) represents the stack before the instruction has been executed and the part after the arrow the stack after the execution.
- *Errors*: Specifies the possible fatal errors that can occur during the execution of the instruction (see section 10.2).

All instructions except the control flow instructions continue the execution at the following instruction. Control flow instructions specify the next instruction explicitly.

Fatal errors that can be encountered at any time (see section *External Exceptions* in 12.3.4 and *Memory Exhaustion Errors* in 12.3.3) are assumed to be possible with every instruction.

The result of the instruction can be an `invalid` value. This is not explicitly stated with each instruction but is assumed to be the result of the used conversion rule, a load of an invalid or unsupported floating-point constant or a result of an operation with an `invalid` operand.

10.5 Instructions

The following sections contain the descriptions of each instruction divided into subcategories.

10.5.1 Control Flow Instructions

10.5.1.1	**Instruction:** JUMP_FW_S
Opcode:	100iiiii (*iiiii* is the implicit unsigned *offset*)
Parameter:	*Offset* is an unsigned 5-bit integer in the range of 0..31.
Operation:	Jumps forward to an offset. Execution proceeds at the given *offset* from the address of the first byte following this instruction. More specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* + 1 + *offset*.
Operands:	-
Conversion:	-
Result:	-
Operand stack:	No change
Errors:	1 (Verification Failed)

10.5.1.2	**Instruction:** JUMP_FW ***offset***
Opcode:	00000001
Parameter:	*Offset* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Jumps forward to an offset. Execution proceeds at the given *offset* from the address of the first byte following this instruction. More specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* + 2 + *offset*.
Operands:	-
Conversion:	-
Operand stack:	No change
Errors:	1 (Verification Failed)

10.5.1.3	**Instruction:** JUMP_FW_W ***<offset1,offset2>***
Opcode:	00000010
Parameter:	*Offset* is an unsigned 16-bit integer <*offset1, offset2*> in the range of 0..65535.
Operation:	Jumps forward to an offset. Execution proceeds at the given *offset* from the address of the first byte following this instruction. More specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* + 3 + *offset*.
Operands:	-
Conversion:	-
Result:	-
Operand stack:	No change
Errors:	1 (Verification Failed)

10.5.1.4	**Instruction:** JUMP_BW_S
Opcode:	101iiiii (*iiiii* is the implicit unsigned *offset*)
Parameter:	*Offset* is an unsigned 5-bit integer in the range of 0..31.
Operation:	Jumps backward to an offset. Execution proceeds at the given *offset* from the address of this instruction. More specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* - *offset*.
Operands:	-
Conversion:	-
Result:	-
Operand stack:	No change
Errors:	1 (Verification Failed)

10.5.1.5	**Instruction:** JUMP_BW ***offset***
Opcode:	00000011
Parameter:	*Offset* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Jumps backward to an offset. Execution proceeds at the given *offset* from the address of this instruction. More specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* - *offset*.
Operands:	-
Conversion:	-
Result:	-
Operand stack:	No change
Errors:	1 (Verification Failed)

10.5.1.6	**Instruction:** JUMP_BW_W ***<offset1,offset2>***
Opcode:	00000100
Parameter:	*Offset* is an unsigned 16-bit integer <*offset1, offset2*> in the range of 0..65535.
Operation:	Jumps backward to an offset. Execution proceeds at the given *offset* from the address of this instruction. More specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* - *offset*.
Operands:	-
Conversion:	-
Result:	-
Operand stack:	No change
Errors:	1 (Verification Failed)

10.5.1.7	**Instruction:** TJUMP_FW_S
Opcode:	110iiiii (*iiiii* is the implicit unsigned *offset*)
Parameter:	*Offset* is an unsigned 5-bit integer in the range of 0..31.
Operation:	Pops a value from the operand stack and jumps forward to an *offset* if the value is either `false` or `invalid`. Execution proceeds at the given *offset* from the address of the first byte following this instruction (more specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* + 1 + *offset*). Otherwise, the execution continues at the next instruction.
Operand:	Any
Conversion:	1 – Boolean(s)
Result:	-
Operand stack:	…, value => …
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.1.8	**Instruction:** TJUMP_FW ***offset***
Opcode:	00000101
Parameter:	*Offset* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Pops a value from the operand stack and jumps forward to an *offset* if the value is either `false` or `invalid`. Execution proceeds at the given *offset* from the address of the first byte following this instruction (more specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* + 2 + *offset*). Otherwise, the execution continues at the next instruction.
Operand:	Any
Conversion:	1 – Boolean(s)
Result:	-
Operand stack:	…, value => …
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.1.9	**Instruction:** TJUMP_FW_W ***<offset1,offset2>***
Opcode:	00000110
Parameter:	*Offset* is an unsigned 16-bit integer <*offset1, offset2*> in the range of 0..65535.
Operation:	Pops a value from the operand stack and jumps forward to an *offset* if the value is either `false` or `invalid`. Execution proceeds at the given *offset* from the address of the first byte following this instruction (more specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* + 3 + *offset*). Otherwise, the execution continues at the next instruction.
Operand:	Any
Conversion:	1 – Boolean(s)
Result:	-
Operand stack:	…, value => …
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.1.10	**Instruction:** TJUMP_BW ***offset***
Opcode:	00000111
Parameter:	*Offset* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Pops a value from the operand stack and jumps backward to an *offset* if the value is either `false` or `invalid`. Execution proceeds at the given *offset* from the address of this instruction (more specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* - *offset*). Otherwise, the execution continues at the next instruction.
Operand:	Any
Conversion:	1 – Boolean(s)
Result:	-
Operand stack:	…, value => …
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.1.11	**Instruction:** TJUMP_BW_W ***<offset1,offset2>***
Opcode:	00001000
Parameter:	*Offset* is an unsigned 16-bit integer <*offset1, offset2*> in the range of 0..65535.
Operation:	Pops a value from the operand stack and jumps backward to an *offset* if the value is either `false` or `invalid`. Execution proceeds at the given *offset* from the address of this instruction (more specifically, if the address of this instruction is *n* and the value of the offset is *offset* then the next instruction to be executed is at address: *n* - *offset*). Otherwise, the execution continues at the next instruction.
Operand:	Any
Conversion:	1 – Boolean(s)
Result:	-
Operand stack:	…, value => …
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.2 Function Call Instructions

