LEXICAL ELEMENTS AND GRAMMAR OF C#

A C# **program** consists of one or more **source files**, known formally as **compilation units**(Compilation units). A source file is an ordered sequence of Unicode characters. Source files typically have a one-to-one correspondence with files in a file system, but this correspondence is not required. For maximal portability, it is recommended that files in a file system be encoded with the UTF-8 encoding.
Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.

2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.

3. Syntactic analysis, which translates the stream of tokens into executable code.

Grammars

This specification presents the syntax of the C# programming language using two grammars. The **lexical grammar** (Lexical grammar) defines how Unicode characters are combined to form line terminators, white space, comments, tokens, and pre-processing directives. The **syntactic grammar**(Syntactic grammar) defines how the tokens resulting from the lexical grammar are combined to form C# programs.

Grammar notation

The lexical and syntactic grammars are presented in Backus-Naur form using the notation of the ANTLR grammar tool.

Lexical grammar

The lexical grammar of C# is presented in Lexical analysis, Tokens, and Pre-processing directives. The terminal symbols of the lexical grammar are the characters of the Unicode character set,

and the lexical grammar specifies how characters are combined to form tokens (Tokens), white space (White space), comments (Comments), and pre-processing directives (Pre-processing directives).
Every source file in a C# program must conform to the input production of the lexical grammar (Lexical analysis).

Syntactic grammar

The syntactic grammar of C# is presented in the chapters and appendices that follow this chapter. The terminal symbols of the syntactic grammar are the tokens defined by the lexical grammar, and the syntactic grammar specifies how tokens are combined to form C# programs.
Every source file in a C# program must conform to the compilation_unit production of the syntactic grammar (Compilation units).

Lexical analysis

The input production defines the lexical structure of a C# source file. Each source file in a C# program must conform to this lexical grammar production.

```
input
    : input_section?
    ;

input_section
    : input_section_part+
    ;

input_section_part
    : input_element* new_line
    | pp_directive
    ;

input_element
    : whitespace
    | comment
    | token
    ;
```

Five basic elements make up the lexical structure of a C# source file: Line terminators (Line terminators), white space (White space), comments (Comments), tokens (Tokens), and pre-processing directives (Pre-processing directives). Of these basic elements, only tokens are significant in the syntactic grammar of a C# program (Syntactic grammar).

The lexical processing of a C# source file consists of reducing the file into a sequence of tokens which becomes the input to the syntactic analysis. Line terminators, white space, and comments can serve to separate tokens, and pre-processing directives can cause sections of the source file to be skipped, but otherwise these lexical elements have no impact on the syntactic structure of a C# program.

In the case of interpolated string literals (Interpolated string literals) a single token is initially produced by lexical analysis, but is broken up into several input elements which are repeatedly subjected to lexical analysis until all interpolated string literals have been resolved. The resulting tokens then serve as input to the syntactic analysis.

When several lexical grammar productions match a sequence of characters in a source file, the lexical processing always forms the longest possible lexical element. For example, the character sequence // is processed as the beginning of a single-line comment because that lexical element is longer than a single / token.

Line terminators

Line terminators divide the characters of a C# source file into lines.

```
new_line
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Carriage return character (U+000D) followed by line feed
character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;
```

For compatibility with source code editing tools that add end-of-file markers, and to enable a source file to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to every source file in a C# program:

- If the last character of the source file is a Control-Z character (U+001A), this character is deleted.

- A carriage-return character (U+000D) is added to the end of the source file if that source file is non-empty and if the last character of the source file is not a carriage return (U+000D), a line feed (U+000A), a line separator (U+2028), or a paragraph separator (U+2029).

Comments

Two forms of comments are supported: single-line comments and delimited comments.**Single-line comments** start with the characters //and extend to the end of the source line. **Delimited comments** start with the characters /* and end with the characters */. Delimited comments may span multiple lines.

```
comment
    : single_line_comment
    | delimited_comment
    ;

single_line_comment
    : '//' input_character*
    ;

input_character
    : '<Any Unicode character except a new_line_character>'
    ;

new_line_character
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;

delimited_comment
    : '/*' delimited_comment_section* asterisk* '/'
    ;

delimited_comment_section
    : '/'
    | asterisk* not_slash_or_asterisk
    ;

asterisk
    : '*'
    ;

not_slash_or_asterisk
```

```
    : '<Any Unicode character except / or *>'
    ;
```

Comments do not nest. The character sequences/* and */have no special meaning within a//comment, and the character sequences // and/*have no special meaning within a delimited comment.
Comments are not processed within character and string literals.

The example

C#Copy

```
/* Hello, world program
   This program writes "hello, world" to the console
*/
class Hello
{
    static void Main() {
        System.Console.WriteLine("hello, world");
    }
}
```

includes a delimited comment.

The example

C#Copy

```
// Hello, world program
// This program writes "hello, world" to the console
//
class Hello // any name will do for this class
{
    static void Main() { // this method must be named "Main"
        System.Console.WriteLine("hello, world");
    }
}
```

shows several single-line comments.

White space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.

```
whitespace
    : '<Any character with Unicode class Zs>'
    | '<Horizontal tab character (U+0009)>'
    | '<Vertical tab character (U+000B)>'
    | '<Form feed character (U+000C)>'
```

```
    ;
```

Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

```
token
    : identifier
    | keyword
    | integer_literal
    | real_literal
    | character_literal
    | string_literal
    | interpolated_string_literal
    | operator_or_punctuator
    ;
```

Unicode character escape sequences

A Unicode character escape sequence represents a Unicode character. Unicode character escape sequences are processed in identifiers (Identifiers), character literals (Character literals), and regular string literals (String literals). A Unicode character escape is not processed in any other location (for example, to form an operator, punctuator, or keyword).

```
unicode_escape_sequence
    : '\\u' hex_digit hex_digit hex_digit hex_digit
    | '\\U' hex_digit hex_digit hex_digit hex_digit hex_digit
hex_digit hex_digit hex_digit
    ;
```

