```
//This a a definitional interpreter for ECMAScript 5
//implemented using ECMAScript 3

//Copyright 2009, Ecma International

//This a an early and incomplete versions of the ES5 definitional interpreter
//Currently it only implements significant portions of sections 8 and 9.


function ES5() {
// A "Module" function that encapuslates the definitional interpreter

// Terminology:
// "ECMAScript"  is used to refer to the language that is specified by
// this definitional interpreter
// "JavaScript" is used to refer to the langage in which this definitional
// interpreter is written.
//
// Conventions:
// Identifers that are prefix with "impl" or "Impl" are artifacts of this
// implementation and not part of the ECMAScript specification. Arguments
// and local variables of implementation specific functions are not prefixed
// because the entire function is prefixed. Names such as toString that have
// specific JavaScript meanings can't have "impl" prefixes even though they
// really should be.


//implementation convenience  functions
function implCreate(proto) {
    // In ES5 we would us Object.create directly
    var creator = function() {};
    creator.prototype=proto;
    return new creator;
    };

function implExtend(obj,template) {
    // add template properties to an object
    for (var p in template) obj[p] = template[p];
    return obj;
    }

// ESassert is called to test both implicit and explicit assertions
// contained in the ECMAScript specification
function ESassert(predicate, msg) {
   if (predicate) return;
   throw new Error("ECMAScript internal assertion failed"+(msg?msg:""));
   }

function implTODO(str, Continue) {
    if (!Continue) ESassert(false, str);
   }


// Each section of the specification has a corresponding JavaScript object
// whose properties are the abstract operations defined by that section of
// the specification

///// Section 4
var S4 = {
   isPrimitiveValue: function (val) {   //4.3.2
      // The definition of this abstraction operation should probaby move
      // into section 8.
      var type = val.type;
      return type===ECMAScriptType.Undefined || type===ECMAScriptType.Null ||
         type===ECMAScriptType.Boolean || type===ECMAScriptType.String ||
         type===ECMAScriptType.Number;
      }
   };
```

```
///// Section 5
function ImplInternalExceptionRecord(name,desc) {
   this.name=name;
   this.message = desc;
   return this;
   }
ImplInternalExceptionRecord.prototype=Error.prototype;


var S5 = {
   throwException: function (implName,implDesc) {      //5.2 - last paragrpah
      var implExcpt = new ImplInternalExceptionRecord(implName, implDesc);
      throw implExcpt;
      }
   }


///// Section 8

//The ECMAScript specification defines the following types.
//Specification types are only used internally as part of the specification
//and are not directly observable to ECMAScript programs.
var ECMAScriptType = {
   Undefined: {typeName: "Undefined", specificationType: false},    //8.1
   Null: {typeName: "Null", specificationType: false},              //8.2
   Boolean: {typeName: "Boolean", specificationType: false},        //8.3
   String: {typeName: "String", specificationType: false},          //8.4
   Number: {typeName: "Number", specificationType: false},          //8.5
   Object: {typeName: "Object", specificationType: false},          //8.6
   Reference: {typeName: "Reference", specificationType: true},     //8.7
   List: {typeName: "List", specificationType: true},               //8.8
   Completion: {typeName: "Completion", specificationType: true},   //8.9
   PropertyDescriptor:
      {typeName: "PropertyDescriptor", specificationType: true},  //8.10
   PropertyIdentifier:
      {typeName: "PropteryIdentifier", specificationType: true},  //8.10
   LexicalEnvironment:
      {typeName: "LexicalEnvironment", specificationType: true},  //8.11 & 10.2
   EnvironmentRecord:
      {typeName: "EnvironmentRecord", specificationType: true}    //8.11 & 10.2.1
   };

//The algorithms of the ECMAScript specification operates upon ECMAScript Values.
//Each ECMAScript value is an instance of one of the ECMAScript types.

//Javascript constructor for creating individual ECMAScrpt values.
function ECMAScriptValue(EStype,rest) {
     //First argument is the ECMAScriptType of the value.
     //The rest of the arguments are dependent upon the ECMAScriptType
     //and are used to initialize the value
     this.type = EStype;
     return EStype.implInit.apply(EStype,[].concat(this,[].slice.call(arguments,1)));
     return this;
     }
ECMAScriptValue.prototype= {
    type: undefined,       // the ECMAScriptType of this ECMAScriptValue
    toString: function () {// for debugging. should really have an impl prefix but
can't
        var s = '[ESValue '+this.type.typeName;
         s +=JSON.stringify(this,null,(typeof this==='object')*3);
         return s+']';
         }
    }

/*------ Start of initializers definitions for various kinds of ECMAScriptValue -----
-*/
```

```
//Methods used to initialize ECMAScriptValue instance for the various
ECMAScriptTypes.
//Called from the ECMAScriptValue constructor

// these types require no specification instantiation actions
ECMAScriptType.Undefined.implInit=
ECMAScriptType.Null.implInit=
    function implInit(obj,args) {
        //default implementation does nothing
        return obj
        }


//8.3 Boolean
// Boolean values have no specified state, but an implementaion-specific
// field that distingishes instances is present for debugging.
// The constructor should only be called twice: to create the to Boolean
// values, true and false, which are then accessed using ESV.
// The second argument to the constructor is either a JavaScript true or false value.
ECMAScriptType.Boolean.implInit = function (bool,value) {
    bool.implValue = value;  //used for debugging output
    return bool;
    }

//8.4 String
// String values have a list of 16-bit integers and the length of the list
// as their internal state.
// The second argument to the constructor is a JavaScript string that provides
// the list of integer values.
ECMAScriptType.String.implInit = function (str, text) {
    str.characters = text;
    str.stringLength = text.length;
    return str;
    }

//8.5 Number
// Number values have a IEEE 64-bit binary floating point number as their
// internal state.
// The second argument to the constructor is a JavaScript Number value which
// provides the number  value.