10.5.2.1	**Instruction:** CALL_S
Opcode:	01100iii (*iii* is the implicit *findex*)
Parameter:	*Findex* is an unsigned 3-bit integer in the range of 0..7.
Operation:	Pops function arguments from the operand stack, initialises the function variables (arguments and local variables) and calls a local function defined in the same function pool. Execution proceeds from the first instruction of the function *findex*.
Operands:	Variable number, any type
Conversion:	-
Result:	Any (function return value)
Operand stack:	…, [arg1, [arg2 …]] => …, ret-value
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.2.2	**Instruction:** CALL ***findex***
Opcode:	00001001
Parameter:	*Findex* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Pops function arguments from the operand stack, initialises the function variables (arguments and local variables) and calls a local function defined in the same function pool. Execution proceeds from the first instruction of the function *findex*.
Operands:	Variable number, any type
Conversion:	-
Result:	Any (function return value)
Operand stack:	…, [arg1, [arg2 …]] => …, ret-value
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.2.3	**Instruction:** CALL_LIB_S ***lindex***
Opcode:	01101iii (*iii* is the implicit *findex*)
Parameters:	*Findex* is an unsigned 3-bit integer in the range of 0..7. *Lindex* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Pops function arguments from the operand stack, initialises the function variables (arguments and local variables) and calls a library function *findex* defined in the specified library *lindex*.
Operands:	Variable number (specified by the called library function), any type
Conversion:	-
Result:	Any (function return value)
Operand stack:	…, [arg1, [arg2 …]] => …, ret-value
Errors:	1 (Verification Failed), 2 (Fatal Library Function Error), 7 (Stack Underflow), 8 (Programmed Abort)

10.5.2.4	**Instruction:** CALL_LIB ***findex lindex***
Opcode:	00001010
Parameters:	*Findex* is an unsigned 8-bit integer in the range of 0..255. *Lindex* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Pops function arguments from the operand stack, initialises the function variables (arguments and local variables) and calls a library function *findex* defined in the specified library *lindex*.
Operands:	Variable number (specified by the called library function), any type
Conversion:	-
Result:	Any (function return value)
Operand stack:	…, [arg1, [arg2 …]] => …, ret-value
Errors:	1 (Verification Failed), 2 (Fatal Library Function Error), 7 (Stack Underflow), 8 (Programmed Abort)

10.5.2.5	**Instruction:** CALL_LIB_W ***findex <lindex1, lindex2>***
Opcode:	00001011
Parameters:	*Findex* is an unsigned 8-bit integer in the range of 0..255. *Lindex* is an unsigned 16-bit integer <*lindex1,lindex2*> in the range of 0..65535.
Operation:	Pops function arguments from the operand stack, initialises the function variables (arguments and local variables) and calls a library function *findex* defined in the specified library *lindex*.
Operands:	Variable number (specified by the called library function), any type
Conversion:	-
Result:	Any (function return value)
Operand stack:	…, [arg1, [arg2 …]] => …, ret-value
Errors:	1 (Verification Failed), 2 (Fatal Library Function Error), 7 (Stack Underflow), 8 (Programmed Abort)

10.5.2.6	**Instruction:** CALL_URL ***urlindex findex args***
Opcode:	00001100
Parameters:	*Urlindex* is an unsigned 8-bit integer in the range of 0..255 that must point to the constant pool containing a valid URL. The referred constant type must be between 4 and 6. *Findex* is an unsigned 8-bit integer in the range of 0..255 that must point to the constant pool containing a valid function name. The referred constant type must be 4. *Args* is an unsigned 8-bit integer in the range of 0..255 that must contain the number of function arguments pushed on the operand stack.
Operation:	Pops function arguments from the operand stack, initialises the function variables (arguments and local variables) and calls a function specified by *findex* defined in the specified URL address *urlindex*.
Operands:	Variable number (specified by *args*), any type
Conversion:	-
Result:	Any (function return value)
Operand stack:	…, [arg1, [arg2 …]] => …, ret-value
Errors:	1 (Verification Failed), 3 (Invalid Function Arguments), 4 (External Function Not Found), 5 (Unable to Load Compilation Unit), 6 (Access Violation), 7 (Stack Underflow)

10.5.2.7	**Instruction:** CALL_URL_W ***<urlindex1,urlindex2> <findex1,findex2> args***
Opcode:	00001101
Parameters:	*Urlindex* is an unsigned 16-bit integer <*urlindex1,urlindex2*> in the range of 0..65535 that must point to the constant pool containing a valid URL. The referred constant type must be between 4 and 6. *Findex* is an unsigned 16-bit integer <*findex1,findex2*> in the range of 0..65535 that must point to the constant pool containing a valid function name. The referred constant type must be 4. *Args* is an unsigned integer in the range of 0..255 that must contain the number of function arguments pushed on the operand stack.
Operation:	Pops function arguments from the operand stack, initialises the function variables (arguments and local variables) and calls a function specified by *findex* defined in the specified URL address *urlindex*.
Operands:	Variable number (specified by *args*), any type
Conversion:	-
Result:	Any (function return value)
Operand stack:	…, [arg1, [arg2 …]] => …, ret-value
Errors:	1 (Verification Failed), 3 (Invalid Function Arguments), 4 (External Function Not Found), 5 (Unable to Load Compilation Unit), 6 (Access Violation), 7 (Stack Underflow)

10.5.3 Variable Access and Manipulation

10.5.3.1	**Instruction:** LOAD_VAR_S
Opcode:	111iiiii (*iiiii* is the implicit *vindex*)
Parameter:	*Vindex* is an unsigned 5-bit integer in the range of 0..31.
Operation:	Pushes the value of the variable *vindex* on the operand stack.
Operands:	-
Conversion:	-
Result:	Any (content of the variable)
Operand stack:	… => …, value
Errors:	1 (Verification Failed)

10.5.3.2	**Instructions:** LOAD_VAR ***vindex***
Opcode:	00001110
Parameter:	*Vindex* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Pushes the value of the variable *vindex* on the operand stack.
Operands:	-
Conversion:	-
Result:	Any (content of the variable)
Operand stack:	… => …, value
Errors:	1 (Verification Failed)

10.5.3.3	**Instruction:** STORE_VAR_S
Opcode:	0100iiii (*iiii* is the implicit *vindex*)
Parameter:	*Vindex* is an unsigned 4-bit integer in the range of 0..15.
Operation:	Pops the value from the operand stack and stores it into the variable *vindex*.
Operand:	Any
Conversion:	8 - Any
Result:	-
Operand stack:	…, value => …
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.3.4	**Instruction:** STORE_VAR ***vindex***
Opcode:	00001111
Parameter:	*Vindex* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Pops the value from the operand stack and stores it into the variable *vindex*.
Operand:	Any
Conversion:	8 - Any
Result:	-
Operand stack:	…, value => …
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.3.5	**Instruction:** INCR_VAR_S
Opcode:	01110iii (*iii* is the implicit *vindex*)
Parameter:	*Vindex* is an unsigned 3-bit integer in the range of 0..7.
Operation:	Increments the value of a variable *vindex* by one.
Operands:	-
Conversion:	5 – Integer or floating-point (unary)
Result:	-
Operand stack:	No change
Errors:	1 (Verification Failed)

