WMLScript Specification

Draft Version 23-Jan-1998

Wireless Application Protocol
WMLScript Language Specification

Disclaimer:

This document is a draft document of the WMLScript Language Specification, and is subject to change without notice.

Contents

1. Scope

Wireless Application Protocol (WAP) is a result of continuous work to define an industry wide standard for developing applications over wireless communication networks. The scope for the WAP working group 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 script support to the client. The language is based on JavaScript™ [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 JavaScript and WMLScript is the fact that WMLScript is compiled into bytecode *before* it is sent to the client. This way the narrowband communication channels available today can be optimally utilised and the memory requirements for the client kept to the minimum. For the same reasons, many of the advanced features of the JavaScript language have been dropped to make the language small, easier to compile into bytecode, and easier to learn.

2. Document Status

This document is a preliminary draft. It is published to solicit comments from WAP members and other interested parties. The document is subject to change without any notice. It may be updated, replaced, or dropped at any time. Publishing the document does not imply endorsement nor does it imply that it will be part of a published WAP standard or recommendation. Contents of this draft reflect "work in progress." Any references to the document's content should only cite them as "work in progress."

This document is available online in the following formats:

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

2.1. Copyright Notice

2.2. Errata

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

2.3. Comments

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

3. References

3.1. Normative references

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

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

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

[RFC1738] "Uniform Resource Locators (URL)", T. Berners-Lee, et al., December 1994. URL: ftp://ds.internic.net/rfc/rfc1738.txt

[RFC1808] "Relative Uniform Resource Locators", R. Fielding, June 1995. URL: ftp://ds.internic.net/rfc/rfc1808.txt

[RFC2068] "Hypertext Transfer Protocol - HTTP/1.1", R. Fielding, et al., January 1997. URL: ftp://ds.internic.net/rfc/rfc2068.txt

[RFC2119] "Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997. URL: ftp://ds.internic.net/rfc/rfc2119.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, version 0.9", Wireless Application Protocol Architecture Working Group, 1997. URL: http://www.wapforum.org/

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

[WMLStdLib] "WMLScript Standard Libraries Specification", WAP Forum, January 30, 1998. URL: http://www.wapforum.org/

[WSP] "Wireless Session Protocol", WAP Forum, January 30, 1998. URL: http://www.wapforum.org/

[XML] "Extensible Markup Language (XML), W3C Proposed Recommendation 8-December-1997, PR-xml-971208", T. Bray, et al, December 8, 1997. URL: http://www.w3.org/TR/PR-xml

3.2. Informative References

[CLang] ANSI X3.159-1989: American National Standard for Information Systems - Programming Language - C, American National Standards Institute (1989).

[HDML2] "Handheld Device Markup Language Specification", P. King, et al., April 11, 1997. URL: http://www.uplanet.com/pub/hdml_w3c/hdml20-1.html

[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

[ISO8879] "Information Processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML)", ISO 8879:1986.

[Java] "The Java Language Specification", Gosling, James, Bill Joy and Guy Steele. Addison-Wesley Publishing Company 1996.

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

[RFC2044] "UTF-8, a transformation format of Unicode and ISO 10646", F. Yergeau, October 1996. URL: ftp://ds.internic.net/rfc/rfc2044.txt

[RFC2045] "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", N. Freed, et al., November 1996. URL: ftp://ds.internic.net/rfc/rfc2045.txt

[RFC2048] "Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures", N. Freed, et al., November 1996. URL: ftp://ds.internic.net/rfc/rfc2048.txt

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

[WTA] "Wireless Telephony Application Specification", Wireless Application Working Group, 1998. URL: http://www.wapforum.org/

[WTAI] "Wireless Telephony Application Interface Specification", Wireless Application Working Group, 1998. 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].

Author - an author is a person or program that writes or generates WML, WMLScript or other content.

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

Card - a navigable part of a WML document (deck). May contain information to present on the screen, instructions for gathering user input, etc.

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

Client Server Communication – communication between a client and a server. Typically the server performs a task (such as generating content) on behalf of the server. Results of the task are usually sent back to the client (e.g., generated content.)

Content - synonym for resources.

Content Encoding – when used as a verb, content encoding indicates the act of converting content 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.

Content Generator – devices (or applications) that generate or format content. Typically content generators are on origin servers.

Deck - a WML document. May contain WMLScript.

Deprecated - an element, attribute or other construct that is outdated by other constructs and should not be used by applications. Deprecated constructs remain in the specification for a variety of purposes, including ease of application migration, backward compatibility, etc. Deprecated elements may become obsolete in a future specification.

Device - a is 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. Also known as ECMAScript.

Obsolete - this term indicates a construct or element which is no longer supported, and for which there is no guarantee of support by a given user agent.

Origin Server - the server on which a given resource resides or is to be created.

Peer-to-peer – direct communication between two terminals typically thought of as clients without involving an intermediate server. Also known as client-to-client communication.

Resource - A network data object or service that can be identified by a URI, as defined in section 3.2. Resources may be available in multiple representations (e.g. multiple languages, data formats, size, 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.

SGML - the Standardized Generalized Markup Language (defined in [ISO8879]) is a general-purpose language for domain-specific markup languages.

Terminal - a device. Also called a mobile terminal or mobile station.

Transcode - the act of converting from one character set to another, e.g., conversion from UCS-2 to UTF-8.

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.

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

Terminal - A terminal (or device) is a network entity that is capable of sending and receiving packets of information and has a unique device address. Also called a mobile terminal or mobile station.

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.

vCard - Internet Mail Consortium (IMC) electronic business card.

vCalendar - Internet Mail Consortium (IMC) electronic calendar record.

4.2. Abbreviations

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

API	Application Programming Interface
BNF	Backus-Naur Form
CGI	Client Gateway Interface
ECMA	European Computer Manufacturer Association
ETSI	European Telecommunication Standardization Institute
GSM	Global System for Mobile Communication
HDML	Handheld Markup Language [HDML2]
HTML	HyperText Markup Language [HTML4]

HTTP	HyperText Transfer Protocol [RFC2068]
IANA	Internet Assigned Number Authority
IMC	Internet Mail Consortium
LSB	Least Significant Bits
MSB	Most Significant Bits
MSC	Mobile Switch Center
PDA	Personal Digital Assistant
RFC	Request For Comments
SAP	Service Access Point
SGML	Standardized Generalized Markup Language [ISO8879]
SSL	Secure Socket Layer
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator [RFC1738]
URN	Uniform Resource Name
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
XML	Extensible Markup Language

5. Overview

5.1. Why Scripting?

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 unless you modify 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 the facilities of the phone 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 phone software and configuring a phone 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, supports more advanced UI functions, adds intelligence to the client, provides access to the phone and its peripherals, and reduces the amount of bandwidth needed to send data between the gateway and the client.

WMLScript is loosely based on JavaScript 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 new WMLScript language is to be as close as possible to 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 chapter *WMLScript Grammar* 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, and newlines that appear between tokens in programs, except those that are part of string constants.

6.1.3. Mandatory Semicolons

Each statement in WMLScript has to be followed by a semicolon.[1]

6.1.4. Comments

Comments follow the C++ style. Line comments start with // and end in the end of the line. Comments consisting of multiple lines start with /* and end with */. Nested comments are not supported.[2]

6.1.5. Literals

6.1.5.1. Integer Literals

Integer literals can be represented in three different ways: base-10 integers, octals, and hexadecimals.

