Darwin Language Specification

Version 1.0
Draft 1

Paul Vick

Table of Contents

1. Introduction

Darwin is a general purpose programming language for .NET Core.

1.1 Grammar Notation

This specification specifies a lexical and a syntatic grammar. Each grammar is defined using a set of *rules*. Each rule starts with a name followed by a colon and then a set of *productions*, one per line. The productions of a rule define the different possible ways to match the rule. For example the following:

```
expression
    : literal
    | expression operator expression
    ;
```

defines a rule *expression* that either matches a single *literal* or an *expression* followed by an *operator* and then another *expression*. The rule named *start* is the start rule for a grammar.

A production contains a set of sequential *terms*. A term can be one of the following:

- **A string.** Strings are specified in quotes ('`if`') and match the given sequence of Unicode characters.
- **A meta-string.** A meta-string is specified by angle brackets and describes the string matched informally. (For example, "< the Unicode value U+0000 >".)
- **A rule name.** A rule name is specified in *italics* and matches the rule named.
- **A group.** A list of terms can be contained by parenthesis ("()") to allow multiple terms to be modified by repetition operators.
- **A range.** A range of character values can be specified by separating them with two dots ("..").

Terms can be modified with the repetition operators "?", "+", and "*". The operator "?" indicates a term is optional and can appear zero or one times. The operator "+" indicates a term must appear one or more times. And the operator "*" indicates a term may appear zero or more times.

> **Note:** The grammars in this specification are not intended to be formal grammars (that is, usable by any particular parser or lexer generator).

2. Lexical Grammar

The first step in processing Darwin code is to translate a stream of Unicode characters into an ordered set of lexical tokens.

Note: All references to "Unicode" in this specification refer to Unicode version 9.0.

start
: *token**
;

token
: *whitespace*
| *comment*
| *identifier*
| *literal*
| *punctuator*
| *operator*
;

2.1 Whitespace

Whitespace serves to separate tokens but has no other significance in the language. Whitespace is defined as any character with the Unicode property `White_Space`. A *line terminator* is whitespace that marks the lexical end of a line.

whitespace
: `<Any Unicode character with the property 'White_Space' except a line-terminator>`
| *line-terminator*
;

line-terminator
: `<Carriage return character (U+000D)>`
| `<Line feed character (U+000A)>`
| `<Carriage return character (U+000D)>` `<Line feed character (U+000A)>`
;

2.2 Comments

Comments are text that serve as comments on the source code and are treated as whitespace. A comment extends only until the next line terminator.

comment
: '`#`' *comment-element**
;

comment-element
: `<Any Unicode character except a line-terminator.>`
;

2.3 Identifiers

An *identifier* is a name. Darwin identifiers conform to the Unicode Standard Annex 31, version 9.0. The language includes the following requirements from that specification:

- **UAX31-R1**: Darwin specifies a profile that includes the underscore (_) character in the `Start` set. It also removes the `Other_ID_Start` characters from `Start` and `Other_ID_Continue` characters from `Continue`, as backwards compatibility with previous Unicode specifications is not a concern.
- **UAX31-R4**: Darwin normalizes identifiers using normalization form NFC.

Two things are worth noting regarding Darwin identifiers. First, unlike some languages, Unicode formatting characters are not allowed in identifiers to reduce the possibility of confusion between identical looking identifiers. Second, Darwin defines no reserved keywords.

identifier
: *identifier-start* *identifier-character*?
;

identifier-start
: *letter-character*
| '_'
;

identifier-character
: *connector-character*
| *letter-character*
| *decimal-character*
| *combining-character*
;

letter-character
: `<Unicode alphabetic character (classes Lu, Ll, Lt, Lm, Lo, Nl)>`
;

decimal-character
: `<Unicode numeric character (class Nd)>`
;

combining-character
: `<Unicode mark character (classes Mn, Mc)>`
;

connector-character
: `<Unicode connection character (class Pc)>`
;

2.4 Literals

A *literal* is a textual representation of a value.

literal
: *integer-literal*
| *floating-point-literal*
| *string-literal*
| *domain-specific-literal*
;

2.4.1 Integer Literals

An *integer literal* is a textual representation of an integral numeric value. Integer literals can either be specified in decimal (base 10) or hexadecimal (base 16) notation. Hexadecimal notation is prefixed by `0x` and uses the letters `a` through `f` to represent the additional digit values.

```
integer-literal
    : decimal-literal
    | hexadecimal-literal
    ;

decimal-literal
    : digit+
    ;

digit
    : '0'..'9'
    ;

hexadecimal-literal
    : '0x' hexadecimal-digit+
    ;

hexadecimal-digit
    : '0'..'9'
    | 'a'..'f'
    | 'A'..'F'
    ;
```

2.4.2 Floating-Point Literals

A *floating-point literal* is a textual representation of a real numeric value.

```
floating-point-literal
    : digit+ '.' digit+ exponent?
    | digit+ exponent
    ;

exponent
    : exponent-character sign? digit+
    ;

exponent-character
    : 'e'
    | 'E'
    ;

sign
    : '+'
    | '-'
    ;
```

2.4.3 String Literals

A *string literal* is a textual representation of a string value. String literals are delimited by double quotes (`"`), although double quotes can be represented in the string literal by two double quotes in a row (i.e. the string literal `"""Hello, world!"" she said."` represents the string `"Hello, world!" she said.`).

String literals support *expression holes*. An expression hole is delimited by a dollar sign followed by curly braces (`${}`) and contains a Darwin expression. The behavior of expression holes is described later in this specification.

string-literal
: '"' *string-literal-element** '"'
;

string-literal-element
: *string-literal-character*
| *string-literal-hole*
;

string-literal-character
: <Any character except a double quote, open brace, or close brace.>
| '"' '"'
;

string-literal-hole

;

2.4.4 Domain-Specific Literals

A *domain-specific literal* is a literal whose language is not part of the Darwin language. For example, an XML literal or a JSON literal could be expressed using a domain-specific literal. Domain specific literals are delimited by backticks (`) and can contain any Unicode character, including line terminators. Inside of a domain-specific literal, a backtick can be represented by two backticks (``).

Like string literals, domain-specific literals support expression holes. An expression hole is delimited by a dollar sign followed by curly braces (`${}`) and contains a Darwin expression. The behavior of expression holes is described later in this specification.

domain-specific-literal
: '`' *domain-specific-literal-element*\* '`'
;

domain-specific-literal-element
: *domain-specific-literal-character*
| *domain-specific-literal-hole*
;

domain-specific-literal-character
: <Any character except a back tick (`).>
| '`' '`'
;

domain-specific-literal-hole

;

2.5 Punctuators

A *punctuator* serves to delimit syntatic structures.

Note: Some operators are also used as punctuators in the syntactic language.

punctuator
: '('
| ')'
| '{'
| '}'
| '['
| ']'
| ','
| ';'
| ':'
;

2.6 Operators

An *operator* specifies an operation in the language.

operator
: *operators*+
;

operators
: '~'
| '!'
| '@'
| '$'
| '%'
| '^'
| '&'
| '*'
| '-'
| '+'
| '='
| '\'
| '|'
| '<'
| '>'
| '.'
| '?'
| '/'
;