10.5.3.6	**Instruction:** INCR_VAR ***vindex***
Opcode:	00010000
Parameter:	*Vindex* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Increments the value of a variable *vindex* by one.
Operands:	-
Conversion:	5 – Integer or floating-point (unary)
Result:	-
Operand stack:	No change
Errors:	1 (Verification Failed)

10.5.3.7	**Instruction:** DECR_VAR ***vindex***
Opcode:	00010001
Operation:	Decrements the value of a variable *vindex* by one.
Parameter:	*Vindex* is an unsigned 8-bit integer in the range of 0..255.
Operands:	-
Conversion:	5 – Integer or floating-point (unary)
Result:	-
Operand stack:	No change
Errors:	1 (Verification Failed)

10.5.4 Access To Constants

10.5.4.1	**Instruction:** LOAD_CONST_S
Opcode:	0101iiii (*iiii* is the implicit *cindex*)
Parameter:	*Cindex* is an unsigned 4-bit integer in the range of 0..15 that points to the constant pool containing the actual constant. The referred constant type must be between 0 and 6.
Operation:	Pushes the value of the constant denoted by *cindex* on the operand stack.
Operands:	-
Conversion:	-
Result:	Any (content of the constant)
Operand stack:	… => …, value
Errors:	1 (Verification Failed)

10.5.4.2	**Instruction:** LOAD_CONST ***cindex***
Opcode:	00010010
Parameter:	*Cindex* is an unsigned 8-bit integer in the range of 0..255 that points to the constant pool containing the actual constant. The referred constant type must be between 0 and 6.
Operation:	Pushes the value of the constant denoted by *cindex* on the operand stack.
Operands:	-
Conversion:	-
Result:	Any (content of the constant)
Operand stack:	… => …, value
Errors:	1 (Verification Failed)

10.5.4.3	**Instruction:** LOAD_CONST_W ***<cindex1,cindex2>***
Opcode:	00010011
Parameter:	*Cindex* is an unsigned 16-bit integer <*cindex1,cindex2*> in the range of 0..65535 that points to the constant pool containing the actual constant. The referred constant type must be between 0 and 6.
Operation:	Pushes the value of the constant *cindex* on the operand stack.
Operands:	-
Conversion:	-
Result:	Any (content of the constant)
Operand stack:	… => …, value
Errors:	1 (Verification Failed)

10.5.4.4	**Instruction:** CONST_0
Opcode:	00010100
Parameters:	-
Operation:	Pushes an integer value 0 on the operand stack.
Operands:	-
Conversion:	-
Result:	Integer
Operand stack:	… => …, value_0
Errors:	-

10.5.4.5	**Instruction:** CONST_1
Opcode:	00010101
Parameters:	-
Operation:	Pushes an integer value 1 on the operand stack.
Operands:	-
Conversion:	-
Result:	Integer
Operand stack:	… => …, value_1
Errors:	-

10.5.4.6	**Instruction:** CONST_M1
Opcode:	00010110
Parameters:	-
Operation:	Pushes an integer value –1 on the operand stack.
Operands:	-
Conversion:	-
Result:	Integer
Operand stack:	… => …, value_-1
Errors:	-

10.5.4.7	**Instruction:** CONST_ES
Opcode:	00010111
Parameters:	-
Operation:	Pushes an empty string on the operand stack.
Operands:	-
Conversion:	-
Result:	String
Operand stack:	… => …, value_""
Errors:	-

10.5.4.8	**Instruction:** CONST_INVALID
Opcode:	00011000
Parameters:	-
Operation:	Pushes an `invalid` value on the operand stack.
Operands:	-
Conversion:	-
Result:	Invalid
Operand stack:	… => …, value_invalid
Errors:	-

10.5.4.9	**Instruction:** CONST_TRUE
Opcode:	00011001
Parameters:	-
Operation:	Pushes a boolean value `true` on the operand stack.
Operands:	-
Conversion:	-
Result:	Boolean
Operand stack:	… => …, value_true
Errors:	-

10.5.4.10	**Instruction:** CONST_FALSE
Opcode:	00011010
Parameters:	-
Operation:	Pushes a boolean value `false` on the operand stack.
Operands:	-
Conversion:	-
Result:	Boolean
Operand stack:	… => …, value_false
Errors:	-

10.5.5 Arithmetic Instructions

10.5.5.1	**Instruction:** INCR
Opcode:	00011011
Parameters:	-
Operation:	Increments the value on the top of the operand stack by one.
Operand:	Integer or floating-point
Conversion:	5 – Integer or floating-point (unary)
Result:	Integer or floating-point (incremented by one)
Operand stack:	…, value => …, value+1
Errors:	7 (Stack Underflow)

10.5.5.2	**Instruction:** DECR
Opcode:	00011100
Parameters:	-
Operation:	Decrements the value on the top of the operand stack by one.
Operand:	Integer or floating-point
Conversion:	5 – Integer or floating-point (unary)
Result:	Integer or floating-point (decremented by one)
Operand stack:	…, value => …, value-1
Errors:	7 (Stack Underflow)

10.5.5.3	**Instruction:** ADD_ASG ***vindex***
Opcode:	00011101
Parameter:	*Vindex* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Pops a value from the operand stack and adds the value to the variable *vindex*.
Operands:	Integers, floating-points or strings
Conversion:	7 – Integers, floating-points or strings
Result:	*For integers or floating-points*: variable containing the result of the addition *For strings*: variable containing the result of string concatenation
Operand stack:	…, value => …
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.5.4	**Instruction:** SUB_ASG ***vindex***
Opcode:	00011110
Parameter:	*Vindex* is an unsigned 8-bit integer in the range of 0..255.
Operation:	Pops a value (subtractor) from the operand stack and subtracts the value from the variable *vindex*.
Operands:	Integers or floating-points
Conversion:	6 – Integers or floating-points
Result:	Variable containing the result of the subtraction
Operand stack:	…, value => …
Errors:	1 (Verification Failed), 7 (Stack Underflow)

10.5.5.5	**Instruction:** UMINUS
Opcode:	00011111
Parameters:	-
Operation:	Pops a value from the operand stack and performs a unary minus operation on it and pushes the result back on the operand stack.
Operand:	Integer or floating-point
Conversion:	5 – Integer or floating-point (unary)
Result:	Integer or floating-point (negated)
Operand stack:	…, value => …, -value
Errors:	7 (Stack Underflow)

10.5.5.6	**Instruction:** ADD
Opcode:	00100000
Parameters:	-
Operation:	Pops two values from the operand stack and performs an add operation on them and pushes the result back on the operand stack.
Operands:	Integers, floating-points or strings
Conversion:	7 – Integers, floating-points or strings
Result:	*For integers or floating-points*: the result of the addition *For strings*: the result of the concatenation
Operand stack:	…, value1, value2 => …, value1 + value2
Errors:	7 (Stack Underflow)