Syntax:

DecimalIntegerLiteral **::**

`0`

NonZeroDigit DecimalDigits$_{opt}$

[1] Compatibility note: JavaScript supports optional semicolons.

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

NonZeroDigit **:: one of**

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

HexIntegerLiteral **::**

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

HexDigit **:: one of**

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

OctalIntegerLiteral **::**

`0` *OctalDigit*
OctalLiteral OctalDigit

OctalDigit **:: one of**

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

The minimum and maximum sizes[3] for integer literals are: –2147483648 and 2147483647. No automatic conversions from integer literals to floating-point literals are performed in cases of overflow.

Notice that these are the minimum and maximum sizes for the integer literals only. The minimum and maximum sizes for variables containing values of integer type are specified in the section 6.2.3.1.

6.1.5.2. Floating-Point Literals

Floating-point literals can contain a decimal point and they may contain an exponential part, too.

Syntax:

DecimalFloatLiteral **::**

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

DecimalDigits **::**

DecimalDigit
DecimalDigits DecimalDigit

ExponentPart **::**

ExponentIndicator SignedInteger

ExponentIndicator **:: one of**

`e` `E`

SignedInteger **::**

DecimalDigits
`+` *DecimalDigits*
`-` *DecimalDigits*

See section 6.2.3.2. for more information about floating-point precision and size.

---

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

6.1.5.3. String Literals

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

```
"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	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. Since there are two possible truth-values, the two boolean literals are:

```
true
false
```

6.1.6. Identifiers

Identifiers are used to name and refer to two different elements of WMLScript: variables and functions. Identifiers[4] cannot start with a digit but can start with an underscore (_). Examples of legal identifiers are:

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

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.

---

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

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 syntax specification):

```
break  continue  false  true void  while
```

6.2. Variables and Data Types

This section describes the two most important concepts of WMLScript language: variables and internal data types. A variable is a name associated with a data value, which allows us to store and manipulate data in our programs. WMLScript supports only local variables[5] and variables used as function parameters.

6.2.1. Variable Declaration

Variable declaration is compulsory[6] in WMLScript. Variable declaration is done simply by using the *var* keyword:

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

Uninitialised variables are initialised to contain an empty string ("").

6.2.2. Untyped Variables

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*. This means that the programmer does not have to specify variable types and any variable can contain any type of data at any given time. WMLScript will automatically convert between the different types as needed.

```
var number = 12;         // Integer
number     = "XII";      // String
var res    = "1" + "2";  // Equals to "12"
```

Some operators (see 6.3.1. for more information about assignment operators) require that the left operand is a variable. Thus, in addition to the four data types supported by WMLScript, a fifth type *variable* is used to specify that a variable name must be provided.

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

6.2.3. Numeric Values

WMLScript supports two different numeric variable values: *integer* and *floating-point* values. Variables can be initialised with integer and floating-point literals and several operators can be used to modify their values during the run-time. Integer values are converted to floating point values and *visa versa* if the operands of the operator require it.

---

5 Compatibility note: JavaScript supports global variables, too.

6 Compatibility note: JavaScript supports automatic declaration, too.

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

In cases where the value can be either an integer or a floating-point, a more generic type *number* is used instead.

6.2.3.1. Integer Size

The size (number of bits) of the integer is not specified. However, the minimum and maximum sizes[7] for integer values have to be at least: –134217728 and 134217727 (28 bits or more). The exact sizes are not known during the compilation time. Math library functions can be used to get these values during the run-time:

`Lang.maxInt()`	Maximum representable integer value supported (greater or equal to 134217727)
`Lang.minInt()`	Minimum representable integer value supported (smaller or equal to –134217728)

The size of the integer literal can be larger than the size reserved for the integer variables. When an integer literal larger than specified minimum or maximum sizes for integer values is assigned to a variable, the maximum (for positive values) or minimum (for negative values) is used instead.

6.2.3.2. Floating-point Size

The precision[8] of floating-point values is: 1.17549435E-38 and 3.40282347E+38 (both negative and positive values). They are represented by using single precision [IEEE754] floating-point numbers.

Special floating-point values are supported as part of the *Lang* library:

`Lang.maxFloat()`	Maximum representable floating-point value supported (3.40282347E+38)
`Lang.minFloat()`	Minimum representable floating-point value supported (-3.40282347E+38)
`Lang.floatPrecision()`	Smallest representable floating-point value supported (1.17549435E-38)

6.2.4. 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 supported by the standard string library.

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

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

[8] Compatibility note: JavaScript does not specify precision for floating point values.

6.2.5. 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 a logical expression (see 6.3.3. for more information).

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

6.3. Operators and Expressions

The following sections describe the different operations supported by WMLScript and how they can be used to form expressions that are more complex.

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
%=	remainder and assign
<<=	bitwise left shift and assign
>>=	bitwise right shift with sign and assign
>>>=	bitwise right shift zero fill and assign
&=	bitwise AND and assign
^=	bitwise XOR and assign
|=	bitwise OR 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)
-	minus
*	multiply
/	divide

Operator	Operation
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
-	negation
--	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 complement (unary)

Logical AND evaluates first the first operand and if it is *false* the second operand is not evaluated. If the first operand evaluates to *true* the second operand is evaluated, too. Similarly the logical OR evaluates the first operand and if it is *true*, the second operand is not evaluated at all. If the first operand if *false* the second operand is evaluated.

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

WMLScript requires a value of boolean type for logical operations. Conversions from other types to boolean type and *vice versa* are supported.

6.3.4. String Operators

WMLScript supports string catenation as a built-in operation. The + and += operators used with strings perform a concatenation on the strings. Other string operations[9] are supported by a special *String* library.

```
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

Examples:

```
var res = (myAmount > yourAmount);
```

6.3.6. Array Operators

WMLScript does not support arrays[10] as such. However, the standard *String* library 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:

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

Comma used in the function call to separate parameters and in the variable declarations to separate multiple variable declarations is not a comma operator. In these cases, the comma operator can only be used inside the parenthesis.

---

9 Compatibility note: JavaScript supports String objects and a length attribute for each string. Since WMLScript does not support objects, similar functionality is provided by the library support.

10 Compatibility note: JavaScript supports arrays.

```
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 also the conditional (**?:**) operator which can be used to selectively evaluate one of the given two operands based on the boolean value of the first operand. The result of the conditional operator is the value of the evaluated operand:

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

6.3.9. typeof Operator

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

Type	Code
Integer:	0
Float:	1
String:	2
Boolean:	3

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

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

6.3.10. Expressions

As a general-purpose language, 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
```

More complex expressions can be defined by using simple expressions together with operators and function calls.

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

6.3.11. Expression Bindings

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

Precedence[11]	**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 minus (negation)
1	R	~	integer	integer	bitwise complement (unary)
1	R	!	boolean	boolean	logical complement (unary)
1	R	typeof	any	integer	return internal data type (unary)
2	L	*	numbers	number	multiplication
2	L	/	numbers	floating-point	division
2	L	div	numbers	integer	integer division
2	L	%	numbers	integer	remainder
3	L	+, -	numbers	number	addition, subtraction
3	L	+	strings	string	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	L	? :	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 or string	number or string	assignment with addition or concatenation
13	R	<<=, >>=,	variable,	integer	assignment with bitwise

11 Binding: 0 binds tightest

Precedence[11]	**Associativity**	**Operator**	**Operand types**	**Result type**	**Operation performed**
		>>>=, &=, ^=, |=	integer		operation
14	L	,	any	any	multiple evaluation

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.
- 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 returns always a value. By default it is an empty string (`""`) but a *return* statement can be used to specify other return values.

