Norbert's Corner

Supplementary Characters for ECMAScript

May 8, 2012

ECMAScript, the standard underlying JavaScript, has so far been ambivalent about supporting supplementary characters, those Unicode characters that are outside the 16-bit code space originally envisioned for Unicode and require more than 16 bits for their encoding. The ECMAScript standard specifies UTF-16, the Unicode encoding form designed to extend the original 16-bit encoding to support supplementary characters, as the encoding of source text and strings, and implementations generally allow supplementary characters to be used. Browsers have surrounded ECMAScript implementations with text input, text rendering, DOM APIs, and XMLHttpRequest with full Unicode support, and generally use full UTF-16 to exchange text with their ECMAScript subsystem. Developers have used this to build applications that support supplementary characters. However, some text processing functionality in ECMAScript itself is defined to operate on code units separately, so that it cannot correctly interpret supplementary characters.

This article proposes a set of ECMAScript specification changes that enable correct processing of supplementary characters while maintaining compatibility with existing applications. The basic idea is to keep the existing text representation, but change operations that interpret text to be based on Unicode code points. Supplementary characters are then interpreted as atomic entities with their correct Unicode semantics, just like characters in the Basic Multilingual Plane. This is similar to the design of supplementary character support in Java.

All changes described are relative to the ECMAScript Language Specification, 5.1 edition. The outline of this proposal has been presented and discussed at the Ecma TC 39 meeting on March 29, 2012; this updated version reflects the consensus reached at the meeting, but provides more detail.

Terms and Definitions

The ECMAScript Language Specification currently uses the term "character" throughout, but redefines it to actually mean "code unit". In order to clearly describe a system that is based on 16-bit code units, but supports all Unicode characters, we need to be a bit more precise in our terminology. This proposal relies on the following three definitions, which would replace the definition of "String value" in section 4.3.16, and would largely remove the need to use the overloaded term "character".

4.3.16 String value: primitive value that is a finite ordered sequence of zero or more code units

NOTE: A String value is a member of the String type. Where ECMAScript operations interpret code units, they are interpreted as UTF-16 code units. However, ECMAScript does not place any restrictions or requirements on the sequence of code units in a String value, so it may be ill-formed when interpreted as a UTF-16 code unit sequence.

4.3.17 Code unit: unsigned 16-bit integer. All values from 0 to $FFFF_{16}$ are allowed. Code unit values are described by the string "0x" followed by four hexadecimal digits.

4.3.18 Code point: unsigned integer value in the range from 0 to $10FFFF_{16}$. Code point values are described by the string "U+" followed by four to six hexadecimal digits.

NOTE: This definition follows that of the Unicode standard, so this specification may use "code point" and "Unicode code point" interchangeably.

NOTE: This specification may use the word "character" to refer to specific assigned code points in the Basic Multilingual Plane, where the distinction between code units and code points is irrelevant.

4.3.19 Surrogate code unit: A code unit in the range from 0xD800 to 0xDFFF.

4.3.20 Surrogate pair: A code unit sequence consisting of a code unit in the range from 0xD800 to 0xDBFF followed by a code unit in the range from 0xDC00 to 0xDFFF.

NOTE: Surrogate pairs are used in UTF-16 to represent code points above U+FFFF. If a surrogate code unit occurring within a UTF-16 code unit sequence is not part of a surrogate pair (an "unpaired surrogate"), the sequence is ill-formed.

Text Interpretation

In order to support supplementary characters, some ECMAScript operations will have to interpret UTF-16 code unit sequences as code point sequences. This new section describes how. For compatibility with existing applications, it has to allow surrogate code points (code points between U+D800 and U+DFFF which can never represent characters).

5.3 Text Interpretation

Text is represented in ECMAScript as a sequence of UTF-16 code units, but sometimes interpreted as a sequence of code points. Interpretation is as follows:

- A code unit in the range 0 to 0xD7FF or in the range 0xE000 to 0xFFFF is interpreted as a code point with the same value.
- A sequence of two code units, where the first code unit *c1* is in the range 0xD800 to 0xDBFF and the second code unit *c2* is in the range 0xDC00 to 0xDFFF, is a surrogate pair and is interpreted as a code point with the value (*c1* - 0xD800) * 0x400 + (*c2* - 0xDC00) + 0x10000.
- A code unit that is in the range 0xD800 to 0xDFFF, but is not part of a surrogate pair, is interpreted as a code point with the same value.

Unicode Code Point Escape Sequences

Unicode Code Point Escape Sequences

A new form of Unicode escape sequence, originally proposed by Markus Scherer, allows the expression of code points rather than code units. For example, the String containing the character 𠮷 can be written as "\u{20BB7}" in addition to the existing forms "𠮷" and "\uD842\uDFB7". The escape sequence uses between one and six hexadecimal digits, but only values up to U+10FFFF are allowed. The new form is more readable than the old escape form (although the character itself of course is best in that respect) and easier to find in Unicode tables. Unicode code point escape sequences can be used in string literals, in the patterns of regular expressions with the Unicode flag set (see the following section), and in identifiers.

