ECMASCRIPT LANGUAGE SPECIFICATION  
ECMA COMMITTEE #39  
VERSION 0.3

JANUARY 10, 1997

FEEDBACK
Please send feedback regarding this document to Randy Solton (rsolton@wpo.borland.com) or Mike Gardner (mgardner@wpo.borland.com)
CHAPTER 0

INTRODUCTION

There are three known implementations of ECMAScript in common use today: Netscape JavaScript 1.1, Borland JavaScript 1.1 and Microsoft JScript. All three implementations share a great deal in common. For the purposes of the document, it is considered, “the norm”, whenever all three existing implementations exactly agree on a particular language element. Whenever any one of the implementations deviates from the norm, it is called out with Note or footnote. Any features that are unique to a given implementation are listed in Appendix Proposed Extensions.
CHAPTER 1

NOTATIONAL CONVENTIONS

1.1 SYNTACTIC AND LEXICAL PRODUCTIONS
Terminal symbols are shown in fixed width font in the productions of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring 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 a colon ‘:’ for syntactic productions or by two colons ‘::’ for lexical productions and then followed by one or more definitions. Definitions appear on separate lines. Alternatively, when the words "one of" follow the colon ‘:’ in a grammar definition or follow the two colons ‘::’ in a lexical production, they signify that each of the terminal symbols on the following line or lines is an alternative definition.

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.

The right-hand side of a lexical production may specify that certain expansions are not permitted by using the word "except" and then indicating the expansions to be excluded.

1.2 ALGORITHM CONVENTIONS
We often use a numbered list to specify steps in an algorithm. When the algorithm is to produce a value as a result, we use the directive “return x” to indicate that the result of the algorithm is the value of x and that the algorithm should terminate. We use the notation Result(n) as short hand for “the result of step n”. We also use Type(x) as short hand for “the type of x”. If an algorithm is defined to “generate a runtime error”, execution of the algorithm (and any calling algorithms) is terminated and no result is returned.

These algorithms are used to clarify semantics. In practice, there may be more efficient algorithms available to implement a given feature.
CHAPTER 2

SOURCE TEXT

2.1 UNICODE
ECMAScript 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:

\u \text{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.

2.2 END OF SOURCE
For purposes of describing the grammar of ECMAScript, the source text is assumed to be terminated by an “end of source” character, unicode character \u0000. We represent the end of source character by <EOS>.

<EOS>
CHAPTER 3

LEXICAL CONVENTIONS

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

3.1 WHITE SPACE

White space characters are used to improve source text readability and to separate tokens, indivisible
lexical units, from each other but are otherwise insignificant. White space may occur between any two
tokens, but not within a token. White space may also occur inside a string, where it is significant.

The following characters are considered white space:

<table>
<thead>
<tr>
<th>Unicode Value</th>
<th>Name</th>
<th>Formal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\u0009</td>
<td>Tab</td>
<td>&lt;TAB&gt;</td>
</tr>
<tr>
<td>\u000A</td>
<td>Line Feed (LF)</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>\u000D</td>
<td>Carriage Return (CR)</td>
<td>&lt;CR&gt;</td>
</tr>
<tr>
<td>\u0019</td>
<td>End of medium (^Z)</td>
<td>&lt;EOM&gt;</td>
</tr>
<tr>
<td>\u0020</td>
<td>Space</td>
<td>&lt;SP&gt;</td>
</tr>
</tbody>
</table>

Syntax

\hspace{1cm}

WhiteSpace ::

    SimpleWhiteSpace WhiteSpace\opt
    LineTerminator WhiteSpace\opt
    Comment WhiteSpace\opt

SimpleWhiteSpace ::

    <TAB>
    <EOM>
    <SP>

LineTerminator ::

    <CR>
    <LF>

3.2 COMMENTS

Description

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

Syntax

Comment ::

    MultiLineComment
    SingleLineComment

MultiLineComment ::

    /* MultiLineCommentChars*/
MultiLineCommentChars::
   MultiLineNotAsteriskChar MultiLineCommentChars
   * PostAsteriskCommentChars

PostAsteriskCommentChars::
   MultiLineNotFowardSlashChar
   MultiLineCommentChars

MultiLineNotAsteriskChar:
   <any Unicode character except asterisk* and <EOS>>

MultiLineNotFowardSlashChar:
   <any Unicode character except forward-slash / and <EOS>>

SingleLineComment::
   // SingleLineCommentChars LineTerminator
   // SingleLineCommentChars EndOfSource

SingleLineCommentChars::
   <any Unicode character except <LF>, <CR> and <EOS>> SingleLineCommentChars

3.3 TOKENS

Syntax
Token ::
   ReservedWord
   Identifier
   Punctuator
   Literal
   EndOfSource

3.3.1 Reserved Words

Description
Reserved words cannot be used as identifiers.

ReservedWord::
   Keyword
   FutureReservedWord
   NullLiteral
   BooleanLiteral
3.3.1.1 Keywords
The following keywords are in use in either the the Borland ECMAScript implementation, the Netscape
1.1 ECMAScript implementation, the Microsoft JScript implementation or all three.

Syntax

Keyword:: one of

<table>
<thead>
<tr>
<th>Argument</th>
<th>Break</th>
<th>Case</th>
<th>Catch</th>
<th>Else</th>
</tr>
</thead>
<tbody>
<tr>
<td>arguments^1,3</td>
<td>continue</td>
<td>default^1,3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>class^2</td>
<td>finally^1</td>
<td>for</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extends^1</td>
<td>implicit^3</td>
<td>in</td>
<td>new</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>switch^1,3</td>
<td>this</td>
<td>try^1</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>var</td>
<td>void</td>
<td>while</td>
<td></td>
</tr>
<tr>
<td>typeof</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.3.1.2 Future Reserved Words
The following keywords are not currently used in any ECMAScript implementation but are nevertheless
reserved for future borrowing from the Java language.

Syntax

FutureReservedWord:: one of

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Boolean</th>
<th>Byte</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>Do</td>
<td>Double</td>
<td>Final</td>
</tr>
<tr>
<td>Float</td>
<td>Goto</td>
<td>Implements</td>
<td>Import</td>
</tr>
<tr>
<td>Instanceof</td>
<td>Int</td>
<td>Interface</td>
<td>Long</td>
</tr>
<tr>
<td>Native</td>
<td>Package</td>
<td>Private</td>
<td>Protected</td>
</tr>
<tr>
<td>Public</td>
<td>Short</td>
<td>Static</td>
<td>Super</td>
</tr>
<tr>
<td>Synchronized</td>
<td>Throws</td>
<td>Transient</td>
<td>Volatile</td>
</tr>
</tbody>
</table>

3.3.2 IDENTIFIERS

Description
An identifier is a sequence of letters, digits and special characters that must begin with a letter.
ECMAScript identifiers are case sensitive: identifiers whose characters differ only in case are considered
unique.

Syntax

Identifier::

IdentifierName but not ReservedWord

IdentifierName::

IdentifierLetter

IdentifierName IdentifierLetter

IdentifierName DecimalDigit

IdentifierLetter:: one of

a b c d e f g h i j k l m n o p q r s t u v w x y z

^1 Borland Only
^2 Netscape Only
^3 Microsoft Only
DecimalDigit:: one of 0 1 2 3 4 5 6 7 8 9

3.3.3 PUNCTUATORS

Syntax
  Punctuator:: one of
    =  >  <  ==  <=  >=
    !  !  ~  ?  :
    .  &&  |  |  ++  --  _
    -  *  /  &  |  ^
    %  <<  >>  >>>  +=  -=
    *= /= &= |= ^= %=
    <<= >>= >>>= ( ) { }
    [ ] ; ,

3.3.4 LITERALS

Syntax
  Literal::
    Nullable
    BooleanLiteral
    NumericLiteral
    StringLiteral

3.3.4.1 Nullable

Syntax
  Nullable::
    null

3.3.4.2 Boolean Literals

Syntax
  BooleanLiteral::
    true
    false

3.3.4.3 Numeric Literals

Syntax
  NumericLiteral::
    IntegerLiteral
    FloatingPointLiteral

  IntegerLiteral::
    DecimalIntegerLiteral
    HexIntegerLiteral
    OctalIntegerLiteral
DecimalIntegerLiteral::
  0
  NonZeroDigit Digits opt

Digits ::
  Digit
  Digits Digit

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

HexIntegerLiteral::
  0x HexDigit
  0X HexDigit
  HexLiteral 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

FloatingPointLiteral::
  Digits . Digits opt ExponentPart opt
  . Digits ExponentPart opt
  Digits ExponentPart

ExponentPart::
  ExponentIndicator SignedInteger

ExponentIndicator:: one of
  e E

SignedInteger::
  Sign opt Digits

Sign:: one of
  + -

3.3.4.4 String Literals

A string literal is zero or more characters enclosed in single or double quotes. Each character may be represented by an escape sequence.