Functions in WMLScript are not data types[12] 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. External functions can be called from outside the compilation unit in which they are defined. *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.

[12] Compatibility note: Functions in JavaScript 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 function is defined. *Local script functions* (defined inside the same compilation unit) can be called simply by providing the function name and a comma separated list of parameters (number of parameters must match the number of parameters[13] accepted by the function).

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

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

External function calls (functions defined in another compilation unit that can be referenced by a URL) can be called by specifying a mapping between the external unit and a name. This name and the hash symbol (#) can be used to prefix the standard function call syntax:

```
use url OtherDeck "http://www.host.com/deck.wml"

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

Standard library function calls (functions defined in a library that is part of the WAP standard [WMLStdLib]) can be called by using the standard name of the library (see 6.6. for more information). This name and the dot symbol (.) can be used to prefix the standard function call syntax:

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

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

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

[13] Compatibility note: JavaScript supports a variable number of arguments in a function call.

6.5. Statements

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

The following statements are available in WMLScript[14]: expression statement, compound statement, break, continue, for, if...else, return, var, while.

6.5.1. Expression Statement

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

Syntax:

ExpressionStatement **:**

Expression

Expression **:**

AssignmentExpression
Expression **,** *AssignmentExpression*

Examples:

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

6.5.2. 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 **;**

[14] Compatibility note: JavaScript supports also *for..in* and *with* statements.

Example:

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

6.5.3. var

This statement can be used to declare variables with initialisation (optional, variables are initialised to empty string ("") by default). The scope of the declared variable is the current block where the variable is declared.

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.

Examples:

```
function test() {
  var nbr_hits = 0,
      cust_nbr = 0;
};

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 and c can be used
  };
  return a;            // Only variable a is accessible
};
```

6.5.4. if...else

This statement can be used to specify conditional execution of statements. It consists of a condition and one or two statements and executes a statement if the specified condition is *true*. If the condition is *false*, another (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 the conversions) to a boolean value. If condition evaluates to `true`, the first statement is executed. If condition evaluates to `false`, 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...";
```

6.5.5. while

This statement can be 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: The *Expression* (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`, 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;
};
```

6.5.6. for

This statement can be 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 second *Expression* (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);
};
```

6.5.7. break

This statement can be used to terminate the current *while* or *for* loop and continue the program execution from the statement following the terminated loop.

Syntax:

BreakStatement **:**

```
break
```

Example:

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

6.5.8. continue

This statement can be 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.

Syntax:

ContinueStatement **:**

```
continue
```

Example:

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

6.5.9. return

This statement can be used inside the function body to specify the function return value.

Syntax:

ReturnStatement **:**

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

Example:

```
function square( x ) {
  if (!(Lang.isNumber(x))) return "NaN";
  return x * x;
};
```

6.6. Libraries

WMLScript supports the usage of libraries[15]. Libraries are named collections of functions that belong logically together. These functions can be called by using a special '.' (dot) operator with the library name and the function name with parameters:

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

6.6.1. Standard Libraries

WMLScript specifies a set of standard libraries that must be supported. Standard libraries are specified in more detail in the *WMLScript Standard Libraries Specification* [WMLStdLib].

6.7. Pragmas

WMLScript supports the usage of two different but related *pragmas*. Pragmas declare names that can be used in the following functions and are declared in the beginning of the compilation unit.

---

15 Compatibility note: JavaScript does not support libraries. It supports a set of predefined objects with attributes. WMLScript uses libraries to support similar functionality.

Syntax:

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

Pragmas **:**
ScriptPragma
LibraryPragma
Pragmas LibraryPragma
Pragmas ScriptPragma

ScriptPragma **:**
use url *ExternalScriptName StringLiteral* **;**$_{opt}$

LibraryPragma **:**
use lib *LibraryName StringLiteral*$_{opt}$ **;**$_{opt}$

The first pragma is used to specify the location (URL) and name of the external resource containing script functions and to give it a local name that can be used inside the following function definitions:

```
use url OtherDeck "http://www.host.com/app/deck.wml"

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

The second one is used to specify the libraries to be used inside a compilation unit:

```
use lib Lang   // Optional
use lib String // Optional

function testOK(str, index) {
  if (Lang.isInt(index)) {
    return (String.charAt(str,index) == "OK");
  } else {
    return "";
  };
};
```

This pragma can be used with the standard libraries [WMLStdLib] and future non-standard library[16] extension. However, standard libraries may be used without the pragma declaration, too.

[16] This pragma can be used to specify the location (URL) and the name of the *library interface specification* and to give it a local name that can be used inside the following function definitions. URL follows the standard URL syntax and encoding rules specified in [RFC1738]. Specifically, only a subset of US-ASCII characters are accepted as part of the URL string.

7. Automatic Data Type Conversions

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.

7.1. When to Convert?

WMLScript is a weakly typed language. That means that, on the language level, the variable declarations do not specify a type. However, internally the language handles the following 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

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

Conversions from one type to another are not always needed. This is the case in the following situations:

- When the operand(s) for the operation are required to be of certain type and the provided values are correct (mathematical operations when the operands are of type integer).
- When the operation accepts a set of types as its operands (+ operator does integer/floating-point addition, if the parameters are numbers and, string concatenation, if the parameters are strings).

Conversion from one type to another is required when an operation requires that the operands are of certain type and at least one of them is of different type. Two different cases have to be handled:

- *When the operand(s) for the operation are required to be of certain type*: the conversions in this case are simple; just convert (if possible) the parameters to the required type and perform the operation (mathematical operations such as -, / and *).
- *When the operation accepts a set of types as its operands and the operation depends on the type of the given parameters*: the conversions are performed based on the type and identity (divisor, dividend etc.) of the operands.

The first case is simple and the following sections describe how conversions from one type to another are done. The following is a table of operations requiring a specific type as its operand(s):

Operator	**Operand types**	**Operation performed**
++	number	pre- or post-increment (unary)
--	number	pre- or post-decrement (unary)
-	number	unary minus (negation)
~	integer	bitwise complement (unary)
!	boolean	logical complement (unary)
*. /, div, %	numbers	multiplication, division, integer division, remainder
-	numbers	addition, subtraction
<<	integers	bitwise left shift

Operator	Operand types	Operation performed
>>	integers	bitwise right shift with sign
>>>	integers	bitwise right shift with zero fill
&	integers	bitwise AND
^	integers	bitwise XOR
|	integers	bitwise OR
&&	booleans	logical AND
||	booleans	logical OR
*=, /=, %=, -=	variable and number	assignment with numeric operation
<<=, >>=, >>>=, &=, ^=, |=	variable and integer	assignment with bitwise operation

The second case requires standard rules to be specified on how the operand types define which operation is performed and when the conversions take place in case they are needed. The following table contains the conversion rules:

Operator	Operand types	Operation performed
+	numbers or strings	▪ If the operands are of type number then perform a mathematical addition, otherwise ▪ if the operands are of type boolean then convert the boolean values to numbers and perform a mathematical addition, otherwise ▪ if one of the operands is of type number and the other one is of type boolean, convert the boolean value to a number and perform a mathematical addition, otherwise ▪ perform a string concatenation (automatic conversion of boolean and number types to strings)
<, <=, >, >=	booleans, numbers or strings	▪ If the operands are of type number then perform a mathematical comparison, otherwise ▪ if the operands are of type boolean then convert the boolean values to numbers and perform a mathematical comparison, otherwise ▪ if one of the operands is of type number and the other one is of type boolean, convert the boolean value to a number and perform a mathematical comparison, otherwise ▪ if the operands are of type string then perform a string comparison, otherwise ▪ if the operands can be converted into numbers then perform a mathematical comparison, otherwise ▪ return `"NaN"` (*Not a Number*)
==, !=	booleans, numbers or strings	▪ If the operands are of type boolean then perform a boolean comparison, otherwise ▪ if the operands are of type number then perform a mathematical comparison, otherwise

Operator	Operand types	Operation performed
		▪ if one of the operands is of type number and the other one is of type boolean, convert the boolean value to a number and perform a mathematical comparison, otherwise ▪ if the operands are of type string then perform a string comparison, otherwise ▪ if the operands can be converted into numbers then perform a mathematical comparison, otherwise ▪ return `false`
=	variable and any type	Assignment: always performed
+=	variable and number or string	▪ If the operands are of type number then perform a mathematical addition, otherwise ▪ if the operands are of type boolean then convert the boolean values to numbers and perform a mathematical addition, otherwise ▪ if one of the operands is of type number and the other one is of type boolean, convert the boolean value to a number and perform a mathematical addition, otherwise ▪ perform a string concatenation (automatic conversion of boolean and number types to strings)
,	any type	Evaluation: always performed

7.2. Conversions to String

Automatic conversion to strings is performed every time a string type is needed for the operation. The conversion is done in the following way:

- Integer values are converted to strings in the obvious way: the string contains the digits of the decimal representation of the number (345123 => "345123").
- Floating-point values are converted to strings in the following way: the string contains the digits of the decimal representation of the floating-point number (345.123 => "345.123") in an implementation-specific way.
- The boolean value `true` is converted to string `"true"`, and the value `false` is converted to the string `"false"`.

7.3. Conversions to Numbers

Automatic conversion to numbers is performed every time a number type is needed for the operation. The conversion takes place when a numeric argument is needed for a build in function or when various arithmetic, comparison, etc. operators need a numeric operand. The conversion is done in the following way:

- If a string contains the decimal representation of an integer or floating-point number (with no trailing non-numeric characters), the string is converted to that number. As a special case, the empty string (`""`) is converted to the number 0. If the conversion fails, the result is a string `"NaN"` (*Not a Number*)[17].
- The boolean value `true` is converted to integer value 1, `false` to 0.

7.4. Conversions to Booleans

Automatic conversion to boolean is performed every time a boolean type is expected in an expression. The conversion is done in the following way:

- The integer/floating-point value 0 is converted to `false`. All other numbers are converted to `true`.
- The empty string (`""`) is converted to `false`. All other strings are converted to `true`.

[17] Compatibility note: JavaScript displays an error message.

8. WMLScript 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, some sort of user controlled abortion mechanism is needed.

