THE JSCRIPT LANGUAGE SPECIFICATION

Version 0.1

MICROSOFT INTELLECTUAL PROPERTY STATEMENT
Microsoft agrees to grant, and does grant to ECMA, a perpetual, nonexclusive, royalty-free, world-wide right and license under any Microsoft copyrights in this contribution to copy, publish and distribute the contribution, as well as a right and license of the same scope to any derivative works prepared by ECMA and based on, or incorporating all or part of the contribution. Microsoft further agrees that, upon adoption of this contribution as an Internet Standard, any party will be able to obtain a license under applicable Microsoft rights to implement and use the technology described in this contribution for the purpose of supporting the Internet Standard, under reasonable and nondiscriminatory terms to be negotiated with Microsoft. Microsoft expressly reserves all other rights it may have in the material and subject matter of this contribution. Microsoft expressly disclaims any and all warranties regarding this contribution including any warranty that (a) this contribution does not violate the rights of others, (b) the owners, if any, of other rights in this contribution have been informed of the rights and permissions granted to ECMA herein or (c) any required authorizations from such owners have been obtained.

COPYRIGHT NOTICE
Copyright ©1996 Microsoft Corporation.
All rights reserved.

DOCUMENTATION NOTATION
Throughout this document, comments in italic and colored in red indicate differences with Java language conventions.

FEEDBACK
Please send feedback on this document to Robert Welland (robwell@microsoft.com).
# CONTENTS

The JScript Language Specification..........................................................1
Microsoft Intellectual Property Statement.............................................1
Copyright Notice.......................................................................................1
Feedback....................................................................................................1
Contents.....................................................................................................2

## 1 Notational Conventions........................................................................5

## 2 Unicode...............................................................................................7

## 3 Lexical Conventions.............................................................................7
### 3.1 White Space.....................................................................................7
### 3.2 Comments.......................................................................................7
### 3.3 Tokens.............................................................................................8
#### 3.3.1 Keywords....................................................................................8
#### 3.3.2 Identifiers.................................................................................9
#### 3.3.3 Operators..................................................................................9
### 3.4 Literals............................................................................................10
#### 3.4.1 The Null Literal.......................................................................10
#### 3.4.2 Boolean Literals.......................................................................10
#### 3.4.3 Numeric Literals......................................................................10
#### 3.4.4 String Literals.........................................................................10

## 4 Types..................................................................................................10
### 4.1 The Undefined Type.......................................................................10
### 4.2 The Null Type...............................................................................10
### 4.3 The Boolean Type..........................................................................10
### 4.4 The Number Type.........................................................................10
### 4.5 The String Type............................................................................10
### 4.6 The Object Type............................................................................10
### 4.7 The Array Subtype of Object.........................................................10

## 5 Properties..........................................................................................1
### 5.1 Object Properties...........................................................................10
### 5.2 Property Inheritance.....................................................................10

## 6 Built-in Objects..................................................................................10
### 6.1 Constructors................................................................................10
#### 6.1.1 Boolean...................................................................................10
#### 6.1.2 Number.....................................................................................10
#### 6.1.3 String.......................................................................................10
#### 6.1.4 Object........................................................................................10
#### 6.1.5 Array........................................................................................10
### 6.2 Behavior Parents...........................................................................10
#### 6.2.1 Number.....................................................................................10
#### 6.2.1.1 NaN......................................................................................10
#### 6.2.2 String.......................................................................................10
#### 6.2.2.1 Length...................................................................................10
#### 6.2.2.2 IndexOf................................................................................10
#### 6.2.2.3 LastIndexOf...........................................................................10
#### 6.2.2.4 Substring..............................................................................10
#### 6.2.2.5 charAt...................................................................................10
#### 6.2.2.6 toLowerCase..........................................................................10
#### 6.2.2.7 toUpperCase..........................................................................10
#### 6.2.2.8 split.......................................................................................10
### 6.2.3 Array........................................................................................10
#### 6.2.3.1 join.......................................................................................10
#### 6.2.3.2 reverse................................................................................10
6.2.3.3 sort.................................................................................................................. 10
6.3 Math.......................................................................................................................... 10
   6.3.1 The abs Method................................................................................................. 10
   6.3.2 The acos Method............................................................................................. 10
   6.3.3 The asin Method............................................................................................... 10
   6.3.4 The atan Method.............................................................................................. 10
   6.3.5 The ceil Method.............................................................................................. 10
   6.3.6 The cos Method............................................................................................... 10
   6.3.7 The E Property................................................................................................. 10
   6.3.8 The exp Method............................................................................................... 10
   6.3.9 The floor Method............................................................................................. 10
   6.3.10 The LN10 Property.......................................................................................... 10
   6.3.11 The LN2 Property........................................................................................... 10
   6.3.12 The LOG10E Property.................................................................................... 10
   6.3.13 The PI Property.............................................................................................. 10
   6.3.14 The SQRT1_2 Property.................................................................................. 10
   6.3.15 The SQRT2 Property...................................................................................... 10
7 Variable Scoping........................................................................................................ 10
   7.1.1 The Scope of Variables..................................................................................... 10
   7.1.2 Implicit Global Variables.................................................................................. 10
8 Coercion...................................................................................................................... 10
   8.1 Reference Comparisons....................................................................................... 10
   8.2 Operators: &&, ||, !.............................................................................................. 10
   8.3 Unary Operators: ++, --, -, ~............................................................................. 10
   8.4 String Conversions.............................................................................................. 10
   8.5 Bitwise Operators and Numbers......................................................................... 10
9 Expressions.................................................................................................................. 10
   9.1 Primary Expressions............................................................................................ 10
      9.1.1 The this Keyword......................................................................................... 10
      9.1.2 The arguments Keyword............................................................................. 10
      9.1.3 Variable reference....................................................................................... 10
   9.2 Postfix Operators.................................................................................................. 10
      9.2.1 Property Accessors....................................................................................... 10
      9.2.2 Function Calls.............................................................................................. 10
      9.2.3 Object Method Calls.................................................................................... 10
      9.2.4 Postfix Increment and Decrement Operators............................................. 10
   9.3 Unary Operators.................................................................................................... 10
      9.3.1 The new Expression...................................................................................... 10
      9.3.2 The delete Expression.................................................................................. 10
      9.3.3 The void Expression..................................................................................... 10
      9.3.4 The typeof Expression............................................................................... 10
      9.3.5 Prefix Increment and Decrement Operators.............................................. 10
      9.3.6 Unary +- Operators..................................................................................... 10
      9.3.7 The Bitwise NOT Operator (~).................................................................... 10
      9.3.8 Logical NOT Operator (!)........................................................................... 10
   9.4 Multiplicative Operators........................................................................................ 10
      9.4.1 The Division Operator (/)............................................................................. 10
      9.4.2 The Multiplication Operator (*)................................................................ 10
      9.4.3 The Modulus Operator (%)....................................................................... 10
   9.5 Additive Operators................................................................................................ 10
      9.5.1 The Addition Operator (+).......................................................................... 10
      9.5.2 The Subtraction Operator (-)..................................................................... 10
   9.6 Bitwise Shift Operators........................................................................................ 10
      9.6.1 The Left Shift Operator (<<)...................................................................... 10
      9.6.2 The Signed Right Shift Operator (>>)......................................................... 10
      9.6.3 The Unsigned Right Shift Operator (>>>)...----------------------------------- 10
## NOTATIONAL CONVENTIONS

