5.3 Informative Comments

This section provides a short description of each ABNF rule listed above. The related brief and long syntax rules are grouped together with the same description. Where the syntaxes are the same, the rule is listed once and preceded with the text "BS/LS". Where the brief and long syntaxes are different, both rules are listed separately and preceded with "BS" and "LS" respectively.

BS/LS: **expressionConstraint** = ws (refinedExpressionConstraint / compoundExpressionConstraint / dottedExpressionConstraint / subExpressionConstraint) ws	
	An expression constraint is either a refined expression constraint, a compound expression constraint, a dotted expression constraint, or a sub expression constraint.
BS/LS: **refinedExpressionConstraint** = subExpressionConstraint ws ":" ws eclRefinement	
	A refined expression constraint includes a subexpression constraint followed by a refinement.
BS/LS: **compoundExpressionConstraint** = conjunctionExpressionConstraint / disjunctionExpressionConstraint / exclusionExpressionConstraint	
	A compound expression constraint contains two or more expression constraints joined by either a conjunction, disjunction or exclusion. When potential ambiguity in binary operator precedence may occur, round brackets must be used to clearly disambiguate the order in which these operator are applied. Brackets are not required in expression constraints in which all binary operators are conjunctions, or all binary operators are disjunctions. Please note that unary operators (i.e. constraint operators and member of functions) are always applied before binary operators (i.e. conjunction, disjunction and exclusion).
BS/LS: **conjunctionExpressionConstraint** = subExpressionConstraint 1*(ws conjunction ws subExpressionConstraint)	
	A conjunction expression constraint combines two or more expression constraints with a conjunction ("and") operator. More than one conjunction may be used without brackets. However any compound expression constraint (using a different binary operator) that appears within a conjunction expression constraint must be enclosed by brackets.
BS/LS: **disjunctionExpressionConstraint** = subExpressionConstraint 1*(ws disjunction ws subExpressionConstraint)	
	A disjunction expression constraint combines two or more expression constraints with a disjunction ("or") operator. More than one disjunction may be used without brackets. However any compound expression constraint (using a different binary operator) that appears within a disjunction expression constraint must be enclosed by brackets.
BS/LS: **exclusionExpressionConstraint** = subExpressionConstraint ws exclusion ws subExpressionConstraint	
	An exclusion expression constraint combines two expression constrains with an exclusion ("minus") operator. A single exclusion operator may be used without brackets. However when the operands of the exclusion expression constraint are compound, these compound expression constraints must be enclosed by brackets.
BS/LS: **dottedExpressionConstraint** = subExpressionConstraint 1*(ws dottedExpressionAttribute)	
	A dotted expression constraint contains a sub expression constraint, followed by one or more dotted attributes. When a single dotted attribute is used, the result is the set of attribute values (for the given attribute name) of each concept that results from evaluating the subExpressionConstraint. When more than one dotted attribute is used, each dottedExpressionAttribute is sequentially evaluated (from left to right) against the given result set.

BS/LS: **dottedExpressionAttribute** = dot ws eclAttributeName	
	A dotted expression attribute consists of a 'dot', followed by an attribute name. Please note that the attribute name may be represented by any sub expression constraint.
BS/LS: **subExpressionConstraint** = [constraintOperator ws] [memberOf ws] (eclFocusConcept / "(" ws expressionConstraint ws ")")	
	A sub expression constraint optionally begins with a constraint operator and/or a memberOf function. It then includes either a single focus concept or an expression constraint (enclosed in brackets). A memberOf function should be used only when the eclFocusConcept or expressionConstraint refers to a reference set concept, a set of reference set concepts, or a wild card. When both a constraintOperator and a memberOf function are used, they are applied from the inside to out (i.e. from right to left) - see 5.4 Operator Precedence. Therefore, if a constraintOperator is followed by a memberOf function, then the memberOf function is processed prior to the constraintOperator.
BS/LS: **eclFocusConcept** = eclConceptReference / wildCard	
	A focus concept is a concept reference or a wild card.
BS/LS: **dot** = "."	
	A dot connects an expression constraint with an attribute whose values are included in the result.
BS: **memberOf** = "^" **LS:** **memberOf** = "^" / ("m"/"M") ("e"/"E") ("m"/"M") ("b"/"B") ("e"/"E") ("r"/"R") ("o"/"O") ("f"/"F")	
	The 'memberOf' function returns the set of referenced components in the reference set whose concept identifier follows. In the brief syntax, the memberOf function is represented using the "^" symbol. In the long syntax, the text "memberOf " (case insensitive and followed by at least one white space) is also allowed.
BS/LS: **eclConceptReference** = conceptId [ws "|" ws term ws "|"]	
	A conceptReference is represented by a ConceptId, optionally followed by a term enclosed by a pair of "|" characters. Whitespace before or after the ConceptId is ignored as is any whitespace between the initial "|" characters and the first non-whitespace character in the term or between the last non-whitespace character and before second "|" character.
BS/LS: **conceptId** = sctId	
	The ConceptId must be a valid SNOMED CT identifier for a concept. The initial digit may not be zero. The smallest number of digits is six, and the maximum is 18.
BS/LS: **term** = 1*nonwsnonpipe *(1*SP 1*nonwsnonpipe)	
	The term must be the term from a SNOMED CT description that is associated with the concept identified by the preceding concept identifier. For example, the term could be the preferred description, or the preferred description associated with a particular translation. The term may include valid UTF-8 characters except for the pipe "
BS: **wildCard** = "*" **LS:** **wildCard** = "*" / (("a"/"A") ("n"/"N") ("y"/"Y"))	
	A wild card represents any concept in the given substrate. In the brief syntax, a wildcard is represented using the "*" symbol. In the long syntax, the text "ANY" (case insensitive) is also allowed.