ECMAScriptType.Number.implInit = function (num,value) {
    num.IEEEValue = value;
    return num;
    }

//8.6 Object
// Objects values have internal properties and methods and a list of named
// properties as their internal state.
// The constructor for an ECMAScript.Object value does not take a second argument.
//
// The ES5 spec. doesn't specify how to model own properties so we use a List of
// PropertyIdentifiers. This should become part of the spec.
ECMAScriptType.Object.implInit = function (obj) {
    implExtend(obj,objectCommonInternalProperties);
    obj['[[implProperties]]'] = new ECMAScriptValue(ECMAScriptType.List);
    return obj;
    }

//8.7
// Reference values have three fields: base, referencedName, and strictReferece
// as their internal state.
// The second, third, and fourth arguments to the constructor provide the values
// of these fields.
ECMAScriptType.Reference.implInit = function(ref, base, referencedName,
strictReference){
    ref.base = base;
    ref.referencedName = referencedName;
    ref.strictReference = strictReference;
```

```
    return ref;
      }

//8.8
ECMAScriptType.List.implInit = function (list, element1) {
   list.implElements = Array.prototype.slice.call(arguments,1);
   list.implFindIndx = function(tester) {
      for (var indx in this.implElements) {
          if (tester(this.implElements[indx])) return indx ;
          }
      return undefined;
      }
   list.implFind = function(tester) {
      for (var indx in this.implElements) {
          var elem=this.implElements[indx];
          if (tester(elem)) return elem;
          }
      return undefined;
      }
   list.implAppend = function(element) {
      this.implElements.push(element);
      };
   return list;
   }

//8.9
// Completion values have three fields: type, value, and target as their internal
state.
// The second, third, and fourth arguments to the constructor provide the values of
// these fields.
// The property used to model the "type" field is named "kind" to avoid a name
conflict
// with the generic ECMAScriptValue type property. It would probably be a good idea
for
// the prose specification to also adopt this naming change.
ECMAScriptType.Completion.implInit = function (completion, type, value, target) {
     completion.kind = type;
     completion.value = value;
     completion.target = target;
     completion.isAbruptCompletion =
         function () {return this.kind!==ECMAScriptType.Completion.normal};
     return completion;
     }
//Allowed values for the Completion kind (type) field.
ECMAScriptType.Completion.normal="normal";
ECMAScriptType.Completion["break"]="break";
ECMAScriptType.Completion["continue"]="continue";
ECMAScriptType.Completion["return"]="return";
ECMAScriptType.Completion["throw"]="throw";

//8.10 Property Descriptor
var implDescriptorFieldList =  // the valid field names for property descriptors

['[[Value]]','[[Writable]]','[[Get]]','[[Set]]','[[Enumerable]]','[[Configurable]]'];
function implValidatePropertyAttributes(desc) {
    // Return false if the argument object contains an invalid combination of
property
    // descriptor fields
    implTODO("body of implValidatePropertyAttributes");
    return true;
    }

ECMAScriptType.PropertyDescriptor.implInit=function  (pd,attrs) {
   // pd is an ECMAScriptValue that is being initialized as a Property Descrptor
   // attrs is an object whose properties are property descriptor fields,
   //      the values of these properties should be ECMAScriptValue objects rather
than
```

```
   //      corresponding JavaScript values of the underlying JavaScript
implementation
   ESassert(implValidatePropertyAttributes(attrs));
   for (var f in implDescriptorFieldList) {
      var p = implDescriptorFieldList[f];
      if (p in attrs) pd[p]=attrs[p];
      }

   /* method sets fields of aProperty Descriptor ECMAScript value to their defaults*/
   pd.implPopulateDefaults = function () {  //8.6.1 Table 3 and 8.10
      ESassert(S8.IsGenericDescriptor(this) === ESV("false"));
      if (S8.IsDataDescriptor(this)=== ESV("true")) {
         if (!this.implHasField('[[Value]]')) this['[[Value]]'] = ESV['undefined'];
         if (!this.implHasField('[[Writable]]')) this['[[Writable]]'] = ESV['false'];
         }
      if (S8.IsAccessorDescriptor(this)=== ESV("true")) {
         if (!this.implHasField('[[Get]]')) this['[[Get]]'] = ESV['undefined'];
         if (!this.implHasField('[[Set]]')) this['[[Set]]'] = ESV['undefined'];
         }
      if (!this.implHasField('[[Enumerable]]')) this['[[Enumerable]]'] =
ESV['false'];
      if (!this.implHasField('[[Configurable]]')) this['[[Configurable]]'] =
ESV['false'];
      return this;
      };

   /* method tests if aProperty Descriptor ECMAScript value has a specific field*/
   pd.implHasField = function (fieldName) {
      return this[fieldName] !== undefined;
      }

   /* method returns an Array of all field names of a Property Descriptor ECMAScript
value */
   pd.implFieldList = function () {
       var fields = [];
      for (var f in implDescriptorFieldList) {
         if (implDescriptorFieldList[f] in this)
fields.push(implDescriptorFieldList[f]);
         }
      return fields;
      }

   /* method for cloning a Property Descriptor ECMAScript value */
   pd.implClone() = function () {
      return new ECMAScriptValue(ECMAScriptType.PropertyDescriptor,this);
      }

   return pd;
   }

//8.10 Property Identifier
ECMAScriptType.PropertyIdentifier.implInit=function (pi,name, descriptor) {
    //pi is an ECMAScriptValue that is being initialized as a Property Identifier
    //name is an ECMAScrpt.String value that is the name of the property
    //descriptor is an ECMAScript.PropertyDescript value
    pi.name = name;
    pi.descriptor = descriptor;
    return pi;
    }

//8.11 Lexical Environment
ECMAScriptType.LexicalEnvironment.implInit=
   function implInit(obj,args) {
      implTODO("ECMAScriptType.LexicalEnvironment.implInit needs implementation");
      return obj
      }

//8.11 Environment Record
```

```
ECMAScriptType.EnvironmentRecord.implInit=
   function implInit(obj,args) {
      implTODO("ECMAScriptType.EnvironmentRecord.implInit needs implementation");
      return obj
      }
/*------ end of initializers definitions for various kinds of ECMAScriptValue ------
*/



// Symbolic names for named ECMAScript Values.