10.5.5.7	**Instruction:** SUB
Opcode:	00100001
Parameters:	-
Operation:	Pops two values from the operand stack and performs a subtract operation on them and pushes the result back on the operand stack.
Operands:	Integers or floating-points
Conversion:	6 – Integers or floating-points
Result:	Integer or floating-point
Operand stack:	…, value1, value2 => …, value1 - value2
Errors:	7 (Stack Underflow)

10.5.5.8	**Instruction:** MUL
Opcode:	00100010
Parameters:	-
Operation:	Pops two values from the operand stack, performs a multiplication operation on them and pushes the result back on the operand stack.
Operands:	Integers or floating-points
Conversion:	6 – Integers or floating-points
Result:	Integer or floating-point
Operand stack:	…, value1, value2 => …, value1 * value2
Errors:	7 (Stack Underflow)

10.5.5.9	**Instruction:** DIV
Opcode:	00100011
Parameters:	-
Operation:	Pops two values from the operand stack, performs a division operation on them and pushes the result back on the operand stack.
Operands:	Integers or floating-points
Conversion:	6 – Integers or floating-points
Result:	Floating-point
Operand stack:	…, value1, value2 => …, value1 / value2
Errors:	7 (Stack Underflow)

10.5.5.10	**Instruction:** IDIV
Opcode:	00100100
Parameters:	-
Operation:	Pops two values from the operand stack, performs an integer division operation on them and pushes the result back on the operand stack.
Operands:	Integers
Conversion:	2 – Integer(s)
Result:	Integer
Operand stack:	…, value1, value2 => …, value1 IDIV value2
Errors:	7 (Stack Underflow)

10.5.5.11	**Instruction:** REM
Opcode:	00100101
Parameters:	-
Operation:	Pops two values from the operand stack, performs a reminder operation on them (the sign of the result equals the sign of the dividend) and pushes the result back on the operand stack.
Operands:	Integers
Conversion:	2 – Integer(s)
Result:	Integer
Operand stack:	…, value1, value2 => …, value1 % value2
Errors:	7 (Stack Underflow)

10.5.6 Bitwise Instructions

10.5.6.1	**Instruction:** B_AND
Opcode:	00100110
Parameters:	-
Operation:	Pops two values from the operand stack and performs a bitwise and operation on them and pushes the result back on the operand stack.
Operands:	Integers
Conversion:	2 – Integer(s)
Result:	Integer
Operand stack:	…, value1, value2 => …, value1 & value2
Errors:	7 (Stack Underflow)

10.5.6.2	**Instruction:** B_OR
Opcode:	00100111
Parameters:	-
Operation:	Pops two values from the operand stack and performs a bitwise or operation on them and pushes the result back on the operand stack.
Operands:	Integers
Conversion:	2 – Integer(s)
Result:	Integer
Operand stack:	…, value1, value2 => …, value1 | value2
Errors:	7 (Stack Underflow)

10.5.6.3	**Instruction:** B_XOR
Opcode:	00101000
Parameters:	-
Operation:	Pops two values from the operand stack, performs a bitwise xor operation on them and pushes the result back on the operand stack.
Operands:	Integers
Conversion:	2 – Integer(s)
Result:	Integer
Operand stack:	…, value1, value2 => …, value1 ^ value2
Errors:	7 (Stack Underflow)

10.5.6.4	**Instruction:** B_NOT
Opcode:	00101001
Parameters:	-
Operation:	Pops a value from the operand stack and performs a bitwise complement operation on it and pushes the result back on the operand stack.
Operands:	Integer
Conversion:	2 – Integer(s)
Result:	Integer
Operand stack:	…, value => …, ~value
Errors:	7 (Stack Underflow)

10.5.6.5	**Instruction:** B_LSHIFT
Opcode:	00101010
Parameters:	-
Operation:	Pops two values from the operand stack, performs a bitwise left-shift operation on them and pushes the result back on the operand stack.
Operands:	Integers
Conversion:	2 – Integer(s)
Result:	Integer
Operand stack:	…, value, amount => …, value << amount
Errors:	7 (Stack Underflow)

10.5.6.6	**Instruction:** B_RSSHIFT
Opcode:	00101011
Parameters:	-
Operation:	Pops two values from the operand stack, performs a bitwise signed right-shift operation on them and pushes the result back on the operand stack.
Operands:	Integers
Conversion:	2 – Integer(s)
Result:	Integer
Operand stack:	…, value, amount => …, value >> amount
Errors:	7 (Stack Underflow)

10.5.6.7	**Instruction:** B_RSZSHIFT
Opcode:	00101100
Parameters:	-
Operation:	Pops two values from the operand stack and performs a bitwise right-shift with zero operation on them and pushes the result back on the operand stack.
Operands:	Integers
Conversion:	2 – Integer(s)
Result:	Integer
Operand stack:	…, value, amount => …, value >>> amount
Errors:	7 (Stack Underflow)

10.5.7 Comparison Instructions

10.5.7.1	**Instruction:** EQ
Opcode:	00101101
Parameters:	-
Operation:	Pops two values from the operand stack, performs a logical equality operation on them and pushes the result back on the operand stack.
Operands:	Integers, floating-points or strings
Conversion:	7 – Integers, floating-points or strings
Result:	Boolean
Operand stack:	…, value1, value2 => …, value1 EQ value2
Errors:	7 (Stack Underflow)

10.5.7.2	**Instruction:** LE
Opcode:	00101110
Parameters:	-
Operation:	Pops two values from the operand stack, performs a logical less-or-equal operation on them and pushes the result back on the operand stack.
Operands:	Integers, floating-points or strings
Conversion:	7 – Integers, floating-points or strings
Result:	Boolean
Operand stack:	…, value1, value2 => …, value1 LE value2
Errors:	7 (Stack Underflow)

10.5.7.3	**Instruction:** LT
Opcode:	00101111
Parameters:	-
Operation:	Pops two values from the operand stack, performs a logical less-than operation on them and pushes the result back on the operand stack.
Operands:	Integers, floating-points or strings
Conversion:	7 – Integers, floating-points or strings
Result:	Boolean
Operand stack:	…, value1, value2 => …, value1 LT value2
Errors:	7 (Stack Underflow)

10.5.7.4	**Instruction:** GE
Opcode:	00110000
Parameters:	-
Operation:	Pops two values from the operand stack, performs a logical greater-or-equal operation on them and pushes the result back on the operand stack.
Operands:	Integers, floating-points or strings
Conversion:	7 – Integers, floating-points or strings
Result:	Boolean
Operand stack:	…, value1, value2 => …, value1 GE value2
Errors:	7 (Stack Underflow)

10.5.7.5	**Instruction:** GT
Opcode:	00110001
Parameters:	-
Operation:	Pops two values from the operand stack, performs a greater-than operation on them and pushes the result back on the operand stack.
Operands:	Integers, floating-points or strings
Conversion:	7 – Integers, floating-points or strings
Result:	Boolean
Operand stack:	…, value1, value2 => …, value1 GT value2
Errors:	7 (Stack Underflow)