Note that ECMAScript cannot add the new escape form to JSON, because JSON is defined by its own specification.

Details (skip)

This section covers parts of the specification common to string literals, regular expression patterns, and identifiers as well as those specific to string literals. Parts of the specification specific to regular expression literals and identifiers are covered in subsequent sections.

In clause 6, replace the paragraph starting with “In string literals” with:

> In string literals, regular expression patterns, and identifiers, code units may also be expressed as Unicode escape sequences. There are two forms of Unicode escape sequences:
>
> - Unicode code unit escape sequences consist of six characters, namely \u plus four hexadecimal digits, and contribute one code unit.
> - Unicode code point escape sequences consist of five to ten characters, namely \u{ plus one to six hexadecimal digits plus }, and contribute one or two code units.
>
> Within a string literal, a Unicode escape sequence contributes code units to the value of the literal. With a regular expression pattern, a Unicode escape sequence contributes code units to character literals or to characters within classes; they cannot represent the terminal symbols of the RegExp grammar. Within an identifier, an escape sequence contributes code units to the identifier.

In section 7.8.4, change the production for EscapeSequence to:

> *EscapeSequence* ::
> *CharacterEscapeSequence*
> **0** [lookahead ∉ *DecimalDigit*]
> *HexEscapeSequence*
> *UnicodeEscapeSequence*
> *UnicodeCodePointEscapeSequence*

After the production for UnicodeEscapeSequence, insert:

> *UnicodeCodePointEscapeSequence* ::
> **u{** *HexDigit* $HexDigit_{opt}$ $HexDigit_{opt}$ $HexDigit_{opt}$ $HexDigit_{opt}$ $HexDigit_{opt}$ **}**

At the end of the bullet list in that section, insert:

- The CV of *UnicodeCodePointEscapeSequence* :: **u{** *HexDigit* $HexDigit_{opt}$ $HexDigit_{opt}$ $HexDigit_{opt}$ $HexDigit_{opt}$ $HexDigit_{opt}$ **}** is calculated as follows:
  - Let *cpv* be 0.
  - For each *HexDigit* provided in sequence

- For each *HexDigit* provided, in sequence:
  - Let *cpv* be *cpv* * 16 plus the MV of the *HexDigit*.
- If *cpv* > 0x10FFFF, report a syntax error.
- If *cpv* > 0xFFFF, then the CV is the code unit sequence of CV1 and CV2, where CV1 is ((*cpv* - 0x10000) >> 10) + 0xD800 and CV2 is ((*cpv* - 0x10000) % 0x400) + 0xDC00.
- Else the CV is the code unit with value *cpv*.

Regular Expressions

Regular expressions are the most important part of ECMAScript that needs to support supplementary characters. Both patterns and the text to be matched have to be interpreted as code points. This provides the following benefits:

- /^.$/ now matches any Unicode code point; in 5.1, it matches only BMP characters.
- Supplementary characters can now be used to define the end points of a range; in 5.1, this results in a syntax error.
- If a Quantifier follows an Atom that consists of a supplementary code point, then it now applies to the complete code point, not just its second surrogate code unit.
- Supplementary characters are now treated as entities in a class; in 5.1, the code units representing them become separate class members.
- Case insensitive matching works for supplementary characters.

Interpreting patterns and input text as sequences of code points may have a compatibility impact. For example, some applications may have processed binary data with regular expressions where neither the “characters” in the patterns nor the input to be matched are text. Others might be surprised that `s.match(/^.$/)[0].length` can now be 2.

To avoid compatibility issues, applications have to request code point interpretation with a "u" (Unicode) flag. This flag also triggers other changes that enable better processing of strings and improved compliance with the Unicode Regular Expressions standard:

- Identity escapes are restricted to escaping the regular expression syntax characters, and the extensions defined in the “match web reality” proposal are not supported. This enables the introduction of escape sequences for Unicode code points (see above), for character classes based on Unicode properties, or for grapheme clusters.
- Unicode code point escape sequences are allowed.
- Case insensitive matching uses case folding as specified in the Unicode standard. The details below only describe simple loose matches, as required for basic Unicode support, but an upgrade to default loose matches would be possible (the latter would enable, for example, /ss/ui to match "ß", LATIN SMALL LETTER SHARP S, whose case-folded form is "ss").
- Possibly the definition of the character classes \d\D\w\W\b\B is extended to their Unicode extensions, such as all characters in the Unicode category “Number, decimal” for \d, as proposed by Steven Levithan. Whether this can be done under the same flag or requires a different one still needs discussion.