BS/LS: **constraintOperator** = childOf / descendantOrSelfOf / descendantOf / parentOf / ancestorOrSelfOf / ancestorOf	
	A constraint operator is either 'childOf', 'descendantOrSelfOf', 'descendantOf', 'parentOf', 'ancestorOrSelfOf', or 'ancestorOf'.
BS: **descendantOf** = "<" **LS:** **descendantOf** = "<" / (("d"/"D") ("e"/"E") ("s"/"S") ("c"/"C") ("e"/"E") ("n"/"N") ("d"/"D")("a"/"A") ("n"/"N") ("t"/"T") ("o"/"O")("f"/"F") mws)	
	The descendantOf operator returns the set of all subtypes of the given concept (or set of concepts). In the brief syntax, the descendantOf operator is represented using the symbol "<". In the long syntax, the text "descendantOf" (case insensitive and followed by at least one white space) is also allowed.
BS: **descendantOrSelfOf** = "<<" **LS:** **descendantOrSelfOf** = "<<" / (("d"/"D") ("e"/"E") ("s"/"S") ("c"/"C") ("e"/"E") ("n"/"N") ("d"/"D") ("a"/"A") ("n"/"N") ("t"/"T") ("o"/"O")("r"/"R") ("s"/"S")("e"/"E") ("l"/"L") ("f"/"F") ("o"/"O")("f"/"F") mws)	
	The descendantOrSelfOf operator returns the set of all subtypes of the given concept (or set of concepts), plus the concept (or set of concepts) itself. In the brief syntax, the descendantOrSelfOf operator is represented using the symbols "<<". In the long syntax, the text "descendantOrSelfOf" (case insensitive and followed by at least one white space) is also allowed.
BS: **childOf** = "<!" **LS:** **childOf** = "<!" / (("c"/"C") ("h"/"H") ("i"/"I") ("l"/"L") ("d"/"D") ("o"/"O") ("f"/"F") mws)	
	The childOf operator returns the set of all immediate children of the given concept (or set of concepts). In the brief syntax, the childOf operator is represented using the symbols "<!". In the long syntax, the text "childOf" (case insensitive and followed by at least one white space) is also allowed.
BS: **ancestorOf** = ">" **LS:** **ancestorOf** = ">" / (("a"/"A") ("n"/"N") ("c"/"C") ("e"/"E") ("s"/"S") ("t"/"T") ("o"/"O") ("r"/"R") ("o"/"O")("f"/"F") mws)	
	The ancestorOf operator returns the set of all supertypes of the given concept (or set of concepts). In the brief syntax, the ancestorOf operator is represented using the symbol ">". In the long syntax, the text "ancestorOf " (case insensitive and followed by at least one white space) is also allowed.
BS: **ancestorOrSelfOf** = ">>" **LS:** **ancestorOrSelfOf** = ">>" / (("a"/"A") ("n"/"N") ("c"/"C") ("e"/"E") ("s"/"S") ("t"/"T") ("o"/"O") ("r"/"R") ("o"/"O") ("r"/"R") ("s"/"S") ("e"/"E") ("l"/"L" ("f"/"F") ("o"/"O")("f"/"F") mws)	
	The ancestorOrSelfOf operator returns the set of all supertypes of the given concept (or set of concepts), plus the concept (or set of concepts) itself. In the brief syntax, the ancestorOrSelfOf operator is represented using the symbols ">>". In the long syntax, the text "ancestorOrSelfOf" (case insensitive and followed by at least one white space) is also allowed.
BS: **parentOf** = ">!" **LS:** **parentOf** = ">!" / (("p"/"P") ("a"/"A") ("r"/"R") ("e"/"E") ("n"/"N") ("t"/"T") ("o"/"O") ("f"/"F") mws)	
	The parentOf operator returns the set of all immediate parents of the given concept (or set of concepts). In the brief syntax, the parentOf operator is represented using the symbols ">!". In the long syntax, the text "parentOf" (case insensitive and followed by at least one white space) is also allowed.
BS/LS: **conjunction** = (("a"/"A") ("n"/"N") ("d"/"D") mws) / ","	