This specification organizes sections into five categories: syntax, constraints, description, semantics, and discussion.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Describes the syntax of some language entity</td>
</tr>
<tr>
<td>Constraints</td>
<td>Adds textual constraints to the syntax description. These constraints normally cannot be described using formal syntax.</td>
</tr>
<tr>
<td>Description</td>
<td>A textual description of the entity.</td>
</tr>
<tr>
<td>Semantics</td>
<td>The semantics of the language entity. This and the syntax category are normally the most rigorous section of the section.</td>
</tr>
<tr>
<td>Discussion</td>
<td>A textual discussion of the language entity being described. This section is to give greater insight through examples and non-formal text.</td>
</tr>
</tbody>
</table>

The syntax notation is similar to that used in the ANSI C Language Specification\(^1\). Syntactic categories are represented by *italic* type. Literals are specified in *fixed-font* bold type. Syntactic productions are specified by the rule name followed by a colon followed by one or more definitions. Definitions are placed on separate lines. Alternately, definitions can be space separated if the definitions are prefaced by the words "one of". Appending the subscript "opt" to the symbol specifies that the construct is optional. The basic syntax of this descriptive form is as follows:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production</td>
<td>ProductionSymbol : LineSeparatedDefinitions OR ProductionSymbol : one of SpaceSeparatedDefinitions</td>
<td>Syntax rules (productions) are specified by a name (RuleName) and a rule body (RuleBody) separated by a &quot;:&quot;. If the words &quot;one of&quot; are placed after the colon then the production definitions are space separated.</td>
</tr>
<tr>
<td>LineSeparatedDefinitions</td>
<td>One or more of: Definition (&lt;newline&gt;)</td>
<td>Alternate definitions are placed on separate lines.</td>
</tr>
<tr>
<td>SpaceSeparatedDefinitions</td>
<td>One or more of: Definition (&lt;space&gt;)</td>
<td>Alternate definitions are placed on the same line and separated by white space. This convention is normally used to specify lexical categories.</td>
</tr>
<tr>
<td>Definition</td>
<td>One or more of: SyntacticElement</td>
<td>A rule statement consists of one or more syntactic elements.</td>
</tr>
<tr>
<td>SyntacticElement</td>
<td>Literal ProductionSymbol (\text{Literal}<em>{opt}) ProductionSymbol(\text{Literal}</em>{opt})</td>
<td>A syntactic element can either be a literal or a production symbol that specifies a production rule. If the syntactic element is optional then literals or production symbols are subscripted with &quot;opt&quot;.</td>
</tr>
<tr>
<td>Literal</td>
<td>Text in fixed-spaced bold font. Examples: one of while ( ) { }</td>
<td>Literals specify text as it must appear in the source text. Literals are specified in a <em>fixed-spaced bold font</em>.</td>
</tr>
<tr>
<td>ProductionSymbol</td>
<td>An <em>italic</em> identifier. Example: one of WhileStatement ArithmeticExpression</td>
<td>A production symbol is a symbol that specifies a production. Production symbols are in <em>italic font</em>.</td>
</tr>
</tbody>
</table>
UNICODE

JScript source text is represented as Unicode version 2.0. To support ASCII based systems, it is possible to represent any Unicode value as a sequence of ASCII values within a source document. Within an ASCII based source document, non-escaped values are translated to their Unicode equivalents. Escaped values are represented in the source text as a “\u” followed by four hex digits:

*UnicodeEscapeSequence:*
\u HexDigit HexDigit HexDigit 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

A 16-bit Unicode value is derived from the hex digits. The leftmost digits constitute the high order bits of the Unicode value. A Unicode escape sequence can occur anywhere in the source text and will be translated into its Unicode equivalent.

Non-ASCII Unicode values are limited to string constants and comment text. All other occurrences are in error. **Unlike Java, Unicode characters are not allowed in identifiers.**

3 LEXICAL CONVENTIONS

The source text to a JScript program is first converted into a sequence of tokens and white space. A token is a sequence of characters from the source text that constitutes a lexical element. The source text is scanned from left to right, repeatedly taking the longest possible sequence of characters as the next token.

White space is a sequence of whitespace characters and comments. White space may occur between any two tokens, but it is not required.

3.1 WHITE SPACE

White space characters are used to separate tokens from one another. White space can also be used to format the source text. Because white space characters are used to delineate lexical units they cannot be used within a lexical unit. The following characters are considered white space:

<table>
<thead>
<tr>
<th>Unicode Value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\u000A</td>
<td>Tab</td>
</tr>
<tr>
<td>\u000D</td>
<td>Newline</td>
</tr>
<tr>
<td>\u0010</td>
<td>Line feed</td>
</tr>
<tr>
<td>\u0019</td>
<td>End of medium (^Z)</td>
</tr>
<tr>
<td>\u0020</td>
<td>Space</td>
</tr>
</tbody>
</table>

3.2 COMMENTS

Like white space, comments separate tokens.

**Syntax**

```
Comment:
    MultiLineComment
    SingleLineComment

MultiLineComment:
    /* MultiLineCommentChars */
```
**MultiLineCommentChars:**
<any Unicode character except asterisk * > MultiLineCommentChars
* PostAsteriskCommentChars

**PostAsteriskCommentChars**
<any Unicode character except forward-slash / > MultiLineCommentChars

**SingleLineComment:**
// SingleLineCommentChars <newline>

**SingleLineCommentChars**
<any Unicode character except newline> SingleLineCommentChars

**Description**
Comments can be either single or multi-line. Multi-line comments cannot nest.

*The Java documentation comment (/** text */ ) convention is not explicitly endorsed.*

### 3.3 Tokens

**Syntax**

Token:
Keyword
Identifier
Punctuator
Operator
Literal

#### 3.3.1 Keywords

**Syntax**

Keyword: one of

<table>
<thead>
<tr>
<th>abstract</th>
<th>else</th>
<th>int</th>
<th>switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>arguments</td>
<td>extends</td>
<td>interface</td>
<td>synchronized</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
<td>long</td>
<td>this</td>
</tr>
<tr>
<td>break</td>
<td>final</td>
<td>native</td>
<td>throw</td>
</tr>
<tr>
<td>byte</td>
<td>finally</td>
<td>new</td>
<td>throws</td>
</tr>
<tr>
<td>case</td>
<td>float</td>
<td>null</td>
<td>transient</td>
</tr>
<tr>
<td>catch</td>
<td>for</td>
<td>package</td>
<td>true</td>
</tr>
<tr>
<td>char</td>
<td>function</td>
<td>private</td>
<td>try</td>
</tr>
<tr>
<td>class</td>
<td>goto</td>
<td>protected</td>
<td>typeof</td>
</tr>
<tr>
<td>const</td>
<td>if</td>
<td>public</td>
<td>var</td>
</tr>
<tr>
<td>continue</td>
<td>implements</td>
<td>return</td>
<td>void</td>
</tr>
<tr>
<td>default</td>
<td>import</td>
<td>short</td>
<td>while</td>
</tr>
<tr>
<td>do</td>
<td>in</td>
<td>static</td>
<td>with</td>
</tr>
<tr>
<td>double</td>
<td>instanceof</td>
<td>super</td>
<td></td>
</tr>
</tbody>
</table>

**Description**
Keyword tokens are reserved and cannot be used as identifiers. Keywords are case-sensitive; for example, **double** is a keyword, but **Double** is an identifier.

**Discussion**
True, false, and null are, semantically speaking, actually literal constants.

Many of these keywords are not used by the JScript language and a reserved for future use. The keywords currently in use by JScript are:
 ISSUE: What is the rationale for reserving Java specific keywords?