10.5.7.6	**Instruction:** NE
Opcode:	00110010
Parameters:	-
Operation:	Pops two values from the operand stack, performs a logical not-equal operation on them and pushes the result back on the operand stack.
Operands:	Integers, floating-points or strings
Conversion:	7 – Integers, floating-points or strings
Result:	Boolean
Operand stack:	…, value1, value2 => …, value1 NE value2
Errors:	7 (Stack Underflow)

10.5.8 Logical Instructions

10.5.8.1	**Instruction:** NOT
Opcode:	00110011
Parameters:	-
Operation:	Pops a value from the operand stack and performs a logical complement operation on it and pushes the result back on the operand stack.
Operands:	Boolean
Conversion:	1 – Boolean(s)
Result:	Boolean
Operand stack:	…, value => …, !value
Errors:	7 (Stack Underflow)

10.5.8.2	**Instruction:** SCAND
Opcode:	00110100
Parameters:	-
Operation:	Pops a value from the operand stack and converts it to a boolean value. If the converted value is `false` or `invalid` then the converted value itself is pushed on the operand stack and the boolean value `false` is pushed on the operand stack. If the converted value is `true` then the converted value itself is pushed on the operand stack.
Operands:	Any
Conversion:	1 – Boolean(s)
Result:	Boolean
Operand stack:	…, value => …, false, false (in case the value is `false`) …, value => …, true (in case the value is `true`) …, value => …, invalid, false (in case the value is `invalid`)
Errors:	7 (Stack Underflow)

10.5.8.3	**Instruction:** SCOR
Opcode:	00110101
Parameters:	-
Operation:	Pops a value from the operand stack and converts it to a boolean value. If the converted value is `false` then the boolean value `true` is pushed on the operand stack. If the converted value is `true` or `invalid` then the converted value itself is pushed on the operand stack and the boolean value `false` is pushed on the operand stack.
Operands:	Any
Conversion:	1 – Boolean(s)
Result:	Boolean
Operand stack:	…, value => …, true (in case the value is `false`) …, value => …, true, false (in case the value is `true`) …, value => …, invalid, false (in case the value is `invalid`)
Errors:	7 (Stack Underflow)

10.5.8.4	**Instruction:** TOBOOL
Opcode:	00110110
Parameters:	-
Operation:	Pops a value from the operand stack and converts the value to a boolean value and pushes the converted value on the operand stack. If the popped value is `invalid` then an `invalid` value is pushed back on the operand stack.
Operands:	Any
Conversion:	1 – Boolean(s)
Result:	Boolean
Operand stack:	…, value => …, tobool
Errors:	7 (Stack Underflow)

10.5.9 Stack Instructions

10.5.9.1	**Instruction:** POP
Opcode:	00110111
Parameters:	-
Operation:	Pops a value from the operand stack.
Operands:	Any
Conversion:	-
Result:	-
Operand stack:	…, value => …
Errors:	7 (Stack Underflow)

10.5.10 Access to Operand Type

10.5.10.1	**Instruction:** TYPEOF
Opcode:	00111000
Parameters:	-
Operation:	Pops a value from the operand stack and checks its type. Pushes the result as an integer on the operand stack. The possible results are: 0 = Integer, 1 = Floating-point, 2 = String, 3 = Boolean, 4 = Invalid
Operands:	Any
Conversion:	-
Result:	Integer
Operand stack:	…, value => …, typeof?
Errors:	7 (Stack Underflow)

10.5.10.2	**Instruction:** ISVALID
Opcode:	00111001
Parameters:	-
Operation:	Pops a value from the operand stack and checks its type. If the type is *invalid* a boolean value `false` is pushed on the operand stack, otherwise a boolean value `true` is pushed on the operand stack.
Operands:	Any
Conversion:	-
Result:	Boolean
Operand stack:	…, value => …, valid?
Errors:	7 (Stack Underflow)

10.5.11 Function Return Instructions

10.5.11.1	**Instruction:** RETURN
Opcode:	00111010
Parameters:	-
Operation:	Returns the control back to the caller. The return value is on the top of the operand stack. The execution continues at the next instruction following the function call of the calling function.
Operands:	Any
Conversion:	-
Result:	-
Operand stack:	…, ret-value => …, ret-value
Errors:	7 (Stack Underflow)

10.5.11.2	**Instruction:** RETURN_ES
Opcode:	00111011
Parameters:	-
Operation:	Pushes an empty string on the operand stack and returns the control back to the caller. The execution continues at the next instruction following the function call of the calling function.
Operands:	-
Conversion:	-
Result:	-
Operand stack:	… => …, ""
Errors:	-

10.5.12 Miscellaneous Instructions

10.5.12.1	**Instruction:** DEBUG
Opcode:	00111100
Parameters:	-
Operation:	No operation. Reserved for debugging and profiling purposes.
Operands:	-
Conversion:	-
Result:	-
Operand stack:	No change
Errors:	-

11. BYTECODE VERIFICATION

Bytecode verification takes place before or while the bytecode is used for execution. The purpose of the verification is to make sure that the content follows the WMLScript bytecode specification. In case of verification failure, the failed bytecode should not be used for execution or the execution must be aborted and failure signalled to the caller of the WMLScript interpreter.

The following checks are to be executed in the WMLScript Interpreter either before the execution is started or during the execution of WMLScript bytecode.

11.1 Integrity Check

The following list contains checks that must be used to verify the integrity of the WMLScript bytecode *before* it is executed:

- Check that the *version number* is correct: The bytecode version number must be compared with the bytecode version number supported by the WMLScript interpreter. The *major* version numbers must match. The *minor* version number of the bytecode must be less than or equal to the *minor* version number supported by the WMLScript interpreter.

- Check that the *size of the bytecode* is correct: The size specified in the bytecode must match exactly the byte size of the content.

- Check the *constant pool*:
  - The *number of constants* is correct: The number of constants specified in the constant pool must match the number of constants stored into the constant pool.
  - The *types of constants* are valid: The numbers used to specify the constant types in the constant pool must match the supported constant types. Reserved constant types (7-255) result in a verification failure.
  - The *sizes of constants* are valid: Each constant must allocate only the correct number of bytes specified by the WMLScript bytecode specification (fixed size constants such as integers) or the size parameter provided as part of the constant entity (constants of varying size such as strings).

- Check the *pragma pool*:
  - The *number of pragmas* is correct: The number of pragmas specified in the pragma pool must match the number of pragmas stored into the pragma pool.
  - The *types of pragmas* are valid: The numbers used to specify the pragma types in the pragma pool must match the supported pragma types. Reserved pragma types (4-255) result in a verification failure.
  - The *constant pool indexes* are valid:
    - The access control domain and path must point to string constants.
    - The constant pool indexes used in meta-information pragmas must point to string constants.