// ESV is an implementation artifact used to access these symbolic names in a manner
// that does not conflict with JavaScript's usage of some of the same names.
var ESV = {
   "undefined": new ECMAScriptValue(ECMAScriptType.Undefined),    //8.1
   "null": new ECMAScriptValue(ECMAScriptType.Null),               //8.2
   "true": new ECMAScriptValue(ECMAScriptType.Boolean, true),           //8.3
   "false": new ECMAScriptValue(ECMAScriptType.Boolean, false),   //8.3
   "NaN": new ECMAScriptValue(ECMAScriptType.Number,NaN),             //8.5
   emptyString: new ECMAScriptValue(ECMAScriptType.String,""),
   hint: {
      String:  'ESV.hint.String',
      Number:  'ESV.hint.Number'
      },
   empty:  'ESV.empty',     //8.9
   emptyList: new ECMAScriptValue(ECMAScriptType.List)
   };
// impl -- toJSON is used for debugging displays of ESV values
for (var n in ESV) (function(n){ESV[n].toJSON=function() {return
"ESV['"+n+"']"}})(n);
delete ESV.hint.toJSON; //hint is an interior node only used for hierarchical naming

// Symbolic names for some ECMAScript String values that are widely used literally
// in the ECMAScript specification
var ESS = {
   "value": new ECMAScriptValue(ECMAScriptType.String,"value"),
   "writable": new ECMAScriptValue(ECMAScriptType.String,"writable"),
   "put": new ECMAScriptValue(ECMAScriptType.String,"put"),
   "get": new ECMAScriptValue(ECMAScriptType.String,"get"),
   "enumerable": new ECMAScriptValue(ECMAScriptType.String,"get"),
   "configurable": new ECMAScriptValue(ECMAScriptType.String,"configurable"),
   "prototype": new ECMAScriptValue(ECMAScriptType.String,"prototype")
   }


//8.6.2 Common Object Internal Properties and Methods

var objectCommonInternalProperties = {
   '[[Prototype]]':      undefined,
   '[[Class]]':          undefined,
   '[[Extensible]]':     ESV["false"],
   '[[implProperties]]': undefined
         //the spec. does specify how to model own properties
         //so we use a List of PropertyIdentifiers
    };

var objectCommonInternalMethods = {
   '[[GetOwnProperty]]': function (P) {             //8.12.1
      var O = this;
      var properties = O['[[implProperties]]'];
      var X = properties.implFind(
         function (pi) {return S9.SameValue(pi.name,P)===ESV["true"]});
      if (X === undefined) return  ESV["undefined"];
      var D = X.descriptor.implClone();
      return D;
      },

   '[[GetProperty]]': function (P) {          //8.12.2
```

```
   var O = this;
   var prop = O['[[GetOwnProperty]]'](P);
   if (prop !== ESV["undefined"]) return prop;
   var proto = O['[P[rototype]]'];
   if (proto === ESV["null"]) return ESV["undefined"];
   return proto['[[GetProperty]]'](P);
   },

'[[Get]]': function (P) {                  //8.12.3
   var O = this;
   var getter;
   var desc=O['[[GetProperty]]'](P);
   if (desc === ESV["undefined"]) return ESV["undefined"];
   if (S8.IsDataDescriptor(desc) === ESV["true"]) return desc['[[Value]]'];
   ESasset(S8.IsAccessorDescriptor(desc) === ESV["true"] );
   getter = desc['[[Get]]'];
   if (getter === ESV["undefined"]) return ESV["undefined"];
   return getter['[[Call]]'](O, ESV.emptyList);
   },

'[[CanPut]]': function (P) {               //8.12.4
   var O = this;
   var desc=O['[[GetOwnProperty]]'](P);
   if (desc !== ESV["undefined"]) {
      if (S8.IsAccessorDescriptor(desc) === ESV["true"]) {
         if (desc['[[Set]]'] === ESV["undefined"]) return ESV["false"];
         else return ESV["true"];
         }
      else {
         ESassert(S8.isDataDescriptor(desc) === ESV["true"] );
         return desc['[[Writable]]'];
         }
      }
   var proto = O['[[Prototype]]'];
   if (proto === ESV["null"]) return O['[Extensible]'];
   var inherited = proto['[[GetProperty]]'](P);
   if (inherited === ESV["undefined"]) return O['[[Extensible]]'];
   if (S8.IsAccessorDescriptor((inherited) === ESV["true"])) {
      if (inherited['[[Set]]'] === ESV["undefined"]) return ESV["false"];
      else return ESV["true"];
      }
   ESassert(S8.isDataDescriptor(inherited) === ESV["true"] );
   if (O['[[Extensible]]'] === ESV["false"]) return ESV["false"];
   else return inherited['[[Writable]]'];
   },

'[[Put]]': function (P, V, Throw) {  //8.12.5
   var O = this;
   var valueDesc, setter, newDesc;
   if (O['[[CanPut]]'](P) === ESV["false"]) {
      if (Throw === ESV["true"])
         S5.throwException("TypeError", "Can't put property "+P);
      else return;
      }
   var ownDesc = O['[[GetOwnProperty]]'](P);
   if (S8.IsDataDescriptor(ownDesc) === ESV["true"]) {
      valueDesc =
         new ECMAScriptValue(ECMAScriptType.PropertyDescriptor,{"[[Value]]": V});
      O['[[DefineOwnProperty]]'](P, valueDesc, Throw);
      return;
      }
   var desc = O['[[GetProperty]]'](P);
   if (S8.IsAccessorDescriptor(desc) === ESV["true"]) {
      setter=desc['[[Set]]'];
      ESassert(setter !== ESV["undefined"]);
      setter['[[Call]]'](O, new ECMAScriptValue(ECMAScriptType.List,V));
      }
   else {
```

```
      newDesc =
         new ECMAScriptValue(ECMAScriptType.PropertyDescriptor,
                {"[[Value]]": V, "[[Writable]]": ESV["true"],
                 "[[Enumerable]]":ESV["true"], "[[Configurable]]": ESV["true"]
                });
      O['[[DefineOwnProperty]]'](P, neweDesc, Throw);
      }
   return;
   },