3.3.2 Identifiers

Syntax

Identifier: IdentifierPreface IdentifierBody

IdentifierPreface: one of

Letter
_ $

IdentifierBody: IdentifierBody BodyCharacter

BodyCharacter: one of

Letter
0 1 2 3 4 5 6 7 8 9
_ $

Letter: one of

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

Description
An identifier is a sequence of letters, digits, and extended alphabetic characters that must begin with a character from a distinguished set of characters called the identifier preface. JScript is a case sensitive language; that is, identifiers that differ only in the case of one or more of their constituent characters are considered to be unique. So, for example, the identifier xxx is different from the identifier Xxx. There is no defined limit on the length of identifiers.

Semantics
An identifier denotes a variable, a function argument, a function, an object property, or an object method.

3.3.3 Operators

Syntax

Operator: one of

=   |   ==  <>  <<  ~
+=  <<=  &   >>  !
-=  >>=  ==  >>>  ++
*=>  >>>=  !=  +   --
/=  ::=  <   -   []
%=  ||  <=  *   .
&&  >=  /   ()
^=  |   >=  %   ?:
Semantics
An operator specifies an operation to be performed that produces a value. Operators are applied to one or more operands.

3.4 LITERALS
Syntax
Literal:
   NullLiteral
   BooleanLiteral
   NumericLiteral
   StringLiteral

Constraints
Literal must specify values that are in the range of values for their corresponding type.

3.4.1 The Null Literal
Syntax
NullLiteral:
   null

Semantics
The null type consists of a single unique value called “null”. The null literal specifies this immutable value.

3.4.2 Boolean Literals
Syntax
BooleanLiteral: one of
   true false

Semantics
The boolean type consists of two values: true and false. These elements can be directly specified using the true and false keywords:

3.4.3 Numeric Literals
Syntax
NumericLiteral:
   DecimalLiteral
   HexLiteral
   OctalLiteral

DecimalLiteral:
   IntegralLiteral
   RealLiteral Exponent

IntegralLiteral:
   NonzeroDecimalDigit DecimalDigits

NonzeroDecimalDigit: one of
   1 2 3 4 5 6 7 8 9

DecimalDigits:
   DecimalDigit
   DecimalDigits DecimalDigit
Description
Number literals can be specified in hexadecimal, octal, and decimal. The hexadecimal and octal representations can only be used to represent integers. The decimal representation can be used to specify either integers or real numbers (possibly in scientific notation).

Semantics
Numeric literals cannot exceed the precision of the number type. The number type is capable of representing at least 64 bit IEEE floating point numbers.

3.4.4 String Literals
Syntax
StringLiteral:
  " NotDoubleQuoteStringCharacters "
  ' NotSingleQuoteStringCharacters '

NotDoubleQuoteStringCharacters:
  Any Unicode character except double-quote " , backslash \ and the newline character EscapeSequence

NotSingleQuoteStringCharacters:
  Any Unicode character except single-quote ' , backslash \ and the newline character EscapeSequence
EscapeSequence:
  CharacterEscapeSequence
  HexEscapeSequence
  UnicodeEscapeSequence

CharacterEscapeSequence: one of
  \
  \"  \\b  \f  \n  \r  \t

HexEscapeSequence:
  \x HexDigits

Description
Strings can contain any Unicode characters. The maximum string constant supported must be at least 32,000 characters long.

4 TYPES
There are seven types of entities in JScript: Undefined, Null, Boolean, Number, String, Object, and Array.

4.1 THE UNDEFINED TYPE
The undefined type consists of a single unique value, called undefined.

4.2 THE NULL TYPE
The null type consists of a single unique value called null.

4.3 THE BOOLEAN TYPE
The Boolean type consists of two unique values, called true and false.

4.4 THE NUMBER TYPE
The Number type consists of integer and real values. The Number type has at least the precision and range of 64 bit IEEE floating point numbers.

4.5 THE STRING TYPE
Elements of the String type are arrays of Unicode characters. Strings are immutable: once created, the length and contents of a string cannot be changed.

4.6 THE OBJECT TYPE
Objects are unordered collections of name-value pairs called “slots”. In each pair, the name is a string, and the value can be of any type. Objects are mutable: values may be changed, and slots may be added and removed at any time.

4.7 THE ARRAY SUBTYPE OF OBJECT
An array is an object with a property called length whose value is always one greater than the largest integral slot name in the object. Assignment to the length property is a run-time error.
Objects and non-reference values may have “properties”. A property is a value associated with a name string.

5.1 Object Properties
Every slot of an object has a corresponding property with the same name and value as the slot.

5.2 Property Inheritance
All entities contain two properties <BehaviorParent> and <SlotParent> whose names are undefined. The slots in these objects are accessible as properties of the entity. A property named \( X \) is retrieved as follows:

1. If the entity has a property called \( X \), its value is retrieved.
2. Otherwise, if the <SlotParent> of the entity has a slot called \( X \), its value is retrieved.
3. Otherwise, if the <BehaviorParent> of the entity has a slot called \( X \), its value is retrieved.
4. Otherwise, the value undefined is retrieved.

Refer to the description of the new operator for a description of how <BehaviorParent> and <SlotParent> are initialized.

6 Built-in Objects
There are a number of predefined objects in JScript.

6.1 Constructors
Some types have an associated constructor function that can be used with the new operator. The Object and Array constructors return objects and arrays. The Number and String constructors construct objects that
wrap entities of those types. A wrapper object has a valueOf method that returns the original entity, and a toString method that returns the string representation of the original entity. The constructor functions also may have properties and methods useful for all members of the type. Entities may be coerced to objects by converting them to their types’ corresponding constructors.

References to the constructor functions are contained in global variables named for the types.

6.1.1 Boolean
The Boolean constructor has two modes.
- With no arguments, it returns an object whose valueOf method returns false.
- With one argument, which must be a boolean, it returns an object whose valueOf method returns the argument.

6.1.2 Number
The Number constructor has two modes.
- With no arguments, it returns an object whose valueOf method returns 0.
- With one argument, which must be a number, it returns an object whose valueOf method returns the argument.

6.1.3 String
The String constructor takes one argument, a string.

6.1.4 Object
The Object constructor takes no arguments and returns an empty object.

6.1.5 Array
The Array constructor has three modes:
- With no arguments, it creates an array with a length property of 0.
- With one argument \( X \), if \( X \) is a Number, it creates an array with a length property of \( X \). Otherwise, it creates an array with a length property of 1 whose “0” slot contains \( X \).
- With more than one argument, it creates an array with the arguments as the values of its numbered slots, numbered from zero left-to-right. The length property of the array is the number of arguments.

6.2 Behavior Parents
Each type has an associated <BehaviorParent> object, which contains the standard built-in behaviors for the type. Each type’s <BehaviorParent> includes a valueOf method that returns the object itself, and a toString method that returns the string representation of the object. Some have more properties, as follows.

6.2.1 Number
Aside from valueOf and toString, the properties of Number’s <BehaviorParent> are as follows:

6.2.1.1 NaN
A property with a constant value that is equivalent to the IEEE “Not-a-Number” value.

6.2.2 String
Aside from valueOf and toString, the properties of String’s <BehaviorParent> are as follows.

6.2.2.1 Length
An integer giving the total number of characters in the string. This property cannot be altered by assignment.
6.2.2.2 IndexOf
A function object.

With one argument, searchString, which is coerced to a string, returns the index in this of the first occurrence of searchString, or –1 if not found.

With two arguments, searchString and startIndex, begins the search at startIndex.

6.2.2.3 lastIndexOf
A function object.

With one argument, searchString, which is coerced to a string, returns the index in this of the last occurrence of searchString, or –1 if not found.

With two arguments, searchString and startIndex, begins the (backwards) search at startIndex.