8.1. Error Detection

Error detection in WMLScript means that the programmers are able to check for errors before they occur. The following cases are possible:

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

The goal of error detection is to give tools for the programmer to detect errors (if possible) that would lead to erroneous behaviour.

8.2. Error Handling

Error handling takes place after an error has 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.
- *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.

8.3. Fatal Errors

8.3.1. Memory Errors

These errors are related to the dynamic behaviour of the WMLScript Bytecode Interpreter and its memory usage.

8.3.1.1. StackOverflow

Description: Indicates a stack overflow.

Generated:	At any time, when an application recourses too deep or attempts to push too many variables onto the stack.
Example:	`function f(x) { f(x+1); };`
Severity:	Fatal
Predictable:	No.
Solution:	Halt program. Pop up a dialog box to the user and exit application gracefully.

8.3.1.2. StackUnderflow

Description:	Indicates a stack underflow because of an application error (compiler generated bad code).
Generated:	At any time, when an application attempts to pop an empty stack.
Example:	Only generated if compiler generates bad code
Severity:	Fatal
Predictable:	No
Solution:	Halt program. Pop up a dialog box to the user and exit application gracefully.

8.3.1.3. OutOfMemory

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

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

Severity:	Fatal
Predictable:	No
Solution:	Halt program. Pop up a dialog box to the user and exit application gracefully.

8.3.2. Invalid Instruction Errors

These errors are related to the instructions being executed by the WMLScript Bytecode Interpreter. They may be indications of invalid instructions or instructions that cannot be completed.

8.3.2.1. InvalidOpcode

Description:	Reports that the executed bytecode contains an invalid opcode.
Generated:	At any time.
Example:	Compiler generates an invalid opcode that is not specified in the WMLScript bytecode.
Severity:	Fatal
Predictable:	No
Solution:	Either halt program at the place an invalid opcode is encountered or before the bytecode is loaded for execution (if bytecode verification is used). Pop up a dialog box to the user and exit application gracefully.

8.3.2.2. InvalidFunctionCall

Description:	Reports an invalid function call (erroneous call generated by compiler).
Generated:	At any time, when an application attempts to call a function.
Example:	Compiler generates non-existing function index for a local function call.
Severity:	Fatal

Predictable:	No
Solution:	Halt program. Pop up a dialog box to the user and exit application gracefully.

8.3.2.3. FunctionNotFound

Description:	Reports that a call to a function could not be completed.
Generated:	At any time, when an application attempts to call a function.
Example:	`var a = 3*OtherDeck#doThis(param);`
Severity:	Fatal
Predictable:	No
Solution:	Halt program. Pop up a dialog box to the user and exit application gracefully.

8.3.3. External Exceptions

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

8.3.3.1. User Initiated

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

8.3.3.2. System Initiated

Description:	Indicates that an external fatal exception occurred while an application 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:	Exit program.

8.4. Non-Fatal Errors

8.4.1. Arithmetic Errors

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

8.4.1.1. DivideByZeroError

Description:	Indicates that a division by zero in the application
Generated:	At any time, when an application attempts to divide by 0.

Example:	`var a = 10;`
	`var b = 0;`
	`var z = a / b;`

Severity:	Non-fatal

Predictable: Yes
Solution: The result is a string `"NaN"`.

8.4.1.2. IntegerOverflow

Description: Reports an arithmetic overflow of integers (an integer with value larger than MAX_INT or smaller than MIN_INT)
Generated: At any time, when an application attempts to evaluate an integer expression resulting in a too large number.
Example:
```
var a = Lang.maxInt();
var b = Lang.maxInt();
var c = a + b;
```
Severity: Non-fatal
Predictable: Yes (but difficult in certain cases)
Solution: A rollover takes place.

8.4.1.3. FloatOverflow

Description: Reports an arithmetic overflow (a number with value larger than MAX_VALUE)
Generated: At any time, when an application attempts to set a variable larger to the MAX_VALUE.
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 a string `"NaN"`.

8.4.1.4. FloatUnderflow

Description: Reports an arithmetic underflow (a number with value smaller than MIN_VALUE)
Generated: At any time, when an application attempts to set a variable smaller to the MIN_VALUE
Example:
```
var a = 2.25e-308;
var b = 2.25e-308;
var c = a * b;
```
Severity: Non-fatal
Predictable: Yes (but difficult in certain cases)
Solution: The result is 0.

8.4.2. Conversion Errors

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

8.4.2.1. NotaNumber

Description: Reports an invalid conversion of a string to a number (the string does not have an appropriate format).
Generated: At any time, when any function tries to convert a string to a numeric number.
Example:
```
var a = “This”;
var b = Lang.parseInt(a);   // b = "NaN"
var c = 1/b;                // c = "NaN"
var d = Lang.parseFloat(c); // d = "NaN"
```
Severity: Non-fatal

Predictable:	Yes
Solution:	The result is a string `"NaN"`

8.5. 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 user mechanisms by which errors can be detected before they happen.
- Use the same error handling mechanisms as WMLScript operations and build-in function in cases where error should be reported back to the caller.
- Minimise the possibility of fatal errors in library functions.

9. WMLScript Grammar

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

9.1. 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.

9.1.1. 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.

9.1.2. Lexical Grammar

A *lexical grammar* for WMLScript is given in section 9.2. 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. A multi-line comment is likewise 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.

9.1.3. Syntactic Grammar

The *syntactic grammar* for WMLScript is given in section 9.3. 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.

9.1.4. 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

9.1.5. 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.

9.2. WMLScript Lexical Grammar

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

SourceCharacter **::**
any Unicode character

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

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

Keyword **:: one of**

break	**extern**	**lib**	**var**
continue	**for**	**return**	**while**
div	**function**	**typeof**	**url**
else	**if**	**use**	

KeywordNotUsedByWMLScript **:: one of**

delete	**this**
in	**void**
new	**with**
null	

FutureReservedWord **:: one of**

case	**do**	**sizeof**	**switch**
const	**import**	**struct**	

Identifier **::**

IdentifierName **but not** *ReservedWord*

IdentifierName **::**

IdentifierLetter
IdentifierName IdentifierLetter
IdentifierName DecimalDigit

IdentifierLetter **:: one of**[18]

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**[19]

=	**>**	**<**	**==**	**<=**	**>=**
!=	**,**	**!**	**~**	**?**	**:**
.	**&&**	**||**	**++**	**--**	**+**
-	*****	**/**	**&**	**|**	**^**
%	**<<**	**>>**	**>>>**	**+=**	**-=**
***=**	**/=**	**&=**	**|=**	**^=**	**%=**

[18] Compatibility note: JavaScript supports the usage of dollar sign ($) in identifier names, too.

[19] Compatibility note: JavaScript supports arrays and square brackets ([ ]), too.