Syntax
  StringLiteral::
    “ DoubleStringCharacters opt “
    ‘ SingleStringCharacters opt ’
DoubleStringCharacter::
    Any Unicode character except double-quote “, backslash \, <CR>, <LF> or <EOS>
CharacterEscapeSequence

SingleStringCharacter::
    Any Unicode character except single-quote ’, backslash \, <CR>, <LF> or <EOS>
EscapeSequence

EscapeSequence::
    CharacterEscapeSequence
    OctalEscapeSequence
    HexEscapeSequence

CharacterEscapeSequence:: one of
    \'    \"    \\    \b    \f    \n    \r    \t

HexEscapeSequence::
    \x 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

OctalEscapeSequence::
    \ OctalDigit
    \ OctalDigit OctalDigit
    \ ZeroToThree OctalDigit OctalDigit

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

ZeroToThree:: one of
    0 1 2 3

Issue: Do we want to support octal and hex escape sequences.? If so, how do we handle unicode?

The following table describes the set of character escape characters:

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Name</th>
<th>Formal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>Backspace</td>
<td>&lt;BS&gt;</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab</td>
<td>&lt;HT&gt;</td>
</tr>
<tr>
<td>\n</td>
<td>line feed (new line)</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>\f</td>
<td>form feed</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
<td>&lt;CR&gt;</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote</td>
<td>&quot;</td>
</tr>
<tr>
<td>'</td>
<td>Single quote</td>
<td>‘</td>
</tr>
</tbody>
</table>

Note1: Only the Netscape implementation currently supports OctalEscapeSequence and HexEscapeSequence for character escape sequences.

Issue: If Hex escape sequences are to be supported, are they unicode?

Constraints
The maximum string constant supported must be at least 32000 characters long.
**Issue:** What is the explanation for this number 32000?

### 3.4 AutomaticSemicolonInsertion

**Description**

ECMAScript statements must be terminated with a semicolon. Under certain conditions, the parser injects semicolons into the token stream. When a token (called the offending token) is encountered that is not allowed by any production of the grammar, the parser will inject a semicolon immediately before the offending token in the following situations:

1. If the offending token is separated from the previous token by at least one LineTerminator.
2. If the offending token is EndOfSource.
3. If the offending token is } . This rule may only be applied once per instance of }.

**Discussion**

For example, the source

```
{ 1 2 } 3<EOS>
```

is not a valid sentence in the ECMAScript grammar. The source

```
{ 1
2 } 3<EOS>
```

is also not a valid ECMAScript sentence, but is transformed by automatic semicolon insertion into the following:

```
{ 1
;2 ;} 3;<EOS>
```

which is a valid ECMAScript sentence.
CHAPTER 4

TYPES
A value is an entity that takes on one of eight types. There are seven standard types and one internal type called Reference. Values of type Reference are only used as intermediate results of expression evaluation and cannot be stored to properties of objects.

4.1 THE UNDEFINED TYPE
The Undefined type has exactly one value called undefined. Any variable that has not been assigned a value is of type undefined.

4.2 THE NULL TYPE
The Null type has exactly one value called null.

4.3 THE BOOLEAN TYPE
The Boolean type represents a logical entity and consists of two unique values called true and false.

4.4 THE NUMBER TYPE
The Number type consists of all real values, as well as three special values called Positive Infinity, Negative Infinity and NaN.

4.5 THE OBJECT TYPE
An Object is an unordered collection of properties. Each property consists of a name, a value and an attribute.

4.5.1 Property Attributes
A property can have zero or more attributes from the following set:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadOnly</td>
<td>The property is a read-only property. Attempts to write to the property will be ignored.</td>
</tr>
<tr>
<td>ErrorOnWrite</td>
<td>This attribute has precedence over the ReadOnly attribute. Attempts to write to the property will result in a runtime error and the property will not be changed.</td>
</tr>
<tr>
<td>DontEnum</td>
<td>The property is not included in the for-in enumeration. See the description of the for-in statement in section 8.4.3 The for..in Statement</td>
</tr>
<tr>
<td>NotImplicit</td>
<td>The property is not accessible to an implicit property access.</td>
</tr>
<tr>
<td>NotExplicit</td>
<td>The property is not accessible to an explicit property access.</td>
</tr>
<tr>
<td>Permanent</td>
<td>Attempts to delete the property will be ignored. See the description of the delete operator in section 7.3.1 The delete Operator.</td>
</tr>
<tr>
<td>Internal</td>
<td>Internal properties have no name and are not directly accessible via the property accessor operators. How these properties are accessed is implementation specific. How and when some of these properties are used is specified by the language specification. A property with the Internal attribute also has the NotImplicit and NotExplicit attributes.</td>
</tr>
</tbody>
</table>
4.5.2 Property Access

Internal properties and methods are not exposed in the language. For the purposes of this document, we give them names enclosed in double square brackets[[ ]]. When an algorithm uses an internal property of an object and the object does not implement the indicated internal property, a runtime error is generated.

Native ECMAScript objects have an internal property called [[Prototype]]. The value of this property is either null or an object and is used for implementing inheritance. Properties of the [[Prototype]] object are exposed as properties of the child object for the purposes of get access, but not for put access.

There are two types of access for exposed properties: get and put, corresponding to retrieval and assignment.

The following table summarizes the internal properties related to property access:

<table>
<thead>
<tr>
<th>Property</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Get]]</td>
<td>(PropertyName, AccessMode)</td>
<td>Returns the value of the property.</td>
</tr>
<tr>
<td>[[Put]]</td>
<td>(PropertyName, AccessMode, Value)</td>
<td>Sets the property to value.</td>
</tr>
<tr>
<td>[[Prototype]]</td>
<td>None</td>
<td>Returns an the parent object</td>
</tr>
<tr>
<td>[[HasProperty]]</td>
<td>(PropertyName, AccessMode)</td>
<td>Returns a boolean value indicating whether the object already has a member with the given name and is accessible via the given access mode.</td>
</tr>
<tr>
<td>[[Construct]]</td>
<td>Optional user provided parameters</td>
<td>(Constructor) Constructs an object. Invoked via the new operator.</td>
</tr>
<tr>
<td>[[Call]]</td>
<td>Optional user provided parameters</td>
<td>(Function) Calls the method on the object.</td>
</tr>
</tbody>
</table>

There are two property access contexts: implicit and explicit. Explicit access is used when an object is named explicitly as in

```
myObject.x = 5
```

Implicit access is used when the object is not explicitly named such as within a with block as in:

```
with (myObject) {
    x = 5;
}
```

Assume $O$ is a ECMAScript object and $P$ is a string.

4.5.2.1 HasProperty

When the [[HasProperty]] method of $O$ is called with property name $P$, the following steps are taken:

1. If $O$ doesn’t have a property with name $P$, go to step 5.
2. If the access mode is explicit and the property has the NotExplicit attribute, return false.
3. If the access mode is implicit and the property has the NotImplicit attribute, return false.
4. Return true.
5. If the [[Prototype]] of $O$ is null, return false.
6. Call the [[HasProperty]] method of [[Prototype]] with property name $P$ and the same access mode.
7. Return Result(6).

4.5.2.2 Get

When the [[Get]] method of $O$ is called with property name $P$, the following steps are taken:

1. If $O$ doesn’t have a property with name $P$, go to step 6.
2. If the access mode is explicit and the property has the NotExplicit attribute, return **undefined**
3. If the access mode is implicit and the property has the NotImplicit attribute, return **undefined**
4. Get the value of the property.
5. Return Result(4).
6. If the [[Prototype]] of O is **null**, return **undefined**
7. Call the [[Get]] method of [[Prototype]] with property name P and the same access mode.
8. Return Result(7).

### 4.5.2.3 Put with Explicit Access Mode

When the [[Put]] method of O is called with property P and value V in explicit access mode, the following steps are taken:
1. If O doesn’t have a property with name P, go to step 6.
2. If the property has the ErrorOnWrite attribute, generate a runtime error.
3. If the property has the ReadOnly attribute, return.
4. Set the value of the property to V and clear the NotExplicit attribute.
5. Return.
6. Create a property with name P, set its value to V and give it empty attributes.
7. Return.

### 4.5.2.3 Put with Implicit Access Mode

When the [[Put]] method of O is called with property P and value V in implicit access mode, the following steps are taken:
1. If O doesn’t have a built-in property with name P then generate a runtime error.
2. If the property has the NotImplicit attribute, generate a runtime error.
3. If the property has the ErrorOnWrite attribute, generate a runtime error.
4. If the property has the ReadOnly attribute, return.
5. Set the value of the property to V.
6. Return

### 4.6 The String Type

The String Type is a sequence of Unicode characters. Note: Concatenations (+) operator, relational operators and equality operators apply to this type.