This section updates subclause 15.10 of the ECMAScript Language Specification, to define the behavior with the flag set, except for the last item in the list above. It relies on the definitions and text interpretation provided above. For a complete specification, this needs to be combined with the existing behavior, which is still provided when the flag is not set.

Details (skip)

Make the following substitutions throughout sections 15.10 and A7:

- SourceCharacter → SourceCodePoint

- SourceCharacter → SourceCodePoint
- PatternCharacter → PatternCodePoint
- CharacterEscape → CodePointEscape
- CharacterClassEscape → CodePointClassEscape
- CharacterClass → CodePointClass
- CharSet → CodePointSet
- CharacterSetMatcher → CodePointSetMatcher

15.10 RegExp (Regular Expression) Objects

Before the period at the end of the first sentence, insert: ", after preprocessing the pattern as described in 15.10.4.1. When matching against the SourceCodePoint non-terminal, it interprets the pattern String as a sequence of code points, as described in 5.3".

15.10.1 Patterns

Replace the definition of IdentityEscape with:

IdentityEscape :: **one of**
^ $ \ . * + ? ( ) [ ] { } |

At the end of 15.10.1, insert:

SourceCodePoint ::
any code point

15.10.2.1 Notation

Replace the first two bullet items in the first list with:

- *Input* is the String being matched by the regular expression pattern. *Input* is interpreted as a sequence of code points, as described in 5.3. The notation *Input*{*n*} means the code point starting at position *n* of *Input*, where *n* can range between 0 (inclusive) and *InputLength* (exclusive).
- *InputLength* is the number of code units in the *Input* String.

Replace the first bullet item in the second list with:

- A *CodePointSet* is a mathematical set of code points.

In the second bullet item in the same list, replace "the index of the last input character" with "the index of the last code unit of the last input code point".

In the last bullet item in the same list, replace "a character or" with "a code point or", and "a character *ch* means that the escape sequence is interpreted as the character *ch*" with "a code point *cp* means that the escape sequence is interpreted as the code point *cp*".

15.10.2.6 Assertion

Replace "character *Input*[*e*–1]" with "the result of calling CodePointEndingAt with argument *e*-1".

Replace "character *Input*[*e*]" with "code point *Input*{*e*}".

Replace the sentence of "IsWordChar(e-1)" with "IsWordChar(result)" and the sentence of "IsWordChar(e)" with

Replace both occurrences of "IsWordChar(e–1)" with "IsWordChar(e, true)", and both occurrences of "IsWordChar(e)" with "IsWordChar(e, false)".

In the description of IsWordChar, after "an integer parameter *e*" insert "and a boolean parameter *before*".

Replace the first two steps of the IsWordChar algorithm with:

- If *before* is true, then decrease *e* by 1.
- If *e* == –1 or *e* == *InputLength*, return false.
- If *before* is true, then let *c* be the result of calling CodePointEndingAt with argument *e*; else let *c* be the code point *Input*{*e*}.

Add the following abstract operation:

> The abstract operation CodePointEndingAt takes an integer parameter *e*, which must be non-negative and less than *InputLength*, and performs the following:
>
> - If *e* > 0 and *Input*{*e*-1} > U+FFFF then return *Input*{*e*-1}.
> - Else return *Input*{*e*}.

15.10.2.8 Atom

Replace the first two steps of the algorithm for Atom :: PatternCharacter with:

- Let *cp* be the code point represented by *PatternCodePoint*.
- Let *A* be a one-element CodePointSet containing the code point *cp*.

Replace the first step of the algorithm for Atom :: . with:

- Let *A* be the set of all code points except *LineTerminator*.

In step 3.1.4 of the algorithm for Atom :: (Disjunction), replace "characters are the characters" with "code units are the code units".

Replace steps 1.3 and 1.4 of the algorithm for CodePointSetMatcher with:

- Let *cp* be the code point *Input*{*e*}.
- Let *cc* be the result of Canonicalize(*cp*).

Replace step 1.8 of the same algorithm with:

- If *cp* > 0xFFFF then increment *e* by 2; else increment *e* by 1.
- Let *y* be the State (*e*, *cap*).

Replace the abstract operation Canonicalize with:

> The abstract operation Canonicalize takes a code point parameter *cp* and performs the following steps:
>
> - If *IgnoreCase* is false, return *cp*.
> - If the file CaseFolding.txt of the Unicode character database provides a simple or common case folding mapping for *cp*, then return that mapping.
> - Return *cp*.

Replace the last paragraph of note 3 with:

In case-insignificant matches with the Unicode flag set all characters are implicitly case-folded using the simple mapping provided by the Unicode standard immediately before they are compared. The simple mapping always maps to a single code point, so it does not map, for example, “ß” (U+00DF) to “ss”. It may however map a code point outside the Basic Latin range to a character within, for example, “ſ” (U+017F) to “s”, so that a pattern /[a-z]/ui may match code points outside the Basic Latin range.