A Unicode escape sequence represents the single Unicode character formed by the hexadecimal number following the "\u" or "\U" characters. Since C# uses a 16-bit encoding of Unicode code points in characters and string values, a Unicode character in the range U+10000 to U+10FFFF is not permitted in a character literal and is represented using a Unicode surrogate pair in a string literal. Unicode characters with code points above 0x10FFFF are not supported.
Multiple translations are not performed. For instance, the string literal "\u005Cu005C" is equivalent to "\u005C" rather than "\". The Unicode value \u005C is the character "\".
The example

C#Copy

```
class Class1
{
    static void Test(bool \u0066) {
        char c = '\u0066';
        if (\u0066)
            System.Console.WriteLine(c.ToString());
    }
}
```

shows several uses of \u0066, which is the escape sequence for the letter "f". The program is equivalent to

C#Copy

```
class Class1
{
    static void Test(bool f) {
        char c = 'f';
        if (f)
            System.Console.WriteLine(c.ToString());
    }
}
```

Identifiers

The rules for identifiers given in this section correspond exactly to those recommended by the Unicode Standard Annex 31, except that underscore is allowed as an initial character (as is traditional in the C programming language), Unicode escape sequences are permitted in identifiers, and the "@" character is allowed as a prefix to enable keywords to be used as identifiers.

```
identifier
    : available_identifier
    | '@' identifier_or_keyword
    ;

available_identifier
    : '<An identifier_or_keyword that is not a keyword>'
    ;

identifier_or_keyword
    : identifier_start_character identifier_part_character*
    ;

identifier_start_character
    : letter_character
    | '_'
```

```
    ;

identifier_part_character
    : letter_character
    | decimal_digit_character
    | connecting_character
    | combining_character
    | formatting_character
    ;

letter_character
    : '<A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl>'
    | '<A unicode_escape_sequence representing a character of
classes Lu, Ll, Lt, Lm, Lo, or Nl>'
    ;

combining_character
    : '<A Unicode character of classes Mn or Mc>'
    | '<A unicode_escape_sequence representing a character of
classes Mn or Mc>'
    ;

decimal_digit_character
    : '<A Unicode character of the class Nd>'
    | '<A unicode_escape_sequence representing a character of the
class Nd>'
    ;

connecting_character
    : '<A Unicode character of the class Pc>'
    | '<A unicode_escape_sequence representing a character of the
class Pc>'
    ;

formatting_character
    : '<A Unicode character of the class Cf>'
    | '<A unicode_escape_sequence representing a character of the
class Cf>'
    ;
```

For information on the Unicode character classes mentioned above, see The Unicode Standard, Version 3.0, section 4.5.

Examples of valid identifiers include "identifier1", "_identifier2", and "@if".

An identifier in a conforming program must be in the canonical format defined by Unicode Normalization Form C, as defined by Unicode Standard Annex 15. The behavior when encountering an identifier not in Normalization Form C is implementation-defined; however, a diagnostic is not required.

The prefix "@" enables the use of keywords as identifiers, which is useful when interfacing with other programming languages. The character @ is not actually part of the identifier, so the identifier might be seen in other languages as a normal identifier, without the prefix. An identifier with an @ prefix is called a **verbatim identifier**. Use of the @ prefix for identifiers that are not keywords is permitted, but strongly discouraged as a matter of style.
The example:

C#Copy

```
class @class
{
    public static void @static(bool @bool) {
        if (@bool)
            System.Console.WriteLine("true");
        else
            System.Console.WriteLine("false");
    }
}

class Class1
{
    static void M() {
        cl\u0061ss.st\u0061tic(true);
    }
}
```

defines a class named "class" with a static method named "static" that takes a parameter named "bool". Note that since Unicode escapes are not permitted in keywords, the token "cl\u0061ss" is an identifier, and is the same identifier as "@class".
Two identifiers are considered the same if they are identical after the following transformations are applied, in order:

- The prefix "@", if used, is removed.
- Each unicode_escape_sequence is transformed into its corresponding Unicode character.
- Any formatting_characters are removed.

Identifiers containing two consecutive underscore characters (U+005F) are reserved for use by the implementation. For example, an implementation might provide extended keywords that begin with two underscores.

Keywords

A **keyword** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the @ character.

```
keyword
    : 'abstract' | 'as'       | 'base'       | 'bool'      |
'break'
    | 'byte'     | 'case'     | 'catch'      | 'char'      |
'checked'
    | 'class'    | 'const'    | 'continue'   | 'decimal'   |
'default'
    | 'delegate' | 'do'       | 'double'     | 'else'      |
'enum'
    | 'event'    | 'explicit' | 'extern'     | 'false'     |
'finally'
    | 'fixed'    | 'float'    | 'for'        | 'foreach'   |
'goto'
    | 'if'       | 'implicit' | 'in'         | 'int'       |
'interface'
    | 'internal' | 'is'       | 'lock'       | 'long'      |
'namespace'
    | 'new'      | 'null'     | 'object'     | 'operator'  | 'out'
    | 'override' | 'params'   | 'private'    | 'protected' |
'public'
    | 'readonly' | 'ref'      | 'return'     | 'sbyte'     |
'sealed'
    | 'short'    | 'sizeof'   | 'stackalloc' | 'static'    |
'string'
    | 'struct'   | 'switch'   | 'this'       | 'throw'     |
'true'
    | 'try'      | 'typeof'   | 'uint'       | 'ulong'     |
'unchecked'
    | 'unsafe'   | 'ushort'   | 'using'      | 'virtual'   |
'void'
    | 'volatile' | 'while'
    ;
```

In some places in the grammar, specific identifiers have special meaning, but are not keywords. Such identifiers are sometimes referred to as "contextual keywords". For example, within a property declaration, the "get" and "set" identifiers have special meaning (Accessors). An identifier other than get or set is never permitted in these locations, so this use does not conflict with a use of these words as identifiers. In other cases, such as with the identifier "var" in implicitly typed local variable declarations (Local variable declarations), a contextual keyword can conflict with declared names. In such cases, the declared name takes precedence over the use of the identifier as a contextual keyword.