'[[HasProperty]]': function (P) {                //8.12.6
   var O = this;
   var desc = O['[[GetProperty]]'](P);
   if (desc === ESV["undefined"]) return ESV["false"];
   return ESV["true"];
   },

'[[Delete]]': function (P, Throw) {              //8.12.7
   var O = this;
   var implProperties, implIndx;
   var desc=O['[[GetOwnProperty]]'](P);
   if (desc === ESV["undefined"]) return ESV["true"];
   if (desc['[[Configurable]]'] === ESV["true"]) {
      implProperties = O['[[implProperties]]'];
      implIndx = implProperties.implFindIndx(
         function (pi) {return S9.SameValue(pi.name,P)===ESV["true"]});
      ESassert (implIndx!==undefined);
      implProperties.implElements.splice(implIndx,1);
      return ESV["true"];
      }
   else if (Throw === ESV["true"])
      S5.throwException("TypeError", "Can't [[Delete]] property "+P);
   return ESV["false"];
   },

'[[DefaultValue]]': function (hint) {                  //8.12.8
   var O = this;
   if (hint === ESV.hint.String) return StringDefaultValue();
   if (hint === ESV.hint.Number) return NumberDefaultValue();
   ESassert(hint===undefined);
   if (O['[[Class]]'].text==='Date') return StringDefaultValue();
   return NumberDefaultValue();

   function StringDefaultValue() {
      var str;
      var toString =
         O['[[Get]]'](new ECMAScriptValue(ECMAScriptType.String,'toString'));
      if (S9.IsCallable(toString ) === ESV["true"]) {
         str = O['[[Call]]'](O, ESV.emptyList);
         if (S4.isPrimitiveValue(str)) return str;
         }
      var toValue =
         O['[[Get]]'](new ECMAScriptValue(ECMAScriptType.String,'valueOf'));
      if (S9.IsCallable(toValue ) === ESV["true"]) {
         str = O['[[Call]]'](O, ESV.emptyList);
         if (S4.isPrimitiveValue(str)) return str;
         }
      S5.throwException("TypeError", "Can't get [[DefaultValue]]");
      }
   function NumberDefaultValue(){
      var str;
      var toValue =
         O['[[Get]]'](new ECMAScriptValue(ECMAScriptType.String,'valueOf'));
      if (S9.IsCallable(toValue ) === ESV["true"]) {
         str = O['[[Call]]'](O, ESV.emptyList);
         if (S4.isPrimitiveValue(str)) return str;
         }
      var toString =
```

```
            O['[[Get]]'](new ECMAScriptValue(ECMAScriptType.String,'toString'));
         if (S9.IsCallable(toString ) === ESV["true"]) {
            str = O['[[Call]]'](O, ESV.emptyList);
            if (S4.isPrimitiveValue(str)) return str;
            }
         S5.throwException("TypeError", "Can't get [[DefaultValue]]");
         }
      },

   '[[DefineOwnProperty]]': function (P, Desc, Throw) {                  //8.12.9
      function Reject() {
         if (Throw===ESV["true"])
            S5.throwException("TypeError", "Invalid property definition");
         else return false;
         }
      var O = this;
      var implProperties, implPI;
      var current = O['[[GetProperty]]'](P);
      var extensible = O['[[Extensible]]'];
      if (current === ESV["undefined"] && extensible===ESV["false"]) return Reject();
      if (current === ESV["undefined"] && extensible===ESV["true"]) {
         if (S8.IsGenericDescriptor(Desc)===ESV["true"] ||
                S8.IsDataDescriptor(Desc)===ESV["true"]) {
            //step 4.a.i
            implProperties = O['[[implProperties]]'];
            implPI = new ECMAScriptValue(ECMAScriptType.PropertyIdentifier,P,
                           (new
ECMAScriptValue(ECMAScriptType.PropertyDescriptor,Desc)
                            ).implPopulateDefaults());
            implProperties.implAppend(implPI);
            }
         else {
            // step 4.b.i
            ESAssert(S8.IsAccessorDescriptor(Desc)===ESV["true"]);
            implProperties = O['[[implProperties]]'];
            implPI = new ECMAScriptValue(ECMAScriptType.PropertyIdentifier,P,
                           (new
ECMAScriptValue(ECMAScriptType.PropertyDescriptor,Desc)
                            ).implPopulateDefaults());
            implProperties.implAppend(implPI);
            }
         // step 4.c
         return ESV["true"];
         }
      // step 5
      var implDescFields = Desc.implFieldList();
      if (implDescFields.length == 0) return ESV["true"];
      // step 6
      var implDifferences = false;
      for (var implIndx in implDescFields) {
         var implField = implDescFields[implIndx];
         if (!(implField in current) ||
                S9.SameValue(Desc[implField],current[implField])=== ESV["false"] ) {
            implDifferences = true;
            break;
            }
         }
      if (!implDifferences) return ESV["true"];
      //step 7
      if (current['[[Configurable]]'] === ESV["false"]) {
         if (Desc['[[Configurable]]'] === ESV["true"]) return Reject();
         if (current['[[Enumeriable]]']=== ESV["true"] &&
            Desc['[[Enumeriable]]'] === ESV["false"]) return Reject();
         if (current['[[Enumeriable]]'] === ESV["false"] &&
            Desc['[[Enumeriable]]'] === ESV["true"]) return Reject();
         }
      //step 8
      if (S8.IsGenericDescriptor(Desc)===ESV["true"]) {
```

```
         /* no further validation required */
         }
      // step 9
      else if (S8.IsDataDescriptor(current) !== S8.IsDataDescriptor(Desc)) {
         // 9.a
         if (current['[[Configurable]]'] === ESV["false"]) return Reject();