15.10.2.9 AtomEscape

In the algorithm for AtomEscape :: DecimalEscape, replace step 2 with:

- If *E* is a code point, then
  - Let *cp* be *E*’s code point.
  - Let *A* be a one-element CodePointSet containing the code point *cp*.
  - Call CodePointSetMatcher(*A*, false) and return its Matcher result.

In step 5.8 of the same algorithm, replace “Canonicalize(s[i]) is not the same character as Canonicalize(Input [e+i])” with “Canonicalize(s{i}) is not the same code point as Canonicalize(Input{e+i})”.

Replace the first two steps of the algorithm for AtomEscape :: CodePointEscape with:

- Evaluate *CodePointEscape* to obtain a code point *cp*.
- Let *A* be a one-element CodePointSet containing the code point *cp*.

15.10.2.10 CodePointEscape

In the first paragraph, replace “character” with “code point”. In the following table, replace “Code Unit” with “Code Point”, and replace all occurrences of “\u” with “U+”.

Replace the algorithm for CodePointEscape :: c ControlLetter with:

- Let *cp* be the code point represented by *ControlLetter*.
- Return the remainder of dividing *cp* by 32.

In the remaining three paragraphs, replace “character” with “code point”.

15.10.2.11 DecimalEscape

In step 2 of the algorithm, replace “a <NUL> character (Unicode value 0000)” with “the code point U+0000 (NULL)”.

15.10.2.12 CharacterClassEscape

Replace all occurrences of “characters” with “code points”.

15.10.2.15 NonemptyClassRanges

In step 4 of the algorithm for NonemptyClassRanges :: ClassAtom - ClassAtom ClassRanges, replace “CharacterRange” with “CodePointRange”.

Replace the description of the abstract operation CharacterRange with:

The abstract operation CodePointRange takes two CodePointSet parameters A and B and performs the following:

- If *A* does not contain exactly one code point or *B* does not contain exactly one code point then throw a SyntaxError exception.
- Let *a* be the one code point in CodePointSet *A*.
- Let *b* be the one code point in CodePointSet *B*.
- If *a* > *b* then throw a SyntaxError exception.
- Return the set containing all code points from *a* through *b*, inclusive.

In note 1, 2, and 3, replace all occurrences of “character” with “code point”.

In note 2, replace “ASCII letters” with “letters in the Basic Latin block”.

15.10.2.17 ClassAtom

Replace “character” with “code point”.

15.10.2.18 ClassAtomNoDash

Replace “character” with “code point”.

15.10.2.19 ClassEscape

Replace steps 2-4 of the algorithm for ClassEscape :: DecimalEscape with:

- If *E* is not a code point then throw a SyntaxError exception.
- Let *cp* be *E*’s code point.
- Return the one-element CodePointSet containing the code point *cp*.

Replace “the one character <BS> (Unicode value 0008)” with “the one code point U+0008 (BACKSPACE)”.

Replace all remaining occurrences of “character” with “code point”.

15.10.4.1 new RegExp(pattern, flags)

After the first paragraph of section 15.10.4.1, insert the description of preprocessing steps which remove the need for subsequent processing to worry about Unicode escape sequences while looking for supplementary code points or surrogate code units.

If *F* contains the character "u", then *P* is preprocessed as follows:

- For each Unicode escape sequence consisting of the character “\” followed by a UnicodeCodePointEscapeSequence occurring within the pattern, determine the CV of the UnicodeCodePointEscapeSequence as described in 7.8.4, and replace the escape sequence with a sequence of one or two Unicode code unit escape sequences for the one or two code units of the CV, each starting with “\”.
- For each Unicode escape sequence consisting of the character “\” followed by a UnicodeCodeUnitEscapeSequence, including the ones generated by the step above, determine the CV of the UnicodeCodeUnitEscapeSequence as described in 7.8.4. If the CV is above 0x00FF, replace the escape sequence with the CV. (Unicode code unit escape sequences representing code units up to 0x00FF are left unchanged to avoid confusion with syntactically significant characters.)

In the following paragraph, replace “the characters of P do” with “P interpreted as a code point sequence does”, and “the

characters of P as" with "P interpreted as a code point sequence as".

In the third paragraph, replace ""g", "i", or "m"" with ""g", "i", "m", or "u"".

After the paragraph starting with "The multiline property", insert:

> The unicode property of the newly constructed object is set to a Boolean value that is true if *F* contains the character "u" and false otherwise.

15.10.6.4 RegExp.prototype.toString()

Replace "and "m" if the multiline property is true" with ""m" if the multiline property is true, and "u" if the unicode property is true".