	A conjunction is represented either by the word "and" (case insensitive and followed by at least one white space), or by a comma.
BS/LS: **disjunction** = ("o"/"O") ("r"/"R") mws	
	A disjunction is represented by the word "or" (case insensitive and followed by at least one white space).
BS/LS: **exclusion** = ("m"/"M") ("i"/"I") ("n"/"N") ("u"/"U") ("s"/"S") mws	
	The exclusion operator is represented by the word "minus" (case insensitive and followed by at least one white space).
BS/LS: **eclRefinement** = subRefinement ws [conjunctionRefinementSet / disjunctionRefinementSet]	
	A refinement contains all the grouped and ungrouped attributes that refine the set of clinical meanings satisfied by the expression constraint. Refinements may represent the conjunction or disjunction of two smaller refinements, and may optionally be placed in brackets. Where both conjunction and disjunction are used, brackets are mandatory to disambiguate the intended meaning.
BS/LS: **conjunctionRefinementSet** = 1*(ws conjunction ws subRefinement)	
	A conjunction refinement set consists of one or more conjunction operators, each followed by a subRefinement.
BS/LS: **disjunctionRefinementSet** = 1*(ws disjunction ws subRefinement)	
	A disjunction refinement set consists of one or more disjunction operators, each followed by a subRefinement.
BS/LS: **subRefinement** = eclAttributeSet / eclAttributeGroup / "(" ws eclRefinement ws ")"	
	A subRefinement is either an attribute set, an attribute group or a bracketed refinement.
BS/LS: **eclAttributeSet** = subAttributeSet ws [conjunctionAttributeSet / disjunctionAttributeSet]	
	An attribute set contains one or more attribute name-value pairs separated by a conjunction or disjunction operator. An attribute set may optionally be placed in brackets.
BS/LS: **conjunctionAttributeSet** = 1*(ws conjunction ws subAttributeSet)	
	A conjunction attribute set consists of one or more conjunction operators, each followed by a subAttributeSet.
BS/LS: **disjunctionAttributeSet** = 1*(ws disjunction ws subAttributeSet)	
	A disjunction attribute set consists of one or more disjunction operators, each followed by a subAttributeSet.
BS/LS: **subAttributeSet** = eclAttribute / "(" ws eclAttributeSet ws ")"	
	A subAttributeSet is either an attribute or a bracketed attribute set.
BS/LS: **eclAttributeGroup** = ["[" cardinality "]" ws] "{" ws eclAttributeSet ws "}"	
	An attribute group contains a collection of attributes that operate together as part of the refinement of the containing expression con straint. An attribute group may optionally be preceded by a cardinality. An attribute group cardinality indicates the minimum and maximum number of attribute groups that must satisfy the given attributeSet constraint for the expression constraint to be satisfied.
BS/LS: **eclAttribute** = ["[" cardinality "]" ws] [reverseFlag ws] eclAttributeName ws (expressionComparisonOperator ws subExpressionConstraint / numericComparisonOperator ws "#" numericValue / stringComparisonOperator ws QM stringValue QM)	