         // 9.b
         if (S8.IsDataDescriptor(current) === ESV["true"]) {
            // 9.b.i
            delete current['[[Value]]'];
            delete current['[[Writable]]'];
            current['[[Get]]'] = ESV['undefined'];
            current['[[Set]]'] = ESV['undefined'];
            }
         else {
            // 9.c.i
            delete current['[[Get]]'];
            delete current['[[Set]]'];
            current['[[Value]]'] = ESV['undefined'];
            current['[[Writable]]'] = ESV['false'];
            }
         }
      // step 10
      else if (S8.IsDataDescriptor(current) === ESV["true"] &&
               S8.IsDataDescriptor(Desc) === ESV["true"]) {
         // 10.a
         if (current['[[Configurable]]'] === ESV["false"]) {
            // 10.a.i
            if (current['[[Writable]]']===ESV["false"] &&
                Desc['[[Writable]]']===ESV["true"]) return Reject();
            // 10.a.ii
            if (current['[[Writable]]']===ESV["false"])
            // 10.a.ii.1
               if ('[[Value]]' in Desc &&
                   S9.SameValue(Desc['[[Value]]'],
                                current['[[Value]]'])=== ESV["false"]
                   ) return Reject();
            }
         // 10.b
         ESassert(current['[[Configurable]]'] === ESV["true"]);
         }
      // step 11
      else {
         ESAssert(S8.IsAccessorDescriptor(current) === ESV["true"] &&
            S8.IsAccessorDescriptor(Desc) === ESV["true"]);
         // 11.a
         if (current['[[Configurable]]'] === ESV["false"]) {
            if ('[[Set]]' in Desc &&
                  S9.SameValue(Desc['[[Set]]'],current['[[Set]]']) === ESV["false"]
               ) return Reject();
            if ('[[Get]]' in Desc &&
                  S9.SameValue(Desc['[[Get]]'],current['[[Get]]']) === ESV["false"]
               ) return Reject();
            }
         }
      // step 12
      for ( implIndx in implDescFields) {
         implField = implDescFields[implIndx];
         current[implField] = Desc[implField];
         }
      // step 13
      return ESV["true"];
      }

   } /* objectCommonInternalMethods */

// In this implementation host objects are wrappered by an object that enforces
// specified host object constraints.
```

```
// These definetions define the behavior for such wrappers
var hostObjectBehavior = implCreate(objectCommonInternalMethods);
hostObjectBehavior.implHostObj = undefined;  //reference to actual host object
hostObjectBehavior.isHostObject = true;
hostObjectBehavior['[[DefaultValue]]']= function (hint) {//8.12.9 last paragraph
   var result;
   if (this.implHostObj['[[DefaultValue]]']) {
      result = this.implHostObj['[[DefaultValue]]'].call(this.implHostObj,hint);
      ESassert(S4.isPrimitiveValue(result));
      return result;
      }
   return objectCommonInternalProperties['[[DefaultValue]]'].call(this,hint);
   }

function implMakeHostObject(hostObj) {
   var obj = new ECMAScriptValue(ECMAScriptType.Object);
   implExtend(obj,hostObjectBehavior);
   obj.implHostObject = hostObj;
   return obj;
   }

// Section 8 Abstract Operations
var S8 = {
   Type: function (x) {                // 8
      ESassert (x.type.typeName in ECMAScriptType &&
                x.type === ECMAScriptType[x.type.typeName]);
      return x.type;
      },

   //Operations upon Reference type values
   GetBase: function (V) {         //8.7
      ESassert(S8.Type(V) === ECMAScriptType.Reference);
      return V.base;
      },
   GetReferencedName: function (V) {  //8.7
      ESassert(S8.Type(V) ===ECMAScriptType.Reference);
      return V.referenceName;
      },
   IsStrictReference: function (V) {  //8.7
      ESassert(S8.Type(V) ===ECMAScriptType.Reference);
      if (V.strictReference===ESV["true"]) return ESV("true");
      else return ESV("false");
      },
   HasPrimitiveBase: function (V) {  //8.7
      ESassert(S8.Type(V)===ECMAScriptType.Reference);
      if (S8.Type(V.base)===ECMAScriptType.Boolean ||
          S8.Type(V.base)===ECMAScriptType.String ||
          S8.Type(V.base)===ECMAScriptType.Number
         ) return ESV("true");
      else return ESV("false");
      },
   IsPropertyReference: function (V) {  //8.7
      ESassert(S8.Type(V)===ECMAScriptType.Reference);
      if (S8.Type(V.base)===ECMAScriptType.Object ||
          HasPrimitiveBase(V)===ESV['true']) return ESV("true");
      else return ESV("false");
      },
   IsUnresolvableReference: function (V) {  //8.7
      ESassert(S8.Type(V)===ECMAScriptType.Reference);
      if (V.base===ESV["undefined"]) return ESV("true");
      else return ESV("false");
      },
   GetValue: function (V) {  //8.7.1
      if (S8.Type(V)!==ECMAScriptType.Reference) return V;
      var base = S8.GetBase(V);
      if (S8.IsUnresolvedReference(V)===ESV['true'])
         S5.throwException("ReferenceException", "8.7.1 step 3");
      if (S8.IsPropertyReference(V)===ESV['true']) {
```

```
         var get;
         if (S8.HasPrimitiveBase(V)===ESV['false']) get = base['[[Get]]'];
         else get = primitiveValue_Get;
         return get.call(base, S8.GetReferencedName(V));
         }
      else {
         //base must be an Environment Record
         ESassert(S8.Type(base)===ECMAScriptType.EnvironmentRecord);
         return