15.10.7.5 unicode

After section 15.10.7.4, insert:

> ### 15.10.7.5 unicode
>
> The value of the unicode property is a Boolean value indicating whether the flags contained the character "u". This property shall have the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

Other Text Processing Functions

As a general principle, all functions in ECMAScript that interpret strings have to recognize supplementary characters as atomic entities and interpret them according to their Unicode semantics. As it turns out, beyond the functions using regular expressions there aren't many in ECMAScript 5.1 that violate this principle: The String case conversion functions (toLowerCase, toLocaleLowerCase, toUpperCase, toLocaleUpperCase) and the relational comparison for strings (11.8.5).

For the relational comparison for strings, stability and compatibility may matter more than semantic correctness – to sort strings for display to the user, you want to use the Collator objects of the upcoming ECMAScript Internationalization API anyway. I'm therefore not proposing to change this operation.

Changing the case conversion functions is theoretically also an incompatible change. However, the only affected code points are those in the Deseret block of Unicode, and it's very unlikely that applications would depend on Deseret characters not being mapped to lower case while calling toLowerCase. Note that Safari is already following the Unicode standard rather than the ECMAScript standard in implementing these functions.

The String.prototype.trim method is updated to allow for white space characters with code points outside the BMP that may be added in the future. The change is compatible because currently (as of Unicode 6.1) no such characters are assigned.

Details (skip)

15.5.4.16 String.prototype.toLowerCase ( )

Before the first sentence, insert: "This method interprets a string as a sequence of code points, as described in 5.3."

Replace step 3 of the algorithm with:

- Let *cpList* be a new List.

- For each code point *scp* in *S*, from beginning to end, do
  - If the Unicode Character Database provides a language insensitive lower case equivalent of *scp*, then append the code points provided to *cpList*; otherwise append *scp* itself.
- Let *L* be the result of calling String.fromCodePoint, providing the code points in *cpList* as arguments.

Remove the paragraph following the algorithm.

Remove “in Unicode 2.1.8 and later”.

In the first sentence of Note 1, replace both occurrences of “characters” with “code points”.

15.5.4.20 String.prototype.trim ( )

Append the following sentence to step 3 of the algorithm: “When determining whether a character is in Unicode general category “Zs”, code unit sequences are interpreted as code point sequences as specified in 5.3.”.

Upgrade to Unicode with Supplementary Characters

The previous sections were narrowly focused on functionality needed to process end user text. This section is the omnibus proposal for upgrading ECMAScript to a Unicode version released in this century, adding supplementary character support to identifiers, and removing the redefinition of characters as code units.

The baseline Unicode version is changed to 5.1, which was released on April 2008 and is the version supported by Windows 7 and the .NET framework 4, probably the oldest platforms on which somebody might implement ECMAScript 6 (implementations on older platforms would have to bring their own tables for identifier characters and case conversion). Unicode 5.1 includes supplementary characters, so all restrictions to the Basic Multilingual Plane are removed, along with the one and only reference to UCS-2 in the specification.

Identifiers are specified differently in 5.1 than in previous Unicode versions; the proposed definition of ECMAScript identifiers aligns more closely with the Unicode specification, and allows supplementary characters in identifiers. Unicode escapes in identifiers are handled as a separate preprocessing step to avoid additional complexity in the IdentifierName grammar.

The ECMAScript Language Specification, 5.1 edition, redefines characters as code units, creating endless confusion. This proposal removes most uses of “character” and uses the appropriate choice of “code unit” or “code point” instead.

In order to maintain compatibility with existing ECMAScript, the proposal does not change the representation of source text or String values. There’s more on this topic in the What About UTF-32? section below. Also for compatibility, ill-formed UTF-16 code sequences and surrogate code points are allowed.

Details (skip)

Global Substitutions

Replace any occurrence of the word “character” with “code unit”, and any occurrence of the word “characters” with “code units”, throughout the document, except in the following situations:

- A different change is described in this document.
- The occurrence is in one of the following sections:
  - 7.1 Unicode Format-Control Characters

    - 7.2 White Space
    - 7.3 Line Terminators
    - 15.10 RegExp (RegularExpression) Objects
    - A.6 Universal Resource Identifier Character Classes
    - A.7 Regular Expressions
- The word is used to refer to specific assigned code points in the Basic Multilingual Plane, where the distinction between code units and code points is irrelevant, e.g. “the comma character”.