Literals

A **literal** is a source code representation of a value.

```
literal
    : boolean_literal
    | integer_literal
    | real_literal
    | character_literal
    | string_literal
    | null_literal
    ;
```

Boolean literals

There are two boolean literal values: true and false.

```
boolean_literal
    : 'true'
    | 'false'
    ;
```

The type of a boolean_literal is bool.

Integer literals

Integer literals are used to write values of types int, uint, long, and ulong. Integer literals have two possible forms: decimal and hexadecimal.

```
integer_literal
    : decimal_integer_literal
    | hexadecimal_integer_literal
    ;

decimal_integer_literal
    : decimal_digit+ integer_type_suffix?
    ;

decimal_digit
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    ;
```

```
integer_type_suffix
    : 'U' | 'u' | 'L' | 'l' | 'UL' | 'Ul' | 'uL' | 'ul' | 'LU' |
'Lu' | 'lU' | 'lu'
    ;

hexadecimal_integer_literal
    : '0x' hex_digit+ integer_type_suffix?
    | '0X' hex_digit+ integer_type_suffix?
    ;

hex_digit
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' |
'e' | 'f';
```

The type of an integer literal is determined as follows:

- If the literal has no suffix, it has the first of these types in which its value can be represented: int, uint,long,ulong.
- If the literal is suffixed by U or u, it has the first of these types in which its value can be represented: uint,ulong.
- If the literal is suffixed by L or l, it has the first of these types in which its value can be represented: long,ulong.
- If the literal is suffixed by UL, Ul, uL, ul, LU, Lu, lU, or lu, it is of type ulong.

If the value represented by an integer literal is outside the range of the ulong type, a compile-time error occurs.
As a matter of style, it is suggested that "L" be used instead of "l" when writing literals of type long, since it is easy to confuse the letter "l" with the digit "1".
To permit the smallest possible int and long values to be written as decimal integer literals, the following two rules exist:

- When a decimal_integer_literal with the value 2147483648 (2^31) and no integer_type_suffixappears as the token immediately following a unary minus operator token (Unary minus operator), the result is a constant of type in with the value -2147483648 (-2^31). In all other situations, such a decimal_integer_literal is of type uint.
- When a decimal_integer_literal with the value 9223372036854775808 (2^63) and no integer_type_suffix or the integer_type_suffix L o l appears as the token immediately following a unary minus operator token (Unary minus operator), the result is a constant of type longwith the value -9223372036854775808 (-2^63). In all other situations, such a decimal_integer_literal is of type ulong.

Real literals

Real literals are used to write values of types float, double, and decimal.

```
real_literal
    : decimal_digit+ '.' decimal_digit+ exponent_part?
real_type_suffix?
    | '.' decimal_digit+ exponent_part? real_type_suffix?
    | decimal_digit+ exponent_part real_type_suffix?
    | decimal_digit+ real_type_suffix
    ;

exponent_part
    : 'e' sign? decimal_digit+
    | 'E' sign? decimal_digit+
    ;

sign
    : '+'
    | '-'
    ;

real_type_suffix
    : 'F' | 'f' | 'D' | 'd' | 'M' | 'm'
    ;
```

If no real_type_suffix is specified, the type of the real literal is double. Otherwise, the real type suffix determines the type of the real literal, as follows:

- A real literal suffixed by F or f is of type float. For example, the literals 1f, 1.5f, 1e10f, and 123.456F are all of type float.
- A real literal suffixed by D or d is of type double. For example, the literals 1d, 1.5d, 1e10d, and 123.456D are all of type double.
- A real literal suffixed byMormis of typedecimal. For example, the literals1m,1.5m,1e10m, and123.456Mare all of typedecimal. This literal is converted to adecimalvalue by taking the exact value, and, if necessary, rounding to the nearest representable value using banker's rounding (The decimal type). Any scale apparent in the literal is preserved unless the value is rounded or the value is zero (in which latter case the sign and scale will be 0). Hence, the literal2.900mwill be parsed to form the decimal with sign0, coefficient2900, and scale3.

If the specified literal cannot be represented in the indicated type, a compile-time error occurs.
The value of a real literal of typefloatordoubleis determined by using the IEEE "round to nearest" mode.
Note that in a real literal, decimal digits are always required after the decimal point. For example,1.3Fis a real literal but1.Fis not.

Character literals

A character literal represents a single character, and usually consists of a character in quotes, as in'a'.
Note: The ANTLR grammar notation makes the following confusing! In ANTLR, when you write\'it stands for a single quote'. And when you write\\it stands for a single backslash\. Therefore the first rule for a character literal means it starts with a single quote, then a character, then a single quote. And the eleven possible simple escape sequences are\',\",\\,\0,\a,\b,\f,\n,\r,\t,\v.

```
character_literal
    : '\'' character '\''
    ;

character
    : single_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    ;

single_character
    : '<Any character except \' (U+0027), \\ (U+005C), and
new_line_character>'
    ;

simple_escape_sequence
    : '\\\'' | '\\"' | '\\\\' | '\\0' | '\\a' | '\\b' | '\\f' |
'\\n' | '\\r' | '\\t' | '\\v'
    ;

hexadecimal_escape_sequence
    : '\\x' hex_digit hex_digit? hex_digit? hex_digit?;
```

A character that follows a backslash character (\) in acharactermust be one of the following characters:',",\,0,a,b,f,n,r,t,u,U,x,v. Otherwise, a compile-time error occurs.

A hexadecimal escape sequence represents a single Unicode character, with the value formed by the hexadecimal number following "\x".
If the value represented by a character literal is greater thanU+FFFF, a compile-time error occurs.
A Unicode character escape sequence (Unicode character escape sequences) in a character literal must be in the rangeU+0000toU+FFFF.
A simple escape sequence represents a Unicode character encoding, as described in the table below.

Escape sequence	**Character name**	**Unicode encoding**
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

The type of acharacter_literalischar.

