Ecma/TC39/2008/056

Proposed ECMAScript 4th Edition Specification Draft 2

11 July 2008

The Proposed ECMAScript 4th Edition (ES4) Specification is divided into roughly four components:

Syntax

Concrete syntax and translation into abstract syntax

Core semantics

Values and storage, types, scopes, name resolution, and name lookup

Computation

Loading, definition and binding, expressions, and statements

Libraries

Pre-defined data types, objects, functions, and constants

The present draft is incomplete. It includes the grammar for the concrete syntax, the entire second component (the core semantics) and fourth component (the predefined libraries). The third component, computation, is missing.

As this is an early draft, there are some redundancies among the chapters, some background information may be absent, there are some rough parts in the formal specifications, and there is a fair amount of nonuniformity in the notation used. Several chapters contain tutorial material that will move elsewhere, eventually. There are circular dependencies between chapters.

The editors anticipate delivering future drafts to TC39 about two weeks before every scheduled TC meeting. In the interval between TC meetings drafts of parts of the specification (in the form of "speclets") are being circulated and debated on the ES4 discussion list [1].

Familiarity with ECMAScript 3rd Edition (ES3) Specification is required to understand the present draft.

Specification Formalism

The ES4 specification makes use of the following formalisms.

Syntax

The surface syntax is specified using a context-free grammar formalism similar to those used to specify ES3, C, C++, Java, and C#.

The surface syntax is mapped to abstract syntax trees (ASTs). Some syntactic forms are desugared into core forms in this mapping. For example, destructuring bindings are desugared into a combination of simple bindings, temporary variables, and assignment expressions. ASTs represent only the core forms of the language.

Core semantics and computation

The semantics of the AST form of the language are expressed in terms of operations on ASTs and on additional structures that represent ECMAScript values, semantic (primitive) values, and metadata. The operations as well as all the data structures for ASTs, ECMAScript values, semantics values, and metadata are expressed in the language Standard ML [2].

Most data structures encountered in the following chapters belong to the ASTs.

Libraries

The semantics of the pre-defined types, objects, functions, and constants are expressed largely in terms of ES4 code operating on ES4 data structures.

The bulk of the ES4 specification consists of normative prose. The Standard ML code and the ES4 fragments are used to specify precisely only certain aspects of the language, namely those that were described by informal pseudocode in the ES3 spec, some that were described only by prose in the ES3 spec, and some that are new to ES4 (such as the type system).

There are three significant reasons why we use Standard ML and ES4 instead of pseudocode. First, these are well-defined languages with significantly greater expressiveness than the low-level pseudocode. Second, as they are real programming

languages, the specification fragments can be checked by machine, given a surrounding test infrastructure. Finally, with care, the machine checked code can be included directly in the specification document.

Over the course of the past year, members of the ES4 working group have been creating just such a test infrastructure in the form of a complete reference implementation of ES4. The Standard ML and ES4 fragments in this specification have--with very few exceptions--been excerpted from that reference implementation. (In some respects the reference implementation and the specification are not yet syncronized, and that will sometimes be visible in the chapters that follow.)

One real danger with using a live implementation for specification purposes is overspecification. Some care has been taken to abstract the implementation away from those details that we do not wish to specify. In the case of the libraries, this strategy has been very successful; in the case of the core language, there is more work to be done. But there is no indication at this point that it cannot be done. We also note that the ES3 Specification is very detailed and that there is only a small difference in specificity between most pseudocode fragments of the ES3 Specification and the corresponding Standard ML fragments of the ES4 specification.

Another danger with using a live implementation is that there may be mismatches between the modelling language and the modelled language. If we wish to model ES4 "double" values (IEEE 64-bit floating point values with all the aspects required by the ES3 spec) directly as Standard ML Real64.real values, then the two types must correspond very closely. Otherwise we must abstract away from the modelling language to some extent.

A final note about the terms "semantic" and "semantics" as used in the core language specification: they invariably mean "at the level of Standard ML".

The Themes of ES4

The central themes in the design of ES4 have been the following.

Compatibility with ES3; interoperation between ES3 and ES4

Compatibility with ES3 cannot be sacrificed except in situations where compatibility does not make a practical difference, where existing implementations differ or a *de facto* standard has evolved, or where compatibility would fatally imperil other goals. Therefore, one central goal of the work has been that ES4 implementations should run all existing ES3 programs and that ES3 and ES4 code should be able to be intermixed in a single global environment. (The ES4 working group has released a paper [4] examining the compatibility between ES3 and ES4.)