6.2.2.4 Substring
A function object taking two arguments, first and limit, which are coerced to integers. If first is less than limit, returns a string containing the characters of this starting with index first and extending to index limit – 1. If first is greater than limit, returns a string containing the characters of this starting with index limit and extending to index first – 1. If first is equal to limit, returns an empty string.

6.2.2.5 charAt
A function object taking one argument, index, which is coerced to an integer. If index is less than zero or greater than this.length – 1, an empty string is returned. Otherwise, a string containing the character at index index in this is returned.

6.2.2.6 toLowerCase
A function object taking no arguments. Returns a string containing the contents of this converted to lower case.

6.2.2.7 toUpperCase
A function object taking no arguments. Returns a string containing the contents of this converted to upper case.

6.2.2.8 split
A function object taking one argument, separator, which is coerced to a string. Returns an array of strings generated by separating this into substrings based on separator.

6.2.3 Array
Aside from valueOf and toString, the properties of Array’s <BehaviorParent> are as follows:

6.2.3.1 join
A function object taking no arguments, or one argument, separator, which is coerced to a string. Returns a string consisting of the elements of this, coerced to strings, concatenated with separator in between each pair of elements. The no-argument form is the same as passing “,” as separator.

6.2.3.2 reverse
A function object taking no arguments. Reverses the order of the elements of this.
6.2.3.3 sort
A function object taking no arguments, or one argument, compareFunc, a function of two arguments. Sorts the elements of this according to the comparisons done by compareFunc. The no-argument form uses dictionary order of the string conversions of the elements.

6.3 MATH
The Math object has properties that are functions and constants useful for mathematical computation. The value of the global variable “Math” is a reference to the Math object.

6.3.1 The abs Method
Description
Determines the absolute value of its numeric argument.
Syntax
Math.abs( number )
The number argument is a numeric expression for which the absolute value is sought.
Remarks
The return value is the absolute value of the number argument.

6.3.2 The acos Method
Description
Computes the arccosine of its numeric argument.
Syntax
Math.acos( number )
The number argument is a numeric expression for which the arccosine is sought.
Remarks
The return value is the arccosine of the number argument.

6.3.3 The asin Method
Description
Computes the arcsine of its numeric argument.
Syntax
Math.asin( number )
The number argument is a numeric expression for which the arcsine is sought.
Remarks
The return value is the arcsine of its numeric argument.

6.3.4 The atan Method
Description
Computes the arctangent of its numeric argument.
Syntax
Math.atan( number )
The number argument is a numeric expression for which the arctangent is sought.
Remarks
The return value is the arctangent of its numeric argument.

6.3.5 The ceil Method

Description
Determines the smallest integer greater than or equal to its numeric argument.

Syntax
Math.ceil( number )
The number argument is a numeric expression.

Remarks
The return value is an integer value equal to the smallest integer greater than or equal to its numeric argument.

6.3.6 The cos Method

Description
Computes the cosine of its numeric argument.

Syntax
Math.cos( number )
The number argument is a numeric expression for which the cosine is sought.

Remarks
The return value is the cosine of its numeric argument.

6.3.7 The E Property

Description
Euler's constant, the base of natural algorithms. The E property is approximately equal to 2.718.

Syntax
var numVar
numVar = Math.E

6.3.8 The exp Method

Description
Computes Error! Bookmark not defined. to the power of the supplied numeric argument.

Syntax
Math.exp( number )
The number argument is a numeric expression representing the power of e.

Remarks
The return value is enumber The constant e is Euler's constantm, approximately equal to 2.178 and number is the supplied argument.

6.3.9 The floor Method

Description
Computes the greatest integer less than or equal to its numeric argument.
Syntax
Math.floor( number )
The number argument is a numeric expression.

6.3.10 The LN10 Property

Remarks
The return value is an integer value equal to the greatest integer less than or equal to its numeric argument.

Description
The natural logarithm of 10.

Syntax
var numVar
numVar = Math.LN10

Remarks
The LN10 property is approximately equal to 2.302.

6.3.11 The LN2 Property

Description
The natural logarithm of 2.

Syntax
var numVar
numVar = Math.LN2

Syntax
The LN2 property is approximately equal to 0.693.

6.3.12 The LOG10E Property

Description
The base 10 logarithm of e, Euler's constant.

Syntax
var varName
varName = objName.LOG10E

Remarks
The LOG10E property, a constant, is approximately equal to 0.434.

6.3.13 The PI Property

Description
The ratio of the circumference of a circle to its diameter.

Syntax
var numVar
numVar = Math.PI

Syntax
The pi property, a constant, is approximately equal to 3.14159.
6.3.14 The SQRT1_2 Property

Description
The square root of 0.5, or one divided by the square root of 2.

Syntax
var numVar
numVar = Math.SQRT1_2

Remarks
The SQRT1_2 property, a constant, is approximately equal to 0.707.

6.3.15 The SQRT2 Property

Description
The square root of 2.

Syntax
var numVar
numVar = Math.SQRT2

Syntax
The SQRT2 property, a constant, is approximately equal to 1.414.

7 VARIABLE SCOPING

7.1.1 The Scope of Variables
The visibility of an variable is called its scope. In JScript, scopes can be nested such that an inner scope is completely contained by an outer scope. If an identifier of the same name is defined by both the inner and outer scopes then the identifier resolves to the entity defined by the inner scope. This is generally termed lexical scoping. JScript has five lexically nested scoping levels: global, host-defined, function arguments, function locals, and with.

<table>
<thead>
<tr>
<th>Context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Global variables are visible from any point in the source text so long as the name used to specify the variable is not masked by a nested scope.</td>
</tr>
<tr>
<td>Host-defined</td>
<td>The host system may define variables that are available anywhere in the source text and hide global variables defined in JScript source. (The definition mechanism is beyond the scope of the JScript specification.)</td>
</tr>
<tr>
<td>Function</td>
<td>A caller passes argument values to functions. These values are bound to argument variables that can be accessed within the body of a function or method.</td>
</tr>
<tr>
<td>Arguments</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Local variables can be defined within a function body and are visible from within that function body. (Note that nested statement blocks within a function do not constitute separate scopes.)</td>
</tr>
<tr>
<td>Locals</td>
<td></td>
</tr>
<tr>
<td>With Block</td>
<td>A with block can be used to associate a scope with a particular object. Identifiers specified within the with block first try to resolve to the properties on the object specified by the with expression.</td>
</tr>
</tbody>
</table>
7.1.2 Implicit Global Variables
If a variable assignment statement is executed whose variable identifier is not found in any scope, a global variable is created.

8 COERCION
Coercion are performed to convert a value of one type into a value of another type. Coercion is used to implement the core unary and binary operators. The following tables detail the coercions performed for the different operators.