String literals

C# supports two forms of string literals:**regular string literals**and**verbatim string literals.**

A regular string literal consists of zero or more characters enclosed in double quotes, as in"hello", and may include both simple escape sequences (such as\tfor the tab character), and hexadecimal and Unicode escape sequences.
A verbatim string literal consists of an@character followed by a double-quote character, zero or more characters, and a closing double-quote character. A simple example is@"hello". In a verbatim string literal, the characters between the delimiters are interpreted verbatim, the only exception being aquote_escape_sequence. In particular, simple escape sequences, and hexadecimal and Unicode escape sequences are not processed in verbatim string literals. A verbatim string literal may span multiple lines.

```
string_literal
    : regular_string_literal
```

```
    | verbatim_string_literal
    ;

regular_string_literal
    : '"' regular_string_literal_character* '"'
    ;

regular_string_literal_character
    : single_regular_string_literal_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    ;

single_regular_string_literal_character
    : '<Any character except " (U+0022), \\ (U+005C), and
new_line_character>'
    ;

verbatim_string_literal
    : '@"' verbatim_string_literal_character* '"'
    ;

verbatim_string_literal_character
    : single_verbatim_string_literal_character
    | quote_escape_sequence
    ;

single_verbatim_string_literal_character
    : '<any character except ">'
    ;

quote_escape_sequence
    : '""'
    ;
```

A character that follows a backslash character (\) in aregular_string_literal_charactermust be one of the following characters:',",\,0,a,b,f,n,r,t,u,U,x,v. Otherwise, a compile-time error occurs.

The example

C#Copy

```
string a = "hello, world";                  // hello, world
string b = @"hello, world";                 // hello, world

string c = "hello \t world";                // hello      world
string d = @"hello \t world";               // hello \t world
```

```
string e = "Joe said \"Hello\" to me";       // Joe said "Hello" 
to me
string f = @"Joe said ""Hello"" to me";      // Joe said "Hello" 
to me

string g = "\\\\server\\share\\file.txt";    // 
\\server\share\file.txt
string h = @"\\server\share\file.txt";       // 
\\server\share\file.txt

string i = "one\r\ntwo\r\nthree";
string j = @"one
two
three";
```

shows a variety of string literals. The last string literal,j, is a verbatim string literal that spans multiple lines. The characters between the quotation marks, including white space such as new line characters, are preserved verbatim.
Since a hexadecimal escape sequence can have a variable number of hex digits, the string literal"\x123"contains a single character with hex value 123. To create a string containing the character with hex value 12 followed by the character 3, one could write"\x00123"or"\x12" + "3"instead.
The type of astring_literalisstring.
Each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator (String equality operators) appear in the same program, these string literals refer to the same string instance. For instance, the output produced by
C#Copy

```
class Test
{
    static void Main() {
        object a = "hello";
        object b = "hello";
        System.Console.WriteLine(a == b);
    }
}
```

isTruebecause the two literals refer to the same string instance.

Interpolated string literals

Interpolated string literals are similar to string literals, but contain holes delimited by{and}, wherein expressions can occur. At runtime, the expressions are evaluated with the purpose of having their textual forms substituted into the string at the place where