base.GetBindingValue(S8.GetReferencedName(V),S8.IsStrictReference(V));
         }
      // Special [[Put]] internal method for primitive values
      function primitiveValue_Get(P) {
         var base = this;
         var O = S9.ToObject(base);
         var desc = O['[[GetProperty]]'](P);
         if (desc === ESV['undefined']) return ESV['undefined'];
         if (S8.IsDataDescriptor(desc) === ESV["true"]) return desc['[[Value]]'];
         ESassert(S8.IsAccessorDescriptor(desc) === ESV["true"]);
         var getter = desc['[[Get]]'];
         if (getter === ESV['undefined']) return ESV['undefined'];
         return getter['[[Call]]'](base,ESV.emptyList);
         }
      },
   PutValue: function (V, W) {  //8.7.2
      if (S8.Type(V)!==ECMAScriptType.Reference)
         S5.throwException("ReferenceError", "8.7.2 step 1");
      var base = S8.GetBase(V);
      if (S8.IsUnresolvedReference(V)===ESV['true']) {
         if (S8.IsStrictReference(V)===ESV['true'])
            S5.throwException("ReferenceError", "8.7.2 step 3.a.i");
         S15.GlobalObject['[[Put]]'](S8.GetReferencedName(V),W,ESV['false']);
         }
      else if (S8.IsPropertyReference(V)===ESV['true']) {
         var put;
         if (S8.HasPrimitiveBase(V)===ESV['false']) put = base['[[Put]]'];
         else put = primitiveValue_Put;
         put.call(base,S8.GetReferencedName(V),W,S8.IsStrictReference(V));
         }
      else {
         ESassert(base.type===ECMAScriptType.EnvironmentRecord);
         base.SetMutableBinding(S8.GetReferencedName(V),
                                W,S8.IsStrictReference(V));
         }
      return;
      // Special [[Put]] internal method for primitive values
      function primitiveValue_Put (P,W,Throw) {
         var base = this;
         var O = S9.ToObject(base);
         if (O['[[CanPut]]'] === ESV['false']){
            if (Throw === ESV['true'])
               S5.throwException("TypeError", "8.7.2 [[Put]] step 2.a");
            else return;
            }
         var ownDesc = O['[[GetOwnProperty]]'](P);
         if (S8.IsDataDescriptor(ownDesc) === ESV["true"]) {
            if (Throw === ESV['true'])
               S5.throwException("TypeError", "8.7.2 [[Put]] step 4.a");
            else return;
            }
         var desc = O['[[GetProperty]]'](P);
         if (S8.IsAccessorDescriptor(desc) === ESV["true"]) {
            var setter = desc['[[Set]]'];
            Esassert(setter !== ESV['undefined']);
            setter['[[Call]]'](base,new ECMAScriptValue(ECMAScriptType.List,W));
            }
         else {
            //Attempt to create an own property on a transient object
```

```
         if (Throw === ESV['true'])
            S5.throwException("TypeError", "8.7.2 [[Put]] step 7.a");
         }
      return;
      }
   },

//Operations upon Property Descriptor type values
IsAccessorDescriptor: function (Desc) {         //8.10.1
   ESassert(S8.Type(Desc) === ECMAScriptType.PropertyDescriptor);
   if (Desc === ESV['undefined']) return ESV['false'];
   if (!('[[Get]]' in Desc) && !('[[Put]]' in Desc))
      return ESV['false'];
   return ESV['true'];
   },
IsDataDescriptor: function (Desc) {           //8.10.2
   ESassert(S8.Type(Desc) === ECMAScriptType.PropertyDescriptor);
   if (Desc === ESV['undefined']) return ESV['false'];
   if (!('[[Value]]' in Desc) && !('[[Writable]]' in Desc))
      return ESV['false'];
   return ESV['true'];
   },
IsGenericDescriptor: function (Desc) {         //8.10.3
   ESassert(S8.Type(Desc) === ECMAScriptType.PropertyDescriptor);
   if (Desc === ESV['undefined']) return ESV['false'];
   if (S8.IsAccessorDescriptor(Desc) === ESV["false"] &&
       S8.IsDataDescriptor(Desc) === ESV["false"]) return ESV['true'];
   return ESV['false'];
   },
FromPropertyDescriptor: function (Desc) {         //8.10.4
   ESassert(S8.Type(Desc) === ECMAScriptType.PropertyDescriptor);
   //Desc must be a fully populated property descriptor
   if (Desc === ESV['undefined']) return ESV['undefined'];
   var obj = S15.asIfByNewObject();
   var implDDesc = new ECMAScriptValue(ECMAScriptType.PropertyDescriptor,
      {'[[Writable]]': ESV["true"], '[[Enumerable]]': ESV["true"],
       '[[Configurable]]':  ESV["true"]}
      );
   if (S8.IsDataDescriptor(Desc) === ESV["true"]) {
      implDDesc['[[Value]]'] = Desc['[[Value]]'];
      obj['[[DefineOwnProperty]]']( 
         ESS["value"], implDDesc, ESV["false"]);
      implDDesc['[[Value]]'] = Desc['[[Writable]]'];
      obj['[[DefineOwnProperty]]']( 
         ESS["writable"], implDDesc, ESV["false"]);
      }
   else {
      ESassert(S8.IsAccessDescriptor(Desc) === ESV["true"]);
      implDDesc['[[Value]]'] = Desc['[[Get]]'];
      obj['[[DefineOwnProperty]]']( 
         ESS["get"], implDDesc, ESV["false"]);
      implDDesc['[[Value]]'] = Desc['[[Set]]'];
      obj['[[DefineOwnProperty]]']( 
         ESS["set"], implDDesc, ESV["false"]);
      }
   implDDesc['[[Value]]'] = Desc['[[Enumerable]]'];
   obj['[[DefineOwnProperty]]']( 
      ESS["enumerable"], implDDesc, ESV["false"]);
   implDDesc['[[Value]]'] = Desc['[[Configurable]]'];
   obj['[[DefineOwnProperty]]']( 
      ESS["configurable"], implDDesc, ESV["false"]);
   return obj;
   },
ToPropertyDescriptor: function (Obj) {            //8.10.5
   if (S8.Type(Obj) !== ECMAScriptType.Object)