Make the following substitutions throughout the document:

- SourceCharacter → SourceCodeUnit (note that a different substitution has already been applied to occurrences of SourceCharacter in sections 15.10 and A7)
- UnicodeEscapeSequence → UnicodeCodeUnitEscapeSequence
- DoubleStringCharacters → DoubleStringCodeUnits
- DoubleStringCharacter → DoubleStringCodeUnit
- SingleStringCharacters → SingleStringCodeUnits
- SingleStringCharacter → SingleStringCodeUnit
- CharacterEscapeSequence → CodeUnitEscapeSequence
- NonEscapeCharacter → NonEscapeCodeUnit
- JSONStringCharacters → JSONStringCodeUnits
- JSONStringCharacter → JSONStringCodeUnit

Replace any occurrence of “\u” within a table under the column heading “code unit value” with “0x”. “\u” is a notation for code unit values in a few places in source text; this shouldn’t be confused with code unit values themselves.

2 Conformance

Replace the second paragraph of clause 2, Conformance, with:

> A conforming implementation of this Standard shall interpret characters in conformance with the Unicode Standard, Version 5.1.0 or later and ISO/IEC 10646 with UTF-16 as the adopted encoding form. If the adopted ISO/IEC 10646 subset is not otherwise specified, it is presumed to be the Unicode set, collection 10646.

3 Normative References

Replace the second entry of clause 3, Normative references, with references to Unicode 5.1 and the corresponding ISO 10646 version.

> ISO/IEC 10646:2003: Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus Amendment 1:2005, Amendment 2:2006, Amendment 3:2008, and Amendment 4:2008, plus additional amendments and corrigenda, or successor
>
> The Unicode Standard, Version 5.0, as amended by Unicode 5.1.0, or successor
>
> Unicode Standard Annex #15, Unicode Normalization Forms, version Unicode 5.1.0, or successor
>
> Unicode Standard Annex #31, Unicode Identifiers and Pattern Syntax, version Unicode 5.1.0, or successor.

5.1.2 The Lexical and RegExp Grammars

In the first paragraph, replace “characters (Unicode code units)” with “code units”.

5.1.6 Grammar Notation

Replace the last sentence of the first paragraph with:

> All terminal symbol characters specified in this way are to be understood as the UTF-16 code units for the appropriate Unicode characters from the Basic Latin block, as opposed to any similar-looking characters from other Unicode blocks.

In the last paragraph, replace “any Unicode code unit” with “any code unit”.

6 Source Text

In clause 6, Source Text, replace the paragraphs before the one starting with “In string literals” with:

> ECMAScript source text is assumed to be a (not necessarily well-formed) sequence of UTF-16 code units for the purposes of this specification. Source text encoded in other character encodings than UTF-16 must be processed as if it was first converted to UTF-16. The text is expected to have been normalised to Unicode Normalization Form C (Canonical Decomposition, followed by Canonical Composition), as described in Unicode Standard Annex #15. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text, themselves. The UTF-16 code unit sequence will be interpreted as Unicode code points, as described in 5.3, and according to the character properties defined by the Unicode Standard, version 5.1.0 or later.
>
> **Syntax**
>
> *SourceCodeUnit* ::
> any code unit

Replace the last two paragraphs of the clause with:

> NOTE: ECMAScript differs from the Java programming language in the behaviour of Unicode escape sequences. In a Java program, if the Unicode escape sequence \u000A, for example, occurs within a single-line comment, it is interpreted as a line terminator (Unicode character U+000A is line feed) and therefore the next character is not part of the comment. Similarly, if the Unicode escape sequence \u000A occurs within a string literal in a Java program, it is likewise interpreted as a line terminator, which is not allowed within a string literal—one must write \n instead of \u000A to cause a line feed to be part of the string value of a string literal. In an ECMAScript program, a Unicode escape sequence occurring within a comment is never interpreted and therefore cannot contribute to termination of the comment. Similarly, a Unicode escape sequence occurring within a string literal in an ECMAScript program always contributes code units to the String value of the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.
>
> While parsing ECMAScript source code, code unit sequences are commonly interpreted as code point sequences, e.g., to determine whether a code unit sequence can form an identifier. The interpretation is as described in section 5.3.

7.2 White Space

Replace “Unicode 3.0” with “Unicode 5.1.0”. Append to the same paragraph: “When supporting a Unicode version that includes characters in general category ‘Zs’ with code points above U+FFFF, the code unit sequence must be interpreted as a code point sequence as described in 5.3 when checking for white space.”.

7.6 Identifier Names and Identifiers

Replace the first three paragraphs of 7.6 with:

> Identifier Names are tokens that are interpreted according to the Default Identifier Syntax given in Unicode Standard Annex #31, Identifier and Pattern Syntax, with some small modifications. The Unicode identifier grammar is based on character properties specified by the Unicode Standard. The code points in the specified categories in version 5.1.0 of the Unicode standard must be treated as in those categories by all conforming ECMAScript implementations.
>
> This standard specifies specific character additions: The dollar sign (U+0024) and the underscore (U+005F) are permitted anywhere in an IdentifierName, and the characters zero width non-joiner (U+200C) and zero width joiner (U+200D) can be used after the first code point.
>
> While scanning identifiers, any Unicode escape sequences consisting of the character "\" followed by either a UnicodeCodeUnitEscapeSequence or a UnicodeCodePointSequence in the source text are replaced by the CV of the UnicodeCodeUnitEscapeSequence or UnicodeCodePointSequence (see 7.8.4). The resulting code unit sequence is then interpreted as a sequence of code points, as described in 5.3.
>
> An Identifier is an IdentifierName that is not a ReservedWord (see 7.6.1).