### 4.7 The InternalReference Type

The Internal Reference Type is not a language data type. It is only defined here for the purposes of aiding this specification.

A **Reference** is a reference to an object’s property. A **Reference** consists of three parts, the **base object**, the **propertyName** and the **access mode**.

In defining the semantics of ECMAScript, the following methods are defined for internal operations:
- GetBase(). Returns the base object component.
- GetPropertyName(). Returns the propertyName component.
- GetAccess(). Returns the access mode component.
- GetValue(). Returns the value of the indicated property using the indicated access mode.
- PutValue(). Sets the indicated property to the indicated value using the indicated access mode.

Values of type **Reference** are only used as intermediate results of expression evaluation and cannot be stored to properties of objects.
4.7.1 GetBase
1. If Type(V) is a Reference, return the base object component of V.
2. Generate a runtime error.

4.7.2 GetProperty
1. If Type(V) is a Reference, return the propertyName component of V.
2. Generate a runtime error.

4.7.3 GetAccess
1. If Type(V) is a Reference, return the access mode component of V.
2. Generate a runtime error.

4.7.4 GetValue
1. If Type(V) is not a Reference, return V.
2. Call GetBase(V)
3. If Result(2) is null, generate a runtime error.
4. Call the [[Get]] method of Result(2), passing GetProperty(V) for the property name and GetAccess(V) for the access mode.
5. Return Result(4).

4.7.5 PutValue
For values V and W, PutValue(V, W) performs:
1. If type (V) is not a Reference, generate a runtime error.
2. Call GetBase(V)
3. If Result(2) is null, go to step 6.
4. Call the [[Put]] method of Result(2), passing GetProperty(V) for the property name, GetAccess(V) for the access mode and W for the value.
5. Return
6. Call the [[Put]] method for the global object, passing GetProperty(V) for the property name, explicit for the access mode and W for the value.
7. Return
CHAPTER 5

TYPE CONVERSION

The ECMAScript runtime system performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion operators. These operators are not a part of the language; they are defined here to aid the specification of the semantics of the language. The conversion operators are polymorphic; that is, they can accept a value of any standard type, but not of type Reference.

5.1 ToPrimitive(PreferredType)

The operator ToPrimitive attempts to convert its argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the hint PreferredType to favor that type.

Conversion occurs according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Return the input argument (no conversion)</td>
</tr>
<tr>
<td>Null</td>
<td>Return the input argument (no conversion)</td>
</tr>
<tr>
<td>Boolean</td>
<td>Return the input argument (no conversion)</td>
</tr>
<tr>
<td>Number</td>
<td>Return the input argument (no conversion)</td>
</tr>
<tr>
<td>String</td>
<td>Return the input argument (no conversion)</td>
</tr>
<tr>
<td>Object</td>
<td>Return the default value of the Object. If the default value is of type Object or Reference, a runtime error is generated. The default value of an object is retrieved by calling the internal [[DefaultValue]] method of the object. The behavior of the [[DefaultValue]] method is defined by this specification for all native ECMAScript objects.</td>
</tr>
</tbody>
</table>

Errors are never generated as a result of calling ToPrimitive.

5.2 ToBoolean

The operator ToBoolean attempts to convert its argument to a value of type Boolean according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>false</td>
</tr>
<tr>
<td>Null</td>
<td>false</td>
</tr>
<tr>
<td>Boolean</td>
<td>Return the input argument (no conversion)</td>
</tr>
</tbody>
</table>
| Number     | 0 → false  
             ≠ 0 → true  
             NaN → false |
| String     | = "" → false (where "" denotes an empty string)  
             ≠ "" → true |
| Object     | true   |

5.3 ToNumber

The operator ToNumber attempts to convert its argument to a value of type Number according to the following table:
<table>
<thead>
<tr>
<th><strong>Input Type</strong></th>
<th><strong>Result</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>NaN</td>
</tr>
<tr>
<td>Null</td>
<td>0</td>
</tr>
</tbody>
</table>
| Boolean       | true → 1  
|               | false → 0 |
| Number        | Return the input argument (no conversion) |
| String        | See grammar and discussion below. |
| Object        | Apply the following steps:  
|               | 1. Call ToPrimitive on the input argument.  
|               | 2. Call ToNumber(Result(1)).  
|               | 3. Return Result(2). |

### 5.3.1 ToNumber Applied to the String Type

ToNumber applied to strings applies the following grammar to the input string. If the grammar cannot interpret the string then the result of ToNumber is **NaN**.

```
StringNumericLiteral::
    StrWhiteSpaceopt StrNumericLiteral StrWhiteSpaceopt

StrWhiteSpace::
    SimpleWhiteSpace StrWhiteSpaceopt

SimpleWhiteSpace::
    <TAB>
    <EOM>
    <SP>

StrNumericLiteral::
    StrDecimalLiteral
    HexLiteral

StrDecimalLiteral::
    Signopt DecimalDigits Exponentopt
    Signopt DecimalDigits . DecimalDigitsopt Exponentopt
    Signopt . DecimalDigits Exponentopt

DecimalDigits::
    DecimalDigit
    DecimalDigits DecimalDigit

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

Exponent::
    e Signopt DecimalDigits
    E Signopt DecimalDigits

Sign:: one of
    + -
```
**HexLiteral::**

- \(0x\) HexDigit
- \(0X\) HexDigit
- HexLiteral 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

**Issues:** Should we allow hex in `ToNumber(string)`?

### 5.4 ToInteger

The operator ToInteger attempts to convert its argument to an integral numeric value. This operator functions as follows:

1. Call `ToNumber` on the input argument.
2. If `Result(1)` is \(NaN\), return 0.
3. If `Result(1)` is \(±\infty\), return `Result(1)`.
4. Compute \(\text{sign}(\text{Result}(1)) \times \text{floor}(\text{abs}(\text{Result}(1)))\).
5. Return `Result(4)`.

### 5.5 ToInt32: (signed 32 bit integer)

The operator ToInt32 attempts to convert its argument to an integral numeric value representable as a signed 32 bit integer. This operator functions as follows:

1. Call `ToNumber` on the input argument.
2. If `Result(1)` is \(NaN\), return 0.
3. Return whatever IEEE does. **Issue:** define this.

### 5.6 ToUint32: (unsigned 32 bit integer)

The operator ToUint32 attempts to convert its argument to an integral numeric value representable as an unsigned 32 bit integer. This operator functions as follows:

1. Call `ToNumber` on the input argument.
2. If `Result(1)` is \(NaN\), return 0.
3. Return whatever IEEE does. **Issue:** define this.

### 5.7 ToString

The operator ToString attempts to convert its argument to a value of type String according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>&quot;undefined&quot;</td>
</tr>
<tr>
<td>Null</td>
<td>&quot;null&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>true ➔ &quot;true&quot;</td>
</tr>
<tr>
<td></td>
<td>false ➔ &quot;false&quot;</td>
</tr>
<tr>
<td>Number</td>
<td>See discussion below.</td>
</tr>
<tr>
<td>String</td>
<td>Return the input argument (no conversion)</td>
</tr>
<tr>
<td>Object</td>
<td>Apply the following steps:</td>
</tr>
</tbody>
</table>
1. Call ToPrimitive on the input argument.
2. Call ToString(Result(1)).
3. Return Result(2).

5.7.1 ToString Applied to the Number Type

**Issue:** define this.

5.8 **ToObject**

The operator ToObject attempts to convert its argument to a value of type Object according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>generate a runtime error</td>
</tr>
<tr>
<td>Null</td>
<td>generate a runtime error</td>
</tr>
<tr>
<td>Boolean</td>
<td>Create a Boolean object whose default value is the value of the boolean. See the <a href="#">Object Model section</a> for a description of the Boolean object.</td>
</tr>
<tr>
<td>Number</td>
<td>Create a Number object whose default value is the value of the number. See the <a href="#">Object Model section</a> for a description of the Number object.</td>
</tr>
<tr>
<td>String</td>
<td>Create a String object whose default value is the value of the string. See the <a href="#">Object Model section</a> for a description of the String object.</td>
</tr>
<tr>
<td>Object</td>
<td>Return the input argument (no conversion)</td>
</tr>
</tbody>
</table>
CHAPTER 6

VARIABLES

6.1 SCOPE RESOLUTION

All names have scope. A fully qualified name identifies the scope explicitly. Unqualified names have an implicit scope.

The semantics of resolving the implicit scope, entails searching all active objects in a scope chain (stack), that is maintained by the ECMAScript runtime system, until a suitable scope is found. This can be accomplished by iteratively calling the [[HasProperty]] method for each object in the scope chain.