Programming in the large

The largest ECMAScript programs are significantly larger in 2008 than they were in 1999, when the ES3 Specification was released. The ES3 facilities for programming in the large are relatively poor (though often serviceable) and ES4 should (in the opinion of the working group) provide improved facilities for encapsulation, consistency, error checking, and library building.

Program evolution

While ES is increasingly being used for large programs, it is also being used for small scripts and for programs that start small but (sometimes) evolve over time to become large. ES4 must provide support for program evolution at various granularities.

Convenience features

Conveniences help make a language more useful (they hide tedious detail). The conveniences expected by programmers change over time, as new patterns and ideas make it into the mainstream or supplant other ideas; ES4 should not fight this but should incorporate patterns and ideas from other languages if appropriate and possible.

Bug fixes

Not all designs work out as expected. For example, the stripping of Unicode format control characters specified by ES3 annoys programmers who like to embed those characters in literal strings and regular expressions, where they are required either for proper formatting on output or to properly match input. If an incompatible change to the language that repairs a design bug is likely to fix significantly more programs than it breaks (as is the case for banning format control character stripping), then the change should be considered for ES4.

Self-hosting

On the principle that what is good for the language is good for its users, it is a goal that the bulk of the pre-defined libraries (data types, functions, objects, and constants) should be expressible in ES4. For example, though ES3 does not provide a class abstraction the pre-defined objects of ES3 are most easily expressed in class terms, and classes and interfaces proliferate in host object models for ES3. For these reasons (and others), classes are a candidate feature for ES4.

A Brief Overview of ES4 Facilities

In October 2007 the ES4 working group released a paper [5] outlining the feature space of ES4. Since that time a significant number of proposed features have been removed from the language; yet others have been streamlined or generalized. The most significant facilities that remain in the draft language are the following.

Namespaces

Namespaces are values that serve as tags on names; they help to segregate the names of independently developed subprograms, they aid versioning, and they make it possible to control access to properties.

Types and type constraints

Classes in the style of Java and (non-recursive) structural record and array types are provided as more reliable alternatives to ES3 constructor functions. The advantage of classes and the structural types is that they *fix* object properties, thereby providing programs with greater integrity against accidental or malicious changes.

Bound variables and fixed object properties can be *annotated* with types. The annotations place constraints on the variables and properties, which can be updated only with values conforming to the type constraint.

Crucially, type annotations are always optional; the programmer adds type annotations on an as-needed basis. This results in a type system that is *gradual*.

The type system is complemented with interface types in the style of Java and with union types that provide a kind of *post hoc* interfaces.

Finally, types can be aliased, and both type aliases and class and interface definitions can be parameterized.

Creature comforts

ES4 adds proper block scoping and constant bindings, new control structures such as iterators and generators, and getters and setters on objects.

There also is a fair amount of new syntactic sugar, including succinct function definitions and function expressions, destructuring binding and assignment, and array comprehensions.

Upgraded libraries

The ES4 libraries add new methods to objects inherited from ES3, and provide new data types (such as homogenous vectors, maps, and decimal floating point numbers), new top-level functions (such as object identity hashing), and reflection facilities.

Performance

Though the performance of such highly dynamic languages as ES4 is a problem that may well be easiest to solve with an advanced, on-line optimizing implementation, the language design can contribute. Better performance can be supported in the language by providing optimization hooks, such as type annotations, early-bindable methods, and dedicated syntax.

Programs, Compilation Units, and Compilation Phases

A complete ES4 program is comprised of a possibly unbounded linear sequence of *compilation units* (each of which conforms to the *Program* nonterminal in the grammar) that is loaded into a *global environment*. Each compilation unit may affect the global environment by contributing definitions of new names to it or by changing the values stored in it. There may be multiple global environments. ES4 provides no mechanism by which code running in one global environment can gain access to another global environment; however, implementations of ES4 may provide such mechanisms, and the ES4 Specification will occasionally address the meaning of code that operates on multiple global environments.

Compilation units pass through three phases: parsing, definition, and evaluation. Parsing and definition together transform the compilation unit from source form into an AST form, in the process (a) checking that the compilation unit conforms to the grammar and to extra-grammatical, syntactic constraints, and (b) resolving names that denote namespaces and types. Evaluation performs the operations that are specified by the resulting AST.

References

[1] es4-discuss@mozilla.org
[2] Milner, Tofte, Harper, McQueen, "The Definition of Standard ML - Revised", MIT Press, 1997
[3] ECMAScript 4 reference implementation, `http://www.ecmascript.org/download.php`
[4] "Compatibility between ES3 and proposed ES4", `http://www.ecmascript.org/es4/spec/incompatibilities.pdf`
[5] "Proposed ECMAScript 4th Edition -- Language overview", `http://www.ecmascript.org/es4/spec/overview.pdf`