- Check the *function pool*:
  - The *number of functions* is correct: The number of functions specified in the function pool must match the number of functions stored into the function pool.
  - The *function name table* is correct:

    - The *number of function names* is correct: The number of function names specified in the function name table must match the number of function names stored into the function name table.
    - The *function name indexes* are correct: The indexes must point to existing functions in the function pool.
    - The *function names* contain only valid function name characters: Function names must follow the WMLScript function name syntax.
    - There is at least one name in the table.
  - The *function prologue* is correct:
    - The *number of arguments* and *local variables* is correct: The sum of the number of arguments and local variables must be less or equal to 256.
    - The *size of the function* is correct: The size specified in the function prologue must match exactly the byte size of the function.

11.2 Runtime Validity Checks

The following list contains the checks that must be done *during* the execution to verify that the used instructions are valid and they use valid parameter values:

- Check that the bytecode contains only valid *instructions*: Only instructions that are defined in chapter 10 are valid.
- Check that *local variable references* are valid: The references must be within the boundaries specified by the number of function local variables in the function prologue.
- Check that *constant references* are valid:
  - The references must be within the boundaries specified by the number of constants in the constant pool.
  - The references must point to the valid constant types specified by each instruction:
    - In case of URL references, the referred constant strings must contain a valid URL (see [RFC2396]).
    - In case of Function Name references, the referred constant strings must contain a valid WMLScript function name.
- Check that the standard *library indexes* and *library function indexes* are valid: The indexes must be within the boundaries specified by the WMLScript Standard Libraries specification [WMLSLibs].
- Check that *local function call indexes* are valid: The function indexes must match with the number of functions specified in the function pool.
- Check that the *jumps* are within function boundaries: All jumps must have a target inside the function in which they are specified.
- Check that the targets of *jumps* are valid: The target of all jumps must be the beginning of an instruction.
- Check that the *ends* of the functions are valid: Functions must not *end* in the middle of an instruction.

12. RUN-TIME ERROR DETECTION AND HANDLING

Since WMLScript functions are used to implement services for users that expect the terminals (in particular mobile phones) to work properly in all situations, error handling is of utmost importance. This means that while the language does not provide, for example, an exception mechanism, it should provide tools to either prevent errors from happening or tools to notice them and take appropriate actions. Aborting a program execution should be the last resort used only in cases where nothing else is possible.

The following section lists errors that can happen when downloading bytecode and executing it. It does not contain programming errors (such as infinite loop etc.). For these cases a user controlled abortion mechanism is needed.

12.1 Error Detection

The goal of error detection is to give tools for the programmer to detect errors (if possible) that would lead to erroneous behaviour. Since WMLScript is a weakly typed language, special functionality has been provided to detect errors that are caused by invalid data types :

- Check that the given variable contains the *right value*: WMLScript supports type validation library [WMLSLibs] functions such as *Lang.isInt()*, *Lang.isFloat()*, *Lang.parseInt()* and *Lang.parseFloat()*.
- Check that the given variable contains a value that is of *right type*: WMLScript supports the operators *typeof* and *isvalid* that can be used for this purpose.

12.2 Error Handling

Error handling takes place after an error has already happened. This is the case when the error could not be prevented by error detection (memory limits, external signals etc.) or it would have been too difficult to do so (overflow, underflow etc.). These 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 agents, program abortion should always be signalled to the calling user agent. It is then its responsibility to take the appropriate actions to signal the user of errors.
- *Non-fatal errors*: These are errors that can be signalled back to the program as special return values and the program can decide on the appropriate action.

The following error descriptions are divided into sections based on their fatality.

12.3 Fatal Errors

12.3.1 Bytecode Errors

These errors are related to the bytecode and the instructions being 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.

12.3.1.1 Verification Failed

Description:	Reports that the specified bytecode for the called compilation unit did not pass the verification (see section 11 for more information about bytecode verification).
Generated:	At any time when a program attempts to call an external function.
Example:	`var a = 3*OtherScript#doThis(param);`
Severity:	Fatal.
Predictable:	Is detected during the bytecode verification.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.3.1.2 Fatal Library Function Error

Description:	Reports that a call to a library function resulted in a fatal error.
Generated:	At any time when a call to a library function is used (CALL_LIB). Typically, this is an unexpected error in the library function implementation.
Example:	`var a = String.format(param);`
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.3.1.3 Invalid Function Arguments

Description:	Reports that the number of arguments specified for a function call do not match with the number of arguments specified in the called function.
Generated:	At any time a call to an external function is used (CALL_URL).
Example:	Compiler generates an invalid parameter to an instruction or the number of parameters in the called function has changed.
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.3.1.4 External Function Not Found

Description:	Reports that a call to an external function could not be found from the specified compilation unit.
Generated:	At any time, when a program attempts to call an external function (CALL_URL).
Example:	`var a = 3*OtherScript#doThis(param);`
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.3.1.5 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 the specified compilation unit does not exist in the network server.
Generated:	At any time, when a program attempts to call an external function (CALL_URL).
Example:	`var a = 3*OtherScript#doThis(param);`
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.3.1.6 Access Violation

Description:	Reports an access violation. The called external function resides in a protected compilation unit.
Generated:	At any time when a program attempts to call an external function (CALL_URL).
Example:	`var a = 3*OtherScript#doThis(param);`
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.3.1.7 Stack Underflow

Description:	Indicates a stack underflow because of a program error (compiler generated bad code).
Generated:	At any time when a program attempts to pop an empty stack.
Example:	Only generated if compiler generates bad code.
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.3.2 Program Specified Abortion

This error is generated when a WMLScript function calls the library function *Lang.abort()* (see [WMLSLibs]) to abort the execution.

12.3.2.1 Programmed Abort

Description:	Reports that the execution of the bytecode was aborted by a call to *Lang.abort()* function.
Generated:	At any time when a program makes a cal to *Lang.abort()* function..
Example:	`Lang.abort("Unrecoverable error");`
Severity:	Fatal.
Predictable:	No.

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

12.3.3 Memory Exhaustion Errors

These errors are related to the dynamic behaviour of the WMLScript interpreter (see section 8.1 for more information) and its memory usage.

12.3.3.1 Stack Overflow

Description:	Indicates a stack overflow.
Generated:	At any time when a program recourses too deep or attempts to push too many variables onto the operand stack.
Example:	`function f(x) { f(x+1); };`
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.3.3.2 Out of Memory

Description:	Indicates that no more memory resources are available to the interpreter.
Generated:	At any time when the operating system fails to allocate more space for the interpreter.

Example:	`function f(x) {`
	`  x=x+”abcdefghijklmnopqrstuvzyxy”;`
	`  f(x);`
	`};`

Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.3.4 External Exceptions

The following exceptions are initiated outside of the WMLScript Bytecode Interpreter.