	An attribute is a name-value pair expressing a single refinement of the containing expression constraint. Either the attribute value must satisfy (or not) the given expression constraint, the attribute value is compared with a given numeric value (integer or decimal) using a numeric comparison operator, or the attribute value must be equal to (or not equal to) the given string value. The attribute may optionally be preceded by a cardinality constraint and/or a reverse flag.
BS/LS: **cardinality** = minValue to maxValue	
	The cardinality represents a constraint on the minimum and maximum number of times that the given attribute or attribute group may appear in a matching expression. The cardinality is enclosed in square brackets with the minimum cardinality appearing first, followed by a separator (two dots in the brief syntax), and then the maximum cardinality.
BS/LS: **minValue** = nonNegativeIntegerValue	
	A value that represents the minimum number of times that an attribute or attribute group may appear. The minimum cardinality must always be less than or equal to the maximum cardinality.
BS: **to** = ".." **LS:** **to** = ".." / (mws ("t"/"T") ("o"/"O") mws)	
	In the brief syntax, the minimum and maximum cardinality are separated by two dots (i.e. ".."). In the long syntax, the text "to" (case insensitive with at least one white space before and after) is also allowed between the two cardinalities.
BS/LS: **maxValue** = nonNegativeIntegerValue / many	
	A value that represents the maximum number of times that an attribute or attribute group may appear. A maximum cardinality of 'many' indicates that there is no limit on the number of times the attribute may appear.
BS: **many** = "*" **LS:** **many** = "*" / (("m"/"M") ("a"/"A") ("n"/"N") ("y"/"Y"))	
	In the brief syntax, a cardinality of 'many' is represented using the symbol "*". In the long syntax, the text "many" (case insensitive, with no trailing space) is also allowed.
BS: **reverseFlag** = "R" **LS:** **reverseFlag** = (("r"/"R") ("e"/"E") ("v"/"V") ("e"/"E") ("r"/"R") ("s"/"S") ("e"/"E") ("o"/"O") ("f"/"F")) / "R"	
	When a reverse flag is used on an attribute, the matching relationships are traversed in the reverse of the normal direction. This means that the target concept of each relationship must match the focus concept to which the attribute is applied, while the source concept of the relationship must match the attribute value. In the brief syntax, the reverse flag is represented using the character "R" (in uppercase). In the long syntax, the text "reverseOf " (case insensitive) is also allowed.
BS/LS: **eclAttributeName** = subExpressionConstraint	
	The attribute name is the name of an attribute (or relationship type) to which a value is applied to refine the meaning of a containing expression constraint. The attribute name is represented using a subExpressionConstraint, as defined above.
BS: **expressionComparisonOperator** = "=" / "!=" **LS:** **expressionComparisonOperator** = "=" / "!=" / ("n"/"N") ("o"/"O") ("t"/"T") ws "=" / "<>"	

	Attributes whose value is a concept may be compared to an expression constraint using either equals ("=") or not equals ("!="). In the long syntax "<>" and "not =" (case insensitive) are also valid ways to represent not equals.
BS: **numericComparisonOperator** = "=" / "!=" / "<=" / "<" / ">=" / ">" **LS:** **numericComparisonOperator** = "=" / "!=" / ("n"/"N") ("o"/"O") ("t"/"T") ws "=" / "<>" / "<=" / "<" / ">=" / ">"	
	Attributes whose value is numeric (i.e. integer or decimal) may be compared to a specific concrete value using a variety of comparison operators, including equals ("="), less than ("<"), less than or equals ("<="), greater than (">"), greater than or equals (">=") and not equals ("!="). In the long syntax "<>" and "not =" (case insensitive) are also valid ways to represent not equals.
BS: **stringComparisonOperator** = "=" / "!=" **LS:** **stringComparisonOperator** = "=" / "!=" / ("n"/"N") ("o"/"O") ("t"/"T") ws "=" / "<>"	
	Attributes whose value is numeric may be compared to an expression constraint using either equals ("=") or not equals ("!="). In the long syntax "<>" and "not =" (case insensitive) are also valid ways to represent not equals.
BS/LS: **numericValue** = ["-"/"+"] (decimalValue / integerValue)	
	A numeric value is either an integer or a decimal. Positive numbers optionally start with a plus sign ("+"), while negative integers begin with a minus sign ("-").
BS/LS: **stringValue** = 1*(anyNonEscapedChar / escapedChar)	
	A string value includes one or more of any printable ASCII characters enclosed in quotation marks. Quotes and backslash characters within the string must be preceded by the escape character ("\").
BS/LS: **integerValue** = digitNonZero *digit / zero	
	An integer value is either starts with a non-zero digit followed by zero to many additional digits, or is the integer zero itself.
BS/LS: **decimalValue** = integerValue "." 1*digit	
	A decimal value starts with an integer. This is followed by a decimal point and one to many digits.
BS/LS: **nonNegativeIntegerValue** = (digitNonZero *digit) / zero	
	A non-negative integer value (i.e. positive integers or zero), without a preceding plus sign ("+").
BS/LS: **sctId** = digitNonZero 5*17(digit)	
	A SNOMED CT id is used to represent an attribute id or a concept id. The initial digit may not be zero. The smallest number of digits is six, and the maximum is 18.
BS/LS: **ws** = *(SP / HTAB / CR / LF / comment)	