Note: this is merely a semantic description. Actual implementation may vary.

6.2 Global Object

There is a unique global object that is always the last element in the scope chain. The global object contains as properties the following:

• Variables declared in global code. These have initial value undefined and attributes
  { DontEnum }.
• Declared functions. The value is the function object and the attributes are { DontEnum }.
• Builtin objects such as Math, String, Date, parseInt, etc. These have attributes { DontEnum }.
• Implicitly created variables. Recall that if PutVal is called on a Reference value whose base value is null, [[Put]] is called on the global object. This creates a new member of the global object.

Depending on the host environment, the global object may have itself as a named property (such as window in HTML) and may have additional properties, defined by the host, including internal properties.

Issue: should declared functions and builtin objects have attributes {DontEnum, ErrorOnWrite, Permanent}?

6.3 Local Variables

There are four types of source code identifiable executions scopes in ECMAScript: global, function code, eval code, anonymous code. If a local variable is created within an execution scope as with the var statement, the name associated with the variable is added to the active source code execution scope.

6.4 With Blocks

Once an execution scope is entered, whether it be global, function, eval or anonymous code, the only mechanism for modifying the scope chain is the with statement. When execution enters a with block, the object specified in the with statement is added to the front of the scope chain. When execution leaves a with block, whether normally or via a break or continue statement, the object is removed from the scope chain. The object being removed will always be the first object in the scope chain.
6.5 Eval Code

In eval code, the scope chain is initially the same as in the calling context. If the calling context has an arguments object, variables and functions declared in eval code get added to the arguments object. Otherwise, they are added to the global object. The eval code has the same arguments and this objects as its calling context. Eval code can be called from any type of code, including eval code.

6.6 Initial Variable Values

Every variable in a ECMAScript program has a value:

• If a global or local variable is used before it is set, its value is undefined.
• If an object property is used before it is set, its value is undefined.
• A formal function parameter is initialized to the corresponding actual argument value. If there is no corresponding actual argument, the formal parameter’s value is undefined.
CHAPTER 7

EXPRESSIONS

7.1 PRIMARY EXPRESSIONS

Syntax

PrimaryExpression:

this

identifier

Literal

( Expression )

7.1.1 The this Keyword

The this keyword has the following meanings:

1. In global code, this refers to the global object.
2. In eval code, this refers to the same object as in the calling context.
3. When function code or anonymous code is used as a constructor, that is, in the context of a new expression, this refers to the object being constructed.
4. When function code or anonymous code is called, the value of this is supplied by the caller. If the caller provides null, the global object is used instead.

7.1.2 Identifier Reference

An Identifier is evaluated using the scoping rules stated §6.1 Scope Resolution. The result of an Identifier is always a value of type Reference.

7.1.3 Literal Reference

A Literal is evaluated as described in section §3.4 Literals.

7.1.4 The Grouping Operator

The production PrimaryExpression: ( Expression ) is evaluated as follows:

1. Evaluate Expression. This may be of type Reference.
2. Return Result(1).

Issue: should this call GetValue(Result(1)) so that the result is never of type Reference? The above definition allows:

(x.y) = 10
(x.y)++

to assign 11 to x.y. It also specifies that

(x.y)(1,2,3)

calls method y with x as the “this” value, instead of the global object as “this” value.

7.2 POSTFIX EXPRESSIONS

Syntax
The postfix increment operators and property accessor operators\$[ ]\$ and \$\cdot\$ appear in both the 
MemberExpression and CallExpression productions. Generally we will refer to the productions involving 
MemberExpression with the understanding that the same remarks apply to CallExpression. Similarly, the 
CallExpression production includes three definitions involving the Arguments non-terminal. We will refer 
to the definition involving CallExpression.

### 7.2.1 Property Accessors

Properties are accessed by name, using either the dot notation MemberExpression \cdot Identifier or the 
bracket notation MemberExpression[ Expression ].

The dot notation is transformed using the following syntactic conversion:

\[ \text{MemberExpression} \cdot \text{Identifier} \]

is exactly equivalent to:
where <identifier-string> is a string literal containing the same sequence of characters as the identifier.

The production `MemberExpression: MemberExpression[ Expression ]` is evaluated as follows:
1. Evaluate `MemberExpression`.
2. Call `GetValue(Result(1))`.
3. Evaluate `Expression`.
4. Call `GetValue(Result(3))`.
5. Call `ToObject(Result(2))`.
6. Call `ToString(Result(4))`.
7. Return a value of type `Reference` whose base value is `Result(5)`, member name is `Result(6)` and access mode is explicit.

### 7.2.2 Postfix Increment and Decrement Operators

The production `MemberExpression: MemberExpression IncrementOperator` is evaluated as follows:
1. Evaluate `MemberExpression`.
2. Call `GetValue(Result(1))`.
3. Call `ToPrimitive(Result(1))`.
4. Call `ToNumber(Result(2))`.
5. For `++`, `Result(5)` is `Result(4)` increased by one. For `--`, `Result(5)` is `Result(4)` decreased by one. In either case, if `Result(4)` is `NaN` or `Infinity`, `Result(5)` is the same as `Result(4)`.
6. Call `PutValue(Result(1), Result(5))`.
7. Return `Result(2)`.

**Issue:** Should the expression result (step 6) be the value from step 2 or step 3? Since the prefix increment and decrement operators evaluate to the result of step 4 (a number) maybe the postfix operators should evaluate to the result of step 3.

### 7.2.3 The `new` Operator

The production `NewCallExpression: new MemberExpression Arguments` is evaluated as follows:
1. Evaluate `MemberExpression`.
2. Call `GetValue(Result(1))`.
3. For each `AssignmentExpression` in `ArgumentList`, in left to right order, evaluate `AssignmentExpression` and call `GetValue` on the result. Keep all of these values in an internal list.
4. If `Type(Result(2))` is not Object, generate a runtime error.
5. If `Result(2)` does not implement the internal `[[Construct]]` method, generate a runtime error.
6. Call the `[[Construct]]` method on `Result(2)`, providing the list generated in step 3 as the parameters.
7. If `Type(Result(6))` is not Object, generate a runtime error.
8. Return `Result(6)`.

### 7.2.4 Function Calls

The production `CallExpression: CallExpression Arguments` is evaluated as follows:
1. Evaluate `CallExpression`.
2. For each `AssignmentExpression` in `ArgumentList`, in left to right order, evaluate `AssignmentExpression` and call `GetValue` on the result. Keep all of these values in an internal list.
3. Call `GetValue(Result(1))`.
4. If `Type(Result(3))` is not Object, generate a runtime error.
5. If `Result(3)` does not implement the internal `[[Call]]` method, generate a runtime error.
6. If `Type(Result(1))` is `Reference`, `Result(6)` is `GetBase(Result(1))`. Otherwise `Result(6)` `null`. 
7. Call the [[Call]] method on Result(3), providing Result(6) as the `this` value and providing the list generated in step 2 as the parameters.
8. Return Result(7).

**Note:** Result(7) will never be of type Reference for native ECMAScript objects. Whether an external object can return a value of type Reference is implementation dependent.

### 7.3 Unary Operators

**Syntax**

```
UnaryExpression:
  PostfixExpression
  delete UnaryExpression
  void UnaryExpression
  typeof UnaryExpression
  IncrementOperator UnaryExpression
  + UnaryExpression
  - UnaryExpression
  ~ UnaryExpression
  ! UnaryExpression
```

#### 7.3.1 The `delete` Operator

The production `UnaryExpression: delete UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`.
2. Call GetBase(Result(1)).
3. Call GetProperty(Result(1)).
4. If `Type(Result(2))` is not Object, generate a runtime error.
5. If `Result(2)` does not implement the internal `[[Delete]]` method, generate a runtime error.
6. Call the `[[Delete]]` method on `Result(2)`, providing `Result(3)` as the property name to delete.
7. Return `undefined`

#### 7.3.2 The `void` Operator

The production `UnaryExpression: void UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`.
2. Call GetValue(Result(1)).
3. Return `undefined`

#### 7.3.3 The `typeof` Operator

The production `UnaryExpression: typeof UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`.
2. If `Type(Result(1))` is Reference and GetBase(Result(1)) is `null`, return "undefined".
3. Call GetValue(Result(1)).
4. Return a string determined by `Type(Result(3))` according to the following table:

<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>Object (implements</td>
<td>&quot;function&quot;</td>
</tr>
</tbody>
</table>
7.3.4 Prefix Increment and Decrement Operators

The production `UnaryExpression: IncrementOperator UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`
2. Call `GetValue(Result(1))`. Call `ToPrimitive(Result(2))`.
3. Call `ToPrimitive(Result(1))`.
4. Call `ToNumber(Result(2))`.
5. For `++`, `Result(5)` is `Result(4)` increased by one. For `--`, `Result(5)` is `Result(4)` decreased by one. In either case, if `Result(4)` is \texttt{NaN} or \texttt{±Infinity}, `Result(5)` is the same as `Result(4)`.
6. Call `PutValue(Result(1), Result(5))`.
7. Return `Result(5)`.