the hole occurs. The syntax and semantics of string interpolation are described in section (Interpolated strings).
Like string literals, interpolated string literals can be either regular or verbatim. Interpolated regular string literals are delimited by$"and", and interpolated verbatim string literals are delimited by$@"and".
Like other literals, lexical analysis of an interpolated string literal initially results in a single token, as per the grammar below. However, before syntactic analysis, the single token of an interpolated string literal is broken into several tokens for the parts of the string enclosing the holes, and the input elements occurring in the holes are lexically analysed again. This may in turn produce more interpolated string literals to be processed, but, if lexically correct, will eventually lead to a sequence of tokens for syntactic analysis to process.

```
interpolated_string_literal
    : '$' interpolated_regular_string_literal
    | '$' interpolated_verbatim_string_literal
    ;

interpolated_regular_string_literal
    : interpolated_regular_string_whole
    | interpolated_regular_string_start
interpolated_regular_string_literal_body
interpolated_regular_string_end
    ;

interpolated_regular_string_literal_body
    : regular_balanced_text
    | interpolated_regular_string_literal_body
interpolated_regular_string_mid regular_balanced_text
    ;

interpolated_regular_string_whole
    : '"' interpolated_regular_string_character* '"'
    ;

interpolated_regular_string_start
    : '"' interpolated_regular_string_character* '{'
    ;

interpolated_regular_string_mid
    : interpolation_format? '}'
interpolated_regular_string_characters_after_brace? '{'
    ;

interpolated_regular_string_end
```

```
    : interpolation_format? '}'
interpolated_regular_string_characters_after_brace? '"'
    ;

interpolated_regular_string_characters_after_brace
    : interpolated_regular_string_character_no_brace
    | interpolated_regular_string_characters_after_brace
interpolated_regular_string_character
    ;

interpolated_regular_string_character
    : single_interpolated_regular_string_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    | open_brace_escape_sequence
    | close_brace_escape_sequence
    ;

interpolated_regular_string_character_no_brace
    : '<Any interpolated_regular_string_character except
close_brace_escape_sequence and any hexadecimal_escape_sequence or
unicode_escape_sequence designating } (U+007D)>'
    ;

single_interpolated_regular_string_character
    : '<Any character except \" (U+0022), \\ (U+005C), { (U+007B),
} (U+007D), and new_line_character>'
    ;

open_brace_escape_sequence
    : '{{'
    ;

    close_brace_escape_sequence
    : '}}'
    ;

regular_balanced_text
    : regular_balanced_text_part+
    ;

regular_balanced_text_part
    : single_regular_balanced_text_character
    | delimited_comment
    | '@' identifier_or_keyword
    | string_literal
    | interpolated_string_literal
    | '(' regular_balanced_text ')'
    | '[' regular_balanced_text ']'
```

```
    | '{' regular_balanced_text '}'
    ;

single_regular_balanced_text_character
    : '<Any character except / (U+002F), @ (U+0040), \" (U+0022),
$ (U+0024), ( (U+0028), ) (U+0029), [ (U+005B), ] (U+005D),
{ (U+007B), } (U+007D) and new_line_character>'
    | '</ (U+002F), if not directly followed by / (U+002F) or *
(U+002A)>'
    ;

interpolation_format
    : interpolation_format_character+
    ;

interpolation_format_character
    : '<Any character except \" (U+0022), : (U+003A), { (U+007B)
and } (U+007D)>'
    ;

interpolated_verbatim_string_literal
    : interpolated_verbatim_string_whole
    | interpolated_verbatim_string_start
interpolated_verbatim_string_literal_body
interpolated_verbatim_string_end
    ;

interpolated_verbatim_string_literal_body
    : verbatim_balanced_text
    | interpolated_verbatim_string_literal_body
interpolated_verbatim_string_mid verbatim_balanced_text
    ;

interpolated_verbatim_string_whole
    : '@"' interpolated_verbatim_string_character* '"'
    ;

interpolated_verbatim_string_start
    : '@"' interpolated_verbatim_string_character* '{'
    ;

interpolated_verbatim_string_mid
    : interpolation_format? '}'
interpolated_verbatim_string_characters_after_brace? '{'
    ;

interpolated_verbatim_string_end
    : interpolation_format? '}'
interpolated_verbatim_string_characters_after_brace? '"'
    ;
```

```
interpolated_verbatim_string_characters_after_brace
    : interpolated_verbatim_string_character_no_brace
    | interpolated_verbatim_string_characters_after_brace
interpolated_verbatim_string_character
    ;

interpolated_verbatim_string_character
    : single_interpolated_verbatim_string_character
    | quote_escape_sequence
    | open_brace_escape_sequence
    | close_brace_escape_sequence
    ;

interpolated_verbatim_string_character_no_brace
    : '<Any interpolated_verbatim_string_character except
close_brace_escape_sequence>'
    ;

single_interpolated_verbatim_string_character
    : '<Any character except \" (U+0022), { (U+007B) and }
(U+007D)>'
    ;

verbatim_balanced_text
    : verbatim_balanced_text_part+
    ;

verbatim_balanced_text_part
    : single_verbatim_balanced_text_character
    | comment
    | '@' identifier_or_keyword
    | string_literal
    | interpolated_string_literal
    | '(' verbatim_balanced_text ')'
    | '[' verbatim_balanced_text ']'
    | '{' verbatim_balanced_text '}'
    ;

single_verbatim_balanced_text_character
    : '<Any character except / (U+002F), @ (U+0040), \" (U+0022),
$ (U+0024), ( (U+0028), ) (U+0029), [ (U+005B), ] (U+005D),
{ (U+007B) and } (U+007D)>'
    | '</ (U+002F), if not directly followed by / (U+002F) or *
(U+002A)>'
    ;
```

Aninterpolated_string_literaltoken is reinterpreted as multiple tokens and other input elements as follows, in order of occurrence in theinterpolated_string_literal:

- Occurrences of the following are reinterpreted as separate individual tokens: the leading$sign,interpolated_regular_string_whole,interpolated_regular_string_start,interpolated_regular_string_mid,interpolated_regular_string_end,interpolated_verbatim_string_whole,interpolated_verbatim_string_start,interpolated_verbatim_string_midandinterpolated_verbatim_string_end.
- Occurrences ofregular_balanced_textandverbatim_balanced_textbetween these are reprocessed as aninput_section(Lexical analysis) and are reinterpreted as the resulting sequence of input elements. These may in turn include interpolated string literal tokens to be reinterpreted.

Syntactic analysis will recombine the tokens into aninterpolated_string_expression(Interpolated strings).
Examples TODO

The null literal

```
null_literal
    : 'null'
    ;
```

Thenull_literalcan be implicitly converted to a reference type or nullable type.

Operators and punctuators

There are several kinds of operators and punctuators. Operators are used in expressions to describe operations involving one or more operands. For example, the expressiona + buses the+operator to add the two operandsaandb. Punctuators are for grouping and separating.

```
operator_or_punctuator
    : '{'  | '}'  | '['  | ']'  | '('   | ')'  | '.'  | ','  | ':'
| ';'
    | '+'  | '-'  | '*'  | '/'  | '%'   | '&'  | '|'  | '^'  | '!'
| '~'
    | '='  | '<'  | '>'  | '?'  | '??'  | '::' | '++' | '--' |
'&&' | '||'
    | '->' | '==' | '!=' | '<=' | '>='  | '+=' | '-=' | '*=' |
'/=' | '%='
```

```
    | '&=' | '|=' | '^=' | '<<' | '<<=' | '=>'
    ;

right_shift
    : '>>'
    ;

right_shift_assignment
    : '>>='
    ;
```

The vertical bar in theright_shiftandright_shift_assignmentproductions are used to indicate that, unlike other productions in the syntactic grammar, no characters of any kind (not even whitespace) are allowed between the tokens. These productions are treated specially in order to enable the correct handling oftype_parameter_lists (Type parameters).

Pre-processing directives

The pre-processing directives provide the ability to conditionally skip sections of source files, to report error and warning conditions, and to delineate distinct regions of source code. The term "pre-processing directives" is used only for consistency with the C and C++ programming languages. In C#, there is no separate pre-processing step; pre-processing directives are processed as part of the lexical analysis phase.

```
pp_directive
    : pp_declaration
    | pp_conditional
    | pp_line
    | pp_diagnostic
    | pp_region
    | pp_pragma
    ;
```

The following pre-processing directives are available:

- #defineand#undef, which are used to define and undefine, respectively, conditional compilation symbols (Declaration directives).

- #if,#elif,#else, and#endif, which are used to conditionally skip sections of source code (Conditional compilation directives).
- #line, which is used to control line numbers emitted for errors and warnings (Line directives).
- #errorand#warning, which are used to issue errors and warnings, respectively (Diagnostic directives).
- #regionand#endregion, which are used to explicitly mark sections of source code (Region directives).
- #pragma, which is used to specify optional contextual information to the compiler (Pragma directives).