The following table details the conversions performed on different types of arguments for the + and += operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>non-numeric</th>
<th>numeric</th>
<th>string</th>
<th>undefined</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Object</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>+</td>
<td>non-numeric</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>+</td>
<td>numeric</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>+</td>
<td>string</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>+</td>
<td>undefined</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>+</td>
<td>null</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
<td>String</td>
</tr>
</tbody>
</table>

The following table details the coercions for the equality operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>non-numeric</th>
<th>numeric</th>
<th>string</th>
<th>undefined</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Object</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>!=</td>
<td>non-numeric</td>
<td>false</td>
<td>String</td>
<td>Error</td>
<td>Error</td>
<td>false</td>
</tr>
<tr>
<td>non-numeric</td>
<td>numeric</td>
<td>false</td>
<td>String</td>
<td>Error</td>
<td>Error</td>
<td>false</td>
</tr>
<tr>
<td>numeric</td>
<td>string</td>
<td>false</td>
<td>String</td>
<td>Numeric</td>
<td>Numeric</td>
<td>false</td>
</tr>
<tr>
<td>numeric</td>
<td>undefined</td>
<td>false</td>
<td>String</td>
<td>Error</td>
<td>Error</td>
<td>false</td>
</tr>
<tr>
<td>numeric</td>
<td>null</td>
<td>false</td>
<td>String</td>
<td>Error</td>
<td>Error</td>
<td>false</td>
</tr>
</tbody>
</table>

The following table details the coercions for the relational operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>non-numeric</th>
<th>numeric</th>
<th>string</th>
<th>undefined</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Object</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>&gt;</td>
<td>non-numeric</td>
<td>false</td>
<td>String</td>
<td>Error</td>
<td>Error</td>
<td>true</td>
</tr>
<tr>
<td>&gt;=</td>
<td>numeric</td>
<td>false</td>
<td>String</td>
<td>Error</td>
<td>Error</td>
<td>true</td>
</tr>
<tr>
<td>&gt;=</td>
<td>string</td>
<td>false</td>
<td>String</td>
<td>Error</td>
<td>Error</td>
<td>true</td>
</tr>
<tr>
<td>&gt;=</td>
<td>undefined</td>
<td>false</td>
<td>String</td>
<td>Error</td>
<td>Error</td>
<td>true</td>
</tr>
<tr>
<td>&gt;=</td>
<td>null</td>
<td>false</td>
<td>String</td>
<td>Error</td>
<td>Error</td>
<td>true</td>
</tr>
</tbody>
</table>
8.1 Reference Comparisons

Objects compare by reference, so

(Math == Math) === true
(Math == Date) === false

8.2 Operators: &&, ||, !

The following are the rules for converting to a boolean:
- All objects are true.
- Strings are false if and only if they are empty.
- Null and undefined are false.
- Numbers are false if and only if they are 0.

8.3 Unary Operators: ++, --, -, ~

These work as follows:
- Applied to undefined or null signals a runtime error.
- Objects are converted to strings (toString function), then the string is converted to a numbers.
- Strings are converted to numbers (error if not possible), then the operator applied to the number.
- Booleans are treated as numbers (false is 0 and true is 1).
8.4 **STRING CONVERSIONS**
Conversion from number from string uses the same rules as parseFloat, except that in cases their behavior differs. Error cases involving conversion (rather than parsing) result in runtime errors.

8.5 **BITWISE OPERATORS AND NUMBERS**
Numbers are truncated as necessary to produce the 32-bit integers used by the bitwise operators.

9 **EXPRESSIONS**

9.1 **PRIMARY EXPRESSIONS**

**Syntax**

```
PrimaryExpression:
    this
    arguments
    Identifier
    Literal
    ( Expression )
```

9.1.1 **The this Keyword**
The `this` keyword has meaning in two contexts:
1. When a function is called as a method of an object; in this case the `this` pointer refers to the calling object.
2. When a function is being called as a constructor; that is, in the context of a new expression. In this case, the `this` pointer refers to the object being constructed.

In all other cases the `this` pointer is undefined.

9.1.2 **The arguments Keyword**
The `arguments` keyword refers to an array that represents the arguments of current activation frame. The `arguments` keyword is generally used when a function takes a variable number of arguments. The arguments array holds the argument values in the same order as they are evaluated, that is, the 0th array element contains the 0th (leftmost) argument, the 1st array element contains the 1st argument and so on.

This interpretation of the "arguments" within a function body differs from existing practice but has two important advantages of the current mechanism:
1. It can be much more efficiently implemented, especially in the case of recursive functions.
2. It eliminates some complex and confusing semantic issues that arise as a result of the arguments to an activation frame being accessible from a function object.

9.1.3 **Variable reference**
An identifier resolves itself using the scoping rules stated in section 7.

9.2 **POSTFIX OPERATORS**

**Syntax**

```
MemberExpression:
    PostfixExpression [ Expression ]
    PostfixExpression . Identifier
```

```
FunctionCallExpression:
```
9.2.1 Property Accessors
Properties are accessed by name, using either the dot notation $PostfixExpression . Identifier$ or the bracket notation $PostfixExpression [ Expression ]$.

The dot notation is transformed using the following syntactic conversion:

$$PostfixExpression . Identifier$$

is exactly equivalent to:

$$PostfixExpression [ <identifier-string> ]$$

Where <identifier-string> is a string literal containing the same characters as the identifier.

The bracket notation is transformed using the following syntactic conversion:

$$PostfixExpression [ Expression ]$$

is exactly equivalent to:

$$PostfixExpression [ "" + ( Expression ) ]$$

9.2.2 Function Calls
Function calls are evaluated as follows:
1. $PostfixExpression$ must evaluate to a function object.
2. $ArgumentList$ is evaluated from left to right.
3. An activation frame is created to hold the arguments.
4. A special this reference in the activation frame is set to undefined. This reference is used to resolve references resulting from use of the this keyword.
5. If a reference in the function body is made to the arguments keyword, then a special arguments reference must be initialized to point to the array of arguments as specified by the arguments keyword. Note that this step is only required if the function body refers to the arguments keyword.
6. The function body is evaluated using the activation frame as the execution context.
7. Any return value is captured for the caller and the activation frame is discarded.
8. Control is returned to the caller.

9.2.3 Object Method Calls
Object method calls are evaluated as follows:
1. The PostfixExpression preceding the ( ArgumentList ) must be a MemberExpression. The MemberExpression is evaluated and must yield a property that evaluates to a function object. Otherwise, a runtime error is signalled.
2. The argument list is evaluated from left to right.
3. An activation frame is created to hold the arguments.
4. A special `this` reference in the activation frame is bound to the target entity mentioned in step 1. This reference is used to resolve references resulting from the use of the `this` keyword.

5. If a reference in the function body is made to the `arguments` keyword, then a special arguments reference must be initialized to point to the array of arguments as specified by the `arguments` keyword. Note that this step is only required if the function body refers to the `arguments` keyword.

6. Any return value is captured for the caller and the activation frame is discarded.

7. Control is returned to the caller.

9.2.4 Postfix Increment and Decrement Operators

These operators are evaluated as follows:
1. PostfixExpression is evaluated and must yield an lvalue.
2. The lvalue from step 1 is dereferenced and an attempt is made to coerce it to a number. If this fails then a runtime error is signaled.
3. The value from step 1 is increased or decreased by one as necessary.
4. The lvalue is assigned the value from step 3.
5. The value from step 1 is the expression result.

9.3 Unary Operators

Syntax

```
UnaryExpression:
   PostfixExpression
   new FunctionCallExpression
   delete MemberExpression
   void UnaryExpression
   typeof UnaryExpression
   ++ UnaryExpression
   -- UnaryExpression
   ~ UnaryExpression
   ! UnaryExpression
```

9.3.1 The `new` Expression

The `new` operator creates new entities using constructor functions. A `new` expression performs the following steps:

1. A new, empty object is created.
2. The `<BehaviorParent>` property is initialized in an implementation-dependent manner to support the type-specific operations on the entity.
3. The `<SlotParent>` property is initialized with a reference to the constructor function’s `prototype` slot.
4. The constructor function is executed with `this` bound to the new object.
5. The value of the `new` expression is a reference to the object created in step 1.

9.3.2 The `delete` Expression

The `delete` expression removes a property from an entity, if found. Delete is implemented as follows:

1. The delete `MemberExpression` is evaluated.
2. If the `MemberExpression` does not resolve to a valid entity property then `false` is returned as the expression result.
3. Otherwise, the referred to property is removed from the entity and `true` is returned as the expression result.
4. Attempting to removing built-in properties may result in a runtime error being signaled.
9.3.3 The `void` Expression
The `void` expression discards the value `UnaryExpression` and returns `undefined`.