\textbf{Issue:} should we return `Result(5)` or `GetValue(Result(1))`? `Result(5)` is the simplest and fastest.

7.3.5 Unary `+` and `-` Operators

The production `UnaryExpression: + UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`
2. Call `GetValue(Result(1))`.
3. Call `ToNumber(Result(2))`.
4. Return `Result(3)`.

The production `UnaryExpression: - UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`
2. Call `GetValue(Result(1))`.
3. Call `ToNumber(Result(2))`.
4. If `Result(3)` is \texttt{NaN}, return \texttt{NaN}.
5. Negate `Result(3)`.
6. Return `Result(5)`.

\textbf{Issue:} Should unary plus just Evaluate `UnaryExpression`?

7.3.6 The Bitwise NOT Operator (`~`)

The production `UnaryExpression: ~ UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`
2. Call `GetValue(Result(1))`.
3. Call `ToInt32(Result(2))`.
4. Apply bitwise complement to `Result(3)`.
5. Return `Result(4)`.

7.3.7 Logical NOT Operator (`!`)

The production `UnaryExpression: ! UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`
2. Call `GetValue(Result(1))`.
3. Call `ToBoolean(Result(2))`.
4. If `Result(3)` is \texttt{true}, return \texttt{false}.
5. Return \texttt{true}. 
7.4 **Multiplicative Operators**

**Syntax**

\[ \text{MultiplicativeExpression} : \]

\[ \text{UnaryExpression} \]

\[ \text{MultiplicativeExpression} \ast \text{UnaryExpression} \]

\[ \text{MultiplicativeExpression} / \text{UnaryExpression} \]

\[ \text{MultiplicativeExpression} \% \text{UnaryExpression} \]

**Semantics**
The production \( \text{MultiplicativeExpression} : \text{MultiplicativeExpression} @ \text{UnaryExpression} \) where @ stands for one of the operators in the above definitions, is evaluated as follows:

1. Evaluate \( \text{MultiplicativeExpression} \).
2. Call GetValue(Result(1)).
3. Evaluate \( \text{UnaryExpression} \).
4. Call GetValue(Result(3)).
5. Call ToNumber(Result(2)).
6. Call ToNumber(Result(4)).
7. If Result(5) is \( \text{NaN} \) or Result(6) is \( \text{NaN} \) then return \( \text{NaN} \).
8. If @ is either \( / \) or \( \% \) and Result(6) is 0 then return \( \text{NaN} \).
9. Apply the specified operation \( \ast, /, \) or \( \% \) to Result(5) and Result(6) in accordance to the IEEE definition of those operators.
10. Return Result(9).

**Issue:** define step 9 better.

7.5 **Additive Operators**

**Syntax**

\[ \text{AdditiveExpression} : \]

\[ \text{MultiplicativeExpression} \]

\[ \text{AdditiveExpression} + \text{MultiplicativeExpression} \]

\[ \text{AdditiveExpression} - \text{MultiplicativeExpression} \]

7.5.1 **The Addition Operator** ( + )
The addition operator either performs string concatenation or numeric addition.

The production \( \text{AdditiveExpression} : \text{AdditiveExpression} + \text{MultiplicativeExpression} \) is evaluated as follows:

1. Evaluate \( \text{AdditiveExpression} \).
2. Call GetValue(Result(1)).
3. Evaluate \( \text{MultiplicativeExpression} \).
4. Call GetValue(Result(3)).
5. Call ToPrimitive(Result(2)).
6. Call ToPrimitive(Result(4)).
7. If Type(Result(5)) is String or Type(Result(6)) is String, go to step 13.
8. Call ToNumber(Result(5)).
9. Call ToNumber(Result(6)).
10. If Result(8) or Result(9) is \( \text{NaN} \), return \( \text{NaN} \).
11. Apply the addition operation to Result(8) and Result(9) in accordance with the IEEE definition.
12. Return Result(11).
13. Call ToString(Result(5)).
14. Call ToString(Result(6)).
15. Concatenate Result(13) followed by Result(14).

**Issue:** Define step 11 better.

### 7.5.2 The Subtraction Operator ( - )

The production \( \text{AdditiveExpression} : \text{AdditiveExpression} - \text{MultiplicativeExpression} \) is evaluated as follows:

1. Evaluate \( \text{AdditiveExpression} \).
2. Call GetValue(Result(1)).
3. Evaluate \( \text{MultiplicativeExpression} \).
4. Call GetValue(Result(3)).
5. Call ToPrimitive(Result(2)).
6. Call ToPrimitive(Result(4)).
7. Call ToNumber(Result(2)).
8. Call ToNumber(Result(4)).
9. If Result(5) or Result(6) is NaN, return NaN.
10. Apply the subtraction operation to Result(7) and Result(8) in accordance with the IEEE definition.
11. Return Result(12).

**Issue:** Define step 10 better.

### 7.6 Bitwise Shift Operators

**Syntax**

\[
\text{ShiftExpression} :
\begin{align*}
\text{ShiftExpression} & \ll \text{AdditiveExpression} \\
\text{ShiftExpression} & \gg \text{AdditiveExpression} \\
\text{ShiftExpression} & \ggg \text{AdditiveExpression}
\end{align*}
\]

The result of evaluating \( \text{ShiftExpression} \) is always truncated to 32 bits. If the result of evaluating \( \text{ShiftExpression} \) produces a fractional component, the fractional component is discarded. The result of evaluating \( \text{AdditiveExpression} \) is always truncated to five bits.

#### 7.6.1 The Left Shift Operator ( \( \ll \) )

Performs a bitwise left shift operation on the left argument by the amount specified by the right argument.

The production \( \text{ShiftExpression} : \text{ShiftExpression} \ll \text{AdditiveExpression} \) is evaluated as follows:

1. Evaluate \( \text{ShiftExpression} \).
2. Call GetValue(Result(1)).
3. Evaluate \( \text{AdditiveExpression} \).
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToInt32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Left shift Result(5) by Result(7) bits. The result is a signed 32 bit integer.
9. Return Result(8).

**Issue:** Should we convert the left operand ToUint32 instead? Or should there be \( \lll \) operator that does this?
7.6.2 The Signed Right Shift Operator ( >> )
Performs a sign-filling bitwise right shift operation on the left argument by the amount specified by the right argument.

The production \texttt{ShiftExpression: ShiftExpression >> AdditiveExpression} is evaluated as follows:
1. Evaluate \texttt{ShiftExpression}.
2. Call GetValue(Result(1)).
3. Evaluate \texttt{AdditiveExpression}.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToInt32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Perform sign-extending right shift of Result(5) by Result(7) bits. The most significant bit is propagated. The result is a signed 32 bit integer.
9. Return Result(8).

7.6.3 The Unsigned Right Shift Operator ( >>> )
Performs a zero-filling bitwise right shift operation on the left argument by the amount specified by the right argument.

The production \texttt{ShiftExpression: ShiftExpression >>> AdditiveExpression} is evaluated as follows:
1. Evaluate \texttt{ShiftExpression}.
2. Call GetValue(Result(1)).
3. Evaluate \texttt{AdditiveExpression}.
4. Call GetValue(Result(3)).
5. Call ToUint32(Result(2)).
6. Call ToInt32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Perform zero-filling right shift of Result(5) by Result(7) bits. Vacated bits are filled with zero. The result is an unsigned 32 bit integer.
9. Return Result(8).

7.7 RELATIONAL OPERATORS

Syntax
\texttt{RelationalExpression: ShiftExpression RelationalExpression < ShiftExpression}
\texttt{RelationalExpression > ShiftExpression}
\texttt{RelationalExpression <= ShiftExpression}
\texttt{RelationalExpression >= ShiftExpression}

Semantics
In the discussion below, the following special operators will be used:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric@</td>
<td>Where @ represents one of the relational operators. The operands are of type Number. This is the standard IEEE operator with the provision that if either operand is NaN, the result is false.</td>
</tr>
<tr>
<td>Character@</td>
<td>Where @ represents one of the relational operators. The operands are of type String. The operands are compared character by character lexicographically in the unicode character set. If the operands are of different length and all</td>
</tr>
</tbody>
</table>
characters up to the length of the shorter operand are the same, the longer string is considered to be greater.

| Boolean@ | Where @ represents one of the relational operators. The operands are of type Boolean. If one operand is true and the other is false, the result is false. Otherwise, the result is true. |

The relational operators operate on two types: numbers and strings. The addition operator operates on the same types. Therefore, the conversion rules are the same for the addition operator and relational operators.