	Optional whitespace characters (space, tab, carriage return, linefeed or a comment) are ignored everywhere in the expression except: 1. Whitespace within a conceptId is an error. **Note:** Whitespace before or after the last digit of a valid Identifier is ignored. 2. Non-consecutive spaces within a term are treated as a significant character of the term. **Note:** Whitespace before the first or after the last non-whitespace character of a term is ignored 3. Whitespace within the quotation marks of a concrete value is treated as a significant character.
BS/LS: **mws** = 1*(SP / HTAB / CR / LF / comment)	
	Mandatory whitespace (i.e. space, tab, carriage return, linefeed or a comment) is required after certain keywords, including "And" and "Or".
BS/LS: **comment** = "/*" *(nonStarChar / starWithNonLSlash) "*/"	
	A comment, which provides additional human-readable details about the expression constraint. Comments begin with a forward slash directly followed by a star (i.e. "/*") and end with a star directly followed by a forward slash (i.e. "*/").
BS/LS: **nonStarChar** = SP / HTAB / CR / LF / %x21-29 / %x2B-7E /UTF8-2 / UTF8-3 / UTF8-4	
	A character that is not a star (i.e. not %x2A).
BS/LS: **starWithNonLSlash** = %x2A nonLSlash	
	A star (i.e. "*") followed by a character that is not a forward slash (i.e. not "/").
BS/LS: **nonLSlash** = SP / HTAB / CR / LF / %x21-2E / %x30-7E /UTF8-2 / UTF8-3 / UTF8-4	
	A character that is not a forward slash (i.e. not "/").
BS/LS: **SP** = %x20	
	Space character.
BS/LS: **HTAB** = %x09	
	Tab character.
BS/LS: **CR** = %x0D	
	Carriage return character.
BS/LS: **LF** = %x0A	
	Line feed character.
BS/LS: **QM** = %x22	
	Quotation mark character.
BS/LS: **digit** = %x30-39	
	Any digit 0 through 9.
BS/LS: **zero** = %x30	
	The digit 0.
BS/LS: **digitNonZero** = %x31-39	

	Digits 1 through 9, but excluding 0. The first character of a concept identifier is constrained to a digit other than zero.
BS/LS: **nonwsnonpipe**= %x21-7B / %x7D-7E / UTF8-2 / UTF8-3 / UTF8-4	
	Non whitespace (and non pipe) includes printable ASCII characters (these are also valid UTF8 characters encoded as one octet) and also includes all UTF8 characters encoded as 2- 3- or 4-octet sequences. It excludes space (which is %x20) and the pipe character "
BS/LS: **anyNonEscapedChar** = SP / HTAB / CR / LF / %x20-21 / %x23-5B / %x5D-7E / UTF8-2 / UTF8-3 / UTF8-4	
	anyNonEscapedChar includes any printable ASCII characters which do not need to be preceded by an escape character (i.e. "\"). This includes valid UTF8 characters encoded as one octet and all UTF8 characters encoded as 2, 3 or 4 octet sequences. It does, however, exclude the quotation mark (") and the backslash (). See RFC 3629 (UTF-8, a transformation format of ISO 10646 authored by the Network Working Group).
BS/LS: **escapedChar** = BS QM / BS BS	
	The double quotation mark and the back slash character must both be escaped within a string-based concrete value by preceding them with a back slash.
BS/LS: **UTF8-2** = %xC2-DF UTF8-tail	
	UTF8 characters encoded as 2-octet sequences.
BS/LS: **UTF8-3** = %xE0 %xA0-BF UTF8-tail / %xE1-EC 2(UTF8-tail) / %xED %x80-9F UTF8-tail / %xEE-EF 2(UTF8-tail)	
	UTF8 characters encoded as 3-octet sequences.
BS/LS: **UTF8-4** = %xF0 %x90-BF 2(UTF8-tail) / %xF1-F3 3(UTF8-tail) / %xF4 %x80-8F 2(UTF8-tail)	
	UTF8 characters encoded as 4-octet sequences.
BS/LS: **UTF8-tail** = %x80-BF	
	UTF8 characters encoded as 8-octet sequences.