12.3.4.1 User Initiated

Description:	Indicates that the user wants to abort the execution of the program (reset button etc.)
Generated:	At any time.
Example:	User presses reset button while an application is running.
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.3.4.2 System Initiated

Description:	Indicates that an external fatal exception occurred while a program is running and it must be aborted. Exceptions can be originated from a low battery, power off, etc.
Generated:	At any time.
Example:	The system is automatically switching off due to a low battery.
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the WMLScript interpreter.

12.4 Non-Fatal Errors

12.4.1 Computational Errors

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

12.4.1.1 Divide by Zero

Description:	Indicates a division by zero.
Generated:	At any time when 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.
Predictable:	Yes.
Solution:	The result is an `invalid` value.

12.4.1.2 Integer Overflow

Description:	Reports an arithmetic integer overflow.
Generated:	At any time when a program attempts to execute an integer operation.

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

Severity:	Non-fatal.
Predictable:	Yes (but difficult in certain cases).
Solution:	The result is an `invalid` value.

12.4.1.3 Floating-Point Overflow

Description:	Reports an arithmetic floating-point overflow.
Generated:	At any time when 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.

Predictable: Yes (but difficult in certain cases).
Solution: The result is an `invalid` value.

12.4.1.4 Floating-Point Underflow

Description: Reports an arithmetic underflow.
Generated: At any time when 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.
Predictable: Yes (but difficult in certain cases).
Solution: The result is a floating-point value `0.0`.

12.4.2 Constant Reference Errors

These errors are related to run-time references to constants in the constant pool.

12.4.2.1 Not a Number Floating-Point Constant

Description: Reports a reference to a floating-point literal in the constant pool that is *Not a Number* [IEEE754].
Generated: At any time when 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.
Predictable: Yes.
Solution: The result is an `invalid` value.

12.4.2.2 Infinite Floating-Point Constant

Description: Reports a reference to a floating-point literal in the constant pool that is either positive or negative infinity [IEEE754].
Generated: At any time when 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.
Predictable: Yes.
Solution: The result is an `invalid` value.

12.4.2.3 Illegal Floating-Point Reference

Description: Reports an erroneous reference to a floating-point value in the constant pool.
Generated: At any time when a program attempts to use floating-point values and the environments supports only integer values.
Example:
```
var a = 3.14;
```
Severity: Non-fatal.

Predictable:	Can be detected during the run-time.
Solution:	The result is an `invalid` value.

12.4.3 Conversion Errors

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

12.4.3.1 Integer Too Large

Description:	Indicates a conversion to an integer value where the integer value is too large (positive/negative).
Generated:	At any time when an application attempts to make an automatic conversion to an integer value.
Example:	`var a = -"99999999999999999999999999999999999999999999";`
Severity:	Non-fatal.
Predictable:	No.
Solution:	The result is an `invalid` value.

12.4.3.2 Floating-Point Too Large

Description:	Indicates a conversion to a floating-point value where the floating-point value is too large (positive/negative).
Generated:	At any time when an application attempts to make an automatic conversion to a floating-point value.
Example:	`var a = -"9999999.9999999999e99999";`
Severity:	Non-fatal.
Predictable:	No.
Solution:	The result is an `invalid` value.

12.4.3.3 Floating-Point Too Small

Description:	Indicates a conversion to a floating-point value where the floating-point value is too small (positive/negative).
Generated:	At any time when an application attempts to make an automatic conversion to a floating-point value.
Example:	`var a = -"0.01e-99";`
Severity:	Non-fatal.
Predictable:	No.
Solution:	The result is a floating-point value `0.0`.

12.5 Library Calls and Errors

Since WMLScript supports the usage of libraries, there is a possibility that errors take place inside the library functions. Design and the behaviour of the library functions are not part of the WMLScript language specification. However, following guidelines should be followed when designing libraries:

- Provide the library users mechanisms by which errors can be detected before they happen.
- Use the same error handling mechanisms as WMLScript operators in cases where error should be reported back to the caller.
- Minimise the possibility of fatal errors in all library functions.

13. SUPPORT FOR INTEGER ONLY DEVICES

The WMLScript language has been designed to run also on devices that do not support floating-point operations. The following rules apply when WMLScript is used with such devices:

- Variables can only contain the following internal data types:
  - Boolean
  - Integer
  - String
  - Invalid
- Any LOAD_CONST bytecode that refers to a floating point constant in the constant pool will push an `invalid` value on the operand stack instead of the constant value.
- Division (/) operation returns always an `invalid` value.
- Assignment with division (/=) operation always results in an `invalid` value.
- All conversion rules related to floating-points are ignored.
- URL call with a floating-point value as an argument results in a failure to execute the call due to an invalid URL syntax.

The programmer can use *Lang.float()* [WMLSLibs] to test (during the run-time) if floating-point operations are supported.

14. CONTENT TYPES

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

- Textual form: `text/vnd.wap.wmlscript`
- Binary form: `application/vnd.wap.wmlscriptc`

15. STATIC CONFORMANCE REQUIREMENTS

This static conformance clause defines a minimum set of features that can be implemented to ensure that WMLScript encoders and interpreters will be able to inter-operate. While both interpreter behavior and encoder behavior is described in the WMLScript specification, not all items apply to both entities, so there are separate tables for each. A feature can be optional or mandatory.

15.1 Encoder

15.1.1 Core Capabilities

Item	Function	Reference	Mandatory/Optional
WMLS-001	Floating Point Support	Support for Integer Only Devices	M
WMLS-002	WMLScript Standard Libraries	[WMLSLibs]	M

15.1.2 WMLScript Language Core

Item	Function	Reference	Mandatory/Optional
WMLS-003	Language is case-sensitive	Case Sensitivity	M
WMLS-004	Ignore extra white space and line break between program tokens	Whitespace and Line Breaks	M
WMLS-005	Semicolon is used to terminate certain statements	Usage of Semicolons	M
WMLS-006	Multi-line and single-line comments	Comments	M
WMLS-007	Disallows nested comments	Comments	M
WMLS-008	Integer literals	Integer Literals	M
WMLS-009	Floating point literals	Floating-Point Literals	M