Replace the fifth paragraph of 7.6 with:

> ECMAScript implementations may recognise identifier code points defined in later editions of the Unicode Standard. If portability is a concern, programmers should only employ identifier code points defined in Unicode 5.1.0.

Replace the productions starting with the one for IdentifierStart with:

IdentifierStart ::
UnicodeIDStart
$
_

IdentifierPart ::
UnicodeIDContinue
$
_
<ZWNJ>
<ZWJ>

UnicodeIDStart ::
any code point with the Unicode property "ID_Start"

UnicodeOtherIDContinue ::
any code point with the Unicode property "ID_Continue"

7.8.3 Numeric Literals

Replace the sentence starting with "The source character immediately following..." with "The source code unit sequence immediately following a NumericLiteral must not start with an IdentifierStart or DecimalDigit."

7.8.4 String Literals

Replace the first paragraph with:

> A string literal is zero or more code units enclosed in the code units representing single (0x0027) or double (0x0022) quotes. Each code unit may be represented by an escape sequence, or a pair of surrogate code units may be represented by a Unicode code point escape sequence. All code units may appear literally in a string literal except for the code units for the closing quote character, backslash (0x005C), carriage return (0x000D), line feed (0x000A), line separator (0x2028), and paragraph separator (0x2029). Any code unit may appear in the form of an escape sequence.

Replace “is a <NUL> character (Unicode value 0000)” with “is the code unit 0x0000, which represents the character NULL”.

Replace “is the character whose code unit value” with “is the code unit whose value”.

8.4 The String Type

In the first paragraph, remove “(see Clause 6)”. There’s nothing relevant to see there.

Replace the second paragraph with:

> Where ECMAScript operations interpret String contents, each element is interpreted as a single UTF-16 code unit. However, ECMAScript does not place any restrictions or requirements on the sequence of code units in a String value, so they may be ill-formed when interpreted as UTF-16 code unit sequences. Operations that do not interpret String contents treat them as sequences of undifferentiated 16-bit unsigned integers. No operations ensure that Strings are in normalized form. Only operations that are explicitly specified to be language or locale sensitive produce language-sensitive results.

11.8.5 The Abstract Relational Comparison Algorithm

Replace both occurrences of “integer that is the code unit value for the character” with “value of the code unit”. This leaves the algorithm using code unit semantics for compatibility with older versions of ECMAScript. For this kind of algorithm, stability is probably more important than semantic correctness.

15.1.3 URI Handling Function Properties

Replace “in Surrogates, section 3.7, of the Unicode Standard” with “in the description of UTF-16 in section 3.9, Unicode Encoding Forms, of the Unicode Standard”.

15.5.4.5 String.prototype.charCodeAt

Replace both occurrences of “the code unit value of the character” with “the value of the code unit”.

15.12.1.1 The JSON Lexical Grammar

Leave “the characters ‘JSON’” as is.

15.12.3 stringify (value [, replacer [, space]])

Replace step 2.c of the algorithm for the abstract operation Quote with:

- Else if C < 0x0020

Leave the word “character” in step 8.b.iii.1 of the abstract operation JO as is, but fix the preceding word.

In note 3, after “Control characters” insert “in the range U+0000 to U+001F”.

Code Point Based String Accessors

The following proposed functions would make it easier for developers to implement code point based functionality. They mirror the functionality of String.fromCharCode and String.prototype.charCodeAt. It’s unfortunately not possible to extend the existing functions to handle code points: fromCharCode currently coerces larger values into 16 bits by setting all unneeded bits to 0, and callers passing in larger values might be confused if their interpretation changed. charCodeAt is specified to return a value below 0x10000, and callers may not be able to handle larger values.

15.5.3.2 String.fromCodePoint ([cp0 [, cp1 [, ...]]])

Returns a String value containing as many code units as necessary to represent the code points given by the arguments. Each argument specifies one code point to be represented in the resulting String, with the first argument specifying the first code point, and so on, from beginning to end. The function behaves as if it were defined as:

```
String.fromCodePoint = function () {
    var chars = [], i;
    for (i = 0; i < arguments.length; i++) {
        var c = Number(arguments[i]);
        if (!isFinite(c) || c < 0 || c > 0x10FFFF || Math.floor(c) !== c) {
            throw new RangeError("Invalid code point " + c);
        }
        if (c < 0x10000) {
            chars.push(c);
        } else {
            c -= 0x10000;
            chars.push((c >> 10) + 0xD800);
            chars.push((c % 0x400) + 0xDC00);
        }
    }
    return String.fromCharCode.apply(undefined, chars);
};
```

The length property of the fromCodePoint function is 1.