9.3.4 The `typeof` Expression
`typeof` returns a string representation of the type of the given `UnaryExpression`. The strings returned for the various types are as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>&quot;undefined&quot;</td>
</tr>
<tr>
<td>Null</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>&quot;boolean&quot;</td>
</tr>
<tr>
<td>Number</td>
<td>&quot;number&quot;</td>
</tr>
<tr>
<td>String</td>
<td>&quot;string&quot;</td>
</tr>
<tr>
<td>Object</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Array</td>
<td>&quot;object&quot;</td>
</tr>
</tbody>
</table>

9.3.5 Prefix Increment and Decrement Operators
These operators are evaluated as follows:
1. `UnaryExpression` is evaluated and must yield an lvalue.
2. The lvalue from step 1 is dereferenced and an attempt is made to coerce it to a number. If this fails then a runtime error is signaled.
3. The value from step 1 is increased or decreased by one as necessary.
4. The lvalue is assigned the value from step 3.
5. The value from step 3 is the expression result.

9.3.6 Unary `+-` Operators
The `UnaryExpression` is evaluated and must be coercible to a number. If the coercion fails then a runtime error is signalled. Otherwise, if the operator specified is the negation operator then the number is negated and returned as the result of the expression.

9.3.7 The Bitwise NOT Operator ( ~ )
The `UnaryExpression` is evaluated and must be coercible to a number. If the coercion fails then a runtime error is signalled. Otherwise, the number is converted into a 32-bit integer and a bitwise NOT is applied to the value and the result returned as the result of the expression.

9.3.8 Logical NOT Operator ( ! )
The `UnaryExpression` is evaluated and must be coercible to a boolean. If the coercion fails then a runtime error is signalled. Otherwise, the logical not is applied to the boolean value and returned.

9.4 Multiplicative Operators
Syntax

```
MultiplicativeExpression:
  UnaryExpression
  MultiplicativeExpression * UnaryExpression
  MultiplicativeExpression / UnaryExpression
  MultiplicativeExpression % UnaryExpression
```
9.4.1 The Division Operator ( / )
The MultiplicativeExpression and the UnaryExpressions are evaluated (in that order) and must be coercible to numbers. If the coercion fails on either argument then a runtime error is signaled. If MultiplicativeExpression fails to coerce then UnaryExpression is not evaluated at all.

MultiplicativeExpression is then divided by the UnaryExpression. If UnaryExpression is zero then a runtime error is signaled. The division is performed as if both arguments where represented as floating point.

9.4.2 The Multiplication Operator ( * )
The MultiplicativeExpression and the UnaryExpressions are evaluated (in that order) and must be coercible to numbers. If the coercion fails on either argument then a runtime error is signaled. If MultiplicativeExpression fails to coerce then UnaryExpression is not evaluated at all.

The product of MultiplicativeExpression and UnaryExpression is computed. The multiplication is performed as if both arguments where represented as floating point.

9.4.3 The Modulus Operator ( % )
The MultiplicativeExpression and the UnaryExpression are evaluated (in that order) and must be coercible to numbers. If the coercion fails on either argument then a runtime error is signaled. If MultiplicativeExpression fails to coerce then UnaryExpression is not evaluated at all.

Returns the value MultiplicativeExpression - i * UnaryExpression, for some integer i such that, if UnaryExpression != 0 then the result has the same sign as MultiplicativeExpression and magnitude less than the magnitude of UnaryExpression. If UnaryExpression is zero then the result is undefined.

9.5 ADDITIVE OPERATORS
Syntax
AdditiveExpression:
  MultiplicativeExpression
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression

9.5.1 The Addition Operator ( + )
The addition operator either performs string concatenation or numeric addition. If either argument can only be coerced to a string then string concatenation is performed. This occurs when one of the arguments is a string, function, or object entity.

9.5.2 The Subtraction Operator ( - )
The subtraction operator performs numeric subtraction. Both arguments must be coercible to numbers. Otherwise a runtime error is signalled.

9.6 BITWISE SHIFT OPERATORS
Syntax
ShiftExpression:
  ShiftExpression << AdditiveExpression
  ShiftExpression >> AdditiveExpression
  ShiftExpression >>> AdditiveExpression
9.6.1 The Left Shift Operator (\texttt{\$\$\$})
Performs a left shift operation on the left argument by the amount specified by the right argument. The left argument is coerced to a 32 bit value and only the bottom 5 bits of the right argument are used to specify the shift amount.

9.6.2 The Signed Right Shift Operator (\texttt{\$\$})
Performs a sign-filling right shift operation on the left argument by the amount specified by the right argument. The left argument is coerced to a 32 bit value and only the bottom 5 bits of the right argument are used to specify the shift amount.

9.6.3 The Unsigned Right Shift Operator (\texttt{\$\$\$})
Performs a zero-filling right shift operation on the left argument by the amount specified by the right argument. The left argument is coerced to a 32 bit value and only the bottom 5 bits of the right argument are used to specify the shift amount.

9.7 RELATIONAL OPERATORS
Syntax
\[
\text{RelationalExpression:}
\begin{align*}
\text{ShiftExpression} \\
\text{RelationalExpression} \ < \ \text{ShiftExpression} \\
\text{RelationalExpression} \ > \ \text{ShiftExpression} \\
\text{RelationalExpression} \ <= \ \text{ShiftExpression} \\
\text{RelationalExpression} \ >= \ \text{ShiftExpression}
\end{align*}
\]
Semantics
The relational operators perform either string magnitude comparisons or numeric magnitude comparisons. If both can be coerced to either form then string comparisons are used. If one or neither of the arguments can be coerced to strings or numbers then a runtime error is generated.

9.8 EQUALITY OPERATORS
Syntax
\[
\text{EqualityExpression:}
\begin{align*}
\text{RelationalExpression} \\
\text{EqualityExpression} \ == \ \text{RelationalExpression} \\
\text{EqualityExpression} \ != \ \text{RelationalExpression}
\end{align*}
\]
Semantics
Compares the two entities for equality or inequality. There are three possible comparisons that can be performed:
- Identity comparisons - this comparison is performed on objects (including null) and functions.
- String comparisons - this is performed if both arguments can be coerced to strings.
- Numeric comparisons - this is performed if both arguments can be converted to numbers.

9.9 BINARY BITWISE OPERATORS
Syntax
\[
\text{BitwiseANDExpression:}
\begin{align*}
\text{EqualityExpression} \\
\text{BitwiseANDExpression} \ & \ \text{EqualityExpression}
\end{align*}
\]
\[
\text{BitwiseXORExpression:}
\begin{align*}
\text{BitwiseANDExpression} \\
\text{BitwiseXORExpression} \ ^ \ \text{BitwiseANDExpression}
\end{align*}
\]
**9.10 Binary Logical Operators**

**Syntax**

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

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

**Semantics**
Perform logical operations on the left and right arguments. Both arguments are coerced to boolean values and the expression result is computed by applying the appropriate logical operator (`&&` or `||`).


**Syntax**

- **ConditionalExpression:**
  - LogicalORExpression
  - LogicalORExpression ? Expression : ConditionalExpression

**Semantics**
The LogicalORExpression is coerced to a boolean. If the result is true then Expression is evaluated; otherwise ConditionalExpression is evaluated.

**9.11.1 Conditional Expressions**

**9.12 Assignment Operators**

**Syntax**

- **AssignmentExpression:**
  - ConditionalExpression
  - UnaryExpression AssignmentOperator AssignmentExpression

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

**9.12.1 Simple Assignment ( = )**
The UnaryExpression is evaluated and must evaluate to an lvalue. The AssignmentExpression is then evaluated and the result is used to set the lvalue. A runtime error is generated if AssignmentExpression is undefined.