A pre-processing directive always occupies a separate line of source code and always begins with a#character and a pre-processing directive name. White space may occur before the#character and between the#character and the directive name. A source line containing a#define,#undef,#if,#elif,#else,#endif,#line, or#endregiondirective may end with a single-line comment. Delimited comments (the/* */style of comments) are not permitted on source lines containing pre-processing directives. Pre-processing directives are not tokens and are not part of the syntactic grammar of C#. However, pre-processing directives can be used to include or exclude sequences of tokens and can in that way affect the meaning of a C# program. For example, when compiled, the program:

C#Copy

```
#define A
#undef B

class C
{
#if A
    void F() {}
#else
    void G() {}
#endif

#if B
    void H() {}
#else
    void I() {}
#endif
}
```

results in the exact same sequence of tokens as the program:
C#Copy

```
class C
{
    void F() {}
    void I() {}
}
```

Thus, whereas lexically, the two programs are quite different, syntactically, they are identical.

Conditional compilation symbols

The conditional compilation functionality provided by the#if,#elif,#else, and#endifdirectives is controlled through pre-processing expressions (Pre-processing expressions) and conditional compilation symbols.

```
conditional_symbol
    : '<Any identifier_or_keyword except true or false>'
    ;
```

A conditional compilation symbol has two possible states:**defined**or**undefined**. At the beginning of the lexical processing of a source file, a conditional compilation symbol is undefined unless it has been explicitly defined by an external mechanism (such as a command-line compiler option). When a#definedirective is processed, the conditional compilation symbol named in that directive becomes defined in that source file. The symbol remains defined until an#undefdirective for that same symbol is processed, or until the end of the source file is reached. An implication of this is that#defineand#undefdirectives in one source file have no effect on other source files in the same program.

When referenced in a pre-processing expression, a defined conditional compilation symbol has the boolean valuetrue, and an undefined conditional compilation symbol has the boolean valuefalse. There is no requirement that conditional compilation symbols be explicitly declared before they are referenced in pre-processing expressions. Instead, undeclared symbols are simply undefined and thus have the valuefalse.

The name space for conditional compilation symbols is distinct and separate from all other named entities in a C# program. Conditional compilation symbols can only be referenced in#defineand#undefdirectives and in pre-processing expressions.

Pre-processing expressions

Pre-processing expressions can occur in#ifand#elifdirectives. The operators!,==,!=,&&and||are permitted in pre-processing expressions, and parentheses may be used for grouping.

```
pp_expression
    : whitespace? pp_or_expression whitespace?
    ;

pp_or_expression
    : pp_and_expression
    | pp_or_expression whitespace? '||' whitespace?
pp_and_expression
    ;

pp_and_expression
    : pp_equality_expression
    | pp_and_expression whitespace? '&&' whitespace?
pp_equality_expression
    ;

pp_equality_expression
    : pp_unary_expression
    | pp_equality_expression whitespace? '==' whitespace?
pp_unary_expression
    | pp_equality_expression whitespace? '!=' whitespace?
pp_unary_expression
    ;

pp_unary_expression
    : pp_primary_expression
    | '!' whitespace? pp_unary_expression
    ;

pp_primary_expression
    : 'true'
    | 'false'
    | conditional_symbol
    | '(' whitespace? pp_expression whitespace? ')'
    ;
```

When referenced in a pre-processing expression, a defined conditional compilation symbol has the boolean valuetrue, and an undefined conditional compilation symbol has the boolean valuefalse.
Evaluation of a pre-processing expression always yields a boolean value. The rules of evaluation for a pre-processing expression are the same as those for a constant expression (Constant

expressions), except that the only user-defined entities that can be referenced are conditional compilation symbols.

Declaration directives

The declaration directives are used to define or undefine conditional compilation symbols.

```
pp_declaration
    : whitespace? '#' whitespace? 'define' whitespace
conditional_symbol pp_new_line
    | whitespace? '#' whitespace? 'undef' whitespace
conditional_symbol pp_new_line
    ;

pp_new_line
    : whitespace? single_line_comment? new_line
    ;
```

The processing of a#definedirective causes the given conditional compilation symbol to become defined, starting with the source line that follows the directive. Likewise, the processing of an#undefdirective causes the given conditional compilation symbol to become undefined, starting with the source line that follows the directive.
Any#defineand#undefdirectives in a source file must occur before the firsttoken(Tokens) in the source file; otherwise a compile-time error occurs. In intuitive terms,#defineand#undefdirectives must precede any "real code" in the source file.
The example:

C#Copy

```
#define Enterprise

#if Professional || Enterprise
    #define Advanced
#endif

namespace Megacorp.Data
{
    #if Advanced
    class PivotTable {...}
    #endif
}
```

is valid because the#definedirectives precede the first token (thenamespacekeyword) in the source file.

The following example results in a compile-time error because a#definefollows real code:
C#Copy

```
#define A
namespace N
{
    #define B
    #if B
    class Class1 {}
    #endif
}
```

A#definemay define a conditional compilation symbol that is already defined, without there being any intervening#undeffor that symbol. The example below defines a conditional compilation symbolAand then defines it again.
C#Copy

```
#define A
#define A
```

A#undefmay "undefine" a conditional compilation symbol that is not defined. The example below defines a conditional compilation symbolAand then undefines it twice; although the second#undefhas no effect, it is still valid.
C#Copy

```
#define A
#undef A
#undef A
```