```
<<=     >>=     >>>=     (     )     {
}       ;
```

Literal **::**[20]

BooleanLiteral
NumericLiteral
StringLiteral

BooleanLiteral **::**[21]

`true`
`false`

NumericLiteral **::**

DecimalIntegerLiteral
HexIntegerLiteral
OctalIntegerLiteral
DecimalFloatLiteral

DecimalIntegerLiteral **::**

`0`
NonZeroDigit DecimalDigits$_{opt}$

NonZeroDigit **:: one of**

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

HexIntegerLiteral **::**

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

HexDigit **:: one of**

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

OctalIntegerLiteral **::**

`0` *OctalDigit*
OctalLiteral OctalDigit

OctalDigit **:: one of**

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

DecimalFloatLiteral **::**

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

DecimalDigits **::**

DecimalDigit
DecimalDigits DecimalDigit

ExponentPart **::**

ExponentIndicator SignedInteger

---

20 Compatibility note: JavaScript supports *Null* literal, too.

21 Compatibility note: JavaScript supports both lower and upper case boolean literals.

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*

9.3. WMLScript Syntactic Grammar

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

PrimaryExpression **:**[22]
Identifier
Literal
(*Expression* **)**

CallExpression **:** [23]
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 **--**

---

22 Compatibility note: JavaScript supports objects and *this*, too.

23 Compatibility note: JavaScript 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 **:**[24]

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

MultiplicativeExpression **:**[25]

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

AdditiveExpression **:**

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

ShiftExpression **:**

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

RelationalExpression **:**

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

EqualityExpression **:**

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

BitwiseANDExpression **:**

EqualityExpression
BitwiseANDExpression **&** *EqualityExpression*

BitwiseXORExpression **:**

BitwiseANDExpression
BitwiseXORExpression **^** *BitwiseANDExpression*

---

24 Compatibility note: JavaScript operators *delete*, *void* and unary + are not supported. *parseInt* and *parseFloat* are supported as library functions.

25 Compatibility note: Integer division (*div*) is not supported by JavaScript.

BitwiseORExpression **:**
BitwiseXORExpression
BitwiseORExpression **|** *BitwiseXORExpression*

LogicalANDExpression **:**
BitwiseORExpression
LogicalANDExpression **&&** *BitwiseORExpression*

LogicalORExpression **:**
LogicalANDExpression
LogicalORExpression **||** *LogicalANDExpression*

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

AssignmentExpression **:**
ConditionalExpression
AssignmentList

AssignmentList **:**
Identifier AssignmentOperator AssignmentExpression

AssignmentOperator **:: one of**
= *= /= %= += -= <<= >>= >>>= &= ^= |=

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

Statement **:**[26]
Block
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement

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

StatementList **:**
Statement **;**
StatementList Statement **;**

VariableStatement **:**
var *VariableDeclarationList*

---

26 Compatibility note: JavaScript *with* statement is not supported.

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

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

VariableInitializer **:**
= *ConditionalExpression*

EmptyStatement **:**
;

ExpressionStatement **:**
Expression

IfStatement **:**[27]
if (*Expression* **)** *Statement* **else** *Statement*
if (*Expression* **)** *Statement*

IterationStatement **:**[28]
WhileStatement
ForStatement

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

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

ContinueStatement **:**
continue

BreakStatement **:**
break

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

FunctionDeclaration **:**[29]
extern$_{opt}$ **function** *Identifier* **(** *FormalParameterList*$_{opt}$ **)** *Block* **;**$_{opt}$

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

---

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

28 Compatibility note: JavaScript *for in* statement is not supported.

29 Compatibility note: JavaScript does not support keyword *extern*.

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

Pragmas **:**[30]
ScriptPragma
LibraryPragma
Pragmas LibraryPragma
Pragmas ScriptPragma

LibraryPragma **:**
use lib *LibraryName StringLiteral*$_{opt}$ **;**$_{opt}$

ScriptPragma **:**
use url *ExternalScriptName StringLiteral* **;**$_{opt}$

FunctionDeclarations **:**
FunctionDeclaration
FunctionDeclarations FunctionDeclaration

30 Compatibility note: JavaScript does not support *pragmas*.

10. WMLScript Binary Encoding

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.

10.1. WMLScript Compilation

WMLScript compiler encodes one WMLScript compilation unit into WMLScript bytecode using the encoding rules presented in the following sections. A WMLScript *compilation unit* (see section 9.1.3.) 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.

10.2. WMLScript 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:

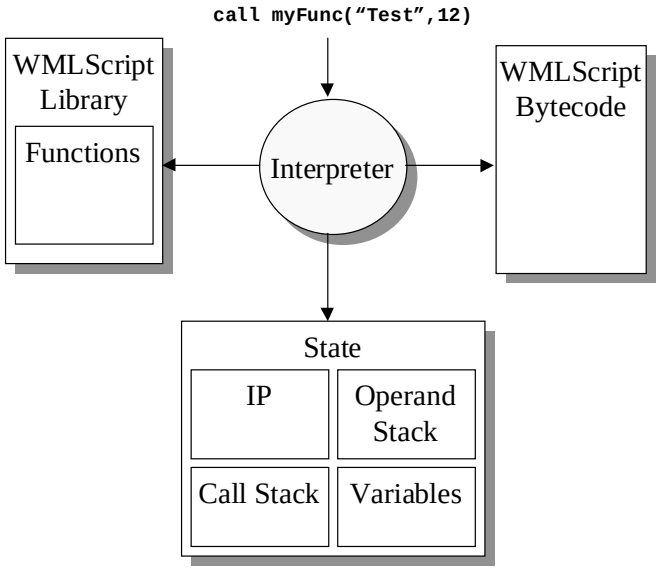

Figure 1: General architecture of the WMLScript interpreter

The WMLScript interpreter can be used to call and execute functions in a compilation unit encoded as WMLScript bytecode. Each function in the bytecode defines the number of *parameters* and the *opcodes* used to express its behaviour. Thus, a call to a WMLScript function must specify the called function and its parameters. Once the execution is finished, the WMLScript interpreter returns the *control* and the *return value* back to the caller.

Execution of a WMLScript function means executing the opcodes residing in the WMLScript bytecode. While a function is being executed, the WMLScript interpreter maintains the following *state* information:

- *IP (Instruction Pointer)*: This points to an opcode in the bytecode that is being executed.
- *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.

10.3. Conventions

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

10.3.1. 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
u_int8	8 bit unsigned integer
int16	16 bit signed integer
u_int16	16 bit unsigned integer
int32	32 bit signed integer
u_int32	32 bit unsigned integer
mb_u_int32	32 bit unsigned integer, in multi-byte integer format. See 10.3.2. 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.

10.3.2. 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`.

10.3.3. Character Encoding

The encoding of all *constant strings* in the WMLScript bytecode is specified by transport or container meta-information, and is expected to use the same mechanisms as the textual WMLScript format. Specifically, it is assumed that a *charset* declaration will accompany the WMLScript content in any form, and will indicate the encoding of all strings. The WMLScript bytecode can support any string encoding. As with the textual format of WMLScript, it is also assumed that WMLScript constructs and identifiers can be represented in the target character encoding.

URLs are written only with the graphic printable characters of the US-ASCII coded character set. Thus, the *URL constants* in the constant pool use a fixed 8-bit US-ASCII encoding.

Function names in WMLScript are written by using a subset of US-ASCII characters. Thus, the *function names* in the function name table use a fixed 8-bit US-ASCII encoding, too.

10.3.4. Presentation Format

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 format is presented using the following table format:

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.
The name of the next section. Any number of sections can be presented in one table.		
...		

The following rules apply:

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

10.4. WMLScript Bytecode

From the encoding point of view, the WMLScript language 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 10.5.	Contains general information related to the bytecode.
ConstantPool	See 10.6.	Contains the information of all constants specified as part of the WMLScript functions that are encoded into bytecode.
FunctionPool	See 10.7.	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.

10.5. 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 language and its encoding. The version byte contains the *major version* minus one in the upper 4 bits and the *minor version* in the lower 4 bits. Thus, version number 1.0 is encoded as `0x00`.
CodeSize	mb_u_int32	The size of the rest of the bytecode (not including the version number and this variable) in bytes

10.6. 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 opcodes use this index to refer to specific constants.

Name	Data type and size	Comment
NumberOfConstants	u_int16	Specifies how many constants are encoded in this pool.
Constants...	See 10.6.1.	Contains the definitions for each constant in the constant pool. The number of constants is specified by NumberOfConstants.

10.6.1. 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 10.6.1.1., 10.6.1.2., 10.6.1.3., and 10.6.1.4.	Type specific value definition.

The following encoding for constant types is used:

Code	Type
0	8 bit signed integer
1	16 bit signed integer
2	32 bit signed integer
3	32 bit float
4	String
5	URL (US-ASCII encoding)
6-255	Reserved for future use

10.6.1.1. Integers