**9.12.2 Compound Assignment ( op= )**
A compound expression E1 op= E2 is identical to E1 = E1 op E2 except E1 is evaluated only once.
9.13 COMMA OPERATOR ( , )

Syntax
   Expression:
      AssignmentExpression
   Expression , AssignmentExpression

Semantics
The sequence of expressions is evaluated from left to right and the last expression is returned as the
expression result.

10 STATEMENTS

Syntax
   Statement:
      Block
      VariableStatement
      ExpressionStatement
      IfStatement
      IterationStatement
      ControlFlowStatements
      SwitchStatement
      WithStatement

10.1 BLOCK

Syntax
   Block:
      { Statements_opt }

   Statements:
      Statement
      Statements Statement

Semantics
The statements in a block are executed in sequence.

10.2 VARIABLE STATEMENT

Syntax
   VariableStatement::
      var VariableDeclarationList ;

   VariableDeclarationList:
      VariableDeclaration
      VariableDeclarationList , VariableDeclaration

   VariableDeclaration:
      Identifier Initializer_opt

   Initializer:
      = AssignmentExpression

Semantics
If the variable statement occurs inside a FunctionDeclaration, they are defined with function-local scope in
that function. Otherwise, they are defined with global scope. Variables with no Initializer are initialized to
the undefined value. Variables with Initializers are initialized to the values of their AssignmentExpressions, which are executed in order of appearance.
10.3 **EXPRESSION STATEMENT**

**Syntax**

\[ \text{ExpressionStatement:} \]
\[ \text{Expression}_{opt} ; \]

**Semantics**

The expression in an `ExpressionStatement` is executed for side effects; its value is ignored. An `ExpressionStatement` with no `Expression` performs no operations.

10.4 **THE `if` STATEMENT**

**Syntax**

\[ \text{IfStatement:} \]
\[ \text{if ( Expression ) Statement} \]
\[ \text{if ( Expression ) Statement} \text{ else Statement} \]

An `else` is associated with the lexically immediately preceding `else-less if` that is in the same block (but not in an enclosed block).

**Semantics**

The `Expression`’s value is coerced to a Boolean entity (the “condition”). In both forms, the first `Statement` is executed if and only if the condition is `true`. In the second form, the second `Statement` is executed if and only if the condition is `false`.

10.5 **ITERATION STATEMENTS**

**Syntax**

\[ \text{IterationStatement:} \]
\[ \text{while ( Expression ) Statement} \]
\[ \text{for ( Expression}_{opt} ; \text{ Expression}_{opt} ; \text{ Expression}_{opt} ) \text{ Statement} \]
\[ \text{for ( var}_{opt} \text{ Identifier in Expression ) Statement} \]

**Semantics**

10.5.1 **The `while` Statement**

`Statement` is executed repeatedly. Before each execution of `Statement`, the `Expression` is evaluated and the result is coerced to a Boolean entity (the “condition”). If the condition is `false`, execution of the `while` statement ceases.

10.5.2 **The `for` Statement**

The first `Expression` is executed, if present. Then `Statement` is executed repeatedly. Before each execution of `Statement`, the second `Expression` is evaluated and the result is coerced to a Boolean entity (the “condition”). If the condition is `false`, execution of the `for` statement ceases. After each execution of `Statement`, the third `Expression` is evaluated, if present.

If the second `Expression` is not present, the condition is always `false`.

10.5.3 **The `for..in` Statement**

`Expression` is evaluated (call its value the “subject”). `Statement` is executed repeatedly; before each execution, the name of a slot in the subject is assigned to the variable named by `Identifier`. Slots are chosen in no defined order, but each slot is chosen exactly once. When the slots are exhausted, execution of the `for` statement ceases.
10.6 CONTROL FLOW STATEMENTS

Syntax

```
ControlFlowStatement:
  continue ;
  break ;
  return Expression_opt ;
```

10.6.1 The continue Statement

The `continue` statement can only be used inside a `while`, `for`, or `for..in` loop. Executing the `continue` statement stops the current iteration of the loop and continues program flow with the beginning of the loop. This has the following effects on the different types of loops:

- `while` loops test their condition, and if true, execute the loop again.
- `for` loops execute their increment expression, and if the test expression is true, execute the loop again.
- `for..in` loops proceed to the next slot and execute the loop again.

10.6.2 The break Statement

The `break` statement can only be used inside a `while`, `for`, or `for..in` loop. Executing the `break` statement causes execution of the enclosing loop to cease.

10.6.3 The return Statement

The `return` statement can only be used inside a `FunctionDeclaration`. It causes a function to return a value to the caller. If `Expression` is omitted, the return value is the `undefined` value. Otherwise, the return value is the value of `Expression`.

10.7 THE with STATEMENT

Syntax

```
WithStatement:
  with ( Expression ) Statement
```

Semantics

`Expression` is evaluated (call its value the “subject”). `Statement` is executed in a nested dynamic scope defined by the properties of the subject: variable reference and assignment expressions using variable names that are property names of the subject access and assign those properties. (See the section “Variable Scoping”.)

11 FUNCTION DEFINITION

Syntax

```
FunctionDeclaration:
  function Identifier ( FormalParameterList_opt ) Block
```

```
FormalParameterList:
  Identifier
  FormalParameterList , Identifier
```

Semantics

Defines a global variable whose name is the `Identifier` and whose value is a function object with the given parameter list and statements.

12 PROGRAM

Syntax
1.1.1

13 LANGUAGE SYNTAX SUMMARY

13.1 UNICODE

UnicodeEscapeSequence:
\u HexDigit HexDigit HexDigit 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

13.2 LEXICAL GRAMMAR

Any amount of Whitespace and Comments may appear between any two Tokens. For clarity, this is not explicitly stated in the grammar.

13.2.1 Whitespace

Whitespace:
<tab>
<linefeed>
<newline>
<end-of-medium>
<space>

13.2.2 Comments

Comment:
MultiLineComment
SingleLineComment

MultiLineComment:
/ * MultiLineCommentChars */

MultiLineCommentChars:
<any Unicode character except asterisk * > MultiLineCommentChars_opt
* PostAsteriskCommentChars_opt
13.2.3 Tokens

Token:
Keyword
Identifier
Punctuator
Operator
Literal

13.2.3.1 Keywords

Keyword: one of
abstract     else     int     switch
arguments    extends  interface  synchronized
boolean      false     long     this
break        final     native    throw
byte         finally   new     throws
case         float     null     transient
catch        for       package    true
char         function private    try
class        goto      protected  typeof
cost         if         public    var
continue     implements return    void
default      import    short     while
do           in         static    with
double       instanceof super

KeywordsInUse: one of
arguments    else     in     true
break        false    new     typeof
case         for       null    var
continue     function return    void
default      goto     switch    while
do           if        this    with

13.2.3.2 Identifiers

Identifier:
IdentifierPreface IdentifierBody

IdentifierPreface: one of
Letter
_ $
IdentifierBody:
  BodyCharacter
  IdentifierBody BodyCharacter

BodyCharacter: one of
  Letter
  \ 1 2 3 4 5 6 7 8 9
  _ $
  Letter: one of
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

13.2.3.3 Operators

Operator: one of
  = != ^ << ~
  += <<= &&>>
  -= >= &>>
  *= >>>= ||
  /= ?=: <= %
  %== | | /= ?: 

13.2.3.4 Literals

Literal:
  NullLiteral
  BooleanLiteral
  NumericLiteral
  StringLiteral

NullLiteral:
null

BooleanLiteral: one of
  true false

NumericLiteral:
  DecimalLiteral
  HexLiteral
  OctalLiteral

DecimalLiteral:
  IntegralLiteral
  RealLiteral Exponent

IntegralLiteral:
  NonzeroDecimalDigit DecimalDigits

NonzeroDecimalDigit: one of
  1 2 3 4 5 6 7 8 9
DecimalDigits:
  DecimalDigit
  DecimalDigits DecimalDigit

Digit: one of
  0 1 2 3 4 5 6 7 8 9

RealLiteral:
  DecimalDigits opt . DecimalDigits DecimalDigits .