Item	Function	Reference	Mandatory/Optional
WMLS-010	String literals, single and double quoted	String Literals	M
WMLS-011	Special escape sequences	String Literals	M
WMLS-012	Boolean literals	Boolean Literals	M
WMLS-013	Invalid literal	Invalid Literal	M
WMLS-014	Identifier syntax	Identifiers	M
WMLS-015	Variable scope and lifetime	Variable Scope and Lifetime	M
WMLS-016	Integer size	Integer Size	M
WMLS-017	Floating point size	Floating-point Size	M
WMLS-018	Assignment operators	Assignment Operators	M
WMLS-019	Arithmetic operators	Arithmetic Operators	M
WMLS-020	Logical operators	Logical Operators	M
WMLS-021	String operators	String Operators	M
WMLS-022	Comparison operators	Comparison Operators	M
WMLS-023	Array operators	Array Operators	M
WMLS-024	Comma operator	Comma Operator	M
WMLS-025	Conditional operator	Conditional Operator	M
WMLS-026	typeof operator	typeof Operator	M
WMLS-027	isvalid operator	isvalid Operator	M
WMLS-028	Expression bindings	Expression Bindings	M
WMLS-029	Function declaration	1Declaration	M
WMLS-030	Local script functions calls	Local Script Functions	M
WMLS-031	External function calls	External Functions	M
WMLS-032	Library function calls	Library Functions	M
WMLS-033	Default function return value	Default Return Value	M
WMLS-034	Empty statement	Empty Statement	M
WMLS-035	Block statement	Block Statement	M
WMLS-036	Variable statement	Variable Statement	M
WMLS-037	if statement	If Statement	M
WMLS-038	while statement	While Statement	M
WMLS-039	for statement	For Statement	M

Item	Function	Reference	Mandatory/Optional
WMLS-040	break statement	Break Statement	M
WMLS-041	continue statement	Continue Statement	M
WMLS-042	return statement	Return Statement	M
WMLS-043	External compilation unit pragma	External Compilation Units	M
WMLS-044	Access control pragma	Access Control	M
WMLS-045	Meta information pragma	Meta-Information	M

15.1.3 Function Calls

Item	Function	Reference	Mandatory/Optional
WMLS-046	Function argument passing	Passing of Function Arguments	M
WMLS-047	Allocation of variable indexes	Allocation of Variable Indexes	M
WMLS-048	Automatic function return value	Automatic Function Return Value	M
WMLS-049	Variable initialization	Initialisation of Variables	M

15.1.4 Binary Format

Item	Function	Reference	Mandatory/Optional
WMLS-050	Binary format data types	Used Data Types	M
WMLS-051	Multi-byte integer format	Multi-byte Integer Format	M
WMLS-052	Character encoding	Character Encoding	M
WMLS-053	Header Info	Bytecode Header	M
WMLS-054	Constant Pool	Constant Pool	M
WMLS-055	Pragma Pool	Pragma Pool	M
WMLS-056	Function Pool	Function Pool	M

15.1.5 Instruction Set

Item	Function	Reference	Encoder Status
WMLS-057	Control flow instructions	Control Flow Instructions	M
WMLS-058	Function call instructions	Function Call Instructions	M
WMLS-059	Variable access and manipulation	Variable Access and Manipulation	M
WMLS-060	Access to constants	Access To Constants	M
WMLS-061	Arithmetic instructions	Arithmetic Instructions	M
WMLS-062	Bitwise instructions	Bitwise Instructions	M
WMLS-063	Comparison instructions	Comparison Instructions	M
WMLS-064	Logical instructions	Logical Instructions	M
WMLS-065	Stack instructions	Stack Instructions	M
WMLS-066	Access to operand type	Access to Operand Type	M
WMLS-067	Return instructions	Function Return Instructions	M
WMLS-068	Debug instruction	Miscellaneous Instructions	O

15.2 Interpreter

15.2.1 Core Capabilities

Item	Function	Reference	Mandatory/Optional
WMLS-069	Support for interpreting WMLScript bytecode	WMLScript Bytecode Interpreter	M
WMLS-070	WMLScript Standard Libraries	[WMLSLibs]	M

15.2.2 Automatic Data Conversion

Item	Function	Reference	Mandatory/Optional
WMLS-071	Conversions to String	Conversions to String	M
WMLS-072	Conversions to Integer	Conversions to Integer	M
WMLS-073	Conversions to Floating Point	Conversions to Floating-Point	O
WMLS-074	Conversions to Boolean	Conversions to Boolean	M
WMLS-075	Conversions to Invalid	Conversions to Invalid	M
WMLS-076	Operator data type conversion rules	Operator Data Type Conversion Rules	M

15.2.3 Function Calls

Item	Function	Reference	Mandatory/Optional
WMLS-077	URL schemes	URL Schemes	M
WMLS-078	Fragment anchor	Fragment Anchors	M
WMLS-079	URL call syntax	URL Call Syntax	M

Item	Function	Reference	Mandatory/Optional
WMLS-080	URL call parameter passing	URL Calls and Parameter Passing	M
WMLS-081	Support for relative URLs	Relative URLs	M
WMLS-082	Function argument passing	Passing of Function Arguments	M
WMLS-083	Allocation of variable indexes	Allocation of Variable Indexes	M
WMLS-084	Automatic function return value	Automatic Function Return Value	M
WMLS-085	Variable initialization	Initialisation of Variables	M
WMLS-086	Access control	Access Control	M

15.2.4 Binary Format

Item	Function	Reference	Mandatory/Optional
WMLS-087	Binary format data types	Used Data Types	M
WMLS-088	Multi-byte integer format	Multi-byte Integer Format	M
WMLS-089	Character encoding	Character Encoding	M
WMLS-090	Header Info	Bytecode Header	M
WMLS-091	Constant Pool	Constant Pool	M
WMLS-092	Pragma Pool	Pragma Pool	M
WMLS-093	Function Pool	Function Pool	M

15.2.5 Instruction Set

Item	Function	Reference	Mandatory/Optional
WMLS-094	Control flow instructions	Control Flow Instructions	M
WMLS-095	Function call instructions	Function Call Instructions	M

Item	Function	Reference	Mandatory/Optional
WMLS-096	Variable access and manipulation	Variable Access and Manipulation	M
WMLS-097	Access to constants	Access To Constants	M
WMLS-098	Arithmetic instructions	Arithmetic Instructions	M
WMLS-099	Bitwise instructions	Bitwise Instructions	M
WMLS-100	Comparison instructions	Comparison Instructions	M
WMLS-101	Logical instructions	Logical Instructions	M
WMLS-102	Stack instructions	Stack Instructions	M
WMLS-103	Access to operand type	Access to Operand Type	M
WMLS-104	Return instructions	Function Return Instructions	M
WMLS-105	Debug instruction	Miscellaneous Instructions	M

15.2.6 Error Handling

Item	Function	Reference	Mandatory/Optional
WMLS-106	Bytecode verification	Integrity Check	M
WMLS-107	Runtime validity checking	Runtime Validity Checks	M
WMLS-108	Support for general error handling	Error Handling	M
WMLS-109	Handling of fatal errors	Fatal Errors	M
WMLS-110	Handling of non-fatal errors	Non-Fatal Errors	M

15.2.7 Support for Integer Only Devices

Item	Function	Reference	Mandatory/Optional
WMLS-111	Support for floating-point operations	Support for Integer Only Devices	O