The production $\text{RelationalExpression: RelationalExpression} @ \text{ShiftExpression}$ where @ represents one of the relational operators, is evaluated as follows:

1. Evaluate $\text{RelationalExpression}$
2. Call GetValue(Result(1)).
3. Evaluate $\text{ShiftExpression}$
4. Call GetValue(Result(3)).
5. Call ToPrimitive(Result(2)).
6. Call ToPrimitive(Result(4)).
7. If Type(Result(5)) is String or Type(Result(6)) is String, go to step 13.
8. Call ToNumber(Result(5)).
9. Call ToNumber(Result(6)).
10. Apply Numeric@ to Result(8) and Result(9).
11. Return Result(10).
12. Call ToString(Result(5)).
13. Call ToString(Result(6)).
14. Apply Character@ to Result(12) and Result(13).
15. Return Result(14).

### 7.8 Equality Operators

**Syntax**

EqualityExpression:

- RelationalExpression
- EqualityExpression == RelationalExpression
- EqualityExpression != RelationalExpression

**Semantics**

The equality operators maintain the following invariants:

1. $A \neq B$ is equivalent to $!(A == B)$.
2. $A == B$ is equivalent to $B == A$, except in the order of evaluation of A and B.
3. If $A == B$ and $B == C$, then $A == C$, assuming no side effects.

As no conversions are applied to the operands, equality is always transitive.

The production $\text{EqualityExpression: EqualityExpression} == \text{RelationalExpression}$ is evaluated as follows:

1. Evaluate $\text{EqualityExpression}$
2. Call GetValue(Result(1)).
3. Evaluate $\text{RelationalExpression}$
4. Call GetValue(Result(3)).
5. If Type(Result(2)) is different than Type(Result(4)) return false.
6. If Type(Result(2)) is Undefined, return true.
7. If Type(Result(2)) is Null, return true.
8. If Type(Result(2)) is Number, apply Numeric== to Result(2) and Result(4) and return the result.
9. If Type(Result(2)) is String, apply Character== to Result(2) and Result(4) and return the result.
10. If Type(Result(2)) is Boolean, apply Boolean== to Result(2) and Result(4) and return the result.
11. If Type(Result(2)) is Object, return \texttt{true} or \texttt{false} according to whether the two object references refer to the same object.

The production \textit{EqualityExpression: EqualityExpression} \texttt{!= RelationalExpression} is evaluated as follows:
1. Evaluate the production \textit{EqualityExpression} \texttt{== RelationalExpression}
2. If Result(1) is \texttt{true}, return \texttt{false}.
3. Return \texttt{true}.

\textbf{Discussion}
String comparison can be forced by: \texttt{" + a == " + b}
Numeric comparison can be forced by: \texttt{a - 0 == b - 0}
Boolean comparison can be forced by: \texttt{!a == !b}
Equality operators are infallible.

\subsection*{7.9 Binary Bitwise Operators}

\textbf{Syntax}

\textit{BitwiseANDExpression:}
\begin{itemize}
  \item EqualityExpression
  \item BitwiseANDExpression & EqualityExpression
\end{itemize}

\textit{BitwiseXORExpression:}
\begin{itemize}
  \item BitwiseANDExpression
  \item BitwiseXORExpression ^ BitwiseANDExpression
\end{itemize}

\textit{BitwiseORExpression:}
\begin{itemize}
  \item BitwiseXORExpression
  \item BitwiseORExpression | BitwiseXORExpression
\end{itemize}

\textbf{Semantics}
The production \textit{A : A @ B, where @ is one of the bitwise operators in the productions above, is evaluated as follows:}
1. Evaluate A.
2. Call GetValue(Result(1)).
3. Evaluate B.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToInt32(Result(4)).
7. Apply the bitwise operator @ to Result(5) and Result(6). The result is a signed 32 bit integer.
8. Return Result(7).

\textbf{Issue:} should these apply ToInt32 or ToUint32? Current implementations do ToInt32: \texttt{(-1 & -1)} produces \texttt{-1}. Maybe we should use ToUint32.

\subsection*{7.10 Binary Logical Operators}

\textbf{Syntax}

\textit{LogicalANDExpression:}
\begin{itemize}
  \item BitwiseORExpression
  \item LogicalANDExpression & BitwiseORExpression
\end{itemize}
LogicalORExpression:
  LogicalANDExpression
  LogicalORExpression || LogicalANDExpression

Semantics
The production LogicalANDExpression: LogicalANDExpression && BitwiseORExpression is evaluated as follows:
1. Evaluate LogicalANDExpression
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is false, return false.
5. Evaluate BitwiseORExpression
6. Call GetValue(Result(5)).
7. Call ToBoolean(Result(6)).
8. Return Result(7).

The production LogicalORExpression: LogicalORExpression || LogicalANDExpression is evaluated as follows:
1. Evaluate LogicalORExpression
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is true, return true.
5. Evaluate LogicalANDExpression
6. Call GetValue(Result(5)).
7. Call ToBoolean(Result(6)).
8. Return Result(7).

7.11 CONDITIONALOPERATOR( ? : )

Syntax
  ConditionalExpression:
    LogicalORExpression
    LogicalORExpression ? Expression : ConditionalExpression

Semantics
The production ConditionalExpression: LogicalORExpression? Expression : ConditionalExpression is evaluated as follows:
1. Evaluate LogicalORExpression
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is false, go to step 8.
5. Evaluate Expression.
6. Call GetValue(Result(5)).
7. Return Result(6).
8. Evaluate ConditionalExpression
9. Call GetValue(Result(8)).
10. Return Result(9).

Issue: Currently implementations don’t call GetValue on the result before returning. That is, steps 6 and 8 are not performed. Should we preserve this?
7.12 ASSIGNMENT OPERATORS

Syntax

AssignmentExpression:
  ConditionalExpression
  UnaryExpression AssignmentOperator AssignmentExpression

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

7.12.1 Simple Assignment ( = )
The production AssignmentExpression: UnaryExpression = AssignmentExpression is evaluated as follows:
1. Evaluate UnaryExpression
2. Evaluate AssignmentExpression
3. Call GetValue(Result(2)).
4. Call SetVal(Result(1), Result(3)).
5. Return Result(3).

Issue: should we return Result(3), Result(1) or GetValue(Result(1))? Result(3) is the simplest and fastest. The Borland implementation and C++ return Result(1) so that expressions like (a = b) = c work. Same issue applies to the compound assignment operators.

7.12.2 Compound Assignment ( op= )
The production AssignmentExpression: UnaryExpression @= AssignmentExpression where @ represents one of the operators indicated above, is evaluated as follows:
1. Evaluate UnaryExpression
2. Evaluate AssignmentExpression
3. Call GetValue(Result(1)).
4. Call GetValue(Result(2)).
5. Apply operator @ to Result(3) and Result(4).
6. Call SetVal(Result(1), Result(5)).
7. Return Result(5).

7.13 COMMA OPERATOR ( , )

Syntax

Expression:
  AssignmentExpression
  Expression , AssignmentExpression

Semantics
The production Expression: Expression , AssignmentExpression is evaluated as follows:
1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Evaluate AssignmentExpression
4. Call GetValue(Result(3)).
5. Return Result(4).
CHAPTER 8

STATEMENTS

Syntax

Statement : Block
            VariableStatement
            ExpressionStatement
            IfStatement
            IterationStatement
            ControlFlowStatement
            WithStatement

Block : { StatementListopt }

StatementList : Statement
               StatementList Statement

Semantics

The production StatementList : StatementList Statement is evaluated as follows:
1. Evaluate StatementList.
2. Evaluate Statement.

8.1 VARIABLESTATEMENT

Syntax

VariableStatement : var VariableDeclarationList;

VariableDeclarationList : VariableDeclaration
                         VariableDeclarationList, VariableDeclaration

VariableDeclaration : Identifier Initializeropt

Initializer : = AssignmentExpression

Description

If the variable statement occurs inside a FunctionDeclaration the variables are defined with function-local scope in that function. Otherwise, they are defined with global scope, that is, they are created as members of the global object as described in section 6.2 Global Object Variables are created when the execution scope is entered. A Block does not define a new execution scope. Only Program and FunctionDeclaration produce a new scope. Eval code and anonymous code also define a new execution scope, but these are not an explicit part of the grammar of ECMAScript. Variables are initialized to the
**undefined** value when created. A variable with an **Initializer** is assigned the value of its **AssignmentExpression** when the **VariableStatement** is executed.