Conditional compilation directives

The conditional compilation directives are used to conditionally include or exclude portions of a source file.

```
pp_conditional
    : pp_if_section pp_elif_section* pp_else_section? pp_endif
    ;

pp_if_section
    : whitespace? '#' whitespace? 'if' whitespace pp_expression
pp_new_line conditional_section?
    ;

pp_elif_section
    : whitespace? '#' whitespace? 'elif' whitespace pp_expression
pp_new_line conditional_section?
```

```
    ;

pp_else_section:
    | whitespace? '#' whitespace? 'else' pp_new_line
conditional_section?
    ;

pp_endif
    : whitespace? '#' whitespace? 'endif' pp_new_line
    ;

conditional_section
    : input_section
    | skipped_section
    ;

skipped_section
    : skipped_section_part+
    ;

skipped_section_part
    : skipped_characters? new_line
    | pp_directive
    ;

skipped_characters
    : whitespace? not_number_sign input_character*
    ;

not_number_sign
    : '<Any input_character except #>'
    ;
```

As indicated by the syntax, conditional compilation directives must be written as sets consisting of, in order, an#ifdirective, zero or more#elifdirectives, zero or one#elsedirective, and an#endifdirective. Between the directives are conditional sections of source code. Each section is controlled by the immediately preceding directive. A conditional section may itself contain nested conditional compilation directives provided these directives form complete sets.
App_conditionalselects at most one of the containedconditional_sections for normal lexical processing:

- Thepp_expressions of the#ifand#elifdirectives are evaluated in order until one yieldstrue. If an expression yieldstrue, theconditional_sectionof the corresponding directive is selected.

- If allpp_expressions yieldfalse, and if an#elsedirective is present, theconditional_sectionof the#elsedirective is selected.
- Otherwise, noconditional_sectionis selected.

The selectedconditional_section, if any, is processed as a normalinput_section: the source code contained in the section must adhere to the lexical grammar; tokens are generated from the source code in the section; and pre-processing directives in the section have the prescribed effects.

The remainingconditional_sections, if any, are processed asskipped_sections: except for pre-processing directives, the source code in the section need not adhere to the lexical grammar; no tokens are generated from the source code in the section; and pre-processing directives in the section must be lexically correct but are not otherwise processed. Within aconditional_sectionthat is being processed as askipped_section, any nestedconditional_sections (contained in nested#if...#endifand#region...#endregionconstructs) are also processed asskipped_sections.

The following example illustrates how conditional compilation directives can nest:

C#Copy

```
#define Debug       // Debugging on
#undef Trace        // Tracing off

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
            #if Trace
                WriteToLog(this.ToString());
            #endif
        #endif
        CommitHelper();
    }
}
```

Except for pre-processing directives, skipped source code is not subject to lexical analysis. For example, the following is valid despite the unterminated comment in the#elsesection:

C#Copy

```
#define Debug        // Debugging on

class PurchaseTransaction
```

```
{
    void Commit() {
        #if Debug
            CheckConsistency();
        #else
            /* Do something else
        #endif
    }
}
```

Note, however, that pre-processing directives are required to be lexically correct even in skipped sections of source code.
Pre-processing directives are not processed when they appear inside multi-line input elements. For example, the program:
C#Copy

```
class Hello
{
    static void Main() {
        System.Console.WriteLine(@"hello,
#if Debug
        world
#else
        Nebraska
#endif
        ");
    }
}
```

results in the output:
Copy

```
hello,
#if Debug
        world
#else
        Nebraska
#endif
```

In peculiar cases, the set of pre-processing directives that is processed might depend on the evaluation of thepp_expression. The example:
C#Copy

```
#if X
    /*
#else
    /* */ class Q { }
#endif
```

always produces the same token stream (classQ{}), regardless of whether or notXis defined. IfXis defined, the only processed directives are#ifand#endif, due to the multi-line comment. IfXis undefined, then three directives (#if,#else,#endif) are part of the directive set.

Diagnostic directives

The diagnostic directives are used to explicitly generate error and warning messages that are reported in the same way as other compile-time errors and warnings.

```
pp_diagnostic
    : whitespace? '#' whitespace? 'error' pp_message
    | whitespace? '#' whitespace? 'warning' pp_message
    ;

pp_message
    : new_line
    | whitespace input_character* new_line
    ;
```

The example:

C#Copy

```
#warning Code review needed before check-in

#if Debug && Retail
    #error A build can't be both debug and retail
#endif

class Test {...}
```

always produces a warning ("Code review needed before check-in"), and produces a compile-time error ("A build can't be both debug and retail") if the conditional symbols Debug andRetailare both defined. Note that app_messag can contain arbitrary text; specifically, it need not contain well-formed tokens, as shown by the single quote in the wordcan't.

Region directives

The region directives are used to explicitly mark regions of source code.

```
pp_region
```

```
    : pp_start_region conditional_section? pp_end_region
    ;

pp_start_region
    : whitespace? '#' whitespace? 'region' pp_message
    ;

pp_end_region
    : whitespace? '#' whitespace? 'endregion' pp_message
    ;
```

No semantic meaning is attached to a region; regions are intended for use by the programmer or by automated tools to mark a section of source code. The message specified in a #region or #endregion directive likewise has no semantic meaning; it merely serves to identify the region. Matching #region and #endregion directives may have different pp_messages.
The lexical processing of a region:

C#Copy

```
#region
...
#endregion
```

corresponds exactly to the lexical processing of a conditional compilation directive of the form:

C#Copy

```
#if true
...
#endif
```

Line directives

Line directives may be used to alter the line numbers and source file names that are reported by the compiler in output such as warnings and errors, and that are used by caller info attributes (Caller info attributes).
Line directives are most commonly used in meta-programming tools that generate C# source code from some other text input.

```
pp_line
    : whitespace? '#' whitespace? 'line' whitespace line_indicator
pp_new_line
    ;

line_indicator
    : decimal_digit+ whitespace file_name
```

```
    | decimal_digit+
    | 'default'
    | 'hidden'
    ;

file_name
    : '"' file_name_character+ '"'
    ;

file_name_character
    : '<Any input_character except ">'
    ;
```

When no #line directives are present, the compiler reports true line numbers and source file names in its output. When processing a #line directive that includes a line_indicator that is not default, the compiler treats the line after the directive as having the given line number (and file name, if specified).
A #line default directive reverses the effect of all preceding #line directives. The compiler reports true line information for subsequent lines, precisely as if no #line directives had been processed.
A #line hidden directive has no effect on the file and line numbers reported in error messages, but does affect source level debugging. When debugging, all lines between a #line hiddendirective and the subsequent #line directive (that is not #line hidden) have no line number information. When stepping through code in the debugger, these lines will be skipped entirely.
Note that a file_name differs from a regular string literal in that escape characters are not processed; the "\" character simply designates an ordinary backslash character within a file_name.