      S5.throwException("TypeError", "8.10.5 step 1");
   var desc = new ECMAScriptValue(ECMAScriptType.PropertyDescriptor);
   if (Obj['[[hasProperty]]'](ESS["enumerable"]) === ESV["true"]) {
```

```
         var enum = Obj['[[Get]]'](ESS["enumerable"]);
         desc['[[Enumerable]]'] = S9.ToBoolean(enum);
         }
      if (Obj['[[hasProperty]]'](ESS["configurable"]) === ESV["true"]) {
         var conf = Obj['[[Get]]'](ESS["configurable"]);
         desc['[[Configurable]]'] = S9.ToBoolean(conf);
         }
      if (Obj['[[hasProperty]]'](ESS["value"]) === ESV["true"]) {
         var value = Obj['[[Get]]'](ESS["value"]);
         desc['[[Value]]'] = value;
         }
      if (Obj['[[hasProperty]]'](ESS["writable"]) === ESV["true"]) {
         var writable = Obj['[[Get]]'](ESS["writable"]);
         desc['[[Writable]]'] = S9.ToBoolean(writable);
         }
      if (Obj['[[hasProperty]]'](ESS["get"]) === ESV["true"]) {
         var getter = Obj['[[Get]]'](ESS["get"]);
         if (S9.IsCallable(getter) === ESV["false"] && getter !== ESV['undefined'])
            S5.throwException("TypeError", "8.10.5 step 7.b");
         desc['[[Get]]'] = getter ;
         }
      if (Obj['[[hasProperty]]'](ESS["set"]) === ESV["true"]) {
         var setter = Obj['[[Set]]'](ESS["set"]);
         if (S9.IsCallable(setter) === ESV["false"] && setter !== ESV['undefined'])
            S5.throwException("TypeError", "8.10.5 step 8.b");
         desc['[[Set]]'] = setter ;
         }
      if ('[[Get]]' in desc || '[[Set]]' in desc) {
         if ('[[Value]]' in desc || '[[Writable]]' in desc)
             S5.throwException("TypeError", "8.10.5 step 9.a");
         }
      return desc;
      }

} /* End Section 8 Abstract Operations */


/// Section 9

// Section 9 Abstract Operations
var S9 = {
   ToPrimitive: function (input,PreferedType){ //9.1
      ESassert(!S8.Type(argument).specificationType);
      if (S8.Type(input) === ECMAScriptType.Undefined) return input;
      if (S8.Type(input) === ECMAScriptType.Null) return input;
      if (S8.Type(input) === ECMAScriptType.Boolean) return input;
      if (S8.Type(input) === ECMAScriptType.Number) return input;
      if (S8.Type(input) === ECMAScriptType.String) return input;
      if (S8.Type(input) === ECMAScriptType.Object) {
         if (arguments.length==1) return input['[[DefaultValue]]']();
         else return input['[[DefaultValue]]'](PreferedType);
         }
      },

   ToBoolean: function (input){    //9.2
     ESassert(!S8.Type(argument).specificationType);
     if (S8.Type(input) === ECMAScriptType.Undefined) return ESV["false"];
      if (S8.Type(input) === ECMAScriptType.Null) return ESV["false"];
      if (S8.Type(input) === ECMAScriptType.Boolean) return input;
      if (S8.Type(input) === ECMAScriptType.Number) {
         if (input.numberValue===ESV["NaN"] || input.numberValue === +0 ||
             input.numberValue === -0
            ) return ESV["false"];
         else return ESV["true"];
         }
      if (S8.Type(input) === ECMAScriptType.String) {
         if (input.stringLength == 0) return ESV["false"];
         else return ESV["true"];
```

```
      }
   if (S8.Type(input) === ECMAScriptType.Object)  return ESV["true"];
   },

ToNumber: function (input){    //9.3
   ESassert(!S8.Type(argument).specificationType);
   if (S8.Type(input) === ECMAScriptType.Undefined) return ESV["NaN"];
   if (S8.Type(input) === ECMAScriptType.Null)
      return new ECMAScriptValue(ECMAScriptType.Number, +0);
   if (S8.Type(input) === ECMAScriptType.Boolean) {
      if (input === ESV("true"))
         return new ECMAScriptValue(ECMAScriptType.Number,1);
      else
         /* input === ESV("false") */
         return new ECMAScriptValue(ECMAScriptType.Number,+0);
      }
   if (S8.Type(input) === ECMAScriptType.Number) return input;
   if (S8.Type(input) === ECMAScriptType.String)
      return ToNumberAppliedToString(input.stringValue);
   if (S8.Type(input) === ECMAScriptType.Object) {
      var primValue = S9.ToPrimitive(input,ESV.hint.Number);
      return S9.ToNumber(primValue);
      }
   },

ToInteger: function (argument){   //9.4
  ESassert(!S8.Type(argument).specificationType);
  implTODO("9.4 ToInteger");
   },

ToInt32: function (argument){    //9.5
   ESassert(!S8.Type(argument).specificationType);
   implTODO("9.5 ToInt32");
   },

ToUInt32: function (argument){    //9.6
   ESassert(!S8.Type(argument).specificationType);
   implTODO("9.5 ToUInt32");
   },

ToUInt16: function (argument){    //9.7
   ESassert(!S8.Type(argument).specificationType);
   implTODO("9.5 ToUInt16");
   },

ToString: function (input){    //9.3
   ESassert(!S8.Type(argument).specificationType);
   if (S8.Type(input) === ECMAScriptType.Undefined)
      return new ECMAScriptValue(ECMAScriptType.String, "undefined");
   if (S8.Type(input) === ECMAScriptType.Null)
      return new ECMAScriptValue(ECMAScriptType.String, "null");
   if (S8.Type(input) === ECMAScriptType.Boolean) {
      if (input === ESV("true"))
         return new ECMAScriptValue(ECMAScriptType.String, "true");
      else
         /* input === ESV("false") */
         return new ECMAScriptValue(ECMAScriptType.String, "false");
      }
   if (S8.Type(input) === ECMAScriptType.Number)
      return ToStringAppliedToNumber(input.IEEEValue);