WMLScript bytecode supports 8 bit, 16 bit and 32 bit signed integer constants.

10.6.1.1.1. 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.

10.6.1.1.2. 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.

10.6.1.1.3. 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.

10.6.1.2. 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.

10.6.1.3. Strings

Strings are encoded by first specifying their length and then the content:

Name	Data type and size	Comment
StringSize	mb_u_int32	The size of the following string in bytes (not containing this variable).
ConstantString	StringSize bytes	The value of the string constant. See 10.3.3. for more information about string encoding.

The used string encoding is not specified in the bytecode. See 10.3.3. for more information about encoding.

10.6.1.4. URLs

URLs are encoded by first specifying their length and then the content.

Name	Data type and size	Comment
URLSize	mb_u_int32	The size of the following URL in bytes (not containing this variable).
ConstantString	URLSize bytes	The value of the URL constant. See 10.3.3. for more information about URL encoding.

10.7. Function Pool

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 opcodes 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 10.7.1.	Function name table contains the names of all external functions present in the bytecode.
Functions...	See 10.7.2.	Contains the bytecode for each function.

10.7.1. 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

Name	Data type and size	Comment
		following table.
FunctionNames...	See 10.7.1.1.	Each of the external function name represented in the same order as the functions are stored into the function pool.

10.7.1.1. Function Names

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

Name	Data type and size	Comment
FunctionIndex	u_int8	The index of the external function for which the following name if 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. See 10.3.3. for more information about function name encoding.

10.7.2. Functions

Each function is defined by its prologue and its opcodes:

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 specified by the function.
FunctionSize	mb_u_int32	Size of the following opcodes (not including this variable) in bytes.
Opcodes...	See 10.7.2.1.	Contains the function opcodes.

10.7.2.1. Opcodes

The idea of opcodes is to specify a set of assembly level instructions that can be used to encode the operations of the script. These instructions are defined in such a way that they are easy to implement on a variety of platforms.

WMLScript opcodes have been defined so that they provide at least the minimal set of instructions by which WMLScript language operations can be presented. Since the bytecode is being transferred from the gateway to the client through a narrowband connection, the selected opcodes have been optimised so that the compilers can generate code of minimal size. In some cases, this has meant that we have several opcodes for the same operation depending on how much space is needed to encode the information.

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 opcodes	Used for
1XXPPPPP	4	JUMP_FW_S, JUMP_BW_S, TJUMP_FW_S, LOAD_VAR_S
010XPPPP	2	STORE_VAR_S, LOAD_CONST_S
011XXPPP	4	CALL_S, CALL_LIB_S, CALL_URL_S, INCR_VAR_S
00XXXXXX	63	The rest of the opcodes
00000000	1	ESC, Reserved

The following table contains the definitions of opcodes for WMLScript where each opcode and its parameters are given a symbolic name. They are followed by the encoding of the opcode into 8-bit value. Some opcodes are optimised and can contain an *implicit parameter* as part of the encoding meaning that a set of bits from the 8 bit encoding is reserved for a parameter value. The next column describes the operation of the opcode, its parameters, and the effects they have on the execution and operand stack. Finally, the effect on the operand stack are described by using notation where the part before the arrow (⇨) represent the stack before the opcode has been executed and the part after the arrow the stack after the execution.

Opcode name	Opcode	Description	Operand Stack
JUMP_FW_S	100iiiii *iiiii* is the implicit unsigned *offset*	Jump forward to an offset. Execution proceeds at the given *offset* from the address of the first byte following this instruction. *Offset* is an unsigned 5-bit integer in the range of 0..31	No change
JUMP_FW *offset*	00000001	Jump forward to an offset. Execution proceeds at the given *offset* from the address of the first byte following this instruction. *Offset* is an unsigned 8-bit integer in the range of 0..255	No change
JUMP_FW_W *offset1* *offset2*	00000010	Jump forward to an offset. Execution proceeds at the given *offset* from the address of the first byte following this instruction. *Offset* is an unsigned 16-bit integer <*offset1*, *offset2*> in the range of 0..65535.	No change
JUMP_BW_S	101iiiii *iiiii* is the implicit unsigned *offset*	Jump backward to an offset. Execution proceeds at the given *offset* from the address of this instruction. *Offset* is an unsigned 5-bit integer in the range of 0..31	No change
JUMP_BW *offset*	00000011	Jump backward to an offset. Execution proceeds at the given *offset* from the address of this instruction. *Offset* is an unsigned 8-bit integer in the range of 0..255	No change
JUMP_BW_W *offset1* *offset2*	00000100	Jump backward to an offset. Execution proceeds at the given *offset* from the address of this instruction. *Offset* is an unsigned 16-bit integer <*offset1*, *offset2*> in the range of 0..65535.	No change
TJUMP_FW_S	110iiiii *iiiii* is the implicit unsigned *offset*	Pop a value from the operand stack and jump forward to an *offset* if the value is *false*. Execution proceeds at the given *offset* from the address of the first byte following this instruction. Otherwise, the execution continues at the next instruction.	…, value ⇨ …

Opcode name	Opcode	Description	Operand Stack
		Offset is an unsigned 5-bit integer in the range of 0..31	
TJUMP_FW *offset*	00000101	Pop a value from the operand stack and jump forward to an *offset* if the value is *false*. Execution proceeds at the given *offset* from the address of the first byte following this instruction. Otherwise, the execution continues at the next instruction. *Offset* is an unsigned 8-bit integer in the range of 0..255	…, value ⇨ …
TJUMP_FW_W *offset1* *offset2*	00000110	Pop a value from the operand stack and jump forward to an *offset* if the value is *false*. Execution proceeds at the given *offset* from the address of the first byte following this instruction. Otherwise, the execution continues at the next instruction. *Offset* is an unsigned 16-bit integer <*offset1, offset2*> in the range of 0..65535.	…, value ⇨ …
TJUMP_BW *offset*	00000111	Pop a value from the operand stack and branch backward to an *offset* if the value is *false*. Execution proceeds at the given *offset* from the address of this instruction. Otherwise, the execution continues at the next instruction. *Offset* is an unsigned 8-bit integer in the range of 0..255	…, value ⇨ …
TJUMP_BW_W *offset1* *offset2*	00001000	Pop a value from the operand stack and branch backward to an *offset* if the value is *false*. Execution proceeds at the given *offset* from the address of this instruction. Otherwise, the execution continues at the next instruction. *Offset* is an unsigned 16-bit integer <*offset1, offset2*> in the range of 0..65535	…, value ⇨ …
CALL_S	01100iii *iii* is the implicit *findex*	Calls a local function defined in the same function pool. Execution proceeds from the first instruction of the function *findex*. *Findex* is an unsigned 3-bit integer in the range of 0..7	…, [arg1, [arg2 …]] ⇨ …, ret-value
CALL *findex*	00001001	Calls a local function defined in the same function pool. Execution proceeds from the first instruction of the function *findex*. *Findex* is an unsigned 8-bit integer in the range of 0..255	…, [arg1, [arg2 …]] ⇨ …, ret-value
CALL_LIB_S *lindex*	01101iii *iii* is the implicit *findex*	Calls a library function *findex* defined in the specified library *lindex*. *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	…, [arg1, [arg2 …]] ⇨ …, ret-value
CALL_LIB *findex* *lindex*	00001010	Calls a library function *findex* defined in the specified library *lindex*. *Findex* is an unsigned 8-bit integer in the range of 0..255 *Lindex* is an unsigned 8-bit integer in the	…, [arg1, [arg2 …]] ⇨ …, ret-value

Opcode name	Opcode	Description	Operand Stack
		range of 0..255	
CALL_LIB_W *findex* *lindex1* *lindex2*	00001011	Calls a library function *findex* defined in the specified library *lindex*. *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	…, [arg1, [arg2 …]] ⇨ …, ret-value
CALL_URL_S	01110iii *iii* is the implicit *urlindex*	Calls a function defined in the specified URL address *urlindex*. *Urlindex* is an unsigned 3-bit integer in the range of 0..7 that points to the constant pool containing the actual URL	…, [arg1, [arg2 …]] ⇨ …, ret-value
CALL_URL *urlindex*	00001100	Calls a function defined in the specified URL address *urlindex*. *Urlindex* is an unsigned 8-bit integer in the range of 0..255 that points to the constant pool containing the actual URL	…, [arg1, [arg2 …]] ⇨ …, ret-value
LOAD_VAR_S	111iiiii *iiiii* is the implicit *vindex*	Pushes the value of the variable *vindex* onto the operand stack. *Vindex* is an unsigned 5-bit integer in the range of 0..31	… ⇨ …, value
LOAD_VAR *vindex*	00001101	Pushes the value of the variable *vindex* onto the operand stack. *Vindex* is an unsigned 8-bit integer in the range of 0..255	… ⇨ …, value