Pragma directives

The #pragma preprocessing directive is used to specify optional contextual information to the compiler. The information supplied in a #pragma directive will never change program semantics.

```
pp_pragma
    : whitespace? '#' whitespace? 'pragma' whitespace pragma_body
pp_new_line
    ;

pragma_body
    : pragma_warning_body
    ;
```

C# provides #pragma directives to control compiler warnings. Future versions of the language may include additional #pragma directives. To ensure interoperability with other C# compilers, the Microsoft C# compiler does not issue compilation errors for unknown #pragma directives; such directives do however generate warnings.

Pragma warning

The #pragma warning directive is used to disable or restore all or a particular set of warning messages during compilation of the subsequent program text.

```
pragma_warning_body
    : 'warning' whitespace warning_action
    | 'warning' whitespace warning_action whitespace warning_list
    ;

warning_action
    : 'disable'
    | 'restore'
    ;

warning_list
    : decimal_digit+ (whitespace? ',' whitespace? decimal_digit+)*
    ;
```

A #pragma warning directive that omits the warning list affects all warnings. A #pragma warningdirective the includes a warning list affects only those warnings that are specified in the list.
A #pragma warning disable directive disables all or the given set of warnings.
A #pragma warning restore directive restores all or the given set of warnings to the state that was in effect at the beginning of the compilation unit. Note that if a particular warning was disabled externally, a #pragma warning restore (whether for all or the specific warning) will not re-enable that warning.
The following example shows use of #pragma warning to temporarily disable the warning reported when obsoleted members are referenced, using the warning number from the Microsoft C# compiler.
C#Copy

```
using System;

class Program
{
    [Obsolete]
    static void Foo() {}
```

```
    static void Main() {
#pragma warning disable 612
    Foo();
#pragma warning restore 612
    }
}
```

ORTHOGONALITY:

In computer programming, orthogonality means that operations change just one thing without affecting others. The term is most-frequently used regarding assembly instruction sets, as orthogonal instruction set. Orthogonality in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language. It is associated with simplicity; the more orthogonal the design, the fewer exceptions. This makes it easier to learn, read and write programs in a programming language. The meaning of an orthogonal feature is independent of context; the key parameters are symmetry and consistency (for example, a pointer is an orthogonal concept).

C:

The C language is somewhat inconsistent in its treatment of concepts and language structure, making it difficult for the user to learn (and use) the language. Examples of exceptions follow:

(a)Structures (but not arrays) may be returned from a function.

(b)An array can be returned if it is inside a structure.

(c)A member of a structure can be any data type (except void, or the structure of the same type).

(d)An array element can be any data type (except void). Everything is passed by value (except arrays).

(e)Void can be used as a type in a structure, but a variable of this type cannot be declared in a function.

Lack of Orthogonality in C++

```
classtype o_class;

classtype o_class(1, 2);

//classtype o_class(); preserve orthogonality
```

The above code demonstrates a lack of orthogonality in the c++ language. This lack of orthogonality is shown when comparing the different syntax for declaring objects of classtype. In the first statement, the declaration of o_class does not use the optional parameter list by omitting the parentheses. Compare this with the second statement which uses the parameter list. The commented code shows how the C++ would handle declaration of default declaration if it preserved orthogonality.

JAVA:

It happens that Log4j, a popular open source logging package for Java, is a good example of a modular design based on orthogonality.

The dimensions of Log4j:

Logging is just a fancier version of the System.out.println() statement, and Log4j is a utility package that abstracts the mechanics of logging on the Java platform. Among other things, Log4j features allow developers to do the following:

(a)Log to different appenders (not only the console but also to files, network locations, relational databases, operating system log utilities, and more)

(b)Log at several levels (such as ERROR, WARN, INFO, and DEBUG)

(c)Centrally control how much information is logged at a given logging level
(d)Use different layouts to define how a logging event is rendered into a string.

While Log4j does have other features, I will focus on these three dimensions of its functionality in order to explore the concept and benefits of orthogonality. Note that my discussion is based on Log4j version 1.2.17.

Considering Log4j types as aspects:

Appenders, level, and layout are three aspects of Log4j that can be seen as independent dimensions. I use the term aspect here as a synonym for concern, meaning a piece of interest or focus in a program. In this case, it is easy to define these three concerns based on the questions that each addresses:

(a)Appender: Where should the log event data be sent for display or storage?

(b)Layout: How should a log event be presented?

(c)Level: Which log events should be processed?

UNREADABLE STATEMENTS:

PYTHON:

```
def remap((path,data)):
     lineData = {}
     for wf,lines in data:
          for ln in lines: lineData[ln] = lineData.get(ln,set())
               lineData[ln].add(wf) return (path,lineData)
               covMap = dict(map(remap, allCovData.items()))
```

C++ & C:

```
int(*name(args)) (args); /*function returning pointer to
function*/
```

```
char (*(*f( )) [ ]) ( ); /* f is a function returning a pointer to
array [ ] of pointer to a function returning char */
```

Java:

```
new Thread( () -> System.out.println("In Java8, Lambda expression
rocks !!") ).start(); //Lambda expressions
```

```java
//Anonymous inner class
class AnonymousDemo {
     public static void main(String[] args) {
     Age oj1 = new Age() {
          @Override
               public void getAge() {
                    System.out.print("Age is "+x);
               }
     };
     oj1.getAge();
 }
}
```

CHARACTER SETS OF PROGRAMMING LANGUAGES:

Java – UTF8

C++ - ASCII and Unicode

C – ANCI C

Python – UTF18 and UCS8