APPENDIX B: PROPOSED EXTENSIONS

5B.1 THE CLASS STATEMENT¹

Syntax

```
ClassDeclaration :
   class IdentifierFormalParameters opt ExtendsClause opt { ClassBody }
```

```
FormalParameters :
   ( FormalParameterList opt )
```

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

```
ExtendsClause :
   extends Identifier ActualArguments opt
```

```
ActualArguments :
   ( 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.

6B.2 THE TRY AND THROW STATEMENTS¹

0B.2.1 The try Statement¹

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 ) Block

FinallyClause:
    finally Block

1B.2.2 The Throw Statment

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

7 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)

2B.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(input argument, hint Date).</td>
</tr>
<tr>
<td></td>
<td>2. Call ToDate(Result(1)).</td>
</tr>
<tr>
<td></td>
<td>Return Result(2).</td>
</tr>
</tbody>
</table>

3B.3.2 ToDate Applied to the String Type

Issue: define this.
8B.4 Implicit This

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.

9B.5 The switch Statement

Syntax

\[
\text{SwitchStatement} : \quad \text{switch} \ ( \text{Expression} \ ) \ \text{CaseBlock}
\]

\[
\text{CaseBlock} : \quad \{ \ \text{CaseClauses}_{\text{opt}} \ \}
\]

\[
\text{CaseClauses} : \quad \text{CaseClause} \ \text{CaseClauses} \ \text{CaseClause}_{\text{opt}}
\]

\[
\text{CaseClause} : \quad \text{case} \ \text{Expression} : \ \text{StatementList}_{\text{opt}}
\]

\[
\text{DefaultClause} : \quad \text{default} : \ \text{StatementList}_{\text{opt}}
\]

Semantics

The `SwitchStatement` adds a label to the break label stack, which is described in section 12.5. 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:

902. If the continue label stack is not empty, `PushContinue(9)`.
903. `PushBreak(6)`.
904. Evaluate `Expression`.
905. Call `GetValue(Result(3))`.
906. Evaluate `CaseBlock`, passing it `Result(4)` as a parameter.
907. `PopBreak(6)`.
908. If the continue label stack is not empty, `PopContinue(9)`.
909. Return.
910. `PopBreak(6)`.
911. `PopContinue(9)`.
912. `JumpContinue`.

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

913. For the next `CaseClause` in `CaseClauses1`, in source text order, evaluate `CaseClause`. If there is no such `CaseClause`, go to step 6.
914. If `input` is not equal to `Result(1)` (as defined by the `!=` operator), go to step 1.
915. Execute the `StatementList` of this `CaseClause`.
916. Execute the `StatementList` of each subsequent `CaseClause` in `CaseClauses1`.
917. Go to step 11.
918. For the next `CaseClause` in `CaseClauses2`, in source text order, evaluate `CaseClause`. If there is no such `CaseClause`, go to step 11.
919. If `input` is not equal to `Result(6)` (as defined by the `!=` operator), go to step 6.
920. Execute the `StatementList` of this `CaseClause`.
921. Execute the `StatementList` of each subsequent `CaseClause` in `CaseClauses2`.
922. Return.
923. Execute the `StatementList` of `DefaultClause`.
924. Execute the `StatementList` of each `CaseClause` in `CaseClauses2`. 
925. Return.

If CaseClauses1 is omitted, steps 1 through 5 are omitted from execution. If DefaultClause is omitted (in which case CaseClauses2 is also omitted), steps 11 and 12 are omitted from execution. If CaseClauses2 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 : StatementList_{opt} is evaluated as follows:

926. Evaluate Expression.
927. Call GetValue(Result(1)).
928. 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.

10 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.

11 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.

12 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.

13 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.

14 B.10 PREPROCESSOR

15 B.11 THE DO..WHILE STATEMENT

16 B.12 BINARY OBJECT

17 B.13 LABELS WITH BREAK AND CONTINUE

As in Java, allow statements to be labeled with an identifier followed by a colon. Allow a label to appear in a break or continue statement. The label referred to by a break or continue statement must be an iteration statement that contains the break or continue statement in question.
The use of labels makes code more readable and more robust. In addition, it makes possible certain transfers of control that otherwise could not be easily expressed at all.