STORE_VAR_S	0100iiii *iiii* is the implicit *vindex*	Pops the value from the operand stack and stores it into the variable *vindex*. *Vindex* is an unsigned 4-bit integer in the range of 0..15	…, value ⇨ …
STORE_VAR *vindex*	00001110	Pops the value from the operand stack and stores it into the variable *vindex*. *Vindex* is an unsigned 8-bit integer in the range of 0..255	…, value ⇨ …
LOAD_CONST_S	0101iiii *iiii* is the implicit *cindex*	Pushes the value of the constant *cindex* onto the operand stack. *Cindex* is an unsigned 4-bit integer in the range of 0..15	… ⇨ …, value
LOAD_CONST *cindex*	00001111	Pushes the value of the constant *cindex* onto the operand stack. *Cindex* is an unsigned 8-bit integer in the range of 0..255	… ⇨ …, value
LOAD_CONST_W *cindex1* *cindex2*	00010000	Pushes the value of the constant *cindex* onto the operand stack. *Cindex* is an unsigned 16-bit integer <*cindex1,cindex2*> in the range of 0..65535	… ⇨ …, value
INCR_VAR_S	01111iii *iii* is the implicit *vindex*	Increments the value of a variable *vindex* by one. *Vindex* is an unsigned 3-bit integer in the range of 0..7	No change
INCR_VAR *vindex*	00010001	Increments the value of a variable *vindex* by one. *Vindex* is an unsigned 8-bit integer in the	No change

Opcode name	Opcode	Description	Operand Stack
		range of 0..255	
DECR_VAR *vindex*	00010010	Decrements the value of a variable *vindex* by one. *Vindex* is an unsigned 8-bit integer in the range of 0..255	No change
INCR	00010011	Increments the value on the top of the operand stack by one.	…, value ⇨ …, value+1
DECR	00010100	Decrements the value on the top of the operand stack by one.	…, value ⇨ …, value-1
ADD_ASG *vindex*	00010101	Pops a value from the operand stack and adds the value to the variable *vindex*. *Vindex* is an unsigned 8-bit integer in the range of 0..255	…, value ⇨ …
SUB_ASG *vindex*	00010110	Pops a value (subtractor) from the operand stack and subtracts the value from the variable *vindex*. *Vindex* is an unsigned 8-bit integer in the range of 0..255	…, value ⇨ …
UMINUS	00010111	Pops a value from the operand stack and performs a unary minus operation on it and pushes the result back to the operand stack.	…, value ⇨ …, -value
ADD	00011000	Pops two values from the operand stack and performs an add operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1+value2
SUB	00011001	Pops two values from the operand stack and performs a subtract operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1-value2
MUL	00011010	Pops two values from the operand stack and performs a multiplication operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1*value2
DIV	00011011	Pops two values from the operand stack and performs a division operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1/value2
REM	00011100	Pops two values from the operand stack and performs a reminder operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1 % value2
B_AND	00011101	Pops two values from the operand stack and performs a bitwise and operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1 & value2
B_OR	00011110	Pops two values from the operand stack and performs a bitwise or operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1 | value2
B_XOR	00011111	Pops two values from the operand stack and performs a bitwise xor operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1 ^ value2
LSHIFT	00100000	Pops two values from the operand stack and performs a bitwise left-shift operation on them and pushes the result back to the operand stack.	…, value, amount ⇨ …, value1 << value2
RSSHIFT	00100001	Pops two values from the operand stack and performs a bitwise signed right-shift operation	…, value, amount ⇨ …, value1 >> value2

Opcode name	Opcode	Description	Operand Stack
		on them and pushes the result back to the operand stack.	
RSZSHIFT	00100010	Pops two values from the operand stack and performs a bitwise right-shift with zero operation on them and pushes the result back to the operand stack.	…, value, amount ⇨ …, value1 >>> value2
EQ	00100011	Pops two values from the operand stack and performs a logical equality operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1 EQ value2
LE	00100100	Pops two values from the operand stack and performs a logical larger-or-equal operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1 LE value2
LT	00100101	Pops two values from the operand stack and performs a logical larger-than operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1 LT value2
GE	00100110	Pops two values from the operand stack and performs a logical greater-or-equal operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1 GE value2
GT	00100111	Pops two values from the operand stack and performs a greater-than operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1 GT value2
NE	00101000	Pops two values from the operand stack and performs a logical not-equal operation on them and pushes the result back to the operand stack.	…, value1, value2 ⇨ …, value1 NE value2
NEG	00101001	Pops a value from the operand stack and performs a logical negation operation on it and pushes the result back to the operand stack.	…, value ⇨ …, !value
COMPL	00101010	Pops a value from the operand stack and performs a bitwise complement operation on it and pushes the result back to the operand stack.	…, value ⇨ …, ~value
RETURN	00101011	Pops a return value from the operand stack. The execution continues at the next instruction following the function call of the calling function.	…, ret-value ⇨ …
POP	00101100	Pops a value from the operand stack.	…, value ⇨ …
DUP	00101101	Duplicates the value on top of the operand stack.	…, value ⇨ …, value, value
CONST_0	00101110	Loads an integer value 0 to the operand stack.	… ⇨ …, value_0
CONST_1	00101111	Loads an integer value 1 to the operand stack.	… ⇨ …, value_1
CONST_M1	00110000	Loads an integer value –1 to the operand stack.	… ⇨ …, value_-1
CONST_ES	00110001	Loads an empty string to the operand stack.	… ⇨ …, value_""

Opcode name	Opcode	Description	Operand Stack
CONST_NaN	00110010	Loads "NaN" (Not a Number) string to the operand stack.	… ⇨ …, value_"NaN"
CONST_TRUE	00110011	Loads a boolean value `true` to the operand stack.	… ⇨ …, value_true
CONST_FALSE	00110100	Loads a boolean value `false` to the operand stack.	… ⇨ …, value_false
TYPEOF	00110101	Pops a value from the operand stack and checks if it is a boolean, integer, float or string. Returns the result as an integer to the operand stack. The possible results are: 0 = Integer, 1 = Float, 2 = String, 3 = Boolean	…, value ⇨ …, typeof
IS_INT	00110110	Pops a value from the operand stack and checks if it is of type integer or can be converted into an integer. Returns the boolean result to the operand stack.	…, value ⇨ …, boolean
IS_FLOAT	00110111	Pops a value from the operand stack and checks if it is of type float or can be converted into a float. Returns the boolean result to the operand stack.	…, value ⇨ …, boolean
IS_NUMBER	00111000	Pops a value from the operand stack and checks if it is either of type integer or float or can be converted into an integer or a float. Returns the boolean result to the operand stack.	…, value ⇨ …, boolean
PARSE_INT	00111001	Pops a value from the operand stack and converts it into an integer value and pushes it into the operand stack. If the conversion fails, `"NaN"` is pushed into the operand stack.	…, value ⇨ …, integer
PARSE_FLOAT	00111010	Pops a value from the operand stack and converts it into a float value and pushes it into the operand stack. If the conversion fails, `"NaN"` is pushed into the operand stack.	…, value ⇨ …, float
RETURN_ES	00111011	Returns an empty string. The execution continues at the next instruction following the function call of the calling function.	No change
UNDEFINED	00111100	Reserved for the future.	Not defined
UNDEFINED	00111101	Reserved for the future.	Not defined
UNDEFINED	00111110	Reserved for the future.	Not defined
DEBUG	00111111	Reserved for debugging and profiling purposes.	No change
ESC	00000000	Reserved for future use.	Not defined

10.7.2.2. Opcode Encoding Rules

The following rules apply to the encoding of WMLScript functions into opcodes:

- **Order of function arguments in the operand stack**: 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; first argument is the lowest argument in the operand stack and the last parameter is on top of the operand stack.

- **End of function**: WMLScript function must return an empty string in case the end of the function is encountered without a return statement.
- **Special opcodes for library function call optimisation**: A set of opcodes (IS_INT, IS_FLOAT, IS_NUMBER, PARSE_INT, PARSE_FLOAT) may be used to compile certain library function calls into a more optimised format. The behaviour of the opcode and the corresponding function call must be identical. See [WMLStdLib] for more information about the following functions in the *Lang* library:

Opcode	**Library function**
IS_INT	Lang.isInt()
IS_FLOAT	Lang.isFloat()
IS_NUMBER	Lang.isNumber()
PARSE_INT	Lang.parseInt()
PARSE_FLOAT	Lang.parseFloat()

10.8. Limitations

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

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 number of functions in the function pool	255
Maximum number of function parameters	255
Maximum number of local variables / function	255
Maximum length of function name	255
Maximum number of libraries	65536
Maximum number of functions / library	256
Minimum size of integer	28 bits
Size of float	32 bits