**Semantics**
The production **VariableStatement**: `var VariableDeclarationList;` is evaluated as follows:
1. Evaluate **VariableDeclarationList**
2. Return.

The production **VariableDeclarationList**: `VariableDeclarationList, VariableDeclaration` is evaluated as follows:
1. Evaluate **VariableDeclarationList**
2. Evaluate **VariableDeclaration**
3. Return.

The production **VariableDeclaration**: `Identifier = AssignmentExpression` is evaluated as follows:
1. Evaluate **Identifier**.
2. Evaluate **AssignmentExpression**
3. Call GetValue(Result(2)).
4. Call PutValue(Result(1), Result(3)).
5. Return.

### 8.2 **EXPRESSIONSTATEMENT**

**Syntax**

```
ExpressionStatement:
   Expressionopt ;
```

**Semantics**
The production **ExpressionStatement**: `Expression ;` is evaluated as follows:
1. Evaluate **Expression**.
2. Call GetValue(Result(1)).

### 8.3 **THE if STATEMENT**

**Syntax**

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

**Semantics**
The production **IfStatement**: `if ( Expression ) Statement else Statement` is evaluated as follows:
1. Evaluate **Expression**.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **false**, go to step 7.
5. Evaluate **Statement**.
6. Return.
7. Evaluate **Statement**.
8. Return.

The production **IfStatement**: `if ( Expression ) Statement` is evaluated as follows:
1. Evaluate **Expression**.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is false, return.
5. Evaluate Statement.
6. Return.

8.4 ITERATION STATEMENTS

Syntax

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

for (varopt Identifier in Expression) Statement

Description

These statements all define a “continue label” and a “break label” for use by an enclosing continue or break statement. For the purposes of this specification, a label is a step number in an algorithm. Continue labels are held in a continue label stack and break labels are held in a break label stack. These stacks are local to the current scope. To execute continue or break statement, execution control is transferred to the label specified by the top value of the corresponding label stack. If an implementation of ECMAScript has distinct compile and execute phases, the label stacks need only be maintained during compilation as the label that continue or break statement jumps to is not dependent on any runtime state.

The WithStatement affects both stacks for the purposes of clean up: to remove its object from the scope chain.

In algorithms, we use “PushBreak(n)” as short hand for “Push Step(n) on the break label stack”. Similarly we use “PushContinue(n)”, “PopBreak(n)” and “PopContinue(n)” as short hand for the obvious phrases. We use “JumpBreak” as short hand for “Transfer execution control to the position indicated by the top label of the break label stack” and similarly for “JumpContinue”.

8.4.1 The while Statement

The production IterationStatement: while (Expression) Statement is evaluated as follows:
1. PushContinue(3).
2. PushBreak(9).
3. Evaluate Expression.
4. Call GetValue(Result(3)).
5. Call ToBoolean(Result(4)).
6. If Result(5) is false, go to 9.
7. Evaluate Statement.
8. Go to step 2.
9. PopBreak(9).
10. PopContinue(3).
11. Return.

8.4.2 The for Statement

The production IterationStatement: for (Expressionopt ; Expressionopt ; Expressionopt) Statement is evaluated as follows:
1. PushContinue(10).
2. PushBreak(13).
3. Evaluate Expression₁.
4. Call GetValue(Result(3)).
5. Evaluate Expression₂.
6. Call GetValue(Result(5)).
7. Call ToBoolean(Result(6)).
8. If Result(7) is \textbf{false}, go to step 13.
10. Evaluate Expression₃.
11. Call GetValue(Result(10)).
12. Go to step 5.
13. PopBreak(13).
14. PopContinue(10).
15. Return.

If Expression₁ is omitted from the source text, steps 3 and 4 are omitted from execution. If Expression₂ is omitted from the source text, step 5 is omitted from execution and the result of step 5 is \textbf{true}. If Expression₃ is omitted from the source text, steps 10 and 11 are omitted from execution.

**Issue:** define the var version.

### 8.4.3 The \texttt{for..in} Statement

The production $\text{IterationStatement: } \texttt{for} \ (\text{Expression}_1 \ \text{in} \ \text{Expression}_2) \ \text{Statement}$ is evaluated as follows:

1. PushContinue(6).
2. PushBreak(11).
3. Evaluate Expression₂.
4. Call GetValue(Result(3)).
5. Call ToObject(Result(4)).
6. Get the name of the next property of Result(5) which doesn’t have the DontEnum attribute. If there is no such property, go to step 11.
7. Evaluate Expression₁.
8. Call PutValue(Result(7), Result(6)).
11. PopBreak(11).
12. PopContinue(6).
13. Return.

The mechanics of enumerating the properties (step 6) is implementation dependent. The order of enumeration is defined by the object. Properties of the object being enumerated may be dropped during enumeration. If new properties are added to the object enumerated during enumeration are not guaranteed to be visited in the active enumeration.

**Issue:** should we restrict Expression₁?

**Issue:** define the var version.

### 8.5 \texttt{CONTROLFLOWSTATEMENTS}

**Syntax**

\[
\text{ControlFlowStatement:} \quad \begin{align*}
\text{continue; } \\
\text{break; } \\
\text{return } \text{Expression}_{opt}; 
\end{align*}
\]
8.5.1 The continue Statement
The continue statement can only be used when the continue label stack contains at least one label. This is only the case inside a while, for, or for..in loop. The continue statement is evaluated as:
1. JumpContinue.

See section 8.4 Iteration Statements for a description of the continue label stack and the JumpContinue directive.

8.5.2 The break Statement
The break statement can only be used when the break label stack contains at least one label. This is only the case inside a while, for or for..in loop. The break statement is evaluated as:
1. JumpBreak

See section 8.4 Iteration Statements for a description of the break label stack and the JumpBreak directive.

8.5.3 The return Statement
The return statement can only be used inside the block of a FunctionDeclaration. It causes a function to cease execution and 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.

Issue: The Current Microsoft anf Netscape implementations generate an error if one return statement in a function returns a value and another return in the same function doesn’t return a value. The current Borland implementation allows this. Which behavior do we want.

Issue: Current implementations treat

```
    return
    a + b
```

as

```
    return;
    a+b;
```

instead of as

```
    return a+b;
```

Do we want this behavior, and if so, how do we describe it in the spec?

8.7 THE with STATEMENT
Syntax
WithStatement:
```
    with ( Expression ) Statement
```

Description
The WithStatement affects the break label stack and continue label stack for clean up purposes only.

Semantics
The production WithStatement: with ( Expression ) Statement is evaluated as follows:
1. If the continue label stack is not empty, PushContinue(12).
2. If the break label stack is not empty, PushBreak(16).
3. Evaluate Expression.
4. Call GetValue(Result(3)).
5. Call ToObject(Result(4)).
6. Add Result(5) to the front of the scope chain.
7. Evaluate Statement.
8. Remove Result(5) from the front of the scope chain.
9. If the break label stack is not empty, PopBreak(16).
10. If the continue label stack is not empty, PopContinue(12).
11. Return.
12. Remove Result(5) from the front of the scope chain.
13. If the break label stack is not empty, PopBreak(16).
14. PopContinue(12).
15. JumpContinue.
16. Remove Result(5) from the front of the scope chain.
17. PopBreak(16).
18. If the continue label stack is not empty, PopContinue(12).

Discussion
Most of the complexity of this algorithm is to handle jumps out of the WithStatement. Any jumps out of the WithStatement must be trapped to remove the object from the scope chain.
CHAPTER 9

FUNCTION DEFINITION

Syntax

FunctionDeclaration:
    function Identifier ( FormalParameterListopt ) Block

FormalParameterList:
    Identifier
    FormalParameterList, Identifier

Semantics
Defines a property of the global object whose name is the Identifier and whose value is a function object with the given parameter list and statements.
CHAPTER 10

PROGRAM

Syntax

Program:
  SourceElements EndOfSource

SourceElements:
  SourceElement
  SourceElements SourceElement

SourceElement:
  Statement
  FunctionDefinition
  ClassDefinition
CHAPTER 11

NATIVE ECMAScript Objects

11.1 Web Browser Hosted Objects

11.2 HTTP Server Hosted Objects
REFERENCES

ANSI X3.159-1989: *American National Standard for Information Systems - Programming Language - C*


APPENDIX A

OPEN ISSUES

A.1 && and || Semantics
What is the value and type of (true && 10)? What is the value and type of (false || 10). Do we follow C++/Java semantics or Perl semantics?

A.2 Eval function
Define object scoping within Eval block.

A.3 Host Supplied members of scope chains vs. Implicit this.