15.5.4.6 String.prototype.codePointAt (pos)

Returns a Number (a nonnegative integer less than or equal to 0x10FFFF) representing the code point value of the code unit sequence starting at position *pos* in the String resulting from converting this object to a String. If there is no code unit at that position, the result is undefined. The function behaves as if it were defined as:

```
String.prototype.codePointAt = function (index) {
    var str = String(this);
    if (index < 0 || index >= str.length) {
        return undefined;
```

```
    }
    var first = str.charCodeAt(index);
    if (first >= 0xD800 && first <= 0xDBFF && str.length > index + 1) {
        var second = str.charCodeAt(index + 1);
        if (second >= 0xDC00 && second <= 0xDFFF) {
            return ((first - 0xD800) << 10) + (second - 0xDC00) + 0x10000;
        }
    }
    return first;
};
```

NOTE: The codePointAt function is intentionally generic; it does not require that its this value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

15.5.4.7 String.prototype.[iterator]

Returns an iterator that iterates over the code points (not code units) of a String value, returning each code point as a String value. The function behaves as if it were defined as:

```
String.prototype.[iterator] = function() {
    var s = this;
    return {
        index: 0,
        next: function() {
            if (this.index >= s.length) {
                throw StopIteration;
            }
            let cp = s.codePointAt(index);
            index += cp > 0xFFFF ? 2 : 1;
            return String.fromCodePoint(cp);
        }
    }
}
```

What About UTF-32?

There have been several proposals to change ECMAScript to use Unicode code points directly to represent source code and String values – since code points need at least 21 bits, and computers don't have 21-bit integers, this effectively means UTF-32. By eliminating the duality code unit / code point, this would make text processing in ECMAScript easier to understand, and it would slightly simplify the work of developers implementing low-level string processing in ECMAScript.

However, this change would also create serious compatibility issues. Take the string "吉野家" (Yoshinoya), the name of a Japanese beef bowl restaurant chain. Since the first character is a supplementary character, the length of the string in a current ECMAScript implementation is 4, and the code points start at positions 0, 2, and 3. In a UTF-32 based implementation, the length of the string would be 3, and the code points start at positions 0, 1, and 2. The two systems, and libraries and applications running on them, could not easily exchange length or position information about this string. Similarly, positions in ECMAScript strings would no longer match DOM offsets for the same strings, because DOM string offsets (e.g., in the CharacterData interface) are UTF-16 based. Maybe somebody can come up with a solution that solves these issue so that developers don't have to worry about them, but I haven't seen it yet (previous discussions on the ECMAScript mailing listed started here and here, continued here).

continued here).

In addition, code points are in many cases not the right abstraction either for text processing. Many operations have to operate on grapheme clusters, the substrings that a user would perceive as a character.

To really help developers, the focus shouldn't be on access to individual code points. It should be on more and better functions to process text at higher levels of abstractions. Regular expressions with support for Unicode properties and grapheme clusters would be an excellent start.

Updates

Since 2012-03-22, based on feedback on the ECMAScript mailing list and from the TC 39 meeting on 2012-03-29:

- Added definitions of surrogate code unit and surrogate pair.
- Integrated Unicode code point escapes in string literals, regular expression patterns, and identifiers into the proposal.
- Required the "u" flag for code point semantics in regular expressions, but also made it the "little red switch" that enables comprehensive Unicode support. Within this mode:
  - Restricted identity escapes to syntax characters.
  - Excluded application of web reality proposal.
  - Changed case insensitive matching to use simple Unicode case folding.
- Removed compatibility hack in regular expressions because non-"u" mode covers compatibility.
- Added specification changes for String.prototype.trim and other white space handling.
- Changed the default String iterator to return a String value.
- Fixed description of result value of String.prototype.codePointAt and some positions in the "What about UTF-32" section.

Since the original version of 2012-03-15, based on feedback on the ECMAScript mailing list:

- Restricted the second transformation of workaround regular expressions to the case where the second range is exactly [\uDC00-\uDFFF].
- Indicated that "u" may not be the actual character for the flag for code point mode in regular expressions, as a "u" flag has already been proposed for Unicode-aware digit and word character matching. Added that code point mode can also be enabled by containment within Harmony modules.
- Proposed the elimination of special cases in case insensitive comparison in code point mode.
- Made codePointAt return undefined rather than NaN when given an out-of-range *index*.
- Added String code point iterator based on the Harmony iterator proposal.