   if (S8.Type(input) === ECMAScriptType.String) return input;
   if (S8.Type(input) === ECMAScriptType.Object) {
      var primValue = S9.ToPrimitive(input,ESV.hint.String);
      return S9.ToString(primValue);
      }
   },

ToObject: function (argument) {   //9.9
```

```
      ESassert(!S8.Type(argument).specificationType);
      if (S8.Type(argument) === ECMAScriptType.Undefined ||
          S8.Type(argument) === ECMAScriptType.Null
         ) S5.throwException("TypeError", "9.9 ToObject");
      if (S8.Type(argument) === ECMAScriptType.Boolean) {
         return implTODO("create a Boolean Object");
         }
      if (S8.Type(argument) === ECMAScriptType.Number) {
         return implTODO("create a Number Object");
         }
      if (S8.Type(argument) === ECMAScriptType.String) {
         return implTODO("create a String Object");
         }
      ESassert(!S8.Type(argument)=== ECMAScriptType.Object);
      return argument;
      },

   CheckObjectCoercible: function (esVal) {          //9.10
      ESassert(!S8.Type(esVal).specificationType);
      if (S8.Type(esVal) === ECMAScriptType.Undefined ||
          S8.Type(esVal) === ECMAScriptType.Null
         ) S5.throwException("TypeError", "9.10 CheckObjectCoercible");
      return;
      },

   IsCallable: function (esVal) {             //9.11
      ESassert(!S8.Type(esVal).specificationType);
      if (S8.Type(esVal) === ECMAScriptType.Object &&
          esVal['[[Call]]'] !== undefined) return ESV["true"];
      return ESV["false"];
      },

   SameValue: function (x, y) {         //9.12
      ESassert(!S8.Type(x).specificationType);
      ESassert(!S8.Type(y).specificationType);
      if (S8.Type(x) !== S8.Type(y)) return ESV["false"];
      if (S8.Type(x) === ECMAScriptType.Undefined) return ESV["true"];
      if (S8.Type(x) === ECMAScriptType.Null) return ESV["true"];
      if (S8.Type(x) === ECMAScriptType.Number) {
         if (x===ESV["NaN"] && y===ESV["NaN"]) return ESV["true"];
         if (x.numberValue === +0 && y.numberValue === -0) return ESV["false"];
         if (x.numberValue === -0 && y.numberValue === +0) return ESV["false"];
         if (x.numberValue === y.numberValue) return ESV["true"];
         return ESV["false"];
         }
      if (S8.Type(x) === ECMAScriptType.String) {
         if (x.stringLength === y.stringLength &&
             x.characters === y.characters) return ESV["true"];
         return ESV["false"];
         }
      if (S8.Type(x) === ECMAScriptType.Boolean) {
         if (x === y) return ESV["true"];
         return ESV["false"];
         }
      if (S8.Type(x) === ECMAScriptType.Object) {
         if (x === y) return ESV["true"];
         return ESV["false"];
         }
      return ESV["false"];
      }
   } /* End Section 9 Abstract Operations */

//9.3.1
function ToNumberAppliedToString(str){
    implTODO("ToNumberAppliedToString");
    }

//9.8.1
```

```
function ToStringAppliedToNumber(m){
    implTODO("ToStringAppliedToNumber");
    }

return  {// interface object for externally exercising the interpreter
         S4:S4,
         S5: S5,
         S8: S8,
         S9: S9,
         ECMAScriptType: ECMAScriptType,
         ECMAScriptValue: ECMAScriptValue,
         ESV: ESV,
         private_for_testing: {
            ESassert: ESassert,
            implTODO: implTODO,
            implCreate: implCreate,
            implExtend: implExtend,
            ESS: ESS,
            objectCommonInternalMethods : objectCommonInternalMethods,
            hostObjectBehavior : hostObjectBehavior ,
            implMakeHostObject: implMakeHostObject
            }
        };



}



function tests(engine) {

   var results;
   with (engine) {
      results = {
        0: "Test results:",
        // check well known values
        1:  ESV["undefined"].type===ECMAScriptType.Undefined,
        2:  ESV["null"].type===ECMAScriptType.Null,
        3:  ESV["true"].type===ECMAScriptType.Boolean,
        4:  ESV["true"].implValue===true,
        5:  ESV["false"].type===ECMAScriptType.Boolean,
        6:  ESV["false"].implValue===false,
        7:  ESV["NaN"].type===ECMAScriptType.Number,
        8:  ESV["NaN"].IEEEValue !=ESV["NaN"].IEEEValue , //NaN!=NaN is true
        9:  isNaN(ESV["NaN"].IEEEValue),
        10: ESV.emptyString.type===ECMAScriptType.String,
        11: ESV.emptyString.characters===="",
        12: ESV.emptyString.stringLength===0,
        13: (""+ESV.hint.String)==="ESV.hint.String",
        14: (""+ESV.hint.Number)==="ESV.hint.Number",
        15: (""+ESV.empty)==="ESV.empty",
        16: ESV.emptyList.implElements.length===0,

        // check utility functions
        17: test_ESassert(),
        18: test_implTODO(),
       length: 19
        }
      }
   return results;

   function test_ESassert() {
      with (engine.private_for_testing) {
         var test1, test2;
         try {ESassert(true); test1=true;} catch (e) {test1=false};
         try {ESassert(false); test2=false;} catch (e) {test2=true};
```

```
          return test1 && test2;
          }
       }

   function test_implTODO() {
      with (engine.private_for_testing) {
         try {implTODO("XXX"); return false} catch (e) {return true}
         }
      }

}


var e=ES5();
alert(Array.prototype.join.call(tests(e),","));
```