A.4 Lifetime of Activation Record Object (has scope chain)

A.5 Should arguments object include local variables.
APPENDIX B

PROPOSED EXTENSIONS

B.1 THE CLASS STATEMENT

Class Definition

Syntax

ClassDeclaration:
  class Identifier ( FormalParameterList_{opt} ) ExtendsClause_{opt} { ClassBody }

FormalParameterList:
  Identifier
  FormalParameterList, Identifier

ExtendsClause:
  extends Identifier( ExpressionList_{opt} )

ClassBody:
  Constructor_{opt} Methods_{opt}

Constructor:
  StatementList

Methods:
  FunctionDefinition
  Methods FunctionDefinition

Semantics
Similar to a function except:
• The class name space is global but distinct from the global function name space.
• The functions (methods) defined within a class definition are in a name space private to the class.
• The inclusion of methods automatically creates one property in the constructed object for each method defined.
• Classes may not be called directly but rather can only be used via the new operator.

B.2 THE TRY AND THROW STATEMENTS

B.2.1 THE TRY STATEMENT

A try statement executes a block. If a value is thrown and the try statement has one or more catch clauses that can catch it, then control will be transferred to the first such catch clause. If the try statement has a finally clause, then the finally block of code is executed no matter whether the try block completes normally or abruptly and regardless of whether a catch clause is first given control.

TryStatement :
  try Block Catches
try Block Catchesopt FinallyClause

Catches:
  CatchClause
Catches CatchClause

CatchClause:
  catch ( FormalParameter ) Statement

FinallyClause:
  finally Statement

B.2.2 The Throw Statement

A throw statement causes an exception to be thrown. The result is an immediate transfer of control that may exit multiple statements and method invocations until a try statement is found that catches the thrown value. If no such try statement is found, then a runtime error is generated.

ThrowStatement:
  throw Expression

B.3 The Date Type

The Date Type is used to represent date and time. It is a Julian value on which certain operations such as date arithmetic are defined. Arithmetic operators, relational operators and equality operators apply to this type.

Note 1: Of the three current ECMAScript implementations, only the Borland implementation currently supports date operators. This feature is really just a convenience that can be implemented with Date Object methods. However, the same argument can be made for the String type.

Note 2: Of the three current ECMAScript implementations, only the Borland implementation currently implements dates as Julian dates and thus dates before (January 1970). Without this representation, dates are very limited in their usage (i.e. you cannot otherwise, represent arbitrary dates, for example from existing databases)

B.3.1 ToDate

The operator ToDate attempts to convert its argument to a value of subtype Date Object according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Blank date value.</td>
</tr>
<tr>
<td>Null</td>
<td>Blank date value.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Blank date value.</td>
</tr>
<tr>
<td>Number</td>
<td>Blank date value.</td>
</tr>
<tr>
<td>String</td>
<td>See discussion below.</td>
</tr>
<tr>
<td>Date</td>
<td>Return the input argument (no conversion)</td>
</tr>
<tr>
<td>Object</td>
<td>Apply the following steps:</td>
</tr>
<tr>
<td></td>
<td>1. Call ToPrimitive on the input argument.</td>
</tr>
<tr>
<td></td>
<td>2. Call ToDate(Result(1)).</td>
</tr>
<tr>
<td></td>
<td>Return Result(2).</td>
</tr>
</tbody>
</table>

B.3.1.1 ToDate Applied to the String Type
**Issue:** define this.

### B.4 ImplicitThis

In function code where the function definition specifies the `implicit` keyword, the `this` object is placed in the scope chain immediately before the global object.

### B.5 The switch Statement

#### Syntax

```
SwitchStatement:
  switch ( Expression ) CaseBlock

CaseBlock:
  { CaseClauses opt }
  { CaseClauses opt DefaultClause CaseClauses opt }

CaseClauses:
  CaseClause
  CaseClauses CaseClause

CaseClause:
  case Expression : StatementList opt

DefaultClause:
  default : StatementList opt
```

#### Semantics

The `SwitchStatement` adds a label to the break label stack, which is described in section 8.4 Iteration Statements. It also adds a label to the continue label stack for clean up purposes only.

The production `SwitchStatement: switch ( Expression ) CaseBlock` is evaluated as follows:

1. If the continue label stack is not empty, PushContinue(9).
2. PushBreak(6).
3. Evaluate `Expression`.
4. Call GetValue(Result(3)).
5. Evaluate `CaseBlock`, passing it Result(4) as a parameter.
6. PopBreak(6).
7. If the continue label stack is not empty, PopContinue(9).
8. Return.
10. PopContinue(9).
11. JumpContinue.

The production `CaseBlock: { CaseClauses opt DefaultClause CaseClauses opt }` is given an input parameter, `input`, and is evaluated as follows:

1. For the next `CaseClause` in `CaseClauses`, in source text order, evaluate `CaseClause`. If there is no such `CaseClause`, go to step 6.
2. If `input` is not equal to Result(1) (as defined by the `!=` operator), go to step 1.
3. Execute the `StatementList` of this `CaseClause`.
4. Execute the `StatementList` of each subsequent `CaseClause` in `CaseClauses`.
5. Go to step 11.
6. For the next CaseClause in CaseClauses, in source text order, evaluate CaseClause. If there is no such CaseClause, go to step 11.
7. If input is not equal to Result(6) (as defined by the != operator), go to step 6.
8. Execute the StatementList of this CaseClause.
9. Execute the StatementList of each subsequent CaseClause in CaseClauses.
10. Return.
11. Execute the StatementList of DefaultClause.
12. Execute the StatementList of each CaseClause in CaseClauses.
13. Return.

If CaseClauses is omitted, steps 1 through 5 are omitted from execution. If DefaultClause is omitted (in which case CaseClauses is also omitted), steps 11 and 12 are omitted from execution. If CaseClauses is omitted, steps 6 through 10 and 12 are omitted from execution.

Typically there will be a break statement in one or more StatementList, which will transfer execution back to the break label for the SwitchStatement.

The production CaseClause: case Expression : StatementListopt is evaluated as follows:
1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Return Result(2).

Note that evaluating CaseClause does not execute the associated StatementList. It simply evaluates the Expression and returns the value, which the CaseBlock algorithm uses to determine which StatementList to start executing.

B.6 CONVERSION FUNCTIONS
The conversion functions, ToBoolean, ToNumber, ToInteger, ToInt32, ToUint32, ToString, and ToObject are global functions that operate as described in this document.

B.7 ASSIGNMENT-ONLY OPERATOR ( := )

The assignment-only operator operates identically to the assignment operator (=) except that if the given lvalue doesn’t already exist, prior to the statements execution, a runtime error is generated.

B.8 SEALING OF AN OBJECT
A facility to prevent an object from being further expanded may be invoked at any time after an object has been constructed. This is semantically the dynamic equivalent to the static Java final class modifier. This facility may be implemented as a method of the object, a global function, or, if the class statement is adopted, as a class modifier to class. Once an object has been sealed or finalized, any attempt to add a new property to the object results in a runtime error.

B.9 THE arguments KEYWORD

The arguments keyword refers to the arguments object. Within global code, arguments returns null. Within eval code, arguments returns the same value as in the calling context.

Discussion:
This interpretation of the "arguments" within a function body differs from existing practice but has two important advantages over 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.
   It solves scope resolution issues related to using arguments within a with block on an object that has an arguments member, such as Math.
APPENDIX C

PEOPLE CONTACTS
Brendan Eich (brendan@netscape.com)
C. Rand McKinney (rand@netscape.com)
Donna Converse (converse@netscape.com)
Randy T. Solton (rsolton@wpo.borland.com)
Mike Gardner (mgardner@wpo.borland.com)
Shon Katzenberger (shonk@microsoft.com)
Robert Welland (robwell@microsoft.com)
INDEX

-- 27
-  subtraction .................................. 31
!
!  Logical NOT ................................. 29
&
&  bitwise AND ................................. 34
&&  logical AND ................................. 34
, ,  comma operator ............................. 36
? ?  conditional expression ................. 35
~
~  Bitwise NOT ................................. 29
+
++ ............................................. 27
<
<<
  left shift .................................... 31
=
=  assignment ................................... 36
>
>>  right shift ................................. 32
>>>  unsigned right shift ..................... 32
A
arguments ...................................... 25
arrays .......................................... 26
B
break ............................................ 41
C
control flow .................................... 40
E
expression
  primary ....................................... 25
I
if 38
iteration ....................................... 39
O
op=
  compound assignment ...................... 36
operators
  additive, semantics ....................... 30
  equality ...................................... 33
  postfix ...................................... 25
  relational .................................. 32
  unary ....................................... 28
R
return .......................................... 41
S
shift ............................................. 31
source text .................................... 44, 45
statements ..................................... 37
  expression .................................. 37, 38
U
Unicode ......................................... 7
V
void ............................................ 28
W
while ............................................. 39
White Space .................................. 14
with .............................................. 41, 49, 50