Exponent:
  e Sign opt DecimalDigits
  E Sign opt DecimalDigits

Sign: one of
  + -

HexLiteral:
  0x HexDigits
  0X HexDigits

HexDigits:
  HexDigit
  HexDigits 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

OctalLiteral:
  0 OctalDigits

OctalDigits: one of
  0 1 2 3 4 5 6 7

StringLiteral:
  " NotDoubleQuoteStringCharacters "
  ' NotSingleQuoteStringCharacters '

NotDoubleQuoteStringCharacters:
  Any Unicode character except double-quote " , backslash \ and the newline character
  EscapeSequence

NotSingleQuoteStringCharacters:
  Any Unicode character except single-quote ' , backslash \ and the newline character
  EscapeSequence

EscapeSequence:
  CharacterEscapeSequence
  HexEscapeSequence
  UnicodeEscapeSequence

CharacterEscapeSequence: one of

35
HexEscapeSequence:
\x HexDigits

13.3 Phrase Structure Grammar

13.3.1 Expressions

PrimaryExpression:
  this
  arguments
  Identifier
  Literal
  ( Expression )

MemberExpression:
  PostfixExpression [ Expression ]
  PostfixExpression . Identifier

FunctionCallExpression:
  PostfixExpression ( ArgumentList )

PostfixExpression:
  PrimaryExpression
  MemberExpression
  FunctionCallExpression
  PostfixExpression ++
  PostfixExpression --

UnaryExpression:
  PostfixExpression
  new FunctionCallExpression
  delete MemberExpression
  void UnaryExpression
  typeof UnaryExpression
  ++ UnaryExpression
  -- UnaryExpression
  + UnaryExpression
  - UnaryExpression
  ~ UnaryExpression
  ! UnaryExpression

MultiplicativeExpression:
  UnaryExpression
  MultiplicativeExpression * UnaryExpression
  MultiplicativeExpression / UnaryExpression
  MultiplicativeExpression % UnaryExpression

AdditiveExpression:
  MultiplicativeExpression
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression

ShiftExpression:
  AdditiveExpression
  ShiftExpression << AdditiveExpression
  ShiftExpression >>= AdditiveExpression
  ShiftExpression >>> AdditiveExpression

RelationalExpression:
  ShiftExpression
  RelationalExpression < ShiftExpression
  RelationalExpression > ShiftExpression
  RelationalExpression <= ShiftExpression
  RelationalExpression >= ShiftExpression

EqualityExpression:
  RelationalExpression
  EqualityExpression == RelationalExpression
  EqualityExpression != RelationalExpression

BitwiseANDExpression:
  EqualityExpression
  BitwiseANDExpression & EqualityExpression

BitwiseXORExpression:
  BitwiseANDExpression
  BitwiseXORExpression ^ BitwiseANDExpression

BitwiseORExpression:
  BitwiseXORExpression
  BitwiseORExpression | BitwiseXORExpression

LogicalANDExpression:
  BitwiseORExpression
  LogicalANDExpression && BitwiseORExpression

LogicalORExpression:
  LogicalANDExpression
  LogicalORExpression || LogicalANDExpression

ConditionalExpression:
  LogicalORExpression
  LogicalORExpression ? Expression : ConditionalExpression

AssignmentExpression:
  ConditionalExpression
  UnaryExpression AssignmentOperator AssignmentExpression

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

Expression:
  AssignmentExpression
  Expression , AssignmentExpression
13.3.2 Statements

Statement:
   Block
   VariableStatement
   ExpressionStatement
   IfStatement
   IterationStatement
   ControlFlowStatements
   SwitchStatement
   WithStatement

Block:
   { Statementsopt }

Statements:
   Statement
   Statements Statement

VariableStatement::
   var VariableDeclarationList ;

VariableDeclarationList:
   VariableDeclaration
   VariableDeclarationList , VariableDeclaration

VariableDeclaration:
   Identifier Initializeropt

Initializer:
   = AssignmentExpression

ExpressionStatement:
   Expressionopt ;

IfStatement:
   if ( Expression ) Statement
   if ( Expression ) Statement else Statement

IterationStatement:
   while ( Expression ) Statement
   for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
   for ( var Identifier in Expression ) Statement

ControlFlowStatement:
   continue ;
   break ;
   return Expressionopt ;

SwitchStatement:
   switch ( Expression ) CaseStatements

CaseStatements:
   CaseStatement
   CaseStatements CaseStatement
CaseStatement:
   case Literal : Statement
   default : Statement

WithStatement:
   with ( Expression ) Statement

13.3.3 Function Definition

FunctionDeclaration:
   function Identifier ( FormalParameterList opt ) Block

FormalParameterList:
   Identifier
   FormalParameterList , Identifier

13.3.4 Program

Program:
   SourceElements

SourceElements:
   SourceElement
   SourceElements SourceElement

SourceElement:
   Statement
   FunctionDefinition

14 REFERENCES
INDEX

- subtraction.................................................. 25
-- 23
!
  Logical NOT.............................................. 24
%
  modulus................................................... 25
&
  bitwise AND............................................. 26
&&
  logical AND............................................. 27
*
  multiplication.......................................... 25
,
  comma operator........................................ 28
/
  division.................................................. 25
?
  conditional expression.............................. 27
~
  Bitwise NOT............................................. 24
++
  increment.............................................. 23
<<
  left shift............................................... 26
=
  assignment............................................. 27
>>
  right shift............................................. 26
>>>.
  unsigned right shift.................................. 26
arguments............................................... 22
arrays.................................................... 22
Boolean
  literal.................................................. 9
break..................................................... 30
Comments............................................... 6
control flow........................................... 30
Copyright Notice..................................... 1
equation.................................................. 21
expression............................................... 21
expressions
  syntax summary...................................... 35
for........................................................ 29
for . . in............................................... 30
function
  calls.................................................. 22
definition.............................................. 30
function definition
  syntax summary..................................... 38
Identifiers............................................. 8
if 29
iteration................................................ 29
Keywords................................................. 7
  used by JScript........................................ 8, 32
language syntax summary............................. 31
lexical grammar
  syntax summary...................................... 31
Lexical
  White Space.......................................... 6
operators
  additive, semantics.................................. 25
equality.................................................. 26
postfix.................................................. 22
relational.............................................. 26
unary.................................................... 23
Operators............................................... 8
phrase structure grammar
  syntax summary..................................... 35
references.............................................. 38
return................................................... 30
shift..................................................... 25
source text............................................ 31
  syntax summary...................................... 38
statements............................................. 28
  blocks................................................ 28
expression............................................. 28, 29
  syntax summary..................................... 37
this..................................................... 21
Token.................................................... 6
type
  boolean............................................... 11
  null.................................................... 11
  string............................................... 11
  undefined.......................................... 11
typeof.................................................. 24
types.................................................... 11
unicode
  syntax summary..................................... 31
Unicode................................................. 6
void.................................................... 24
while................................................... 29
White Space.......................................... 6
with.................................................... 30